UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

MÓDULO DE DIAGNÓSTICO EN ROBOTS DE SERVICIO

T E S I S
QUE PARA OPTAR POR EL GRADO DE:
MAESTRO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

P R E S E N T A:
I N G. E D U A R D O T E L L O R A M O S

Director de Tesis:
DR. LUIS ALBERTO PINEDA CORTÉS
Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas, UNAM

Ciudad Universitaria, CD. MX **Junio, 2017**

Índice

Índice de figuras y tablas

Dedicatoria

A mi madre por apoyarme
A mi hermana por inspirarme
A mis profesores por enseñarme

Agradecimientos

Me gustaría agradecer a todos aquellos que hicieron posible que yo realizara este trabajo.

Agradezco en primer lugar a mi director de tesis, el Dr. Luis Alberto Pineda Cortés por haberme presentado la oportunidad de trabajar en este proyecto de tesis que encontré muy interesante y profundamente informativo.

También agradezco al Dr. Thomas Eiter por haberme recibido en su grupo en Viena para que pudiera yo obtener la soltura necesaria con el paradigma de *Answer Set Programming* para poder realizar este trabajo.

Me gustaría agradecer a mis sinodales, Dr, Jesús Savage Carmona, Dr. Ivan Vladimir Meza Ruiz y Dr. Gibran Fuentes Pineda por sus comentarios y observaciones sobre mi trabajo de tesis.

Finalmente me gustaría agradecer al CONACYT por la beca otorgada que me permitió realizar mis estudios así como la beca mixta que me permitió trabajar con el Dr. Thomas Eiter en su grupo de trabajo en la Technische Universitat Wien.

Resumen:

En este trabajo se analiza si es posible resolver un problema de inferencia en el robot de servicio Golem utilizando *Answer Set Programing*. El problema a resolver se da en el contexto de la prueba del supermercado diseñada por grupo Golem. Este problema ha sido anteriormente resuelto utilizando árboles de búsqueda con heurísticas (POP).[23] El punto de resolverlo usando *Answer Set Programming* es que de ser posible se contaría con una equivalencia en el manejo de la base de datos de Golem así como una equivalencia de los resultados obtenidos del módulo de inferencia desarrollado por el grupo. Una vez que se muestra que es posible resolver el problema de ambas formas sin que esto altere el desempeño o resultados de la prueba se puede afirmar que los algoritmos propuestos son equivalentes por lo que se puede esperar que *Answer Set Programming* sea un paradigma válido para desarrollar otras aplicaciones para el robot de servicio Golem.

En el trabajo se hace una revisión de la bibliografía disponible relativa al tema de diagnóstico, *Answer Set Programming* y las tecnologías relacionadas. Se muestra el paradigma que se decidió usar para resolver el problema y se explica cómo se implementa haciendo un análisis desde las partes más básicas hasta el sistema completo. Se describe la prueba a ser resuelta y se describe el funcionamiento del submódulo de diagnóstico del módulo de inferencia con suficiente detalle para mostrar lo que está haciendo el código.

El trabajo también muestra los resultados de la ejecución de la prueba junto con los resultados de la ejecución de la misma prueba utilizando el módulo de inferencia diseñado por grupo Golem. Se analizan los resultados y finalmente se presenta que en conclusión, es posible utilizar *Answer Set Programming* para resolver el problema de diagnóstico presentado, por lo que es razonable pensar que este paradigma puede ser utilizado para resolver otros problemas similares en robots de servicio. Se recomienda cuidar que cuando se decida utilizar *Answer Set Programming* se tenga en cuenta el tamaño del universo de la prueba pues, para universos grandes, las técnicas presentadas en este trabajo pueden crecer mucho en tamaño y tiempo de ejecución.

Capítulo 1: Introducción

Los agentes racionales son aquellos que tienen metas claras y una preferencia sobre cómo alcanzarlas; esta preferencia se basa en la optimización de algún aspecto de la realización de la tarea. Durante el ejercicio de sus labores, estos agentes están inscritos en el ciclo inferencial de la vida cotidiana el cual involucra inferencias conceptual y deliberativa.

La inferencia conceptual se utiliza durante el ciclo de comunicación para asociar conceptos. Por ejemplo, saber que Juan es hijo de María y que Raúl también lo es, permite saber que ambos son hermanos aunque no se sepa esta última relación explícitamente.

La inferencia deliberativa se utiliza cuando es necesario hacer una reflexión para generar una nueva idea mediante la creación y exploración de un espacio de búsqueda. Por ejemplo, calcular las jugadas en un partido de ajedrez y extrapolar las posibles respuestas del contrincante permite generar una estrategia para ganar la contienda. Este tipo de pensamiento es útil cuando algún aspecto del mundo es diferente a lo que se esperaba y por consiguiente es necesario reevaluar el curso de acción y modificarlo para que las metas del agente se cumplan pese a la discrepancia detectada.

Una forma del ciclo de inferencia deliberativa se puede dividir en tres partes. El diagnóstico permite saber qué es lo que ocurrió para que el mundo se encuentre en el estado actual. La toma de decisión consta en saber qué hacer tomando en cuenta los intereses y el diagnóstico. La planeación es la generación de una secuencia de acciones que permitan llevar a cabo la decisión.

El problema del diagnóstico se puede expresar de la siguiente forma: Se emplea razonamiento abductivo (de efectos a causas) para encontrar una razón de ser plausible a cierto fenómeno observado. Por ejemplo, cuando un médico postula un enfermedad a manera de hipótesis que explique un conjunto de síntomas o un niño formule posibles justificaciones para el hecho de que el pasto está mojado.

El propósito de los robots de servicio es atender a las personas en el ambiente que se desenvuelven sin tener que adecuarlo para el robot. Como los humanos no siempre se comportan de maneras predecibles, los robots necesitan contar con herramientas necesarias para navegar por el mundo cuando ocurren cambios imprevistos.

Para poder desenvolverse en el mundo y ejecutar las tareas que se le solicitan, un robot necesita contar con la representación del mundo en la forma de una base de conocimientos. Se pueden crear algoritmos que permitan manejar este repositorio de manera similar a la que los agentes racionales usan durante el ciclo de inferencia conceptual para generar conocimiento nuevo a partir del conocimiento ya existente en memoria.

El grupo Golem del IIMAS de la UNAM busca explorar aspectos diversos de la cognición a través de la metáfora provista por el robot de servicio. Los problemas a los que se enfrentan los agentes racionales pueden ser modelados al hacer uso de la lógica de manera que sea posible resolverlos

mediante algoritmos computacionales. El estudio de las soluciones propuestas permite entrever características del pensamiento en un ambiente controlado.

Para realizar sus estudios en robots de servicio el grupo Golem ha desarrollado el lenguaje SitLog que permite enmarcar la actividad de una de éstas máquinas en un ambiente de tareas descritas por modelos de diálogo que se navegan para ir cambiando el estado del robot así como sus acciones y expectativas. La idea detrás de este paradigma es que el robot se mantiene situado en la tarea al siempre contar con información sobre el mundo y posibles reacciones a ésta. Cuando las observaciones realizadas difieren de la representación del mundo que se tiene en la base de conocimientos se considera que el robot "ha perdido la situación" y es necesario realizar ajustes para que éste pueda continuar trabajando. Se pueden programar algoritmos que le permitan a la máquina responder a este problema de manera similar a la que los agentes racionales usan durante el ciclo de inferencia deliberativa para generar hipótesis plausibles que la ayuden a llevar a término la tarea que le haya sido encomendada.

Se considera un modelo estable a un conjunto de reglas lógicas no negadas que contiene todos los átomos aterrizados y que no son contradictorias entre sí. *Answer Set Programming* (*ASP*) es un paradigma de programación declarativa que se puede usar para resolver problemas de búsqueda a través de estos modelos. *ASP* permite la representación del conocimiento de una forma más natural que los lenguajes procedurales y permite el uso de lógica no monotónica para reducir el conjunto de respuestas provistas dado conocimiento nuevo. El uso de *ASP* en la robótica de servicio es deseable dado que la incorporación de conocimiento es más fácil que en otros paradigmas de programación.

Este trabajo se centra en averiguar si es posible y deseable resolver el problema del diagnóstico como parte de un módulo de inferencia deliberativa mediante la utilización de *ASP* en el contexto de robots de servicio.

Para probar la solución propuesta en este trabajo se utiliza el robot Golem-III y se resuelve la prueba del supermercado que consiste en llevar objetos al cliente que los solicita desde distintas repisas aún si éstos se encuentran en un lugar distinto al originalmente supuesto. El módulo de inferencia empleado se programó junto con el Ing. Ricardo Adolfo Fierro Villaneda quien se encargó de la toma de decisión y planeación. El grupo Golem ya cuenta con un módulo de inferencia deliberativa que usa para resolver esta prueba el cual emplea búsqueda heurística en un espacio del problema; por lo que otro objetivo es poder sustituirlo con el módulo desarrollado en *ASP* para así establecer una equivalencia de los algoritmos al emplear las mismas entradas y regresar los mismos resultados.

Para la parte de diagnóstico se propone emplear la metodología "*Guess and Check*" de *ASP* para generar un espacio de búsqueda conformado por las posibles acciones que el asistente pudo haber tomado para colocar los objetos en las repisas donde se supone están y hacer así una o más cadenas de acciones a manera de hipótesis. Las secuencias válidas explican las observaciones del ambiente (tanto las que coinciden con la información contenida en la base de conocimientos como las que la contradicen).

Este trabajo también explora un cambio de enfoque al diagnóstico empleado para la prueba del restaurante que reduce el espacio de búsqueda sin cambiar los resultados por lo que es más eficiente a una implementación directa del paradigma originalmente propuesto.

Dado que es posible que la robótica de servicio tenga un impacto importante tanto en lo social como en lo económico, es recomendable contar con algoritmos que le permitan a estas máquinas continuar con sus labores aunque el ambiente cambie de formas inesperadas. Por otro lado el análisis de las soluciones propuestas es de interés en el estúdio de la cognición en otro tipo de agentes racionales.

El presente documento se divide de la siguiente manera: **Capítulo 2: Trabajo Anterior**, donde se presenta parte de la bibliografía que trata con el tema de Golem, la prueba del supermercado y *Answer Set Programming*. El **Capítulo 3: Metodología**, describe la prueba del supermercado, el uso de conjuntos solución (Answer Sets) para la resolución del problema, y los sistemas de inferencia; el de grupo Golem y el generado para este trabajo. El **Capítulo 4: Algoritmos**, explica el funcionamiento de los algoritmos que permiten resolver el problema de diagnóstico, el capítulo está dividido según los archivos (programas) que se emplean. **Capítulo 5: Experimentos, resultados y análisis**, donde se explica la ejecución de la prueba, se presentan los resultados de las ejecuciones del algoritmo utilizando el robot, y se hace un análisis de los resultados obtenidos. **Capítulo 6: Conclusiones**, donde se explica el significado de los resultados y se hace una recomendación sobre la utilización de *Answer Set Programming*. En los **Apéndices**, se presenta el código utilizado (tanto en Prolog como en *ASP*) así como una breve explicación de sintaxis de los predicados de *Answer Set Programming* empleados en la solución del problema. En éstos también se encuentra la implementación de la metodología "*Guess and check*" en un acercamiento al problema que emplea fluents así como una comparación de las bases de conocimiento resultantes de la ejecución del algoritmo.

Capítulo 2: Trabajo anterior

2.1 Razonamiento

De acuerdo a Mercier y Sperber (2011) el razonamiento se da a través de procesos llamados "inferencia intuitiva" que son mayormente inconscientes y producen una representación mental a partir de una anterior; esta inferencia se realiza sobre objetos, eventos o representaciones y sirve para generar nuevas creencias a partir de creencias previas, expectativas a partir de observaciones o planes a partir de creencias y preferencias.[36]

El proceso de razonamiento vía la inferencia intuitiva se basa en representaciones iniciales (premisas) y representaciones finales (conclusiones). La inferencia que lo caracteriza se alcanza de forma necesaria o probabilística y su función es aumentar o corregir la información disponible a un proceso cognitivo.

Pese a que el razonamiento humano es frecuentemente fallido, éste le permite ir más allá de la percepción, el hábito y el instinto. Este proceso es reconocido por la persona pero la naturaleza del mismo no es evidente en el momento en el que se ejecuta.[36]

Por otro lado, para Brachman y Levesque (2003) el razonamiento es la manipulación formal de símbolos que representan creencias para hacer explícito lo que se sabe de manera. Así, éste es una forma de cálculo en el que los elementos manipulados son proposiciones. La representación de un objeto o concepto cuenta con partes bien definidas y una sintaxis clara. Este acercamiento es válido en un sistema basado en el conocimiento que siga la hipótesis de la representación del conocimiento de Brian Smith.[19]

El razonamiento es el proceso a través del cual la representación de un concepto asociado a un objeto físico o un evento se manipula en aras de generar la representación de un nuevo concepto. La manipulación de estos símbolos se da a través de inferencias que son alcanzadas de distintas maneras pero dado que el interés de este trabajo es el estudio de la metáfora de la cognición que provee un agente racional, tal como el robot de servicio, es que se busca un sistema que permita la manipulación de estos símbolos de una manera que se procese en algoritmos computacionales; esto se logra a través del uso de la lógica.

2.2 Razonamiento por Lógica

Para Brachman y Levesque la ventaja de emplear la lógica en problemas de representación de conocimiento y razonamiento es que ésta cuenta con herramientas que permiten manipular símbolos para generar consecuencias de manera análoga a los procesos de la razón. La lógica también está bien provista para analizar problemas en el nivel del conocimiento en sistemas basados en conocimientos (según la teoría de Allan Newell).[19]

La lógica a usarse es la lógica de predicados o lógica de primer orden (FOL por sus siglas en inglés) ya que de forma sencilla permite expresar ideas y operar sobre estas. Una explicación a profundidad de la sintaxis y semántica de la lógica de primer orden se encuentra en el capítulo 8 de Russell y Norvig (1995), en el capítulo 2 de Brachman y Levesque (2003), o en el capítulo 5 de Allwood y Andersson, (1977).

El razonamiento basado en lógica emplea inferencias para generar las consecuencias que permiten ir de una idea a otra, según el tipo de inferencia que emplea se divide en tres categorías: deducción, inducción y abducción.

El razonamiento deductivo genera una conclusión dada la aplicación de una regla a una premisa; por ejemplo como el esquema de inferencia del modus ponens.

$$\begin{array}{l} P \rightarrow Q \\ \underline{P \quad .} \\ Q \end{array}$$

Ejemplo: si P = *llueve* y Q = *el pasto está mojado.*

El razonamiento inductivo genera una regla que generaliza o abstrae cierta experiencia.

Si se observa que $P(a) \rightarrow Q$, $P(b) \rightarrow Q, \ldots$
Se induce que $\forall x\, P(x) \rightarrow Q$

Ejemplo: Si se ha visto repetidamente que después de que ha llovido el pasto se encuentra mojado, se induce la regla de que la lluvia moja el pasto.

Sin embargo esta inferencia no es válida en el mismo sentido que la deductiva y su estudio se contempla en el aprendizaje.

Finalmente, El razonamiento abductivo considera las premisas que satisfacen una conclusión y busca presentar aquellas que sean más probables de todas aquellas que son posibles.

$$\begin{array}{l} P_1 \rightarrow Q \\ P_2 \rightarrow Q \\ \ldots \\ \underline{P_n \rightarrow Q} \\ Q \end{array}$$

Quien de P_1, P_2, P_3… es el más plausible

Ejemplo: Si se sabe que la lluvia moja el pasto y que los rociadores mojan el pasto y se observa que el pasto está mojado se abduce que ayer llovió o que ayer se activaron los rociadores. Es importante notar que ésta forma de razonamiento no es válida ya que puede haber excepciones ya que un conocimiento que se adquiere en el futuro puede cambiar el resultado.

2.3 Razonamiento abductivo

Para Brachman y Levesque el razonamiento abductivo se engloba en la pregunta de qué sería necesario para que X fuera cierto. Con X una proposición lógica o dicho de otra manera, " aparte de lo que sé, ¿qué más necesitaría ser cierto para que X sea cierto?". [19]

La forma en que la abducción fue formalizada por Charles Pierce es la siguiente:

> "Una Abducción es un método para formar una predicción general sin ninguna verdadera seguridad de que tendrá éxito, sea en un caso especial o con carácter general, teniendo como justificación que es la única esperanza posible de regular nuestra conducta futura racionalmente, y que la Inducción, partiendo de experiencias pasadas, nos alienta fuertemente a esperar que tendrá éxito en el futuro. ".[31]

Eshghi y Kowalski (1989) explican en su artículo que si se realiza el razonamiento abductivo a través del empleo de la negación por falla la semántica corresponde y se generaliza a la semántica de modelos estables (esta es la que se emplea en *Answer Set Programming*).[13]

"La abducción es una forma de razonamiento no monotónico puesto que las hipótesis que son consistentes con un estado del conocimiento se pueden volver inconsistentes cuando se añade nuevo conocimiento".[13]

2.4 Diagnóstico

Para Brachman y Levesque el diagnóstico es particularmente útil cuando se razona sobre causas y efectos; por ejemplo, cuando se tiene una serie de hechos de la forma (Enfermedad ^ … ⊃ Síntomas) (La elipsis representa cualificadores) en una base de conocimientos y se busca encontrar una enfermedad (o enfermedades) que expliquen los síntomas que han sido observados en un paciente. Es así que el diagnóstico generado no es una consecuencia lógica sino una conjetura.[19]

Peischl y Wotawa explican que el diagnóstico basado en modelos parte de "anomalías" en el mismo de forma que las causas que mejor explican el comportamiento anómalo se infieren de un objeto observado.[21]

Es importante mencionar desde ahora que el acercamiento empleado en este trabajo es la generación dinámica de diagnósticos que expliquen las observaciones del robot y que permitan generar nuevas creencias sobre el ambiente en el que se encuentra.

2.5 Cálculo de eventos

Dado que los diagnósticos generados al emplear el módulo de inferencia en *ASP* de Golem son secuencias de acciones, y que éstos se generan dinámicamente a partir de un mundo observado a un mundo inicial; para este trabajo es importante contar con un formalismo que facilite la creación de dichas secuencias. El formalismo a emplear es denominado "Event Calculus" y éste nos permite analizar las secuencias de acciones mediante el uso de conceptos tales como Acciones (eventos), Fluents y Tiempos (aunque no se emplea el conjunto completo de axiomas de EC). A continuación se mencionan algunos de los trabajos relacionados con este paradigma.

Shanahan explica la importancia de contar con un formalismo para describir acciones y sus efectos y explica su relación con los tres tipos de razonamiento (deductivo, inductivo y abductivo) así como la relación con el problema del marco (McCarthy & Hayes, 1969). En un artículo posterior, el autor discute el empleo de EC para modelar un planeador a base de razonamiento abductivo.[33] [34]

Esghi (1988) usa Event Calculus para representar el tiempo y el cambio en problemas de planeación y la abducción como el mecanismo para solucionarlos. El autor afirma que este enfoque evita el problema del marco.[12]

2.6 Abducción en robótica

Tanto el razonamiento abductivo como el diagnóstico se han empleado en el campo de la robótica anteriormente; a continuación se muestran algunos ejemplos:

Miroslav Janícek (2012) emplea el razonamiento abductivo en un marco abductivo contextual continuo para comprender el diálogo natural situado a través de un conjunto de explicaciones anulables que buscan inferir la intención del locutor.[18]

Murray Shanahan usa razonamiento abductivo para que un robot construya modelos del ambiente en el que se encuentra a partir de la información incompleta e incierta que proviene de los sentidos.[35] [32]

2.7 Diagnóstico en robótica

Pineda, et al., del grupo Golem emplean diagnóstico dentro de un módulo de inferencia basado en Prolog para diversas tareas; por ejemplo para inferir las posibles acciones de un supuesto asistente humano al acomodar insumos en un escenario de supermercado. El módulo se llama cuando se busca explicar la posición de los objetos en el escenario.[23]

Generalmente el diagnóstico se emplea para detectar y reparar fallas en sistemas o para tratar malestares según los síntomas que exhibe algún agente; a continuación se citan algunos ejemplos.

Evans, et al., emplean un microscopio robótico para generar y mandar muestras de tejidos a sitios remotos para su diagnóstico.[15]

Huan, et al., emplean diagnóstico basado en modelos para manejar las ejecuciones fallidas para un robot que debe conectar cables a diferentes conectores durante el ensamblaje de componentes electrónicos.[17]

Bongard y Lipson emplean el diagnóstico generado evolutivamente para manejar fallas en robots que operan en lugares remotos o inaccesibles tales como sondas espaciales o robots que se encuentren en ambientes peligrosos para lo seres humanos.[4]

2.8 ASP en robótica

A continuación se presentan ejemplos del trabajo que se ha realizado en el área de robótica que emplea *Answer Set Programming*.

Erdem, Aker y Patoglu presentan una aplicación que emplea *ASP* para controlar robots de aseo del hogar a través de la incorporación de sentido común y razonamiento espacial y temporal como unidades discretas de razonamiento. Esta aplicación también cuenta con un algoritmo que genera y monitorea la ejecución de planes para que las tareas se desarrollen de forma segura y que los robots se puedan recuperar de fallas.[11]

Eppe y Bhatt emplean *ASP* en una silla de ruedas robótica para proveer al robot de una explicación para las anormalidades que experimenta durante el ejercicio de sus tareas así como de un nuevo plan que le permita continuar con su labor.[10]

En su libro, Gelfond y Kahl explican la forma de diseñar agentes basados en el conocimiento y cómo implementarlos en *ASP*, cómo diseñar una base de conocimientos que haga uso de defaults así como el diseño de agentes de planeación y diagnóstico. Estos sistemas se plantean tanto para agentes computacionales como para robots (un ejemplo es el del mundo de bloques).[16]

Capítulo 3: Metodología

3.1 Descripción del problema

Para evaluar la capacidad de Golem para trabajar en situaciones de incertidumbre, el grupo Golem diseñó la prueba del supermercado en la que se sitúa al robot como encargado de una tienda en la que hay productos y éste es el responsable de entregarlos al cliente. El aspecto de inferencia interviene cuando el objeto requerido por el cliente no se encuentra en la repisa designada por lo que en ese momento se llama al módulo de inferencia el cual consta de tres partes. La primer parte, el diagnóstico, se ocupa de generar una explicación a la discrepancia detectada. La explicación se presenta como una lista de acciones que transforma un mundo inicial donde todos los objetos se encuentran en el almacén a un mundo donde los objetos se encuentran en la posición del mundo actualmente observado. La segunda parte, la decisión, genera una lista de objetivos generales que le permitirán al robot intentar llevar a cabo su tarea al adaptar su comportamiento a las nuevas circunstancias y mantener organizado el supermercado. Finalmente la tercer parte del módulo de inferencia construye un plan para transformar el estado actual al deseado.

Dado que el sistema se llama continuamente, este tiene la capacidad de dar respuestas tales como diagnósticos vacíos (en caso de que los objetos continuen en la bodega) o diagnósticos que no son antecedidos por un error sino que son tan sólo la explicación de cómo los objetos llegaron a sus posiciones designadas, decisiones vacías, si es que no hay órdenes por satisfacer y planes vacíos en caso de que el planeador reciba una decisión vacía.

Esta prueba ha sido codificada como una tarea en SitLog; por lo tanto se emplean modelos de diálogo para regir las acciones que el robot ejecuta. Golem comienza la prueba a la espera de la mención de los objetos que han sido depositados por el asistente, y luego pregunta si el cliente desea algo. El robot tiene la capacidad de responder preguntas sobre el mundo como le fue descrito mediante el uso de su base de conocimiento y la utiliza para almacenar peticiones de artículos (tanto de artículos que son entregables como de artículos que no lo son por no tener ejemplares de los mismos). Una vez que el cliente ha pedido su artículo el agente se mueve para buscarlo y en cuanto lo encuentra regresa a entregar el pedido.

El error a reparar por el módulo de inferencia se presenta en caso de que Golem encuentre objetos fuera de su lugar. Es importante mencionar que dado el sistema de puntos que se usa en el sub módulo de decisión las prioridades del robot son entregar en cuanto sea posible, acomodar un artículo si queda de paso mientras busca un artículo requerido que no está en su lugar y finalmente buscar un artículo requerido sin ordenar otro artículo si este no queda de paso o no hay artículo que ordenar.

Las tareas utilizadas en Golem son representadas por listas de modelos de diálogo que se encadenan mediante arcos que se activan cuando una expectativa se cumple; de esta forma podemos pensar una tarea como una serie de modelos de diálogo en los que al entrar se hace algo y se espera una respuesta y según sea ésta se elige el siguiente modelo de diálogo que permitirá continuar con la tarea.

El sistema generado en *ASP* para resolver el problema del restaurante está estructurado de la siguiente forma: Consta de 3 partes principales al igual que el módulo de inferencia diseñado por grupo Golem, Diagnóstico, Decisión y Planeación. A grandes rasgos, en caso de que algún objeto haya sido observado en una repisa en la que no debería estar el módulo de Diagnóstico genera la narrativa que explica cómo llegó al lugar observado y hace un cambio en la base de conocimiento para reflejar la nueva posición del objeto. Si un objeto no es observado en una repisa en la que sí debería estar, el subsistema de diagnóstico infiere la posible localización del objeto de entre las repisas que aún no han sido observadas con miras a que el número de objetos por repisa sea lo más parecido posible al número de objetos por repisa que hubiera sido correcto de haberse colocado los objetos en el lugar correcto. El sistema de Decisión consulta la base de conocimiento modificada por el submódulo de diagnosis y busca tanto órdenes aún no entregadas como objetos fuera de su lugar; si hay órdenes por entregar buscará hacerlo; si hay objetos por acomodar buscará acomodarlo solamente si el lugar donde se supone deben estar es el lugar al que tiene que ir para seguir buscando el objeto que quiere entregar. Finalmente el submódulo de Planeación genera el espacio de posibles planes mediante un conjunto de reglas y restricciones que le permiten alcanzar la cadena de eventos óptima para llevar a cabo la decisión.

La parte que describe este documento es el submódulo encargado de generar soluciones al problema de diagnóstico mediante *Answer Set Programing* (para una discusión más específica sobre los algoritmos que conforman el sub módulo o el código propiamente se pueden consultar en el capítulo 4 o los anexos respectivamente).

El submódulo utiliza 2 tipos de programas, aquellos que se ejecutan en Prolog y aquellos que se ejecutan en DLV (es decir en *Answer Set Programming*). Los algoritmos que se ejecutan desde Prolog son de manipulación y transporte de datos; se traduce la base de conocimientos a un formato compatible con DLV, se guardan resultados en archivos, etc..; estos archivos no generan la inferencia ni manejan la base de conocimientos propiamente. La discusión de las funciones de estos archivos se puede encontrar en el capítulo 4 y anexos).

Lo que ahora nos interesa explorar es la metodología de resolución del problema de inferencia y diagnóstico mediante *Answer Set Programming*.

La premisa central del problema de diagnóstico es que se cuenta con una serie de objetos en un almacén y hay que explicar cómo llegaron a los lugares donde se cree (o se sabe) que están. El algoritmo se basa en que el asistente humano puede mover a lo más 2 objetos a la vez (uno con cada mano) y que es deseable que el número de objetos por repisa se parezca lo más posible al número de objetos por repisa resultante de la declaración inicial del asistente.

Primero es necesario explicar el acercamiento que se utilizó para generar los resultados en *Answer Set Programming*. Aunque hay muchas formas de proceder (Doble negación y Saturación por ejemplo), la que se eligió es la que se denomina *"Guess and check methodology"*.[9]

La aproximación de "*Guess and check*" se basa en la siguiente idea: utilizar el no determinismo de las cabezas disyuntivas para generar un acervo de posibles soluciones y utilizar restricciones para eliminar aquellas soluciones que no sean satisfactorias de manera que sólo se obtengan las soluciones deseadas. Este modelo también acepta predicados auxiliares (que permiten la generación de las soluciones sin ser necesariamente parte de la solución).

Ejemplo:

```
p(a,X) v p(b,X) v p(c,X):- t(X).      } Guess
 :- p(b,_).                           } Check
t(1).                                 } Predicado auxiliar
```

La parte "*Guess*" con información del predicado auxiliar genera las siguientes soluciones posibles (notar que sin un filtro, el predicado auxiliar forma parte del conjunto solución)

```
{t(1), p(c,1)},{t(1), p(b,1)},{t(1), p(a,1)}
```

La parte "*Check*" indica que el predicado `p(b,_)` ("_" significa cualquier valor) es falso por lo que la solución entera es inconsistente y por ende se desecha.

Así quedamos con las soluciones

```
{t(1), p(c,1)},{t(1), p(a,1)}
```

Es así que mediante esta metodología se ejecutan los algoritmos de *Answer Set Programming* en DLV para resolver el problema de diagnóstico del que trata este trabajo. Es importante hacer notar que en adición a esto se utiliza negación suave y fuerte, restricciones suaves, filtros y predicados potencialmente no finitos en la resolución del problema del diagnóstico, si se desea saber más sobre los mismos se puede consultar Eiter, Ianni et al. y Bihlmeyer, Faber, et al.[9] [2]

En particular la metodología para obtener el diagnóstico (y por consiguiente los cambios necesarios a la base de conocimientos) se basa en la idea de Reiter[28] que indica que es posible obtener un diagnóstico como parte del sistema (first principles) en lugar de obtenerlo de un acervo de diagnósticos pregenerados. Con esto último en mente, el proceso de diagnóstico se genera con una narrativa de acciones que de ser ejecutadas secuencialmente transformarán el mundo de uno en que los objetos se encuentran en la bodega al mundo en el que se encuentran en las repisas donde se asume que están dichos objetos por creencias anteriores, observaciones o inferencias.

La creencia por creencias anteriores es sencillamente la continuación de una creencia cuando no hay nada que la niegue. La creencia por observación es sencillamente aquella que se da después de observar el mundo y encontrar evidencia sensorial de la posición de un objeto. De mayor interés es la creencia por inferencia.

