Universidad Nacional del Sur

Tesis de Doctor en Ciencias de la Computación

Planificación y Formalización de Acciones para Agentes Inteligentes

Diego R. Garcia

Bahía Blanca Argentina

2011

Universidad Nacional del Sur

Tesis de Doctor en Ciencias de la Computación

Planificación y Formalización de Acciones para Agentes Inteligentes

Diego R. Garcia

Bahía Blanca Argentina

2011

Prefacio

Esta Tesis es presentada como parte de los requisitos para optar al grado académico de Doctor en Ciencias de la Computación, de la Universidad Nacional del Sur, y no ha sido presentada previamente para la obtención de otro título en esta Universidad u otras. La misma contiene los resultados obtenidos en investigaciones llevadas a cabo en el Departamento de Ciencias de la Computación, durante el período comprendido entre el 1 de abril de 2004 y el 30 de septiembre de 2011, bajo la dirección del Dr. Guillermo R. Simari, Profesor Titular del Departamento de Ciencias de la Computación.

Diego Ramiro García
drg@cs.uns.edu.ar
Departamento de Ciencias de la Computación
Universidad Nacional del Sur
Bahía Blanca, 17 de octubre de 2011

Agradecimientos

En primer lugar, quiero agradecer a Guillermo Simari, quien me dio la oportunidad de realizar mi formación científica en Ciencias de la Computación bajo su dirección. Esta tesis no hubiera sido posible sin su apoyo y dedicación.

Agradezco a las instituciones que confiaron en mí y me brindaron el apoyo económico para que pueda concretar este trabajo: la Universidad Nacional del Sur, la Agencia Nacional de Promoción Científica y Tecnológica (ANPCyT) y el Concejo Nacional de Investigaciones Científicas y Técnicas (CONICET).

Quiero agradecer a todos los integrantes del Departamento de Ciencias e Ingeniería de la Computación por el ambiente de trabajo. En especial a mis compañeros de la sala de becarios, por estar siempre dispuestos a dar una mano y con quienes compartí la experiencia de realizar una tesis doctoral.

Agradezco a toda mi familia. Principalmente a mis padres, por su amor, su apoyo, su dedicación y esfuerzo en mi formación como persona. Todo lo que soy se lo debo a ellos.

Quiero a agradecer a Luciana, por estar siempre a mi lado y permitirme compartir todo con ella.

Finalmente, el agradecimiento mas grande es para Alejandro García, por estar siempre para darme su apoyo, sobre todo en los momentos difíciles.

Gracias a todos, por todo.

Diego.

Resumen

En esta tesis se define un formalismo que combina acciones y argumentación rebatible para representar dominios y problemas de planificación. Lo novedoso de este formalismo es que permite representar conocimiento acerca del dominio y definir acciones utilizando la Programación en Lógica Rebatible. El conocimiento del dominio permite razonar rebatiblemente acerca de las precondiciones, restricciones y efectos de las acciones, las cuales permiten cambiar el entorno para lograr las metas de un problema de planificación.

Además, se define un nuevo método de planificación que combina técnicas de Planificación de Orden Parcial con Argumentación Rebatible, y permite resolver problemas de planificación definidos utilizando el formalismo de representación antes mencionado. La incorporación de Argumentación Rebatible permite utilizar el conocimiento del dominio para razonar rebatiblemente durante la construcción de un plan.

Los formalismos basados en argumentación rebatible permiten representar y razonar con conocimiento que involucre información incompleta o errónea. Esta característica está presente en muchos aspectos de un proceso de planificación, donde no siempre es posible contar de ante mano con toda la información necesaria para resolver un problema.

Índice general

Capítulo 1

Introducción

Una de las metas de la Inteligencia Artificial es la construcción de agentes autónomos racionales, capaces de actuar en ambientes complejos y dinámicos. Si bien la habilidad de hacer y ejecutar planes es un ingrediente esencial para un agente de este tipo, los problemas de planificación que enfrenta contrastan de manera significativa con los tipos de problemas que son resueltos por los métodos de planificación tradicionales. Esto se debe principalmente a las simplificaciones que se asumen sobre los problemas de planificación para disminuir su complejidad. Muchos avances se han hecho en cuanto a la eficiencia de los algoritmos y en menor medida en cuanto a la expresividad para representar problemas del mundo real.

En particular, los sistemas de planificación tradicionales no realizan ningún tipo de razonamiento acerca de las consecuencias de las acciones. El diseñador es el encargado de razonar y luego volcar el resultado de ese razonamiento en la descripción de las acciones. En ambientes complejos y dinámicos resulta imposible determinar de antemano cuales son las consecuencias de las acciones, porque la información cambia constantemente y dependen de muchos factores. Además, no todos los efectos de una acción son relevantes en el contexto de un problema particular.

En esta tesis se define un formalismo que combina acciones y argumentación rebatible para representar dominios y problemas de planificación. Lo novedoso de este formalismo es que permite representar conocimiento acerca del dominio y definir acciones utilizando la Programación en Lógica Rebatible. Por un lado, el conocimiento y la argumentación rebatible se utilizan para razonar acerca de las precondiciones, restricciones y efectos de

las acciones. Por otra parte, las acciones son utilizadas para cambiar el entorno y poder lograr las metas.

Además, se define un nuevo método de planificación que combina técnicas de Planificación de Orden Parcial con Argumentación Rebatible, y permite resolver problemas de planificación definidos utilizando el formalismo de representación antes mencionado. La incorporación de Argumentación Rebatible permite utilizar el conocimiento para razonar de manera tentativa durante la construcción de un plan.

1.1. Motivación

Para resolver un problema de planificación un agente debe estar provisto de un conjunto adecuado de acciones. La representación de estas acciones, debe considerar todas las precondiciones y efectos que son relevantes para resolver el problema. Consideremos por ejemplo las consecuencias de la acción de encender un fósforo. Un efecto relevante de esta acción podría ser producir fuego. Sin embargo, hay muchas otras consecuencias que pueden ser deducidas de este efecto como producir luz, producir humo, entre otros. Estos efectos podrían ser considerados irrelevantes y no ser incluidos en la representación de la acción. Por otra parte, considerar todos los efectos posibles en la representación de una acción no es una alternativa viable dado que algunos efectos se producen por la interacción de la acción con el entorno o con otras acciones.

En lugar de considerar todos efectos posibles en la representación de la acción, los agentes deberían poder razonar para obtener aquellas consecuencias que pueden deducirse de los efectos de las acciones. Este razonamiento debe ser rebatible dado que los efectos que se deducen pueden cambiar dependiendo de la información disponible sobre el entorno y las acciones que realice el agente.

Consideremos por ejemplo una acción que consiste en encender el interruptor de la luz para iluminar una habitación. Un efecto de esta acción será que *el interrutor está encendido*. El efecto *hay luz en la habitación* puede ser deducido como una consecuencia del efecto *el interruptor está encendido*. Por lo tanto, además de la acción, el agente necesita contar con conocimiento expresado a través de la siguiente regla rebatible: "*si el interruptor está encendido hay razones para creer que hay luz en la habitación*". Si *hay luz en la habitación* fuera considerado como un efecto de la acción resultaría difícil considerar

excepciones como que la *lámpara está quemada*. Sin embargo este tipo de situaciones son modeladas apropiadamente por un formalismo de argumentación rebatible. Por ejemplo, esta situación puede ser representada por la siguiente regla rebatible: "*si el interruptor esta encendido y la lámpara está quemada hay razones para creer que no hay luz en la habitación*".

1.2. Contribuciones

La contribución principal de esta tesis es la definición de un nuevo método de planificación denominado APOP (Argumentative Partial Order Planning) que combina Argumentación Rebatible con técnicas de Planificación de Orden Parcial (POP). La incorporación de Argumentación Rebatible permite utilizar conocimiento del dominio para razonar rebatiblemente durante la construcción de un plan.

Para representar dominios y problemas de planificación, se define un formalismo denominado DAKAR (Defeasible Argumentation for Knowledge and Action Representation) que combina acciones y argumentación rebatible. Lo novedoso de este formalismo es que permite representar conocimiento acerca del dominio y definir acciones utilizando la Programación en Lógica Rebatible. Por un lado, el conocimiento y la argumentación rebatible se utilizan para razonar acerca de las precondiciones, restricciones y efectos de las acciones. Por otra parte, las acciones son utilizadas para cambiar el entorno y poder lograr las metas.

Para poder incorporar argumentación rebatible al proceso de planificación, se define una nueva estructura de plan llamada *plan parcial argumentativo* que combina acciones y argumentos. Se definen nuevos tipos de amenazas (llamadas *threats* en POP) para capturar todas las interferencias que pueden surgir de la interacción entre acciones y argumentos, y se definen nuevos métodos para resolver cada una de ellas.

Otra contribución importante es el diseño y desarrollo de un algoritmo de planificación para APOP. Al igual que el algoritmo tradicional de POP, el algoritmo de APOP realiza una búsqueda sobre el espacio de planes. Una característica distintiva es la incorporación de una etapa de expansión previa a la búsqueda, que permite simplificar notablemente el algoritmo.

Por último, cabe destacar que gran parte de los resultados obtenidos a lo largo del desarrollo de esta tesis fueron publicados en [GSG07, GGS08].

1.3. Organización de la tesis

Esta tesis está organizada de la siguiente manera:

Capítulo 2: En este capítulo se introducen los conceptos necesarios para el desarrollo de la tesis. En primer lugar se presentan los principales conceptos y terminología del área de *planificación*. Se presenta un lenguaje de representación de acciones simple, del cual derivan la mayoría de los lenguajes de representación, y que se utilizará para describir una técnica de planificación de propósito general conocida como *Planificación de Orden Parcial* (POP). Además, se presenta un resumen de *Programación en Lógica Rebatible* (en inglés *Defeasible Logic Programming*). Este formalismo combina técnicas de programación en lógica con argumentación rebatible y será utilizado para representar conocimiento en un nuevo lenguaje de representación de acciones definido en el capítulo 3, y en el capítulo 4 será combinado con técnicas de POP para definir un nuevo método de planificación.

Capítulo 3: En este capítulo se define un formalismo llamado DAKAR (*Defeasible Argumentation for Knowledge and Action Representation*) el cual combina acciones y argumentación rebatible para representar dominios y problemas de planificación. Lo novedoso de este formalismo es que permite representar conocimiento acerca del dominio y definir acciones utilizando la Programación en Lógica Rebatible.

Capítulo 4: Se define un nuevo método de planificación denominado APOP (*Argumentative Partial Order Planning*) que combina técnicas de Planificación de Orden Parcial (POP) con Argumentación Rebatible, y se presenta una extensión al algoritmo de POP que permite resolver problemas de planificación definidos utilizando DAKAR como formalismo de representación.

Capítulo 5: Durante el desarrollo de esta tesis se implementó un planificador en base a los algoritmos propuestos, junto con una herramienta de visualización que permite observar el proceso de planificación y los planes obtenidos. En este capítulo se describen algunos detalles de la implementación, funcionamiento y uso de las herramientas implementadas.

Capítulo 6: En este capítulo se describen las principales características de algunos de los lenguajes y formalismos de representación de acciones, dominios y problemas de planificación existentes, para realizar una comparación con el formalismo de representación DAKAR definido en el capitulo 3. Se describen además algunas propuestas que combinan argumentación y planificación, que son comparadas con el enfoque propuesto en esta tesis.

Capítulo 7: Finalmente se detallan los resultados y conclusiones obtenidas.

Capítulo 2

Conceptos Preliminares

En este capítulo se introducirán los conceptos fundamentales para el desarrollo de la tesis. En la sección 2.1 se introducirán los principales conceptos y terminología del área de *planificación*. En la sección 2.1.1 se describirá STRIPS, un lenguaje de representación de acciones simple del cual derivan la mayoría de los lenguajes de representación, y que se utilizará, en la sección 2.1.2, para describir una técnica de planificación de propósito general conocida como *Planificación de Orden Parcial*. Por último, en la sección 2.2 se presenta un resumen de *Programación en Lógica Rebatible* (en inglés *Defeasible Logic Programming*). Este formalismo, que combina técnicas de la programación en lógica con argumentación rebatible, será utilizado para representar conocimiento en un nuevo formalismo de representación de acciones, dominios y problemas de planificación definido en el capítulo 3, y en el capítulo 4 será combinado con técnicas de planificación de orden parcial para definir un nuevo método de planificación.

2.1. Planificación

La tarea de sintetizar un *curso de acción* que le permita a un agente lograr sus metas, se denomina *planificación*. Para realizar esta tarea, el agente debe tener en cuenta las metas que quiere lograr, las acciones que es capaz de ejecutar y cual es el estado actual del entorno en el cual se realizarán dichas acciones. La representación de estos elementos se denomina problema de planificación y su definición es la siguiente:

Definición 2.1 (Problema de planificación)
Un *problema de planificación* puede definirse formalmente como una terna $\langle \mathrm{I}, \mathrm{G}, \mathrm{A} \rangle$ [Wel94], donde:

- I es una descripción del *estado inicial* del mundo en algún lenguaje formal L.
- G es una descripción de las *metas* del agente en el lenguaje formal L.
- A es una descripción de las *acciones* que el agente puede ejecutar, en el lenguaje formal L.

Definición 2.2 (Dominio de Planificación)
Un *dominio de planificación* consiste de una caracterización del mundo donde actúa el agente y de las acciones que puede ejecutar, sin considerar la descripción del estado inicial y de las metas que dependen de cada problema de planificación en particular.

Los dominios de planificación pueden variar en tamaño y complejidad desde las operaciones de carga del transbordador espacial, hasta *dominios* "de juguete" como *el mundo de bloques*. Este último es uno de los dominios mas populares y utilizado en la bibliografía para ejemplificar los conceptos del área, dado que es un dominio reducido, simple y claro.

Ejemplo 2.1 El *mundo de bloques* (ver figura 2.1) es un dominio de planificación restringido a una mesa sobre la cual hay bloques. Un bloque puede estar sobre la mesa o sobre otro bloque. Además se cuenta con dos acciones que permiten mover bloques:

- *apilar un bloque sobre otro*: consiste en tomar un bloque que se encuentra sobre la mesa y ponerlo sobre otro bloque.
- *desapilar un bloque*: consiste en tomar un bloque que se encuentra sobre otro bloque y ponerlo sobre la mesa.

Un bloque se puede mover siempre y cuando esté libre (no tiene otro bloque encima). Para poder apilar un bloque sobre otro, este último también debe estar libre.

Dado un dominio de planificación, la definición de un problema de planificación se completa brindando la especificación del estado inicial y las metas del problema. La Figura 2.1 muestra un problema de planificación para este dominio. En el estado inicial hay tres bloques a, b y c que están libres y sobre la mesa. La meta del problema consiste en

construir una pila donde el bloque b esté sobre el bloque a y el bloque c esté sobre el bloque b.

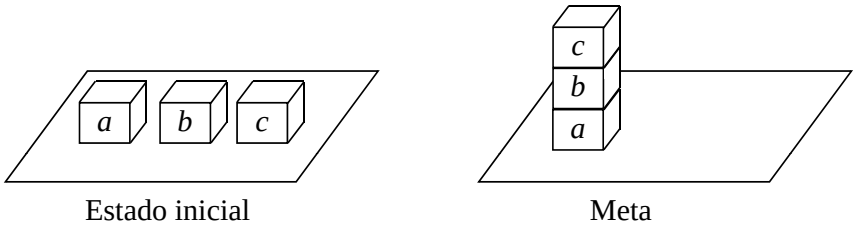

Figura 2.1: Mundo de bloques.

Solucionar un problema de planificación consiste en obtener un *plan*, esto es, una secuencia de acciones que al ser ejecutada por un agente en un estado del mundo que satisface la descripción del estado inicial, le permita al agente lograr sus metas. El algoritmo encargado de obtener un plan para solucionar un problema de planificación recibe el nombre de *planificador*. Por ejemplo, un plan para solucionar el problema planteado en el ejemplo 2.1 podría ser: apilar el bloque b sobre el bloque a y luego apilar el bloque c sobre el bloque b. En general, es posible que exista más de un plan para solucionar un problema de planificación, y también puede darse el caso que no exista ninguno.

La Definición 2.1 de un problema de planificación especifica una *clase* de problemas de planificación parametrizada por los lenguajes utilizados para representar el mundo, las metas y las acciones. Existe actualmente un amplio espectro de lenguajes de representación con diferentes niveles de expresividad. En general, la tarea de diseñar un algoritmo de planificación ó *planificador* se torna más compleja para lenguajes de representación más expresivos, y a su vez la eficiencia del algoritmo decrece.

2.1.1. Lenguaje de representación STRIPS

El nombre "STRIPS" proviene de las siglas en ingles de: *"STandford Research Institute Problem Solver"* un planificador muy famoso e influyente construido en los 70's para controlar el robot "*Shakey*" [FN71]. El mismo nombre fue usado más tarde para referirse al lenguaje formal utilizado por este planificador para representar acciones y problemas de planificación. Este lenguaje ha sufrido pequeñas modificaciones y extensiones, pero sigue siendo la base de la mayoría de los lenguajes utilizados por los sistemas de planificación actuales.

Existen en la literatura muchas notaciones alternativas y equivalentes para definir acciones siguiendo la representación STRIPS. Para mantener una notación uniforme con el resto de la tesis, se utilizará la notación y terminología de la lógica y la programación en lógica.

Definición 2.3 (Lenguaje STRIPS) El lenguaje STRIPS se define sobre tres conjuntos disjuntos y finitos de símbolos:

- un conjunto de variables $\mathcal{V}$.
- un conjunto de constantes $\mathcal{C}$.
- un conjunto de predicados $\mathcal{P}$. Una función llamada *aridad* le asignará a cada elemento $\mathcal{P}$ un número natural.

Las fórmulas válidas del lenguaje STRIPS se definen de la siguiente forma:

- una variable $X \in \mathcal{V}$ es un *término*
- una constante $c \in \mathcal{C}$ es un *término*
- Si $p \in \mathcal{P}$ es un predicado con $aridad(P) = i$ y $t_1, ..., t_i$ son términos, entonces $p(t_1, ..., t_i)$ es una *fórmula*. Las fórmulas que consisten de un solo predicado se denominan *fórmulas atómicas* o *átomos*.
- Un *literal* es un átomo A o un átomo negado "$\neg A$". Un literal se dirá negativo si es un átomo negado, y positivo en caso contrario.
- Si $l_1, ..., l_n$ son *literales*, entonces $\{l_1, ..., l_n\}$ es una *fórmula* que representa una *conjunción de literales*.

Las *fórmulas*, *átomos* y *literales* que no contienen variables (esto es, formadas sólo a partir de $\mathcal{C}$ y $\mathcal{P}$) se denominan *fórmulas fijas*[1], *átomos fijos* y *literales fijos* respectivamente. Una convención habitual para distinguir los elementos del lenguaje consiste en denotar las variables con una letra mayúscula inicial.

[1] En inglés *grounded*

El lenguaje STRIPS permite representar *acciones* y *estados del mundo*, y definir *reglas sintácticas* para transformar la representación de un estado mediante la aplicación de una acción.

Definición 2.4 (Estado) En el lenguaje STRIPS un *estado* se representa por medio de un conjunto $\{l_1, ..., l_n\}$ de literales fijos positivos.

La representación de un estado contiene todos los literales que son verdaderos en el estado del mundo representado, los literales que no son mencionados se asume que son falsos.

Ejemplo 2.2 Considere el dominio del *mundo de bloques* planteado en el ejemplo 2.1. Para poder representar un estado de este dominio se utilizarán los siguientes literales:

- $sobre\_mesa(X)$, para representar que un bloque X está sobre la mesa.
- $sobre(X, Y)$, para representar que un bloque X está sobre otro bloque Y.
- $libre(X)$, para representar que un bloque X no tiene ningún bloque encima.

Considere el *estado inicial* del problema planteado en la figura 2.1 donde hay tres bloques a, b y c que se encuentran libres y sobre la mesa. Este estado se representa en STRIPS por medio del siguiente conjunto:

$$\mathrm{I} = \{libre(a), libre(b), libre(c), sobre\_mesa(a), sobre\_mesa(b), sobre\_mesa(c)\}$$

Las acciones se representan mediante una descripción de sus *precondiciones* y *efectos*. Las precondiciones de una acción especifican las condiciones que deben cumplirse en un estado del mundo para que la acción pueda ejecutarse. Los efectos de una acción describen de que forma se modificará el mundo cuando la acción sea ejecutada.

Definición 2.5 (operador y acción STRIPS) Un *operador* STRIPS es un par $A = \langle \mathsf{P}, \mathsf{E} \rangle$ donde:

- A es un átomo que representa el nombre del operador.
- P es un conjunto de átomos, que representan las precondiciones de la acción.
- E es un conjunto consistente [2] de literales, que representan los efectos de la acción.

[2] Un conjunto de literales es *consistente* si no contiene simultáneamente un átomo A y su negación $\neg A$.

Una *acción* es un *operador instanciado*, es decir, un operador donde todos sus átomos y literales son fijos.

Un *operador* representa el conjunto de todas las instancias de acciones posibles que se pueden generar sustituyendo las variables por constantes del problema de planificación.

Aunque existe en la literatura otra forma para representar los operadores STRIPS (ver la observación 2.1), en esta tesis se utilizará la variante introducida en la definición 2.5.

Ejemplo 2.3 Considere las acciones definidas para el *mundo de bloques* planteado en el ejemplo 2.1. La acción de *apilar un bloque* X sobre otro bloque Y puede representarse de manera general por el operador STRIPS $apilar(X, Y) = \langle \mathsf{P}_1, \mathsf{E}_1 \rangle$, donde:

- $\mathsf{P}_1 = \{sobre\_mesa(X), libre(X), libre(Y)\}$ y
- $\mathsf{E}_1 = \{\neg sobre\_mesa(X), \neg libre(Y), sobre(X, Y)\}$.

Las precondiciones establecen que para poder apilar un bloque X sobre un bloque Y, X e Y deben estar libres y X debe estar sobre la mesa. Los efectos establecen que luego de apilar X sobre Y, X no estará sobre la mesa, Y no estará libre y X estará sobre Y.

Las variables mencionadas en un operador son *variables esquemáticas*, esto es, un operador es una abreviatura de todas las posibles acciones que se pueden obtener substituyendo esas variables por constantes del dominio. Por ejemplo, la acción $apilar(b, a) = \langle \mathsf{P}_2, \mathsf{E}_2 \rangle$, donde:

- $\mathsf{P}_2 = \{sobre\_mesa(b), libre(b), libre(a)\}$ y
- $\mathsf{E}_2 = \{\neg sobre\_mesa(b), \neg libre(a), sobre(b, a)\}$,

es una instancia particular del operador $apilar(X, Y)$ para el problema planteado en el ejemplo 2.1 y se obtiene sustituyendo las variables X e Y con las constantes b y a respectivamente.

La acción de *desapilar un bloque* X puede representarse por medio del siguiente operador $desapilar(X, Y) = \langle \mathsf{P}_3, \mathsf{E}_3 \rangle$, donde:

- $\mathsf{P}_3 = \{sobre(X, Y), libre(X)\}$ y
- $\mathsf{E}_3 = \{\neg sobre(X, Y), sobre\_mesa(X), libre(Y)\}$.

Una convención habitual en STRIPS consiste en que todas las apariciones de una variable deben ser reemplazadas por la misma constante, y las variables diferentes deben ser reemplazadas por constantes diferentes. En la sección 6.2 se introduce un lenguaje llamado PDDL que permite definir explícitamente restricciones de igualdad y desigualdad para cada acción en particular.

Una acción puede aplicarse o ejecutarse en un estado cuando sus precondiciones se cumplen en dicho estado. Cuando una acción es ejecutada, cambia la descripción del estado de la siguiente forma: todos los literales positivos presentes en los efectos de la acción se agregan a la descripción del estado, mientras que todos los literales negativos son quitados.

Definición 2.6 (Aplicación de una acción STRIPS) Se dice que una acción $A = \langle \mathsf{P}, \mathsf{E} \rangle$ es *aplicable* en un estado S si $\mathsf{P} \subseteq S$. El resultado de ejecutar una acción aplicable $A = \langle \mathsf{P}, \mathsf{E} \rangle$ en un estado S, es el estado:

$$Res(A, S) = S \setminus \{D | \neg D \in \mathsf{E}\} \cup \{A | A \in \mathsf{E}, A \text{ es un literal positivo } \}$$

La aplicación de una acción define una regla sintáctica para transformar la representación de un estado en otra, agregando y quitando átomos. Desde el punto de vista semántico la aplicación de acciones describe transiciones de estados en el espacio de estados.

Ejemplo 2.4 Considere el estado

$$\mathrm{I} = \{libre(a), libre(b), libre(c), sobre\_mesa(a), sobre\_mesa(b), sobre\_mesa(c)\}$$

y la acción $apilar(b, a) = \langle \mathsf{P}, \mathsf{E} \rangle$, donde:

- $\mathsf{P} = \{sobre\_mesa(b), libre(b), libre(a)\}$ y
- $\mathsf{E} = \{\neg sobre\_mesa(b), \neg libre(a), sobre(b, a)\}$,

La acción $apilar(b, a)$ es *aplicable* en el estado I dado que $\mathsf{P} \subset \mathrm{I}$. El resultado de aplicar $apilar(b, a)$ en I es el estado (ver figura 2.2):

$$\begin{aligned} Res(apilar(b, a), \mathrm{I}) &= \mathrm{I} \setminus \{sobre\_mesa(b), libre(a)\} \cup \{sobre(b, a)\} \\ &= \{libre(b), libre(c), sobre\_mesa(a), sobre\_mesa(c), sobre(b, a)\} \end{aligned}$$

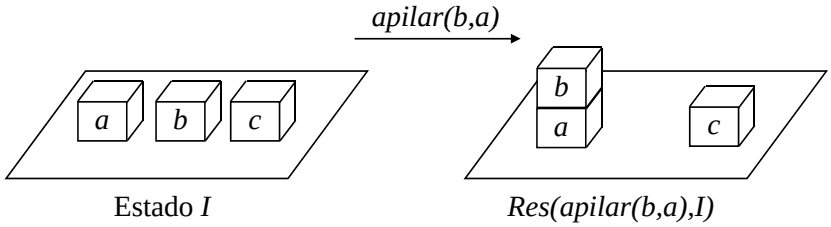

Figura 2.2: Aplicación de una acción STRIPS.

Observación 2.1 Cabe destacar que existe en la literatura una variante del lenguaje STRIPS donde no se utilizan literales negativos y los efectos de las acciones se describen mediante dos conjuntos *Add* y *Del* que distinguen los átomos que se agregarán y se quitarán respectivamente. Por ejemplo, en esta variante la acción $apilar(b, a)$ se representaría de la siguiente forma: $\langle \mathsf{P}, Add, Del \rangle$, donde

- $\mathsf{P} = \{sobre\_mesa(b), libre(b), libre(a)\}$ y
- $Add = \{sobre(b, a)\}$,
- $Del = \{sobre\_mesa(b), libre(a)\}$,

Se adoptará la primera representación dado los desarrollos posteriores.

Definición 2.7 (Estado inicial, metas, problema de planificación STRIPS) En la representación STRIPS, el *estado inicial* del mundo se representa como cualquier otro estado del mundo mediante un conjunto de literales fijos positivos. La descripción de las *metas* del agente está restringida a una conjunción de literales fijos positivos. Un *problema de planificación* STRIPS se define como una terna $\langle \mathrm{I}, \mathrm{G}, \mathrm{A} \rangle$ donde:

- I es un conjunto de literales fijos positivos que describe el *estado inicial*.
- G es un conjunto de literales fijos positivos que describe las *metas* del agente.
- A es un conjunto de *operadores* STRIPS.

Ejemplo 2.5 El problema de planificación introducido en el ejemplo 2.1 se define mediante el lenguaje STRIPS como la terna $\langle \mathrm{I}, \mathrm{G}, \mathrm{A} \rangle$ donde:

- $\mathrm{I} = \{libre(a), libre(b), libre(c), sobre\_mesa(a), sobre\_mesa(b), sobre\_mesa(c)\}$

- G = $\{sobre(b, a), sobre(c, b)\}$
- A = $\{apilar(X, Y) = \langle \mathsf{P}_1, \mathsf{E}_1 \rangle$, $desapilar(X, Y) = \langle \mathsf{P}_3, \mathsf{E}_3 \rangle\}$ el conjunto de los operadores definidos en el ejemplo 2.3.

Solucionar un problema de planificación consiste en obtener un *plan* o secuencia aplicable de acciones que al ser ejecutada en el estado inicial permita alcanzar un estado donde se satisfacen las metas.

Definición 2.8 [Secuencia aplicable de Acciones/Plan]
Una secuencia de acciones de acciones S es *a*plicable a partir de un estado I si:

- $S = [\,]$ (es una secuencia vacía) ó
- $S = [a_1, a_2, \ldots, a_n]$, la acción a_1 es aplicable en I y la secuencia $[a_2, \ldots, a_n]$ es *a*plicable a partir de $Res(a_1, \mathrm{I})$

El *resultado de aplicar* S en el estado I es el estado:

$$Res(S, \mathrm{I}) = Res(a_n, \ldots Res(a_2, Res(a_1, \mathrm{I})) \ldots)$$

Luego, la solución para un problema de planificación se define como sigue.

Definición 2.9 [Solución]
Sea $\langle \mathrm{I}, \mathrm{G}, \mathrm{A} \rangle$ un problema de planificación. Una secuencia de acciones S es una solución para $\langle \mathrm{I}, \mathrm{G}, \mathrm{A} \rangle$ si:

- S es apliable a partir de I y
- $\mathrm{G} \subseteq Res(S, \mathrm{I})$

Ejemplo 2.6 Considere el problema de planificación definido en el ejemplo 2.5. Un plan posible para solucionar este problema de planificación sería la secuencia: $[apilar(b, a), apilar(c, b)]$.

2.1.2. Planificación de Orden Parcial

En esta sección se describe un método de planificación conocido como *Planificación de Orden Parcial* (en inglés *Partial Order Planning*) [PW92, Wel94] que abreviaremos POP. La característica principal de POP es que los planes obtenidos están representados como secuencias parcialmente ordenadas de acciones. En este sentido, POP es una técnica de planificación de "menor compromiso" (*Least Commitment Planning*) [Wel94], dado que sólo se tienen en cuenta las restricciones de orden esenciales.

Como se mencionó anteriormente, los algoritmos de planificación están íntimamente relacionados con el lenguaje de representación utilizado. Por lo tanto, para simplificar la descripción del algoritmo de POP se utilizará el lenguaje de representación STRIPS (Sección 2.1.1), aunque existen algoritmos de POP más avanzados que permiten manejar lenguajes más expresivos como PDDL (el cual se introducirá en la Sección 6.2).

Planes Parciales, Vínculos Causales y Amenazas

El proceso llevado a cabo por la mayoría de los algoritmos de POP consiste en realizar una búsqueda por el espacio de planes de orden parcial. En este espacio, los nodos representan planes parcialmente especificados y los arcos denotan operaciones de refinamiento sobre planes, tales como agregar una acción a un plan o imponer una restricción de orden sobre las acciones del mismo.

A continuación se define la estructura utilizada para representar planes denominada *plan de orden parcial*, o simplemente *plan parcial*.

Definición 2.10 (Plan Parcial)
Un *plan parcial* es una cuaterna $P = (\mathcal{S}, \mathcal{O}, \mathcal{L}, \mathcal{SG})$ donde:

- $\mathcal{S}$ es un conjunto de *pasos* (en inglés *steps*) de la forma (S, A, P, E) donde S es el nombre (único) del paso, A es una instancia de acción [3] asociada al paso, P y E son las precondiciones y efectos de A respectivamente.
- $\mathcal{O}$ es un conjunto de *restricciones de orden* en inglés (*ordering constraints*) de la forma $S_i \prec S_j$, donde $\{(S_i, A_i, P_i, E_i), (S_j, A_j, P_j, E_j)\} \subseteq \mathcal{S}$. Las restricciones de orden establecen un orden explícito entre dos pasos del plan.

[3] Para simplificar la explicación del algoritmo sólo se permitirán acciones totalmente instanciadas dentro de un plan, aunque existen algoritmos capaces de manejar operadores parcialmente instanciados.

- $\mathcal{L}$ es un conjunto de *vínculos causales* (en inglés *causal links*) de la forma $S_i \xrightarrow{p} S_j$ donde $\{(S_i, A_i, P_i, E_i), (S_j, A_j, P_j, E_j)\} \subseteq \mathcal{S}$, $p \in E_i$ y $p \in P_j$.
- $\mathcal{SG}$ es un conjunto de *submetas* o *condiciones pendientes* (en inglés *open conditions*) de la forma (p, S) donde $(S, A, P, E) \in \mathcal{S}$, $p \in P$ y no existe ningún vínculo causal $X \xrightarrow{p} S$ en $\mathcal{L}$.

Cada *paso* representa la ejecución de una acción dentro del plan. La misma acción puede estar asociada a diferentes pasos de un plan, por esto es necesario distinguir entre acciones y pasos.

Una restricción de orden de la forma $S_i \prec S_j$ establece que el paso S_i debe ejecutarse antes que S_j. Las restricciones de orden deben ser *consistentes*, en el sentido que debe existir al menos un *orden total* que las satisfaga [Wel94]. Es decir, el conjunto de restricciones de orden $\mathcal{O}$ define un *orden parcial estricto* sobre los pasos del plan.

Definición 2.11 (clausura transitiva de un conjunto de restricciones de orden) Sea $\mathcal{O}$ un conjunto de restricciones de orden. La clausura transitiva de $\mathcal{O}$ se denotará $\mathcal{O}^+$.

Definición 2.12 (restricciones de orden consistentes) Sea $\mathcal{O}$ un conjunto de restricciones de orden y sea $S_i \prec S_j$ una restricción de orden. Se dice que $S_i \prec S_j$ *es consistente con* $\mathcal{O}$, si $\mathcal{O} \cup \{S_i \prec S_j\}$ es un *orden parcial estricto*, o equivalentemente $S_j \prec S_i \notin \mathcal{O}^+$. A su vez, se dice que $S_i \prec S_k \prec S_j$ *es consistente con* $\mathcal{O}$, si $\mathcal{O} \cup \{S_i \prec S_k, S_k \prec S_j\}$ es un *orden parcial estricto*, o equivalentemente $S_k \prec S_i \notin \mathcal{O}^+$ y $S_j \prec S_k \notin \mathcal{O}^+$.

Los vínculos causales se utilizan para registrar el origen de cada precondición que pertenece a cada paso y establecen dependencias entre los pasos del plan. Un vínculo de la forma $S_i \xrightarrow{p} S_j$ representa que la precondición p del paso S_j es lograda por un efecto del paso S_i en el plan. Dado que una causa debe aparecer antes del efecto que produce, un vínculo causal también representa una restricción de orden.

El conjunto $\mathcal{SG}$ contiene precondiciones de pasos del plan que no están sustentadas por ningún vínculo causal, es decir, las precondiciones que no han sido logradas aún.

Ejemplo 2.7 Sea $P = (\mathcal{S}, \mathcal{O}, \mathcal{L}, \mathcal{SG})$ un plan parcial donde:

- $\mathcal{S} = \{(S_1, A_1, \emptyset, \{r\}), (S_2, A_2, \{r\}, \{p\}), (S_3, A_3, \{s\}, \{q\}), (S_4, A_4, \{p, q\}, \emptyset)\}$,

- $\mathcal{O} = \{S_2 \prec S_4, S_3 \prec S_4, S_1 \prec S_2, S_1 \prec S_3\}$,
- $\mathcal{L} = \{S_1 \xrightarrow{r} S_2, S_2 \xrightarrow{p} S_4, S_3 \xrightarrow{q} S_4\}$ y
- $\mathcal{SG} = \{(s, S_3)\}$

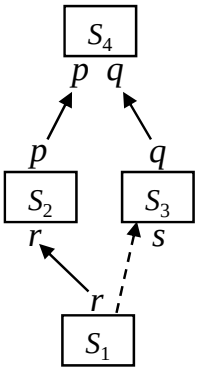

Figura 2.3: Representación gráfica del plan de orden parcial del ejemplo 2.7.

La figura 2.3 muestra una representación gráfica del plan parcial P del ejemplo 2.7. Los rectángulos representan los pasos del plan y están etiquetados con el nombre del paso. Los literales que aparecen debajo y sobre un paso representan las precondiciones y efectos del paso respectivamente. Las flechas continuas entre el efecto de un paso y la precondición de otro paso representan vínculos causales. Por último, las flechas rayadas entre un paso y otro representan restricciones de orden. Dado que los vínculos causales imponen también restricciones de orden, se puede pensar que debajo de cada flecha continua hay una flecha rayada.

Los vínculos causales también se utilizan para detectar posibles *interferencias* entre los pasos que puedan invalidar un plan. Considere la situación planteada en la figura 2.4(a). Note que S_k tiene $\neg p$ como efecto y las restricciones de orden permiten que S_k se ejecute entre S_i y S_j. Si esto último ocurre S_k negará p, rompiendo el vínculo entre S_i y S_j e impidiendo que S_j pueda ejecutarse.

Estas interferencias se denominan *amenazas* (en inglés *threats*) y se definen de la siguiente forma:

Definición 2.13 (amenaza) Sea $P = (\mathcal{S}, \mathcal{O}, \mathcal{L}, \mathcal{SG})$ un plan parcial, $S_i \xrightarrow{p} S_j$ un vínculo causal en $\mathcal{L}$ y a S_k un paso en $\mathcal{S}$ ($S_k \neq S_i$ y $S_k \neq S_j$). Se dice que S_k *amenaza* a (o es una *amenaza* para) $S_i \xrightarrow{p} S_j$ si (ver figura 2.4(a)):

- $S_i \prec S_k \prec S_j$ *es consistente con* $\mathcal{O}$, y
- S_k tiene $\neg p$ como efecto.

Para evitar que las *amenazas* invaliden un plan es necesario identificarlas y resolverlas. Hay dos alternativas o *métodos*, para resolver una amenaza:

- *avance* (*promotion*): si $S_j \prec S_k$ *es consistente con* $\mathcal{O}$, entonces agregar $S_j \prec S_k$ al conjunto $\mathcal{O}$ (figura 2.4(b))
- *retroceso* (*demotion*): si $S_k \prec S_i$ *es consistente con* $\mathcal{O}$, entonces agregar $S_k \prec S_i$ al conjunto $\mathcal{O}$ (figura 2.4(c))

Ambos métodos agregan una restricción de orden al plan para impedir que S_k se ejecute entre S_i y S_j. Cualquiera de los dos métodos puede aplicarse para resolver una amenaza, siempre y cuando la restricción de orden que se agrega sea consistente con las restricciones de orden del plan. Puede ocurrir que ningún método pueda aplicarse para resolver una amenaza porque ambas restricciones de orden resultan inconsistentes (ver figura 2.5. Por lo tanto no siempre es posible resolver una amenaza.

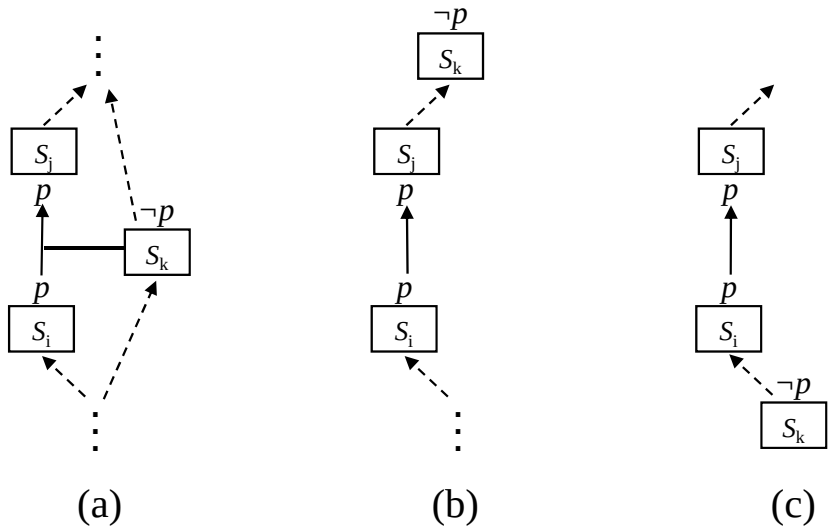

Figura 2.4: **(a)** El paso S_k *amenaza* el vínculo causal $S_i \xrightarrow{p} S_j$. La amenaza puede ser resuelta por: **(b)** *avance* (*promotion*) ó **(c)** *retroceso* (*demotion*).

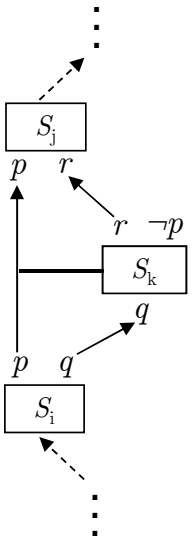

Figura 2.5: Amenaza sin solución: no es posible aplicar ningún método, porque tanto $S_k \prec S_i$ como $S_j \prec S_k$ no son consistentes con las restricciones de orden existentes $S_i \prec S_k$ y $S_k \prec S_j$.

El algoritmo de POP

Como se mencionó anteriormente, el proceso llevado a cabo por la mayoría de los algoritmos de POP consiste en realizar una búsqueda por el espacio de planes de orden parcial. Para mantener una notación uniforme y simplificar el algoritmo, los problemas de planificación se representan mediante un plan parcial denominado *plan nulo* o *plan inicial* (ver definición 2.14) que consiste de dos pasos especiales denominados *inicio* y fin. Estos pasos no se corresponden con la ejecución de ninguna acción, sino que se utilizan para representar el estado inicial y las metas del problema. El paso *inicio* no tiene precondiciones y sus efectos consisten de los literales que son verdaderos en el estado inicial. El paso fin no tiene efectos y sus precondiciones consisten de las metas del problema. Un plan nulo se define formalmente como sigue.

Definición 2.14 (plan nulo) Un *plan nulo* (ver figura 2.6) para un problema de planificación $\langle \mathrm{I}, \mathrm{G}, \mathrm{A} \rangle$, es un plan $PN = (\mathcal{S}, \mathcal{O}, \mathcal{L}, \mathcal{SG})$, donde:

- $\mathcal{S} = \{(inicio, \cdot, \emptyset, \mathrm{I}), (fin, \cdot, \mathrm{G}, \emptyset)\}$ [4]

[4]En lo que sigue se utilizará el símbolo "$\cdot$" como un carácter *comodín* para denotar cualquier elemento de una tupla.

- $\mathcal{O} = \{inicio \prec fin\}$,
- $\mathcal{L} = \emptyset$
- $\mathcal{SG} = \{(m, fin) \mid m \in \mathrm{G}\}$.

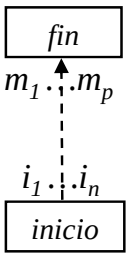

Figura 2.6: *Plan nulo* (o *plan inicial*) para un problema de planificación $\langle \mathrm{I}, \mathrm{G}, \mathrm{A} \rangle$, donde $\mathrm{I} = \{i_1, ..., i_n\}$ y $\mathrm{G} = \{m_1, ..., m_p\}$

Ejemplo 2.8 Considere el problema de planificación planteado en el ejemplo 2.1 para el mundo de bloques. Con el fin de mejorar la claridad de las figuras los literales serán abreviados. El literal $libre(X)$ será reemplazado por $l(X)$, $sobre\_mesa(X)$ por $sm(X)$ y $sobre(X, Y)$ por $s(X, Y)$. Luego el problema se define como $\langle \mathrm{I}, \mathrm{G}, \mathrm{A} \rangle$, donde:

- $\mathrm{I} = \{l(a), l(b), l(c), sm(a), sm(b), sm(c)\}$
- $\mathrm{G} = \{s(b, a), s(b, c)\}$
- $\mathrm{A} = \left\{ \begin{array}{c} apilar(X, Y) = \langle \{l(X), l(Y), sm(X)\}, \{\neg l(Y), \neg sm(X), s(X, Y)\} \rangle \\ desapilar(X, Y) = \langle \{l(X), s(X, Y)\}, \{\neg s(X, Y), l(Y), sm(X), \} \rangle \end{array} \right\}$

El plan nulo para este problema se define como $PN = (\mathcal{S}, \mathcal{O}, \mathcal{L}, \mathcal{SG})$, donde :

- $\mathcal{S} = \{(inicio, \cdot, \emptyset, \{l(a), l(b), l(c), sm(a), sm(b), sm(c)\}), (fin, \cdot, \{s(b, a), s(b, c)\}, \emptyset)\}$,
- $\mathcal{O} = \{inicio \prec fin\}$,
- $\mathcal{L} = \emptyset$
- $\mathcal{SG} = \{(s(b, a), fin), (s(b, c), fin)\}$.

El plan nulo es el nodo de partida de la búsqueda. Durante la misma, el algoritmo intenta completar el plan agregando pasos y/o vínculos causales para lograr submetas, y restricciones de orden para resolver amenazas. La búsqueda termina cuando se encuentra una solución al problema de planificación. Desde el punto de vista del algoritmo un plan parcial es una solución, si no contiene *amenazas* y todas las precondiciones de los pasos del plan están sustentadas por algún vínculo causal (esto es, $\mathcal{SG} = \emptyset$). Para definir formalmente el concepto de solución para un plan parcial, es necesario establecer primero una correspondencia entre los planes parcialmente ordenados y los planes totalmente ordenados (esto es, secuencias de acciones).

Definición 2.15 (orden topológico de un plan parcial)
Sea $P = (\mathcal{S}, \mathcal{O}, \mathcal{L}, \mathcal{SG})$ un plan parcial. Una secuencia totalmente ordenada de pasos de acción $Sec = [(inicio, \cdot, \cdot, \cdot), (s_1, accion_1, \cdot, \cdot), \ldots, (s_n, accion_n, \cdot, \cdot), (fin, \cdot, \cdot, \cdot)]$ es un orden topológico para P si:

- $\{(inicio, \cdot, \cdot, \cdot), (s_1, accion_1, \cdot, \cdot), \ldots, (s_n, accion_n, \cdot, \cdot), (fin, \cdot, \cdot, \cdot)\} = \mathcal{S}$, esto es, cada paso de la secuencia pertenece a los pasos de acción del plan y viceversa.
- $s_i \prec s_j$ *es consistente con* $\mathcal{O}$, para todo $0 \leq i < j \leq n$

Se denominará $Plan(Sec) = [accion_1, \ldots, accion_n]$ a la secuencia de acciones que se obtiene de reemplazar cada paso de acción en S por su acción correspondiente, con excepción de los pasos *inicio* y fin.

Luego, el concepto de solución para un plan parcial se define como sigue.

Definición 2.16 (plan parcial solución)
Un plan parcial P es una solución para un problema de planificación $\langle \mathrm{I}, \mathrm{G}, \mathrm{A} \rangle$ si *para todo* orden topológico $S = [(inicio, \cdot, \cdot, \cdot), (s_1, accion_1, \cdot, \cdot), \ldots, (s_n, accion_n, \cdot, \cdot), (fin, \cdot, \cdot, \cdot)]$ para P, $Plan(S) = [accion_1, \ldots, accion_n]$ es una solución para $\langle \mathrm{I}, \mathrm{G}, \mathrm{A} \rangle$.

La figura 2.7 ilustra el proceso llevado a cabo por el algoritmo durante la búsqueda de un plan para el problema definido en ejemplo 2.8. Dado que hay muchos caminos posibles, solo se muestra una posible secuencia de planes parciales obtenidos durante la búsqueda, que lleva a encontrar una solución. El algoritmo comienza construyendo el *plan inicial* para el problema (figura 2.7(a)) y luego intenta completarlo agregando pasos para lograr

las *submetas*, que inicialmente son las precondiciones del paso fin. Supongamos que el algoritmo selecciona la submeta $s(b, a)$ del paso fin y elige la acción $apilar(b, a)$ para lograrla. Luego, agrega un nuevo paso y el vínculo causal correspondiente obteniendo el plan parcial que se muestra en la figura 2.7(b). Las precondiciones del paso incorporado se convierten en nuevas submetas del plan. En este caso, las precondiciones de $apilar(b, a)$ son efectos del paso *inicio*, por lo tanto no es necesario agregar un nuevo paso para lograrlas. Simplemente se agregan los vínculos causales correspondientes, obteniendo el plan parcial que se muestra en la figura 2.7(c). Luego, la submeta restante es la precondición de $s(c, b)$ del paso *fin*. Supongamos que el algoritmo elige la acción $apilar(c, b)$ para lograrla, luego agrega un nuevo paso y el vínculo causal correspondiente obteniendo el plan parcial de la figura 2.7(d).

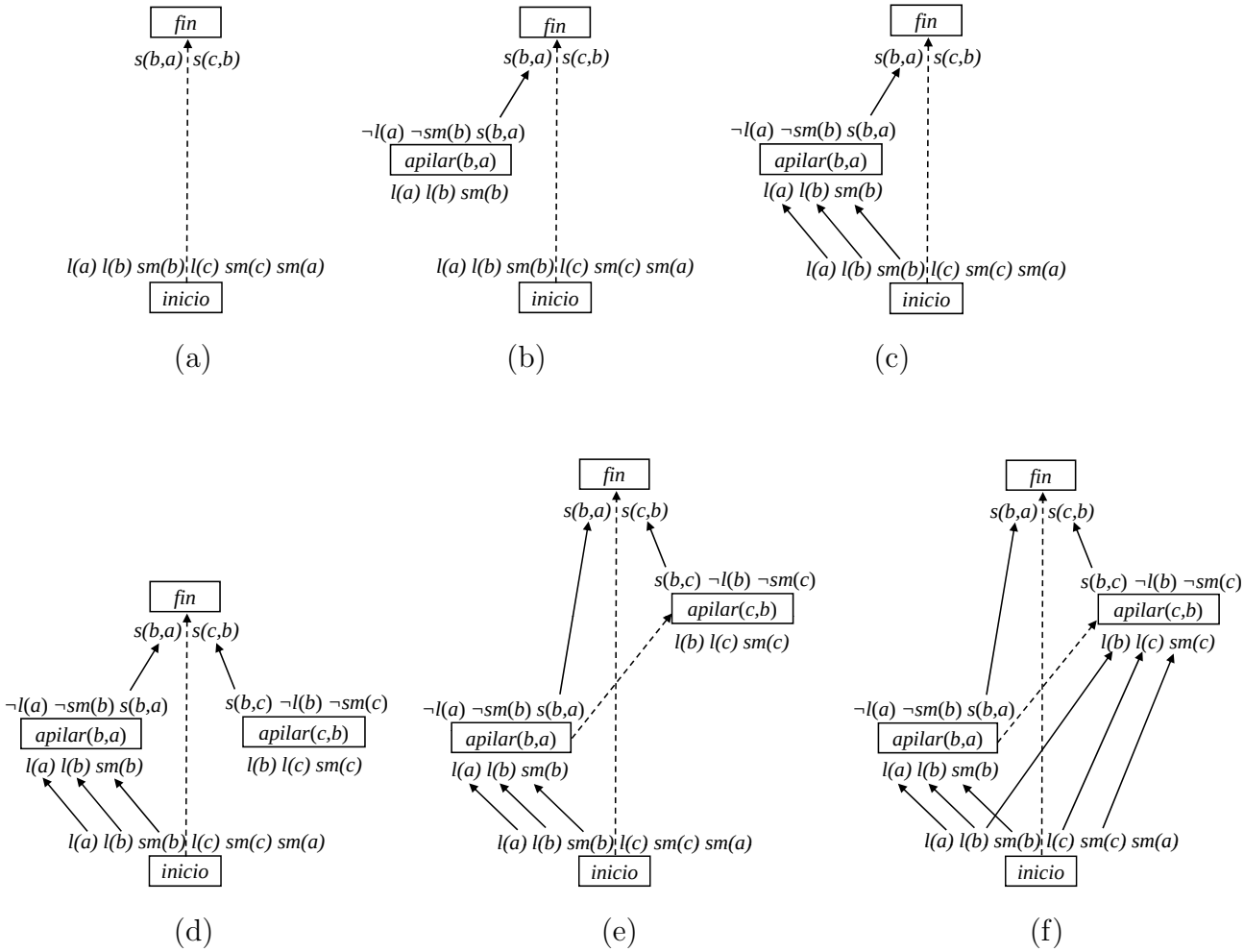

Figura 2.7: Mundo de bloques y POP

Hasta el momento solo se han agregado pasos y vínculos causales, pero en ningún momento se verificó si existen *amenazas* que puedan invalidar el plan. Si prestamos atención a la figura 2.7(d) podemos ver que el paso $apilar(c, b)$ *amenaza* el vínculo causal $inicio \xrightarrow{l(b)} apilar(b, a)$ dado que $\neg l(b)$ es un efecto de $apilar(c, b)$. Esta amenaza debe ser resuelta aplicando alguno de los métodos presentados anteriormente. El método *retroceso* no puede aplicarse dado que el paso $apilar(c, b)$ no puede aparecer antes del paso *inicio* (esto es, $apilar(c, b) \prec inicio$ no es consistente con $\mathcal{O}$). En este caso la única opción para solucionar la amenaza es aplicar el método *avance*, el cual consiste en agregar una restricción de orden para que el paso $apilar(c, b)$ se ejecute después del paso $apilar(b, a)$. Luego el algoritmo agregará la restricción $apilar(b, a) \prec apilar(c, b)$ al plan parcial, obteniendo el plan parcial de la figura 2.7(e). Una vez resuelta la amenaza, el algoritmo intenta lograr las submetas restantes, que en este caso son las precondiciones del paso $apilar(c, b)$. Todas estas precondiciones aparecen en los efectos del paso *inicio*, luego para lograrlas el algoritmo simplemente agrega los vínculos causales correspondientes, obteniendo el plan parcial de la figura 2.7(e) que soluciona el problema.

En las figuras 2.8 y 2.9 se presenta un esquema de un algoritmo de POP que permite planificar en dominios de planificación definidos en el lenguaje STRIPS y solo considera acciones totalmente instanciadas.

El algoritmo está definido utilizando un pseudocódigo que combina sentencias de lenguajes imperativos, fórmulas matemáticas y en algunos casos lenguaje natural. En particular, los símbolos $\downarrow$ y $\uparrow$ sobre los parámetros de las funciones o procedimientos, identifican parámetros de entrada y salida respectivamente. Las sentencias **choose** y **fail** se utilizan para describir el no-determinismo. La primitiva **choose** permite al algoritmo elegir entre diferentes alternativas y mantener un registro de las alternativas pendientes. Cuando el algoritmo encuentra una sentencia **fail** el control vuelve al punto del algoritmo en donde se realizó la última elección y las opciones pendientes son consideradas.

La función principal `POP`, recibe a través de los parámetros de entrada la descripción de un problema de planificación $\langle \mathrm{I}, \mathrm{G}, \mathrm{A} \rangle$ y devuelve, si es que existe, un *Plan* que soluciona el problema. Esta función comienza construyendo un *plan nulo* para el problema (función `Construir_Plan_Nulo`) y luego intenta completarlo a través del procedimiento recursivo `Completar_Plan`. En cada instancia, este procedimiento selecciona una submeta (p, S_j) del plan, elige un paso para lograrla y lo agrega al plan a través del procedimiento `Elegir_Paso`. Cuando se agrega un paso al plan, el algoritmo debe verifi-

```
function POP(I↓,G↓,A↓): Plan;
begin
   Plan:= Construir_Plan_Nulo(I,G);
   Completar_Plan(Plan);
end

function Construir_Plan_Nulo(I↓,G↓,A↓): Plan;
begin
    Plan.S:= {(inicio,·,∅,I), (fin,·,G,∅) }; // pasos
    Plan.O:= {inicio ≺ fin}; // restricciones de orden
    Plan.L:= ∅; // vinculos causales
    Plan.SG:= {(g, fin)| g ∈ G}; // submetas
end

procedure Completar_Plan(Plan↓↑);
begin
   if Plan.SG ≠ ∅ then
    begin
       Sea (p, S_j) ∈ Plan.SG;
       Elegir_Paso(Plan, A, (p, S_j));
       Plan.SG:= Plan.SG \ {(p, S_j)};
       Resolver_Amenazas(Plan);
       Completar_Plan(Plan);
    end
end
```

Figura 2.8: Algoritmo POP

car si se producen nuevas *amenazas* y elegir un método para resolverlas (procedimiento `Resolver_Amenazas`). Si en algún momento el algoritmo falla (esto es, ejecuta una sentencia **fail**) al elegir un paso para lograr una submeta o en resolver una amenaza, vuelve al último punto de elección. Cabe destacar que, si bien el orden en que el algoritmo considera las submetas puede influir en el resultado de la búsqueda, cada una de las submetas debe ser considerada en algún momento para asegurar la completitud del mismo. Luego, la selección de las submetas no constituye un punto de elección.

El procedimiento `Elegir_Paso` se encarga de elegir un paso S_i (ya sea uno existente en el $Plan$ o uno nuevo creado a partir del conjunto de operadores A) y agregarlo al plan para lograr la submeta (p, S_j). Para esto, construye el conjunto $Pasos$, formado por todos los pasos que tienen p como efecto, y en caso de pertenecer al $Plan$ puedan aparecer antes

```
procedure Elegir_Paso(Plan↓↑, A↓, (p, S_j↓));
begin
  Pasos:= {(S, A, P, E) | p ∈ E, S es un paso nuevo creado a partir de A ∈ A ó
                (S, A, P, E) ∈ Plan.S y  S ≺ S_j es consistente con Plan.O };
  if Pasos= ∅ then fail;
  choose (S_i, A_i, P_i, E_i) from Pasos;
  Plan.O := Plan.O ∪ {S_i ≺ S_j};
  Plan.L := Plan.L ∪ {S_i --p--> S_j};
  if S_i ∉ Plan.S then  // S_i es un paso nuevo
   begin
      Plan.S:= Plan.S ∪ (S_i, A_i, P_i, E_i);
      Plan.O:= Plan.O ∪ {inicio ≺ S_i, S_i ≺ fin};
      Plan.SG:= Plan.SG ∪ {(p, S_i)| p ∈ P_i };
   end
end

procedure Resolver_Amenazas(Plan↓↑);
begin
  for each S_i --p--> S_j ∈ Plan.L do
   for each S_k ∈ {(S_k, A_k, P_k, E_k)| (S_k, A_k, P_k, E_k) ∈ Plan.S, ¬ p ∈ E_k y
                                  S_i≺S_k≺S_j es consistente con Plan.O} do
     choose
        avance:
          if S_j≺S_k es consistente con Plan.O then
             Plan.O :=  Plan.O ∪ {S_j≺S_k};
          else fail;
        retroceso:
          if S_k≺S_i  es consistente con Plan.O then
             Plan.O :=  Plan.O ∪ {S_k≺S_i};
          else fail;
     end
end
```

Figura 2.9: Algoritmo POP (continuación)

de S_j. Si este conjunto es vacío (esto es, no existen pasos para lograr (p, S_j)) el algoritmo falla y vuelve al último punto de elección. Sino, elige un paso S_i del conjunto y lo agrega al plan (sólo si es nuevo), junto con el vínculo causal $S_i \xrightarrow{p} S_j$ y la restricción de orden $S_i \prec S_j$. Si el algoritmo falla más adelante, volverá a este punto para elegir otro paso presente en el conjunto.

El procedimiento `Resolver_Amenazas` se encarga de detectar y resolver amenazas en el plan. Para cada paso S_k que amenace a un vínculo causal $S_i \xrightarrow{p} S_j$ este procedimiento elige un método para resolverlo (*avance* o *retroceso*) y agrega al plan la restricción de orden correspondiente, siempre y cuando sea consistente con las restricciones de orden del plan. Nuevamente, si el algoritmo falla más adelante, volverá a este punto para elegir otro método para resolver la amenaza.

2.2. Programas Lógicos Rebatibles

La *Programación en Lógica Rebatible* (en inglés Defeasible Logic Programming, que se abreviará DeLP) [GS04] es un formalismo que combina técnicas de la programación en lógica y argumentación rebatible. Al igual que en la programación en lógica, en DeLP el conocimiento se representa usando hechos y reglas. Un programa lógico rebatible esta formado por un conjunto de *hechos*, *reglas estrictas*, y *reglas rebatibles*, permitiendo éstas últimas la representación de información tentativa. Toda conclusión del programa deberá ser sustentada por algún *argumento* que pueda construirse utilizando las reglas y los hechos del programa. Cuando se utilicen reglas rebatibles para derivar una conclusión C, esta conclusión será tentativa, y podrá ser refutada por información que la contradiga. Para decidir cuando una conclusión C puede aceptarse a partir de un programa lógico rebatible, se realizará un *análisis dialéctico*, construyendo argumentos a favor y en contra de la conclusión C.

Un argumento que sustenta una conclusión C podrá ser atacado por otros argumentos *derrotadores* que lo contradigan y que puedan construirse a partir del programa. Dichos derrotadores podrán a su vez ser atacados, y así sucesivamente, generando una secuencia de argumentos llamada *línea de argumentación*. Cada línea de argumentación deberá satisfacer ciertas propiedades, a fin de evitar que se generen argumentaciones falaces. El proceso completo considera para cada argumento todos sus posibles argumentos derrotadores, lo cual, en lugar de una única línea de argumentación, genera un conjunto de líneas representadas con un *árbol de dialéctica*. Este análisis dialéctico determinará si la conclusión en cuestión está *garantizada* o no, a partir del programa.

2.2.1. Sintaxis de los Programas Lógicos Rebatibles

Un *programa lógico rebatible* está compuesto por *hechos*, *reglas estrictas*, y *reglas rebatibles*, cuya sintaxis se define a continuación.

Definición 2.17 (Hecho)
Un hecho es un literal fijo L. Esto es, un átomo fijo, o un átomo fijo negado usando negación fuerte "$\sim$´´.

Por ejemplo, un programa lógico rebatible podrá contener los hechos: "$interruptor(i_1)$" y "$\sim encendido(i_1)$", para representar que "i_1" es un interruptor y que no esta encendido.

Definición 2.18 (Complemento de un literal)
Sea L un literal y A un átomo, el complemento de L con respecto a la negación fuerte, denotado $\overline{L}$, se define de la siguiente manera:
si $L = A$, entonces $\overline{L} = \sim A$,
si $L = \sim A$, entonces $\overline{L} = A$.

Definición 2.19 (Regla Estricta)
Una *regla estricta* es un par ordenado, denotado "$L_0 \leftarrow L_1, \ldots, L_n$". La $Cabeza$ de la regla, L_0, es un literal fijo y el cuerpo, $\{L_1, \ldots, L_n\}$ $(n > 0)$, es un conjunto finito no vacío de literales fijos.

Definición 2.20 (Regla Rebatible)
Una *regla rebatible* es un par ordenado, denotado "$L_0 -\!\prec L_1, \ldots, L_n$". La $Cabeza$ L_0 es un literal fijo y el cuerpo $\{L_1, \ldots, L_n\}$ $(n > 0)$ es un conjunto finito no vacío de literales fijos.

Sintácticamente, el símbolo "$-\!\prec$" es lo único que distingue a una regla rebatible de una estricta. Pragmáticamente, las *reglas rebatibles* se utilizarán para representar conocimiento rebatible, esto es, información tentativa que puede utilizarse en la medida que no exista información que la contradiga. Las *reglas estrictas* representarán conocimiento seguro y libre de excepciones, esto es, siempre que se crea en los literales que forman el cuerpo, se podrá creer con la misma seguridad en el literal que corresponde a la cabeza.

De esta manera, conocimiento como *"los patos son aves"* se representará utilizando una regla estricta:

$$ave \leftarrow pato$$

mientras que la información *"las aves usualmente vuelan"* se representará con la regla rebatible:

$$vuela \prec ave$$

La sintaxis de las *reglas estrictas* corresponde a la sintaxis de las *reglas básicas* (basic rules) definidas por Lifschitz en [Lif96]. En DeLP se las llama "estrictas" para acentuar la diferencia con las "rebatibles". Las reglas estrictas representan una conexión fuerte entre el cuerpo y la cabeza de la regla, mientras que las reglas rebatibles permiten expresar una conexión débil. Una regla rebatible "$a \prec b$" expresa que "*creer en b provee razones para creer en a*" [SL92]. Por ejemplo, "$encendida(l_1) \prec conectado(i_1, l_1), encendido(i_1)$" expresa que creer en que un interruptor i_1 está encendido y conectado a una lámpara l_1 provee razones para creer que l_1 está encendida. Note que esta regla representa información tentativa dado que podría ocurrir que el interruptor esté encendido pero la lámpara no, por ejemplo, si la lámpara esta rota (representado por la regla: "$\sim encendida(l_1) \prec lampara(l_1), rota(l_1)$") o no hay electricidad ("$\sim encendida(l_1) \prec lampara(l_1), \sim electricidad$").

Definición 2.21 (Programa Lógico Rebatible)
Un programa lógico rebatible (o programa DeLP) es un conjunto $\mathcal{P}$ de hechos, reglas estrictas y reglas rebatibles. En algunas ocasiones se denotará a un programa lógico rebatible $\mathcal{P}$ con el par $\mathcal{P}=(\Pi, \Delta)$, donde Π identifica el subconjunto de hechos y reglas estrictas (que representan el conocimiento no rebatible) y el subconjunto Δ de reglas rebatibles.