La creencia por inferencia se genera mediante la metodología *"Guess and check"* en la que una regla de cabeza disyuntiva explora el espacio de lugares posibles (aquellos que aún no han sido observados) y devuelve las diferentes posibilidades para la nueva creencia, Una serie de restricciones fuertes eliminan las soluciones que pese a ser lógicamente válidas son conflictivas (que un objeto esté en más de un lugar por ejemplo).

Una vez que se tiene el origen (los objetos en el almacén) y el resultado (las nuevas creencias) se revisa mediante predicados auxiliares cuáles son los objetos que se han movido y con esta información se vuelve a usar una etapa *"Guess"* en la que se usan cabezas disyuntivas para generar las posibles combinaciones de 2 objetos (parejas) para que éstos sean movidos por el asistente al mismo tiempo. Una serie de restricciones elimina posibles soluciones (como por ejemplo cuando una pareja menos deseable se mueve antes de una más deseable). Después de ésto se vuelven a usar predicados auxiliares para generar y luego separa las acciones que representan los emparejamientos.

Al final una serie de restricciones suaves (que son parte de *"Check"*) le asignan un costo a cada solución y el sistema regresa el conjunto de soluciones con el menor costo. Si hay más de una solución con este costo mínimo se elige la primera generada.

Para resolver el problema propuesto en este trabajo se decidió hacer lo siguiente: Representar los mundos (inicial, interno y observado) mediante el uso de conjuntos de características, Plantear el problema de la pérdida de situación como la diferencia entre los conjuntos que representan el mundo interno y el mundo observado. Resolver el problema en 3 etapas (diagnóstico, decisión y planeación). Para resolver la parte del diagnóstico, describir las acciones que el asistente debió haber tomado para que el mundo inicial se convirtiera por agencia de dicha cadena de acciones en el mundo observado. Generar las diferentes cadenas de acciones a través de conjuntos solución (Answer Sets), codificar el problema para resolverlo con *Answer Set Programming* ejecutado con el motor DLV.

El problema de la pérdida de situación es común a los agentes que se desenvuelven en un ambiente que no es continuamente observado y que es sujeto a cambios de los que el agente no es notificado (en el caso de mundos no completamente observables por ejemplo).[30] La pérdida de situación se genera cuando el mundo representado por una observación es diferente a alguna parte de la representación interna del mundo.

3.2 Answer Sets

Se generan conjuntos de cláusulas lógicas no contradictorias que proveen una explicación para el fenómeno observado. A estos conjuntos se les llama conjuntos solución (Answer Sets).

Para generar una explicación por secuencia mediante Answer Sets es necesario primero explicar a grosso modo la idea detrás de *Answer Set Programming* (*ASP*); dicha explicación se encuentra en el apéndice 5.

3.3 Diagnóstico de secuencia mediante Answer Sets

Aquí se especificará cómo se usa el mismo paradigma para resolver el problema específico en el caso de la pérdida de situación para el robot Golem en la prueba del Supermercado.

El problema de la pérdida de situación se resume a la discrepancia entre los mundos interno y la representación del mundo observado que se presenta cuando uno o más de los elementos de los conjuntos no se encuentra en el otro o viceversa.

Para resolver el problema de la pérdida de situación se toma la suposición de que la representación del mundo observado por los sentidos es correcta por lo que se buscará explicar cómo es que el mundo inicial se transformó en el mundo observado; para hacer esto se asume que el conocimiento es incompleto de manera que las observaciones se consideren como parte de un mundo que es observado y no necesariamente la totalidad del mundo observado (ya que de no hacerlo así cualquier elemento no observado en el momento se asumirá como faltante del mundo ya que la no información sobre algo se asume como la negación de esa información bajo el paradigma del mundo cerrado).

En el caso específico de la prueba del supermercado la inferencia se realiza sobre las acciones del asistente (las cuales no pueden ser observadas por Golem, por lo que se tienen que inferir por el sistema); en particular sobre las acciones que colocarían un objeto en algún lugar diferente al que se esperaba encontrarlo.

Así la pregunta que se responde en la etapa de diagnóstico es qué hizo el asistente para que el mundo se encuentre en el presente estado que es diferente al que se había supuesto. La inferencia se dispara en un primer momento al describir cómo es que las acciones del asistente cambiaron el mundo inicial en el de la representación interna, pero al encontrar un error (perder la situación) se hace una nueva inferencia para poder explicar cómo es que el mundo inicial se transformó en el mundo observado.

Así contamos con lo siguiente:

Bases:

Locaciones {*storage*, *start*, *shelf*_1, *shelf*_2, *shelf*_3}
Objetos {*Heineken*, *Coke*, *Kellogg's*, *Noodles*, *Biscuits*}
Agentes{*Asistente*}

Características

Característica {*sobre*[*Objeto*, *Locación*, *Tiempo*], *en*[*agente*, *Locación*, *Tiempo*]}

Acciones:

$Acciones\ \{ir\_a, tomar, depositar\}$

Las bases se combinan de la siguiente forma: Se toma una característica y se llenan sus espacios vacíos con elementos de la base indicada.

Una característica puede ser

$en[asistente, shelf\_1, 1]\ o\ sobre[Coke, Shelf\_2, 3],\ etc...$

Así, formamos conjuntos que describen el mundo en el que se desenvuelve el robot.

Mundo inicial
$\{sobre[Heineken, storage, 0], en[Asistente, start, 0]\}$

Mundo interno
$\{sobre[Heineken, shelf\_2, 1], en[Asistente, start, 1]\}$

Mundo observado
$\{sobre[Heineken, shelf\_1, 1], en[Asistente, start, 1]\}$

Se utiliza tiempo 1 para indicar que los mundos interno y observado son posteriores al mundo inicial; en realidad el tiempo de los mundos interno y observado depende de las acciones que se suponga haya tomado el asistente para transformar el mundo inicial en el mundo interno o en el mundo observado.

Notar cómo es que la diferencia en la característica entre el mundo interno y el observado es la discrepancia que se busca diagnosticar.

Para hacer una inferencia se genera una cadena de acciones que operan sobre las mismas bases que las características y tienen características como precondiciones y resultados.

Ejemplo:

Si comenzamos con el mundo

Mundo inicial
$MI = \{sobre[Heineken, storage, 0], en[Asistente, start, 0]\}$

Y queremos ver cómo se transforma en

Mundo observado
$MO = \{sobre[Heineken, shelf\_1, 4], en[Asistente, start, 4]\}$

Hacemos la siguiente lista de acciones que pudo haber tomado el asistente para transformar el mundo inicial *MI* en el mundo observado *MO* (una de varias posibilidades).

1.- Precondición $no(en[asistente, storage, 0])$, Acción $Ir\_a[asistente, storage, 0]$, Poscondición $en[asistente, storage, 1]$, Fluents $[en]$.

2.- Precondición $en[asistente, storage, 1]$, $sobre[Heineken, storage, 1]$, Acción $tomar[Heineken, storage, 1]$, Poscondición $no(sobre[Heineken, storage, 2])$, Fluents $[sobre]$.

3.- Precondición $no(en[asistente, shelf\_1, 2])$, Acción $ir\_a[asistente, shelf\_1, 2]$, Poscondición $en[asistente, shelf\_1, 3]$, Fluents $[en]$.

4.- Precondición $en[asistente, shelf\_1, 3]$, $no(sobre[Heineken, Shelf\_1, 3])$, Acción $depositar[heineken, shelf\_1, 3]$, Poscondición $Sobre[heineken, shelf\_1, 4]$, Fluents $[sobre]$.

5.- Precondición $no(en[asistente, start, 4])$, Acción $Ir\_a[asistente, start, 4]$, Poscondición $en[asistente, start, 5]$, Fluents $[en]$.

De esta manera una de las formas de transformar el mundo *MI* en el mundo *MO* se vuelve:

Diagnóstico: $\{Ir\_a[asistente, storage, 0]$, $tomar[Heineken, storage, 1]$, $ir\_a[asistente, shelf\_1, 2]$, $depositar[heineken, shelf\_1, 3]$, $Ir\_a[asistente, start, 4]\}$

Para llegar a este conjunto solución se necesitó de una serie de cláusulas tales que permitieran la exploración de las diferentes posibilidades; también fue necesario que se hayan rechazado aquellas que no representaban diagnósticos aceptables (aunque teóricamente si fueran conjuntos solución correctamente creados) mediante el uso de restricciones fuertes; y finalmente se ordenaron las posibilidades a través de restricciones débiles de manera que se eligió la que se presentó en el ejemplo en lugar de alguna otra.

3.4 Codificación en *Answer Set Programming*

Se consideran dos partes del programa en *ASP*; la primera es dinámica e incluye los mundos. La segunda parte es estática e incluye las reglas que siempre se cumplen en el proceso de diagnóstico; de esta forma se piensa en la parte dinámica como los datos y la parte estática como el programa aunque todas son cláusulas lógicas y pueden ir en el mismo archivo si así se desea. Se emplea la separación de datos y de programa para cambiar solamente un archivo para diagnosticar varios mundos diferentes pues idealmente nos gustaría tener un programa con la suficiente generalidad para que resuelva el problema de la pérdida de situación cualquiera que ésta sea (acotado a un universo finito de elementos en los mundos).

El programa se encuentra en el apéndice 6. Si se define el problema de la siguiente forma:

```
% Bases
agent(golem).
item(heiniken).
location(start).
location(shelf_1).
location(storage).

% Initial world
f_at(0,golem,start).
f_on(0,storage,heiniken).

% Observed world
goal_reached :-
      f_at(#maxint,golem,start),
      f_on(#maxint,shelf_1,heiniken).
```

El resultado que se obtiene es el siguiente:

{
a_goto(0, *golem*, *storage*),
a_take(1, *golem*, *heiniken*, *storage*),
a_goto(2, *golem*, *shelf*_1),
a_deliver(3, *golem*, *heiniken*, *shelf*_1),
a_goto(4, *golem*, *start*)
}

De esta manera se ha mostrado que si se presenta la solución al problema de la pérdida de situación como diagnóstico, decisión y planeación, y se describe la información que representa los mundos, inicial, interno y observado como conjuntos de características formadas por combinaciones de elementos de bases que son entidades de los mundos que son estudiados; entonces es posible encontrar la respuesta a la parte del diagnóstico mediante el uso de *Answer Set Programming* para generar y filtrar el conjunto de acciones que explican la transformación del mundo incial al mundo observado y de esta forma explicar el fenómeno (discrepancia) observado. Para generar la solución se emplearon los conceptos previamente expuestos (acciones, precondiciones, poscondiciones y fluents).

Un problema con este enfoque es que conforme crece el número de objetos y de locaciones en el mundo, la cantidad de posibilidades crece exponencialmente lo cual se traduce en un importante incremento en el espacio y tiempo. Para evitar este problema se decidió cambiar el enfoque de forma que dada la naturaleza del problema el algoritmo calcule las combinaciones para llevar los objetos (de 2 en 2 a lo más) del almacén a las repisas en vez de las secuencias de acciones. Éste enfoque se usa ya que la secuencia de acciones, una vez determinado el emparejamiento de objeto, se puede resolver

mediante un script. Esto da los mismos resultados pero el espacio del problema es mucho más pequeño ya que los fluents no se generan y sólo se toman en cuenta los del principio y del final. El programa usado para resolver el problema se analiza en el capítulo algoritmos y se presenta en su totalidad en los apéndices.

Capítulo 4: Algoritmos

Aquí se describen los algoritmos utilizados en el submódulo de diagnóstico; el código completo comentado se encuentra en los apéndices 1 (código en Prolog) y 2 (código en *Answer Set Programming*). La relación entre los algoritmos y diagramas de los mismos se pueden encontrar en el apéndice 3 (flujo de control).

Como producto del submódulo de diagnóstico se obtiene: a) Una lista de acciones que explica el cambio del mundo original en el que los objetos están en la bodega al mundo actual en el que se encuentran en dónde se esperaba que estuvieran, se observaron que están o se infiere que deben estar al no haber sido observados donde se suponía debían estar. b) Una base de conocimientos que refleja los cambios que el submódulo realiza dadas las observaciones e inferencias realizadas. El orden de ejecución así como las ligas entre las diferentes partes del submódulo de diagnóstico se describen en el apéndice 3 “Flujo de control”.

La prueba del supermercado consta de un modelo de diálogo principal que es el que inicia la prueba y contiene el modelo que llama al módulo de inferencia tanto al principio de la tarea como cuando se requiere planear; la naturaleza oportunista del módulo de inferencia permite que se llame al encontrar errores.

El algoritmo simplificado del submódulo de diagnóstico del módulo de inferencia en *ASP* es el siguiente:

Utilizar DLV para generar un diagnóstico y cambios a la base de conocimientos.

ETR_A_Diagnosis.dl (*Answer Set Programming*)

Éste es el código principal del submódulo ya es el que regresa el diagnóstico así como la lista de cambios a realizar en la base de conocimiento. Se ejecuta junto con ETR_A_Diagnosis_World.dl.

Lo “primero” (dado que teóricamente todas las reglas de *ASP* se cumplen al mismo tiempo) en calcularse es la localización de los objetos. Las reglas para esto son las siguientes:

- Si se vio el objeto en la última observación, esa es la localización en la que se asume que está.
- Si se cree que un objeto no debería de estar en el lugar observado y no se observa entonces se asume que está donde debería estar.
- Si se cree que un objeto debería estar en el lugar de la última observación pero no lo está, debe estar en otro lugar; éste depende de los lugares donde no se hayan ya hecho observaciones y no lo haya movido el robot anteriormente. Esta regla emplea una cabeza disyuntiva que generar una serie de posibilidades de estar o no estar en un lugar, lo cual es lo que nos permite explorar diferentes posibilidades y posteriormente elegir la más deseable mediante el empleo de restricciones ya sean duras o suaves.

- Si no se ha hecho ninguna observación todo debería estar en donde se creía que estaba antes.
- Si el objeto no puede estar en ninguno de los lugares disponibles se asume que no está en ningún lado (esta posibilidad no se contempla por la prueba del supermercado en sí).
- Si el objeto originalmente no aparecía entre los objetos que estaban en la prueba y se observa, se asigna donde se observó (para lo cual fue necesario usar una cabeza disyuntiva y una restricción suave para eliminar las respuestas en las que no aparece el objeto).
- Si un objeto no se observa donde se cree que debería estar pero se sabe que previamente el robot lo ha tomado; se asume que el objeto actualmente está en la mano de Golem y que será colocado inmediatamente en la posición actual.

Dado que calcular todos los diagnósticos para un número relativamente pequeño de objetos y lugares es computacionalmente muy costoso se optó por hacer el cálculo sobre una estructura superior que engloba varias acciones atómicas, de manera que se obtuviera la misma riqueza de soluciones con un costo computacional mucho menor. Para lograr esto se decidió hacer el cálculo sobre emparejamientos de objetos.

Este modo de calcular asume que el asistente tiene la capacidad de transportar a lo más 2 objetos (uno en cada mano) de un lugar a otro de forma que se tienen que realizar varios "viajes" en caso de que el número de objetos en la escena sea mayor a 2.

En este problema se pueden dar 5 tipos de emparejamiento como sigue:

1) Los objetos comienzan juntos y terminan juntos: Este emparejamiento es el que menos acciones requiere, ya que solamente se necesita: 1) Ir al lugar donde están las cosas , 2) tomar una cosa, 3) tomar la otra cosa , 4) ir a donde se tiene que depositar las cosas, 5) depositar una de las cosas, 6) depositar la otra cosa. El orden de tomar cosas depende del orden alfabético de su nombre.

2) Los objetos comienzan juntos y terminan separados: Este emparejamiento junto con el siguiente necesitan la misma cantidad de acciones por lo que son igualmente deseables. 1) Ir al lugar donde están las cosas , 2) tomar una cosa, 3) tomar la otra cosa , 4) ir a donde se tiene que depositar la primer cosa , 5) depositar la primer cosa, 6) ir al lugar donde hay que depositar la segunda cosa, 7) depositar la segunda cosa . El orden de tomar cosas depende del orden alfabético de su nombre.

3) Los objetos comienzan separados y terminan juntos: Este emparejamiento necesita la misma cantidad de acciones que el pasado por lo que son igualmente deseables. 1) Ir al lugar donde está la primer cosa, 2) tomar la primer cosa, 3)ir al lugar donde está la segunda cosa, 4)tomar la segunda cosa , 5) ir a donde se tiene que depositar las cosas , 6) depositar la primer cosa , 7) depositar la segunda cosa. El orden de tomar cosas depende del orden alfabético de su nombre.

4) Los objetos comienzan separados y terminan separados: Este es el emparejamiento menos deseable pues es el que usa la mayor cantidad de acciones para realizarse. 1) Ir al lugar donde está la primer cosa, 2) tomar la primer cosa, 3) ir al lugar donde está la segunda cosa, 4) tomar la segunda cosa, 5) ir a donde se tiene que depositar la primer cosa, 6) depositar la primer cosa, 7) ir al lugar donde hay

que depositar la segunda cosa, 8) depositar la segunda cosa . El orden de tomar cosas depende del orden alfabético de su nombre.

5) En caso de que haya un número impar de objetos, uno de ellos será transportado por sí mismo, este "emparejamiento" requiere de 4 acciones: 1) Ir al lugar donde está la cosa, 2) tomar la cosa, 3) ir a l lugar donde se debe depositar la cosa, 4) depositar la cosa.

Al utilizar esta estructura de emparejamientos se reduce significativamente la cantidad de cálculos necesarios para generar el diagnóstico pues, además de que se recurre a secuencias de acciones pre generadas (por lo que no tiene que ser exploradas), al tomar los objetos en orden alfabético, también se reduce la exploración de opciones trivialmente diferentes (tomar antes el objeto "a" que el objeto "b" vs tomar el objeto "b" antes que el objeto "a"). Es importante mencionar que como las acciones del asistente se calculan, se asume que todos los objetos serán entregados y que no hay una función de costos, probabilidades y recompensas para las acciones.

El código también reconoce todo lo que no está en donde se creía; estas cosas son las que deben ser actualizadas en la base de conocimientos para que las etapas de decisión y planeación funcionen correctamente (el submódulo de diagnóstico generado por grupo Golem funciona de la misma manera).

Al "final" del código, se traducen las parejas y sus acciones agrupadas en una lista no ordenada (pero si con indicadores de orden por tiempo) de acciones atómicas, las cuales se formatean en prolog para que sean aceptadas por las rutinas de Golem.

El código hace otros cálculos como contar el total de objetos (al incluir los recién observados) para calcular el número de parejas que debe formar de manera que no se hagan cálculos extra ya que ASP requiere tener un número previamente determinado y éste debe ser lo suficientemente grande para cubrir todos los cálculos aritméticos. Tener este número también hace que se exploren mucha más soluciones de las necesarias si no se acota al rango para el que se desea generar las soluciones.

El algoritmo es:

- Calcular tiempo a partir de número de objetos `etr_number_of_steps(Pasos)`
- Calcular en qué lugares aún se puede asumir que está un objeto no aun observado `etr_place_where_it_can_be(Lugar)`
- Generar nuevas creencias `etr_new_belief(Pasos,Tipo,Objeto,Lugar)` a partir de creencias pasadas, observaciones o inferencias
- Reconocer que objetos cambian de lugar de la creencia anterior a la nueva creencia `etr_change_of_place(Objeto,Lugar_1,Lugar_2)`
- Ver los cambios de lugar del lugar de inicio al lugar final `etr_difference(on,Objeto,Lugar_1,Lugar_2)`
- Generar los emparejamientos de objetos `etr_pairing(Tipo,Tiempo,Objeto_1,Objeto_2)` o si uno queda solo `etr_alone(Tiempo,Objeto)`

- Generar la secuencia para mover los objetos del emparejamiento `etr_explanation(Tipo,Tiempo,etr_action(Tiempo+1,accion,objeto/lugar),...)`,
- Descomponer la explicación en acciones atómicas `etr_diagnosis(Tiempo,Accion,Objeto/lugar)`

El resultado que genera es del estilo:

```
{etr_change_of_place(kellogs,storage,shelf2),
etr_diagnosis(11,goto,storage), etr_diagnosis(12,take,heineken),
etr_diagnosis(13,take,noodles),etr_diagnosis(14,goto,shelf1),
etr_diagnosis(15,deliver,heineken),
etr_diagnosis(16,deliver,noodles),etr_diagnosis(21,goto,storage),...}
```

Este algoritmo se apoya en programas de servicio que permiten transformar datos y darles permanencia en la base de conocimientos; a continuación se muestra el algoritmo simplificado para el submódulo de diagnóstico.

1) Traducir la base de conocimientos a un formato de DLV.
2) Extraer información relevante para el proceso de diagnóstico.
3) Utilizar DLV para generar un diagnóstico y cambios a la base de conocimientos. Este paso es el que se realiza en el archivo ETR_A_Diagnosis.dl.
4) Utilizar DLV para modificar la base de conocimientos transformada.
5) Utilizar DLV para reacomodar la información de la base de conocimientos transformada y modificada en formato de listas anidadas.
6) Limpiar las respuestas (diagnóstico y base de conocimientos) para que el formato sea idéntico al aceptado por Golem.

El código comentado más extensamente y con explicaciones de cada parte de la funcionalidad se encuentra en el apéndice 1 “Código en Prolog” y apéndice 2 “Código en *Answer Set Programming*”; aquí se explican los procesos ejecutados por los algoritmos en el orden en que se ejecutan,

4.1 ETR_P_User_Functions.pl (Prolog)

Se incluyen los archivos pertinentes escritos en Prolog para tener acceso a todas las funciones definidas. Se llama la parte de preparación para hacer el diagnóstico (que transforma la base de conocimientos a un formato compatible con DLV y se extrae la información esencial para el proceso de diagnóstico). La segunda parte del algoritmo llama la función que orquesta la generación del diagnóstico y la modificación de la base de conocimientos.

El código simplificado es:

```
etr_dlv_link(KB,Diagnóstico,KB_Modificada):-
      etr_ProgresivePreparation(KB),
```

```
    etr_diagnosis(Diagnóstico,KB_Modificada).
```

4.2 ETR_P_Preparation_For_diagnosis.pl (Prolog)

Se toma la base de conocimientos y se llama a las funciones en ETR_P_Transform_DB.pl para transformar la información contenida en la base de conocimientos a un formato compatible con DLV y guardarla en el archivo ETR_A_Prolog_To_ASP.dl, una vez que se tiene la información transformada, se utiliza ETR_A_Prepartion.dl para extraer la información más relevante al proceso de diagnóstico y guardarla en el archivo ETR_A_Diagnosis_World.dl.

El código simplificado es:

```
etr_ProgresivePreparation(KB):-
    etr_DataBase_To_ASP(KB),
    process_create(DLV, ETR_A_Preparation.dl,
    ETR_A_Prolog_To_ASP.dl, Res)
    save(Res,ETR_A_Diagnosis_World.dl).

etr_DataBase_To_ASP(KB):-
    open(KB,golem_KB.txt),
    append(KB,.,Data),
    save(Data,golem_KB_helper.txt),
    open(Data2,golem_KB_helper.txt),
    etr_create_file_for_aspdlv(Data2,KB_ASP),
    save(KB_ASP,ETR_A_Prolog_To_ASP.dl).
```

4.3 ETR_P_Transform_DB_To_ASP.pl (Prolog)

Se toma la base de conocimientos y se transforman las listas de tipo clase a conjuntos de predicados atómicos que son utilizados en DLV, cada elemento de la lista original (cada clase), se analiza para descomponerla en: Clase, Padre, Propiedades, Relaciones e Instancias, las Instancias se descomponen en Nombre de Instancia, Propiedades de instancia y Relaciones de instancia. Se toma en cuenta la posibilidad de que las propiedades (tanto de clase como de instancia) sean negativas. Hay un manejo particular de las clases que representan las acciones de Golem pues éstas no se utilizan dado que el diagnóstico se hace sobre las acciones del asistente; esta información no es relevante y sólo es encapsulada para su eventual recuperación al final del proceso de diagnóstico.

El código simplificado es:

```
etr_create_file_for_aspdlv([]).
etr_create_file_for_aspdlv([class(X,actions,P,R,I)|T]):-
    write_parent(X,actions),
    wirte_special_class(X,P,R,I),
```

```
      etr_create_file_for_aspdlv(T).
etr_create_file_for_aspdlv([class(X,Y,P,R,I)|T]):-
      wirte_class(X),
      write_parent(X,Y),
      wirte_property_list(X,P),
      write_relationship_list(X,R),
      write_instance_list(X.I),
      etr_create_file_for_aspdlv(T).
```

El código transforma clases de la base de conocimientos como :

```
class(cereal,food,[inv=>0],[],[[id=>c1,[brand=>kellogs,original_id=>c
1,in=>shelf2],[]]])
```

En predicados atómicos como:

```
class("cereal").
parent("cereal",food).
property("cereal",inv,0).
instance("cereal",id,c1).
property(c1,brand,kellogs).
property(c1,original_id,c1).
property(c1,in,shelf2).
```

4.4 ETR_A_Prepartion.dl (*Answer Set Programming*)

Se ejecuta junto con ETR_A_Prolog_To_ASP.dl (La base de conocimientos de Golem en un formato que DLV entiende como *ASP*). Este programa esencialmente se usa para filtrar la información que es relevante para el proceso de diagnóstico. Se generan diferentes reglas para las observaciones nuevas (aquellas que se efectuaron en el lugar en el que está Golem actualmente) y las observaciones viejas (las que se hicieron anteriormente). Esta distinción es importante para el proceso de diagnóstico. La salida de este programa se guarda en el archivo ETR_A_Diagnosis_World.dl.

El algoritmo es:

- Generar los siguientes predicados: **`etr_object(O), etr_location(L), etr_at_start(human,start), etr_at_final(human,start), etr_original(0,O,storage), etr_belief(O,L), etr_new_observation(O,L), etr_old_observation(O,L), etr_place(L), etr_grasped(O).`**

El código genera predicados como los siguientes:

Objetos en escena

```
etr_object(kellogs).
```

Locaciones de la escena
```
etr_location(start).
```

Lugar donde se asume que estaban los objetos originalmente
```
etr_original(0,kellogs,storage).
```

Creencia actual sobre la localización actual de los objetos
```
etr_belief(kellogs,shelf2).
```

Las últimas observaciones que ha realizado Golem
```
etr_new_observation(coke,shelf3).
```

Observaciones Pasadas que ha hecho golem
```
etr_old_observation(kellogs,shelf2).
```

Los objetos que han sido tomados (y por ende movidos) por Golem.
```
etr_grasped(noodles).
```

4.5 ETR_P_Diagnosis.pl (Prolog)

Este código llama el programa en *Answer Set Programming* que genera la lista de acciones que conforma el diagnóstico (aunque aún no depuradas) y los cambios a la base de datos, llama a las funciones que depuran el diagnóstico y también llama al código en *Answer Set Programming* que incorpora los cambios a la base de conocimientos y la vuelve a acomodar en formato de listas anidadas. Una vez que se han realizado estas acciones, se llama al código que depura las listas anidadas y las agrega en una sola lista de clases con listas anidadas que es la base de conocimientos con los cambios necesarios incorporados.

El código simplificado es:

```
etr_diagnosis(DiagnosisList,New_Database):-
      process_create(DLV,ETR_A_Diagnosis.dl,
      ETR_A_Diagnosis_World.dl'], Res_1),
      etr_split(Res_1,Diagnosis,Changes),
      save(Changes,ETR_A_Changes.dl),
      etr_bubble_sort(Diagnosis, SortedDiagnosis),
      etr_make_diagnosis_list(SortedDiagnosis, DiagnosisList),
      process_create(DLV, ETR_A_Transform_Base_01.dl,
      ETR_A_Prolog_To_ASP.dl,ETR_A_Changes.dl,Res_2),
      save(Res_2,ETR_A_Auxiliary_Base.dl),
      process_create(DLV, ETR_A_Transform_Base_02.dl,
      ETR_A_Auxiliary_Base.dl, Res_3),
```

```
save(Res_3,ETR_A_Return_Base.dl),
process_create(DLV, ETR_A_Return.dl, ETR_A_Return_Base.dl,
Res_4),
etr_format_data_base(Res_4,New_Database).
```

El código genera un diagnóstico de la forma:
`[move(storage),take(coke),take(heineken),move(shelf1),deliver(coke),deliver(heineken),...]`

Así como una base de conocimientos de la forma :
`[[class(cereal,food,[inv=>0],[],[[id=>c1,[brand=>kellogs,original_id=>c1,in=>shelf2],[]]])],...]`

4.6 ETR_A_Diagnosis.dl (*Answer Set Programming*)

Este algoritmo ya fue discutido al principio del capítulo, se incluye aquí una mención para mostrar su lugar en la ejecución del submódulo de diagnóstico.

4.7 ETR_A_Transform_Base_01.dl (*Answer Set Programming*)

Este código se llama junto con ETR_A_Prolog_To_ASP.dl y ETR_A_Changes.dl y cambia las propiedades de localización de los objetos que lo necesiten según lo indicado en el archivo de cambios; se pasa toda la base de conocimiento tal cual está con la excepción de la propiedad de localización que se pasa con el nuevo lugar si lo hay en el archivo de cambios y con el lugar que tenía antes si el objeto no aparece en el archivo de cambios. El resultado de este código se guarda en ETR_A_Auxiliary_Base.dl.

El algoritmo es:

- Generar los siguientes predicados: **`properta(A,in,Z)`**, **`properta(A,in,C)`**, **`class(A)`**, **`parent(A,B)`**, **`properta(A,B,C)`**, **`-properta(A,B,C)`**, **`instance(A,B,C)`**, **`special_class(A,B,C,D,E).`**

Para generar esta información se apoya en los siguientes predicados:
`etr_no_change(A), etr_change(A).`

Así al tomar en cuenta que en el archivo de cambios aparezca:

`etr_change_of_place(kellogs,storage,shelf2),`

Un predicado que originalmente se mostraba de la siguiente forma en la base de conocimientos que se usa para *Answer Set Programming*:

```
property(c1,in,storage).
```

Pasa de la siguiente forma

```
properta(c1,in,shelf2).
```

4.8 ETR_A_Transform_Base_02.dl (*Answer Set Programming*)

Este código se llama junto con ETR_A_Auxiliary_Base.dl y se pasa la base de conocimiento tal cual viene en ese archivo excepto por las propiedades de localización que son regresadas a su nombre original. El resultado de este program se guarda en ETR_A_Return_Base.dl .