Observación 2.2 Es importante destacar, que aunque las reglas de un programa DeLP utilizan únicamente literales fijos, siguiendo la convención habitual [Lif96], en algunos ejemplos se utilizarán variables solamente como una forma de denotar *esquemas de reglas*. Dado un esquema de regla R, se define $Ground(R)$ como el conjunto de todas las instancias de R con literales fijos. Dado un programa $\mathcal{P}$ se define [Lif96]

$$Ground(\mathcal{P}) = \bigcup_{R \in \mathcal{P}} Ground(R)$$

Es claro que $Ground(R)$ y $Ground(\mathcal{P})$ siguen las definiciones de regla y programa lógico rebatible dadas antes. Para diferenciar las variables de los demás elementos del programa, estas serán denotadas con una letra mayúscula inicial.

Ejemplo 2.9 A continuación se presenta un programa DeLP $\mathcal{P}_{2,9}$ donde se representa información sobre la iluminación de una oficina.

$$\Pi_{2.9} = \left\{ \begin{array}{l} luz \leftarrow luz\_natural. \\ luz\_natural \leftarrow persiana\_alta, es\_dia. \\ luz \leftarrow luz\_artificial. \\ \sim encendida(L) \leftarrow lampara(L), rota(L). \\ persiana\_alta. \\ es\_dia. \\ \sim electricidad. \\ interruptor(i_1). \\ encendido(i_1). \\ roto(i_1),\ lampara(l_1). \\ conectado(i_1, l_1). \\ interruptor(i_2). \\ encendido(i_2). \\ mojado(i_2). \\ lampara(l_2) \\ rota(l_2) \\ conectado(i_2, l_2). \end{array} \right\}$$

$$\Delta_{2.9} = \left\{ \begin{array}{l} luz\_artificial \prec encendida(L), lampara(L). \\ peligroso(I) \prec interruptor(I), mojado(I). \\ \sim peligroso(I) \prec interruptor(I), mojado(I), \sim electricidad. \\ encendida(L) \prec conectado(I, L), encendido(I). \\ \sim encendida(L) \prec lampara(L), \sim electricidad. \\ \sim encendida(L) \prec conectado(I, L), encendido(I), roto(I). \end{array} \right\}$$

2.2.2. Derivaciones Rebatibles

A continuación se introducirá el concepto de *derivación rebatible*, con el cual se define qué literales pueden ser derivados utilizando las reglas de un programa lógico rebatible.

Definición 2.22 (Derivación Rebatible de un literal)
Sea $\mathcal{P}$= (Π, Δ) un programa y L un literal. Una derivación rebatible para L a partir de $\mathcal{P}$, consiste de una secuencia finita de literales fijos $L_1, L_2, \ldots, L_n = L$, provisto de que

exista una secuencia de reglas estrictas o rebatibles $R_1, R_2, \ldots, R_n$ del programa $\mathcal{P}$, de tal forma que para cada literal L_i en la secuencia, tenemos que:

(a) L_i es un hecho, o

(b) existe en $\mathcal{P}$ una regla R_i con cabeza L_i y cuerpo $B_1, B_2, \ldots, B_k$ donde todo literal B_j del cuerpo $(1 \leq j \leq k)$ es un elemento de la secuencia que precede a L_i.

La derivación se dice *rebatible*, porque aunque un literal L pueda ser derivado, puede existir en el programa información que contradiga a L, y entonces L no será aceptado inmediatamente como una creencia válida del programa.

Observación 2.3 Como se expresó en la observación 2.2, en algunos ejemplos se utilizarán esquemas de reglas. Si un programa $\mathcal{P}$ tiene esquemas de reglas, para las derivaciones rebatibles se asume que se utiliza a $Ground(\mathcal{P})$.

Por ejemplo, a partir del programa del Ejemplo 2.9, es posible obtener una derivación rebatible para el literal "luz", ya que existe la secuencia de literales: $interruptor(i_1)$, $encendido(i_1)$, $lampara(l_1)$, $conectado(i_1, l_1)$, $encendida(L)$, $luz\_artificial$, luz; que se obtiene utilizando las reglas:

$$luz \leftarrow luz\_artificial$$
$$luz\_artificial \prec encendida(l_1), lampara(l_1).$$
$$encendida(l_1) \prec conectado(i_1, l_1), encendido(i_1).$$

Observación 2.4 Si un programa lógico rebatible $\mathcal{P}$ no tiene hechos, entonces no puede obtenerse ninguna derivación rebatible.

Dado un programa $\mathcal{P}$ y un literal L puede existir más de una derivación rebatible para L. Inclusive pueden existir derivaciones utilizando sólo reglas rebatibles, o sólo reglas estrictas. En particular, una derivación a partir de $(\Pi, \emptyset)$ (esto es, sólo utiliza reglas estrictas) recibe el nombre de *derivación estricta*. Por ejemplo, el literal $luz\_natural$ tiene una derivación estricta a partir de del programa del Ejemplo 2.9 ya que su derivación utiliza solamente los hechos $persiana\_alta$, es_dia y la regla estricta $luz\_natural \leftarrow persiana\_alta, es\_dia$.

Como se verá más adelante, las derivaciones estrictas sustentan conclusiones que no podrán ser refutadas, ya que se basan en reglas que no son rebatibles.

Como las reglas de programa permiten utilizar literales negados en la cabeza, entonces es posible derivar literales complementarios. Por ejemplo, a partir del programa del Ejemplo 2.9 existen derivaciones rebatibles para "*encendida*(l_1)" y "∼*encendida*(l_1)". Esto nos lleva a la siguiente definición.

Definición 2.23 (Conjunto Contradictorio) Un conjunto de reglas y hechos es *contradictorio* si y solo si, a partir de ese conjunto es posible obtener derivaciones rebatibles para un literal L y su complemento $\overline{L}$ (definición 2.18).

De esta manera, se podrá representar información contradictoria en un programa lógico rebatible. Considere el programa $\mathcal{P}_{2,9} = (\Pi_{2.9}, \Delta_{2.9})$ del ejemplo 2.9. El conjunto $\Pi_{2.9}$ no es contradictorio, sin embargo $\Pi_{2.9} \cup \Delta_{2.9}$ es un conjunto contradictorio, ya que pueden derivarse rebatiblemente los literales "*encendida*(l_1)" y "∼*encendida*(l_1)".

El uso de la negación fuerte enriquece la expresividad del lenguaje, y permite la representación de información contradictoria. Como las reglas rebatibles permiten representar información tentativa, en general en los programas $\mathcal{P}$=(Π, Δ), el conjunto $\Pi \cup \Delta$ será contradictorio. Sin embargo, el conjunto Π, de hechos y reglas estrictas, es usado para representar información no-rebatible, y por lo tanto debe poseer cierta coherencia interna.

Observación 2.5 En todo programa lógico rebatible $\mathcal{P}$=(Π, Δ), se asume que el conjunto Π es un conjunto no contradictorio. De esta forma, no existen dos literales complementarios que tengan derivaciones estrictas a partir de $\mathcal{P}$.

2.2.3. Argumentación Rebatible

Como se mencionó anteriormente, a partir de un programa DeLP es posible derivar literales complementarios. Por ejemplo, a partir del programa del Ejemplo 2.9 existen derivaciones rebatibles para "*encendida*(l_1)" y "∼*encendida*(l_1)". Para decidir si un literal L es aceptado como una creencia válida de un programa, DeLP incorpora un formalismo de *argumentación rebatible* que le permite identificar las porciones de conocimiento que están en contradicción. Este proceso involucra la construcción de *argumentos* para un literal y *contra-argumentos* para los argumentos obtenidos. Luego, se utiliza un análisis dialéctico para decidir cual información prevalece y determinar si el literal está aceptado.

A continuación se presenta la noción de estructura de argumento.

Definición 2.24 (Estructura de argumento)
Sea L un literal y $\mathcal{P}$=(Π, Δ) un programa lógico rebatible, una *estructura de argumento* para L es un par $\langle \mathcal{A}, L \rangle$, donde $\mathcal{A}$ es conjunto de reglas rebatibles de Δ, tal que:

1. existe una derivación rebatible para L a partir de $\Pi \cup \mathcal{A}$.
2. $\Pi \cup \mathcal{A}$ es no contradictorio, y
3. $\mathcal{A}$ es minimal, es decir, no existe un subconjunto propio $\mathcal{A}'$ de $\mathcal{A}$ tal que $\mathcal{A}'$ satisface las condiciones (1) y (2).

Ejemplo 2.10 A partir del programa del ejemplo 2.9 es posible obtener una estructura de argumento para el literal "$encendida(l_1)$":

$$\langle \{encendida(l_1) -\prec conectado(i_1, l_1), encendido(i_1)\}, encendida(l_1) \rangle$$

También es posible obtener una estructura de argumento para "$\sim encendida(l_1)$":

$$\langle \{\sim encendida(l_1) -\prec lampara(l_1), \sim electricidad\}, \sim encendida(l_1) \rangle$$

Sin embargo, no es posible obtener un argumento para el literal "$encendida(l_2)$":, porque la derivación rebatible de "$encendida(l_2)$" unida al conjunto Π es un conjunto contradictorio (observe que "$\sim encendida(l_2)$" tiene una derivación estricta utilizando los hechos $lampara(l_2)$, $rota(l_2)$) y la regla $\sim encendida(l_2) \leftarrow lampara(l_2), rota(l_2)$).

Definición 2.25 (Igualdad de estructuras de argumentos) Dos estructuras de argumento $\langle \mathcal{A}_1, L_1 \rangle$ y $\langle \mathcal{A}_2, L_2 \rangle$ son iguales si y sólo si, $\mathcal{A}_1 = \mathcal{A}_2$, y $L_1 = L_2$.

Definición 2.26 (Sub-estructura de argumento) Una estructura de argumento $\langle \mathcal{B}, Q \rangle$ es una sub-estructura de argumento de $\langle \mathcal{A}, L \rangle$ si y sólo si, $\mathcal{B} \subseteq \mathcal{A}$.

Ejemplo 2.11 Considérese el siguiente programa:

$$\sim d \leftarrow b, e$$
$$e$$
$$a \leftarrow b$$
$$b -\prec c$$
$$c$$
$$\sim a -\prec d$$
$$d -\prec e$$

Aquí, la estructura de argumento $\langle\{b \prec c\}, a\rangle$ es una sub-estructura de argumento de $\langle\{b \prec c\}, \sim d\rangle$ y viceversa. Sin embargo, $\langle\{b \prec c\}, a\rangle$ no es igual a $\langle\{b \prec c\}, \sim d\rangle$, ya que difieren en el literal que sustentan.

Proposición 2.1 Sea $\mathcal{P}$=(Π, Δ) un programa lógico rebatible. Existe una derivación estricta para un literal L a partir de $\mathcal{P}$, si y sólo si, existe una única estructura de argumento $\langle\mathcal{A}, L\rangle$, donde $\mathcal{A}$=$\emptyset$.
Demostración: si existe una derivación estricta para L a partir de Π, entonces se verifica la condición (1) de la definición de argumento con $\Pi \cup \emptyset$. Debido a que el conjunto Π es no contradictorio, entonces también se verifica trivialmente la condición (2). Finalmente, como no puede existir un subconjunto propio de $\mathcal{A}$=$\emptyset$, se cumple la condición (3) y por lo tanto $\langle\{\}, L\rangle$ es una estructura de argumento. Dicha estructura es única debido a que cualquier otro conjunto de reglas rebatibles que cumpla las condiciones (1) y (2) será un superconjunto de $\emptyset$. La demostración en el otro sentido es trivial.

2.2.4. Desacuerdo, ataque y contra-argumentación

Como se mencionó anteriormente, el proceso argumentativo realizado por DeLP debe identificar las porciones de conocimiento que están en contradicción. En particular, dos literales complementarios representan información estrictamente contradictoria. Este concepto se generaliza a través de la siguiente definición.

Definición 2.27 (Literales en Desacuerdo)
Sea $\mathcal{P}$=(Π, Δ) un programa, y Π el conjunto de reglas estrictas y hechos del programa. Dos literales L y L_1 están en *desacuerdo*, si y sólo si el conjunto $\Pi \cup \{L, L_1\}$ es contradictorio.

El ejemplo más simple de literales en desacuerdo son dos literales complementarios como "p" y "$\sim p$", ya que $\{p, \sim p\} \cup \Pi$ es contradictorio, cualquiera sea el conjunto Π. Sin embargo, como muestra el siguiente ejemplo, dos literales que no sean complementarios, como "p" y "q", también pueden estar en desacuerdo, si unidos al conjunto Π permiten derivar dos literales complementarios.

Ejemplo 2.12 Dado el conjunto $\Pi = \{(b), (a \leftarrow b, p), (\sim a \leftarrow c), (c \leftarrow q)\}$, los literales "$p$" y "$q$" están en desacuerdo, ya que a partir de $\{p, q\} \cup \Pi$ es posible derivar tanto al literal "a", como al literal "$\sim a$".

A continuación se define una noción de conflicto entre dos estructuras de argumento basada en la noción de literales en desacuerdo.

Definición 2.28 (Contra-argumento o ataque)
Sean $\langle \mathcal{A}_1, L_1 \rangle$ y $\langle \mathcal{A}_2, L_2 \rangle$ dos estructuras de argumento obtenidas a partir de un programa $\mathcal{P}$. Decimos que $\langle \mathcal{A}_1, L_1 \rangle$ *contra-argumenta* a $\langle \mathcal{A}_2, L_2 \rangle$ en el literal L, si y sólo si, existe una sub-estructura de argumento $\langle \mathcal{A}, L \rangle$ de $\langle \mathcal{A}_2, L_2 \rangle$ tal que L y L_1 están en desacuerdo. El argumento $\langle \mathcal{A}, L \rangle$ se llama *sub-argumento de desacuerdo*, y el literal L será el *punto de contra-argumentación*.

Si $\langle \mathcal{A}_1, L_1 \rangle$ *contra-argumenta* a $\langle \mathcal{A}_2, L_2 \rangle$, entonces también se dirá que $\langle \mathcal{A}_1, L_1 \rangle$ *ataca* a $\langle \mathcal{A}_2, L_2 \rangle$, o que $\langle \mathcal{A}_1, L_1 \rangle$ es un contra-argumento para $\langle \mathcal{A}_2, L_2 \rangle$.

Ejemplo 2.13 A partir del programa $\mathcal{P}_{2,9}$ del ejemplo 2.9 es posible construir la estructura de argumento $\langle \mathcal{A}_2, luz\_artificial \rangle$, con

$$\mathcal{A}_2 = \left\{ \begin{array}{l} luz\_artificial \prec encendida(l_1), lampara(l_1), \\ encendida(l_1) \prec conectado(i_1, l_1), encendido(i_1) \end{array} \right\}$$

y la estructura de argumento $\langle \mathcal{A}_1, \sim encendida(l_1) \rangle$, con:

$$\mathcal{A}_1 = \{\sim encendida(l_1) \prec lampara(l_1), \sim electricidad\}$$

La estructura de argumento $\langle \mathcal{A}_1, \sim encendida(l_1) \rangle$ es un contra-argumento para la estructura $\langle \mathcal{A}_2, luz\_artificial \rangle$. El subargumento de desacuerdo es $\langle \{encendida(l_1) \prec conectado(i_1, l_1), encendido(i_1)\}, encendida(l_1) \rangle$, y el punto de contra-argumentación es "$encendida(l_1)$".

Por otra parte, a partir del programa $\mathcal{P}_{2,9}$ es posible construir la estructura de argumento $\langle \mathcal{A}_4, peligroso(i_1) \rangle$, con

$$\mathcal{A}_4 = \{ peligroso(i_1) \prec interruptor(i_1), mojado(i_1)\}$$

y la estructura de argumento $\langle \mathcal{A}_3, \sim peligroso(i_1) \rangle$, con:

$$\mathcal{A}_3 = \{\sim peligroso(i_1) \prec interruptor(i_1), mojado(i_1), \sim electricidad\}$$

La estructura de argumento $\langle \mathcal{A}_3, \sim peligroso(i_1)\rangle$ es un contra-argumento para la estructura $\langle \mathcal{A}_4, peligroso(i_1)\rangle$. En este caso, el subargumento de desacuerdo es el mismo $\langle \mathcal{A}_4, peligroso(i_1)\rangle$, y el punto de contra-argumentación es "$peligroso(i_1)$". Nótese además que recíprocamente, $\langle \mathcal{A}_4, peligroso(i_1)\rangle$ es un contra-argumento para $\langle \mathcal{A}_3, \sim peligroso(i_1)\rangle$.

Observación 2.6 En la definición 2.28, puede ocurrir que el subargumento $\langle \mathcal{A}, L\rangle$ sea el propio $\langle \mathcal{A}_2, L_2\rangle$, y en este caso se dirá que $\langle \mathcal{A}_1, L_1\rangle$ contra-argumenta o ataca *directamente* a $\langle \mathcal{A}_2, L_2\rangle$. Pero también puede ocurrir que $\langle \mathcal{A}, L\rangle$ sea un subargumento propio de $\langle \mathcal{A}_2, L_2\rangle$, esto es, el ataque se produzca en un punto L interior de $\langle \mathcal{A}_2, L_2\rangle$, entonces se dirá que $\langle \mathcal{A}_1, L_1\rangle$ contra-argumenta o ataca *indirectamente* a $\langle \mathcal{A}_2, L_2\rangle$. La Figura 2.10 muestra estos dos casos.

Observación 2.7 (Notación gráfica) En la Figura 2.10 y en otras a continuación, se utilizará una notación gráfica definida en [SCG94] para estructuras de argumento. En dicha notación, una estructura de argumento es representado con un triángulo, en cuyo vértice superior se indica el literal que sustenta. Una etiqueta en la base del triángulo dá un nombre de referencia a la estructura de argumento. Un triángulo dentro de otro mayor representa una sub-estructura de argumento, y una línea recta entre dos vértices de triángulos representa ataque entre estructuras de argumento (conectando los literales en desacuerdo). Por ejemplo, en la Figura 2.10 la estructura de argumento $\langle \mathcal{A}_2, L_2\rangle$ contiene a una sub-estructura de argumento $\langle \mathcal{A}, L\rangle$, y la estructura de argumento $\langle \mathcal{A}_1, L_1\rangle$ es un contra-argumento que ataca indirectamente en el punto L, con sub-argumento de desacuerdo $\langle \mathcal{A}, L\rangle$.

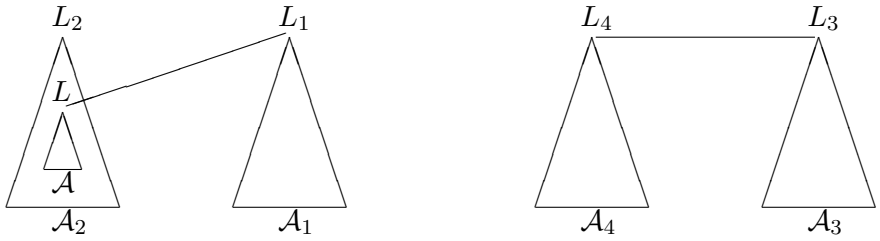

Figura 2.10: Ataque indirecto (izquierda) y ataque directo (derecha)

Ejemplo 2.14 Dado un programa $\mathcal{P}$=(Π, Δ) donde
Π= $\{(c)$, (d), $(\sim p \leftarrow h)\}$
Δ= $\{(a \prec p)$, $(p \prec c)$, $(b \prec \sim p)$,$(h \prec d)\}$
pueden obtenerse las siguientes estructuras de argumento:

$\langle \mathcal{A}, a\rangle = \langle\{(a \prec p), (p \prec c)\}, a\rangle$
$\langle \mathcal{B}, b\rangle = \langle\{(b \prec \sim p), (h \prec d)\}, b\rangle$
$\langle \mathcal{A}_1, p\rangle = \langle\{p \prec c\}, p\rangle$
$\langle \mathcal{B}_1, h\rangle = \langle\{h \prec d\}, h\rangle$
Como puede verse, los literales p y h están en desacuerdo, y además $\langle \mathcal{A}_1, p\rangle$ es un contra-argumento para $\langle \mathcal{B}, b\rangle$ en el literal h, siendo $\langle \mathcal{B}_1, h\rangle$ el contra-argumento de desacuerdo.

La siguiente proposición muestra que aquellos literales que poseen una derivación estricta no pueden ser refutados, ya que no puede existir un contra-argumento que los ataque.

Proposición 2.2 Sea $\langle \mathcal{A}, L\rangle$ una estructura de argumento donde $\mathcal{A} = \emptyset$, esto es, L tiene una derivación estricta, entonces no existe un contra-argumento para $\langle \mathcal{A}, L\rangle$.
Demostración: Si existiera un contra-argumento $\langle \mathcal{A}_2, L_2\rangle$ *para* $\langle \mathcal{A}, L\rangle$*, entonces* $\mathcal{A}_2 \cup \mathcal{A} \cup \Pi$ *sería un conjunto contradictorio. Como* $\mathcal{A}= \emptyset$*, entonces* $\mathcal{A}_2 \cup \Pi$ *sería un conjunto contradictorio, lo cual violaría la condición (2) de la definición de argumento, y por lo tanto* $\langle \mathcal{A}_2, L_2\rangle$ *no sería un argumento.*

También es importante destacar que una estructura de argumento $\langle \mathcal{A}, L\rangle$, con $\mathcal{A} = \emptyset$, es decir, formada con una derivación estricta, no puede ser contra-argumento de ninguna otra estructura $\langle \mathcal{B}, Q\rangle$, ya que $\langle \mathcal{B}, Q\rangle$ no puede existir por definición. Este hecho es formalizado en la siguiente proposición.

Proposición 2.3 Sea $\langle \mathcal{A}, L\rangle$ una estructura de argumento donde $\mathcal{A} = \emptyset$, esto es, L tiene una derivación estricta, entonces no existe argumento $\langle \mathcal{B}, Q\rangle$ tal que $\langle \mathcal{A}, L\rangle$ es un contra-argumento para $\langle \mathcal{B}, Q\rangle$.
Demostración: Si existiera $\langle \mathcal{B}, Q\rangle$*, tal que* $\langle \mathcal{A}, L\rangle$ *fuera un contra-argumento para* $\langle \mathcal{B}, Q\rangle$*, entonces* $\mathcal{B} \cup \mathcal{A} \cup \Pi$ *sería un conjunto contradictorio. Como* $\mathcal{A}= \emptyset$*, entonces* $\mathcal{B} \cup \Pi$ *sería un conjunto contradictorio, lo cual violaría la condición (2) de la definición de argumento, y por lo tanto* $\mathcal{B}$ *no sería un argumento.*

La siguiente proposición muestra que existe una simetría en la noción de contra-argumentación.

Proposición 2.4 Si una estructura de argumento $\langle \mathcal{A}_1, L_1\rangle$ contra-argumenta a otra $\langle \mathcal{A}_2, L_2\rangle$ en el punto L, con el subargumento de desacuerdo $\langle \mathcal{A}, L\rangle$, entonces también

se cumple que $\langle \mathcal{A}, L\rangle$ contra-argumenta a $\langle \mathcal{A}_1, L_1\rangle$ en el punto L_1.
Demostración: como $\langle \mathcal{A}_1, L_1\rangle$ contra-argumenta a $\langle \mathcal{A}_2, L_2\rangle$ en el punto L, entonces L y L_1 están en desacuerdo, por lo tanto, $\langle \mathcal{A}, L\rangle$ contra-argumenta a $\langle \mathcal{A}_1, L_1\rangle$ en el punto L_1, ya que existe un subargumento de $\langle \mathcal{A}_1, L_1\rangle$, que es el mismo $\langle \mathcal{A}_1, L_1\rangle$, tal que L y L_1 están en desacuerdo.

Observación 2.8 Si $\langle \mathcal{A}_1, L_1\rangle$ contra-argumenta a $\langle \mathcal{A}_2, L_2\rangle$ en el punto L_2, esto es en el literal que sustenta $\mathcal{A}_2$, entonces $\langle \mathcal{A}_2, L_2\rangle$ contra-argumenta a $\langle \mathcal{A}_1, L_1\rangle$.

2.2.5. Derrota y líneas de argumentación

La noción de contra-argumentación establece cuando un argumento es atacado por otro. Sin embargo, como muestra la proposición 2.4, siempre que un argumento $\mathcal{A}$ contra-argumenta a otro $\mathcal{B}$, existe un subargumento de $\mathcal{B}$ que es un contra-argumento para $\mathcal{A}$. Esta simetría impide determinar cuando un argumento se impone (derrota) sobre el otro. Para esto, es indispensable contar con un criterio de comparación que permita comparar argumentos y de esta forma decidir si el ataque de un contra-argumento $\langle \mathcal{A}_1, L_1\rangle$ sobre un argumento $\langle \mathcal{A}_2, L_2\rangle$ es lo suficientemente fuerte como para rebatirlo. Si $\langle \mathcal{A}_1, L_1\rangle$ ataca a $\langle \mathcal{A}_2, L_2\rangle$ en el punto L, siendo $\langle \mathcal{A}, L\rangle$ el sub-argumento de desacuerdo en $\langle \mathcal{A}_2, L_2\rangle$, se deben comparar el argumento atacante $\langle \mathcal{A}_1, L_1\rangle$ y el subargumento de desacuerdo $\langle \mathcal{A}, L\rangle$, resultando alguno de los siguientes casos:

(a) $\langle \mathcal{A}_1, L_1\rangle$ es mejor que $\langle \mathcal{A}, L\rangle$,

(b) $\langle \mathcal{A}, L\rangle$ es mejor que $\langle \mathcal{A}_1, L_1\rangle$,

(c) $\langle \mathcal{A}_1, L_1\rangle$ y $\langle \mathcal{A}, L\rangle$ son incomparables,

(d) $\langle \mathcal{A}_1, L_1\rangle$ y $\langle \mathcal{A}, L\rangle$ son equivalentes.

Como la comparación entre argumentos puede definirse de varias maneras, en lo que sigue se asumirá que existe una relación de preferencia "$\succ$" que permite distinguir cuando una estructura de argumento es preferida sobre otra. Por ejemplo, si $\langle \mathcal{A}_1, L_1\rangle$ es mejor que $\langle \mathcal{A}, L\rangle$, se denotará "$\langle \mathcal{A}_1, L_1\rangle \succ \langle \mathcal{A}, L\rangle$".

Definición 2.29 (Derrotador: Propio y de bloqueo)
Sea $\langle \mathcal{A}_1, L_1\rangle$ un contra-argumento para $\langle \mathcal{A}_2, L_2\rangle$ en el punto L, siendo $\langle \mathcal{A}, L\rangle$ el sub-argumento de desacuerdo en $\langle \mathcal{A}_2, L_2\rangle$.

- $\langle \mathcal{A}_1, L_1 \rangle$ un *derrotador propio* para $\langle \mathcal{A}_2, L_2 \rangle$ si $\langle \mathcal{A}_1, L_1 \rangle \succ \langle \mathcal{A}, L \rangle$(esto es, $\langle \mathcal{A}_1, L_1 \rangle$ es mejor que $\langle \mathcal{A}, L \rangle$).
- $\langle \mathcal{A}_1, L_1 \rangle$ un *derrotador de bloqueo* para $\langle \mathcal{A}_2, L_2 \rangle$ si $\langle \mathcal{A}_1, L_1 \rangle \not\succ \langle \mathcal{A}, L \rangle$ y $\langle \mathcal{A}, L \rangle \not\succ \langle \mathcal{A}_1, L_1 \rangle$(esto es, $\langle \mathcal{A}_1, L_1 \rangle$ y $\langle \mathcal{A}, L \rangle$ son incomparables o equivalentes).

Finalmente, $\langle \mathcal{A}_1, L_1 \rangle$ es un *derrotador* para $\langle \mathcal{A}, L \rangle$ en el literal L si, $\langle \mathcal{A}_1, L_1 \rangle$ es un derrotador propio o de bloqueo para $\langle \mathcal{A}_2, L_2 \rangle$.

En DeLP el criterio de comparación de argumentos es modular, lo cual permite utilizar el criterio mas adecuado para el dominio que se desea representar. En la literatura de DeLP se han definido diferentes criterios. Por ejemplo, en [GS04] se define un criterio que utiliza prioridades sobre las reglas y en [FEGS08] se define un criterio basado en prioridades sobre literales seleccionados del programa. Otra alternativa, es utilizar el criterio de especificidad [Poo85, Lou87, SL92] que permite comparar reglas o argumentos sintácticamente, sin la necesidad de especificar explícitamente las prioridades. Informalmente, el criterio de especificidad establece que una estructura de argumento $\langle \mathcal{A}_1, L_1 \rangle$ será preferida a otra estructura $\langle \mathcal{A}_2, L_2 \rangle$, si se cumple alguna de las dos condiciones siguientes:

1. $\langle \mathcal{A}_1, L_1 \rangle$ usa mayor información que $\langle \mathcal{A}_2, L_2 \rangle$, o
2. basándose en la misma información, $\langle \mathcal{A}_1, L_1 \rangle$ es más directo (usa menos reglas) que $\langle \mathcal{A}_2, L_2 \rangle$

Formalmente, el criterio de especificidad generalizada se define como sigue.

Definición 2.30 (Especificidad generalizada) Sea $\mathcal{P}=(\Pi, \Delta)$ un programa lógico rebatible, con $\Pi = \Psi \cup \Omega$, donde Ψ es el conjunto de hechos y Ω es el conjunto de reglas estrictas. Sea $\mathcal{F}$ el conjunto de literales que tienen una derivación rebatible a partir de $\mathcal{P}$. Una estructura de argumento $\langle \mathcal{A}_1, L_1 \rangle$ es *estrictamente más específica* que otra estructura de argumento $\langle \mathcal{A}_2, L_2 \rangle$ (denotado $\langle \mathcal{A}_1, L_1 \rangle \succ \langle \mathcal{A}_2, L_2 \rangle$) si se verifican las dos condiciones siguientes:

1. para todo conjunto $\mathcal{H} \subseteq \mathcal{F}$,
   si existe una derivación rebatible para L_1 a partir de $\Omega \cup \mathcal{H} \cup \mathcal{A}_1$ *($\mathcal{H}$ activa a $\mathcal{A}_1$)*, y no existe una derivación estricta para L_1 a partir de $\Omega \cup \mathcal{H}$, entonces existe una derivación rebatible para L_2 a partir de $\Omega \cup \mathcal{H} \cup \mathcal{A}_2$, *($\mathcal{H}$ activa a $\mathcal{A}_2$)*

2. existe al menos un conjunto $\mathcal{H}' \subseteq \mathcal{F}$ tal que:
   existe una derivación rebatible para L_2 a partir de $\Omega \cup \mathcal{H}' \cup \mathcal{A}_2$ *($\mathcal{H}'$ activa a $\mathcal{A}_2$)*, no existe una derivación estricta de L_2 a partir de $\Omega \cup \mathcal{H}'$, y no existe una derivación rebatible de L_1 a partir de $\Omega \cup \mathcal{H}' \cup \mathcal{A}_1$ *($\mathcal{H}$' no activa a $\mathcal{A}_1$)*.

Los conjuntos $\mathcal{A}_1$ y $\mathcal{A}_2$ contienen reglas rebatibles, y Ω reglas estrictas sin hechos, por lo tanto es claro (Observación 2.4) que no puede existir ninguna derivación rebatible a partir de $\Omega \cup \mathcal{A}_1$, o a partir de $\Omega \cup \mathcal{A}_2$. Los conjuntos $\mathcal{H}$ y $\mathcal{H}'$ contienen literales, los cuales son tomados como hechos para obtener las derivaciones rebatibles. La condición 1 establece que para todo conjunto de literales (hechos) $\mathcal{H} \subseteq \mathcal{F}$, si H permite derivar L_1 utilizando reglas de $\Omega \cup \mathcal{A}_1$, donde al menos una regla de $\mathcal{A}_1$ es utilizada (ya que no existe una derivación estricta de L_1 a partir de $\Omega \cup \mathcal{H}$), entonces ese mismo conjunto $\mathcal{H}$ permite derivar rebatiblemente a L_2 utilizando los reglas de $\Omega \cup \mathcal{A}_2$. En la condición 2, se busca que exista al menos un conjunto $\mathcal{H}' \subseteq \mathcal{F}$, que permita derivar rebatiblemente a L_2 pero que no permita derivar a L_1.

Ejemplo 2.15 Como se mostró en el ejemplo 2.13 la estructura de argumento $\langle \mathcal{A}_4, peligroso(i_1) \rangle$, con

$$\mathcal{A}_4 = \{\ peligroso(i_1) \prec interruptor(i_1), mojado(i_1)\}$$

es un contra-argumento para $\langle \mathcal{A}_3, \sim peligroso(i_1) \rangle$ (y viceversa), con:

$$\mathcal{A}_3 = \{\sim peligroso(i_1) \prec interruptor(i_1), mojado(i_1), \sim electricidad\}$$

Considerando especificidad generalizada como criterio de comparación, tenemos que $\mathcal{A}_3 \succ \mathcal{A}_4$, porque $\mathcal{A}_3$ usa mas información al basarse en "$interruptor(i_1)$", "$mojado(i_1)$" y "$\sim electricidad$", mientras que $\mathcal{A}_4$ "$interruptor(i_1)$" y "$mojado(i_1)$". Luego, $\langle \mathcal{A}_3, \sim peligroso(i_1) \rangle$ es un derrotador propio para $\langle \mathcal{A}_4, peligroso(i_1) \rangle$.

Por otra parte, la estructura de argumento $\langle \mathcal{A}_1, \sim encendida(l_1) \rangle$, con:

$$\mathcal{A}_1 = \{\sim encendida(l_1) \prec lampara(l_1), \sim electricidad\}$$

es un derrotador de bloqueo para $\langle \mathcal{A}_2, luz\_artificial \rangle$, con:

$$\mathcal{A}_2 = \left\{ \begin{array}{l} luz\_artificial \prec encendida(l_1), lampara(l_1), \\ encendida(l_1) \prec conectado(i_1, l_1), encendido(i_1) \end{array} \right\}$$

porque el subargumento de desacuerdo

$$\mathcal{D} = \{encendida(l_1) \prec conectado(i_1, l_1), encendido(i_1)\}$$

y $\mathcal{A}_1$ no son comparables utilizando especificidad generalizada.

La relación de derrota entre los argumentos de un programa sólo establece un orden entre dos argumentos en conflicto. El estado de un argumento $\langle \mathcal{A}_0, L_0 \rangle$ con respecto al programa $\mathcal{P}$, dependerá de la interacción de $\langle \mathcal{A}_0, L_0 \rangle$ con el resto de los argumentos que pueden obtenerse. Por ejemplo, si un argumento $\langle \mathcal{A}_0, L_0 \rangle$ es derrotado por un argumento $\langle \mathcal{A}_1, L_1 \rangle$, entonces $\langle \mathcal{A}_1, L_1 \rangle$ representa razones para rechazar $\langle \mathcal{A}_0, L_0 \rangle$. Sin embargo, podría existir un derrotador $\langle \mathcal{A}_2, L_2 \rangle$ para $\langle \mathcal{A}_1, L_1 \rangle$ que invalide $\langle \mathcal{A}_1, L_1 \rangle$ reinstaurando $\langle \mathcal{A}_0, L_0 \rangle$. La secuencia de interacciones puede ir más lejos, y a su vez puede existir un derrotador $\langle \mathcal{A}_3, L_3 \rangle$ para $\langle \mathcal{A}_2, L_2 \rangle$, que vuelve a invalidar $\langle \mathcal{A}_0, L_0 \rangle$, y asi siguiendo. Esto da origen una secuencia de argumentos [$\langle \mathcal{A}_0, L_0 \rangle$,$\langle \mathcal{A}_1, L_1 \rangle$,$\langle \mathcal{A}_2, L_2 \rangle$ $\langle \mathcal{A}_3, L_3 \rangle$,. . .] llamada *línea de argumentación*, donde cada elemento es un derrotador de su predecesor.

Definición 2.31 (Línea de Argumentación)
Sea $\mathcal{P}$ un programa lógico rebatible, y $\langle \mathcal{A}_0, L_0 \rangle$ una estructura de argumento obtenida a partir de $\mathcal{P}$. Una *línea de argumentación* a partir de $\langle \mathcal{A}_0, L_0 \rangle$, es una secuencia de estructuras de argumentos (obtenidas a partir de $\mathcal{P}$) denotada $\Lambda = [\langle \mathcal{A}_0, L_0 \rangle, \langle \mathcal{A}_1, L_1 \rangle, \langle \mathcal{A}_2, L_2 \rangle \langle \mathcal{A}_3, L_3 \rangle, \ldots]$, donde para cada elemento de la secuencia $\langle \mathcal{A}_i, L_i \rangle$, el elemento siguiente $\langle \mathcal{A}_{i+1}, L_{i+1} \rangle$ es un derrotador para $\langle \mathcal{A}_i, L_i \rangle$.

El primer elemento de una línea de argumentación [$\langle \mathcal{A}_0, L_0 \rangle$,$\langle \mathcal{A}_1, L_1 \rangle$,$\langle \mathcal{A}_2, L_2 \rangle \ldots$] es una estructura que sustenta a un literal L_0. El segundo elemento la línea de argumentación derrota al primero y por consiguiente interfiere con la creencia en L_0. El tercer elemento derrota al segundo y por consiguiente, indirectamente sustenta a L_0, y asi siguiendo. De esta manera, en una línea de argumentación se distinguirán dos conjuntos disjuntos de argumentos: argumentos de soporte para L_0 y de interferencia para L_0.

Definición 2.32 (Argumentos de soporte y de interferencia)
Sea $\mathcal{P}$ un programa lógico rebatible, y $\Lambda = [\langle \mathcal{A}_0, L_0 \rangle, \langle \mathcal{A}_1, L_1 \rangle, \langle \mathcal{A}_2, L_2 \rangle \ldots]$ una línea de argumentación. El conjunto de argumentos de soporte está formado por los elementos de posiciones pares de la secuencia Λ_S= { $\langle \mathcal{A}_0, L_0 \rangle$, $\langle \mathcal{A}_2, L_2 \rangle$, $\langle \mathcal{A}_4, L_4 \rangle$, ... }, mientras que el conjunto de argumentos de interferencia está formado por los elementos de posiciones impares Λ_I= {$\langle \mathcal{A}_1, L_1 \rangle$, $\langle \mathcal{A}_3, L_3 \rangle$, ...}.

Ejemplo 2.16 Considerando los argumentos $\langle \mathcal{A}_1, \sim encendida(l_1)\rangle$ y $\langle \mathcal{D}, encendida(l_1)\rangle$ del ejemplo 2.15, tenemos que $[\langle \mathcal{A}_1, \sim encendida(l_1)\rangle, \langle \mathcal{D}, encendida(l_1)\rangle]$ es una línea de argumentación con un argumento de soporte y uno de interferencia.

Note que si un argumento es reintroducido en la secuencia se produciría una línea de argumentación infinita. Por ejemplo, $[\langle \mathcal{A}_1, \sim encendida(l_1)\rangle, \langle \mathcal{D}, encendida(l_1)\rangle \langle \mathcal{A}_1, \sim encendida(l_1)\rangle, \langle \mathcal{D}, encendida(l_1)\rangle, \dots]$ Este es solo un caso de *argumentación circular*, donde un argumento es reintroducido en una línea de argumentación para defenderse así mismo. Para evitar este y otro tipo de situaciones indeseadas que puedan producir argumentaciones falaces, DeLP impone las siguientes restricciones que establecen cuando una línea de argumentación es *aceptable*:

Definición 2.33 (línea de argumentación aceptable)
Una línea de argumentación $\Lambda = [\langle \mathcal{A}_1, L_1\rangle, \dots, \langle \mathcal{A}_i, L_i\rangle, \dots, \langle \mathcal{A}_n, L_n\rangle]$ será aceptable si:

1. Λ es una secuencia finita.
2. El conjunto Λ_S de estructuras de argumento de soporte es concordante, y el conjunto Λ_I, de estructuras argumentos de interferencia de Λ, también es concordante. Un conjunto $\{\langle \mathcal{A}_i, L_i\rangle\}_{i=1}^{n}$ se dice *concordante*, si el conjunto $\Pi \cup \bigcup_{i=1}^{n} \mathcal{A}_i$ no es contradictorio.
3. Ningún argumento $\langle \mathcal{A}_k, L_k\rangle$ de Λ es un sub-argumento de una estructura de argumento $\langle \mathcal{A}_i, L_i\rangle$ que aparece previamente en Λ $(i < k)$.
4. Para toda estructura $\langle \mathcal{A}_i, L_i\rangle$ de Λ tal que $\langle \mathcal{A}_i, L_i\rangle$ es un derrotador de bloqueo de $\langle \mathcal{A}_{i-1}, L_{i-1}\rangle$, si $\langle \mathcal{A}_{i+1}, L_{i+1}\rangle$ existe, entonces $\langle \mathcal{A}_{i+1}, L_{i+1}\rangle$ es un derrotador propio de $\langle \mathcal{A}_i, L_i\rangle$.

Para determinar si un argumento $\langle \mathcal{A}_0, L_0\rangle$ no está derrotado, no siempre alcanza con construir una línea de argumentación aceptable. Como se muestra en la siguiente sección, para un argumento $\langle \mathcal{A}_0, L_0\rangle$ pueden existir varios derrotadores y cada uno debe ser considerado en una línea de argumentación diferente.

2.2.6. Árboles de dialéctica y literales garantizados

El siguiente ejemplo muestra que en general para cada estructura de argumento puede existir más de un derrotador. La presencia de múltiples derrotadores para una estructura

de argumento produce una ramificación de líneas de argumentación, dando origen a un *árbol de derrotadores* que se denomina *árbol de dialéctica*. En este árbol, cada camino desde la raíz hasta una hoja corresponde a un línea de argumentación.

Ejemplo 2.17 Considérese el siguiente programa lógico rebatible:

$a \prec b$	$\sim d \prec k$	$\sim b \prec c, f$	$\sim f \prec i$
$b \prec c$	e	$f \prec g$	i
c	$\sim b \prec e$	g	$\sim h \prec k$
$\sim b \prec c, d$	$h \prec j$	k	
$d \prec g$	j	$\sim f \prec g, h$	

A partir de este programa, es posible construir una estructura de argumento $\langle \mathcal{A}, a \rangle$, con $\mathcal{A}$=$\{$ $(a \prec b)$, $(b \prec c)$ $\}$. Si se utiliza el criterio de especificidad para comparar estructuras de argumentos, es posible construir tres derrotadores para $\langle \mathcal{A}, a \rangle$ que atacan indirectamente en el literal b:

- $\langle \mathcal{B}_1, \sim b \rangle$, con $\mathcal{B}_1$= $\{$ $(\sim b \prec c, d)$ $\}$
- $\langle \mathcal{B}_2, \sim b \rangle$, con $\mathcal{B}_2$= $\{$ $(\sim b \prec c, f)$, $(f \prec g)$ $\}$
- $\langle \mathcal{B}_3, \sim b \rangle$, con $\mathcal{B}_3$= $\{(\sim b \prec e)$ $\}$

La estructura $\langle \mathcal{B}_1, \sim b \rangle$, a su vez, tiene un derrotador $\langle \mathcal{C}_1, \sim d \rangle$ con $\mathcal{C}_1$= $\{\sim d \prec k\}$. La estructura $\langle \mathcal{B}_2, \sim b \rangle$ tiene dos derrotadores: $\langle \mathcal{C}_2, \sim f \rangle$ con $\mathcal{C}_2$= $\{(\sim f \prec i)\}$, y $\langle \mathcal{C}_3, \sim f \rangle$ con $\mathcal{C}_3$= $\{(\sim f \prec g, h)$, $(h \prec j)\}$. Finalmente, la estructura $\langle \mathcal{C}_3, \sim f \rangle$ tiene el derrotador $\langle \mathcal{D}_1, \sim h \rangle$, donde $\mathcal{D}_1$= $\{(\sim h \prec k)\}$.

En el ejemplo anterior existen cuatro líneas de argumentación aceptables a partir de $\langle \mathcal{A}, a \rangle$:

$\Lambda_1 = [\langle \mathcal{A}, a \rangle, \langle \mathcal{B}_1, \sim b \rangle, \langle \mathcal{C}_1, \sim d \rangle]$
$\Lambda_2 = [\langle \mathcal{A}, a \rangle, \langle \mathcal{B}_2, \sim b \rangle, \langle \mathcal{C}_2, \sim f \rangle]$
$\Lambda_3 = [\langle \mathcal{A}, a \rangle, \langle \mathcal{B}_2, \sim b \rangle, \langle \mathcal{C}_3, \sim f \rangle, \langle \mathcal{C}_3, \sim h \rangle]$
$\Lambda_4 = [\langle \mathcal{A}, a \rangle, \langle \mathcal{B}_3, \sim b \rangle]$

Estas cuatro líneas pueden representarse como un árbol, donde la raíz está etiquetada con $\langle \mathcal{A}, a \rangle$ y los nodos internos representan derrotadores de su nodo padre. Las hojas

del árbol corresponden a estructuras de argumento sin derrotadores. Este tipo de árboles se llama árbol de dialéctica, y se define a continuación. La Figura 2.11 muestra el árbol correspondiente a este ejemplo.

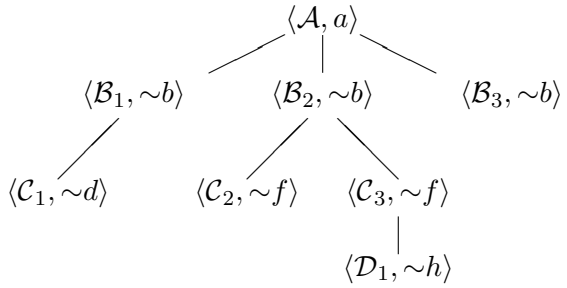

Figura 2.11: Árbol de dialéctica del ejemplo 2.17

Definición 2.34 (Arbol de dialéctica)
Sea $\langle \mathcal{A}_0, L_0 \rangle$ una estructura de argumento obtenida a partir de un programa $\mathcal{P}$. Un árbol de dialéctica para $\langle \mathcal{A}_0, L_0 \rangle$, a partir de $\mathcal{P}$, se denota $\mathcal{T}_{\langle \mathcal{A}_0, L_0 \rangle}$, y se construye de la siguiente forma:

1. La raíz del árbol es etiquetada con $\langle \mathcal{A}_0, L_0 \rangle$.
2. Sea N un nodo del árbol etiquetado $\langle \mathcal{A}_n, L_n \rangle$, y la secuencia de etiquetas $[\langle \mathcal{A}_0, L_0 \rangle, \langle \mathcal{A}_1, L_1 \rangle, \langle \mathcal{A}_2, L_2 \rangle, \ldots, \langle \mathcal{A}_n, L_n \rangle]$ del camino que va desde la raíz hasta el nodo N.
   Sean $\langle \mathcal{B}_1, Q_1 \rangle$, $\langle \mathcal{B}_2, Q_2 \rangle$, ..., $\langle \mathcal{B}_k, Q_k \rangle$ todos los derrotadores de $\langle \mathcal{A}_n, L_n \rangle$.
   Para cada derrotador $\langle \mathcal{B}_i, Q_i \rangle$ $(1 \leq i \leq k)$, tal que, la línea de argumentación $\Lambda = [\langle \mathcal{A}_0, L_0 \rangle, \langle \mathcal{A}_1, L_1 \rangle, \langle \mathcal{A}_2, L_2 \rangle, \ldots, \langle \mathcal{A}_n, L_n \rangle, \langle \mathcal{B}_i, Q_i \rangle]$ sea aceptable, existe un nodo hijo N_i de N etiquetado con $\langle \mathcal{B}_i, Q_i \rangle$.
   Si no existe ningún derrotador para $\langle \mathcal{A}_n, L_n \rangle$ o no existe un derrotador $\langle \mathcal{B}_i, Q_i \rangle$ tal que Λ sea aceptable, entonces el nodo N es una hoja.

Como puede observarse en la Figura 2.11, los nodos hoja del árbol de dialéctica corresponden a argumentos no derrotados. En cambio, un nodo interno que tiene como hijo un nodo hoja, corresponderá a un argumento derrotado. Siguiendo con este análisis, desde las hojas, hacia la raíz, los nodos en un árbol de dialéctica se pueden marcar como "D" *derrotado*, o "U" *no derrotado*[5]. A continuación se especifica el procedimiento de marcado:

[5]En inglés *undefeated*

Procedimiento 2.1 (Marcado de un árbol de dialéctica)
Sea $\mathcal{T}_{\langle \mathcal{A}, L\rangle}$ un árbol de dialéctica para $\langle \mathcal{A}, L\rangle$. Un árbol de dialéctica marcado, denotado $\mathcal{T}^*_{\langle \mathcal{A}, L\rangle}$ puede obtenerse marcando cada nodo en $\mathcal{T}_{\langle \mathcal{A}, L\rangle}$ de la siguiente forma:

1. Todas las hojas de $\mathcal{T}_{\langle \mathcal{A}, L\rangle}$ se marcan con "U" en $\mathcal{T}^*_{\langle \mathcal{A}, L\rangle}$.
2. Sea N un nodo interno de $\mathcal{T}_{\langle \mathcal{A}, L\rangle}$. El nodo N se marca con "U" si todo nodo hijo de N está marcado con "D", y N se marca con "D" si existe al menos un nodo hijo de N marcado con "U".

Este procedimiento sugiere un proceso desde las hojas a la raíz (bottom-up) para marcar los nodos, y determinar el estado de la raíz. La Figura 2.12 muestra el árbol de la figura 2.11 después de aplicarle el procedimiento de marcado.

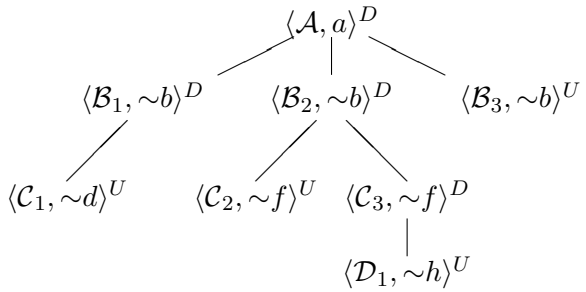

Figura 2.12: Árbol de dialéctica marcado del ejemplo 2.17

Un árbol de dialéctica marcado $\mathcal{T}^*_{\langle \mathcal{A}, L\rangle}$ representa el análisis dialéctico en el cuál todos los argumentos construibles a partir de un programa $\mathcal{P}$ son considerados a fin de decidir el status de un argumento $\langle \mathcal{A}, L\rangle$. En un árbol de dialéctica marcado $\mathcal{T}^*_{\langle \mathcal{A}, L\rangle}$, los nodos marcados con "U" corresponden a argumentos no derrotados y los nodos marcados con "D" a estructuras de argumento derrotadas. Por lo tanto, el status de un argumento $\langle \mathcal{A}, L\rangle$ será "derrotado" si la raíz de $\mathcal{T}^*_{\langle \mathcal{A}, L\rangle}$ queda marcada con "D", o "garantizado"[6] si la raíz tiene como marca "U".

La siguiente definición introduce el concepto que resume todo el análisis dialéctico en la programación en lógica rebatible. Decir que un literal "L" está garantizado a partir de un programa $\mathcal{P}$, significa que un razonador que utilice el programa $\mathcal{P}$, podrá creer en "L".

[6]En inglés *warrant*.

Definición 2.35 (Literales garantizados)
Sea $\mathcal{P}$=(Π, Δ), un programa lógico rebatible, y L un literal. Sea $\langle \mathcal{A}, L\rangle$ una estructura de argumento para L, y $\mathcal{T}^*_{\langle \mathcal{A}, L\rangle}$ el árbol de dialéctica marcado asociado a $\langle \mathcal{A}, L\rangle$. El literal L está *garantizado* si la raíz de $\mathcal{T}^*_{\langle \mathcal{A}, L\rangle}$ está marcada con "U".

Ejemplo 2.18 Considere el programa $\mathcal{P}_{2,9}$ del ejemplo 2.9. Como se móstro en el ejemplo 2.15, $\langle \mathcal{A}_3, \sim peligroso(i_1)\rangle$ es un derrotador propio para $\langle \mathcal{A}_4, peligroso(i_1)\rangle$, y $\langle \mathcal{A}_1, \sim encendida(l_1)\rangle$ es un derrotador de bloqueo para $\langle \mathcal{D}, encendida(l_1)\rangle$ y viceversa.

El literal "$\sim peligroso(i_1)$" está garantizado a partir de $\mathcal{P}_{2,9}$ porque existe un argumento $\langle \mathcal{A}_3, \sim peligroso(i_1)\rangle$ que no tiene derrotadores. El literal "$peligroso(i_1)$" no está garantizado porque existe un único argumento $\langle \mathcal{A}_4, peligroso(i_1)\rangle$, el cual está derrotado por $\langle \mathcal{A}_3, \sim peligroso(i_1)\rangle$ que no tiene derrotadores.

Por otra parte, los literales "$\sim encendida(l_1)$" y "$encendida(l_1)$" no están garantizados a partir de $\mathcal{P}_{2,9}$. Para ambos literales existe un único argumento, $\langle \mathcal{A}_1, \sim encendida(l_1)\rangle$ y para $\langle \mathcal{D}, encendida(l_1)\rangle$, los cuales se derrotan mutuamente.

La siguiente proposición muestra que en particular los literales que son hechos de un programa $\mathcal{P}$, o tienen una derivación estricta a partir de $\mathcal{P}$, siempre están garantizados a partir de $\mathcal{P}$.

Proposición 2.5 Sea $\mathcal{P}$= (Π, Δ) un programa lógico rebatible. Si el literal L tiene una derivación estricta, entonces L está garantizado.
Demostración: Si un literal L posee una derivación estricta, entonces por la Proposición 2.1 existe una estructura de argumento $\langle \mathcal{A}, L\rangle$, con $\mathcal{A} = \emptyset$. Por la Proposición 2.2 no existe contra-argumento para $\mathcal{A}$, con lo cual, el árbol de dialéctica $\mathcal{T}^*_{\langle \mathcal{A}, L\rangle}$ tiene un único nodo raíz y hoja, que es marcado como nodo "U". Por lo tanto, L está garantizado.

2.2.7. Programas Lógicos Extendidos: negación default

La *negación default* (también llamada *negación por falla*) denotada habitualmente con el símbolo "*not*", es utilizada ampliamente en programación en lógica, bases de datos deductivas, y en formalismos de razonamiento no-monótono. En DeLP "*not F*" *será asumido ante la falta de suficiente evidencia en su contra*, esto es, cuando F *no* esta garantizado. Como se describió anteriormente, DeLP utiliza la negación fuerte denotada con el símbolo

"$\sim$". A diferencia de la negación default un literal "$\sim F$" no puede ser asumido, sino que para creer "$\sim F$" el literal "$\sim F$" debe estar *garantizado*.

En lo que sigue se presenta una extensión de DeLP que permite el uso de la negación default solamente en el cuerpo de las reglas rebatibles. Para esto, la sintaxis de los programas DeLP se debe extender de la siguiente forma:

Definición 2.36 (Literal extendido) Un literal extendido es un literal L, o un literal precedido por el símbolo de la negación default "*not* L".

Definición 2.37 (Regla Rebatible Extendida)
Una *regla rebatible extendida* es un par ordenado, denotado "$L_0 -\prec X_1, \ldots, X_n$". La *Cabeza* L_0 es un literal fijo y el cuerpo $\{X_1, \ldots, X_n\}$ $(n > 0)$ es un conjunto finito no vacío de literales extendidos.

Un *programa lógico rebatible extendido* será entonces un conjunto posiblemente infinito de hechos, reglas estrictas y reglas rebatibles extendidas. Igual que antes, se denotará $\mathcal{P}$=(Π, Δ), para identificar al conjunto Π de hechos y reglas estrictas, y al conjunto Δ de reglas rebatibles extendidas del programa $\mathcal{P}$.

Como se mencionó anteriormente, en DeLP un literal extendido "*not* L" será asumido si L no puede ser garantizado por el proceso de dialéctica. Es decir, si L está garantizado, entonces "*not* L" no puede asumirse, y por el contrario, si toda estructura de argumento $\langle \mathcal{A}, L \rangle$ es derrotada, entonces "*not* L" podrá asumirse. Dado que asumir "*not* L" implica un análisis dialéctico, en una derivación rebatible un literal extendido "*not* L" se "asumirá" dejando que luego el proceso dialéctico decida si L está garantizado o no. A continuación figura la definición de derivación rebatible modificada para las reglas extendidas.

Definición 2.38 (Derivación Rebatible con reglas extendidas)
Sea $\mathcal{P}$= (Π, Δ) un programa lógico rebatible extendido y L un literal. Una derivación rebatible para L a partir de $\mathcal{P}$ consiste de una secuencia finita de literales fijos $L_1, L_2, \ldots L_n = L$, provisto de que exista una secuencia de reglas (estrictas o rebatibles) extendidas $R_1, R_2, \ldots R_n$ del programa $\mathcal{P}$, de tal forma que para cada literal L_i en la secuencia tenemos que:

(a) L_i es un hecho, o

(b) existe en $\mathcal{P}$ una regla R_i con cabeza L_i y cuerpo $B_1, B_2, \ldots B_k$ donde todo elemento del cuerpo, que no es un literal precedido por la negación default, es un elemento de la secuencia que precede a L_i.

Ejemplo 2.19 Considere el siguiente programa lógico rebatible extendido $\mathcal{P}_{2,19} = (\Pi_{2.9}, \Delta_{2.9} \cup X)$ que resulta de incorporar el conjunto de reglas X al programa del ejemplo 2.9, donde:

$$X = \left\{ \begin{array}{l} repuesto(L) -\!\prec lampara(L), not\ en\_uso(L). \\ en\_uso(L) -\!\prec conectado(I, L). \\ \sim en\_uso(L) -\!\prec conectado(I, L), roto(I). \\ \sim repuesto(L) -\!\prec lampara(L), rota(L). \end{array} \right\}$$

A partir de este programa existe una derivación rebatible para el literal "$repuesto(l_1)$" obtenida utilizando el hecho "$lampara(l_1)$" y la regla "$repuesto(l_1) -\!\prec lampara(l_1), not\ en\_uso(l_1)$". El literal extendido "$not\ \ en\_uso(l_1)$" es simplemente asumido por la derivación rebatible. Como se mostrará a continuación, el proceso de dialéctica considerará luego si está bien asumirlo o no.

La definición de estructura de argumento también debe ser extendida para considerar negación default. Considérese el programa $\mathcal{P}=(\emptyset, \{p -\!\prec not\ p\})$. A partir de $\mathcal{P}$ es posible obtener una derivación rebatible para el literal "p". Sin embargo, no debería ser posible obtener una estructura de argumento para "p", habiendo asumido "$not\ p$". A continuación se incluye la definición extendida de argumento, donde la condición 2 impide la construcción de este tipo de argumentos.

Definición 2.39 (Estructura de argumento para DeLP extendida)
Sea L un literal y $\mathcal{P}=(\Pi, \Delta)$ un programa lógico rebatible, una *estructura de argumento* para L es un par $\langle \mathcal{A}, L \rangle$, donde $\mathcal{A}$ es conjunto de reglas rebatibles de Δ, tal que:

1. existe una derivación rebatible para L a partir de $\Pi \cup \mathcal{A}$.
2. si H es un literal de la derivación rebatible de L, entonces $not\ H$ no pertenece al cuerpo de ninguna regla rebatible de $\mathcal{A}$,
3. $\Pi \cup \mathcal{A}$ es no contradictorio, y
4. $\mathcal{A}$ es minimal, es decir, no existe un subconjunto propio $\mathcal{A}'$ de $\mathcal{A}$ tal que $\mathcal{A}'$ satisface las condiciones anteriores.

Ejemplo 2.20 Considere el programa lógico rebatible extendido $\mathcal{P}_{2,19}$ definido en el ejemplo 2.19. A partir de $\mathcal{P}_{2,19}$ se puede construir la estructura de argumento $\langle \mathcal{A}, repuesto(l_1) \rangle$, donde:

$$\mathcal{A} = \{repuesto(l_1) \prec lampara(l_1), not\ en\_uso(l_1)\}$$

Como se describió en la sección 2.2.5 el ataque de argumentos está basado en el desacuerdo de literales contradictorios con respecto a la negación fuerte. En los programas lógicos extendidos los literales negados con negación default son "*suposiciones*" (assumptions) sobre las cuales la derivación está basada, y serán también un punto de ataque contra el argumento. Esto es, un argumento como $\mathcal{B}$, basado en la suposición *not* L, puede ser atacado con un argumento para L.

A continuación se presenta la noción de derrota que incorpora este tipo de ataque entre argumentos. A diferencia del concepto de derrota definido en la sección 2.2.5 cuando se ataca una suposición, los argumentos involucrados no son comparados.

Definición 2.40 (Ataque a una suposición)
Sea $\mathcal{P} = (\Pi, \Delta)$ un programa lógico rebatible extendido, y sean $\langle \mathcal{B}, Q \rangle$ y $\langle \mathcal{A}, L \rangle$ dos estructuras de argumento obtenidas a partir de $\mathcal{P}$. $\langle \mathcal{A}, L \rangle$ es un ataque a una suposición de $\langle \mathcal{B}, Q \rangle$, si el literal *not* L es parte del cuerpo de una regla de $\langle \mathcal{B}, Q \rangle$.

Definición 2.41 (Derrotador en DeLP extendida)
Sean $\langle \mathcal{A}_1, L_1 \rangle$ y $\langle \mathcal{A}_2, L_2 \rangle$ dos estructuras de argumento. La estructura $\langle \mathcal{A}_1, L_1 \rangle$ es un *derrotador* para $\langle \mathcal{A}_1, L_1 \rangle$, si se cumple alguna de las siguientes condiciones:
(a) $\langle \mathcal{A}_1, L_1 \rangle$ es un derrotador propio para $\langle \mathcal{A}_2, L_2 \rangle$, o
(b) $\langle \mathcal{A}_1, L_1 \rangle$ es un derrotador de bloqueo para $\langle \mathcal{A}_2, L_2 \rangle$, o
(c) $\langle \mathcal{A}_1, L_1 \rangle$ es un ataque a una suposición de $\langle \mathcal{A}_2, L_2 \rangle$.

Ejemplo 2.21 Como se vio en el ejemplo 2.20, a partir del programa lógico extendido $\mathcal{P}_{2,19}$ se puede construir la estructura de argumento $\langle \mathcal{A}, repuesto(l_1) \rangle$, donde:

$$\mathcal{A} = \{repuesto(l_1) \prec lampara(l_1), not\ en\_uso(l_1)\}$$

Sin embargo, $\langle \mathcal{A}, repuesto(l_1) \rangle$ es atacada en la suposición, por el derrotador $\langle \mathcal{B}, en\_uso(l_1) \rangle$, donde

$$\mathcal{B} = \{en\_uso(l_1) \prec conectado(i_1, l_1).\}$$

Por su parte, la estructura de argumento $\langle \mathcal{B}, en\_uso(l_1) \rangle$, tiene un derrotador propio $\langle \mathcal{C}, \sim en\_uso(l_1) \rangle$, donde

$$\mathcal{C} = \{\sim en\_uso(l_1) -\!\prec conectado(i_1, l_1), roto(i_1).\}$$

Como $\langle \mathcal{C}, \sim en\_uso(l_1) \rangle$ no está derrotada, entonces $\langle \mathcal{B}, en\_uso(l_1) \rangle$ está derrotado, y el literal *not en_uso* puede asumirse sin problemas. De esta manera, el literal "$repuesto(l_1)$" queda garantizado.

En el ejemplo 2.21 puede verse claramente que en DeLP extendida, aunque un literal P tenga una derivación rebatible, el literal extendido *not* P puede asumirse exitosamente, si P no está garantizado.

Capítulo 3

Argumentación Rebatible, Acciones y Problemas de planificación

En este capítulo se define un formalismo para modelar dominios y problemas de planificación que combina acciones y argumentación rebatible. Lo novedoso de este formalismo es que permite definir acciones y representar conocimiento acerca del dominio utilizando la Programación en Lógica Rebatible. Por un lado, el conocimiento y la argumentación rebatible se utilizan para razonar acerca de las precondiciones, restricciones y efectos de las acciones. Por otra parte, las acciones son utilizadas para modificar el mundo y poder lograr las metas. Este formalismo está basado en las ideas publicadas en [GSG07, GGS08]. Algunos conceptos son introducidos por primera vez en esta Tesis, mientras que otros serán extendidos o redefinidos.

El capítulo está organizado de la siguiente manera: En la sección 3.1 se introduce la sintaxis del formalismo y se definen los conceptos básicos de *estado*, *esquema de acción*, *acción* y *dominio de planificación*. En la sección 3.2 se introduce el concepto de *acción aplicable* que describe las condiciones que deben cumplirse para que una acción pueda ser ejecutada en un estado particular Ψ. Luego, en la sección 3.3 se define el efecto que se produce sobre un estado al ejecutar una acción aplicable o una secuencia aplicable de acciones (plan). Por último, en la sección 3.4 se define el concepto de *problema de planificación* para dominios DAKAR y se establece cuando una secuencia de acciones constituye una solución para un problema de planificación.