El algoritmo es:

- Generar los siguientes predicados: **`class(A), parent(A,B), property(A,B,C), -property(A,B,C), instance(A,B,C), special_class(A,B,C,D,E).`**

Así, por ejemplo, un predicado que se recibe de la siguiente manera:

```
properta(c1,original_id,c1).
properta(c1,in,shelf2).
```

Se regresa de la siguiente forma:

```
property(c1,original_id,c1).
property(c1,in,shelf2).
```

4.9 ETR_A_Return.dl (*Answer Set Programming*)

Este código se ejecuta junto con ETR_A_Return_Base.dl. Se usan las capacidades de *ASP* para manejar listas de manera que se forman clases completas según la estructura de listas anidadas de Golem (aunque aún habrá que hacer cambios de formato en Prolog para terminar con una sola lista anidada para la base de conocimientos completa).

Los predicados del archivo ETR_A_Return_Base.dl, se combinan de forma que se arman primero las propiedades y relaciones de las instancias (cada una en su propia lista) de cada clase para luego formar una lista de instancias y al final juntar a una lista de relaciones y propiedades para así termina con una clase encapsulada en un solo predicado.

El algoritmo es:

- Hacer lista de Propiedades de una instancia
  etr_final_instance_properties_list(Class,A,Lista).
- Hacer lista de Relaciones de una instancia
  etr_final_instance_relationship_list(Class,A,Lista).
- Agrupar Instancia junto con propiedades y relaciones de instancia
  etr_instance(Class,etr_operator(Class,B,C),etr_final_instance_properties_list(Class,C,X),etr_final_instance_relationship_list(Class,C,Y)).
- Hacer una lista de instancias de clase
  etr_final_class_instance_list(Clase,Lista).
- Hacer una lista de propiedades de clase
  etr_final_class_properties_list(Clase,Lista).
- Hacer una lista de relaciones de clase
  etr_final_class_relationships_list(Clase,Lista).
- Agrupar Clase junto con su lista de propiedades, su lista de relaciones y su lista de instancias
  etr_class(Clase,Padre,Propiedades,Relaciones,Instancias).

Asi, si se tiene una clase:

```
class("cereal").
parent("cereal",food).
property("cereal",inv,0).
instance("cereal",id,c1).
property(c1,brand,kellogs).
property(c1,original_id,c1).
property(c1,in,storage).
```

Se obtiene el siguiente predicado:

```
etr_class("cereal",food,[etr_class_property("cereal",inv,0)],[],[etr_class_instance("cereal",etr_operator("cereal",id,c1),etr_final_instance_properties_list("cereal",c1,[etr_instance_property(c1,brand,kellogs),etr_instance_property(c1,original_id,c1),etr_instance_property(c1,in,storage)]),etr_final_instance_relationship_list("cereal",c1,[]))])
```

4.10 ETR_P_Return.pl (Prolog)

Este código sirve para transformar la base de conocimientos del formato de listas anidadas generadas en *Answer Set Programming* al formato que utiliza Golem. Cada clase se analiza y se retira la información extra (nombres de predicado y atributos extra para poder ligar la información). El código emplea la información en una cadena de texto que representa la información de la base de conocimientos generada por el código arriba descrito y usa el archivo ETR_Trasform_List.txt como auxiliar en la generación del resultado.

El código simplificado es el siguiente:

```
etr_format_data_base(Text,Database_With_Format):-
    atomic_list_concat(Base, ', ', Text),
    etr_list_base(Base,Database_With_Format).

etr_list_base([H|T],L):-
    H=etr_class(A,B,C,D,E),
    etr_list_class_properties(C,C2),
    etr_list_class_relationships(D,D2),
    etr_list_class_instances(E,E2),
    P=class(A2,B,C2,D2,E2),
    etr_list_base(T,L2),
    L= [P|L2].
```

Así, si se recibe una clase:

```
etr_class("cereal",food,[etr_class_property("cereal",inv,0)],[],[etr_
class_instance("cereal",etr_operator("cereal",id,c1),etr_final_instan
ce_properties_list("cereal",c1,[etr_instance_property(c1,brand,kellog
s),etr_instance_property(c1,original_id,c1),etr_instance_property(c1,
in,storage)]),etr_final_instance_relationship_list("cereal",c1,[]))])
```

Se regresa:

```
class(cereal,food,[inv=>0],[],[[id=>c1,[brand=>kellogs,original_id=>c
1,in=>storage],[]]])
```

Implementación:

El submódulo de diagnóstico de *Answer Set Programming* está estructurado con 3 tipos de archivos: los de control escritos en Prolog, los de trabajo para DLV, escritos en *Answer Set Programming*, y por último los auxiliares que se usan para darle permanencia a los cambios de información de la base de conocimiento, éstos se codifican para ser interpretados ya sea por Prolog o por *Answer Set Programming*.

Archivos de control escritos en Prolog:
ETR_User_Functions.pl
ETR_P_Preparation_For_Diagnosis.pl
ETR_P_Transform_DB_To_ASP.pl
ETR_P_Diagnosis.pl
ETR_P_Return.pl

Archivos de trabajo escritos en *Answer Set Programming*:

ETR_A_Preparation.dl
ETR_A_Diagnosis.dl
ETR_A_Transform_Base_01.dl
ETR_A_Transform_Base_02.dl
ETR_A_Return.dl

Archivos auxiliares:

golem_KB_helper.txt,
ETR_A_Auxiliary_Base.dl,
ETR_A_Changes.dl,
ETR_A_Diagnosis_World.dl,
ETR_A_Prolog_To_ASP.dl,
ETR_A_Return_Base.dl,
ETR_Trasform_List.txt .

Capítulo 5 : Experimento, resultados y análisis

5.1 Experimento

Para probar la validez de la solución y la posibilidad de sustituir el código desarrollado por el grupo Golem por el propuesto en este trabajo, validando así la equivalencia operacional de los algoritmos, se sustituyó el módulo de inferencia en su totalidad mientras se ejecutaba el mismo código de la prueba.

Se realizaron varias ejecuciones del algoritmo con modificaciones en el posicionamiento de los objetos así como ejecuciones en las que por razones de percepción el robot realizó inferencia diferentes a las esperadas (aunque válidas para el mundo que se percibía). También se realizaron pruebas en las que se empleaban componentes diseñados en *ASP* junto a componentes diseñados en Prolog dentro del mismo módulo de inferencia para probar la compatibilidad de los componentes de la siguiente manera: se utilizó el componente diseñado en *ASP* para la parte de diagnóstico junto con los componentes diseñados en Prolog para las partes de toma de decisión y planeación. De la misma forma, se empleo el componente diseñado en Prolog para diagnóstico junto con los componentes diseñados en *ASP* para las partes de toma de decisión y planeación (esto último fue realizado por Ricardo Adolfo Fierro Villaneda). Finalmente se realizó la corrida estándar de la prueba mediante el uso del módulo de inferencia desarrollado por grupo Golem y luego el mismo escenario con el módulo de inferencia en *ASP* de estas ejecuciones se tomaron métricas tales como los tiempos de ejecución y el contenido de las bases de conocimiento que generaban al finalizar la prueba.

El ejercicio transcurre de la siguiente forma:

Al iniciar el ejercicio Golem se encuentra en la posición de inicio y el asistente ha colocado 4 objetos en el escenario: Una caja de cerveza *Heineken* y unos *Noodles* en la repisa 1 ("*shelf of drinks*"), unos *Kellogs* en la repisa 2 ("*shelf of food*") y una lata de *Coca* y unos *Biscuits* en la repisa 3 ("*shelf of bread*").

Golem indica que está a la espera instrucciones y el asistente le informa que ha colocado *Coca*, *Heineken*, *Noodles* y *Biscuits* en el escenario (notar la ausencia de *Kellogs*). Golem registra en su base de datos los productos y les asigna las posiciones esperadas (*Coca* y *Heineken* en la repisa 1, *Noodles* en la repisa 2, *Biscuits* en la repisa 3). Después responde preguntas sobre su base de conocimiento; el cliente le pregunta si tiene *Kellogs* y golem le dice que si pero no en existencia. El cliente le pregunta si tiene frituras mexicanas y Golem le responde que no tiene conocimiento de ese producto. El cliente le pregunta si tiene cerveza y Golem le dice que si y le pregunta si es mayor de edad; el cliente le responde que no por lo que Golem le responde que no le puede traer la *Heineken.* El cliente pregunta si tiene una bebida para toda las edades y golem le responde que si, *Coca*, y que se la traerá. Cuando Golem pregunta si el cliente desea algo más, el cliente dice que no. Con esta información Golem diagnostica que el asistente ha colocado la *Coca* y la *Heineken* en la repisa 1, los *Noodles* en la repisa 2

y los *Biscuits* en la repisa 3. Su decisión es entregar la *Coca* por lo que su plan es ir a la repisa 1 buscar la *Coca*, tomarla, regresar con el cliente y entregarle la *Coca*.

Cuando Golem llega a la repisa 1 analiza la escena y observa que hay *Heineken* y *Noodles* en la repisa 1 pero no hay *Coca*; esto hace que se llame el sistema de inferencia. El sistema infiere que dado que los *Noodles* están fuera de su lugar y la *Coca* también, la *Coca* debe estar en la repisa 2 para mantener el mismo número de objetos por repisa. Su diagnóstico entonces es que el asistente colocó *Heineken* y *Noodles* en la repisa 1, la *Coca* en la 2 y los *Biscuits* en la 3. Su decisión es entregar la *Coca* y reacomodar los *Noodles* (ya que hay que llevarlos a la misma repisa a la que se dirige para buscar la *Coca*). Su plan es Buscar los *Noodles*, tomarlos, ir a la repisa 2, depositarlos, buscar la *Coca*, tomarla, ir con el cliente y entregar la *Coca*.

Cuando Golem llega a la repisa 2 observa que hay *Kellogs* y no hay *Coca*, deposita los *Noodles* y busca la *Coca*; al no encontrarla, llama al sistema de inferencia; éste infiere que dado que la *Coca* no está en la repisa 1 o en la 2 entonces sólo queda que esté en la 3, aunque esto haga que las cantidades de objetos sean diferentes a las inicialmente esperadas. Su diagnóstico entonces es que el asistente llevó la caja de *Heineken* y los *Noodles* a la repisa 1, *Kellogs* a la 2 y los *Biscuits* y la *Coca* a la repisa 3. Su decisión es llevarle la *Coca* al cliente y su plan entonces es ir a la repisa 3, buscar la *Coca*, tomar la coca, ir con el cliente y entregarle la *Coca*.

Al llegar a la repisa 3 golem observa que la *Coca* y los *Biscuits* están en la repisa 3. Golem busca la *Coca*, la encuentra la toma y regresa con el cliente para entregar la *Coca*. Una vez terminado esto revisa su base de conocimiento y encuentra que los *Kellogs* si están disponibles, se los ofrece al cliente pero éste dice que no.

El experimento transcurre de la misma forma para ambas implementaciones del módulo de inferencia.

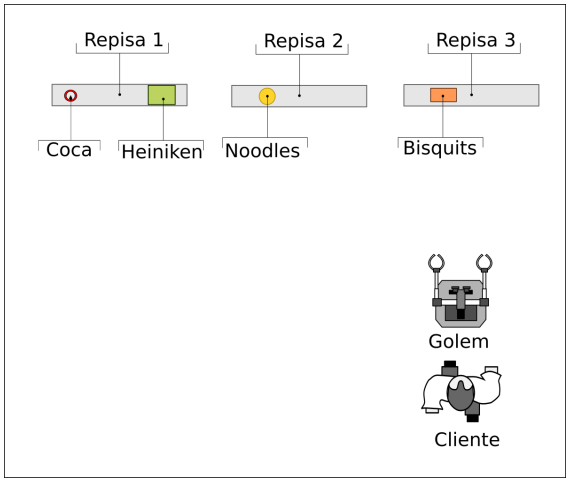

Fig1.

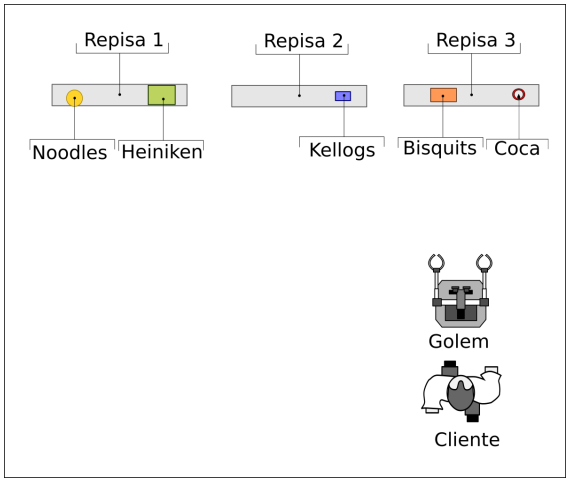

Fig 2.

Fig 1 representa la creencia inicial de Golem dada la información provista por el asistente, Fig 2 representa la realidad.

5.2 Resultados

A continuación se presentan los resultados relevantes al submódulo de diagnóstico del módulo de inferencia que se escribió en *Answer Set Programming*.

Se realizó la misma prueba primero con el módulo de inferencia diseñado por grupo Golem que emplea árboles de búsqueda con heurísticas; después se realiza la misma prueba (el mismo código de hecho) con *Answer Set Programming*.

La siguiente tabla presenta los tiempos en milisegundos que tomó la ejecución de los submódulos del módulo de inferencia durante el ejercicio de la prueba (primero se presentan los tiempos para los submódulos del módulo programado en Prolog y después los del módulo programado en *ASP*).

	Árbol de búsqueda (Prolog)	***Answer Set Programming***
Diagnóstico inicial	233 ms.	29 ms.
Decisión inicial	659 ms.	9 ms.
Planeación inicial	23 ms.	37 ms.
Diagnóstico antes de moverse	2012 ms.	70 ms.
Decisión antes de moverse	928 ms.	51 ms.
Planeación antes de moverse	155 ms.	96 ms.
Diagnóstico en repisa 1	1287 ms.	76 ms.
Decisión en repisa 1	1332 ms.	63 ms.
Planeación en repisa 1	249 ms.	618 ms.
Diagnóstico en repisa 2	1553 ms.	181 ms.
Decisión en repisa 2	841 ms.	56 ms.
Planeación en repisa 2	131 ms.	352 ms.

Tabla 1.- Tiempos (en milisegundos) de ejecución de los submódulos de Diagnóstico, Decisión y Planeación para la prueba del supermercado con 4 objetos, 4 lugares y un quinto objeto revelado a media prueba

La siguiente tabla presenta las salidas del submódulo de diagnóstico en las diferentes llamadas que se realizan a lo largo de la prueba del supermercado. Primero se presentan las del submódulo programado en Prolo y después las del submódulo programado en *ASP*. El resultado muestra cómo aunque los pasos de los diagnósticos son diferentes entre los submódulos las posiciones resultantes de los objetos en ambos submódulos son iguales.

	Árbol de búsqueda (Prolog)	***Answer Set Programming***
Inicialización	**Diagnostico**:[] **Resultado**:(Heineken, storage),(noodles, storage), (bisquits, storage), (coke, storage).	**Diagnostico**:[] **Resultado**:(Heineken, storage),(noodles, storage), (bisquits, storage), (coke, storage).
Antes de moverse	**Diagnostico**:[move(shelf3),deliver(bisquits),move(shelf2),deliver(noodles),move(shelf1),deliver(heineken),deliver(coke)] **Resultado**:(Heineken, shelf1),(noodles, shelf2), (bisquits, shelf3), (coke, shelf1).	**Diagnostico**:[move(storage),take(coke),take(heineken),move(shelf1),deliver(coke),deliver(heineken),move(storage),take(bisquits),take(noodles),move(shelf3),deliver(bisquits),move(shelf2),deliver(noodles)] **Resultado**:(Heineken, shelf1),(noodles, shelf2), (bisquits, shelf3), (coke, shelf1).
Tras no encontrar *Coca* y si encontrar *Noodles* fuera de lugar	**Diagnostico**:[move(shelf3),deliver(bisquits),move(shelf2),deliver(coke),move(shelf1),deliver(heineken),deliver(noodles)] **Resultado**:(Heineken, shelf1),(noodles, shelf1), (bisquits, shelf3), (coke, shelf2).	**Diagnostico**:[move(storage),take(heineken),take(noodles),move(shelf1),deliver(heineken),deliver(noodles),move(storage),take(bisquits),take(coke),move(shelf3),deliver(bisquits),move(shelf2),deliver(coke)] **Resultado**:(Heineken, shelf1),(noodles, shelf1), (bisquits, shelf3), (coke, shelf2).
Tras acomodar *Noodles*, observar *Kellogs* y no encontrar *Coca*.	**Diagnostico**:[move(shelf3),deliver(bisquits),deliver(coke),move(shelf1),deliver(heineken),deliver(noodles),move(shelf2),deliver(k	**Diagnostico**:[move(storage),take(heineken),take(noodles),move(shelf1),deliver(heineken),deliver(noodles),move(storage),take(bi

	ellogs)] **Resultado**:(Heineken, shelf1),(noodles, shelf1), (bisquits, shelf3), (coke, shelf3), (kellogs,shelf2).	squits),take(coke),move(shelf3), deliver(bisquits),deliver(coke),move(storage),take(kellogs),move(shelf2),deliver(kellogs)] **Resultado**:(Heineken, shelf1),(noodles, shelf1), (bisquits, shelf3), (coke, shelf3), (kellogs,shelf2).

Tabla 2.- Diagnósticos recibidos y posiciones resultantes durante la ejecución de la prueba; es importante notar que aunque los diagnósticos no son idénticos, los de *Answer Set Programming* toman en cuenta los "viajes" necesarios para mover los objetos al hacer uso de 2 manos y la necesidad de regresar a la bodega por los objetos restantes; los diagnósticos son equivalentes en el sentido de que los objetos terminan en las mismas repisas (las posiciones resultantes son las mismas).

De la misma forma, la base de conocimientos se modifica en ambos casos de forma similar para representar los cambios de lugar de los objetos que han sido observados en un lugar diferente al esperado o que a través de la inferencia del submódulo de diagnóstico se espera estén en un lugar diferente al originalmente esperado. Los cambios son; tras descubrir que los "noodles" están en la repisa 1 en vez de la repisa 2; se hace el cambio y dado que la misma observación omite la coca, a través de la inferencia se cambia la posición esperada de la coca de la repisa 1 a la repisa 2 (de esta manera se conserva el número de objetos por repisa con 2 objetos en la repisa 1, 1 objeto en la repisa 2 y 1 objeto en la repisa 3). Posteriormente al ir a la repisa 2 se observa que la coca no se encuentra ahí por lo que se hace un segundo cambio a la base de conocimientos para reflejar la nueva inferencia de que la coca se debe encontrar en la repisa 3.

Pese a que el orden de las clases es diferente, la información contenida en las bases es equivalente; esto es lo que permite que la prueba del supermercado se desarrolle en la misma forma con cualquiera de los módulos de inferencia que se use.

Las bases de conocimiento, tanto la original como las resultantes después de haber realizado los ejercicios mediante el algoritmo desarrollado por el grupo Golem (árboles de búsqueda con Heurísticas) como el algoritmo en *Answer Set Programming* se encuentran en el apéndice 4.

Se observa que ambos enfoques son capaces de generar resultados similares en una prueba de inferencia.

La diferencia de tiempos es pequeña (en favor de *Answer Set Programming*) pero por experiencia es de esperarse que al aumentar el tamaño de la prueba (más objetos y más lugares) la diferencia en tiempo se reduzca e incluso se invierte a favor de los árboles de búsqueda, pues si bien en este problema específico fue posible eficientizar el algoritmo de inferencia, dado que se hacía exclusivamente para el asistente, éste no será siempre el caso.

5.3 Análisis

A continuación se presenta un análisis comparativo de la complejidad de los submódulos de diagnóstico de los módulos de inferencia tanto el diseñado por grupo Golem como el que se presenta en este trabajo. El análisis se hace al nivel algoritmo y no se toma en cuenta la complejidad de resolución de cláusulas de las tecnologías empleadas.

Dado que el submódulo de diagnóstico diseñado por grupo golem utiliza un árbol de búsqueda para generar su diagnóstico (como el listado de las acciones que llevan al mundo como es percibido actualmente) se asume como un árbol BFS en el que se exploran todos los nodos de un nivel antes de pasar al siguiente para encontrar el de menor peso; en ese caso se tiene una complejidad relativa a a la entrada (al pensar que la entrada se representa como 3 listas: una de objetos, otra de acciones y la última de lugares) en la escena para así generar las diferentes narrativas; si se toma este cálculo por nivel (que no es exacto pues no todas las acciones tienen sentido con todos los objetos) se obtiene que la complejidad por nivel del árbol es de (Número de Lugares * Número de objetos * Número de acciones) y este número se eleva exponencialmente para cada nivel en el que busca el árbol; se termina con una complejidad de $(A \times B \times C)^L$ si se considera que $A = B = C$ (en el peor de los casos); se termina con una complejidad de $(A^3)^L$ con A un número arbitrario de acciones, objetos o lugares y L la longitud de la explicación. Esto es lo mismo a $O(N^P)$ con N dependiente del tamaño de la entrada y P un polinomio

Para analizar la complejidad del submódulo de diagnóstico en *Answer Set Programming* es necesario primero revisar cómo se generan las respuestas ya que al hacer una regla, ésta presenta una cantidad de posibilidades que crece exponencialmente con el número de variables libres en el cuerpo de la regla, así un regla ejemplo:

```
ob(a)
ob(b)
comb(X,Y):- ob(X), ob(Y).
```

Genera una respuesta que es

```
{ob(a). ob(b). comb(a,a), comb(a,b), comb(b,a), comb(b,b)}.
```

Si se incluye una cabeza disyuntiva

```
ob(a)
ob(b)
comb(X,Y) v -comb(X,Y):- ob(X), ob(Y).
```

Se obtienen 16 respuestas (las 2 cláusulas de las cabeza disyuntiva elevado al número de posibles combinaciones de las cláusulas de inicio):

```
{ob(a), ob(b), -comb(a,a), -comb(b,a), -comb(a,b), -comb(b,b)}
{ob(a), ob(b), comb(a,a), -comb(b,a), -comb(a,b), -comb(b,b)}
```

```
{ob(a), ob(b), -comb(a,a), comb(b,a), -comb(a,b), -comb(b,b)}
{ob(a), ob(b), comb(a,a), comb(b,a), -comb(a,b), -comb(b,b)}
{ob(a), ob(b), -comb(a,a), -comb(b,a), comb(a,b), -comb(b,b)}
{ob(a), ob(b), comb(a,a), -comb(b,a), comb(a,b), -comb(b,b)}
{ob(a), ob(b), -comb(a,a), comb(b,a), comb(a,b), -comb(b,b)}
{ob(a), ob(b), comb(a,a), comb(b,a), comb(a,b), -comb(b,b)}
{ob(a), ob(b), -comb(a,a), -comb(b,a), -comb(a,b), comb(b,b)}
{ob(a), ob(b), comb(a,a), -comb(b,a), -comb(a,b), comb(b,b)}
{ob(a), ob(b), -comb(a,a), comb(b,a), -comb(a,b), comb(b,b)}
{ob(a), ob(b), comb(a,a), comb(b,a), -comb(a,b), comb(b,b)}
{ob(a), ob(b), -comb(a,a), -comb(b,a), comb(a,b), comb(b,b)}
{ob(a), ob(b), comb(a,a), -comb(b,a), comb(a,b), comb(b,b)}
{ob(a), ob(b), -comb(a,a), comb(b,a), comb(a,b), comb(b,b)}
{ob(a), ob(b), comb(a,a), comb(b,a), comb(a,b), comb(b,b)}
```

El programa de diagnóstico tiene aproximadamente 60 cláusulas no disyuntivas con unas 10 cláusulas en el cuerpo como máximo y 7 cláusulas disyuntivas (2 predicados) con 10 cláusulas como máximo, 17 restricciones fuertes y 9 débiles. Las restricciones fuertes se comparan con los resultados una vez generados y se descartan. Por otro lado las restricciones suaves se comparan al final para ver cuáles son las soluciones con menor penalización

Si se toma que se opera sobre la capacidad de hacer parejas (6 reglas disyuntivas) éstas operan sobre la generación de nuevas creencias (2 reglas disyuntivas) y se tiene un número N de artículos y M de lugares en el peor de los escenarios N = M y aproximadamente la mitad de las cláusulas son de 3 partes y el resto se dividen entre cláusulas de 1 ó 2 partes y se tiene que cada regla no disyuntiva representa $1 \times 2N \times 3N^2 \times 5N^3$ o sea $O(N^3)$ predicados generados.

Así, con 60 reglas se tiene $60 \times O(N^3)$ que es $O(N^3)$.

Cada regla disyuntiva representa $2^{(2N \times 3N^2 \times 5N^3)}$ o sea $O(2^{N^3})$ respuestas diferentes.

Como se tienen 7 cláusulas disyuntivas se tiene $O(2^{7N^3})$ que es $O(2^{N^3})$.

Así, el número de respuestas posibles (en el peor escenario sin tomar en cuenta reglas que no se forman por restricciones dentro de la regla) es $O(2^{N^3}) \times O(N^3)$ que es $O(2^{N^3})$.

Esto se expresa como $O(C^N)$ con C > 1 y N dependiente de la entrada.

Así se ve que mientras la solución del problema diseñada por grupo Golem es de orden polinomial, la solución en *Answer Set Programming* es de orden exponencial. Para universos pequeños como se ve en los resultados, éste puede no ser un impedimento, pero conforme la entrada crezca, la capacidad de esta solución, en particular de resolver el problema, será cada vez menos atractiva por la complejidad que representa.

Cabe mencionar que en una encarnación temprana del submódulo de diagnosis se utilizó un acercamiento inspirado en el mundo de cubos presentado en Buccafurri, Faber, et al. y Eiter, Faber, et al. [6] [8] similar al del apéndice 6; sin embargo resultó demasiado complejo computacionalmente y presentaba tiempos de ejecución desmedidamente altos incluso para problemas relativamente pequeños. Esto se menciona para mostrar que el enfoque del problema puede cambiar radicalmente la eficiencia con la que se resuelve al emplear *Answer Set Programming*.

Es importante hacer notar que este análisis representa la forma más básica de resolver un programa de *ASP* a través de aterrizaje de las cláusulas y luego se procesa con un *ASP* solver ésta no es la forma más eficiente de hacerlo pero es la más común (técnicas más "inteligentes" de aterrizaje de variables dependen del programa intérprete que se emplee).

Capítulo 6: Conclusiones

Este trabajo muestra que es posible obtener resultados equivalentes en tiempos similares al utilizar árboles de búsqueda con heurísticas o *Answer Set Programming* para realizar diagnóstico en robots de servicio. Las diferencias en eficiencia, así como en implementación, son lo suficientemente pequeñas para decir que en pruebas acotadas a tamaños similares al de la prueba del supermercado, es razonable pensar que ambos enfoques son viables y que utilizar uno u otro dependerá de las preferencias del desarrollador.

Las características de *Answer Set Programming* aunadas a los predicados para manejo de listas, los predicados aritméticos y los predicados comparativos, y el uso de negación tanto fuerte como negación suave de DLV permiten resolver problemas similares al aquí presentado de forma eficiente e intuitiva. Mayor experiencia tanto con *Answer Set Programming* como con el Sistema DLV permitirán refinar los algoritmos aún más.

La recomendación que se hace se resume en decir que *Answer Set Programming* es una opción válida a los árboles de búsqueda para problemas de diagnóstico, siempre y cuando el tamaño del universo de objetos y lugares no sea mucho mayor al presentado en esta prueba.

Dado el interés de grupo Golem para explorar los procesos cognitivos en agentes robóticos es razonable recomendar el uso de *Answer Set Programming* para enmarcar soluciones a problemas en un paradigma diferente al de Prolog con la seguridad de que si se trata de un dominio pequeño, la tarea podrá ser ejecutada.

Apéndice 1: Código en Prolog

Módulo de inferencia original (creado por Grupo Golem):

```
inference_module:-
      open_kb(KB),
      diagnostician(KB,Diagnostic,NewKB),
      save_kb(NewKB),
      decision_maker(NewKB,Decision,Result),
      dfs_planner(Result,500,NewKB,Plan),
      ddp_generate_explication_dialog(Diagnostic,Decision,Plan,Dialog)
      ,
      assign_func_value(solve_plan(Plan,Dialog)).
```

Módulo de inferencia para *Answer Set Programming* (se hizo también un cambio de función para salvar la base de datos para que esta pudiera seguir funcionando con el resto de las funciones de Golem una vez transformada por el submódulo de diagnóstico).

```
Inference_module:-
      open_kb(KB),
      etr_dlv_link(KB,Diagnostic,NewKB),
      etr_save_kb(NewKB),
      decision_maker_ASP(Decision),
      dfs_planner_ASP(Plan),
      ddp_generate_explication_dialog(Diagnostic,Decision,Plan,Dialog)
      ,
      assign_func_value(solve_plan(Plan,Dialog)).
```

ETR_P_User_functions.pl

Se llama a la inclusión de las funciones de otros archivos para poder hacer uso de todas durante la ejecución del submódulo de diagnóstico del módulo de inferencia en *Answer Set Programming*.

```
:-
include('apps/robocup_2015/ricardo/ETR_P_Preparation_For_Diagnosis.pl
').
:- include('apps/robocup_2015/ricardo/ETR_P_Transform_DB_To_ASP.pl').
:- include('apps/robocup_2015/ricardo/ETR_P_Diagnosis.pl').
:- include('apps/robocup_2015/ricardo/ETR_P_Return.pl').
```

Este código no recibe ninguna entrada y regresa una lista de acciones que describen el proceso mediante el cual los objetos en escena se movieron del almacén a las repisas que ahora ocupan, también regresa la versión de la base de conocimientos modificada de forma que se puedan guardar los cambios para ser usados por los subsecuentes partes del módulo de inferencia.