3.1. Formalismo de representación DAKAR

El nombre DAKAR proviene de las siglas en inglés de: *Defeasible Argumentation for Knowlegde and Action Representation*. Este formalismo utiliza la programación en lógica rebatible (DeLP) (Sección 2.2) como paradigma lógico para representar conocimiento del dominio de planificación y razonar rebatiblemente en un estado particular del mundo.

Definición 3.1 [Conjunto consistente Estado]
En DAKAR un estado del mundo (o simplemente estado), se representa por medio de un conjunto consistente de literales Ψ. Un conjunto de literales es *consistente* si no es contradictorio (Definición 2.23).

Los literales presentes en un estado representan hechos que son verdaderos en el estado del mundo representado.

Ejemplo 3.1 Considere un dominio de planificación donde un agente se encarga de la limpieza y el mantenimiento de una oficina. En la misma existe una ventana con persiana y dos interruptores i_1 e i_2 conectados a dos lámparas l_1 y l_2 respectivamente. El estado donde la persiana de la ventana se encuentra baja, el interruptor i_1 se encuentra encendido y el interruptor i_2 se encuentra apagado puede representarse en DAKAR a través del siguiente conjunto consistente de hechos:

$$\Psi_{3.1} = \{\sim pers\_alta, int(i_1), enc(i_1), lamp(l_1), con(i_1, l_1),$$
$$int(i_2), \sim enc(i_2), lamp(l_2), con(i_2, l_2)\}$$

El literal $pers\_alta$ se utiliza para representar el hecho de que la persiana se encuentra alta. Dado que la persiana sólo puede estar alta o baja y estos hechos son excluyentes, $\sim pers\_alta$ representa que la persiana se encuentra baja. Algo similar ocurre con los interruptores. El literal $enc(i_1)$ representa que el interruptor i_1 está encendido mientras que $\sim enc(i_2)$ representa que el interruptor i_2 está apagado. Los literales $int(I)$ y $lamp(L)$ se utilizan para distinguir los objetos presentes en el dominio. El literal $int(i_1)$ representa que i_1 es un interruptor, mientras que $lamp(l_1)$ representa que l_1 es una lámpara. Por último, el literal $con(i_1, l_1)$ representa que el interruptor i_1 está conectado a la lámpara l_1.

El conocimiento del dominio estará definido por un conjunto de *reglas rebatibles extendidas*(definición 2.37) Δ. Este conocimiento se utilizará junto con la representación de un estado Ψ, para formar un programa lógico rebatible (Ψ, Δ) que permitirá razonar rebatiblemente utilizando DeLP.

Ejemplo 3.2 Para el dominio presentado en el ejemplo 3.1 el agente puede contar con el siguiente conocimiento:

$$\Delta_{3.2} = \left\{ \begin{array}{c} luz -\prec luz\_natural, \\ luz -\prec luz\_artificial, \\ luz\_natural -\prec pers\_alta, es\_dia, \\ luz\_artificial -\prec enc(L), lamp(L), \\ peligroso(I) -\prec int(I), mojado(I), \\ peligroso(I) -\prec int(I), roto(I), \\ \sim peligroso(I) -\prec int(I), \sim electricidad, \\ enc(L) -\prec con(I, L), enc(I), \\ \sim enc(L) -\prec lamp(L), \sim electricidad, \\ \sim enc(L) -\prec lamp(L), rota(L), \\ rep(L) -\prec lamp(L), not\ en\_uso(L), \\ en\_uso(L) -\prec con(I, L), \\ \sim rep(L) -\prec lamp(L), rota(L) \end{array} \right\}$$

Como se mencionó en el sección 2.2, las *reglas rebatibles* se utilizan para representar conocimiento rebatible, esto es, información tentativa que puede utilizarse en la medida que no exista información que la contradiga. Por ejemplo, la regla "$enc(L) -\prec con(I, L), enc(I)$" expresa que *creer en que el interruptor I está encendido y que está conectado a la lámpara L provee razones para creer que la lámpara L está encendida*. Por otra parte la regla "$\sim enc(L) -\prec lamp(L), rota(L)$" expresa que *creer en que la lámpara L está rota provee razones para creer que la lámpara no está encendida*. La regla $rep(L) -\prec lamp(L), not\ en\_uso(L)$, expresa que una lámpara L puede ser usada como repuesto si no está en uso, y la regla $\sim rep(L) -\prec lamp(L), rota(L)$ indica que una lámpara L no puede ser usada como repuesto si está rota.

El conocimiento $\Delta_{3.2}$ puede utilizarse para razonar rebatiblemente en un estado del mundo. Considere el estado $\Psi_{3.1}$ definido en el ejemplo 3.1. A partir del programa lógico rebatible $(\Psi_{3.1}, \Delta_{3.2})$ se puede garantizar que hay luz en la oficina dado que es posible

construir el siguiente argumento no derrotado:

$$\langle\{(luz -\!\prec luz\_artificial), (luz\_artificial -\!\prec enc(l_1), lamp(l_1)),$$
$$(enc(l_1) -\!\prec con(i_1, l_1), enc(i_1))\}, luz\rangle$$

Observación 3.1 Note que el formalismo DAKAR utiliza un subconjunto de DeLP dado que no se consideran reglas estrictas. Como se mencionó anteriormente, el conocimiento del dominio Δ consiste únicamente de reglas rebatibles, y se utiliza junto con la representación de un estado Ψ para formar un programa lógico rebatible (Ψ,Δ) que permite razonar rebatiblemente. El programa (Ψ,Δ) es un caso particular de un programa lógico rebatible, donde el conocimiento estricto consiste únicamente de los hechos mencionados en la descripción del estado y no contiene reglas estrictas.

El problema que se presenta con las reglas estrictas es que al combinarse con la descripción de un estado se puede producir un conjunto de información estricta contradictorio, lo cual impide formar un programa rebatible válido (ver observación 2.5). Dado que los dominios de planificación son dinámicos e involucran estados que cambian por la ejecución de acciones, si se utilizan reglas estrictas no es posible garantizar que siempre se pueda construir un programa válido para razonar en cada estado. Por otra parte, una regla rebatible que no sea atacada se comporta como una regla estricta.

Además del conocimiento, el formalismo permite definir un conjunto Γ de acciones disponibles en el dominio que pueden utilizarse para modificar el estado del mundo.

Definición 3.2 [Esquema de Acción]
Un *esquema de acción* es una quíntupla ordenada $\langle \mathsf{N}, \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C}\rangle$ donde:

- N es un átomo que representa el nombre de la acción.
- A es un conjunto consistente de literales, que representan los *efectos positivos* de N.
- D es un conjunto de *efectos negativos* de la forma $-D$, donde D es un literal y $D \notin \mathsf{A}$.
- P es un conjunto consistente de literales que representan las *precondiciones* de N.
- C es un conjunto de *restricciones* de la forma *not* C[1], donde C es un literal y $C \notin \mathsf{P}$.

[1]recordemos que *not* C representa "C*no está garantizado*"

Intuitivamente un *esquema de acción* representa el conjunto de todas las instancias de acciones posibles que se pueden generar sustituyendo las variables por terminos fijos. Al igual que en DeLP las variables se denotarán con una letra inicial mayúscula para distinguirlas de otros elementos del lenguaje.

Definición 3.3 [**Acción**]
Una *acción* es un *esquema de acción instanciado*, es decir, un esquema de acción donde todos sus átomos y literales son fijos. Una acción se obtiene a partir de un esquema de acción sustituyendo todas las variables por términos fijos, de forma que todas las apariciones de una misma variable sean reemplazadas por el mismo valor.

Nota: Sea Γ un conjunto de esquemas de acción y sea $\langle \mathsf{N}_1, \mathsf{A}_1, \mathsf{D}_1, \mathsf{P}_1, \mathsf{C}_1 \rangle$ una acción. En adelante se utilizará $\langle \mathsf{N}_1, \mathsf{A}_1, \mathsf{D}_1, \mathsf{P}_1, \mathsf{C}_1 \rangle \in \Gamma$ para indicar que la acción se obtiene a partir de un esquema de acción $\langle \mathsf{N}, \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C} \rangle \in \Gamma$.

Una acción $\langle \mathsf{N}, \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C} \rangle$ debe interpretarse intuitivamente de la siguiente manera: *si todos los literales del conjunto* P *están garantizados y los del conjunto* C *no están garantizados en un estado* Ψ*, entonces al ejecutar* N *se agregarán al estado* Ψ *los literales de* A *y los literales de* D *serán eliminados*. Más adelante se definirá formalmente cuando una acción es aplicable y el resultado de su ejecución.

Nota 3.1 (notación alternativa) Una acción o esquema de acción $\langle \mathsf{N}, \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C} \rangle$ se representará también de la siguiente forma:

$$\{A_1, \ldots, A_n\}, \{D_1, \ldots, D_m\}^- \xleftarrow{\mathsf{N}} \{P_1, \ldots, P_k\}, not\ \{C_1, \ldots, C_t\}$$

donde:

- $\{A_1, \ldots, A_n\} = \mathsf{A}$, es el conjunto de efectos positivos.
- $\{D_1, \ldots, D_m\}^- = \{-D_1, \ldots, -D_m\} = \mathsf{D}$, es el conjunto de efectos negativos.
- $\{P_1, \ldots, P_k\} = \mathsf{P}$, es el conjunto de precondiciones.
- $not\ \{C_1, \ldots, C_t\} = \{not\ C_1, \ldots, not\ C_t\} = \mathsf{C}$, es el conjunto de restricciones.

Ejemplo 3.3 Consideremos nuevamente el dominio introducido en el ejemplo 3.1. Supongamos que para hacer la limpieza y el mantenimiento de la oficina el agente cuenta con el siguiente conjunto esquemas de acción:

$$\Gamma_{3.3} = \left\{ \begin{array}{c} \{pers\_alta\}, \{\}^- \xleftarrow{subir\_pers} \{\sim pers\_alta\}, not\ \{\} \\ \{\sim pers\_alta\}, \{\}^- \xleftarrow{bajar\_pers} \{pers\_alta\}, not\ \{\} \\ \{enc(I)\}, \{\}^- \xleftarrow{encender(I)} \{\sim enc(I), int(I)\}, not\ \{peligroso(I)\} \\ \{\sim enc(I)\}, \{\}^- \xleftarrow{apagar(I)} \{enc(I), int(I)\}, not\ \{peligroso(I)\} \\ \{con(I,R)\}, \{con(I,L)\}^- \xleftarrow{cambiar(L,I,R)} \{con(I,L), rep(R), \sim enc(I)\}, not\ \{\} \\ \{oficina\_limpia\}, \{\}^- \xleftarrow{limpiar} \{luz\}, not\ \{\} \end{array} \right\}$$

La acción *subir_pers* permite subir la persiana de la ventana cuando esta se encuentra baja y la acción *bajar_pers* permite bajarla cuando esta se encuentra alta.

El esquema de acción $encender(I)$ puede utilizarse para encender cualquier interruptor I, siempre y cuando I se encuentre apagado ($\sim enc(I)$) y no halla evidencia de que sea peligroso ($not\, peligroso(I)$). Reemplazando la variable I por la constante i_1 obtenemos la siguiente acción $encender(i_1)$ que corresponde a encender el interruptor i_1:

$$\{enc(i_1)\}, \{\}^- \xleftarrow{encender(i_1)} \{\sim enc(i_1), int(i_1)\}, not\ \{peligroso(i_1)\}$$

El esquema de acción $apagar(I)$ permite apagar un interruptor I que se encuentre encendido y no resulte peligroso.

El esquema de acción $cambiar(L, I, R)$ permite reemplazar una lámpara L siempre y cuando el interruptor I al cual está conectada esté apagado y se cuente con una lámpara de repuesto R.

Por último la acción *limpiar* permite dejar la oficina limpia siempre y cuando haya luz para hacerlo.

Como vimos, el formalismo permite definir conocimiento del dominio para razonar rebatiblemente y un conjunto de acciones disponibles para modificar el mundo. Un dominio de planificación en DAKAR está definido conjuntamente por el conocimiento y las acciones.

Definición 3.4 [Dominio de Planificación DAKAR]
Un *dominio de planificación* es una dupla (Δ,Γ) donde:

- Δ es un conjunto de *reglas rebatibles extendidas*.

- Γ es un conjunto de *esquemas de acción*.

Ejemplo 3.4 Considere el conjunto $\Delta_{3.2}$ de reglas rebatibles extendidas definido en el ejemplo 3.2 y el conjunto $\Gamma_{3.3}$ de esquemas de acción definido en el ejemplo 3.3. La dupla $(\Delta_{3.2}, \Gamma_{3.3})$ define el dominio de planificación del agente encargado de la limpieza introducido a través de los ejemplos 3.1, 3.2 y 3.3.

3.2. Acciones aplicables

En una acción $\langle \mathsf{N}, \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C} \rangle$ los conjuntos P y C definen las condiciones que deben cumplirse para que sea aplicable (esto es, pueda ser ejecutada) en un estado Ψ particular del mundo. Estas condiciones serán evaluadas razonando rebatiblemente con (Ψ, Δ) utilizando DeLP. El conjunto P menciona los literales que deben estar garantizados y C menciona los literales que *n*o deben estar garantizados. De esta forma, la ejecución de una acción puede depender de que cierta información sea desconocida (esto es, *no esté garantizada*).

Ejemplo 3.5 Consideremos por ejemplo la acción de encender el interruptor i_1 definida en el ejemplo 3.3: $\{enc(i_1)\}, \{\}^- \xleftarrow{encender(i_1)} \{\sim enc(i_1), int(i_1)\}, not\ \{peligroso(i_1)\}$. Las precondiciones de la acción establecen que para poder encender el interruptor i_1 en un estado Ψ este debe encontrarse apagado (esto es, $\sim enc(i_1)$ debe estar garantizado a partir de (Ψ,$\Delta_{3.2}$). Por otra parte, las restricciones de la acción establecen que no debe haber evidencia de que el interruptor i_1 sea peligroso (esto es, $peligroso(i_1)$ no debe estar garantizado a partir de (Ψ,$\Delta_{3.2}$)). Note que según el conocimiento del dominio $\Delta_{3.2}$ un interruptor resulta peligroso cuando está mojado o está roto (ya que podría producir un cortocircuito). Sin embargo, si no hay electricidad un interruptor no resulta peligroso.

Definición 3.5 [Acción aplicable]
Sea (Δ,Γ) un dominio de planificación y sea Ψ un estado. Una acción $\langle \mathsf{N}, \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C} \rangle$ es *aplicable* en el estado Ψ si:

- cada precondición $P_i \in \mathsf{P}$ está garantizada a partir de (Ψ, Δ), y
- para cada restricción *not* C_i en C, C_i no se puede garantizar a partir de (Ψ, Δ).

Nota 3.2 Según la definición 3.2 en una acción $\langle \mathsf{N}, \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C} \rangle$ si $P \in \mathsf{P}$ entonces $not\, P \notin \mathsf{C}$. Esto se debe a que en DeLP un literal no puede estar garantizado y no estar garantizado a partir del mismo programa (Ψ, Δ), por lo tanto si se permite que $P \in \mathsf{P}$ y $not\, P \in \mathsf{C}$ la acción $\langle \mathsf{N}, \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C} \rangle$ no sería aplicable en ningún estado.

Ejemplo 3.6 Considere el dominio de planificación $(\Delta_{3.2}, \Gamma_{3.3})$ del agente encargado de la limpieza definido en el ejemplo 3.4 y el estado $\Psi_{3.1}$ definido en el ejemplo 3.1, donde:

$$\begin{aligned}\Psi_{3.1}= \quad & \{\sim pers\_alta, int(i_1), enc(i_1), lamp(l_1), con(i_1, l_1), \\ & int(i_2), \sim enc(i_2), lamp(l_2), con(i_2, l_2)\}\end{aligned}$$

La acción *limpiar* es aplicable en $\Psi_{3.1}$ dado que su única precondición es "luz", y esta precondición está garantizada a partir de ($\Psi_{3.1}$, $\Delta_{3.2}$) ya que existe el argumento no derrotado $\langle \mathcal{A}, luz \rangle$, con:

$$\begin{aligned}\mathcal{A} = \{&(luz \prec luz\_artificial), (luz\_artificial \prec enc(l_1), lamp(l_1)), \\ &(enc(l_1) \prec con(i_1, l_1), enc(i_1))\}\end{aligned}$$

Obsérvese que para que una acción no sea aplicable pueden darse muchas situaciones, por ejemplo que todos los argumentos que existen para sustentar alguna precondición estén derrotados (y por ende la precondición no esté garantizada); o puede ocurrir que una precondición tenga un argumento derrotado pero ese derrotador esté a su vez derrotado, etc. El Ejemplo 3.7 a continuación, tiene como objetivo mostrar una gran cantidad de estos casos posibles, a fin de clarificar cuando una acción es aplicable o no.

Ejemplo 3.7 Considere un dominio de planificación proposicional $(\Delta_{3.7}, \Gamma_{3.7})$, donde:

$$\Delta_{3.7} = \left\{ \begin{array}{c} r \prec p, q \\ s \prec p \\ \sim s \prec q \\ t \prec o \\ \sim t \prec s \\ u \prec q \\ \sim u \prec p, q \end{array} \right\}$$

$$\Gamma_{3.7} = \left\{ \begin{array}{c} \{d\}, \{\}^- \xleftarrow{a_1} \{\}, not\ \{\} \\ \{e\}, \{\}^- \xleftarrow{a_2} \{p\}, not\ \{\} \\ \{d\}, \{\}^- \xleftarrow{a_3} \{\sim n\}, not\ \{\} \\ \{\sim e\}, \{\}^- \xleftarrow{a_4} \{r\}, not\ \{\} \\ \{d\}, \{\}^- \xleftarrow{a_5} \{s\}, not\ \{\} \\ \{d\}, \{\}^- \xleftarrow{a_6} \{\sim s\}, not\ \{\} \\ \{n\}, \{\}^- \xleftarrow{a_7} \{t\}, not\ \{\} \\ \{\sim p\}, \{\}^- \xleftarrow{a_8} \{\}, not\ \{c\} \\ \{d\}, \{\}^- \xleftarrow{a_9} \{\}, not\ \{s\} \\ \{d\}, \{\}^- \xleftarrow{a_{10}} \{\}, not\ \{\sim s\} \\ \{d\}, \{\}^- \xleftarrow{a_{11}} \{\sim u\}, not\ \{\} \\ \{d\}, \{\}^- \xleftarrow{a_{12}} \{\}, not\ \{u\} \\ \{\}, \{p\}^- \xleftarrow{a_{13}} \{p, r\}, not\ \{c, s\} \\ \{\}, \{\sim n\}^- \xleftarrow{a_{14}} \{p, \sim n\}, not\ \{c, \sim s\} \\ \{n, \sim q, e\}, \{o, p\}^- \xleftarrow{a_{15}} \{o, p\}, not\ \{n\} \end{array} \right\}$$

En lo que resta del ejemplo, se utilizará *especificidad generalizada*(Definición 2.30) como criterio de comparación entre argumentos.

Sea $\Psi_{3.7} = \{\sim n, o, p, q\}$ un estado. La figura 3.1 muestra todos los argumentos ($\mathcal{A}_1$ a $\mathcal{A}_7$) que se pueden construir a partir de $(\Psi_{3.7}, \Delta_{3.7})$. Cabe recordar que un argumento convencionalmente es representado gráficamente con un triángulo etiquetado en su interior con el nombre del argumento y con su conclusión en el vértice superior del triángulo. Además una flecha con una sola punta indica derrota propia y una flecha de doble punta indica derrota por bloqueo. Como puede observarse en la figura 3.1, $\mathcal{A}_1$ no tiene derrotadores, $\mathcal{A}_3$ es un derrotador de bloqueo para $\mathcal{A}_2$ y viceversa; como $\mathcal{A}_2$ es un subargumento de $\mathcal{A}_5$, entonces $\mathcal{A}_3$ es un derrotador de bloqueo para $\mathcal{A}_5$, el cual a su vez es un derrotador propio de $\mathcal{A}_4$; y finalmente $\mathcal{A}_6$ y $\mathcal{A}_7$ son derrotadores de bloqueo uno del otro.

A continuación se analizarán qué acciones presentes en $\Gamma_{3.7}$ son aplicables a partir de $\Psi_{3.7}$:

1. $\{d\}, \{\}^- \xleftarrow{a_1} \{\}, not\ \{\}$ es aplicable en $\Psi_{3.7}$ porque no tiene precondiciones ni restricciones. Note que esta acción será aplicable en cualquier estado dado que no define ninguna condición que deba cumplirse para poder ser ejecutada.

$\mathcal{A}_1$=$\langle\{r \prec p, q\}, r\rangle$
$\mathcal{A}_2$=$\langle\{s \prec p\}, s\rangle$
$\mathcal{A}_3$=$\langle\{\sim s \prec q\}, \sim s\rangle$
$\mathcal{A}_4$= $\langle\{t \prec o\}, t\rangle$
$\mathcal{A}_5$=$\langle\{(\sim t \prec s)(s \prec p)\}, \sim t\rangle$
$\mathcal{A}_6$=$\langle\{\sim u \prec p, q\}, \sim u\rangle$
$\mathcal{A}_7$=$\langle\{u \prec q\}, u\rangle$

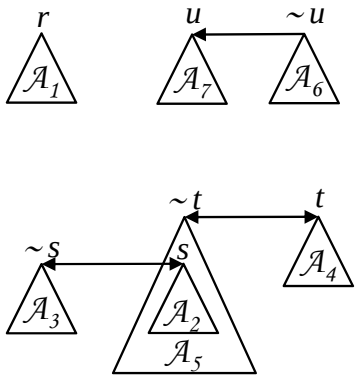

Figura 3.1: Relación de derrota entre los argumentos construidos a partir de $(\Psi_{3.7}, \Delta_{3.7})$

2. $\{e\}, \{\}^- \xleftarrow{a_2} \{p\}, not\ \{\}$ es aplicable en $\Psi_{3.7}$ porque su única precondición p está garantizada a partir de $(\Psi_{3.7}, \Delta_{3.7})$, dado que $p \in \Psi_{3.7}$ (ver proposición 2.5).

3. $\{d\}, \{\sim n\}^- \xleftarrow{a_3} \{\sim n\}, not\ \{\}$ es aplicable en $\Psi_{3.7}$ porque su única precondición $\sim n$ está garantizada a partir de $(\Psi_{3.7}, \Delta_{3.7})$, dado que $\sim n \in \Psi_{3.7}$ (ver proposición 2.5).

4. $\{\sim e\}, \{\}^- \xleftarrow{a_4} \{r\}, not\ \{\}$ es aplicable en $\Psi_{3.7}$ pues la única precondición r de a_2 está garantizada a partir de $(\Psi_{3.7}, \Delta_{3.7})$. En este caso existe un argumento $\mathcal{A}_1$=$\langle\{r \prec p, q\}, r\rangle$ no derrotado construido a partir de $(\Psi_{3.7}, \Delta_{3.7})$.

5. $\{d\}, \{\}^- \xleftarrow{a_5} \{s\}, not\ \{\}$ no es aplicable en $(\Psi_{3.7}, \Delta_{3.7})$ porque la precondición s no está garantizada a partir de $(\Psi_{3.7}, \Delta_{3.7})$. $\mathcal{A}_2$=$\langle\{s \prec p\}, s\rangle$ es el único argumento que se puede construir para sustentar s y está derrotado por $\mathcal{A}_3$=$\langle\{\sim s \prec q\}, \sim s\rangle$.

6. $\{d\}, \{\}^- \xleftarrow{a_6} \{\sim s\}, not\ \{\}$ no es aplicable en $(\Psi_{3.7}, \Delta_{3.7})$ porque la precondición $\sim s$ no está garantizada a partir de $(\Psi_{3.7}, \Delta_{3.7})$. $\mathcal{A}_3$=$\langle\{\sim s \prec q\}, \sim s\rangle$ es el único argumento que se puede construir para sustentar $\sim s$ y está derrotado por $\mathcal{A}_2$=$\langle\{s \prec p\}, s\rangle$.

   Note que a partir de $(\Psi_{3.7}, \Delta_{3.7})$ no es posible garantizar s (ver 5) ni tampoco $\sim s$. Esto ocurre porque $\mathcal{A}_2$=$\langle\{s \prec p\}, s\rangle$ es un *derrotador de bloqueo* (ver definición 2.29) de $\mathcal{A}_3$=$\langle\{\sim s \prec q\}, \sim s\rangle$ y viceversa.

7. $\{d\}, \{\}^- \xleftarrow{a_7} \{t\}, not\ \{\}$ es aplicable porque la precondición t está garantizada a partir de $(\Psi_{3.7}, \Delta_{3.7})$ dado que existe un argumento no derrotado $\mathcal{A}_4$= $\langle\{t \prec o\}, t\rangle$. Si bien existe un derrotador $\mathcal{A}_5$=$\langle\{(\sim t \prec s), (s \prec p)\}, \sim t\rangle$ para $\langle\{t \prec o\}, t\rangle$, este último es derrotado a su vez por $\mathcal{A}_3$=$\langle\{\sim s \prec q\}, \sim s\rangle$.

8. $\{d\},\{\}^- \xleftarrow{a_8} \{\}, not\ \{c\}$ es aplicable porque la restricción c no está garantizada a partir de $(\Psi_{3.7}, \Delta_{3.7})$. No es posible construir ningún argumento a partir de $(\Psi_{3.7}, \Delta_{3.7})$ para sustentar c.

9. $\{d\},\{\}^- \xleftarrow{a_9} \{\}, not\ \{s\}$ es aplicable porque s no está garantizado a partir de $(\Psi_{3.7}, \Delta_{3.7})$ como fue explicado en 5. Comparando con 6 se puede ver claramente que $\sim s$ no es equivalente a *not* s (Sin embargo, si $\sim s$ está garantizado entonces *not* S está satisfecho).

10. $\{d\},\{\}^- \xleftarrow{a_{10}} \{\}, not\ \{\sim s\}$ es aplicable porque $\sim s$ no está garantizado a partir de $(\Psi_{3.7}, \Delta_{3.7})$ como fue explicado en 6.

11. $\{d\},\{\}^- \xleftarrow{a_{11}} \{\sim u\}, not\ \{\}$ es aplicable porque $\sim u$ está garantizado a partir de $(\Psi_{3.7}, \Delta_{3.7})$ dado que existe el argumento no derrotado $\mathcal{A}_6{=}\langle\{\sim u \prec p, q\}, \sim u\rangle$ para sustentar u. Note que, si bien existe el contra-argumento $\mathcal{A}_7{=}\langle\{u \prec q\}, u\rangle$, éste no constituye un derrotador para $\mathcal{A}_6{=}\langle\{\sim u \prec p, q\}, \sim u\rangle$ porque (según el criterio de especificidad generalizada) $\mathcal{A}_6 \succ \mathcal{A}_7$.

12. $\{d\},\{\}^- \xleftarrow{a_{12}} \{\}, not\ \{u\}$ es aplicable porque la restricción u no está garantizada a partir de $(\Psi_{3.7}, \Delta_{3.7})$. Si bien existe el argumento $\mathcal{A}_7{=}\langle\{u \prec q\}, u\rangle$ para sustentar u, este es derrotado por $\mathcal{A}_6{=}\langle\{\sim u \prec p, q\}, \sim u\rangle$.

13. $\{\},\{p\}^- \xleftarrow{a_{13}} \{p, r\}, not\ \{c, s\}$ es aplicable dado que todas las precondiciones (p y r) están garantizadas (ver items 2 y 4) y todas las restricciones (c y s) *no* están garantizadas (ver items 8 y 5) a partir de $(\Psi_{3.7}, \Delta_{3.7})$.

14. $\{\},\{\sim n\}^- \xleftarrow{a_{14}} \{p, \sim n\}, not\ \{c, \sim s\}$ es aplicable porque las precondiciones p y $\sim n$ (ver items 2 y 3) están garantizadas y las restricciones c y $\sim s$ no están garantizadas (ver items 8 y 6)a partir de $(\Psi_{3.7}, \Delta_{3.7})$

15. $\{n, \sim q, e\},\{o, p\}^- \xleftarrow{a_{15}} \{o, p\}, not\ \{n\}$ es aplicable a partir de $(\Psi_{3.7}, \Delta_{3.7})$. Las precondiciones o y p están garantizadas dado que $o, p \in \Psi_{3.7}$. La restricción n no está garantizada a partir de $(\Psi_{3.7}, \Delta_{3.7})$, dado que $\sim n$ está garantizado porque $\sim n \in \Psi_{3.7}$.

3.3. Ejecución de acciones y secuencias de acciones

Es claro que sólo las acciones aplicables pueden ser ejecutadas en un determinado estado del mundo Ψ. La ejecución de una acción afectará directamente el mundo y modificará el conjunto Ψ. El resultado de ejecutar una acción aplicable $Ac{=}\langle \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C}\rangle$ se determina por medio de los conjuntos A y D. El conjunto D menciona los literales que se *quitarán* del conjunto Ψ mientras que A menciona los literales que se *agregarán* a Ψ.

A continuación se definirá formalmente el resultado de ejecutar una acción en un estado, pero antes se incluyen dos definiciones que muestran como se obtiene el complemento de literales y efectos.

Definición 3.6 [Complemento lógico de un literal]
Sea L un literal y A un átomo, el complemento lógico de L con respecto a la negación fuerte, denotado $\overline{L}$, se define de la siguiente manera:
si $L = A$, entonces $\overline{L} = {\sim}A$,
si $L = {\sim}A$, entonces $\overline{L} = A$.
Sea $S = \{L_1, \ldots, L_n\}$ un conjunto de literales: $\overline{S} = \{\overline{L_1}, \ldots, \overline{L_n}\}$.

Definición 3.7 [Complemento de un efecto]
Sea $\langle \mathsf{N}, \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C}\rangle$ una acción, $E \in \mathsf{A} \cup \mathsf{D}$ un efecto de Ac y L un literal. El complemento de E con respecto al operador "$-$", denotado E^{-}, se define de la siguiente manera:
si $E = L$, entonces $E^{-} = -L$,
si $E = -L$, entonces $E^{-} = L$.
Sea $S = \{E_1, \ldots, E_n\}$ un conjunto de efectos: $S^{-} = \{E_1, \ldots, E_n\}^{-} = \{{E_1}^{-}, \ldots, {E_n}^{-}\}$.

Ejemplo 3.8 Sea $S = \{a, b, {\sim}c, d\}$ un conjunto de literales y $E = \{-p, {\sim}q, -{\sim}r, t\}$ un conjunto de efectos. Luego, $\overline{S} = \{{\sim}a, {\sim}b, c, {\sim}d\}$, $S^{-} = \{-{\sim}a, -{\sim}b, -c, -{\sim}d\}$ y $E^{-} = \{p, -{\sim}q, {\sim}r, -t\}$.

Definición 3.8 [Resultado de ejecutar una acción]
Sea (Δ,Γ) un dominio de planificación, Ψ un estado y $\langle \mathsf{N}, \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C}\rangle \in \Gamma$ una acción *aplicable en* Ψ(Definición 3.5). El resultado de ejecutar N en Ψ, denotado $\Psi^{\overleftarrow{\mathsf{N}}}$, se define de la siguiente manera:

$$\Psi^{\overleftarrow{\mathsf{N}}} = ((\Psi \setminus \mathsf{D}^{-}) \setminus \overline{\mathsf{A}}) \cup \mathsf{A}$$

Esto es, se eliminan de Ψ los elementos de D^- y los elementos de $\overline{\mathsf{A}}$, y luego se agregan a Ψ los elementos de A.

Ejemplo 3.9 Considere el dominio de planificación $(\Delta_{3.2}, \Gamma_{3.3})$ del agente encargado de la limpieza definido en el ejemplo 3.4. Como vimos en el ejemplo 3.6 la acción $\{oficina\_limpia\}, \{\}^- \xleftarrow{limpiar} \{luz\}, not\ \{\}$ es aplicable en el estado:

$$\begin{aligned}\Psi_{3.1} = \quad & \{\sim pers\_alta, int(i_1), enc(i_1), lamp(l_1), con(i_1, l_1),\\ & int(i_2), \sim enc(i_2), lamp(l_2), con(i_2, l_2)\}\end{aligned}$$

Luego, el resultado de ejecutar la acción *limpiar* en $\Psi_{3.1}$ es el estado:

$$\begin{aligned}\Psi_{3.1}^{\xleftarrow{limpiar}} = \quad & \{\sim pers\_alta, int(i_1), enc(i_1), lamp(l_1), con(i_1, l_1),\\ & int(i_2), \sim enc(i_2), lamp(l_2), con(i_2, l_2), oficina\_limpia\}\end{aligned}$$

La siguiente proposición muestra que el resultado de aplicar una acción a un estado es una *revisión priorizada* con respecto a los literales presentes en los efectos positivos, ya que estos formarán parte del estado resultante y sus complementos serán removidos. Análogamente, los literales presentes en los efectos negativos son removidos del estado resultante.