Se usa esta función para llamar a las funciones que generan el diagnóstico y alteran la base de conocimiento para reflejar la inferencia y observaciones hechas:

```
etr_dlv_link(KB,DiagnosisList,New_Database):-
      etr_ProgresivePreparation(KB),
      etr_diagnosis(DiagnosisList,New_Database),
      !.
```

Estas funciones auxiliares se usan para la recuperación de las respuestas de DLV, se incluye en este archivo ya que es usado por funciones en varios archivos.

```
etr_read_lines(Out, Lines) :-
    read_line_to_codes(Out, Line1),
    etr_read_lines(Line1, Out, Lines),
    !.
etr_read_lines(end_of_file, _, []).
etr_read_lines(Codes, Out, [Line|Lines]) :-
    atom_codes(Line, Codes),
    read_line_to_codes(Out, Line2),
    etr_read_lines(Line2, Out, Lines),
    !.
```

ETR_P_Preparation_For_Diagnosis.pl

Este código se utiliza para preparar la información contenida en la base de conocimiento de Golem y transformarla a un formato que se pueda manejar en *Answer Set Programming*. Recibe la base de conocimientos y no regresa nada aunque si crea archivos que después serán consultados por otros procesos.

Esta función primero llama a transformar la base de conocimientos del formato actual al que puede usar DLV, después, se llama a DLV (usando la función process_create) con el archivo `ETR_A_Preparation.dl` y los datos que se colocaron en `ETR_A_Prolog_To_ASP.dl`y este regresa los predicados necesarios para que funcione el código que se encarga del diagnóstico. Los datos se guardan en el archivo `ETR_A_Prolog_To_ASP.dl`

```
etr_ProgresivePreparation(KB):-
      etr_DataBase_To_ASP(KB),
      process_create('apps/robocup_2015/ricardo/dlv.i386-linux-elf-sta
      tic.bin', ['-silent', '-n=1',
      '-pfilter=etr_object,etr_location,etr_at_start,etr_at_final,etr_
      original,etr_belief,etr_new_observation,etr_old_observation,etr_
      grasped', 'apps/robocup_2015/ricardo/ETR_A_Preparation.dl',
      'apps/robocup_2015/ricardo/ETR_A_Prolog_To_ASP.dl'],
      [stdout(pipe(B))]),
      etr_read_lines(B, Output),
      Output =[H|_],
      string_concat('{' , Result1, H),
      string_concat(Result2, '}', Result1),
      atomic_list_concat(Atoms, ', ', Result2),
      open('apps/robocup_2015/ricardo/ETR_A_Diagnosis_World.dl',write,
      Stream),
      etr_write_world_to_file(Atoms,Stream),
      close(Stream),
      !.
```

Esta función transforma la base de conocimientos poniendo un punto y guardando el resultado en un archivo y luego leyendo el archivo y al final manda a llamar al proceso que transforma la base de conocimientos de la forma de listas que tiene en Golem a la forma de predicados atómicos que se usa en DLV. el resultado lo guarda en el archivo `ETR_A_Prolog_To_ASP.dl`

```
etr_DataBase_To_ASP(KB):-
      open('knowledge_base/golem_KB.txt',read,Stream),
      read_stream_to_codes(Stream, Querycodes),
      close(Stream),
      append(Querycodes,[46],Data),
      atom_codes(Database,Data),
```

```
      open('apps/robocup_2015/ricardo/golem_KB_helper.txt',write,Strea
      m2),
      write(Stream2,Database),
      close(Stream2),
      open('apps/robocup_2015/ricardo/golem_KB_helper.txt',read,Stream
      3),
      read(Stream3,Database2),
      close(Stream3),
      open('apps/robocup_2015/ricardo/ETR_A_Prolog_To_ASP.dl',write,St
      ream4),
      etr_create_file_for_aspdlv(Database2,Stream4),
      close(Stream4),
      !.
```

Esta función toma una lista y la escribe a un archivo.

```
etr_write_world_to_file([],_).
etr_write_world_to_file([H|T],Stream):-
      write(Stream,H),write(Stream,'.\n'),
      etr_write_world_to_file(T,Stream),
      !.
```

ETR_Transform_BD_To_ASP.pl

Este código transforma la base de conocimientos de Golem de una formato de listas anidadas a un formato de predicados atómicos para que pueda ser interpretada y manejada por DLV. Recibe una lista de listas que contienen la base de conocimiento de golem y genera un archivo (ya que DLV recibe los predicados vía archivo) con los predicados atómicos que representa el mismo conocimiento que la base original.

Este código fue originalmente desarrollado por Ricardo Adolfo Fierro Villaneda y fue adaptado para su uso por Eduardo Tello Ramos

Se define el operador "=>" que es usado en la base de conocimientos.

```
:- op(800,xfx,'=>').
```

Esta función transforma las clases en predicados que pueden ser interpretados por DLV, en caso de ser listas (como la de propiedades, relaciones o instancias) se llama a otras funciones para que hagan lo correspondiente. Para las clases relacionadas con las acciones del robot se generan predicados del tipo "special class" en los que se encapsula la información de tal forma que no afecte el desempeño de DLV y no se pierda información para que pueda ser recuperada al regresar la base de conocimientos al formato de listas anidadas al finalizar el proceso de diagnóstico.

```
etr_create_file_for_aspdlv([],_).
etr_create_file_for_aspdlv([class(X,actions,P,R,I)|Z],Stream):-
      write(Stream,'parent("'), write(Stream,X), write(Stream,'",'),
      write(Stream,actions), write(Stream,').'), nl(Stream),
      write(Stream,'special_class('), write(Stream,X),
      write(Stream,',actions,"'), write(Stream,P),
      write(Stream,'","'), write(Stream,R), write(Stream,'","'),
      write(Stream,I), write(Stream,'").'), nl(Stream),
      etr_create_file_for_aspdlv(Z,Stream),
      !.
etr_create_file_for_aspdlv([class(X,Y,P,R,I)|Z],Stream):-
      write(Stream,'class("'), write(Stream,X), write(Stream,'").'),
      nl(Stream),
      write(Stream,'parent("'), write(Stream,X), write(Stream,'",'),
      write(Stream,Y), write(Stream,').'), nl(Stream),
      etr_write_properties(X,P,Stream),
      etr_write_relationships(X,R,Stream),
      etr_write_instances(X,I,Stream),
      nl(Stream),
      etr_create_file_for_aspdlv(Z,Stream),
      !.
```

Escribe las propiedades de una clase a un archivo en un formato de predicados atómicos que DLV puede manejar.

```
etr_write_properties(_,[],_).
```

Se tratan las propiedades negadas haciéndolas un predicado negativo.

```
etr_write_properties(X,[not(H)|T],Stream):-
      etr_transform_to_string(H,Z),
      etr_name_value(Z, H1,H2),
      write(Stream,'-property("'), write(Stream,X),
      write(Stream,'",'), write(Stream,H1), write(Stream,','),
      write(Stream,H2), write(Stream,').'), nl(Stream),
      etr_write_properties(X,T,Stream),
      !.
etr_write_properties(X,[not(H)|T],Stream):-
      write(Stream,'-property("'), write(Stream,X),
      write(Stream,'",'), write(Stream,H), write(Stream,',null).'),
      nl(Stream),
      etr_write_properties(X,T,Stream),
      !.
```

Se tratan las propiedades transformandolas en predicados

```
etr_write_properties(X,[(H)|T],Stream):-
      etr_transform_to_string(H,Z),
      etr_name_value(Z, H1,H2),
      write(Stream,'property("'), write(Stream,X), write(Stream,'",'),
      write(Stream,H1), write(Stream,','), write(Stream,H2),
      write(Stream,').'), nl(Stream),
      etr_write_properties(X,T,Stream),
      !.
etr_write_properties(X,[H|T],Stream):-
      write(Stream,'property("'), write(Stream,X), write(Stream,'",'),
      write(Stream,H), write(Stream,',null).'),  nl(Stream),
      etr_write_properties(X,T,Stream),
      !.
```

Escribe las relaciones de una clase a un archivo en un formato de predicados atómicos que DLV puede manejar.

```
etr_write_relationships(_,[],_).
```

Se tratan las relaciones negadas haciéndolas un predicado negativo.

```
etr_write_relationships(X,[not(H)|T],Stream):-
      etr_transform_to_string(H,Z),
      etr_name_value(Z, H1,H2),
      write(Stream,'-relation("'), write(Stream,X),
      write(Stream,'",'), write(Stream,H1), write(Stream,','),
      write(Stream,H2), write(Stream,').'), nl(Stream),
```

```
      etr_write_relationships(X,T,Stream),
      !.
etr_write_relationships(X,[not(H)|T],Stream):-
      write(Stream,'-relation("'), write(Stream,X),
      write(Stream,'",'), write(Stream,H), write(Stream,',null).'),
      nl(Stream),
      etr_write_relationships(X,T,Stream),
      !.
```

Se tratan las relaciones transformandolas en predicados

```
etr_write_relationships(X,[(H)|T],Stream):-
      etr_transform_to_string(H,Z),
      etr_name_value(Z, H1,H2),
      write(Stream,'relation("'), write(Stream,X), write(Stream,'",'),
      write(Stream,H1), write(Stream,','), write(Stream,H2),
      write(Stream,').'), nl(Stream),
      etr_write_relationships(X,T,Stream),
      !.
etr_write_relationships(X,[H|T],Stream):-
      write(Stream,'relation("'), write(Stream,X), write(Stream,'",'),
      write(Stream,H), write(Stream,',null).'),  nl(Stream),
      etr_write_relationships(X,T,Stream),
      !.
```

Escribe las instancias de una clase a un archivo en un formato de predicados atómicos que DLV puede manejar.

```
etr_write_instances(_,[],_).
```

En caso de que se tenga una sola instancia

```
etr_write_instances(X,[[H|[P,R]]],Stream):-
      etr_transform_to_string(H,Z),
      etr_name_value(Z, H1,H2),
      write(Stream,'instance("'), write(Stream,X), write(Stream,'",'),
      write(Stream,H1), write(Stream,','), write(Stream,H2),
      write(Stream,').'), nl(Stream),
      etr_write_instance_properties(H2,P,Stream),
      etr_write_instance_relationships(H2,R,Stream),
      !.
```

En caso de que se tengan varias instancias

```
etr_write_instances(X,[[H|[P,R]]|T2],Stream):-
      etr_transform_to_string(H,Z),
      etr_name_value(Z, H1,H2),
      write(Stream,'instance("'), write(Stream,X), write(Stream,'",'),
      write(Stream,H1), write(Stream,','), write(Stream,H2),
      write(Stream,').'), nl(Stream),
```

```
      etr_write_instance_properties(H2,P,Stream),
      etr_write_instance_relationships(H2,R,Stream),
      etr_write_instances(X,T2,Stream),
      !.
```

Escribe las propiedades de una instancia a un archivo en un formato de predicados atómicos que DLV puede manejar.

```
etr_write_instance_properties(_,[],_).
```

Se tratan las propiedades negadas haciéndolas un predicado negativo.

```
etr_write_instance_properties(X,[not(H)|T],Stream):-
      etr_transform_to_string(H,Z),
      etr_name_value(Z, H1,H2),
      write(Stream,-property(X,H1,H2)), write(Stream,'.'),
      nl(Stream),
      etr_write_instance_properties(X,T,Stream),
      !.
etr_write_instance_properties(X,[not(H)|T],Stream):-
      write(Stream,-property(X,H,null)), write(Stream,'.'),
nl(Stream),
      etr_write_instance_properties(X,T,Stream),
      !.
```

Se tratan las propiedades positivas haciéndolas un predicado.

```
etr_write_instance_properties(X,[H|T],Stream):-
      etr_transform_to_string(H,Z),
      etr_name_value(Z, H1,H2),
      H1 = name,
      write(Stream,'property('),  write(Stream,X),  write(Stream,','),
      write(Stream,H1), write(Stream,',"'), write(Stream,H2),
      write(Stream,'").'),  nl(Stream),
      etr_write_instance_properties(X,T,Stream),
      !.
etr_write_instance_properties(X,[H|T],Stream):-
      etr_transform_to_string(H,Z),
      etr_name_value(Z, H1,H2),
      not(H1 = name),
      write(Stream,property(X,H1,H2)), write(Stream,'.'),  nl(Stream),
      etr_write_instance_properties(X,T,Stream),
      !.
etr_write_instance_properties(X,[H|T],Stream):-
      write(Stream,property(X,H,null)), write(Stream,'.'),
      nl(Stream),
      etr_write_instance_properties(X,T,Stream),
      !.
```

Escribe las relaciones de una instancia a un archivo en un formato de predicados atómicos que DLV puede manejar.

```
etr_write_instance_relationships(_,[],_).
```

Se tratan las relaciones negadas haciéndolas un predicado negativo.

```
etr_write_instance_relationships(X,[not(H)|T],Stream):-
      etr_transform_to_string(H,Z),
      etr_name_value(Z, H1,H2),
      write(Stream,-relation(X,H1,H2)), write(Stream,'.'),
      nl(Stream),
      etr_write_instance_relationships(X,T,Stream),
      !.
```

Se tratan las relaciones positivas haciéndolas un predicado.

```
etr_write_instance_relationships(X,[H|T],Stream):-
      etr_transform_to_string(H,Z),
      etr_name_value(Z, H1,H2),
      write(Stream,relation(X,H1,H2)), write(Stream,'.'),  nl(Stream),
      etr_write_instance_relationships(X,T,Stream),
      !.
```

Transforma un átomo en una cadena de texto

```
etr_transform_to_string(Z,A):-
      with_output_to(atom(A), write(Z)),
      !.
```

Descompone las características que usan el operador “=>” en un predicado.

```
etr_name_value(String, Name, Value) :-
      sub_string(String, Before, _, After, '=>'),
      sub_string(String, 0, Before, _, NameString),
      atom_string(Name, NameString),
      sub_string(String, _, After, 0, Value),
      !.
```

ETR_P_Diagnosis.pl

Este código llama el programa en *Answer Set Programming* que genera la lista de acciones que conforma el diagnóstico (aunque aún no depuradas) y los cambios a la base de datos, mana a llamar a las funciones que depuran el diagnóstico y también manda a llamar al código en *Answer Set Programming* que incorpora los cambios a la base de conocimientos y la vuelve a acomodar en formato de listas anidadas, una vez que se han realizado estas acciones, se manda a llamar al código que depura las listas anidadas y las agrega en una sola lista para así tener una sola lista de clases con listas anidadas que es la base de conocimientos con los cambios necesarios incorporados. El código no recibe ningún dato y regresa la lista con las acciones que forma el diagnóstico y la lista que representa la base de conocimientos modificada.

Este código dirige las llamadas a DLV para obtener el diagnóstico y la base de conocimientos modificada.

1) Se llama a DLV(utilizando proces_create) con `ETR_A_Diagnosis.dl` y `ETR_A_Diagnosis_World.dl.`
2) El resultado se divide usando `etr_split` en la lista de acciones que define al diagnóstico y la lista de cambios.
3) Los cambios se escriben en el archivo `ETR_A_Changes.dl`
4) Se ordenan las acciones del diagnóstico usando `etr_bubble_sort`.
5) Se modifican las acciones del diagnóstico para que queden en el formato que acepta el parseador de acciones de la prueba de restaurante usando `etr_make_diagnosis_list`
6) Se llama a DLV (utilizando process_create) con `ETR_A_Transform_Base_01.dl`, `ETR_A_Prolog_To_ASP.dl` y `ETR_A_Changes.dl.`
7) El resultado (la base de conocimientos en formato para DLV modificada por los cambios) se guarda en el archivo `ETR_A_Auxiliary_Base.dl`
8) Se llama a DLV (utilizando process_create) con `ETR_A_Transform_Base_02.dl` y `ETR_A_Auxiliary_Base.dl`
9) El resultado (la base de conocimientos en formato para DLV modificada y arreglada) se guarda en el archivo `ETR_A_Return_Base.dl`
10) Se llama a DLV (utilizando process_create) con `ETR_A_Return.dl` y `ETR_A_Return_Base.dl`
11) El resultado se procesa con `etr_format_data_base` para quitar información adicional y quedarse con la base de conocimientos en el formato que se usa en Golem.

```
etr_diagnosis(DiagnosisList,New_Database):-
      process_create('apps/robocup_2015/ricardo/dlv.i386-linux-elf-sta
      tic.bin', ['-silent', '-n=1',
      '-pfilter=etr_diagnosis,etr_change_of_place',
      'apps/robocup_2015/ricardo/ETR_A_Diagnosis.dl',
      'apps/robocup_2015/ricardo/ETR_A_Diagnosis_World.dl'],
      [stdout(pipe(B))]),
      etr_read_lines(B, Output),
```

```
Output =[H|_],
string_concat('Best model: {' , Result1, H),
string_concat(Result2, '}', Result1),
atomic_list_concat(Diagnosis_Changes, ', ', Result2),
etr_split(Diagnosis_Changes,Diagnosis,Changes),
open('apps/robocup_2015/ricardo/ETR_A_Changes.dl',write,Stream),
etr_write_changes(Changes,Stream),
close(Stream),
etr_bubble_sort(Diagnosis, SortedDiagnosis),
etr_make_diagnosis_list(SortedDiagnosis, DiagnosisList),
process_create('apps/robocup_2015/ricardo/dlv.i386-linux-elf-sta
tic.bin', ['-silent', '-n=1',
'-filter=class,parent,properta,-properta,instance,special_class'
, 'apps/robocup_2015/ricardo/ETR_A_Transform_Base_01.dl',
'apps/robocup_2015/ricardo/ETR_A_Prolog_To_ASP.dl',
'apps/robocup_2015/ricardo/ETR_A_Changes.dl'],
[stdout(pipe(C))]),
etr_read_lines(C, Output2),
Output2 =[H2|_],
string_concat('{' , Result3, H2),
string_concat(Result4, '}', Result3),
atomic_list_concat(Atoms, ', ', Result4),
open('apps/robocup_2015/ricardo/ETR_A_Auxiliary_Base.dl',write,S
tream2),
etr_write_aux_base(Atoms,Stream2),
close(Stream2),
process_create('apps/robocup_2015/ricardo/dlv.i386-linux-elf-sta
tic.bin', ['-silent', '-n=1',
'-filter=class,parent,property,-property,instance,special_class'
, 'apps/robocup_2015/ricardo/ETR_A_Transform_Base_02.dl',
'apps/robocup_2015/ricardo/ETR_A_Auxiliary_Base.dl'],
[stdout(pipe(D))]),
etr_read_lines(D, Output3),
Output3 =[H3|_],
string_concat('{' , Result5, H3),
string_concat(Result6, '}', Result5),
atomic_list_concat(Atoms2, ', ', Result6),
open('apps/robocup_2015/ricardo/ETR_A_Return_Base.dl',write,Stre
am3),
etr_write_base_for_return(Atoms2,Stream3),
close(Stream3),
process_create('apps/robocup_2015/ricardo/dlv.i386-linux-elf-sta
tic.bin', ['-silent', '-n=1', '-nofinitecheck',
```

```
        '-filter=etr_class',
        'apps/robocup_2015/ricardo/ETR_A_Return.dl',
        'apps/robocup_2015/ricardo/ETR_A_Return_Base.dl'],
        [stdout(pipe(E))]),
        etr_read_lines(E, Output4),
        Output4 =[H4|_],
        etr_format_data_base(H4,New_Database),
        !.
```

Esta función escribe una lista a un archivo línea por línea añadiendo un punto al final de cada línea.

```
etr_write_base_for_return([],_).
etr_write_base_for_return([H|T],Stream):-
        string_concat(H,'.',H3),
        write(Stream,H3),nl(Stream),
        etr_write_base_for_return(T,Stream),
        !.
```

Esta función escribe una lista a un archivo línea por línea añadiendo un punto al final de cada línea.

```
etr_write_aux_base([],_).
etr_write_aux_base([H|T],Stream):-
        string_concat(H,'.',H3),
        write(Stream,H3),nl(Stream),
        etr_write_aux_base(T,Stream),
        !.
```

Esta función escribe una lista a un archivo línea por línea añadiendo un punto al final de cada línea.

```
etr_write_changes([],_).
etr_write_changes([H|T],Stream):-
        write(Stream,H),write(Stream,'.'),nl(Stream),
        etr_write_changes(T,Stream),
        !.
```

Esta función divide la respuesta del submódulo de diagnóstico en dos listas, una que contiene el diagnóstico y otras con los cambios a la base de conocimientos.

```
etr_split([],[],[]).
etr_split([H|T],Diagnosis,Changes):-
        etr_split(T,DiagnosisAux,ChangesAux),
        term_to_atom(H1,H),
        (H1=etr_diagnosis(_,_,_) ->
                Diagnosis = [H1|DiagnosisAux],
                Changes = ChangesAux,
```

```
            !
      ;
            Diagnosis = DiagnosisAux,
            Changes=[H1|ChangesAux],
            !
      ),
      !.
```

Esta función transforma el diagnóstico generado por el submódulo de diagnóstico a un formato que pueda ser reconocido por el parser de la prueba de supermercado de Golem.

```
etr_make_diagnosis_list([],[]).
etr_make_diagnosis_list([H|T], List):-
      H = etr_diagnosis(_, A, P),
      (
            A=goto->
            X=move(P),
            !
      ;
            A=take->
            X=take(P),
            !
      ;
            A=deliver->
            X=deliver(P),
            !
      ;
            !
      ),
      etr_make_diagnosis_list(T, AuxiliaryList),
      List = [X|AuxiliaryList],
      !.
```

Ordena la lista de acciones que conforman el diagnóstico en orden ascendente por la primer característica del predicado etr_diagnosis (que es el tiempo en el que se ejecutó) utilizando el algoritmo bubble sort.

```
etr_bubble_sort(List, Sorted):-
      etr_b_sort(List, [], Sorted),
      !.
etr_b_sort([], Acc, Acc).
etr_b_sort([H|T], Acc, Sorted):-
      etr_bubble(H, T, NT, Max),
      etr_b_sort(NT, [Max|Acc], Sorted),
      !.
```

```
etr_bubble(X, [], [], X).
etr_bubble(X, [Y|T], [Y|NT], Max):-
      X = etr_diagnosis(TX, _, _),
      Y = etr_diagnosis(TY, _, _),
      TX > TY,
      etr_bubble(X, T, NT, Max),
      !.
etr_bubble(X, [Y|T], [X|NT], Max):-
      X = etr_diagnosis(TX, _, _),
      Y = etr_diagnosis(TY, _, _),
      TX =< TY,
      etr_bubble(Y, T, NT, Max),
      !.
```

ETR_P_Return.pl

Este código modifica la lista de clases que regresa DLV con las modificaciones derivadas del diagnóstico para que quede en el formato que Golem utiliza. Recibe un texto que representa la lista de clases y regresa la lista con las clases ya en el formato aceptado por Golem.

Se define el operador utilizado en la base de conocimientos de Golem.

```
:- op(800,xfx,=>).
```

Esta función es solo el punto de entrada y transforma el texto en una lista ya propiamente.

```
etr_format_data_base(Text,Database_With_Format):-
    string_concat('{' , Result1, Text),
    string_concat(Result2, '}', Result1),
    atomic_list_concat(Base, ', ', Result2),
    etr_list_base(Base,Database_With_Format),
    !.
```

Esta función analiza cada miembro de la lista de clases y manda a llamar las funciones que modifican las diferentes partes de la clase (propiedades, relaciones e instancias), al final vuelve a empaquetar la información en una clase y la concatena en una lista que devuelve.

```
etr_list_base([],[]).
etr_list_base([H|T],L):-
    etr_fix_text(H,H1),
    H1=etr_class(A,B,C,D,E),
    etr_protect_quotes(A,A2),
    etr_list_class_properties(C,C2),
    etr_list_class_relationships(D,D2),
    etr_list_class_instances(E,E2),
    P=class(A2,B,C2,D2,E2),
    etr_list_base(T,L2),
    L= [P|L2],
    !.
```

Mantiene las comillas en los átomos que tienen más de una palabra

```
etr_protect_quotes(A,A3):-
    atom_codes(A,L),
    (
        member(32,L)->
        string_concat('\'',A,A2),
        string_concat(A2,'\'',A3),
    !
    ;
        A3 = A,
```

```
    !
    ),
    !.
```

Quita comillas dobles de algunos átomos

```
etr_remove_quotes(A,B):-
    etr_fix_text(A,A2),
    atom_codes(A2,L),
    (
        member(32,L)->
        string_concat('"',A1,A2),
        string_concat(A0,'"',A1),
        string_concat('"\'',A0,A3),
        string_concat(A3,'\'"',A4),
        B=A4,
        !
    ;
        B = A,
        !
    ),
    !.
```

Cambia el formato del texto para que pueda ser trabajado cuando tiene comillas.

```
etr_fix_text(A,B):-
        (
            atom(A)->

            atom_codes(A,A1),
            etr_change_quotes(A1,A2),
            atom_codes(A3,A2),
            term_to_atom(B,A3),
            !
        ;
            term_to_atom(A,A0),
            atom_codes(A0,A1),
            etr_change_quotes(A1,A2),
            atom_codes(A3,A2),
            term_to_atom(B,A3),
            !
        ),
        !.
```

Quita comillas de átomos que tienen más de una palabra

```
etr_fix_text2(A,B):-
        atom_codes(A,A1),
        etr_change_quotes(A1,A2),
        atom_codes(A3,A2),
        (
            member(32,A1)->
            string_concat('\'',A3,A4),
            string_concat(A4,'\'',A5),
            B=A5,
            !
        ;
            term_to_atom(B,A3),
            !
        ),
        !.
```

Cambia las comillas de simples a dobles

```
etr_change_quotes([],[]).
etr_change_quotes([H|T],P):-
    ( H=34->
        S=39,
        !
    ;
        S=H,
        !
    ),
    etr_change_quotes(T,Paux),
  P=[S|Paux],
  !.
```

Lista las propiedades de una clase, haciendo las modificaciones necesarias en los casos que lo necesiten (como propiedades negadas por ejemplo).

```
etr_list_class_properties([],[]).
etr_list_class_properties([H|T],P):-
    (
        H=etr_class_property(_,B,C) ->
        etr_fix_text(B,B2),
        etr_fix_text(C,C2),
        (
            C=null->
            etr_list_class_properties(T,PA),
            P=[B|PA],
            !
```

```
        ;
            C=no->
            etr_list_class_properties(T,PA),
            P=[not(B)|PA],
            !
        ;
            string_concat(B2, =>, M),
            string_concat(M, C2, N),
            etr_list_class_properties(T,PA),
            P=[N|PA],
            !
        ),
        !
    ;
        N=H,
        etr_list_class_properties(T,PA),
        P=[N|PA],
        !
    ),
    !.
etr_list_class_properties(X,P):-
    string_concat('[' , Result1, X),
    string_concat(Result2, ']', Result1),
    atomic_list_concat(Props, ', ', Result2),
    etr_list_class_properties(Props,P),
    !.
```

Lista las relaciones de una clase,

```
etr_list_class_relationships([],[]).
etr_list_class_relationships([H|T],P):-
    (
        H=etr_class_relationship(_,B,C) ->
        etr_list_class_relationships(T,PA),
        P=[[B,C]|PA],
        !
    ;
        etr_list_class_relationships(T,PA),
        P=[[H]|PA],
        !
    ),
    !.
etr_list_class_relationships(X,P):-
    string_concat('[' , Result1, X),
```

```
    string_concat(Result2, ']', Result1),
    atomic_list_concat(Props, ', ', Result2),
    (
        Props = [''] ->
        Props2 = [],
        !
    ;
        Props2 = Props,
        !
    ),
    etr_list_class_relationships(Props2,P),
    !.
```

Lista las instancias de una clase, manda a llamar a las funciones que listan las propiedades y relaciones de una instancia.

```
etr_list_class_instances([],[]).
etr_list_class_instances([H|T],P):-
    (
        H=etr_class_instance(_,B,C,D) ->
        B=etr_operator(_,Y,Z),
        (
            atom(Z)->
            etr_fix_text(Y,Y2),
            etr_fix_text(Z,Z2),
            string_concat(Y2, =>, M),
            string_concat(M, Z2, B2),
            etr_list_class_instances(T,PA),
            C= etr_final_instance_properties_list(_,_,C2),
            etr_list_instance_properties(C2,C3),
            D= etr_final_instance_relationship_list(_,_,D2),
            etr_list_instance_relationships(D2,D3),
            P=[[B2,C3,D3]|PA],
            !
        ;
            string_concat(Y, =>, M),
            term_to_atom(Z,Z3),
            string_concat(M, Z3, B2),
            etr_list_class_instances(T,PA),
            C= etr_final_instance_properties_list(_,_,C2),
            etr_list_instance_properties(C2,C3),
            D= etr_final_instance_relationship_list(_,_,D2),
            etr_list_instance_relationships(D2,D3),
            P=[[B2,C3,D3]|PA],
```

```
            !
        ),
        !
    ;
        etr_list_class_instances(T,PA),
        P=[H|PA],
        !
    ),
    !.
etr_list_class_instances(X,P):-
    string_concat('[' , Result1, X),
    string_concat(Result2, ']', Result1),
    atomic_list_concat(Props, ', ', Result2),
    etr_list_class_instances(Props,P),
    !.
```

Lista las propiedades de una instancia

```
etr_list_instance_properties([],[]).
etr_list_instance_properties([H|T],P):-
    H=etr_instance_property(_,Y,Z),
    (

        Z=[],not(Y=observed_objects)->
        B2=[],
        !
    ;
        Z=[_|_],not(Y=content),not(Y=observed_objects)->
        B2=Z,
        !
    ;
        Z=null,not(Y=observed_objects)->
        B2=Y,
        !
    ;
        Y=content->
        etr_list_transform_auxlilar(Z,Z2),
        string_concat(Y, =>, M),
        string_concat(M, Z2, B3),
        string_concat(B2, '.', B3),
        !
    ;
        Y=observed_objects->
        etr_list_transform_auxlilar(Z,Z2),
```

```
        string_concat(Y, =>, M),
        string_concat(M, Z2, B3),
        string_concat(B2, '.', B3),
        !
    ;
        etr_fix_text(Y,Y2),
        etr_fix_text2(Z,Z2),
        (
            string_concat('"',J,Z2)->
            string_concat(S,'"',J),
            string_concat('"\'',S,S2),
            string_concat(S2,'\'"',S3),
            !
        ;
            S3 = Z2,
            !
        ),
        (
            S3="'start'"->
            string_concat(Y2, '=>start', B2),
            !
        ;
            string_concat(Y2, =>, M),
            string_concat(M, S3, B2),
            !
        ),
        !
    ),
    etr_list_instance_properties(T,PA),
    P=[B2|PA],
    !.
```

Salva una lista a un archivo y la lee para que entonces el formato de la lista sea el correcto para seguirla trabajando.

```
etr_list_transform_auxlilar(Z,Z2):-
        open('ETR_Trasform_List.txt',write,Stream),
        write(Stream,'['),
        etr_write_list_auxiliar(Z,Stream),
        write(Stream,'].'),
        close(Stream),
        open('ETR_Trasform_List.txt.txt',read,Stream2),
        read_stream_to_codes(Stream2, Querycodes),
        close(Stream2),
```

```
        atom_codes(Z2,Querycodes),
        !.
```

Escribe una lista a un archivo en una sola línea

```
etr_write_list_auxiliar([],_).
etr_write_list_auxiliar([H|T],Stream):-
    write(Stream,H),
    (
        T=[]->
        !
    ;
        write(Stream,','),
        !
    ),
    etr_write_list_auxiliar(T,Stream),
    !.
```

Lista las relaciones de una instancia

```
etr_list_instance_relationships([],[]).
etr_list_instance_relationships([H|T],P):-
    H=etr_instance_relationship(_,Y,Z),
    etr_fix_text(Y,Y2),
    etr_fix_text(Z,Z2),
    string_concat(Y2, =>, M),
    string_concat(M, Z2, B2),
    etr_list_instance_relationships(T,PA),
    P=[B2|PA],
    !.
```

Apéndice 2: Código en *Answer Set Programming*

El código en *Answer Set Programming* sigue la filosofía de diseño de mantener los datos y el código en archivos separados para que así se pueda modificar solo lo indispensable durante la ejecución del algoritmo de diagnóstico. Por esta razón, aunque tanto los datos como el código en si son en realidad predicados de *Answer Set Programming* y podrían estar en el mismo archivo se mantienen en archivos separados.

Para evitar mencionarlo para cada archivo de código a continuación se presenta la estructura general de los archivos con código en *Answer Set Programming*:

Bloque de inicio:

En el bloque de inicia se tiene un muy breve descripción de la funcionalidad del código del archivo en cuestión, así como el autor del mismo.

La llamada a shell es el código necesario para correr el código desde una ventana de comando situada en la carpeta en la que se encuentra el archivo con el código. (es necesario que el motor de *Answer Set Programming* DLV se encuentre en la misma carpeta).

En la llamada se encuentra una parte fundamental que el el filtro "`-pfilter=`" que permite recibir solamente los predicados listados después del igual que sean positivos esto es; que si se tiene un predicado `predicado()` y se tiene el predicado negativo `-predicado()`, DLV solamente devuelve la versión positiva del predicado.

Es importante entonces recalcar que es posible tener soluciones diferentes que son reportadas como igual ya que la diferencia yace en predicados que no son devueltos por DLV.
Al final se indican los archivos extra con los que es necesario contar para que el código funcione.

Código:

El código se organiza en bloques ya sea por función (todos los predicados dentro del bloque se utilizan para generar un predicado en particular) o por similitud (todos los predicados del bloque generan el mismo predicado con alguna variación en el proceso de generación).

El código se presenta con comentarios que explican brevemente el funcionamiento de cada bloque.

A continuación se presentan las partes del código de *Answer Set Programming* del sub módulo de diagnóstico del módulo de inferencia.

El código difiere del original en formato (se le ha dado un formato distinto para facilitar su lectura en este documento) y se han omitido los comentarios que se encuentran insertados en el código para evitar ambigüedad con respecto a las explicaciones del mismo provistas en el presente capítulo.
Los datos que este código utiliza se encuentran el archivo ETR_Prologo_To_ASP.dl

ETR_A_Preparation.dl

Este código regresa los siguientes predicados:
`etr_object,etr_location,etr_at_start,etr_at_final,etr_original,etr_belief,etr_new_observation,etr_old_observation,etr_grasped`

El siguiente código funciona como un filtro de manera que la información contenida en la base de conocimientos es depurada para que sea más fácil utilizarla para generar el diagnóstico.

Por ejemplo:

La regla `etr_object(O):- property(_,brand,O)` genera predicados de la forma `etr_object(O)`, donde O es un objeto de la base de conocimiento. La regla funciona al generar un predicado diferente para cada predicado de la forma `property(_,brand,O)` (que es la forma en la que se tradujeron las propiedades "brand" de la base de conocimiento que originalmente eran parte de una clase). Así, la clase que en la base de conocimiento original era:

```
class(cereal,food,[inv=>0],[],[[id=>c1,[brand=>kellogs,original_id=>c1,in=>shelf2],[]]])
```

Fue descompuesta en:

```
class("cereal").
parent("cereal",food).
property("cereal",inv,0).
instance("cereal",id,c1).
property(c1,brand,kellogs).
property(c1,original_id,c1).
property(c1,in,shelf2).
```

(Esta transformación ocurre en Prolog usando el código del archivo ETR_P_Transform_DB_To_ASP.pl)

De estas propiedades la que coincide con la regla es `property(c1,brand,kellogs)`, por lo que la regla `etr_object(O):- property(_,brand,O)` se aterriza a `etr_object(kellogs):- property(_,brand,kellogs)` (notar que no nos interesa el contenido de la primer parte del predicado property()).

Así, esta regla termina generando la siguiente lista de "objetos" que serán usados en la parte del diagnóstico del submódulo de diagnóstico del módulo de inferencia:

```
etr_object(kellogs).
etr_object(noodles).
etr_object(coke).
```

```
etr_object(heineken).
etr_object(bisquits).
```

Notar que dentro del código hay reglas que funcionan de manera diferente, por ejemplo la regla

```
etr_at_start(human,start).
```

Simplemente crea el predicado pues siempre se considera cierta (es un hecho).

Hay otro tipo de reglas en las que varios predicados deben ser ciertos al mismo tiempo para que la regla sea cierta y se active, incluidos predicados a los que son dependientes (esto es predicados que no están entre los hechos originales y son derivados "al mismo tiempo" que los que los ocupan aunque en realidad tengan que ser derivados antes para poder usar el resultado por los predicados que dependen de ellos). por ejemplo:

La regla `etr_new_observation(O,L):-`
`property(observation(L),observed_objects,X), etr_object(O),`
`#member(O,X), etr_place(L).`

Se usa para crear predicados de la forma `etr_new_observation(O,L)` donde O es un objeto y L es la locación actual de Golem. Esto se logra tomando los objetos en la escena en los predicados creados "anteriormente" `etr_object(O)`, y la locación actual de Golem `etr_place(L)`, y aterrizando en las propiedades de la base de datos modificada de la forma `property(observation(L),observed_objects,X)`, con X una lista en la que buscamos usando el predicado de DLV `#memeber(O,X)` que regresa verdadero ne caso de que O sea uno de los coponente de la lista X para ver si entre los objetos que se observaron en la locación en la que Golem está actualmente está el objeto O. así se generan los siguientes predicados:

```
etr_new_observation(coke,shelf3).
etr_new_observation(bisquits,shelf3).
```

De esta manera esta regla nos regresa aquellos objetos que fueron observados en el lugar donde Golem se encuentra actualmente (esto es los objetos que fueron observados en la última observación hecha).

A continuación se presenta el código de este programa:

Objetos presentes en la escena

```
etr_object(O):-
    property(_,brand,O).
```

Lugares presentes en la escena

```
etr_location(L):-
```

```
      instance("point",_,L).
```

Posiciones de inicio y término del asistente

```
etr_at_start(human,start).
etr_at_final(human,start).
```

Posición original de un objeto (todos empiezan en el almacén)

```
etr_original(O,O,storage):-
      etr_object(O).
```

Creencia sobre la actual posición de un objeto

```
etr_belief(O,L):-
      property(K,brand,O), property(K,in,L).
```

Observaciones nuevas

```
etr_new_observation(O,L):-
      property(observation(L),observed_objects,X),  etr_object(O),
      #member(O,X), etr_place(L).
```

Observaciones pasadas

```
etr_old_observation(O,L):-
      property(observation(L),observed_objects,X),  etr_object(O),
      #member(O,X), not etr_place(L).
```

Notar el uso de la negación débil que interpretamos como "no es posible encontrar un predicado de la forma `etr_place(L)`" por lo que el predicado busca las observaciones que se hayan hecho en lugares diferentes al que golem ocupa actualmente.

Locación actual e Golem

```
etr_place(L):-
      property(golem,position,L).
```

Objetos anteriormente tomados por Golem

```
etr_grasped(O):-
      instance("grasp object",id,grasp(O)).
```

De esta manera, el código anterior genera una serie de predicados que serán usados en la parte de diagnóstico, para así reducir la cantidad de predicados que se revisan durante el proceso de diagnóstico y esto resulta en un mejora de desempeño del algoritmo. La lista de predicados que genera es la siguiente:

Puntos de partida y terminación del asistente

```
etr_at_start(human,start).
```

```
etr_at_final(human,start).
```

Objetos en escena

```
etr_object(kellogs).
etr_object(noodles).
etr_object(coke).
etr_object(heineken).
etr_object(bisquits).
```

Locaciones de la escena

```
etr_location(start).
etr_location(shelf2).
etr_location(shelf1).
etr_location(shelf3).
```

Lugar donde los objetos se asume estaban originalmente

```
etr_original(0,kellogs,storage).
etr_original(0,noodles,storage).
etr_original(0,coke,storage).
etr_original(0,heineken,storage).
etr_original(0,bisquits,storage).
```

Creencia actual sobre la localización actual de los objetos

```
etr_belief(kellogs,shelf2).
etr_belief(noodles,shelf2).
etr_belief(coke,shelf3).
etr_belief(heineken,shelf1).
etr_belief(bisquits,shelf3).
```

Las últimas observaciones que ha realizado Golem

```
etr_new_observation(coke,shelf3).
etr_new_observation(bisquits,shelf3).
```

Observaciones Pasadas que ha hecho golem

```
etr_old_observation(kellogs,shelf2).
etr_old_observation(noodles,shelf1).
etr_old_observation(heineken,shelf1).
```

Los objetos que han sido tomados (y por ende movidos) por Golem.

```
etr_grasped(noodles).
etr_grasped(coke).
```

ETR_A_Diagnosis.dl

Este código regresa los siguientes predicados:
`etr_diagnosis,etr_change_of_place`

A continuación se presenta el código utilizado para generar el diagnóstico en sí (al igual que los cambios a la base de conocimiento derivados de la inferencia realizada por este módulo), este código utiliza la salida del código arriba revisado.

El propósito de este código es generar una explicación posible sobre cómo fue que el asistente colocó los objetos en el escenario basándose en los objetos reportados por el asistente al principio del ejercicio. Ya que el asistente no se rige por las mismas reglas que el robot, no se toman en cuenta las probabilidades de realizar las acciones ni los costos o recompensas. Lo que sí es parte del paradigma es que el asistente puede mover a lo más 2 objetos por vez (por tener 2 manos) y tiene que regresar al almacén para recoger los objetos que va a depositar.

Este bloque provee una breve explicación del propósito del código, al autor y la llamada al shell para ejecutar el código en una carpeta que contenga el motor de DLV, este código requiere del archivo `ETR_A_Diagnosis_World.dl` que contiene la salida del código arriba revisado.

El código se presenta sin comentarios para evitar ambigüedad.

Answer Set Programming requiere que se le provea de un número máximo para aquellos predicados que emplean utilizan funciones aritméticas, de esta manera se detiene una vez que ha alcanzado ese número pues sin esta restricción se podría describir un predicado que lo hiciera trabajar por un número infinito de pasos. Aunque este número puede ser recibido como un argumento en la línea de comandos, se define aquí ya que se utiliza un esquema en el que los números mayores al número parejas posibles (ver explicación abajo) no se explora.

```
#maxint=1000.
```

Usamos estos Tipos para poder reconocer que proceso fue el que nos llevó a tener la nueva creencia etr_new_belief.

```
etr_type(a).
etr_type(b).
etr_type(c).
etr_type(d).
etr_type(e).
etr_type(f).
etr_type(g).
```

El predicado etr_new_belief representa donde se espera que un objeto en particular se encuentre al final del diagnóstico, se utilizan varios predicados (de diferentes tipos) para obtener la nueva creencia de forma única.

Por ejemplo:

etr_new_belief(X,a,O,L) es la nueva creencia default, si un objeto O fue observado en una locación L entonces decimos que para el tiempo X (del número de pasos requeridos para colocar todos los objetos en sus respectivos lugares) el objeto O se encuentra en L.
El predicado etr_number_of_steps se deriva más abajo en el código.

Otro ejemplo es el predicado **etr_new_belief(X,c,O,L1) v -etr_new_belief(X,c,O,L1);** este se usa para la situación en la que se tiene la creencia de que un objeto O está en un lugar L0 pero no fue observado en ese lugar (aunque si hay observaciones de otros objetos en ese lugar), así se propone un nuevo lugar L1 que está dentro de los lugares donde ese objeto puede estar (del predicado etr_place_where_it_can_be que se calcula más abajo revisando en qué lugares ya ha habido observaciones y regresando el complemento del conjunto de lugares posibles) siempre y cuando este objeto no haya sido tomado anteriormente por golem (pues el haberlo movido indicaría que ya fue llevado a su lugar aunque no haya sido observado ahí). Este predicado genera varios posibles lugares en los que se infiere la localización del objeto (lo cual genera varias respuestas) estas respuestas serán evaluadas y rechazadas o ponderadas utilizando otras restricciones tanto suaves como rígidas.

Es importante notar que el uso del predicado negado **-etr_new_belief(X,c,O,L1)** permite de esta manera tener varias posibilidades pues al utilizarse un lugar L1 la única otra opción que se le da al predicado es genera un negativo con el mismo lugar. Los predicados negados son eliminados por el filtro que se usa al llamar a la consulta "-pfilter" que solamente permite recibir las versiones positivas de los predicados después listados.

Predicado que presenta el lugar donde fueron observados los objetos como su lugar para la nueva creencia.

```
etr_new_belief(X,a,O,L) :-
      etr_new_observation(O,L), etr_object(O), etr_location(L),
      etr_number_of_steps(X).
```

Predicado que presenta el lugar donde se cree que están los objetos como su lugar para la nueva creencia dado que no hay observaciones que contradigan esta creencia.

```
etr_new_belief(X,b,O,L0) :-
      etr_belief(O,L0), not etr_new_observation(O,L1),
      etr_new_observation(_,L1), etr_object(O), etr_location(L0),
      etr_location(L1), L0!=L1, etr_number_of_steps(X).
```

Predicado que presenta los lugares donde aún no ha habido observaciones como posibles destinos de un objeto que no ha sido previamente observado en el lugar donde debería estar.

```
etr_new_belief(X,c,O,L1) v -etr_new_belief(X,c,O,L1):-
      etr_belief(O,L0), not etr_new_observation(O,L0),
      etr_new_observation(_,L0), etr_object(O), etr_location(L0),
      etr_location(L1), L0!=L1, etr_place_where_it_can_be(L1),
      etr_number_of_steps(X), not etr_grasped(O).
```

Predicado que presenta el lugar de la creencia original como el lugar de la nueva creencia para los objetos cuando aún no ha habido observaciones.

```
etr_new_belief(X,d,O,L) :-
      etr_belief(O,L), etr_no_observations, etr_number_of_steps(X).
```

Este predicado (que no es contemplado en el ejercicio) presenta como posibilidad que un objeto se encuentre en ninguna parte si no se le ha observado en donde debería estar y no hay más lugares posibles (pues ya se han observado todos los lugares).

```
etr_new_belief(X,e,O,nowhere) :-
      etr_belief(O,L0), not etr_new_observation(O,L0),
      etr_new_observation(_,L0), etr_object(O), etr_location(L0),
      etr_location(L1), L0!=L1, etr_no_places_exist,
      etr_number_of_steps(X).
```

Se utiliza este predicado (junto con la restricción siguiente) para generar una nueva creencia a partir de un objeto del cual no se tenga una creencia original pero si una observación por no haber sido incluido en la descripción de los objetos colocados provista por el asistente al principio del ejercicio.

```
etr_new_belief(X,f,O,L0) v -etr_new_belief(X,f,O,L0) :-
etr_new_observation(O,L0), etr_number_of_steps(X).
```

Se eliminan aquellas respuestas en las que un objeto aparezca en una nueva creencia de tipo f y en una creencia de algún otro tipo

```
      :- etr_new_belief(X,f,O,_), etr_new_belief(X,T,O,_),
etr_type(T), T!=f, etr_number_of_steps(X).
```

Se busca tener el número más pequeño de nuevas creencias negativas posibles de esta forma a cada una se le da un costo por lo que al final del proceso se regresaran solo aquellas respuestas que tengan la menor cantidad de estas (teniendo por ende la mayor cantidad de creencias positivas esto es una creencia positiva por cada objeto).

```
      :~ -etr_new_belief(_,_,_,_).[1:1]
```

Este predicado presenta el lugar original donde se pensaba esta un objeto si no ha sido recientemente observado si este fue tomado (esto es, objetos que estaban fuera de su lugar y fueron movidos por golem, no cambian de lugar pues lo que nos interesa es dónde creemos que los puso el

asistente y no donde se encuentran ahora si esta posición es el resultado de una manipulación por prate del robot)

```
etr_new_belief(X,g,O,L0):-
      etr_belief(O,L0), not etr_new_observation(O,L0),
      etr_new_observation(_,L0), etr_object(O), etr_location(L0),
      etr_location(L1), L0!=L1, etr_place_where_it_can_be(L1),
      etr_number_of_steps(X), etr_grasped(O).
```

De esta manera, tenemos los lugares donde creemos que los objetos terminaron una vez que fueron manipulados por el asistente al iniciar el ejercicio, este sistema nos permite ir cambiando el diagnóstico conforme vamos recibiendo más información sobre el mundo en las varias llamadas que se hacen a este código a lo largo del ejercicio.

Este predicado se usa para presentar el cambio de lugar que experimentó un objeto entre donde se creía originalmente que estaba y donde ahora se cree que está dadas observaciones recientes e inferencias.

```
etr_change_of_place(O,L0,L1):-
      etr_with_change(O), etr_belief(O,L0), etr_new_belief(_,_,O,L1).
```

Los siguientes predicados los utilizamos para analizar cuáles objetos han cambiado entre donde se creía que se encontraban y donde ahora se cree que están gracias a las inferencias u observaciones recientes.

Presenta objetos cuya locación en la antigua y nueva creencias es es la misma

```
etr_no_change(O):-
      etr_belief(O,L), etr_new_belief(_,_,O,L).
```

Presenta objetos cuya locación cambió de donde se creía que estaban a donde ahora se cree que están.

```
etr_with_change(O):-
      etr_belief(O,L0), etr_new_belief(_,_,O,L1), L0!=L1.
```

Se presenta el objeto que tiene distintas locaciones en las diferentes creencias y se muestran esas locaciones.

```
etr_changed(O,L0,L1):-
      etr_with_change(O), etr_belief(O,L0), etr_new_belief(_,_,O,L1).
```

Presenta dos objetos que tengan la característica en la que uno de ellos estuviera y ya no está en un lugar donde ahora otro está pero antes no estaba. Este predicado va ser útil en implementar la idea de que es deseable que el número de objetos por repisa sea lo más parecido posible entre lo reportado por el asistente al principio del ejercicio y el resultado de la inferencia; Es necesario hacerlo de esta manera pues no podemos contar las nuevas creencias ya que hay predicados con cabeza disyuntiva y los predicados que permiten contar cosas no funcionen con estos.

```
etr_match(O0,O1,L0):-
      etr_changed(O0,L0,L1), etr_changed(O1,L2,L0).
```

Presenta dos objetos que no comparten la característica de que el lugar donde uno se pensaba que estaba sea el lugar donde el otro ahora se piensa que esté y visceversa.

```
etr_no_match(O0,O1):-
      not etr_match(O0,O1,L0), etr_changed(O0,L0,L1), etr_object(O0),
      etr_object(O1), etr_location(L0), etr_location(L1), O0!=O1, not
      etr_no_change(O0), not etr_no_change(O1).
```

Este predicado ve si alguna nueva creencia se ha generado, hacemos esto para poder descartar soluciones en las que no haya nuevas creencias generadas.

```
etr_has_new_belief(T):-
      etr_new_belief(X,T,O,L), etr_object(O), etr_location(L),
      etr_type(T), etr_number_of_steps(X).
```

En caso de que ninguna nueva creencia de algún tipo se haya generado se activa este predicado.

```
etr_has_no_new_belief(T):-
      not etr_has_new_belief(T), etr_type(T).
```

Usamos este predicado para que se añada a los objetos en el storage en caso de que algún objeto que haya sido observado pero no haya sido reportado por el asistente al principio del ejercicio.

```
etr_original(0,O,storage):-
      etr_new_belief(X,f,O,_), etr_number_of_steps(X).
```

El siguiente bloque de predicados utiliza predicados aritméticos y de agregación para que se pueda saber el número de pasos que serán necesarios para trasladar todos los objetos desde la bodega hasta los lugares donde el robot ha entendido, observado o inferido que fueron depositados por el asistente.

Se cuenta cuántos objetos se tienen de inicio.

```
etr_the_original_objects(X):-
      #count{O : etr_object(O)} = X.
```

Se cuenta cuántos objetos fueron recientemente observados

```
etr_the_original_observations(X):-
      #count{O : etr_new_observation(O,_)} = X.
```

Se copia el número de objetos originales

```
etr_factor(X):-
      etr_the_original_objects(X).
```

Se copia el número de objetos observados recientemente

```
etr_factor(X):-
      etr_the_original_observations(X).
```

Se suman los objetos originales mas los observados recientemente

```
etr_addition(X):-
      #sum{O: etr_factor(O)} = X.
```

Se cuentan cuantos objetos aparecen tanto entre los objetos observados como entre los objetos originales

```
etr_repeated(X):-
      #count{O : etr_object(O), etr_new_observation(O,_)} = X.
```

Se restan los repetidos de la suma de las dos listas para tener entonces cuántos objetos hay en total

```
etr_uniques(X):-
      X=A-B, etr_addition(A), etr_repeated(B).
```

Se ve cuántas parejas se pueden formar (se obtiene solo el número entero en caso de haber decimales)

```
etr_pairings(X):-
      X=A/2, etr_uniques(A).
```

Se revisa si el número de objetos es impar

```
etr_loner(0):-
      #mod(Y,2,X), X=0, etr_uniques(Y).
etr_loner(1):-
      #mod(Y,2,X), X=1, etr_uniques(Y).
```

Se obtiene el número real de agrupaciones por par que se pueden hacer (si hay un número impar entonces hay un "par" con un solo objeto).

```
etr_real_pairings(X):-
      X=A+B, etr_pairings(A), etr_loner(B).
```

Se obtiene el número de pasos necesarios para realizar el diagnóstico.

```
etr_number_of_steps(X):-
      X=A+1, etr_real_pairings(A).
```

El siguiente bloque de predicados determina si es que hay (y cuales son) los lugares donde aún puede haber objetos sin observar

Este predicado revisa si hay algún lugar donde aún podría haber objetos sin revisar.

```
etr_places_exist:-
      etr_place_where_it_can_be(_).
```

Este predicado revisa si ya no hay más lugares sin revisar.

```
etr_no_places_exist:-
      not etr_places_exist.
```

Este predicado revisa si por lo menos hay un lugar

```
etr_a_place:-
      etr_location(L0).
```

Este predicado lista los lugares que fueron revisados recientemente

```
etr_place_where_it_can_not_be(L0):-
      etr_new_observation(_,L0).
```

Este predicado declara que el inicio no es un lugar donde pueda haber objetos sin revisar.

```
etr_place_where_it_can_not_be(start).
```

Este predicado nos da el complemento de la unión del conjunto de lugares en la escena y los lugares que fueron recientemente revisados.

```
etr_place_where_it_can_be(L):-
      not etr_place_where_it_can_not_be(L), etr_location(L).
```

Este predicado revisa si hay alguna observación del lugar en el que Golem se encuentra actualmente

```
etr_any_observation:-
      etr_new_observation(_,_).
```

Este predicado reporta si no hay ninguna nueva observación

```
etr_no_observations:-
      not etr_any_observation.
```

Este predicado revisa si un objeto está en una nueva creencia.

```
etr_object_in_new_belief(O):-
      etr_new_belief(_,_,O,_).
```

Este predicado revisa cuáles objetos no aparecen en ninguna observación.

```
etr_object_not_in_new_belief(O):-
      not etr_object_in_new_belief(O), etr_object(O), not
      etr_belief(O,storage).
```

Utilizamos las siguientes restricciones para eliminar respuestas enteras que contengan características que no deseamos aunque las reglas del programa las generen por ser lógicamente correctas, esto se hace siguiendo el modelo *"Guess and check"*.[1]

Esta restricción elimina respuestas en las que un objeto aparece en dos lugares diferentes en las nuevas creencias

```
:- etr_new_belief(X,_,O,L0), etr_new_belief(X,_,O,L1),
etr_object(O), etr_location(L0), etr_location(L1), L0!=L1,
etr_number_of_steps(X).
```

Esta restricción elimina las respuestas en las que no aparecen ni inferencias ni objetos es "ningún lado" si se creía que había un objeto en un lugar, no se tiene observaciones nuevas con ese objeto en ese lugar, pero si hay observaciones de ese lugar y se considera que si había un lugar donde podría haber estado.

```
:- etr_has_no_new_belief(c), etr_has_no_new_belief(e),
etr_belief(O,L0), not etr_new_observation(O,L0),
etr_new_observation(_,L0), etr_object(O), etr_location(L0),
etr_location(L1), L0!=L1, etr_place_where_it_can_be(L1).
```

Ese predicado elimina las respuestas en las que un objeto que sí está en escena no aparece en una nueva creencia.

```
:- etr_object_not_in_new_belief(O), etr_object(O).
```

Se usa esta restricción suave con un costo relativamente bajo (a comparación de otras restricciones suaves usadas) para quedarnos con las respuestas que tengan la menor cantidad de `etr_no_match`, y así seguir la lógica de buscar que se parezcan lo más posible el número de objetos por repisa de lo descrito por el asistente al principio del ejercicio y el número de objetos por repisa resultantes del entendimiento, observaciones e inferencias realizadas por este código.

```
:~ etr_no_match(_,_).[100:1]
```

Se usa este predicado para establecer que el tiempo se usar usando integrales (que están restringidos por #maxint que fue definido al principio)

```
etr_time(T):-
    #int(T).
```

Aquí se generan tipos de emparejamientos (los tipos dependen de si los objetos están juntos y acaban juntos, si empiezan juntos y acaban separados, si empiezan separados y acaban juntos y o si empiezan separados y terminan separados).

[1] Eiter, T., Ianni, G., & Krennwallner, T. (2009). Answer set programming: A primer. In Reasoning Web. Semantic Technologies for Information Systems(pp. 40-110). Springer Berlin Heidelberg.

```
etr_kind(a).
etr_kind(b).
etr_kind(c).
```

Estos predicados revisan si un objeto aparece en alguna de las observaciones anteriores (no la última realizada).

Reporta si un objeto aparece en una observación anterior

```
etr_previously_observed(O):- etr_old_observation(O,L).
```

Reporta si un objeto no aparece en ninguna de las observaciones anteriores.

```
etr_not_previously_observed(O):- not etr_previously_observed(O),
etr_object(O).
```

Este predicado (con 3 diferentes definiciones) reporta si un objeto o el asistente acaban en un lugar diferente al inicial.

Este predicado reporta si se sabe que un objeto cambió de lugar gracias a una observación anterior a la más nueva.

```
etr_difference(on,O,L0,L1):-
      O!=nothing, etr_original(0,O,L0), etr_old_observation(O,L1),
      L0!=L1.
```

Este predicado reporta si un objeto cambió de lugar gracias a una inferencia.

```
etr_difference(on,O,L0,L1):-
      O!=nothing, L0!=L1, etr_original(0,O,L0),
      etr_new_belief(X,_,O,L1), etr_number_of_steps(X),
      etr_not_previously_observed(O).
```

Este predicado reporta si el lugar donde el agente terminó es diferente a donde comenzó.

```
etr_difference(at,A,L0,L1):-
      etr_at_start(A,L0), etr_at_final(A,L1), etr_location(L0),
      etr_location(L1), etr_number_of_steps(X), L0!=L1.
```

Este predicado es la última etapa del diagnóstico, se usa para cambiar los predicados que contienen varias acciones encapsuladas a acciones atómicas (de forma que solo se tiene una acción por predicado). Las restricciones del final se usan para no intentar hacer match con `etr_explanation` que no tengan acciones en la posición que el predicado en particular está buscando extraer.