Proposición 3.1 Sea $\langle \mathsf{N}, \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C} \rangle$ una acción aplicable en un estado Ψ y L un literal.

- Si $L \in \mathsf{A}$ entonces $L \in \Psi^{\xleftarrow{\mathsf{N}}}$ y $\overline{L} \notin \Psi^{\xleftarrow{\mathsf{N}}}$
- Si $-L \in \mathsf{D}$ entonces $L \notin \Psi^{\xleftarrow{\mathsf{N}}}$

Demostración: Directa por definición 3.8.

Observación 3.2 Es importante notar la diferencia que existe entre un literal l presente en los efectos negativos y un literal $\sim l$ presente en los efectos positivos. La ejecución de una acción de la forma $\{\}, \{l\}^- \longleftarrow \{\}, not\ \{\}$ quitará el literal l del estado, mientras que la ejecución de una acción de la forma $\{\sim l\}, \{\}^- \longleftarrow \{\}, not\ \{\}$, además de quitar el literal l, agregará el literal $\sim l$ al estado resultante. Esta diferencia tiene un efecto considerable sobre lo que puede garantizarse en el estado resultante. Como muestra el siguiente ejemplo, sólo quitar el literal l del estado no impedirá que l esté garantizado en el estado resultante (porque podría inferirse usando reglas rebatibles).

Ejemplo 3.10 Sea $\Psi = \{p, l\}$ un estado y (Δ, Γ) un dominio de planificación donde $\Delta = \{l \prec p\}$ y, $\Gamma = \{\{\}, \{l\}^- \xleftarrow{a} \{\}, not\ \{\}\}$. La acción a es aplicable en Ψ, dado que no tiene precondiciones ni restricciones, y su ejecución en Ψ quitará el literal l de Ψ obteniendo el estado $\Psi^{\xleftarrow{a}}$=$\{$p$\}$. El literal l está garantizado a partir de $(\Psi^{\xleftarrow{a}}, \Delta)$ porque existe el argumento no derrotado $\langle l \prec p, l\rangle$.

Sin embargo, cuando la ejecución de una acción agrega $\sim l$, esto impedirá que l esté garantizado en el estado resultante. La siguiente proposición formaliza lo anterior.

Proposición 3.2 Sea (Δ,Γ) un dominio de planificación, Ψ un estado y $\langle \mathsf{N}, \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C}\rangle$ una acción aplicable en Ψ. Si $L \in \mathsf{A}$, entonces $\overline{L}$ no está garantizado a partir de ($\Psi^{\xleftarrow{\mathsf{N}}}$, Δ).

Demostración: Si $L \in A$ entonces $L \in \Psi^{\xleftarrow{\mathsf{N}}}$ por la proposición 3.1. Luego no es posible construir un argumento para $\overline{L}$ a partir de ($\Psi^{\xleftarrow{\mathsf{N}}}$, Δ) por definición de argumento (ver definición 2.24, condición 2) y por lo tanto $\overline{L}$ no está garantizado a partir de ($\Psi^{\xleftarrow{\mathsf{N}}}$, Δ).

La siguiente proposición muestra que la ejecución de una acción aplicable en un estado, produce un conjunto consistente de hechos (esto es, un estado).

Proposición 3.3 [resultado consistente al ejecutar una acción] Sea Ψ un estado y $\langle \mathsf{N}, \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C}\rangle \in \Gamma$ una acción aplicable en Ψ, entonces $\Psi^{\xleftarrow{\mathsf{N}}}$ es *consistente*.
Demostración: Sea $S = ((\Psi \setminus \mathsf{D}^-) \setminus \overline{\mathsf{A}})$, luego $\Psi^{\xleftarrow{\mathsf{N}}} = S \cup \mathsf{A}$. Analicemos $\Psi^{\xleftarrow{\mathsf{N}}}$ por partes:

1. El estado Ψ es consistente (definición 3.1), luego $S = ((\Psi \setminus \mathsf{D}^-) \setminus \overline{\mathsf{A}})$ es consistente ya que no se agregan nuevos literales a Ψ (esto es, $S \subseteq \Psi$).

2. Para que $\Psi^{\xleftarrow{\mathsf{N}}} = S \cup \mathsf{A}$ sea consistente no deben existir dos literales l y $\sim l$ tal que $\{l, \sim l\} \subseteq S \cup \mathsf{A}$. Analicemos todos los casos:

   - $\{l, \sim l\} \not\subseteq S$ porque S es consistente (por 1).
   - $\{l, \sim l\} \not\subseteq \mathsf{A}$ porque A es consistente (definición 3.3)
   - Si $\sim l \in \mathsf{A}$ entonces $l \in \overline{\mathsf{A}}$ luego $l \notin S$ ya que $S = ((\Psi \setminus \mathsf{D}^-) \setminus \overline{\mathsf{A}})$.
   - Si $l \in \mathsf{A}$ entonces $\sim l \in \overline{\mathsf{A}}$ luego $\sim l \notin S$ ya que $S = ((\Psi \setminus \mathsf{D}^-) \setminus \overline{\mathsf{A}})$.

   Luego no existen dos literales l y $\sim l$ tal que $\{l, \sim l\} \subseteq \Psi^{\xleftarrow{\mathsf{N}}}$ por lo tanto $\Psi^{\xleftarrow{\mathsf{N}}}$ es consistente.

Para ilustrar mas casos de acciones aplicables, a continuación se incluye un ejemplo de acciones aplicables sobre un dominio de planificación proposicional.

Ejemplo 3.11 Consideraremos el dominio de planificación proposicional $(\Delta_{3.7}, \Gamma_{3.7})$ y el estado $\Psi_{3.7} = \{\sim n, o, p, q\}$ definido en el ejemplo 3.7. La figura 3.2 muestra una tabla con el resultado de ejecutar algunas acciones de $\Gamma_{3.7}$ que son aplicables en $\Psi_{3.7}$:

$\langle \mathsf{N}, \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C} \rangle$	$\Psi_{3.7}^{\overleftarrow{\mathsf{N}}} = ((\Psi_{3.7} \setminus \mathsf{D}^-) \setminus \overline{\mathsf{A}}) \cup \mathsf{A}$
$\{e\}, \{\}^- \overleftarrow{\ a_2\ } \{p\}, not\ \{\}$	$\Psi_{3.7}^{\overleftarrow{a_2}} = ((\{\sim n, o, p, q\} \setminus \emptyset) \setminus \{\sim e\}) \cup \{e\}$ $= \{\sim n, o, p, q, e\}$
$\{\sim e\}, \{\}^- \overleftarrow{\ a_4\ } \{p\}, not\ \{\}$	$\Psi_{3.7}^{\overleftarrow{a_4}} = ((\{\sim n, o, p, q\} \setminus \emptyset) \setminus \{e\}) \cup \{\sim e\}$ $= \{\sim n, o, p, q, \sim e\}$
$\{n\}, \{\}^- \overleftarrow{\ a_7\ } \{t\}, not\ \{\}$	$\Psi_{3.7}^{\overleftarrow{a_7}} = ((\{\sim n, o, p, q\} \setminus \emptyset) \setminus \{\sim n\}) \cup \{n\}$ $= \{o, p, q, n\}$
$\{\sim p\}, \{\}^- \overleftarrow{\ a_8\ } \{\}, not\ \{c\}$	$\Psi_{3.7}^{\overleftarrow{a_8}} = ((\{\sim n, o, p, q\} \setminus \emptyset) \setminus \{p\}) \cup \{\sim p\}$ $= \{\sim n, o, q, \sim p\}$
$\{\}, \{p\}^- \overleftarrow{\ a_{13}\ } \{p, r\}, not\ \{c, s\}$	$\Psi_{3.7}^{\overleftarrow{a_{13}}} = ((\{\sim n, o, p, q\} \setminus \{p\}) \setminus \emptyset) \cup \emptyset$ $= \{\sim n, o, q\}$
$\{\}, \{\sim n\}^- \overleftarrow{\ a_{14}\ } \{p, \sim n\}, not\ \{c, \sim s\}$	$\Psi_{3.7}^{\overleftarrow{a_{14}}} = ((\{\sim n, o, p, q\} \setminus \{\sim n\}) \setminus \emptyset) \cup \emptyset$ $= \{o, p, q\}$
$\{n, \sim q, e\}, \{o, p\}^- \overleftarrow{\ a_{15}\ } \{o, p\}, not\ \{n\}$	$\Psi_{3.7}^{\overleftarrow{a_{15}}} = ((\{\sim n, o, p, q\} \setminus \{o, p\}) \setminus$ $\{\sim n, q, \sim e\}) \cup \{n, \sim q, e\}$ $= \{n, \sim q, e\}$

Figura 3.2: Resultado de ejecutar acciones aplicables en $\Psi_{3.7}$

Dado que la ejecución de una acción aplicable produce un estado (proposición 3.3), otra acción podría aplicarse sobre éste produciendo un nuevo estado sobre el cual se podrían aplicar otras acciones, y así sucesivamente. Esto produce una secuencia aplicable de acciones que se define como sigue.

Definición 3.9 [Secuencia aplicable de Acciones/Plan]
Sea (Δ,Γ) un dominio de planificación y Ψ un estado. Una secuencia de acciones S es *aplicable* a partir de Ψ si:

- $S = [\,]$ (es una secuencia vacía) ó
- $S = [a_1, a_2, \ldots, a_n]$, la acción a_1 es aplicable en Ψ y la subsecuencia $[a_2, \ldots, a_n]$ es *a*plicable a partir de $\Psi^{\xleftarrow{a_1}}$

Una secuencia aplicable de acciones también se denominará *Plan*.

Proposición 3.4 [subsecuencia aplicable] Sea (Δ,Γ) un dominio de planificación y Ψ un estado. Si $[a_1, a_2, \ldots, a_n]$ es una secuencia de acciones *aplicable* a partir de Ψ, cualquier *subsecuencia* $[a_1, a_2, \ldots, a_k]$ $(k \leq n)$ es *aplicable* a partir de Ψ.
Demostración: Directo por definición 3.9.

Al igual que una acción aplicable, una secuencia aplicable de acciones puede ejecutarse en un estado y producirá un cambio en el mismo. A continuación se define el resultado de ejecutar una secuencia aplicable de acciones.

Definición 3.10 [Resultado de ejecutar un Plan]
Sea (Δ,Γ) un dominio de planificación y Ψ un estado. Sea $S = [a_1, a_2, \ldots, a_n]$ una secuencia de acciones aplicable a partir de Ψ. El resultado de ejecutar $S = [a_1, a_2, \ldots, a_n]$ en Ψ, notado como $\Psi^{\xleftarrow[*]{S}}$ ó $\Psi^{\xleftarrow{[a_1, a_2, \ldots, a_n]}}$, se define como:

$$\Psi^{\xleftarrow{[a_1, a_2, \ldots, a_n]}} = \left\{ \begin{array}{l} \Psi, \text{ si n=0.} \\ (\Psi^{\xleftarrow{[a_1, a_2, \ldots, a_{n-1}]}})^{\xleftarrow{a_n}}, \text{ si } n > 0 \end{array} \right\}$$

Como muestra la siguiente proposición la ejecución de una secuencia de aciones aplicables produce un conjunto consistente de hechos (esto es, un estado).

Proposición 3.5 [resultado consistente al ejecutar un plan] Sea (Δ,Γ) un dominio de planificación y Ψ un estado. Si $[a_1, a_2, \ldots, a_n]$ es una secuencia de acciones aplicable a partir de Ψ, entonces $\Psi^{\xleftarrow{[a_1, a_2, \ldots, a_n]}}$ es consistente.
Demostración: Inducción sobre la cantidad de acciones de la secuencia.

- Caso base: Sea $[a_1]$ una secuencia de una sola acción. Luego $\Psi^{\xleftarrow{[a_1]}} = \Psi^{\xleftarrow{a_1}}$ es consistente. por la proposición 3.3

- hipótesis inductiva: Sea $[a_1, a_2, \ldots, a_k]$ una secuencia de k acciones. Supongamos que $\Psi^{\xleftarrow{[a_1, a_2, \ldots, a_k]}}$ es consistente.
- Paso inductivo: Sea $[a_1, a_2, \ldots, a_k, a_{k+1}]$ una secuencia de $k+1$ acciones. $\Psi^{\xleftarrow{[a_1, a_2, \ldots, a_k, a_{k+1}]}} = (\Psi^{\xleftarrow{[a_1, a_2, \ldots, a_k]}})^{\xleftarrow{a_{k+1}}}$ es consistente por la proposición 3.3

Corolario 3.1 (estados intermedios consistentes) Si $[a_1, a_2, \ldots, a_n]$ es una secuencia de acciones $a$plicable a partir de Ψ, entonces $\Psi^{\xleftarrow{[a_1, a_2, \ldots, a_k]}}$ es consistente para todo $k, 1 <= k <= n$.

En el siguiente ejemplo se analizarán varias secuencias de acciones para determinar si son aplicables a partir de un estado y mostrar el resultado de su ejecución.

Ejemplo 3.12 Considere el dominio de planificación $(\Delta_{3.2}, \Gamma_{3.3})$ del agente encargado de la limpieza definido en el ejemplo 3.4, donde:

$$\Delta_{3.2} = \left\{ \begin{array}{c} luz \prec luz\_natural, \\ luz \prec luz\_artificial, \\ luz\_natural \prec pers\_alta, es\_dia, \\ luz\_artificial \prec enc(L), lamp(L), \\ peligroso(I) \prec int(I), mojado(I), \\ peligroso(I) \prec int(I), roto(I), \\ \sim peligroso(I) \prec int(I), \sim electricidad, \\ enc(L) \prec con(I, L), enc(I), \\ \sim enc(L) \prec lamp(L), \sim electricidad, \\ \sim enc(L) \prec lamp(L), rota(L), \\ rep(L) \prec lamp(L), not\ en\_uso(L), \\ en\_uso(L) \prec con(I, L) \\ \sim rep(L) \prec lamp(L), rota(L), \end{array} \right\}$$

$$\Gamma_{3.3} = \left\{ \begin{array}{c} \{pers\_alta\}, \{\}^- \xleftarrow{subir\_pers} \{\sim pers\_alta\}, not\ \{\} \\ \{\sim pers\_alta\}, \{\}^- \xleftarrow{bajar\_pers} \{pers\_alta\}, not\ \{\} \\ \{enc(I)\}, \{\}^- \xleftarrow{encender(I)} \{\sim enc(I), int(I)\}, not\ \{peligroso(I)\} \\ \{\sim enc(I)\}, \{\}^- \xleftarrow{apagar(I)} \{enc(I), int(I)\}, not\ \{peligroso(I)\} \\ \{con(I, R)\}, \{con(I, L)\}^- \xleftarrow{cambiar(L, I, R)} \{con(I, L), rep(R), \sim enc(I)\}, not\ \{\} \\ \{oficina\_limpia\}, \{\}^- \xleftarrow{limpiar} \{luz\}, not\ \{\} \end{array} \right\}$$

Sea $\Psi_{3.12}$ un estado del mundo donde es de día, la persiana se encuentra baja, los interruptores i_1 e i_2 están apagados y hay 3 lámparas: l_1 y l_2 están rotas y conectadas a los interruptores i_1 e i_2 respectivamente, y l_3 no esta conectada a ningún interruptor. El estado $\Psi_{3.12}$ está definido por:

$$\begin{aligned}\Psi_{3.12}= \quad & \{es\_dia, \sim pers\_alta, int(i_1), \sim enc(i_1), lamp(l_1), rota(l_1), con(i_1, l_1),\\ & int(i_2), \sim enc(i_2), lamp(l_2), rota(l_2), con(i_2, l_2), lamp(l_3)\}\end{aligned}$$

A continuación se analizará para distintas secuencias de acciones si son aplicables a partir de $\Psi_{3.12}$.

1. Sea $S_1 = [encender(i_1), limpiar, apagar(i_1)]$. La primera acción, $encender(i_1)$, es aplicable en $\Psi_{3.12}$ dado que el interruptor i_1 se encuentra apagado y no resulta peligroso en el estado $\Psi_{3.12}$. Note que las precondiciones $\sim enc(i_1)$ y $int(i_1)$ pertenecen a $\Psi_{3.12}$ y por lo tanto están garantizadas a partir de $(\Psi_{3.12}, \Delta_{3.2})$. Por otra parte la restricción $peligroso(i_1)$ no está garantizada dado que no es posible construir un argumento para $peligroso(i_1)$ a partir de $(\Psi_{3.12}, \Delta_{3.2})$. El resultado de ejecutar $encender(i_1)$ en el estado $\Psi_{3.12}$ es:

$$\begin{aligned}\Psi_{3.12}^{\overset{encender(i_1)}{\longleftarrow}} = \quad & \{es\_dia, \sim pers\_alta, int(i_1), enc(i_1), lamp(l_1), rota(l_1), con(i_1, l_1),\\ & int(i_2), \sim enc(i_2), lamp(l_2), rota(l_2), con(i_2, l_2), lamp(l_3)\end{aligned}$$

   La siguiente acción *limpiar* no es aplicable en $\Psi_{3.12}^{\overset{encender(i_1)}{\longleftarrow}}$ dado que no hay luz en la oficina porque la lámpara l_1 está rota. Note que, si bien se puede construir a partir de $(\Psi_{3.12}^{\overset{encender(i_1)}{\longleftarrow}}, \Delta_{3.2})$ el argumento

$$\begin{aligned}\langle\{luz -\!\prec luz\_artificial), (luz\_artificial -\!\prec enc(l_1), lamp(l_1)),\\ (enc(l_1) -\!\prec con(i_1, l_1), enc(i_1)\}, luz\rangle\end{aligned}$$

   para sustentar luz, este está derrotado por $\langle\{\sim enc(l_1) -\!\prec lamp(l_1), rota(l_1))\}, \sim enc\rangle$ Luego la secuencia $S_1 = encender(i_1), limpiar, apagar(i_1)]$ no es aplicable a partir de $\Psi_{3.12}$.

2. Consideremos la secuencia $S_2 = [encender(i_1), cambiar(l_1, i_1, l_3), limpiar, apagar(i_1)]$. Como se explicó para la secuencia anterior la acción $encender(i_1)$ es aplicable en

$\Psi_{3.12}$. La siguiente acción $cambiar(l_1, i_1, l_3)$ no es aplicable en $\Psi_{3.12}^{\overleftarrow{encender(i_1)}}$ porque el interruptor i_1 se encuentra encendido. Note que la precondición $\sim enc(i_1)$ no está garantizada a partir de $(\Psi_{3.12}^{\overleftarrow{encender(i_1)}}, \Delta_{3.2})$ dado que $enc(i) \in \Psi_{3.12}^{\overleftarrow{encender(i_1)}}$. Por lo tanto la secuencia $S_2 = [encender(i_1), cambiar(l_1, i_1, l_3), limpiar, apagar(i_1)]$ no es aplicable en $\Psi_{3.12}$.

3. Sea $S_3 = [cambiar(l_1, i_1, l_3), encender(i_1), limpiar, apagar(i_1)]$. La acción $cambiar(l_1, i_1, l_3)$ es aplicable en $\Psi_{3.12}$ dado que el interruptor i_1 está conectado a la lámpara l_1, i_1 está apagado y la lámpara l_3 puede ser usada como repuesto. Note que las precondiciones $con(l_1, i_1)$ y $\sim enc(i_1)$ están garantizada porque ambas pertenecen a $\Psi_{3.12}$. La precondición $rep(l_3)$ está garantizada dado que a partir de $(\Psi_{3.12}, \Delta_{3.2})$ se puede construir el argumento no derrotado $\langle \{rep(l_3) -\prec lamp(l_3), not\ en\_uso(l_3)\}, rep(l_3)\rangle$. El resultado de ejecutar $cambiar(l_1, i_1, l_3)$ en el estado $\Psi_{3.12}$ es:

$$\begin{aligned}\Psi_{3.12}^{\overleftarrow{cambiar(l_1,i_1,l_3)}} = \quad & \{es\_dia, \sim pers\_alta, int(i_1), \sim enc(i_1), lamp(l_1),\\ & rota(l_1), con(i_1, l_3), int(i_2), \sim enc(i_2),\\ & lamp(l_2), rota(l_2), con(i_2, l_2), lamp(l_3)\}\end{aligned}$$

Cabe destacar que luego de ejecutar $cambiar(l_1, i_1, l_3)$ el literal $rep(l_3)$ ya no estará garantizado. Note que en el estado $\Psi_{3.12}^{\overleftarrow{cambiar(l_1,i_1,l_3)}}$ el argumento $\langle \{rep(l_3) -\prec lamp(l_3), not\ en\_uso(l_3)\}, rep(l_3)\rangle$ estará derrotado en la suposición $not\ en\_uso(l_3)$, ya que al agregarse el literal $con(i_1, l_3)$ a $\Psi_{3.12}^{\overleftarrow{cambiar(l_1,i_1,l_3)}}$ es posible construir el argumento no derrotado $\langle \{en\_uso(l_3) -\prec con(i_1, l_3)\}, en\_uso(l_3)\rangle$ a partir de $(\Psi_{3.12}^{\overleftarrow{cambiar(l_1,i_1,l_3)}}, \Delta_{3.2})$.

La siguiente acción, $encender(i_1)$, es aplicable en $\Psi_{3.12}^{\overleftarrow{cambiar(l_1,i_1,l_3)}}$. Las precondiciones $\sim enc(i_1)$, $int(i_1) \in \Psi_{3.12}^{\overleftarrow{cambiar(l_1,i_1,l_3)}}$ y por lo tanto están garantizadas a partir de $(\Psi_{3.12}^{\overleftarrow{cambiar(l_1,i_1,l_3)}}, \Delta_{3.2})$. Por otra parte la restricción $peligroso(i_1)$ no está garantizada dado que no es posible construir un argumento para $peligroso(i_1)$ a partir de $\Psi_{3.12}^{\overleftarrow{cambiar(l_1,i_1,l_3)}}, \Delta_{3.2})$. El resultado de ejecutar $encender(i_1)$ en el estado $\Psi_{3.12}^{\overleftarrow{cambiar(l_1,i_1,l_3)}}$ es:

$$(\Psi_{3.12}^{\overleftarrow{cambiar(l_1,i_1,l_3)}})^{\overleftarrow{encender(i_1)}} = \{es\_dia, \sim pers\_alta, int(i_1), enc(i_1), lamp(l_1), rota(l_1), con(i_1, l_3), int(i_2), \sim enc(i_2), lamp(l_2), rota(l_2), con(i_2, l_2), lamp(l_3)\}$$

Veamos ahora si la siguiente acción, *limpiar*, es aplicable en $(\Psi_{3.12}^{\overleftarrow{cambiar(l_1,i_1,l_3)}})^{\overleftarrow{encender(i_1)}}$. La única precondición de *limpiar* es *luz* y está garantizada a partir de $((\Psi_{3.12}^{\overleftarrow{cambiar(l_1,i_1,l_3)}})^{\overleftarrow{encender(i_1)}}, \Delta_{3.2})$ dado que existe el argumento no derrotado

$$\langle \{luz \prec luz\_artificial), (luz\_artificial \prec enc(l_3), lamp(l_3)), (enc(l_3) \prec con(i_1, l_3), enc(i_1)\}, luz\rangle$$

Luego la acción *limpiar* es aplicable y el resultado de ejecutar *limpiar* en el estado $(\Psi_{3.12}^{\overleftarrow{cambiar(l_1,i_1,l_3)}})^{\overleftarrow{encender(i_1)}}$ es:

$$((\Psi_{3.12}^{\overleftarrow{cambiar(l_1,i_1,l_3)}})^{\overleftarrow{encender(i_1)}})^{\overleftarrow{limpiar}} = \{es\_dia, \sim pers\_alta, int(i_1), enc(i_1), lamp(l_1), rota(l_1), con(i_1, l_3), int(i_2), \sim enc(i_2), lamp(l_2), rota(l_2), con(i_2, l_2), lamp(l_3), oficina\_limpia\}$$

Por último, la acción $apagar(i_1)$ es aplicable en $((\Psi_{3.12}^{\overleftarrow{cambiar(l_1,i_1,l_3)}})^{\overleftarrow{encender(i_1)}})^{\overleftarrow{limpiar}}$ dado que el interruptor i_1 se encuentra encendido y no resulta peligroso. Note que las precondiciones $enc(i_1)$ y $int(i_1)$ están garantizadas porque pertenecen a $((\Psi_{3.12}^{\overleftarrow{cambiar(l_1,i_1,l_3)}})^{\overleftarrow{encender(i_1)}})^{\overleftarrow{limpiar}}$. Por otra parte, la restricción $peligroso(i_1)$ no está garantizada dado que no es posible construir un argumento para $peligroso(i_1)$ a partir de $(((\Psi_{3.12}^{\overleftarrow{cambiar(l_1,i_1,l_3)}})^{\overleftarrow{encender(i_1)}})^{\overleftarrow{limpiar}}, \Delta_{3.2})$. El resultado de ejecutar $apagar(i_1)$ en el estado $((\Psi_{3.12}^{\overleftarrow{cambiar(l_1,i_1,l_3)}})^{\overleftarrow{encender(i_1)}})^{\overleftarrow{limpiar}}$ es:

$$(((\Psi_{3.12}^{\overleftarrow{cambiar(l_1,i_1,l_3)}})^{\overleftarrow{encender(i_1)}})^{\overleftarrow{limpiar}})^{\overleftarrow{apagar(i_1)}} = \{es\_dia, \sim pers\_alta, int(i_1), \sim enc(i_1), lamp(l_1), rota(l_1), con(i_1, l_3), int(i_2), \sim enc(i_2), lamp(l_2), rota(l_2), con(i_2, l_2), lamp(l_3), oficina\_limpia\}$$

4. Consideremos la secuencia $S_4 = [subir\_pers, limpiar]$. La acción $subir\_pers$ es aplicable en $\Psi_{3.12}$ dado que la precondición $\sim pers\_alta \in \Psi_{3.12}$. El resultado de ejecutar $subir\_pers$ en Ψ es el estado:

$$\Psi_{3.12}^{\overleftarrow{subir\_pers}} = \{es\_dia, pers\_alta, int(i_1), \sim enc(i_1), lamp(l_1), rota(l_1), con(i_1, l_1), int(i_2), \sim enc(i_2), lamp(l_2), rota(l_2), con(i_2, l_2), lamp(l_3)\}$$

La acción *limpiar* también es aplicable en $\Psi_{3.12}^{\overleftarrow{subir\_pers}}$ por que hay luz en la oficina dado que es de día y la persiana está alta. Note que la precondición *luz* está garantizada a partir de $(\Psi_{3.12}^{\overleftarrow{subir\_pers}}, \Gamma_{3.3})$ porque existe el argumento no derrotado

$$\langle \{(luz -\!\prec luz\_natural), (luz\_natural -\!\prec pers\_alta, es\_dia)\}, luz \rangle$$

para sustentar *luz*. El resultado de ejecutar *limpiar* en $\Psi_{3.12}^{\overleftarrow{subir\_pers}}$ es:

$$(\Psi_{3.12}^{\overleftarrow{subir\_pers}})^{\overleftarrow{limpiar}} = \{es\_dia, pers\_alta, int(i_1), \sim enc(i_1), lamp(l_1), rota(l_1), con(i_1, l_1), int(i_2), \sim enc(i_2), lamp(l_2), rota(l_2), con(i_2, l_2), lamp(l_3), oficina\_limpia\}$$

3.4. Problemas de planificación

A continuación se define formalmente un problema de planificación para un dominio DAKAR.

Definición 3.11 [Problema de Planificación]
Sea (Δ,Γ) un dominio de planificación DAKAR. Un problema de planificación sobre (Δ,Γ) está definido por una quíntupla $(\Psi, \Delta, \Gamma, Meta, R)$, donde:

- Ψ es un conjunto consistente de literales que representan el *e*stado inicial.
- Δ es un conjunto de *reglas rebatibles extendidas* que el agente puede utilizar para razonar.
- Γ es un conjunto de *esquemas de acciones* que el agente puede realizar.
- $Meta$ es un conjunto consistente de literales que representan las *metas del agente*.

- R es un conjunto de restricciones de la forma *not* C, donde C es un literal.

Solucionar un problema de planificación consiste en encontrar una secuencia de acciones aplicable a partir del estado Ψ que logre satisfacer las condiciones impuestas por los conjuntos $Meta$ y R. Esto es, todos los literales presentes en $Meta$ deben estar garantizados y para cada restricción $notC$, C no debe estar garantizado. A continuación se define formalmente este concepto.

Definición 3.12 [Solución a un problema de Planificación]
Sea $P = (\Psi, \Delta, \Gamma, Meta, \mathsf{C})$ un problema de planificación. Una solución para P es una secuencia de acciones $[a_1, \ldots, a_n]$ ($a_i \in \Gamma$) que cumple con las siguientes condiciones:

- $[a_1, \ldots, a_n]$ es aplicable a partir de Ψ,
- cada literal en $Meta$ este garantizado a partir de $(\Psi^{\overleftarrow{[a_1, \ldots, a_n]}}, \Delta)$ y
- para cada restricción *not* C_i en R, C_i no se puede garantizar a partir de $(\Psi^{\overleftarrow{[a_1, \ldots, a_n]}}, \Delta)$.

Observe que no se requiere que las metas sean parte del estado resultante ($Meta \subseteq \Psi^{\overleftarrow{[a_1, \ldots, a_n]}}$) sino que estén garantizadas.

Ejemplo 3.13 Considere el dominio de planificación del agente encargado de la limpieza definido en el ejemplo 3.4. Supongamos que el objetivo del agente es dejar limpia la oficina y no dejar ninguna lámpara encendida. y que el estado del mundo es el definido en el ejemplo 3.12, donde: es de día, la persiana se encuentra baja, los interruptores i_1 e i_2 están apagados y hay 3 lámparas: l_1 y l_2 están rotas y conectadas a los interruptores i_1 e i_2 respectivamente, y l_3 no está conectada a ningún interruptor.

Este problema de planificación se define en DAKAR como $(\Psi_{3.12}, \Delta_{3.2}, \Gamma_{3.3}, Meta_{3.13}, \mathsf{C}_{3.13})$, donde:

- $(\Delta_{3.2}, \Gamma_{3.3})$ es el dominio de planificación definido en el ejemplo 3.4.
- $\Psi_{3.12}$ es el estado definido por:

$$\begin{aligned} \Psi_{3.12} \quad = \quad & \{es\_dia, \sim pers\_alta, int(i_1), \sim enc(i_1), lamp(l_1), rota(l_1), con(i_1, l_1), \\ & int(i_2), \sim enc(i_2), lamp(l_2), rota(l_2), con(i_2, l_2), lamp(l_3)\} \end{aligned}$$

- $Meta_{3.13} = \{oficina\_limpia\}$
- $\mathsf{C}_{3.13} = \{not\ luz\_artificial\}$

Un plan posible para solucionar este problema es la secuencia

$$S_3 = [cambiar(l_1, i_1, l_3), encender(i_1), limpiar, apagar(i_1)]$$

dado que:

- $S_3 = [cambiar(l_1, i_1, l_3), encender(i_1), limpiar, apagar(i_1)]$ es aplicable a partir de $\Psi_{3.12}$ (ver ejemplo 3.12).
- $oficina\_limpia$ está garantizado a partir de $(\overset{\overleftarrow{S_3}}{\Psi^*_{3.12}}, \Delta_{3.2})$ porque $oficina\_limpia \in \overset{\overleftarrow{S_3}}{\Psi^*_{3.12}}$
- $luz\_artificial$ no está garantizado a partir de $(\overset{\overleftarrow{S_3}}{\Psi^*_{3.12}}, \Delta_{3.2})$ dado que no es posible construir un argumento para sustentar $luz\_artificial$ porque ambos interruptores están apagados en $\overset{\overleftarrow{S_3}}{\Psi^*_{3.12}}$.

Otro plan posible es la secuencia $S_4 = [subir\_pers, limpiar]$ del ejemplo 3.12.

Por otra parte, la secuencia $[cambiar(l_1, i_1, l_3), encender(i_1), limpiar]$ es aplicable pero no soluciona el problema porque no satisface la restricción $not\ luz\_artificial$. Note que el interruptor i_1 quedará encendido y conectado a la lámpara sana l_3 y por lo tanto se puede garantizar $luz\_artificial$. La secuencia aplicable $[cambiar(l_1, i_1, l_3), encender(i_1), apagar(i_1)]$ tampoco es una solución porque no garantiza la meta $oficina\_limpia$

3.4.1. Eliminando restricciones de un problema de planificación

Si bien desde el punto de vista declarativo es importante contar con la posibilidad de definir restricciones en las acciones y en los problemas de planificación DAKAR, esta característica no agrega poder expresivo al formalismo. De hecho, cualquier problema de planificación P puede transformarse en un problema de planificación sin restricciones

P' que resulte equivalente, en el sentido que toda solución para P también es una solución para P' y viceversa. La ventaja de realizar esta transformación es que simplifica la construcción de métodos y algoritmos de planificación para dominios DAKAR.