Este predicado obtiene la primer acción de la explicación.

```
etr_diagnosis(T,A,P):-
      etr_explanation(K,_,etr_action(T,A,P),_,_,_,_,_,_,_).
```

Este predicado obtiene la segunda acción de la explicación. Se evita hacer match con movimientos sencillos.

```
etr_diagnosis(T,A,P):-
      etr_explanation(K,_,_,etr_action(T,A,P),_,_,_,_,_,_),K!=m.
```

Este predicado obtiene la tercer acción de la explicación. Se evita hacer match con movimientos sencillos.

```
etr_diagnosis(T,A,P):-
      etr_explanation(K,_,_,_,etr_action(T,A,P),_,_,_,_,_),K!=m.
```

Este predicado obtiene la cuarta acción de la explicación. Se evita hacer match con movimientos sencillos.

```
etr_diagnosis(T,A,P):-
      etr_explanation(K,_,_,_,_,etr_action(T,A,P),_,_,_,_),K!=m.
```

Este predicado obtiene la quinta acción de la explicación. Se evita hacer match con movimientos sencillos y movimientos de un solo objeto.

```
etr_diagnosis(T,A,P):-
      etr_explanation(K,_,_,_,_,_,etr_action(T,A,P),_,_,_),K!=m,K!=s.
```

Este predicado obtiene la quinta acción de la explicación. Se evita hacer match con movimientos sencillos y movimientos de un solo objeto.

```
etr_diagnosis(T,A,P):-
      etr_explanation(K,_,_,_,_,_,_,etr_action(T,A,P),_,_),K!=m,K!=s.
```

Este predicado obtiene la quinta acción de la explicación. Se evita hacer match con movimientos sencillos, movimientos de un solo objeto y movimientos de dos objetos que comienzan y terminan juntos.

```
etr_diagnosis(T,A,P):-
      etr_explanation(K,_,_,_,_,_,_,_,etr_action(T,A,P),_),K!=m,K!=s,K
      !=pa.
```

Este predicado obtiene la quinta acción de la explicación. Se evita hacer match con movimientos sencillos, movimientos de un solo objeto, movimientos de dos objetos que comienzan y terminan juntos y movimientos en que los objetos empiezan separados y terminan juntos o empiezan juntos y terminan separados.

```
etr_diagnosis(T,A,P):-
      etr_explanation(K,_,_,_,_,_,_,_,_,etr_action(T,A,P)),K!=m,K!=s,K
      !=pa,K!=pb1,K!=pb2.
```

Este conjunto de predicados hace una serie de acciones de manera que no sea necesario calcularlas ya que una vez que sabemos que, por ejemplo, se van a mover dos objetos de un lugar a otro, se puede hacer la siguiente secuencia, ir al primer lugar, tomar un objeto, tomar el otro, ir al segundo lugar, depositar el primer objeto y depositar el segundo. Dado que es trivial la diferencia entre tomar

primero uno o tomar primero el otro, esto se evita usando cláusulas que evalúan el nombre del objeto para siempre tomar primero el alfabéticamente inferior. Este predicado asigna un tiempo a cada acción dentro de cada explicación para que las acciones puedan ser ordenadas en una lista (esto se hace en prolog), así las cláusulas A representa un tiempo en el que se ejecuta la acción, es importante observar que las A se calculan con operaciones aritméticas mientras que T (que es el tiempo de la explicación) es el que se explora. La diferencia entre T y A es muy importante pues es la que permite tener planes relativamente largos ya que de lo contrario el tiempo que toma hacer exploraciones individuales para cada acción se vuelve prohibitivo.

Este predicado genera la explicación para movimientos sencillos, la cláusula sobre diferencia tiene que ser la de posicionamiento (at) y el tiempo es al final (X) de manera que este sea el movimiento necesario para regresar al inicio. La multiplicación A = T*10 se hace para que las acciones sigan ordenados (no se empalmen) con respecto a las acciones de otros predicados.

```
etr_explanation(m,T,etr_action(A1,goto,L1),nada,nada,nada,nada,nada,nada,nada):-
      etr_difference(at,_,_,L1),T=X,etr_number_of_steps(X),
      etr_time(T), A=T*10, A1=A+1.
```

Este predicado genera la explicación para cuando se mueve un objeto por sí solo (en caso de tener un número impar de objetos en escena).

```
etr_explanation(s,T,etr_action(A1,goto,L0),etr_action(A2,take,I),etr_action(A3,goto,L1),etr_action(A4,deliver,I),nada,nada,nada,nada):-
      etr_alone(T,I), etr_time(T), etr_difference(on,I,L0,L1), T!=0,
      T<X, etr_number_of_steps(X), A=T*10,
      A1=A+1,A2=A+2,A3=A+3,A4=A+4.
```

Este predicado explica el orden de las acciones necesarias cuando se mueven dos objetos que estaban en el mismo lugar originalmente y terminan en el mismo lugar.

```
etr_explanation(pa,T,etr_action(A1,goto,L0),etr_action(A2,take,I0),etr_action(A3,take,I1),etr_action(A4,goto,L1),etr_action(A5,deliver,I0),etr_action(A6,deliver,I1),nada,nada):-
      etr_pairing(a,T,I0,I1), etr_difference(on,I0,L0,L1),
      etr_difference(on,I1,L0,L1), etr_time(T), I0!=I1, I0<I1, T!=0,
      T<X, etr_number_of_steps(X), A=T*10,
      A1=A+1,A2=A+2,A3=A+3,A4=A+4,A5=A+5,A6=A+6.
```

Este predicado genera la explicación del orden de las acciones cuando se mueven dos objetos que estaban en el mismo lugar y terminan en lugares diferentes.

```
etr_explanation(pb1,T,etr_action(A1,goto,L0),etr_action(A2,take,I0),etr_action(A3,take,I1),etr_action(A4,goto,L1),etr_action(A5,deliver,I0),etr_action(A6,goto,L2),etr_action(A7,deliver,I1),nada):-
      etr_pairing(b,T,I0,I1), etr_difference(on,I0,L0,L1),
      etr_difference(on,I1,L0,L2), L1!=L2, etr_time(T), I0!=I1, I0<I1,
```

```
T!=0, T<X, etr_number_of_steps(X), A=T*10,
A1=A+1,A2=A+2,A3=A+3,A4=A+4,A5=A+5,A6=A+6,A7=A+7.
```

Este predicado genera la explicación del orden de las acciones cuando se mueven dos objetos que estaban en lugares diferentes y acaban en el mismo lugar.

```
etr_explanation(pb2,T,etr_action(A1,goto,L0),etr_action(A2,take,I0),e
tr_action(A3,goto,L2),etr_action(A4,take,I1),etr_action(A5,goto,L0),e
tr_action(A6,deliver,I0),etr_action(A7,deliver,I1),nada):-
    etr_pairing(b,T,I0,I1), etr_difference(on,I0,L1,L0),
    etr_difference(on,I1,L2,L0), L1!=L2, etr_time(T), I0!=I1, I0<I1,
    T!=0, T<X, etr_number_of_steps(X), A=T*10,
    A1=A+1,A2=A+2,A3=A+3,A4=A+4,A5=A+5,A6=A+6,A7=A+7.
```

Este predicado genera la explicación del orden de las acciones cuando se mueven dos objetos que estaban en lugares diferentes y acaban en lugares diferentes.

```
etr_explanation(pc,T,etr_action(A1,goto,L0),etr_action(A2,take,I0),et
r_action(A3,goto,L2),etr_action(A4,take,I1),etr_action(A5,goto,L1),et
r_action(A6,deliver,I0),etr_action(A7,goto,L3),etr_action(A8,deliver,
I1)):-
    etr_pairing(c,T,I0,I1), etr_difference(on,I0,L0,L1),
    etr_difference(on,I1,L2,L3), L0!=L1, L2!=L3, etr_time(T),
    I0!=I1, I0<I1, T!=0, T<X, etr_number_of_steps(X), A=T*10,
    A1=A+1,A2=A+2,A3=A+3,A4=A+4,A5=A+5,A6=A+6,A7=A+7,A8=A+8.
```

Se rechazan soluciones en las que hay más de una explicación donde se mueve un objeto por sí solo ya que entonces se podría haber formado una pareja.

```
:- etr_explanation(s,T0,_,_,_,_,_,_,_,_),
etr_explanation(s,T1,_,_,_,_,_,_,_,_), T0 != T1.
```

El siguiente bloque de predicados explora los posibles emparejamientos de objetos que se pueden formar (ya que se asume que el asistente mueve los objetos usando sus dos manos con un objeto por mano como máximo), se prefieran los emparejamientos que representan menos acciones (en los que los objetos comienzan y terminan juntos, por ejemplo).

Este predicado empareja objetos que comienzan y terminan juntos

```
etr_pairing(a,T,I0,I1) v -etr_pairing(a,T,I0,I1):-
    etr_difference(on,I0,L0,L1), etr_difference(on,I1,L0,L1),
    etr_time(T), I0!=I1, I0<I1, T!=0, T<X, etr_number_of_steps(X).
```

Este predicado empareja objetos que comienzan juntos y terminan separados.

```
etr_pairing(b,T,I0,I1) v -etr_pairing(b,T,I0,I1):-
```

```
      etr_difference(on,I0,L0,L1), etr_difference(on,I1,L0,L2),
      L1!=L2, etr_time(T), I0!=I1, I0<I1, T!=0, T<X,
      etr_number_of_steps(X).
```

Este predicado empareja objetos que comienzan separados y terminan juntos.

```
etr_pairing(b,T,I0,I1) v -etr_pairing(b,T,I0,I1):-
      etr_difference(on,I0,L1,L0), etr_difference(on,I1,L2,L0),
      L1!=L2, etr_time(T), I0!=I1, I0<I1, T!=0, T<X,
      etr_number_of_steps(X).
```

Este predicado empareja objetos que comienzan separados y terminan separados

```
etr_pairing(c,T,I0,I1) v -etr_pairing(c,T,I0,I1):-
      etr_difference(on,I0,L0,L1), etr_difference(on,I1,L2,L3),
      L0!=L1, L2!=L3, etr_time(T), I0!=I1, I0<I1, T!=0, T<X,
      etr_number_of_steps(X).
```

Se rechazan emparejamientos para así terminar con la explicación o diagnóstico que considera que el asistente utilizó el mínimo de acciones para colocar los objetos.

Esta restricción elimina soluciones en las que 2 emparejamientos comparten el mismo tiempo.

```
      :- etr_pairing(P0,T,I0,I1), etr_pairing(P1,T,I2,I3),
      etr_time(T), etr_kind(P0), etr_kind(P1), I0!=I2, I1!=I3.
```

Esta restricción elimina soluciones en las que 2 emparejamientos comparten el mismo tiempo y un objeto.

```
      :- etr_pairing(P0,T,I0,I), etr_pairing(P1,T,I2,I), etr_time(T),
      etr_kind(P0), etr_kind(P1),I0!=I2.
```

Esta restricción elimina soluciones en las que 2 emparejamientos comparten el mismo tiempo y un objeto.

```
      :- etr_pairing(P0,T,I,I1), etr_pairing(P1,T,I,I3), etr_time(T),
      etr_kind(P0), etr_kind(P1),I1!=I3.
```

Esta restricción elimina soluciones en las que se comparte un objeto aunque estén en tiempos diferentes

```
      :-  etr_pairing(P0,T0,I0,I1),  etr_pairing(P1,T1,I,I3),  I=I0,
      etr_kind(P0), etr_kind(P1),T0<T1.
```

Esta restricción elimina soluciones en las que se comparte un objeto aunque estén en tiempos diferentes

```
      :-  etr_pairing(P0,T0,I0,I1),  etr_pairing(P1,T1,I,I3),  I=I1,
      etr_kind(P0), etr_kind(P1),T0<T1.
```

Esta restricción elimina soluciones en las que se comparte un objeto aunque estén en tiempos diferentes

```
:-  etr_pairing(P0,T0,I0,I1),  etr_pairing(P1,T1,I3,I),  I=I0,
etr_kind(P0), etr_kind(P1),T0<T1.
```

Esta restricción elimina soluciones en las que se comparte un objeto aunque estén en tiempos diferentes

```
:-  etr_pairing(P0,T0,I0,I1),  etr_pairing(P1,T1,I3,I),  I=I1,
etr_kind(P0), etr_kind(P1),T0<T1.
```

Esta restricción elimina soluciones en las que se tienen emparejamientos menos atractivos (con menos acciones) antes en el tiempo que emparejamientos más atractivos.

```
:- etr_pairing(a,T0,_,_), etr_pairing(b,T1,_,_), T0>=T1.
```

Esta restricción elimina soluciones en las que se tienen emparejamientos menos atractivos (con menos acciones) antes en el tiempo que emparejamientos más atractivos.

```
:- etr_pairing(b,T0,_,_), etr_pairing(c,T1,_,_), T0>=T1.
```

Este predicado explora posibles acomodos de objetos que se tienen que mover solos.

```
etr_alone(T,I) v -etr_alone(T,I):-
    etr_time(T),    etr_difference(on,I,L0,L1),    T!=0,    T<X,
    etr_number_of_steps(X).
```

Esta restricción elimina soluciones en las que se comparte un objeto entre un emparejamiento y un elemento solitario.

```
:- etr_alone(T0,I), etr_pairing(_,T1,I0,I).
```

Esta restricción elimina soluciones en las que se comparte un objeto entre un emparejamiento y un elemento solitario.

```
:- etr_alone(T0,I), etr_pairing(_,T1,I,I0).
```

Esta restricción elimina soluciones en las que aparece un solitario antes en el tiempo que un emparejamiento.

```
:- etr_pairing(_,T0,_,_), etr_alone(T1,_), T0>=T1.
```

Estos predicados se usan para ver si hay tiempos que no contengan algún emparejamiento

```
etr_used_pairing(T):-
    etr_pairing(_,T,_,_), etr_time(T).
etr_empty_pairing(T):-
    etr_time(T), not etr_used_pairing(T).
```

Estos predicados se usan para ver si hay tiempos que no contengan algún elemento solitario

```
etr_used_alone(T):-
```

```
      etr_alone(T,_), etr_time(T).
etr_empty_alone(T):-
      etr_time(T), not etr_used_alone(T).
```

El siguiente bloque de restricciones suaves se usa para asignar un costo a cierto predicados de manera que se puedan obtener las soluciones que contienen el costo más bajo, esto lo hacemos pues aunque hay soluciones que son técnicamente válidas, no son deseables (por ejemplo una solución donde haya emparejamientos en los que los objetos empiezan separados y terminan separados cuando estos objetos podían ser emparejados de manera que comenzarán junto y terminaran juntos y así se podrían ahorrar acciones o por ejemplo que tengan tiempo intermedios vacíos).

Se le asignan costos más bajos a los predicados que se consideran más deseables y costos mayores a los que son menos deseables (aunque ambos sean válidos y puedan en teoría aparecer en la solución provista por el código).

```
:~ etr_pairing(a,T,I0,I1).[1:1]
:~ etr_pairing(b,T,I0,I1).[10:1]
:~ etr_pairing(b,T,I0,I1).[10:1]
:~ etr_pairing(c,T,I0,I1).[100:1]
:~ etr_alone(T,I).[1000:1]
:~ etr_empty_pairing(T), etr_time(T).[10000:1]
:~ etr_empty_alone(T), etr_time(T).[5000:1]
```

En resumen:

Este código toma la información de la base de conocimientos simplificada en el archivo ETR_A_Diagnosis_World.dl y genera una serie de predicados que le permiten;:

1) Proponer las locaciones finales de los objetos basándose en creencias pasadas, observaciones e inferencias que buscan mantener el número de objetos por repisa lo más cercanos posible al número de objetos por repisa que había reportado el asistente.
2) Generar una lista de objetos y la diferencia entre dónde empiezan y dónde acaban según las nuevas creencias.
3) Emparejar los objetos de forma que el transportarlos implique el mínimo de acciones posible.
4) Generar la secuencia de pasos que permiten mover los objetos de un lugar a otro según el tipo de emparejamiento al que pertenecen.
5) Dividir las secuencias en acciones atómicas .
6) Además de lo anterior, el código también genera una lista de cambios necesarios a la base de conocimiento para reflejar las nuevas creencias.

Los predicados generados por este código son similares (dependiendo de la información contenida en la base de conocimiento) a los siguientes:

Se muestran cambios de donde antes se pensaba que había algo a donde se cree que puede estar ahora.

```
etr_change_of_place(kellogs,storage,shelf2),
etr_change_of_place(coke,shelf2,shelf3),
```

Así, se indica que hay que modificar la base de conocimientos para representar que el cereal se encuentra en la repisa de comida y que el refresco está en la repisa de pan.

Se muestra la narrativa ordenada por tiempo (aunque no contigua) de cómo es que el asistente colocó los objetos en la escena.

```
etr_diagnosis(11,goto,storage), etr_diagnosis(21,goto,storage),
etr_diagnosis(31,goto,storage), etr_diagnosis(12,take,heineken),
etr_diagnosis(22,take,bisquits), etr_diagnosis(32,take,kellogs),
etr_diagnosis(33,goto,shelf2), etr_diagnosis(13,take,noodles),
etr_diagnosis(23,take,coke), etr_diagnosis(34,deliver,kellogs),
etr_diagnosis(14,goto,shelf1), etr_diagnosis(24,goto,shelf3),
etr_diagnosis(15,deliver,heineken),
etr_diagnosis(25,deliver,bisquits),
etr_diagnosis(16,deliver,noodles), etr_diagnosis(26,deliver,coke).
```

Si se ordenan (esto se hace en prolog) se obtiene:

```
etr_diagnosis(11,goto,storage),
etr_diagnosis(12,take,heineken),
etr_diagnosis(13,take,noodles),
etr_diagnosis(14,goto,shelf1),
etr_diagnosis(15,deliver,heineken),
etr_diagnosis(16,deliver,noodles),
etr_diagnosis(21,goto,storage),
etr_diagnosis(22,take,bisquits),
etr_diagnosis(23,take,coke),
etr_diagnosis(24,goto,shelf3),
etr_diagnosis(25,deliver,bisquits),
etr_diagnosis(26,deliver,coke),
etr_diagnosis(31,goto,storage),
etr_diagnosis(32,take,kellogs),
etr_diagnosis(33,goto,shelf2),
etr_diagnosis(34,deliver,kellogs),
```

Que es la secuencia de acciones que transforman el mundo en el que los objetos están todos en el almacén, al mundo en que la cerveza y los tallarines está en la repisa de bebidas, el cereal está en la repisa de la comida y el refresco y los bisquets están en la repisa del pan que es como el asistente lo puse originalmente.

ETR_A_Transform_Base_01.dl

Este código regresa los siguientes predicados:
`Class,parent,properta,-properta,instance,special_class`

Dado que se genera el predicado `-properta`, en vez de usar un filtro para positivos `-pfilter=` se usa un filtro normal `-filter=`:

Este código realiza los cálculos necesarios para incorporar los cambios a la base de conocimientos generados por el código de diagnóstico arriba explicado. Utiliza la base de conocimiento ETR_A_Prolog_To_ASP.dl así como los cambios que se encuentran en el archivo ETR_A_Changes.dl.

Estos predicados muestran si un objeto (descrito por su marca) ha experimentado un cambio en su localización.

Muestra cuáles objetos no han cambiado de lugar (son aquellos que no se encuentran en la lista de objetos con cambio de lugar generada por el predicado de abajo)

```
etr_no_change(A):-
      not etr_change(A), property(X,brand,A).
```

Muestra si un objeto objeto ha cambiado de lugar

```
etr_change(A):-
      etr_change_of_place(A,_,_).
```

Se utiliza este predicado por la siguiente razón: como se están pasando las propiedades “in” de los objetos, se tienen que separa en aquellas que pasan directo (pues no tienen un cambio) y aquellas que pasan pero cambiando el lugar en el que están. Se utiliza el cambio de property a properta ya que de lo contrario, lógicamente al obtener los resultados los property originales también pasaría (por lo cual efectivamente habíamos duplicado la propiedad in de los objetos que experimentaron el cambio pero con un nuevo lugar). Este cambio de letra nos permite solo recibir aquellos predicados “nuevos” y así evitar duplicados.

Regresa los predicado property “in” de aquellos objetos que sí experimentaron cambios.

```
properta(A,in,Z):-
      property(A,in,C), etr_change_of_place(X,Y,Z),
      property(A,brand,X).
```

Regresa los predicado property in de aquellos objetos que no experimentaron cambios.

```
properta(A,in,C):-
      property(A,in,C), etr_no_change(X), property(A,brand,X).
```

El resto de la base de conocimientos pasa directamente sin cambios con excepción que todos los predicados que antes se llamaban property, a la salida se llamarán propuesta (incluidos los negativos).

```
class(A):-
      class(A).
parent(A,B):-
      parent(A,B).
properta(A,B,C):-
      property(A,B,C), B!=in.
-properta(A,B,C):-
      -property(A,B,C).
instance(A,B,C):-
      instance(A,B,C).
special_class(A,B,C,D,E):-
      special_class(A,B,C,D,E).
```

Así tomando en cuenta que en el archivo de cambios aparezca

```
etr_change_of_place(kellogs,storage,shelf2),
```

Una clase que originalmente se mostraba de la siguiente forma en la base de conocimientos que se usa para *Answer Set Programming*:

```
class("cereal").
parent("cereal",food).
property("cereal",inv,0).
instance("cereal",id,c1).
property(c1,brand,kellogs).
property(c1,original_id,c1).
property(c1,in,storage).
```

Pasa de la siguiente forma

```
class("cereal").
parent("cereal",food).
properta("cereal",inv,0).
instance("cereal",id,c1).
properta(c1,brand,kellogs).
properta(c1,original_id,c1).
properta(c1,in,shelf2).
```

ETR_A_Transform_Base_02.dl

Este código regresa los siguientes predicados:
`Class,parent,property,-property,instance,special_class`

Dado que se genera el predicado `-property`, en vez de usar un filtro para positivos `-pfilter=` se usa un filtro normal `-filter=`:

Este código simplemente transforma los predicados properta y -properta a property y -property respectivamente para que entonces se tenga la base de conocimientos más cercana posible a la original pero ya con los cambios generados por el código de diagnóstico. Utiliza la base de conocimiento modificada del archivo `ETR_A_Auxiliary_Base.dl`

Esta lista de predicados permite regresar los predicados "properta" a "property" para poder así trabajar sobre los mismos en el siguiente y último paso realizado en *Answer Set Programming* que es regresar la base de conocimientos a un formato de clases como listas con listas para representar propiedades, relaciones e instancias.

```
class(A):-
      class(A).
parent(A,B):-
      parent(A,B).
property(A,B,C):-
      properta(A,B,C).
-property(A,B,C):-
      -properta(A,B,C).
instance(A,B,C):-
      instance(A,B,C).
special_class(A,B,C,D,E):-
      special_class(A,B,C,D,E).
```

Así, por ejemplo, una clase que se recibe de la siguiente manera:

```
class("cereal").
parent("cereal",food).
properta("cereal",inv,0).
instance("cereal",id,c1).
properta(c1,brand,kellogs).
properta(c1,original_id,c1).
properta(c1,in,shelf2).
```

Se regresa de la siguiente forma:

```
class("cereal").
parent("cereal",food).
property("cereal",inv,0).
instance("cereal",id,c1).
property(c1,brand,kellogs).
property(c1,original_id,c1).
property(c1,in,shelf2).
```

ETR_A_Return.dl

Este código regresa los siguientes predicados:
`etr_class`

Como estamos generando listas y esto podría potencialmente un infinito de predicados (si no maneja correctamente la información). Es necesario indicar a DLV que no es necesario que haga una prueba para ver si el código permite ciclos infinitos, esto se hace usando la bandera -nofinitecheck de lo contrario, el sistema DLV no podría manejar el código dada la presencia de posibles ciclos infinitos.

Este código transforma la base de conocimientos de una serie de predicados atómicos a predicados del tipo `etr_class` que tiene una estructura similar a la de las clases que se manejan en la base de conocimientos original (aunque se incluye mucha información extra que después es retirada en prolog para así acabar con la base de conocimientos en el mismo formato que se tomó). La transformación antes descrita es importante ya que la base de conocimientos es la forma es que el submódulo de diagnóstico se comunica con él los otros submódulos para así poder realizar las labores necesarias del módulo de inferencia. Este código utiliza la información generada en el código anterior que se encuentra guardada en el archivo ETR_A_Return_Base.dl

Como no utilizamos el operador "=>" en *Answer Set Programming* que se utiliza en la base de conocimiento de Golem para representar una relación entre una característica y su valor, lo hacemos aquí como un predicado, dado que estos operadores se encuentran en las propiedades y en las instancias, son esos predicados los que se transforman en relaciones de operador.

Se presenta una relación del operador "=>" para las propiedades.

```
etr_operator(A,B,C):-
      property(A,B,C).
```

Se presenta una relación del operador "=>" para las instancias.

```
etr_operator(A,B,C):-
      instance(A,B,C).
```

La generación de listas en DLV es un proceso que se usa varias veces en este código por lo que será explicado aquí y luego se asumirá que el resto de las listas que se generan siguen procesos análogos (con variaciones dependiendo de que se está listando).

Se tiene un predicado en caso de que la clase A no tenga propiedades de instancia

```
etr_instance_property_list(Class,A,[]):-
      instance(Class,_,A), not etr_has_instance_properties(A).
```

En caso de que la clase A tenga propiedades de instancia, estas se añaden a la lista vacía del principio (así es como se "inicia" la lista).

```
etr_instance_property_list(Class,A,X):-
```

```
instance(Class,_,A),
#append([],[etr_instance_property(A,B,C)],X),
etr_instance_property(A,B,C).
```

En caso de que se tengan dos propiedades de instancia, se añaden siempre y cuando la propiedad a añadir sea mayor "alfabéticamente" esto se hace para evitar generar diferencias triviales en las que se cuenta con la misma información pero en orden distinto pues el orden no importa en la base de conocimientos de Golem.

```
etr_instance_property_list(Class,A,X):-
    #append([etr_instance_property(A,B,C)],[etr_instance_property(A,
    D,E)],X),   instance(Class,_,A),   etr_instance_property(A,B,C),
    etr_instance_property(A,D,E),
    etr_instance_property(A,B,C)<etr_instance_property(A,D,E).
```

Aquí es donde se puede genera el ciclo infinito pues este predicado está llamando al mismo predicado. Para generar la lista se añade la nueva propiedad de instancia a la lista siempre y cuando esta propiedad no sea ya miembro de la lista a la que va a ser agregada y sea alfabéticamente mayor a la última propiedad de la lista a la que va a ser agregada para así evitar como en el predicado anterior listas trivialmente diferentes (que contienen la misma información pero en orden diferente).

```
etr_instance_property_list(Class,A,X):-
    #append(Y,[etr_instance_property(A,B,C)],X),
    instance(Class,_,A),      etr_instance_property_list(Class,A,Y),
    etr_instance_property(A,B,C),                                  not
    #member(etr_instance_property(A,B,C),       Y),       #last(Y,D),
    etr_instance_property(A,B,C)>D.
```

Este predicado cuenta cuántas propiedades tiene en total la instancia.

```
etr_amount_of_instance_properties(A,B):-
    #count{I : etr_instance_property(A,C,I)} = B,
    etr_has_instance_properties(A), etr_instance_property(A,_,_).
```

Este predicado cuenta cuántas propiedades tiene en total la instancia en caso de que no tenga propiedades se registra 0 como la cantidad.

```
etr_amount_of_instance_properties(A,0):-
    not etr_has_instance_properties(A), instance(_,_,A).
```

Para poder determinar que ya se tiene la lista de propiedades completa, se revisa su longitud contra el número de propiedades de la instancia que había originalmente.

```
etr_final_instance_properties_list(Class,A,X):-
    etr_instance_property_list(Class,A,X), #length(X,Y),
    etr_amount_of_instance_properties(A,Y).
```

Revisa si la instancia tiene por lo menos una propiedad.

```
etr_has_instance_properties(A):-
      instance(_,_,A), property(A,_,_).
```

Transforma las propiedades relacionadas con instancias en predicados de propiedad de instancia para ser agregadas en una lista.

```
etr_instance_property(A,B,C):-
      instance(_,_,A), property(A,B,C).
```

El siguiente bloque es esencialmente igual al anterior pero buscando hacer lista de las relaciones de una instancia en vez de sus propiedades.

Se tiene un predicado en caso de que la clase A no tenga relaciones de instancia

```
etr_instance_relationship_list(Class,A,[]):-
      instance(Class,_,A), not etr_has_instance_relationships(A).
```

En caso de que la clase A tenga relaciones de instancia, estas se añaden a la lista vacía del principio (así es como se "inicia" la lista).

```
etr_instance_relationship_list(Class,A,X):-
      instance(Class,_,A),
      #append([],[etr_instance_relationship(A,B,C)],X),
      etr_instance_relationship(A,B,C).
```

En caso de que se tengan dos relaciones de instancia, se añaden siempre y cuando la relación a añadir sea mayor "alfabéticamente" esto se hace para evitar generar diferencias triviales en las que se cuenta con la misma información pero en orden distinto pues el orden no importa en la base de conocimientos de Golem.

```
etr_instance_relationship_list(Class,A,X):-
      #append([etr_instance_relationship(A,B,C)],[etr_instance_relatio
      nship(A,D,E)],X), instance(Class,_,A),
      etr_instance_relationship(A,B,C),
      etr_instance_relationship(A,D,E),
      etr_instance_relationship(A,B,C)<etr_instance_relationship(A,D,E
      ).
```

Aquí es donde se puede genera el ciclo infinito pues este predicado está llamando al mismo predicado. Para generar la lista se añade la nueva relación de instancia a la lista siempre y cuando esta relación no sea ya miembro de la lista a la que va a ser agregada y sea alfabéticamente mayor a la última relación de la lista a la que va a ser agregada para así evitar como en el predicado anterior listas trivialmente diferentes (que contienen la misma información pero en orden diferente).

```
etr_instance_relationship_list(Class,A,X):-
      #append(Y,[etr_instance_relationship(A,B,C)],X),
      instance(Class,_,A), etr_instance_relationship_list(Class,A,Y),
```

```
      etr_instance_relationship(A,B,C), not
      #member(etr_instance_relationship(A,B,C), Y), #last(Y,D),
      etr_instance_relationship(A,B,C)>D.
```

Este predicado cuenta cuántas relaciones tiene en total la instancia.

```
etr_amount_of_instance_relationships(A,B):-
      #count{I : etr_instance_relationship(A,C,I)} = B,
      etr_has_instance_relationships(A),
      etr_instance_relationship(A,_,_).
```

Este predicado cuenta cuántas relaciones tiene en total la instancia en caso de que no tenga relaciones se registra 0 como la cantidad.

```
etr_amount_of_instance_relationships(A,0):-
      not etr_has_instance_relationships(A), instance(_,_,A).
```

Para poder determinar que ya se tiene la lista de relaciones completa, se revisa su longitud contra el número de relaciones de la instancia que había originalmente.

```
etr_final_instance_relationship_list(Class,A,X):-
      etr_instance_relationship_list(Class,A,X), #length(X,Y),
      etr_amount_of_instance_relationships(A,Y).
```

Revisa si la instancia tiene por lo menos una relación.

```
etr_has_instance_relationships(A):-
      instance(_,_,A), relation(A,_,_).
```

Transforma las relaciones relacionadas con instancias en predicados de relación de instancia para ser agregadas en una lista.

```
etr_instance_relationship(A,B,C):-
      instance(_,_,A), relation(A,B,C).
```

Este predicado toma la información necesaria y genera un solo predicado que contiene la información de una instancia completa con las listas antes generadas así como la clase a la que pertenece.

```
etr_instance(Class,etr_operator(Class,B,C),etr_final_instance_propert
ies_list(Class,C,X),etr_final_instance_relationship_list(Class,C,Y)):
-
      class(Class), etr_operator(Class,B,C),
      etr_final_instance_properties_list(Class,C,X),etr_final_instance
      _relationship_list(Class,C,Y).
```

En este bloque se genera una lista de instancias para cada clase (las instancias que se listan son las instancias generadas con el código previamente descrito).

En caso de que una clase no tenga instancias se usa la lista vacía

```
etr_class_instance_list(A,[]):-
      class(A), not etr_has_class_instances(A).
```

Se añade una instancia para iniciar una lista de instancias.

```
etr_class_instance_list(A,X):-
      class(A),           #append([],[etr_class_instance(A,B,C,D)],X),
      etr_class_instance(A,B,C,D).
```

Este predicado se usa para añadir instancias a la a lista de instancias siempre y cuando una instancia sea alfabéticamente mayor a la otra.

```
etr_class_instance_list(A,X):-
      #append([etr_class_instance(A,B,C,D)],[etr_class_instance(A,E,F,
      G)],X), class(A), etr_class_instance(A,B,C,D),
      etr_class_instance(A,E,F,G),
      etr_class_instance(A,B,C,D)<etr_class_instance(A,E,F,G).
```

Este predicado puede genera un ciclo infinito pues se llama a sí mismo, se añade una instancia a la lista de instancias siempre y cuando la instancia sea alfabéticamente mayor al último miembro de la lista para evitar listas que son trivialmente diferentes al contener la misma información pero en orden diferente.

```
etr_class_instance_list(A,X):-
      #append(Y,[etr_class_instance(A,B,C,D)],X), class(A),
      etr_class_instance_list(A,Y), etr_class_instance(A,B,C,D), not
      #member(etr_class_instance(A,B,C,D), Y), #last(Y,E),
      etr_class_instance(A,B,C,D)>E.
```

Se cuentan cuántas instancia tiene un clase originalmente

```
etr_amount_of_class_instances(A,B):-
      #count{I : etr_class_instance(A,D,I,C)} = B,
      etr_has_class_instances(A), etr_class_instance(A,_,_,_).
```

Si la clase no tiene instancias se asigna 0 como la cantidad

```
etr_amount_of_class_instances(A,0):-
      not etr_has_class_instances(A), class(A).
```

Cuando la lista de instancias tiene la misma cantidad de instancias que había instancias de esa clase originalmente se tiene entonces la lista final

```
etr_final_class_instance_list(A,X):-
      etr_class_instance_list(A,X), #length(X,Y),
      etr_amount_of_class_instances(A,Y).
```

Revisa si la clase tiene por lo menos una instancia

```
etr_has_class_instances(A):-
      class(A), etr_instance(A,_,_,_).
```

Transforma las instancias relacionadas con clases en predicados de instancias de clase para ser agregadas en una lista.

```
etr_class_instance(A,B,C,D):-
      class(A), etr_instance(A,B,C,D).
```

El siguiente bloque de código hace una lista con las propiedades de una clase de la misma manera que los bloques anteriores.

En caso de que una clase no tenga propiedades se usa la lista vacía

```
etr_class_properties_list(A,[]):-
      class(A), not etr_has_class_properties(A).
```

Se añade una instancia para iniciar una lista de propiedades.

```
etr_class_properties_list(A,X):-
      class(A),              #append([],[etr_class_property(A,B,C)],X),
      etr_class_property(A,B,C).
```

Este predicado se usa para añadir propiedades a la a lista de propiedades siempre y cuando una propiedad sea alfabéticamente mayor a la otra.

```
etr_class_properties_list(A,X):-
      #append([etr_class_property(A,B,C)],[etr_class_property(A,E,F)],
      X), class(A), etr_class_property(A,B,C),
      etr_class_property(A,E,F),
      etr_class_property(A,B,C)<etr_class_property(A,E,F).
```

Este predicado puede genera un ciclo infinito pues se llama a sí mismo, se añade una propiedad a la lista de propiedades siempre y cuando la propiedad sea alfabéticamente mayor al último miembro de la lista para evitar listas que son trivialmente diferentes al contener la misma información pero en orden diferente.

```
etr_class_properties_list(A,X):-
      #append(Y,[etr_class_property(A,B,C)],X), class(A),
      etr_class_properties_list(A,Y), etr_class_property(A,B,C), not
      #member(etr_class_property(A,B,C), Y), #last(Y,E),
      etr_class_property(A,B,C)>E.
```

Se cuentan cuántas propiedades tiene un clase originalmente

```
etr_amount_of_class_properties(A,B):-
      #count{I : etr_class_property(A,D,I)} = B,
      etr_has_class_properties(A), etr_class_property(A,_,_).
```

Si la clase no tiene propiedades se asigna 0 como la cantidad

```
etr_amount_of_class_properties(A,0):-
      not etr_has_class_properties(A), class(A).
```

Cuando la lista de propiedades tiene la misma cantidad de propiedades que había propiedades de esa clase originalmente se tiene entonces la lista final

```
etr_final_class_properties_list(A,X):-
      etr_class_properties_list(A,X), #length(X,Y),
      etr_amount_of_class_properties(A,Y).
```

Revisa si la clase tiene por lo menos una propiedad

```
etr_has_class_properties(A):-
      class(A), property(A,_,_).
```

Transforma las propiedades relacionadas con clases en predicados de propiedades de clase para ser agregadas en una lista.

```
etr_class_property(A,B,C):-
      class(A), etr_rasterize_class_property(A,B,C).
```

Estos predicados unifican las propiedades positivas y negativas de forma que ambas puedan ser tratadas de igual forma por el bloque anterior.

Si la propiedad es positiva pasa directa

```
etr_rasterize_class_property(A,B,C):-
      class(A), property(A,B,C).
```

Si la propiedad es negativa esto se indica con un “no” como valor de la propiedad

```
etr_rasterize_class_property(A,B,no):-
      class(A), -property(A,B,C).
```

El siguiente bloque hace una lista de relaciones de clase de la misma forma que los bloques anteriores

En caso de que una clase no tenga relaciones se usa la lista vacía

```
etr_class_relationships_list(A,[]):-
      class(A), not etr_has_class_relationships(A).
```

Se añade una instancia para iniciar una lista de relaciones

```
etr_class_relationships_list(A,X):-
      class(A),          #append([],[etr_class_relationship(A,B,C)],X),
      etr_class_relationship(A,B,C).
```

Este predicado se usa para añadir relaciones a la a lista de relaciones siempre y cuando una relación sea alfabéticamente mayor a la otra.

```
etr_class_relationships_list(A,X):-
      #append([etr_class_relationship(A,B,C)],[etr_class_relationship(
      A,E,F)],X), class(A), etr_class_relationship(A,B,C),
      etr_class_relationship(A,E,F),
      etr_class_relationship(A,B,C)<etr_class_relationship(A,E,F).
```

Este predicado puede genera un ciclo infinito pues se llama a sí mismo, se añade una relación a la lista de relaciones siempre y cuando la relación sea alfabéticamente mayor al último miembro de la lista para evitar listas que son trivialmente diferentes al contener la misma información pero en orden diferente.

```
etr_class_relationships_list(A,X):-
      #append(Y,[etr_class_relationship(A,B,C)],X), class(A),
      etr_class_relationships_list(A,Y),
      etr_class_relationship(A,B,C), not
      #member(etr_class_relationship(A,B,C), Y), #last(Y,E),
      etr_class_relationship(A,B,C)>E.
```

Se cuentan cuántas relaciones tiene un clase originalmente

```
etr_amount_of_class_relationships(A,B):-
      #count{I : etr_class_relationship(A,D,I)} = B,
      etr_has_class_relationships(A), etr_class_relationship(A,_,_).
```

Si la clase no tiene relaciones se asigna 0 como la cantidad

```
etr_amount_of_class_relationships(A,0):-
      not etr_has_class_relationships(A), class(A).
```

Cuando la lista de relaciones tiene la misma cantidad de relaciones que había relaciones de esa clase originalmente se tiene entonces la lista final

```
etr_final_class_relationships_list(A,X):-
      etr_class_relationships_list(A,X), #length(X,Y),
      etr_amount_of_class_relationships(A,Y).
```

Revisa si la clase tiene por lo menos una relación

```
etr_has_class_relationships(A):-
      class(A), relation(A,_,_).
```

Transforma las relaciones relacionadas con clases en predicados de relaciones de clase para ser agregadas en una lista.

```
etr_class_relationship(A,B,C):-
      class(A), relation(A,B,C).
```

El último bloque hace una clase con las listas de propiedades, relaciones e instancias antes formadas por el código, dado que existe la categoría de “clase especial” (clases que contiene la información sobre probabilidad recompensa y costo de acciones) estas clases especiales son rasterizadas al mismo tipo de clase para que todo sea tratado de la misma forma en prolog.

Se forma un predicado de clase con toda la información relevante a una clase en particular

```
etr_class(A,B,C,D,E):-
      class(A), parent(A,B), etr_final_class_properties_list(A,C),
      etr_final_class_relationships_list(A,D),
      etr_final_class_instance_list(A,E).
```

Se transforman las clases especiales en clases iguales al resto formado por este código.

```
etr_class(A,B,C,D,E):-
      special_class(A,B,C,D,E).
```

Asi, si se tiene una clase:

```
class("cereal").
parent("cereal",food).
property("cereal",inv,0).
instance("cereal",id,c1).
property(c1,brand,kellogs).
property(c1,original_id,c1).
property(c1,in,storage).
```

Se obtiene el siguiente predicado:

```
etr_class("cereal",food,[etr_class_property("cereal",inv,0)],[],[etr_
class_instance("cereal",etr_operator("cereal",id,c1),etr_final_instan
ce_properties_list("cereal",c1,[etr_instance_property(c1,brand,kellog
s),etr_instance_property(c1,original_id,c1),etr_instance_property(c1,
in,storage)]),etr_final_instance_relationship_list("cereal",c1,[]))])
```

Apéndice 3: Flujo de control

El ciclo de la tarea de la prueba del supermercado llama al módulo de inferencia en *ASP*, Desde el archivo ETR_User_Functions.pl se llama al archivo ETR_P_Preparation_For_Diagnosis.pl en el cual primero se llama al archivo ETR_P_Transform_DB_To_ASP.pl con lo cual se transforma la base golem_KB.txt a golem_KB_helper.txt (por motivos de formato) y después a ETR_A_Prolog_To_ASP.dl, transformando así la base de conocimiento de Golem de una forma de listas anidades en una froma de reglas atómicas que podrán ser interpretadas por DLV.

Una vez que se tiene la base de datos en un formato que DLV puede trabajar, (ETR_A_Prolog_To_ASP.dl) esta se usará en varios puntos del proceso de diagnóstico.

Primero se llama DLV con el código en ETR_A_Preparation.dl y ETR_A_Prolog_To_ASP.dl para con esto generar ETR_A_Diagnosis_World.dl, que contiene la información relevante para hacer el diagnóstico.

Después llamamos a DLV con el código en ETR_A_Diagnosis.dl y ETR_A_Diagnosis_World.dl, y con esto recibimos la lista con el diagnóstico y los cambios a hacer en la base de conocimiento. Los cambios los guardamos en ETR_A_Changes.dl, y hacemos otra llamada a DLV con el código en el archivo ETR_A_Transform_Base_01.dl y los archivos ETR_A_Prolog_To_ASP.dl y ETR_A_Changes.dl,

Lo que recibimos es la base con los cambios de lugar necesarios, esto se guarda en ETR_A_Auxiliary_Base.dl y subsecuentemente hacemos una llamada a DLV con el código en ETR_A_Transform_Base_02.dl y ETR_A_Auxiliary_Base.dl para con esto obtener la base con los cambios con los nombres de predicados correctos, y guardamos el resultado en ETR_A_Return_Base.dl La última llamada a DLV la hacemos con el código en ETR_A_Return.dl y ETR_A_Return_Base.dl para que DLV regrese la base de conocimiento a un formato de listas para cada clase. Finalmente llamamos al código en ETR_P_Return.pl para transformar las listas que generamos en *ASP* a una sola lista anidada en el formato que Golem reconoce (se genera u archivo auxiliar ETR_Trasform_List.txt). El submódulo de diagnóstico entonces devuelve la lista que representa el diagnóstico en el formato que puede ser interpretado por el generador de diálogos para la prueba del supermercado generado por grupo Golem y una lista que representa la base de conocimientos con los cambios necesarios para reflejar las inferencias realizadas por el submódulo. La base de conocimientos se salva para que pueda ser utilizada por los submódulos de decisión y planeación.

Las salidas de los submódulos de Razonamiento y Planeación se unen a la salida de Diagnóstico para pasar al generador de diálogo de Golem con el cual el robot informa sobre cuál fue su diagnóstico, su decisión y su plan para continuar con su tarea. El plan y el diálogo son regresados al modelo de diálogo para entonces continuar con la ejecución de la tarea.

La siguiente gráfica muestra la interacción de las diferentes partes del submódulo de diagnóstico arriba mencionada.

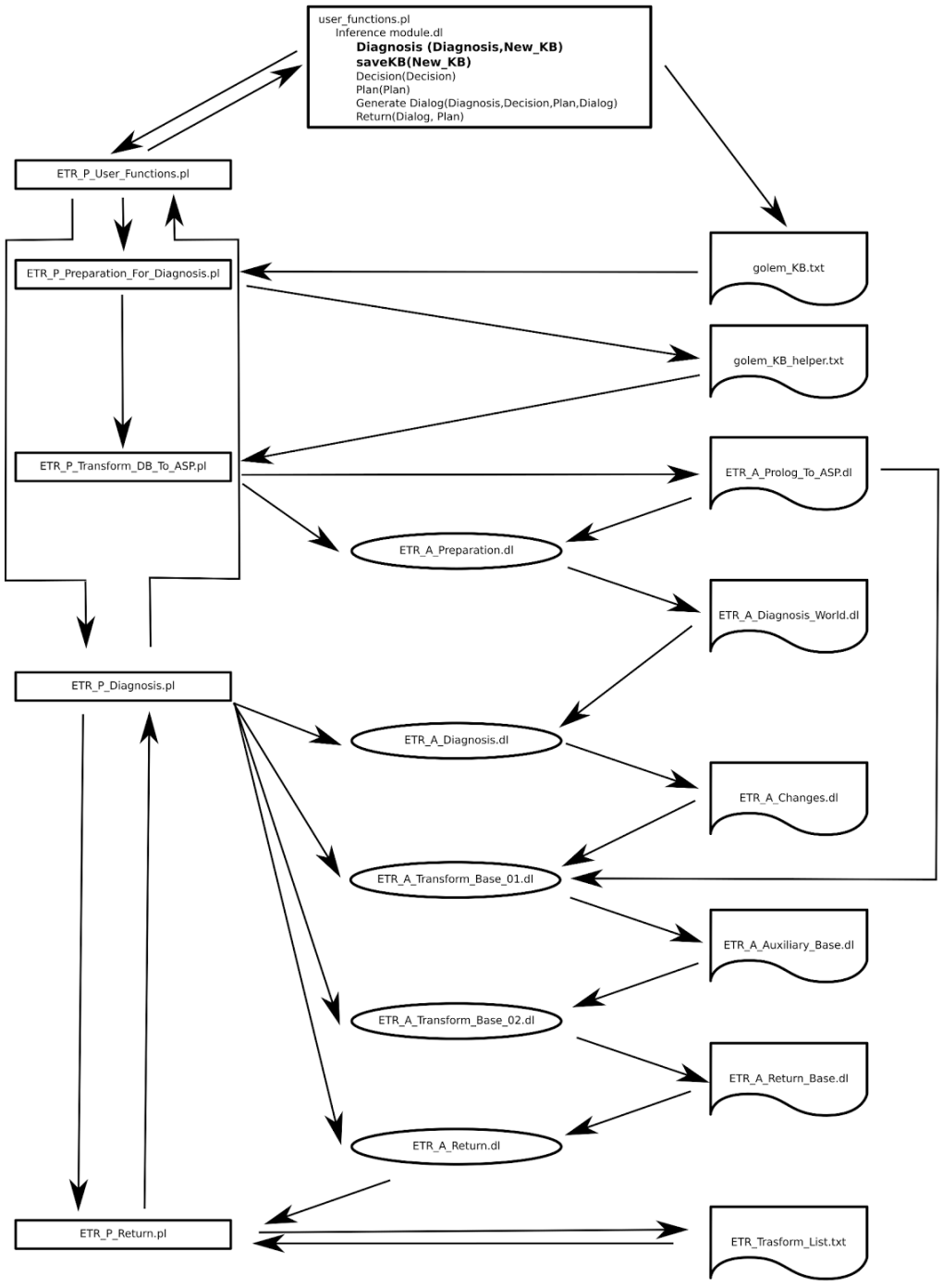

Fig 3.- Diagrama de las interacciones entre los diferentes archivos del subsistema de Diagnóstico, los rectángulos representan programas de control en Prolog, Los óvalos programas en ASP ejecutados en DLV y la salidas representan archivos que se guardan en disco.

La siguiente figura muestra la relaciones en el módulo de inferencia

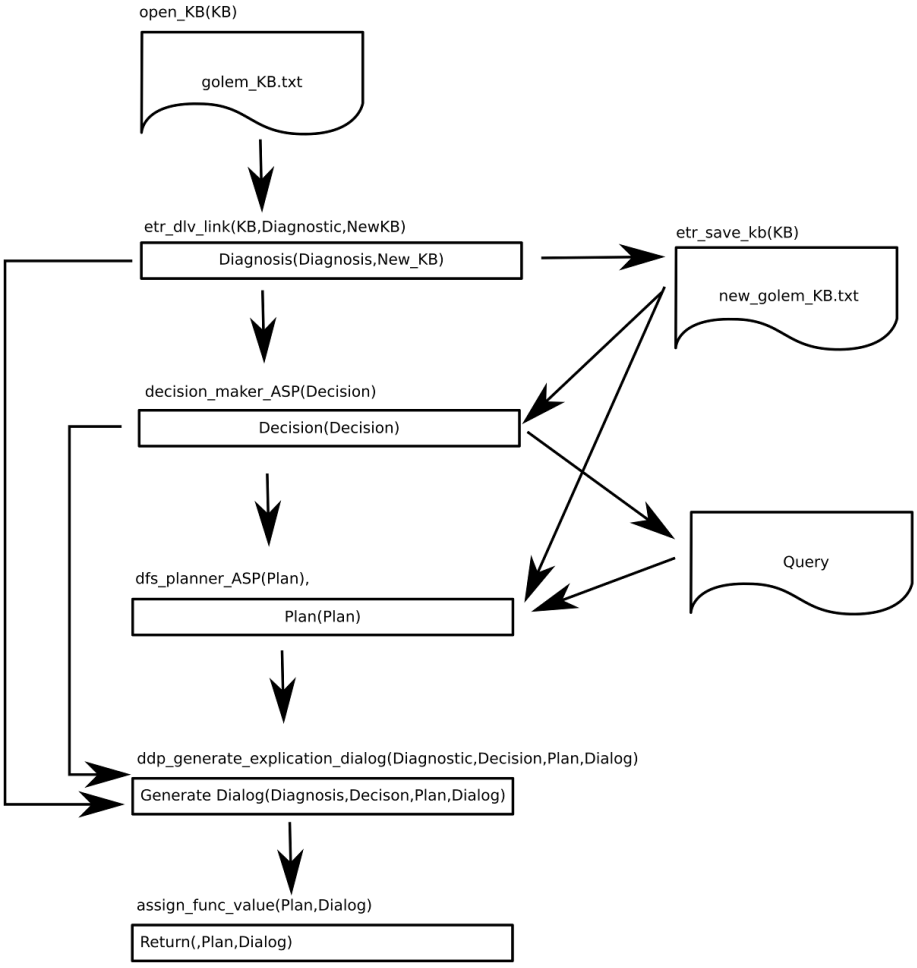

Fig 4.- Diagrama de las relaciones del módulo de inferencia, los rectángulos representan funciones de prolog mientras que las salidas representan archivos guardados en disco.

El módulo de inferencia es llamado para que provea un diagnóstico de cómo es que el mundo se encuentra en el estado presente así como un plan para realizar las órdenes requeridas, este plan depende de lo que golem decida hacer (entre entregar el producto y/o reacomodar artículos que esté fuera de lugar). La inferencia en diagnóstico, la decisión y la planeación se hacen empleando *Answer Set Programming* en el motor DLV.

La siguiente figura representa los modelos de diálogo que codifican la tarea que resuelve la prueba del restaurante

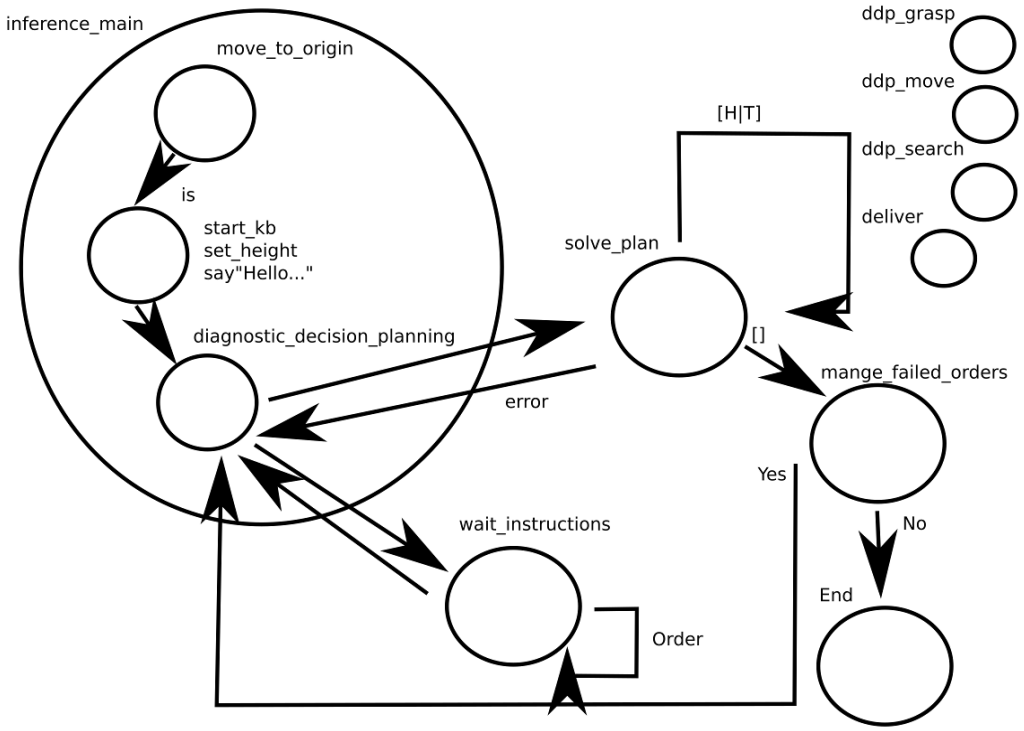

Fig 5.- Diagrama de la relación de los modelos de diálogo que conforman la prueba del supermercado.

Apéndice 4: Comparación de bases de conocimiento

La base de conocimiento original es:

```
[
class(top,none,[],[],[]),
class(object,top,[],[],[]),
class(comestible,object,[graspable,not(on_discount)],[],[]),
class(food,comestible,[shelf=>shelf2],[],[]),
class(cereal,food,[inv=>0],[],[[id=>c1,[brand=>kellogs,in=>storage,
     original_id=>c1],[]]]),
class('instant food',food,[inv=>0],[],[[id=>i1,[brand=>noodles,
     in=>storage,original_id=>i1],[]]]),
class(drink,comestible,[age=>all,shelf=>shelf1],[],[]),
class(soda,drink,[inv=>0],[],[[id=>s1,[brand=>coke,in=>storage,
     original_id=>s1],[]]]),
class(beer,drink,[age=>18,inv=>0],[],[[id=>b1,[brand=>heineken,
     in=>storage,original_id=>b1],[]]]),
class(bread,comestible,[inv=>0,shelf=>shelf3],[],[[id=>d1,
     [brand=>bisquits,on_discount,in=>storage,original_id=>d1],[]]]),
class(point,top,[],[],[[id=>start,[name=>start],[]],[id=>shelf1,
     [name=>'the shelf of drinks'],[]],[id=>shelf2,[name=>'the shelf
     of food'],[]],[id=>shelf3,[name=>'the shelf of bread'],[]]]),
class(robot,top,[],[],[[id=>golem,[position=>start],[]]]),
class(states,top,[],[],[]),
class(events,top,[],[],[]),
class(dialog,events,[],[],[[id=>dialog1,[said_by=>assistant,
     content=>[]],[]]]),class(observation,events,[],[],[]),
class('grasp object',events,[],[],[]),
class(orders,top,[],[],[]),
class(failed_orders,top,[],[],[]),
class(actions,top,[],[],[]),class(move,actions,[p=>0.95,c=>30,r=>0],
     [],[[id=>move1,[p=>1.0,c=>0],[from=>start,to=>start]],[id=>move2
     ,[],[from=>start,to=>shelf1]],[id=>move3,[],[from=>start,to=>she
     lf2]],[id=>move4,[],[from=>start,to=>shelf3]],[id=>move5,[],[fro
     m=>shelf1,to=>start]],[id=>move6,[p=>1.0,c=>0],[from=>shelf1,to=
     >shelf1]],[id=>move7,[],[from=>shelf1,to=>shelf2]],[id=>move8,[]
     ,[from=>shelf1,to=>shelf3]],[id=>move9,[],[from=>shelf2,to=>star
     t]],[id=>move10,[],[from=>shelf2,to=>shelf1]],[id=>move11,[p=>1.
     0,c=>0],[from=>shelf2,to=>shelf2]],[id=>move12,[],[from=>shelf2,
     to=>shelf3]],[id=>move13,[],[from=>shelf3,to=>start]],[id=>move1
     4,[],[from=>shelf3,to=>shelf1]],[id=>move15,[],[from=>shelf3,to=
```

```
      >shelf2]],[id=>move16,[p=>1.0,c=>0],[from=>shelf3,to=>shelf3]]])
      ,
class(search,actions,[p=>0.85,c=>10,r=>0],[],[[id=>search1,[],
      [to=>s1]],[id=>search2,[],[to=>c1]],[id=>search3,[],[to=>i1]],[i
      d=>search4,[p=>0.75],[to=>b1]],[id=>search5,[],[to=>d1]]]),
class(grasp,actions,[p=>0.9,c=>30,r=>100],[],[[id=>grasp1,[],[to=>s1]
      ],[id=>grasp2,[],[to=>c1]],[id=>grasp3,[],[to=>i1]],[id=>grasp4,
      [p=>0.15],[to=>b1]],[id=>grasp5,[],[to=>d1]]]),class(deliver,act
      ions,[p=>0.99,c=>10,r=>500],[],[[id=>deliver1,[],[to=>s1]],[id=>
      deliver2,[],[to=>c1]],[id=>deliver3,[],[to=>i1]],[id=>deliver4,[
      ],[to=>b1]],[id=>deliver5,[],[to=>d1]]])
]
```

Las bases de conocimiento finales son:
(Grupo Golem)

```
[
class(top,none,[],[],[]),
class(object,top,[],[],[]),
class(comestible,object,[graspable,not(on_discount)],[],[]),
class(food,comestible,[shelf=>shelf2],[],[]),
class(cereal,food,[inv=>0],[],[[id=>c1,[brand=>kellogs,
      original_id=>c1,in=>shelf2],[]]]),
class('instant food',food,[inv=>1],[],[[id=>i1,
      [brand=>noodles,original_id=>i1,in=>shelf2],[]]]),
class(drink,comestible,[age=>all,shelf=>shelf1],[],[]),
class(soda,drink,[inv=>1],[],[[id=>s1,[brand=>coke,original_id=>s1,
      in=>shelf3],[]]]),
class(beer,drink,[age=>18,inv=>1],[],[[id=>b1,[brand=>heineken,
      original_id=>b1,in=>shelf1],[]]]),
class(bread,comestible,[shelf=>shelf3,inv=>1],[],[[id=>d1,
      [brand=>bisquits,on_discount,original_id=>d1,in=>shelf3],[]]]),
class(point,top,[],[],[[id=>start,[name=>start],[]],[id=>shelf1
      name=>'the shelf of drinks'],[]],[id=>shelf2,[name=>'the shelf
      of food'],[]],[id=>shelf3,[name=>'the shelf of bread'],[]]]),
class(robot,top,[],[],[[id=>golem,[position=>shelf2],[]]]),
class(states,top,[],[],[]),
class(events,top,[],[],[]),
class(dialog,events,[],[],[[id=>dialog1,[said_by=>assistant,
      content=>[bring(coke,shelf1),bring(heineken,shelf1),bring(noodle
      s,shelf2),bring(bisquits,shelf3)]],[]]]),
class(observation,events,[],[],[[id=>observation(shelf1),
```

```
      [seen_by=>robot,observed_objects=>[heineken,noodles]],[]],[id=>o
      bservation(shelf2),[seen_by=>robot,observed_objects=>[kellogs]],
      []]]),
class('grasp object',events,[],[],[[id=>grasp(noodles),[],[]]]),
class(orders,top,[],[],[[id=>bring(coke),[],[]]]),
class(failed_orders,top,[],[],[[id=>bring(kellogs),
      [reason=>not_disponible],[]]]),
class(actions,top,[],[],[]),
class(move,actions,[p=>0.95,c=>30,r=>0],[],[[id=>move1,[p=>1.0,c=>0],
      [from=>start,to=>start]],[id=>move2,[],[from=>start,to=>shelf1]]
      ,[id=>move3,[],[from=>start,to=>shelf2]],[id=>move4,[],[from=>st
      art,to=>shelf3]],[id=>move5,[],[from=>shelf1,to=>start]],[id=>mo
      ve6,[p=>1.0,c=>0],[from=>shelf1,to=>shelf1]],[id=>move7,[],[from
      =>shelf1,to=>shelf2]],[id=>move8,[],[from=>shelf1,to=>shelf3]],[
      id=>move9,[],[from=>shelf2,to=>start]],[id=>move10,[],[from=>she
      lf2,to=>shelf1]],[id=>move11,[p=>1.0,c=>0],[from=>shelf2,to=>she
      lf2]],[id=>move12,[],[from=>shelf2,to=>shelf3]],[id=>move13,[],[
      from=>shelf3,to=>start]],[id=>move14,[],[from=>shelf3,to=>shelf1
      ]],[id=>move15,[],[from=>shelf3,to=>shelf2]],[id=>move16,[p=>1.0
      ,c=>0],[from=>shelf3,to=>shelf3]]]),
class(search,actions,[p=>0.85,c=>10,r=>0],[],[[id=>search1,[],
      [to=>s1]],[id=>search2,[],[to=>c1]],[id=>search3,[],[to=>i1]],[i
      d=>search4,[p=>0.75],[to=>b1]],[id=>search5,[],[to=>d1]]]),
class(grasp,actions,[p=>0.9,c=>30,r=>100],[],[[id=>grasp1,[],[to=>s1]
      ],[id=>grasp2,[],[to=>c1]],[id=>grasp3,[],[to=>i1]],[id=>grasp4,
      [p=>0.15],[to=>b1]],[id=>grasp5,[],[to=>d1]]]),class(deliver,act
      ions,[p=>0.99,c=>10,r=>500],[],[[id=>deliver1,[],[to=>s1]],[id=>
      deliver2,[],[to=>c1]],[id=>deliver3,[],[to=>i1]],[id=>deliver4,[
      ],[to=>b1]],[id=>deliver5,[],[to=>d1]]])
]
```