A continuación se definirá una función que permite transformar un problema de planificación cualquiera en un problema de planificación equivalente que no contiene restricciones. La equivalencia entre ambos problemas es capturada por el teorema 3.1 definido mas adelante.

Definición 3.13 Sea un $P = (\Psi, \Delta, \Gamma, Meta, R)$ problema de planificación. La función $quitar\_restricciones(P)$ da como resultado un problema de planificación $(\Psi, \Delta', \Gamma', Meta', R')$, donde:

1. $\Delta' = \Delta \cup Reglas\_R \cup Reglas\_\Gamma$, donde:
   - *a*) Por cada restricción $not\ C \in R$, $Reglas\_R$ contiene una regla de la forma $L_C \prec not\ C$, donde:
     - I. L_C es un literal positivo *nuevo*
     - II. $L_C \neq C$ y
     - III. L_C contiene exactamente las mismas variables que C
   - *b*) Por cada restricción $not\ P \in \mathsf{C}$ de cada esquema de acción $\langle \mathsf{N}, \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C} \rangle \in \Gamma$, $Reglas\_\Gamma$ contiene una regla de la forma $L_P \prec not\ P$, donde:
     - I. L_P es un literal positivo *nuevo*,
     - II. $L_P \neq P$ y
     - III. L_P contiene exactamente las mismas variables que P
   - *c*) Si $L \prec not\ X \in Reglas\_R \cup Reglas\_\Gamma$, entonces el literal L no aparece en ninguna otra regla de Δ' y $L \notin \Psi$.
2. Por cada esquema de acción $\langle \mathsf{N}, \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C} \rangle \in \Gamma$, Γ' contiene un esquema de acción de la forma $\langle N', \mathsf{A}, \mathsf{D}, \mathsf{P}', \mathsf{C}' \rangle$, donde:
   - *a*) N' es un nombre *nuevo* que contiene las mismas variables que N,
   - *b*) $\mathsf{C}' = \emptyset$
   - *c*) $\mathsf{P}' = \mathsf{P} \cup \mathsf{P_C}$, con $\mathsf{P_C} = \{L_P \mid L_P \prec not\ P \in Reglas\_\Gamma, not\ P \in \mathsf{C}\}$

3. $Meta' = Meta \cup \{L_C \mid L_C \prec not\, C \in Reglas\_R\}$

4. $R' = \emptyset$

Para eliminar una restricción *not* P presente en una acción N, la función *quitar_restricciones* agrega la regla $L_P \prec not\, P$ a la base de conocimiento Δ (condición $1b$) e incorpora el literal L_P a las precondiciones de la acción (condición $2c$). Dado que L_P es un literal positivo nuevo (esto es, L_P no aparece en nigúna otra regla de Δ' ni en Ψ), la regla $L_P \prec not\, P$ asegura que L_P estará garantizado si, y solo si, P *no* está garantizado. De esta forma, se logra eliminar la restricción sin modificar las condiciones para que la acción sea ejecutable.

Algo similar ocurre para eliminar una restricción *not* C presente en las restricciones del problema de planificación. En este caso, la función *quitar_restricciones* agrega la regla $L_C \prec not\, C$ a la base de conocimiento Δ (condición $1a$) e incorpora el literal L_C a las metas del problema (condición 3).

Nota 3.3 Por la condición $2a$ de la definición 3.13 cada acción $\langle \mathsf{N}, \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C} \rangle \in \Gamma$ se corresponde con una acción $\langle N', \mathsf{A}, \mathsf{D}, \mathsf{P}', \mathsf{C}' \rangle \in \Gamma'$ donde N' es un nombre *nuevo*. Esta condición solo es necesaria para poder distinguir entre las acciones de Γ y Γ' y hacer mas clara la demostración de la equivalencia entre los problemas P y *quitar_restricciones*(P).

Para demostrar la equivalencia entre un problema de planificación P y el problema *quitar_restricciones*(P), primero se introducirán algunas propiedades intermedias. En primer lugar, se mostrará que si una secuencia de acciones $[a_1, \ldots, a_n]$ es aplicable en un estado Ψ y su transformación $[a'_1, \ldots, a'_n]$ también es aplicable en Ψ, entonces ambas secuencias producirán el mismo resultado al aplicarse en Ψ. Esto se debe a que la transformación realizada por *quitar_restricciones*(P) no modifica los efectos de las acciones (Definición 3.13, condición 2).

Lema 3.1 Sea $P = (\Psi, \Delta, \Gamma, Meta, R)$ un problema de planificación y *quitar_restricciones*$(P) = (\Psi, \Delta', \Gamma', Meta', \mathsf{C}')$. Sea $[a_1, \ldots, a_n]$ una secuencia de acciones presentes en Γ y sea $[a'_1, \ldots, a'_n]$ la secuencia de acciones correspondientes en Γ'.

Si $[a_1, \ldots, a_n]$ *y* $[a'_1, \ldots, a'_n]$ *son aplicables a partir de* Ψ *entonces*

$$\Psi^{\xleftarrow{[a'_1, \ldots, a'_n]}} = \Psi^{\xleftarrow{[a_1, \ldots, a_n]}}$$

Demostración: Inducción sobre cantidad de acciones de la secuencia.

- Caso base: Si $[a_1, \ldots, a_n] = [a'_1, \ldots, a'_n] = [\,]$ son secuencias vacías (de 0 acciones), entonces por la definición 3.10: $\Psi^{\underleftarrow{[a'_1, \ldots, a'_n]}} = \Psi^{\underleftarrow{[a_1, \ldots, a_n]}} = \Psi^{\underleftarrow{[\,]}} = \Psi$,

- hipótesis inductiva: Sean $[a_1, \ldots, a_k]$ y $[a'_1, \ldots, a'_k]$ dos secuencias de k acciones ($k > 0$), aplicables a partir de Ψ. Supongamos que $\Psi^{\underleftarrow{[a'_1, \ldots, a'_k]}} = \Psi^{\underleftarrow{[a_1, \ldots, a_k]}}$

- Paso inductivo: Sean $[a_1, \ldots, a_k, a_{k+1}]$ y $[a'_1, \ldots, a'_k, a'_{k+1}]$ dos secuencias de longitud $k+1$, aplicables a partir de Ψ, donde $\langle a'_{k+1}, \mathsf{A}, \mathsf{D}, \mathsf{P}', \mathsf{C}' \rangle \in \Gamma'$ y $\langle a_{k+1}, \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C} \rangle \in \Gamma$.

$$
\begin{aligned}
\Psi^{\underleftarrow{[a'_1, \ldots, a'_k, a'_{k+1}]}} &= (\Psi^{\underleftarrow{[a'_1, \ldots, a'_k]}})^{\underleftarrow{a'_{k+1}}} && \text{(por definición 3.10)} \\
&= (\Psi^{\underleftarrow{[a_1, \ldots, a_k]}})^{\underleftarrow{a'_{k+1}}} && \text{(\textit{por hipotesis inductiva})} \\
&= ((\Psi^{\underleftarrow{[a_1, \ldots, a_k]}} \setminus \mathsf{D}) \setminus \overline{\mathsf{A}}) \cup \mathsf{A} && \text{(por definición 3.8)} \\
&= (\Psi^{\underleftarrow{[a_1, \ldots, a_k]}})^{\underleftarrow{a_{k+1}}} && \text{(por definición 3.8 y } a_{k+1} \text{)} \\
&= \Psi^{\underleftarrow{[a_1, \ldots, a_k, a_{k+1}]}} && \text{(por definición 3.10)}
\end{aligned}
$$

A continuación se mostrará que, dado un estado S y siendo Δ' la transformación de Δ realizada por la función *quitar_restricciones*, un literal L está garantizado a partir de (S, Δ) si, y solo si, L está garantizado en (S,Δ'),

Lema 3.2 Sea $P = (\Psi, \Delta, \Gamma, Meta, R)$ un problema de planificación y $quitar\_restricciones(P) = (\Psi, \Delta', \Gamma', Meta', \mathsf{C}')$. Sea S un estado del mundo y L un literal en $literales(\Delta) \cup S$. Luego, *L está garantizado a partir de (S, Δ) si, y solo si, L está garantizado a partir de (S, Δ').*

Demostración:
Por la definición 3.13 sabemos que $\Delta' = \Delta \cup Reglas\_R \cup Reglas\_\Gamma$ (ver condición 1). Por otra parte, $Reglas\_R \cup Reglas\_\Gamma$ contiene reglas de la forma $Q \prec not\, C$, donde Q no aparecen en ninguna otra regla de Δ' (ver condiciones $1b$I y $1a$I de la definición 3.13). Por lo tanto, $literales(Reglas\_R \cup Reglas\_\Gamma) \cap literales(\Delta) = \emptyset$ y las reglas en $Reglas\_R \cup Reglas\_\Gamma$ no pueden combinarse con las reglas en Δ para formar argumentos. Luego, el conjunto $A_{\Delta'}$ que contiene todos los argumentos que se pueden construir a partir de (S, Δ'), puede dividirse en dos subconjuntos disjuntos A_{Δ} y A_{Reglas}, donde:

1. $A_{\Delta'} = A_{\Delta} \cup A_{Reglas}$,

2. A_{Δ} contiene todos los argumentos que se pueden construir a partir de (S, Δ), y

3. A_{Reglas} contiene todos los argumentos que se pueden construir a partir de $(S, Reglas\_R \cup Reglas\_\Gamma)$.

4. *los argumentos de* A_{Reglas} tienen la forma $\langle \{Q -\prec not\, C\}, Q\rangle$, donde el átomo Q no aparece en ninguna regla de Δ' y por lo tanto *no son derrotadores para ningún ningún argumento de* $A_{\Delta'}$.

($\Rightarrow$) *Si* L *está garantizado a partir de* (S, Δ) *entonces* L *está garantizado a partir de* (S, Δ'):
Si L está garantizado a partir de (S, Δ) entonces existe un argumento $\langle \mathcal{A}, L\rangle$ para L y un árbol de dialéctica marcado $\mathcal{T}^*_{\langle \mathcal{A}, L\rangle}$ cuya raíz está marcada con "U".

Dado que $A_{\Delta'} = A_{\Delta} \cup A_{Reglas}$, todos los argumentos que se pueden construir a partir de (S, Δ) se pueden construir a partir de (S, Δ'). Por otra parte, los argumentos de A_{Reglas} no son derrotadores de ningún ningún argumento de A_{Δ}. Luego, a partir de (S, Δ') se puede construir el mismo árbol de dialéctica $\mathcal{T}^*_{\langle \mathcal{A}, L\rangle}$ cuya raíz está marcada con "U" y por lo tanto L está garantizado a partir de (S, Δ')

($\Leftarrow$) *Si* L *está garantizado a partir de* (S, Δ') *entonces* L *está garantizado a partir de* (S, Δ)

Si L está garantizado a partir de (S, Δ') entonces existe un argumento $\langle \mathcal{A}', L\rangle$ para L y un árbol de dialéctica marcado $\mathcal{T}^*_{\langle \mathcal{A}', L\rangle}$ cuya raíz está marcada con "U".

Dado que $L \in literales(\Delta) \cup S$, pueden darse dos situaciones:

- Si $L \in S$, entonces por el lema 2.5 L está garantizado a partir de (S, Δ) por ser un hecho del programa.

- Si $L \in literales(\Delta)$ entonces, $\langle \mathcal{A}', L\rangle \in A_{\Delta}$ y $\mathcal{T}^*_{\langle \mathcal{A}', L\rangle}$ está construido sólo por argumentos de A_{Δ}, dado que los argumentos de A_{Reglas} no son derrotadores para ningún argumento de A_{Δ}. Luego, a partir de (S, Δ) se puede construir el mismo árbol de dialéctica $\mathcal{T}^*_{\langle \mathcal{A}, L\rangle}$ cuya raíz está marcada con "U" y por lo tanto L está garantizado a partir de (S, Δ).

El lema 3.3, presentado a continuación, muestra que para cada regla nueva $L -\prec not\, P$ incorporada a Δ' en la transformación de Δ, el literal L está garantizado a partir de (S, Δ') si, y solo si, P *no* está garantizado a partir de (S, Δ).

Lema 3.3 Sea $P = (\Psi, \Delta, \Gamma, Meta, R)$ un problema de planificación y $quitar\_restricciones(P) = (\Psi, \Delta', \Gamma', Meta', \mathsf{C}')$. Sea S un estado del mundo y sea $L \prec not\ P \in Reglas\_\Delta \cup Reglas\_\Gamma$. El literal L *está garantizado a partir de* (S, Δ') *si, y solo si,* P *no está garantizado a partir de* (S, Δ).
Demostración:

($\Rightarrow$) *Si* L *está garantizado a partir de* (S, Δ')*, entonces* P *no está garantizado a partir de* (S, Δ).

Si L está garantizado a partir de (S, Δ'), entonces:

(1) existe un argumento $\langle \mathcal{A}', L\rangle$ que no está derrotado

Como $L \prec not\ P \in Reglas\_\Delta \cup Reglas\_\Gamma$, entonces L no aparece en ninguna otra regla de Δ' y por lo tanto $\langle \{L \prec not\ P\}, L\rangle$ es el único argumento que se puede construir para L a partir de (S, Δ'). Luego, $\langle \mathcal{A}', L\rangle = \langle \{L \prec not\ P\}, L\rangle$.

Supongamos por el absurdo que P está garantizado a partir de (S, Δ). Por el lema 3.2 se puede asegurar que P está garantizado a partir de (S, Δ'). Por lo tanto, a partir de (S, Δ') se puede construir un argumento $\langle \mathcal{A}, P\rangle$ que no está derrotado y es un derrotador para $\langle \mathcal{A}', L\rangle = \langle \{L \prec not\ P\}, L\rangle$. Luego, $\langle \mathcal{A}', L\rangle$ está derrotado, lo cual contradice (1). Por lo tanto, P no está garantizado a partir de (S, Δ).

($\Leftarrow$) *Si* P *no está garantizado a partir de* (S, Δ)*, entonces* L *está garantizado a partir de* (S, Δ').

Si P no está garantizado a partir de (S, Δ) entonces pueden ocurrir dos situaciones:

- A partir de (S, Δ) no se puede construir ningún argumento para P.

  Como $L \prec not\ P \in Reglas\_\Delta \cup Reglas\_\Gamma$, entonces $P \prec not \cdots \notin Reglas\_\Delta \cup Reglas\_\Gamma$ (ver definición 3.13, condición $1c$), y por lo tanto tampoco se puede construir ningún argumento para P a partir de $(S, Reglas\_\Delta \cup Reglas\_\Gamma)$.

  El conjunto $A_{\Delta'}$ de todos los argumentos que se pueden construir a partir de (S, Δ'), es la unión del conjunto A_{Δ} de todos los argumentos que se pueden construir a partir de (S, Δ) y el conjunto A_{Reglas} de todos los argumentos que se pueden construir a partir de $(S, Reglas\_\Delta \cup Reglas\_\Gamma)$ (ver lema 3.2, 1). Luego, a partir de (S, Δ') no se puede construir ningún argumento para P y por lo tanto P no está garantizado a partir de (S, Δ').

- a partir de (S, Δ) se puede construir un argumento $\langle \mathcal{A}, P \rangle$ para P y un árbol de dialéctica marcado $\mathcal{T}^*_{\langle \mathcal{A}, P \rangle}$, cuya raíz está marcada con "D".

  Dado que $A_{\Delta'} = A_{\Delta} \cup A_{Reglas}$, todos los argumentos que se pueden construir a partir de (S, Δ) se pueden construir a partir de (S, Δ'). Por otra parte, los argumentos de A_{Reglas} no son derrotadores de ningún ningún argumento de A_{Δ}. Luego, a partir de (S, Δ') se puede construir el mismo árbol de dialéctica $\mathcal{T}^*_{\langle \mathcal{A}, P \rangle}$ cuya raíz está marcada con "D" y por lo tanto P no está garantizado a partir de (S, Δ').

Luego, P no está garantizado a partir de (S, Δ') y como $L \prec not\, P \in Reglas\_\Delta \cup Reglas\_\Gamma \subset \Delta'$, entonces L está garantizado a partir de (S, Δ').

Utilizando los lemas 3.2 y 3.3, se demostrará a continuación que una acción a es aplicable en un estado S si, y solo si, su transformación a' es aplicable en S.

Lema 3.4 Sea $P = (\Psi, \Delta, \Gamma, Meta, R)$ un problema de planificación y $quitar\_restricciones(P) = (\Psi, \Delta', \Gamma', Meta', \mathsf{C}')$. Sea $\langle a, \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C} \rangle$ una acción en Γ y sea $\langle a', \mathsf{A}, \mathsf{D}, \mathsf{P}', \mathsf{C}' \rangle$ la acción correspondiente en Γ'. *La acción a es* aplicable *en un estado S cualquiera si, y solo si, a' es* aplicable *en S.*

($\Rightarrow$) *Si a' es* aplicable *en un estado S cualquiera entonces a es* aplicable *en S.*
Demostración:

- Sea $L \in \mathsf{P}$. Por la definición 3.13, paso $2c$), tenemos que $\mathsf{P}' = \mathsf{P} \cup \mathsf{P_C}$ por lo tanto $L \in \mathsf{P}'$. Como a' es *aplicable* en S, entonces L está garantizado a partir de (S, Δ'). Luego, por el lema 3.2 se puede afirmar que L está garantizado a partir de (S, Δ).

- Sea $not\, C \in \mathsf{C}$. Por la definición 3.13, paso $2c$) tenemos que $L \prec not\, C \in Reglas\_\Gamma$. Como a' es *aplicable* en S entonces L está garantizado a partir de (S, Δ'). Luego, por el lema 3.3 se puede afirmar que C no está garantizado a partir de (S, Δ).

($\Leftarrow$) *Si a es* aplicable *en un estado S cualquiera entonces a' es* aplicable *en S.*
Demostración:
Por la definición 3.13, paso $2c$), tenemos que $\mathsf{C}' = \emptyset$ y $\mathsf{P}' = \mathsf{P} \cup \mathsf{P_C}$. Como $\mathsf{C}' = \emptyset$, luego a' será aplicable a partir de S si todos los literales presentes en P' están garantizados a partir de S. Sea $L \in \mathsf{P}' = \mathsf{P} \cup \mathsf{P_C}$:

- si $L \in \mathsf{P}$, como a *aplicable* en S, entonces L está garantizado a partir de (S, Δ). Luego, por el lema 3.2 se puede afirmar que L está garantizado a partir de (S, Δ')
- si $L \in \mathsf{P_C}$ entonces por la definición 3.13, paso $2c$) tenemos que $L \prec not\ P \in$ $Reglas\_\Gamma$ ($Reglas\_\Gamma \subset \Delta'$) y $not\ P \in \mathsf{C}$. Como a es *aplicable* en S entonces P no está garantizado a partir de (S, Δ). Luego por el lema 3.3 se puede afirmar que L está garantizado a partir de (S, Δ').

Por último, el siguiente teorema muestra la equivalencia entre un problema de planificación P y su transformación $quitar\_restricciones(P)$. Esto es, una secuencia de acciones $[a_1, \ldots, a_n]$ es una solución para P si, y solo si, su transformación $[a'_1, \ldots, a'_n]$ es una solución para $quitar\_restricciones(P)$.

Teorema 3.1 Sea $P = (\Psi, \Delta, \Gamma, Meta, R)$ un problema de planificación y $quitar\_restricciones(P) = (\Psi, \Delta', \Gamma', Meta', \mathsf{C}')$. Sea $[a_1, \ldots, a_n]$ una secuencia de acciones presentes en Γ y sea $[a'_1, \ldots, a'_n]$ la secuencia de acciones correspondientes en Γ'. *La secuencia* $[a_1, \ldots, a_n]$ *es una solución para* P *si, y solo si,* $[a'_1, \ldots, a'_n]$ *es una solución para* $quitar\_restricciones(P)$.
Demostración:

($\Rightarrow$) *Si* $[a_1, \ldots, a_n]$ *es una solución para* P *entonces* $[a'_1, \ldots, a'_n]$ *es una solución para* $quitar\_restricciones(P)$.

- $[a'_1, \ldots, a'_n]$ es aplicable a partir de Ψ.
  Inducción sobre la cantidad de acciones de la secuencia.
  - Caso base: Sea [ ] una secuencia vacía (0 acciones). Por la definición 3.9 es *aplicable* a partir de Ψ.
  - hipótesis inductiva: Sea $[a'_1, \ldots, a'_k]$ una secuencia de k acciones, $k > 0$.
    Supongamos que $[a'_1, \ldots, a'_k]$ es *aplicable* a partir de Ψ
  - Paso inductivo: Sea $[a'_1, \ldots, a'_k, a'_{k+1}]$ una secuencia de $k+1$ acciones.
    Por hipótesis sabemos que $[a_1, \ldots, a_n]$ es una solución para P y por lo tanto es aplicable a partir de Ψ. Luego, por la proposición 3.4 tenemos que:

    (1) cualquier *subsecuencia* $[a_1, \ldots, a_k]$,$k \leq n$ es aplicable a partir de Ψ.

Por otra parte, la *hipótesis inductiva* afirma que $[a'_1, \ldots, a'_k]$ es aplicable a partir de Ψ y por (1) sabemos que $[a_1, \ldots, a_k]$ es aplicable a partir de Ψ. Entonces, por el lema 3.1 se puede asegurar que:

$$(2)\ \Psi^{\overleftarrow{[a'_1, \ldots, a'_k]}} = \Psi^{\overleftarrow{[a_1, \ldots, a_k]}}$$

Por (1) y la definición 3.10 resulta que a_{k+1} es *aplicable* en $\Psi^{\overleftarrow{[a_1, \ldots, a_k]}}$. Luego, por (2) y la lema 3.4 se puede afirmar que:

$$(3)\ a'_{k+1} \text{ es } \textit{aplicable} \text{ a en } \Psi^{\overleftarrow{[a'_1, \ldots, a'_k]}}.$$

Finalmente, por la *hipótesis inductiva* y (3) resulta que $[a'_1, \ldots, a'_k, a'_{k+1}]$ es *aplicable* a partir de Ψ.

- Cada literal en $Meta'$ está garantizado a partir de $(\Psi^{\overleftarrow{[a'_1, \ldots, a'_n]}}, \Delta')$.
  Sea $L \in Meta' = Meta \cup \{L_C \mid L_C \prec not\, C \in Reglas\_\mathsf{C}\}$:
  - Si $L \in Meta$ entonces L está garantizado a partir de $(\Psi^{\overleftarrow{[a_1, \ldots, a_n]}}, \Delta)$, dado que $[a_1, \ldots, a_n]$ es una solución para P. Luego, por el lema 3.2 se puede afirmar que L está garantizado a partir de $(\Psi^{\overleftarrow{[a'_1, \ldots, a'_n]}}, \Delta')$.
  - Si $L \in \{L_C \mid L_C \prec not\, C \in Reglas\_\mathsf{C}\}$ entonces $L \prec not\, C \in Reglas\_\mathsf{C}$ y $C \in R$ (ver definición 2, paso $1a$). Como $C \in R$ y $[a_1, \ldots, a_n]$ es una solución para P entonces C *no* esta garantizado a partir de $(\Psi^{\overleftarrow{[a_1, \ldots, a_n]}}, \Delta)$. Luego por el lema 3.3 se puede afirmar que L está garantizado a partir de $(\Psi^{\overleftarrow{[a'_1, \ldots, a'_n]}}, \Delta')$.
- Para cada restricción $not\, C$ en C', C no está garantizado a partir de $(\Psi^{\overleftarrow{[a'_1, \ldots, a'_n]}}, \Delta')$.
  Trivial, dado que $\mathsf{C}' = \emptyset$.

($\Leftarrow$) *Si* $[a'_1, \ldots, a'_n]$ *es una solución para quitar_restricciones*(P) *entonces* $[a_1, \ldots, a_n]$ *es una solución para* P

- $[a_1, \ldots, a_n]$ es *aplicable* a partir de Ψ.
  Inducción sobre la cantidad de acciones de la secuencia.
  - Caso base: Sea [ ] una secuencia vacía (0 acciones). Por la definición 3.9 es *aplicable* a partir de Ψ.
  - hipótesis inductiva: Sea $[a_1, \ldots, a_k]$ una secuencia de k acciones, $k > 0$.
    Supongamos que $[a_1, \ldots, a_k]$ es *aplicable* a partir de Ψ.

- Paso inductivo: Sea $[a_1, \ldots, a_k, a_{k+1}]$ una secuencia de $k+1$ acciones.
  Por hipótesis sabemos que $[a'_1, \ldots, a'_n]$ es una solución para P y por lo tanto es aplicable a partir de Ψ. Luego, por la proposición 3.4 tenemos que:

  (1) cualquier *subsecuencia* $[a'_1, \ldots, a'_k]$,$k \leq n$ es *aplicable* a partir de Ψ.

  Por otra parte, la *hipótesis inductiva* afirma que $[a_1, \ldots, a_k]$ es *aplicable* a partir de Ψ y por (1) sabemos que $[a'_1, \ldots, a'_k]$ es *aplicable* a partir de Ψ. Entonces, por el lema 3.1 se puede asegurar que:

  $$(2)\ \Psi^{\overleftarrow{[a'_1, \ldots, a'_k]}} = \Psi^{\overleftarrow{[a_1, \ldots, a_k]}}$$

  Por (1) y la definición 3.10 resulta que a'_{k+1} es *aplicable* a partir de $\Psi^{\overleftarrow{[a'_1, \ldots, a'_k]}}$.
  Luego, por (2) y el lema 3.4 se puede afirmar que:

  $$(3)\ a_{k+1} \text{ es } \textit{aplicable} \text{ a partir de } \Psi^{\overleftarrow{[a_1, \ldots, a_k]}}.$$

  Finalmente, por la *hipótesis inductiva* y (3) se puede afirmar que $[a_1, \ldots, a_k, a_{k+1}]$ es *aplicable* a partir de Ψ.

- Cada literal en $Meta$ está garantizado a partir de $(\Psi^{\overleftarrow{[a_1, \ldots, a_n]}}, \Delta)$.
  Sea $L \in Meta$. Por la definición 3.13 paso 3) tenemos que $Meta' \subset Meta$, entonces $L \in Meta'$. Por hipótesis sabemos que $[a'_1, \ldots, a'_n]$ es una solución para $quitar\_restricciones(P)$, entonces L está garantizado a partir de $(\Psi^{\overleftarrow{[a'_1, \ldots, a'_n]}}, \Delta')$. Luego, por el lema 3.2 se puede afirmar que L está garantizado a partir de $(\Psi^{\overleftarrow{[a_1, \ldots, a_n]}}, \Delta)$.

- Para cada restricción *not* C en R, C no está garantizado a partir de $(\Psi^{\overleftarrow{[a_1, \ldots, a_n]}}, \Delta)$.
  Sea *not* $C \in R$. Por la definición 3.13 paso $1a$) tenemos que $L_C \prec$ *not* $C \in Reglas\_R$ y por el paso 3 $L_C \in Meta'$. Por hipótesis sabemos que $[a'_1, \ldots, a'_n]$ es una solución para $quitar\_restricciones(P)$, entonces L_C está garantizado a partir de $(\Psi^{\overleftarrow{[a'_1, \ldots, a'_n]}}, \Delta')$. Luego, por el lema 3.3 se puede afirmar que C *no* está garantizado a partir de $(\Psi^{\overleftarrow{[a_1, \ldots, a_n]}}, \Delta)$.

3.5. Resumen

En este capítulo se introdujo el formalismo DAKAR, el cual combina acciones y argumentación rebatible para modelar dominios y problemas de planificación. Lo novedo-

so de este formalismo es que permite definir acciones y representar conocimiento acerca del dominio utilizando la Programación en Lógica Rebatible, lo que permite razonar rebatiblemente acerca de las precondiciones, restricciones y efectos de las acciones en un determinado estado del mundo.

La sintaxis de DAKAR fue introducida en la sección 3.1 donde se definieron los conceptos básicos de este formalismo: *estado*, representado por un conjunto consistente de hechos; *esquema de acción* y *acción*; y finalmente *dominio de planificación*, el cuál está formado por un par (Δ,Γ) donde, Δ es un conjunto de reglas rebatibles que representan el conocimiento del dominio y Γ es un conjunto de esquemas de acción disponibles para modificar el estado del mundo.

La sección 3.2 introduce el concepto de *acción aplicable* que describe las condiciones que deben cumplirse para que una acción pueda ser ejecutada en un estado Ψ. Estas condiciones son evaluadas razonando rebatiblemente a partir de un programa DeLP formado por el estado Ψ y el conocimiento del dominio Δ.

Luego, en la sección 3.3 se definió el efecto que se produce sobre un estado al ejecutar una acción aplicable o una secuencia aplicable de acciones (plan). También se introdujeron algunas proposiciones que caracterizan a estos resultados, y se mostraron diferentes ejemplos.

Por último, en la sección 3.4 se definió el concepto de *problema de planificación* para dominios DAKAR y se estableció cuando una secuencia de acciones o plan constituye una solución para un problema de planificación. En la sección 3.4.1 se definió una función que permite transformar un problema de planificación en un problema equivalente que no contiene restricciones. A través del teorema 3.1 se demostró la equivalencia entre ambos problemas.

En el capítulo siguiente se mostrará como construir planes para resolver problemas de planificación definidos en DAKAR. Para ello se definirá una extensión de POP que utiliza argumentos además de acciones para satisfacer las precondiciones de las acciones.

Capítulo 4

Argumentación Rebatible y Planificación de Orden Parcial

En este capítulo se define un nuevo método de planificación que combina técnicas de Planificación de Orden Parcial con Argumentación Rebatible, y permite resolver problemas de planificación definidos utilizando DAKAR como formalismo de representación. La incorporación de Argumentación Rebatible permite utilizar el conocimiento del dominio, representado con reglas rebatibles, para razonar rebatiblemente durante la construcción de un plan.

El capítulo está organizado de la siguiente manera: En la sección 4.1 se presenta un ejemplo que introduce como el algoritmo de POP será extendido para incorporar argumentación rebatible. Por medio del ejemplo se introducirán también los conceptos principales que serán utilizados en lo que resta del capítulo. En la sección 4.2 se define una nueva estructura de plan, extendiendo los planes parciales para considerar argumentos como pasos del plan. Luego, en la sección 4.3 se extiende la noción de *amenaza* para considerar los nuevos tipos de interferencias que surgen al combinar acciones y argumentos, y se definen nuevos métodos para resolver cada una de ellas. En base a los conceptos y métodos definidos, en la sección 4.4 se presenta una extensión al algoritmo de POP que permite resolver problemas de planificación DAKAR.

4.1. Introducción

En el formalismo DAKAR una de las condiciones que deben cumplirse para poder ejecutar una acción es que todas sus precondiciones estén garantizadas. Dentro de un plan esto puede lograrse por medio de otras acciones o argumentos. Por otra parte, el método de planificación POP sólo considera el uso de acciones durante la construcción de un plan. Por lo tanto, para poder utilizar POP para construir planes para dominios de planificación DAKAR es necesario considerar el uso de argumentos.

A continuación se presenta un ejemplo sencillo que muestra como el algoritmo tradicional de POP puede ser extendido para considerar acciones y argumentos para la construcción de un plan. Esta nueva estructura de plan se denominará *Plan Parcial Argumentativo* y se definirá en detalle en la sección 4.2. A través del ejemplo también se introducirán intuitivamente algunos conceptos y la representación gráfica que será utilizada en el resto del capítulo.