(*Answer Set Programming*)

```
[
class(comestible,object,[graspable,not(on_discount)],[],[]),
class(food,comestible,[shelf=>shelf2],[],[]),
class(cereal,food,[inv=>0],[],[[id=>c1,[brand=>kellogs,
      original_id=>c1,in=>shelf2],[]]]),
class('instant food',food,[inv=>1],[],[[id=>i1,
      [brand=>noodles,original_id=>i1,in=>shelf2],[]]]),
class(drink,comestible,[shelf=>shelf1,age=>all],[],[]),
class(soda,drink,[inv=>1],[],[[id=>s1,[brand=>coke,
      original_id=>s1,in=>shelf3],[]]]),
```

```
class(beer,drink,[inv=>1,age=>18],[],[[id=>b1,[in=>shelf1,
     original_id=>b1,brand=>heineken],[]]]),
class(bread,comestible,[shelf=>shelf3,inv=>1],[],[[id=>d1,
     [in=>shelf3,original_id=>d1,on_discount,brand=>bisquits],[]]]),
class(top,none,[],[],[]),
class(object,top,[],[],[]),
class(point,top,[],[],[[id=>shelf2,[name=>'the shelf of food'],[]],
     [id=>shelf1,[name=>'the shelf of
     drinks'],[]],[id=>shelf3,[name=>'the shelf of
     bread'],[]],[id=>start,[name=>start],[]]]),
class(robot,top,[],[],[[id=>golem,[position=>shelf2],[]]]),
class(states,top,[],[],[]),
class(events,top,[],[],[]),
class(dialog,events,[],[],[[id=>dialog1,[content=>[bring(coke,shelf1)
     ,bring(heineken,shelf1),bring(noodles,shelf2),bring(bisquits,she
     lf3)],[],said_by=>assistant],[]]]),
class(observation,events,[],[],[[id=>observation(shelf1),
     [seen_by=>robot,observed_objects=>[heineken,noodles]],[]],[id=>o
     bservation(shelf2),[seen_by=>robot,observed_objects=>[kellogs]],
     []]]),
class('grasp object',events,[],[],[[id=>grasp(noodles),[],[]]]),
class(orders,top,[],[],[[id=>bring(coke),[],[]]]),
class(failed_orders,top,[],[],[[id=>bring(kellogs),
     [reason=>not_disponible],[]]]),
class(actions,top,[],[],[]),
class(move,actions,[p=>0.95,c=>30,r=>0],[],[[id=>move1,[p=>1.0,c=>0],
     [from=>start,to=>start]],[id=>move2,[],[from=>start,to=>shelf1]]
     ,[id=>move3,[],[from=>start,to=>shelf2]],[id=>move4,[],[from=>st
     art,to=>shelf3]],[id=>move5,[],[from=>shelf1,to=>start]],[id=>mo
     ve6,[p=>1.0,c=>0],[from=>shelf1,to=>shelf1]],[id=>move7,[],[from
     =>shelf1,to=>shelf2]],[id=>move8,[],[from=>shelf1,to=>shelf3]],[
     id=>move9,[],[from=>shelf2,to=>start]],[id=>move10,[],[from=>she
     lf2,to=>shelf1]],[id=>move11,[p=>1.0,c=>0],[from=>shelf2,to=>she
     lf2]],[id=>move12,[],[from=>shelf2,to=>shelf3]],[id=>move13,[],[
     from=>shelf3,to=>start]],[id=>move14,[],[from=>shelf3,to=>shelf1
     ]],[id=>move15,[],[from=>shelf3,to=>shelf2]],[id=>move16,[p=>1.0
     ,c=>0],[from=>shelf3,to=>shelf3]]]),
class(search,actions,[p=>0.85,c=>10,r=>0],[],[[id=>search1,[],
     [to=>s1]],[id=>search2,[],[to=>c1]],[id=>search3,[],[to=>i1]],[i
     d=>search4,[p=>0.75],[to=>b1]],[id=>search5,[],[to=>d1]]]),
class(grasp,actions,[p=>0.9,c=>30,r=>100],[],[[id=>grasp1,[],[to=>s1]
     ],[id=>grasp2,[],[to=>c1]],[id=>grasp3,[],[to=>i1]],[id=>grasp4,
     [p=>0.15],[to=>b1]],[id=>grasp5,[],[to=>d1]]]),class(deliver,act
```

```
      ions,[p=>0.99,c=>10,r=>500],[],[[id=>deliver1,[],[to=>s1]],[id=>
      deliver2,[],[to=>c1]],[id=>deliver3,[],[to=>i1]],[id=>deliver4,[
      ],[to=>b1]],[id=>deliver5,[],[to=>d1]]])
]
```

Apéndice 5: *Answer Set Programming*

ASP se basa en generar conjuntos de cláusulas lógicas que son coherentes entre sí.

Así, si se tienen las cláusulas *A*, *B* y *C*; y *A* y *B* son coherentes entre sí pero *C* no es coherente con *A*; y *B* y *C* son coherentes entre sí pero *A* no es coherente con *C*, entonces contamos con 2 conjuntos solución; el conjunto de cláusulas *{A, B}* y aparte el conjunto *{B, C}*.

Es claro que si *A* y *C* no son coherentes entre sí, estas no pueden aparecer en un conjunto solución. Es entonces que se necesita una manera de que los elementos *A* y *C* sean posibles en el mundo pero no al mismo tiempo.

Para resolver esto *ASP* emplea disyunción en la declaración de sus reglas.

Si pensamos en el conjunto *A*, *B*, *C*, es claro que este conjunto no puede generar soluciones pues *A* y *C* no son coherentes entre sí (digamos entonces que *C* = *-A* (no *A*) por simplicidad) así es más claro ver que *A*, *B*, *-A* no pueden proveer una solución ya que el conjunto contiene una contradicción.

Empleando la declaración disyuntiva podemos entonces declarar dos reglas

```
A v -A. (A o no A)
B.
```

Esto se traduce en que se generará la regla en dos ocasiones, una vez como *A* y otra vez como *-A*.

Como *B* es compatible con *A* y es compatible con *-A* tenemos 2 conjuntos solución (Answer Sets).

{A,B}
{-A,B}

Cada una de las respuestas representa un conjunto solución diferente.

Otra forma de explorar posibilidades dentro de un mismo conjunto solución es la de la unificación. A través de la unificación se puede generar una regla que tome varios valores.

Ejemplo

```
clausula(a).
clausula(b).
clausula(c).
```

```
regla(X):- clausula(X).
```

Así el conjunto generado como respuesta es

{clausula(a), clausula(b), clausula(c), regla(a), regla(b), regla(c)}

La regla "regla" toma los valores de todas las posibilidades de "cláusulas" que hay en el mundo (*a,b,c*).

Otra consideración importante al generar conjuntos de respuestas es el uso de negación. Se pueden usar dos tipos de negación; negación fuerte para aquellos hechos que niegan directamente otros hechos (como en el caso de *A* y *-A*); y negación suave para aquellos hechos que no se pueden comprobar como ciertos (como *not(A)*).

En el siguiente programa de ejemplo se muestra la utilización de la negación fuerte (-). Este programa genera las posibles amistades de juan entre las personas disponibles en el mundo, es importante notar que Juan no puede ser su propio amigo.

```
persona(juan).
persona(pepe).
persona(raul).

amigo_de_juan(X) v -amigo_de_juan(X):- persona(X), X!=juan.
```

{persona(juan), persona(pepe), persona(raul), -amigo_de_juan(pepe), -amigo_de_juan(raul)}
{persona(juan), persona(pepe), persona(raul), amigo_de_juan(pepe), -amigo_de_juan(raul)}
{persona(juan), persona(pepe), persona(raul), -amigo_de_juan(pepe), amigo_de_juan(raul)}
{persona(juan), persona(pepe), persona(raul), amigo_de_juan(pepe), amigo_de_juan(raul)}

Las combinaciones de amigos y no amigos son parte del conjunto así como las personas que integran el mundo; más adelante se explica cómo filtrar los conjuntos resultantes para que se nos proporcione sólo la información que nos interesa.

El siguiente programa muestra cómo usar la negación suave (not), notar que la regla "amigos_que_le_quedan_a_juan_si_raul_no_es_su_amigo" solo se activará en aquellos conjuntos de respuesta en los que no sea posible encontrar a Raúl como amigo de Juan y contenga por lo menos un amigo de Juan.

```
persona(juan).
persona(pepe).
persona(raul).
```

```
amigo_de_juan(X) v -amigo_de_juan(X):- persona(X), X!=juan.

amigos_que_le_quedan_a_juan_si_raul_no_es_su_amigo(X):-
      amigo_de_juan(X), not amigo_de_juan(raul).
```

{persona(juan), persona(pepe), persona(raul), amigo_de_juan(pepe), -amigo_de_juan(raul), amigos_que_le_quedan_a_juan_si_raul_no_es_su_amigo(pepe)}
{persona(juan), persona(pepe), persona(raul), -amigo_de_juan(pepe), -amigo_de_juan(raul)}
{persona(juan), persona(pepe), persona(raul), -amigo_de_juan(pepe), amigo_de_juan(raul)}
{persona(juan), persona(pepe), persona(raul), amigo_de_juan(pepe), amigo_de_juan(raul)}

Hasta ahora los conjuntos solución siguen conteniendo los hecho iniciales (las personas) y las reglas intermedias (los amigos) cuando lo que nos interesa es saber cuáles son los amigos de Juan si Raúl no es su amigo.

Podemos fijarnos de esos conjuntos de respuestas sólo en las reglas que queremos aplicando filtros así, si decidimos obtener información sobre los amigos que le quedan a Juan si Raul no es su amigo obtenemos:

{amigos_que_le_quedan_a_juan_si_raul_no_es_su_amigo(pepe)}
{}
{}
{}

Es importante notar que aparecen conjuntos vacíos cuando hay combinaciones de los hechos o reglas intermedias que no se piden en el filtrado, para remover esos conjuntos vacíos es necesario utilizar restricciones (se explican adelante).

Se pueden eliminar conjuntos solución completos que contengan ciertas reglas utilizando restricciones, notar que utilizando la restricción "`:- amigo_de_juan(raul).`" se eliminan los conjuntos en los que raul aparece como amigo de juan. La interpretación de la restricción es que dado que la cabeza de la regla está vacía esta es falsa por lo que es falso lo que contenga la regla. Al ser falso no se puede generar un conjunto solución con esta por lo que si se activa la restricción (por existir los elementos necesarios dentro del conjunto solución tentativo) la restricción se cumple y se elimina el conjunto entero.

```
persona(juan).
persona(pepe).
persona(raul).

amigo_de_juan(X) v -amigo_de_juan(X):- persona(X), X!=juan.
```

```
amigos_que_le_quedan_a_juan_si_raul_no_es_su_amigo(X):-
      amigo_de_juan(X), not amigo_de_juan(raul).

      :- amigo_de_juan(raul).
```

{persona(juan), persona(pepe), persona(raul), amigo_de_juan(pepe), -amigo_de_juan(raul), amigos_que_le_quedan_a_juan_si_raul_no_es_su_amigo(pepe)}
{persona(juan), persona(pepe), persona(raul), -amigo_de_juan(pepe), -amigo_de_juan(raul)}

También se puede priorizar los resultados de forma que ciertos resultados sean mejores que otros y así quedarse solo con aquellos que tengan la menor cantidad de características indeseables usando restricciones suaves (se puede también dar pesos diferentes a cada característica para que haya alguna menos deseables que otras)

Notar cómo aunque las restricciones suaves "`:~ -amigo_de_juan(paco).[1:1]`" y "`:~ -amigo_de_juan(abel).[1:1]`" asignan una penalización en caso de paco o abel no sean amigos de juan, la restricción suave "`:~ amigo_de_juan(abel), amigo_de_juan(paco).[10:1]`" es mucho más fuerte por lo que aunque conlleva una penalización, los conjuntos seleccionados no tienen a paco y a abel como amigos de juan al mismo tiempo. Notar también que no aparece la opción en que solo pepe es amigo de juan, esto sucede pues esa opción tendría un costo de 2 (ya que paco y abel aparecen como no amigos de juan) y dado que hay opciones con costo de 1 se muestran esas solamente.

```
persona(juan).
persona(pepe).
persona(raul).
persona(abel).
persona(paco).

amigo_de_juan(X) v -amigo_de_juan(X):- persona(X), X!=juan.

amigos_que_le_quedan_a_juan_si_raul_no_es_su_amigo(X):-
amigo_de_juan(X), not amigo_de_juan(raul).

      :- amigo_de_juan(raul).
      :~ -amigo_de_juan(paco).[1:1]
      :~ -amigo_de_juan(abel).[1:1]
      :~ amigo_de_juan(abel), amigo_de_juan(paco).[10:1]
```

Best model: {persona(juan), persona(pepe), persona(raul), persona(abel), persona(paco), amigo_de_juan(pepe), -amigo_de_juan(raul), amigo_de_juan(abel), -amigo_de_juan(paco),

amigos_que_le_quedan_a_juan_si_raul_no_es_su_amigo(pepe),
amigos_que_le_quedan_a_juan_si_raul_no_es_su_amigo(abel)}
Cost ([Weight:Level]): <[1:1]>

Best model: {persona(juan), persona(pepe), persona(raul), persona(abel), persona(paco), -amigo_de_juan(pepe), -amigo_de_juan(raul), amigo_de_juan(abel), -amigo_de_juan(paco), amigos_que_le_quedan_a_juan_si_raul_no_es_su_amigo(abel)}
Cost ([Weight:Level]): <[1:1]>

Best model: {persona(juan), persona(pepe), persona(raul), persona(abel), persona(paco), amigo_de_juan(pepe), -amigo_de_juan(raul), -amigo_de_juan(abel), amigo_de_juan(paco), amigos_que_le_quedan_a_juan_si_raul_no_es_su_amigo(pepe),
amigos_que_le_quedan_a_juan_si_raul_no_es_su_amigo(paco)}
Cost ([Weight:Level]): <[1:1]>

Best model: {persona(juan), persona(pepe), persona(raul), persona(abel), persona(paco), -amigo_de_juan(pepe), -amigo_de_juan(raul), -amigo_de_juan(abel), amigo_de_juan(paco), amigos_que_le_quedan_a_juan_si_raul_no_es_su_amigo(paco)}
Cost ([Weight:Level]): <[1:1]>

Utilizando entonces, Cabezas disyuntivas, unificación, negación suave y dura así como restricciones duras y suaves, se pueden "explorar" las diferentes posibilidades de generación de conjuntos solución (Answer Sets) a manera de diagnósticos (para el programa final se utilizan otras características del motor de *ASP* tales como manejo de listas y operaciones aritméticas). En el caso de la pérdida de situación se exploran las diferentes combinaciones de acciones que permiten explicar cómo es que el mundo inicial se fue transformando paulatinamente en el mundo observado por la acciones realizadas por el agente..

Apéndice 6: Diagnóstico en *ASP*

A continuación se presenta un ejemplo de programa que diagnostica la secuencia de acciones necesarias para ir de un mundo a otro utilizando todos los elementos discutidos anteriormente incluyendo precondiciones, poscondiciones y fluents. Este código emplea las restricciones suaves para garantizar que se obtengan las acciones más cercanas a 0 posible (así evitando que se generen diagnósticos no mínimos) en caso de que se pide calcular las opciones para un número de pasos mayor al mínimo (calcular este mínimo no entra en este ejemplo pero si en el código utilizado en el trabajo final). En caso de que se pida calcular para un número mayor se generarán varias respuestas iguales. Para 5 pasos se genera 1 respuesta, para 6 se genera 3, para 7 pasos 9, etc… (estas respuestas no son realmente idénticas, lo que es idéntico es la secuencia de acciones que proponen, si se analiza el resto de la respuesta se verán diferencias en fluents o reglas intermedias).

La parte de los datos (en el caso del ejemplo anterior) se ve de la siguiente forma:

```
% Bases
agent(golem).
item(heiniken).
location(start).
location(shelf_1).
location(storage).

% Initial world
f_at(0,golem,start).
f_on(0,storage,heiniken).

% Observed world
goal_reached :-
      f_at(#maxint,golem,start),
      f_on(#maxint,shelf_1,heiniken).
```

La parte del programa (En el caso del ejemplo anterior) se ve de la siguiente manera:

```
% Rules
time(T):- #int(T).
next(T1,T2):- time(T1), time(T2), T2 = T1 + 1.

      :- not goal_reached.
      :~ a_goto(X,_,_).[X:1]
      :~ a_take(X,_,_,_).[X:1]
      :~ a_deliver(X,_,_,_).[X:1]
```

```
% Actions
a_goto(T0,A,L1) :-
     f_at(T0,A,L0),                          %(Precondición)
     f_at(T1,A,L1),                          %(Postcondición)
     time(T0), time(T1), next(T0,T1),
     agent(A),
     location(L0), location(L1), L0!=L1.

a_take(T0,A,I,L) :-
     f_at(T0,A,L),  f_on(T0,L,I),            %(Precondición)
     f_at(T1,A,L),  not f_on(T1,L,I),        %(Postcondición)
     time(T0), time(T1), next(T0,T1),
     agent(A), item(I), location(L).

a_deliver(T0,A,I,L) :-
     f_at(T0,A,L),  not f_on(T0,L,I),        %(Precondición)
     f_at(T1,A,L),  f_on(T1,L,I),            %(Postcondición)
     time(T0), time(T1), next(T0,T1),
     agent(A), item(I), location(L).

action(T,A,goto) v action(T,A,take) v action(T,A,deliver) :-
     time(T), agent(A), T!=#maxint.

     :- not action(T,A,goto), a_goto(T,A,L), time(T), agent(A),
     location(L).

     :- not action(T,A,take), a_take(T,A,I,L), time(T), agent(A),
     item(I), location(L).

     :- not action(T,A,deliver), a_deliver(T,A,I,L), time(T),
     agent(A), item(I), location(L).

     :- a_goto(T,A,L0), a_goto(T,A,L1), time(T), agent(A),
     location(L0), location(L1), L0!=L1.

% Things that change in time (fluents)
f_at(T1,A,L1) v -f_at(T1,A,L1):- f_at(T0,A,L0), time(T0), time(T1),
next(T0,T1), agent(A), location(L1), location(L0).

     :- f_at(T,A,L0), f_at(T,A,L1), time(T), agent(A), location(L0),
     location(L1), L0!=L1.
```

```
:- f_at(T0,A,L0), f_at(T1,A,L1), not a_goto(T0,A,L1), time(T0),
time(T1), next(T0,T1), agent(A), location(L0), location(L1),
L0!=L1.

:- f_at(T0,A,L0), f_at(T1,A,L0), a_goto(T0,A,L1), time(T0),
time(T1), next(T0,T1), agent(A), location(L0), location(L1),
L0!=L1.
```

f_on(T1,L,I1) v -f_on(T1,L,I1):- `time(T0), time(T1), next(T0,T1), item(I0), item(I1), location(L).`

```
:- f_on(T,L0,I), f_on(T,L1,I), time(T), location(L0),
location(L1), L0!=L1, item(I).

:- -f_on(T,L,I), f_on(T,L,I), time(T), location(L), item(I).

:- not f_on(T,L,I), f_on(T,L,I), time(T), location(L), item(I).

:- f_on(T0,L,I), f_on(T1,L,I), a_take(T0,A,I,L), time(T0),
time(T1), next(T0,T1), agent(A), item(I), location(L).

:- -f_on(T0,L,I), -f_on(T1,L,I), a_deliver(T0,A,I,L), time(T0),
time(T1), next(T0,T1), agent(A), item(I), location(L), arm(R).

:- f_on(T0,L,I), -f_on(T1,L,I), not a_take(T0,A,I,L),  time(T0),
time(T1), next(T0,T1), agent(A), item(I), location(L).

:- -f_on(T0,L,I), f_on(T1,L,I), not a_deliver(T0,A,I,L),
time(T0), time(T1), next(T0,T1), agent(A), item(I), location(L).
```

Este es el estado de todos los predicados.

{
agent(golem),
item(heiniken),
location(start),
location(shelf_1),
location(storage),

time(0), time(1), time(2), time(3), time(4), time(5),
next(0,1), next(1,2), next(2,3), next(3,4), next(4,5),

f_at(0,golem,start),

f_on(0,storage,heiniken),

action(0,golem,goto),
a_goto(0,golem,storage),

-f_at(1,golem,start),
-f_at(1,golem,shelf_1),
f_at(1,golem,storage),
-f_on(1,start,heiniken),
-f_on(1,shelf_1,heiniken),
f_on(1,storage,heiniken),

action(1,golem,take),
a_take(1,golem,heiniken,storage),

-f_at(2,golem,start),
-f_at(2,golem,shelf_1),
f_at(2,golem,storage),
-f_on(2,start,heiniken),
-f_on(2,shelf_1,heiniken),
-f_on(2,storage,heiniken),

action(2,golem,goto),
a_goto(2,golem,shelf_1),

-f_at(3,golem,start),
f_at(3,golem,shelf_1),
-f_at(3,golem,storage),
-f_on(3,start,heiniken),
-f_on(3,shelf_1,heiniken),
-f_on(3,storage,heiniken),

action(3,golem,deliver),
a_deliver(3,golem,heiniken,shelf_1),

-f_at(4,golem,start),
f_at(4,golem,shelf_1),
-f_at(4,golem,storage),
-f_on(4,start,heiniken),
f_on(4,shelf_1,heiniken),
-f_on(4,storage,heiniken),

action(4,golem,goto),

a_goto(4,golem,start)

f_at(5,golem,start),
-f_at(5,golem,shelf_1),
-f_at(5,golem,storage),
-f_on(5,start,heiniken),
f_on(5,shelf_1,heiniken),
-f_on(5,storage,heiniken),

goal_reached,
}

Si filtramos la salida para tener solamente los predicados positivos para acciones y fluents (-pfilter=f_at,f_on,a_goto,a_take,a_deliver).

{
f_at(0,golem,start),
f_on(0,storage,heiniken),

a_goto(0,golem,storage),

f_at(1,golem,storage),
f_on(1,storage,heiniken),

a_take(1,golem,heiniken,storage),

f_at(2,golem,storage),

a_goto(2,golem,shelf_1),

f_at(3,golem,shelf_1),

a_deliver(3,golem,heiniken,shelf_1),

f_at(4,golem,shelf_1),
f_on(4,shelf_1,heiniken),

a_goto(4,golem,start),

f_at(5,golem,start),
f_on(5,shelf_1,heiniken)
}

Si filtramos la salida para tener solamente los predicados que permiten realizar el diagnóstico de cómo es que la heineken terminó en la repisa 1 cuando había comenzado en el almacén (-pfilter=a_goto,a_take,a_deliver).

{
a_goto(0,golem,storage),
a_take(1,golem,heiniken,storage),
a_goto(2,golem,shelf_1),
a_deliver(3,golem,heiniken,shelf_1),
a_goto(4,golem,start)
}

Bibliografía

[1] Allwood, J., Andersson, L. G., & Dahl, O. (1977). Logic in linguistics. Cambridge University Press.

[2] Bihlmeyer, R., Faber, W., Ielpa, G, DLV User - Manual retrieved from http://www.dlvsystem.com/html/DLV_User_Manual.html.

[3] Bonatti, P., Calimeri, F., Leone, N., & Ricca, F. (2010, January). Answer set programming. In A 25-year perspective on logic programming (pp. 159-182). Springer-Verlag.

[4] Bongard, J. C., & Lipson, H. (2004, April). Automated damage diagnosis and recovery for remote robotics. In Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on (Vol. 4, pp. 3545-3550). IEEE.

[5] Brewka, G., Eiter, T., & Truszczyński, M. (2011). Answer set programming at a glance. Communications of the ACM, 54(12), 92-103.

[6] Buccafurri, F., Faber, W., & Leone, N. (2002). Disjunctive logic programs with inheritance. TPLP, 2(3), 293-321.

[7] Eiter, T., Faber, W., Leone, N., & Pfeifer, G. (1999). The diagnosis frontend of the dlv system. AI Communications, 12(1-2), 99-111.

[8] Eiter, T., Faber, W., Leone, N., Pfeifer, G., & Polleres, A. (2003). A logic programming approach to knowledge-state planning, II: The DLVK system.Artificial Intelligence, 144(1), 157-211.

[9] Eiter, T., Ianni, G., & Krennwallner, T. (2009). Answer set programming: A primer. In Reasoning Web. Semantic Technologies for Information Systems(pp. 40-110). Springer Berlin Heidelberg.

[10] Eppe, M., & Bhatt, M. (2013). Narrative based postdictive reasoning for cognitive robotics. arXiv preprint arXiv:1306.0665.

[11] Erdem, E., Aker, E., & Patoglu, V. (2012). Answer set programming for collaborative housekeeping robotics: representation, reasoning, and execution. Intelligent Service Robotics, 5(4), 275-291.

[12] Eshghi, K. (1988, August). Abductive Planning with Event Calculus. In ICLP/SLP (pp. 562-579).

[13] Eshghi, K., & Kowalski, R. A. (1989, June). Abduction Compared with Negation by Failure. In ICLP (Vol. 89, pp. 234-255).

[15] Evans, A. J., Chetty, R., Clarke, B. A., Croul, S., Ghazarian, D. M., Kiehl, T. R., ... & Asa, S. L. (2009). Primary frozen section diagnosis by robotic microscopy and virtual slide telepathology: the University Health Network experience. Human pathology, 40(8), 1070-1081.

[16] Gelfond, M., & Kahl, Y. (2014). Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach. Cambridge University Press.

[17] Huang, J., Fukuda, T., & Matsuno, T. (2008). Model-based intelligent fault detection and diagnosis for mating electric connectors in robotic wiring harness assembly systems. IEEE/ASME Transactions on Mechatronics, 13(1), 86-94.

[18] Janíček, M. (2012). Abductive reasoning for continual dialogue understanding. In New Directions in Logic, Language and Computation (pp. 16-31). Springer Berlin Heidelberg.

[19] Levesque, H. J & Brachman, R,J. (2003). Knowledge Representation and Reasoning.

[20] Ortiz, M., & Šimkus, M. (2012, September). Reasoning and query answering in description logics. In Reasoning Web International Summer School (pp. 1-53). Springer Berlin Heidelberg.

[21] Peischl, B., & Wotawa, F. (2003). Model-based diagnosis or reasoning from first principles. IEEE Intelligent Systems, 18(3), 32-37.

[22] Pineda, L. A., Meza, I. V., & Salinas, L. (2010, November). Dialogue model specification and interpretation for intelligent multimodal HCI. In Ibero-American Conference on Artificial Intelligence (pp. 20-29). Springer Berlin Heidelberg.

[23] Pineda, L. A., Rodriguez, A., Fuentes, G. (2015). A Light Non-Monotonic Knowledge-Base for Service Robots.

[24] Pineda, L. A., Rodríguez, A., Fuentes, G., Rascon, C., & Meza, I. V. (2015). Concept and functional structure of a service robot. International Journal of Advanced Robotic Systems, 12.

[25] Pineda, L. A., Salinas, L., Meza, I. V., Rascon, C., & Fuentes, G. (2013). Sitlog: a programming language for service robot tasks. International Journal of Advanced Robotic Systems, 10.

[26] Pineda, L., Meza, I., Aviles, H., Gershenson, C., Rascón, C., Alvarado, M., & Salinas, L. (2011). IOCA: interaction oriented cognitive architecture. Research in Computer Science, 54, 273-284.

[27] Poole, D. (1989). Explanation and prediction: an architecture for default and abductive reasoning. Computational Intelligence, 5(2), 97-110.

[28] Reiter, R. (1987). A theory of diagnosis from first principles. Artificial intelligence, 32(1), 57-95.

[29] Russell, S. & Norvig, P. (1995). Artificial Intelligence, A modern approach. Prentice-Hall, Egnlewood Cliffs, 25, 27.

[30] Russell, S., & Norvig, P. (2004). Inteligencia Artificial Un Enfoque Moderno.

[31] Sanders Pierce, C. (1974). La ciencia de la semiótica. Buenos Aires, Argentina: Ediciones Nueva Visión, 1-109.

[32] Shanahan, M. (1996). Robotics and the Common Sense Informatic Situation'. In ECAI (pp. 684-688). PITMAN.

[33] Shanahan, M. (1999). The event calculus explained. In Artificial intelligence today (pp. 409-430). Springer Berlin Heidelberg.

[34] Shanahan, M. (2000). An abductive event calculus planner. The Journal of Logic Programming, 44(1), 207-240.

[35] Shanahan, M. (2005). Perception as abduction: Turning sensor data into meaningful representation. Cognitive science, 29(1), 103-134.

[36] Sperber, D. & Mercier, H., (2011). Why do humans reason? Arguments for an argumentative theory. Behavioral and brain sciences, 34(02), 57-74. pp 58.