Ejemplo 4.1 Considere el dominio de planificación $(\Delta_{4.1}, \Gamma_{4.1})$ introducido en el ejemplo 3.4, sobre un agente encargado de la limpieza y el mantenimiento de una oficina, donde:

$$\Delta_{4.1} = \left\{ \begin{array}{c} luz -\prec luz\_natural, \\ luz -\prec luz\_artificial, \\ luz\_natural -\prec pers\_alta, es\_dia, \\ luz\_artificial -\prec enc(L), lamp(L), \\ peligroso(I) -\prec int(I), mojado(I), \\ peligroso(I) -\prec int(I), roto(I), \\ \sim peligroso(I) -\prec int(I), \sim electricidad, \\ enc(L) -\prec con(I, L), enc(I), \\ \sim enc(L) -\prec lamp(L), \sim electricidad, \\ \sim enc(L) -\prec lamp(L), rota(L), \\ rep(L) -\prec lamp(L), not\ en\_uso(L), \\ en\_uso(L) -\prec con(I, L), \\ \sim rep(L) -\prec lamp(L), rota(L), \end{array} \right\}$$

$$\Gamma_{4.1} = \left\{ \begin{array}{c} \{pers\_alta\}, \{\}^- \xleftarrow{subir\_pers} \{\sim pers\_alta\}, not\ \{\} \\ \{\sim pers\_alta\}, \{\}^- \xleftarrow{bajar\_pers} \{pers\_alta\}, not\ \{\} \\ \{enc(I)\}, \{\}^- \xleftarrow{encender(I)} \{\sim enc(I), int(I)\}, not\ \{peligroso(I)\} \\ \{\sim enc(I)\}, \{\}^- \xleftarrow{apagar(I)} \{enc(I), int(I)\}, not\ \{peligroso(I)\} \\ \{con(I,R)\}, \{con(I,L)\}^- \xleftarrow{cambiar(L,I,R)} \{con(I,L), rep(R), \sim enc(I)\}, not\ \{\} \\ \{oficina\_limpia\}, \{\}^- \xleftarrow{limpiar} \{luz\}, not\ \{\} \end{array} \right\}$$

Sea $\Psi_{4.1}$ un estado del mundo donde es de día, la persiana de la oficina se encuentra baja, la lámpara l_2 está rota, los interruptores i_1 e i_2 se encuentran apagados y conectados a las lámparas l_1 y l_2 respectivamente. El estado $\Psi_{4.1}$ está definido por:

$$\begin{array}{rl} \Psi_{4.1}= & \{es\_dia, \sim pers\_alta, int(i_1), \sim enc(i_1), lamp(l_1), con(i_1, l_1), \\ & int(i_2), \sim enc(i_2), lamp(l_2), con(i_2, l_2), rota(l_2)\} \end{array}$$

Consideremos el problema de planificación $\mathrm{P}_{4.1} = (\Psi_{4.1}, \Delta_{4.1}, \Gamma_{4.1}, Meta_{4.1}, \mathsf{C}_{4.1})$ donde las metas son $Meta_{4.1} = \{oficina\_limpia\}$ y las restricciones $\mathsf{C}_{4.1} = \{\}$.

La figura 4.1 ilustra como construir un plan para el problema $\mathrm{P}_{4.1}$ incorporando el uso de argumentos al algoritmo de POP tradicional. Para esto se distinguirán dos tipos de pasos en el plan: *pasos de acción* y *pasos de argumento*.

El algoritmo comienza construyendo el *plan inicial* para el problema $\mathrm{P}_{4.1}$ (figura 4.1(a)) y luego intenta completarlo agregando pasos para lograr las *submetas* que inicialmente son las precondiciones del paso fin, en este caso: $oficina\_limpia$. La única forma de lograr la submeta $oficina\_limpia$ es utilizando la acción *limpiar*, por lo tanto se agrega un nuevo paso de acción y el vínculo causal correspondiente obteniendo el plan parcial que se muestra en la figura 4.1(b). Al igual que en el algoritmo de POP tradicional, los pasos de acción representan la ejecución de una acción y se representarán gráficamente a través de un rectángulo etiquetado con el nombre de la acción. Los literales que aparecen debajo y sobre un paso de acción representan las precondiciones y efectos del la acción respectivamente. Las flechas continuas que unen el efecto de un paso de acción con otro literal representan *vínculos causales* y las flechas rayadas entre dos pasos de acción representan *restricciones de orden*.

Una vez agregado el paso de acción *limpiar* (figura 4.1(b)), la precondición *luz* se convierte en una nueva submeta del plan. Observe que no existe ninguna acción disponible en $\Gamma_{4.1}$ que tenga como efecto producir *luz*. Sin embargo, un agente inteligente puede

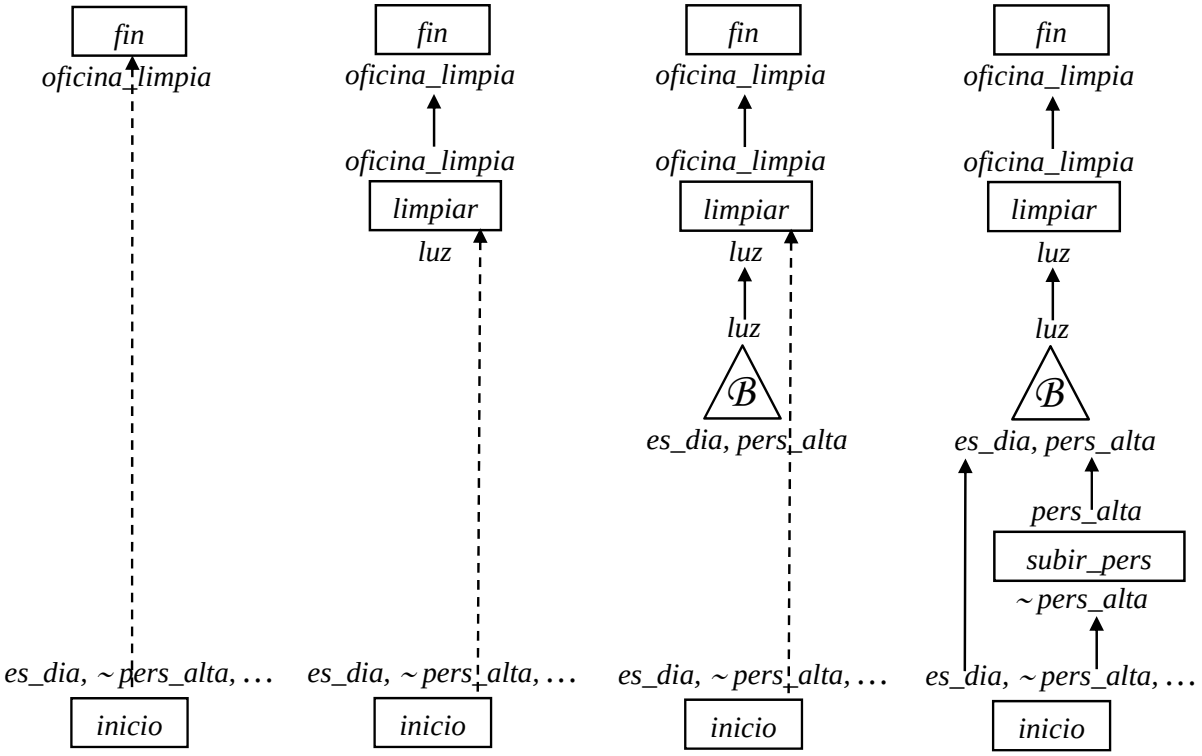

Figura 4.1: Construcción de un plan combinando acciones y argumentos

deducir que subir la persiana o encender un interruptor puede producir *luz* bajo ciertas condiciones. Note que a partir de la base de conocimiento $\Delta_{4.1}$ es posible construir el argumento (*potencial*) $\mathcal{B} = \{(luz -\prec luz\_natural), (luz\_natural -\prec pers\_alta, es\_dia)\}$. Luego, una forma alternativa para lograr *luz* es utilizar el argumento $\mathcal{B}$ para sustentar el literal *luz* y luego completar el plan para satisfacer los literales en la base de $\mathcal{B}$ (definición 4.2).

La figura 4.1(c) muestra esta situación. Para introducir el argumento $\mathcal{B}$ dentro del plan se utilizará un nuevo tipo de paso denominado *paso de argumento* y un nuevo tipo de vínculo denominado *vínculo de soporte*. Un paso de argumento se representará gráficamente por medio de un triángulo y representa el uso de un argumento para sustentar la precondición de un paso de acción. El literal en el tope del triángulo es la *conclusión* del argumento (definición 4.3) y los literales que aparecen en la base del triángulo representan la *base* del mismo. El *vínculo de soporte* se utiliza para vincular la conclusión del paso de argumento con la precondición del paso de acción que este sustenta y se representará gráficamente por medio de una flecha continua.

Cuando el paso de argumento $\mathcal{B}$ se agrega al plan, los literales presentes en su base ($base(\mathcal{B}) = \{es\_dia, pers\_alta\}$) se convierten en nuevas submetas del mismo. Dado que el literal *es_dia* es un efecto del paso *inicio* no es necesario agregar un nuevo paso para lograrlo, simplemente se agrega un vínculo causal. Para lograr el literal *pers_alta* se puede utilizar la acción *subir_pers*, cuya precondición $\sim pers\_alta$ es lograda por el paso *inicio*. Luego se agrega el paso de acción *subir_pers* junto con los vínculos causales correspondientes obteniendo el plan que se muestra en la figura 4.1(d).

Note que el argumento $\mathcal{B}$ es un "*argumento potencial*" porque depende de la existencia de un plan que satisfaga los literales presentes en su base. Este argumento no puede ser construido a partir de un conjunto de hechos como se hace en DeLP. Esto se debe a que al momento de construir el argumento es imposible conocer que literales son verdaderos, porque estos dependen de los pasos de acción que sean elegidos mas adelante durante el proceso de planificación.

A continuación se presenta una definición formal de argumento potencial [CCS01]. Previamente se introducen algunas definiciones que identifican diferentes conjuntos de literales de un argumento.

Definición 4.1 [Cabezas-Cuerpos-Literales] Dado un argumento $\langle \mathcal{B}, h \rangle$, $cabezas(\mathcal{B})$ es el conjunto de todos los literales que aparecen como cabezas de reglas en $\mathcal{B}$. De manera similar, $cuerpos(\mathcal{B})$ es el conjunto de todos los literales presentes en los cuerpos de la reglas en $\mathcal{B}$ (sin incluir los *literales extendidos*). El conjunto de todos los literales que aparecen en $\mathcal{B}$, denotado $literales(\mathcal{B})$, es el conjunto $cabezas(\mathcal{B}) \cup cuerpos(\mathcal{B})$.

Por ejemplo, para el argumento $\langle \mathcal{A}, a \rangle$, con $\mathcal{A} = \{(a -\prec b, c), (b -\prec d), (c -\prec not\ f)\}$, tenemos que: $cabezas(\mathcal{A}) = \{a, b, c\}$, $cuerpos(\mathcal{A}) = \{b, c, d\}$ y $literales(\mathcal{A}) = \{a, b, c, d\}$

Definición 4.2 [Base de un argumento] Dado un argumento $\langle \mathcal{B}, L \rangle$, la base de $\mathcal{B}$ es el conjunto $base(\mathcal{B}) = cuerpos(\mathcal{B}) \setminus cabezas(\mathcal{B})$.

Definición 4.3 [Conclusión de un argumento] Dado un argumento $\langle \mathcal{B}, L \rangle$, la conclusión de $\mathcal{B}$ es el literal $conclusion(\mathcal{B}) = cabezas(\mathcal{B}) \setminus cuerpos(\mathcal{B})$.

Definición 4.4 [Suposiciones de un argumento] Dado un argumento $\langle \mathcal{B}, L \rangle$, $suposiciones(\mathcal{B})$ es el conjunto de todos los *literales extendidos* de la forma $not\ P$ presentes en $\mathcal{B}$.

Considerando nuevamente el argumento $\langle \mathcal{A}, a \rangle$ tenemos que: $base(\mathcal{A}) = \{d\}$, $conclusion(\mathcal{A}) = a$ y $suposiciones(\mathcal{A}) = \{not\ f\}$.

Luego, un *argumento potencial* se define como sigue.

Definición 4.5 [Argumento Potencial]
Sea L un literal fijo y Δ un conjunto de reglas rebatibles fijas. Un subconjunto $\mathcal{A} \subseteq \Delta$ es un *argumento potencial* para L, si $\langle \mathcal{A}, L \rangle$ es un argumento a partir del programa DeLP $(base(\mathcal{A}), \Delta)$.

Ejemplo 4.2 Considere el conjunto de reglas rebatibles

$$\Delta = \{(a \prec b, c), (b \prec d), (c \prec not\ f), (p \prec q), (c \prec \sim d)\}$$

El subconjunto $\mathcal{A}$=$\{(a \prec b, c)$,$(b \prec d)$,$(c \prec not\ f)\}$ es un argumento potencial para a dado que $\mathcal{A} \subseteq \Delta$ y $\langle \mathcal{A}, a \rangle$ es un argumento construido a partir del programa DeLP $(base(\mathcal{A}), \Delta) = (\{c\}, \Delta)$.

El subconjunto $\mathcal{C}$=$\{(a \prec b, c)$,$(p \prec q)\}$ no es es un argumento potencial dado que $\mathcal{C}$ no es un argumento que se pueda construir a partir de $(base(\mathcal{C}), \Delta)$, pues $\mathcal{C}$ no constituye una derivación rebatible (condición 1, definición 2.24).

Observación 4.1 Cabe destacar que un argumento potencial tiene la misma estructura que un argumento, la diferencia está en cómo se construyen. Por lo tanto, sobre un argumento potencial se aplican todas las propiedades y conceptos definidos para un argumento en la sección 2.2.

Otra aspecto importante a tener en cuenta es que el uso del argumento $\mathcal{B}$ dentro del plan de la figura 4.1(c) no es suficiente para garantizar la precondición *luz* de la acción *limpiar*, porque podría existir un derrotador para $\mathcal{B}$. La existencia de este derrotador dependerá de las elecciones hechas más adelante en el proceso de planificación, es decir, un derrotador para $\mathcal{B}$ puede aparecer dentro del plan a medida que se agreguen nuevos pasos de acción al mismo. Al combinar acciones y argumentos en la construcción de un plan surgen nuevos tipos de interferencias entre los pasos del mismo. En la sección 4.3 extenderá la noción de *amenaza* para considerar los nuevos tipos de interferencias y se definirán nuevos métodos para resolver cada una de ellas. Pero antes, en la sección siguiente, se definirá una nueva estructura de plan parcial que considera acciones y argumentos.

4.2. Plan Parcial Argumentativo

En esta sección se extenderá la noción de plan parcial para considerar argumentos como pasos del plan. Un plan parcial que combina acciones y argumentos se denominará *plan parcial argumentativo* y se define como sigue:

Definición 4.6 (Plan Parcial Argumentativo) Sea $(\Psi, \Delta, \Gamma, Meta, R)$ un problema de planificación. Un *plan parcial argumentativo* (PPA) es una séxtupla $(\mathcal{AC}, \mathcal{AR}, \mathcal{O}, \mathcal{CL}, \mathcal{SL}, \mathcal{SG})$ donde:

- $\mathcal{AC}$ es un conjunto de *pasos de acción* de la forma $(S_{ac}, \mathsf{N}, Pre, Ef)$ donde S_{ac} es el nombre del paso, $\langle \mathsf{N}, \mathsf{A}, \mathsf{D}, \mathsf{P}, \mathsf{C} \rangle \in \Gamma$ es una acción asociada al paso, Pre=P son las precondiciones del paso y Ef= $\mathsf{A} \cup \mathsf{D}$ son los efectos del paso.
- $\mathcal{AR}$ es un conjunto de *pasos de argumento* de la forma $(S_{arg}, \mathcal{B}, \Lambda_{\mathcal{B}})$ donde S_{arg} es el nombre del paso, $\mathcal{B}$ es un argumento potencial asociado al paso obtenido a partir de Δ y $\Lambda_{\mathcal{B}}$ es una línea de argumentación asociada al paso.
- $\mathcal{O}$ es un conjunto de *restricciones de orden* de la forma $S_i \prec S_j$, donde $\{(S_i, \cdot, \cdot, \cdot), (S_j, \cdot, \cdot, \cdot)\} \subseteq \mathcal{AC} \cup \mathcal{AR}$
- $\mathcal{CL}$ es un conjunto de *vínculos causales* de la forma $S_i \xrightarrow{P} S_j$ donde $(S_i, A_i, Pre_i, Ef_i) \in \mathcal{AC}$, $P \in Ef_i$ y se cumple una de las siguientes condiciones:
  - $(S_j, A_j, Pre_j, Ef_j) \in \mathcal{AC}, P \in Pre_j$, ó
  - $(S_j, \mathcal{B}, \Lambda_{\mathcal{B}}) \in \mathcal{AR}, P \in base(\mathcal{B})$.
- $\mathcal{SL}$ es un conjunto de *vínculos de soporte* de la forma $S_i \overset{P}{\succ} S_j$ donde:
  - $(S_i, \mathcal{B}, \cdot) \in \mathcal{AR}, (S_j, \cdot, Pre_j, \cdot) \in \mathcal{AC}$, $P \in conclusion(\mathcal{B})$, $P \in Pre_j$, y
  - dados $S_a \overset{\cdot}{\succ} A \in \mathcal{SL}$ y $S_b \overset{\cdot}{\succ} B \in \mathcal{SL}$ con $A \neq B$ se cumple que $S_a \neq S_b$.
- $\mathcal{SG}$ es un conjunto de *submetas* de la forma $(P, S, Tipo, \Lambda_P)$, donde Λ_P es una línea de argumentación asociada a P y se cumple alguna de las siguientes condiciones:
  - $(S, \cdot, Pre, \cdot) \in \mathcal{AC}$, $P \in Pre$ y $Tipo \in \{ac, arg, ac\_arg\}$[1]
  - $(S, \mathcal{B}, \cdot) \in \mathcal{AR}$, $P \in base(\mathcal{B})$ y $Tipo = ac$

[1] los valores *ac*, *arg* y *ac_arg* indican que la submeta debe ser lograda por una acción, por un argumento o por cualquiera de los dos respectivamente.

Cada *paso de acción* representa la ejecución de una acción dentro del plan. La misma acción puede estar asociada a diferentes pasos de acción en el plan, por esto es necesario distinguir entre acciones y pasos de acción. Por ejemplo, el paso de acción $(Ac_1, limpiar, \{luz\}, \{oficina\_limpia\})$ representa la ejecución de la acción *limpiar* en el plan de la figura 4.1(b).

Cada *paso de argumento* representa un proceso deliberativo que utiliza un argumento para sustentar una precondición de algún paso de acción dentro del plan. Al igual que las acciones, un mismo argumento puede estar asociado a diferentes pasos de argumento de un plan. Por ejemplo, el paso de argumento $(Arg_1, \mathcal{B}, [\langle \mathcal{B}, luz \rangle])$ representa el uso del argumento $\mathcal{B} = \langle \{(luz \prec luz\_natural), (luz\_natural \prec pers\_alta, es\_dia)\}, luz \rangle$ en el plan de la figura 4.1(c) para sustentar la precondición *luz* de la acción *limpiar*.

Una restricción de orden la forma $S_i \prec S_j$ establece que S_i debe ejecutarse antes que S_j. El conjunto de restricciones de orden establece un *orden parcial* entre los pasos de acción del plan y debe ser *consistente*, en el sentido que debe existir al menos un *orden total* que las satisfaga. Por ejemplo, la flecha rayada entre las acciones *inicio* y *limpiar* del plan de la figura 4.1(c) corresponde a la restricción de orden $inicio \prec limpiar$.

Los *vínculos causales* y *de soporte* se utilizan para registrar el origen de cada literal y establecer dependencias entre los pasos del plan.

Un vínculo causal de la forma $S_i \xrightarrow{p} S_j$, donde $(S_j, ...) \in \mathcal{AC}$, representa que la precondición p del paso S_j es lograda por un efecto del paso S_i en el plan. Si $(S_j, \mathcal{B}, ...) \in \mathcal{AR}$, el vínculo $S_i \xrightarrow{p} S_j$ representa que el literal p presente en la base de $\mathcal{B}$ es logrado por un efecto del paso S_i. Dado que una causa debe aparecer antes del efecto que produce, un vínculo causal también representa una restricción de orden. Por ejemplo, $Ac_1 \xrightarrow{oficina\_limpia} fin$ corresponde al vínculo causal representado en la figura 4.1 por la flecha continua entre el efecto de *limpiar* y la precondición de fin.

Un vínculo de soporte de la forma $S_i \overset{p}{\succ\!\!-} S_j$ representa que la precondición p del paso S_j es sustentada por la conclusión del argumento asociado al paso S_i Por ejemplo, $\mathcal{A}_1 \overset{oficina\_limpia}{\succ\!\!-} Ac_1$ corresponde al vínculo de soporte representado en la figura 4.1 por la flecha continua entre la conclusión de $\mathcal{B}$ y la precondición de *limpiar*.

El conjunto $\mathcal{SG}$ contiene las submetas que no están aún logradas en el plan. Esto es, precondiciones de acciones o literales presentes en la base de argumentos que no están sustentadas por ningún vínculo causal o de soporte. En una submeta $(p, S, Tipo, \Lambda_p)$, el valor

de $Tipo$ determina qué tipo de paso debe utilizarse para lograr p. Si $Tipo = ac$ la submeta debe ser lograda por un paso de acción, si $Tipo = arg$ la submeta debe ser lograda por un paso de argumento y si $Tipo = ac\_arg$ la submeta puede ser lograda alternativamente por un paso de acción o de argumento. Por ejemplo, la submeta $(luz, Ac_1, ac\_arg, [\,])$ corresponde a la precondición luz de la acción $limpiar$ en el plan de la figura 4.1(b). La submeta $(es\_dia, Arg_1, ac, [\,])$ corresponde al literal es_dia de la base de $\mathcal{B}$ en en el plan de la figura 4.1(c).

Ejemplo 4.3 A continuación se definen los PPA que aparecen en la figura 4.1:

- P(a)=$(\mathcal{AC}_a, \mathcal{AR}_a, \mathcal{O}_a, \mathcal{CL}_a, \mathcal{SL}_a, \mathcal{SG}_a)$,corresponde a la figura 4.1(a), donde:
  - $\mathcal{AC}_a = \{(inicio, \cdot, \emptyset, \{es\_dia, \sim pers\_alta\}), (fin, \cdot, \emptyset, \{oficina\_limpia\})\}$
  - $\mathcal{AR}_a = \emptyset$
  - $\mathcal{O}_a = \{inicio \prec fin\}$
  - $\mathcal{CL}_a = \emptyset$
  - $\mathcal{SL}_a = \emptyset$
  - $\mathcal{SG}_a = \{(oficina\_limpia, fin, ac\_arg, [\,])\}$
- P(b)=$(\mathcal{AC}_b, \mathcal{AR}_b, \mathcal{O}_b, \mathcal{CL}_b, \mathcal{SL}_b, \mathcal{SG}_b)$,corresponde a la figura 4.1(b), donde:
  - $\mathcal{AC}_b = \{(inicio, \cdot, \emptyset, \{es\_dia, \sim pers\_alta\}), (fin, \cdot, \emptyset, \{oficina\_limpia\}),$ $(Ac_1, limpiar, \{luz\}, \{oficina\_limpia\})\}$
  - $\mathcal{AR}_b = \emptyset$
  - $\mathcal{O}_b = \{inicio \prec fin, inicio \prec Ac_1, Ac_1 \prec fin\}$
  - $\mathcal{CL}_b = \{Ac_1 \overset{oficina\_limpia}{\longrightarrow} fin\}$
  - $\mathcal{SL}_b = \emptyset$
  - $\mathcal{SG}_b = \{(luz, Ac_1, ac\_arg, [\,])\}$
- P(c)=$(\mathcal{AC}_c, \mathcal{AR}_c, \mathcal{O}_c, \mathcal{CL}_c, \mathcal{SL}_c, \mathcal{SG}_c)$, corresponde a la figura 4.1(c), donde:
  - $\mathcal{AC}_c = \{(inicio, \cdot, \emptyset, \{es\_dia, \sim pers\_alta\}), (fin, \cdot, \emptyset, \{oficina\_limpia\}),$ $(Ac_1, limpiar, \{luz\}, \{oficina\_limpia\})\}$
  - $\mathcal{AR}_c = \{(Arg_1, \mathcal{B}, [\langle \mathcal{B}, luz \rangle])\}$, con $\mathcal{B} = \langle \{(luz \prec luz\_natural), (luz\_natural \prec pers\_alta, es\_dia)\}, luz \rangle$

- $\mathcal{O}_c = \{inicio \prec fin, inicio \prec Ac_1, Ac_1 \prec fin\}$
- $\mathcal{CL}_c = \{Ac_1 \overset{oficina\_limpia}{\longrightarrow} fin\}$
- $\mathcal{SL}_c = \{Arg_1 \overset{luz}{\succ\!\!-} Ac_1\}$
- $\mathcal{SG}_c = \{(es\_dia, Arg_1, ac, [\,]), (pers\_alta, Arg_1, ac, [\,])\}$

- P(d)=$(\mathcal{AC}_d, \mathcal{AR}_d, \mathcal{O}_d, \mathcal{CL}_d, \mathcal{SL}_d, \mathcal{SG}_d)$, corresponde a la figura 4.1(d), donde:
  - $\mathcal{AC}_d = \{(inicio, \cdot, \emptyset, \{es\_dia, \sim pers\_alta\}), (fin, \cdot, \emptyset, \{oficina\_limpia\}),$ $(Ac_1, limpiar, \{luz\}, \{oficina\_limpia\}), (Ac_2, subir\_pers, \{\sim pers_a lta\}, \{pers_a lta\})\}$
  - $\mathcal{AR}_d = \{(Arg_1, \mathcal{B}, [\langle \mathcal{B}, luz\rangle])\}$, con $\mathcal{B} = \langle\{(luz -\!\prec luz\_natural), (luz\_natural -\!\prec pers\_alta, es\_dia)\}, luz\rangle$
  - $\mathcal{O}_d = \{inicio \prec fin, inicio \prec Ac_1, Ac_1 \prec fin, inicio \prec Ac_2, Ac_2 \prec fin,$ $Ac_2 \prec Ac_1\}$
  - $\mathcal{CL}_d = \{Ac_1 \overset{oficina\_limpia}{\longrightarrow} fin, Ac_2 \overset{pers\_alta}{\longrightarrow} Arg_1, inicio \overset{es\_dia}{\longrightarrow} Arg_1, inicio \overset{\sim pers\_alta}{\longrightarrow} Ac_2, \}$
  - $\mathcal{SL}_d = \{Arg_1 \overset{luz}{\succ\!\!-} Ac_1\}$
  - $\mathcal{SG}_d = \emptyset$

Al igual que en un plan parcial tradicional, las restricciones de orden imponen un orden parcial sobre los pasos de acción de un PPA. Luego, para que un PPA sea una solución para un problema de planificación DAKAR, todo orden topológico debe ser una solución. Formalmente,

Definición 4.7 (orden topológico de un PPA)
Sea $P = (\mathcal{AC}, \mathcal{AR}, \mathcal{O}, \mathcal{CL}, \mathcal{SL}, \mathcal{SG})$ un PPA. Una secuencia totalmente ordenada $S = [(inicio, \cdot, \cdot, \cdot), (s_1, accion_1, \cdot, \cdot), \ldots, (s_n, accion_n, \cdot, \cdot), (fin, \cdot, \cdot, \cdot)]$ de pasos de acción es un orden topológico para P si:

- $\{(inicio, \cdot, \cdot, \cdot), (s_1, accion_1, \cdot, \cdot), \ldots, (s_n, accion_n, \cdot, \cdot), (fin, \cdot, \cdot, \cdot)\} = \mathcal{AC}$, esto es, cada paso de acción de la secuencia pertenece a $\mathcal{AC}$ y viceversa.
- $s_i \prec s_j$ *es consistente con* $\mathcal{O}$, para todo $0 \leq i < j \leq n$

Se denominará $Plan(S) = [accion_1, \ldots, accion_n]$ a la secuencia de acciones que se obtiene de reemplazar cada paso de acción en S por su acción correspondiente, con excepción de los pasos *inicio* y fin. Esto se debe a que los pasos *inicio* y fin se utilizan

para representar el estado inicial y las metas del problema, y no se corresponden con la ejecución de ninguna acción.

Definición 4.8 (PPA solución)
Un PPA P es una solución para un problema de planificación $Prob$ si *para todo* orden topológico $S = [(inicio, \cdot, \cdot, \cdot), (s_1, accion_1, \cdot, \cdot), \ldots, (s_n, accion_n, \cdot, \cdot), (fin, \cdot, \cdot, \cdot)]$ para P, $Plan(S) = [accion_1, \ldots, accion_n]$ es una solución para $Prob$.

Por ejemplo, el PPA $P(d)$ del ejemplo 4.3 (figura 4.1(d)) es una solución para el problema $P_{4.1}$ del ejemplo 4.1, dado que existe un único orden topológico $S = [(inicio, \cdot, \cdot, \cdot), (Ac_1, limpiar, \cdot, \cdot), (Ac_2, subir\_pers, \cdot, \cdot), (inicio, \cdot, \cdot, \cdot)]$ para $P_{4.1}$ y $Plan(S) = [subir\_pers, limpiar]$ es una solución para $P_{4.1}$.

Como se explicó en la sección 2.1.2, en un plan parcial tradicional puede existir un solo tipo de amenaza que se puede resolver a través de dos métodos (*avance* y *retroceso*). Con el objetivo de hacer mas clara la presentación, el ejemplo elegido para introducir los conceptos de las secciones 4.1 y 4.2 no contiene amenazas. Sin embargo, un PPA puede presentar este tipo de amenaza y nuevos tipos que involucran argumentos.

4.3. Amenazas en un Plan Parcial Argumentativo

Al igual que un plan parcial, un plan parcial argumentativo PPA puede contener interferencias entre sus pasos. En un PPA la interacción entre acciones y argumentos genera nuevos tipos de interferencias que deben ser detectadas y resueltas para evitar que invaliden el plan. En esta sección se extenderá la noción de *amenaza* para considerar los distintos tipos de interferencias que surgen en un PPA y definir las siguientes amenazas:

- *amenaza acción-acción* (Sección 4.3.1)
- *amenaza acción-base* (Sección 4.3.2)
- *amenaza acción-argumento* (Sección 4.3.3)
- *amenaza acción-suposición* (Sección 4.3.4)
- *amenaza argumento-argumento* (Sección 4.3.5)

Además, se definirán nuevos métodos para resolver cada una de ellas.

4.3.1. Amenaza acción-acción

Este tipo de amenaza involucra sólo pasos de acción y es una extensión al tipo de amenaza que aparece en un plan parcial tradicional (Definición 2.13) para considerar los efectos negativos. Considere las situaciones planteadas en las figuras 4.2(a) y 4.2(b) que muestran parte de un PPA. En la figura 4.2(a) el paso de acción S_k tiene $\overline{L}$ como efecto, mientras en la figura 4.2(b) el paso de acción S_k tiene L^- como efecto. En ambos casos, las restricciones de orden permiten que S_k se ejecute entre los paso de acción S_i y S_j. Si esto ocurre, S_k romperá el vínculo causal $S_i \xrightarrow{L} S_j$, impidiendo que S_j pueda ejecutarse. Una *amenaza acción-acción* se define formalmente como sigue.

Definición 4.9 (amenaza acción-acción)
Sea $(\mathcal{AC}, \mathcal{AR}, \mathcal{O}, \mathcal{CL}, \mathcal{SL}, \mathcal{SG})$ un PPA, sea $S_i \xrightarrow{L} S_j$ un vínculo causal en $\mathcal{CL}$ donde $(S_j, \cdot, Pre_j, \cdot)$ es un paso de acción en $\mathcal{AC}$ y $L \in Pre_j$. Sea $(S_k, \cdot, \cdot, Ef_k)$ un paso de acción en $\mathcal{AC}$ ($S_k \neq S_i$, $S_k \neq S_j$), se dice que S_k es una *amenaza acción-acción* para $S_i \xrightarrow{L} S_j$ si:

- $\{S_i \prec S_k \prec S_j\}$ *es consistente con* $\mathcal{O}$,
- $\overline{L} \in Ef_k$ ó $L^- \in Ef_k$.

Cabe recordar que, como fue introducido en las definición 2.12, se dice que $S_i \prec S_j$ *es consistente con* $\mathcal{O}$, si $\mathcal{O} \cup \{S_i \prec S_j\}$ es un *orden parcial estricto*, o equivalentemente, $S_j \prec S_i \notin \mathcal{O}^+$ donde $\mathcal{O}^+$ denota la clausura transitiva de $\mathcal{O}$. A su vez, $S_i \prec S_k \prec S_j$ *es consistente con* $\mathcal{O}$, si $\mathcal{O} \cup \{S_i \prec S_k, S_k \prec S_j\}$ es un *orden parcial estricto*.

Para resolver una *amenaza acción-acción* es necesario evitar que S_k se ejecute entre S_i y S_j, esto puede lograrse por medio de uno de los siguientes *métodos*:

- *retroceso*: si $S_k \prec S_i$ *es consistente con* $\mathcal{O}$ entonces $S_k \prec S_i$ se agrega al conjunto $\mathcal{O}$ (figura 4.2(d)),
- *avance*: si $S_j \prec S_k$ *es consistente con* $\mathcal{O}$ entonces $S_j \prec S_k$ se agrega al conjunto $\mathcal{O}$ (figura 4.2(c)).

Cualquiera de los métodos puede aplicarse siempre y cuando la restricción de orden que agrega sea consistente con las restricciones de orden del plan.

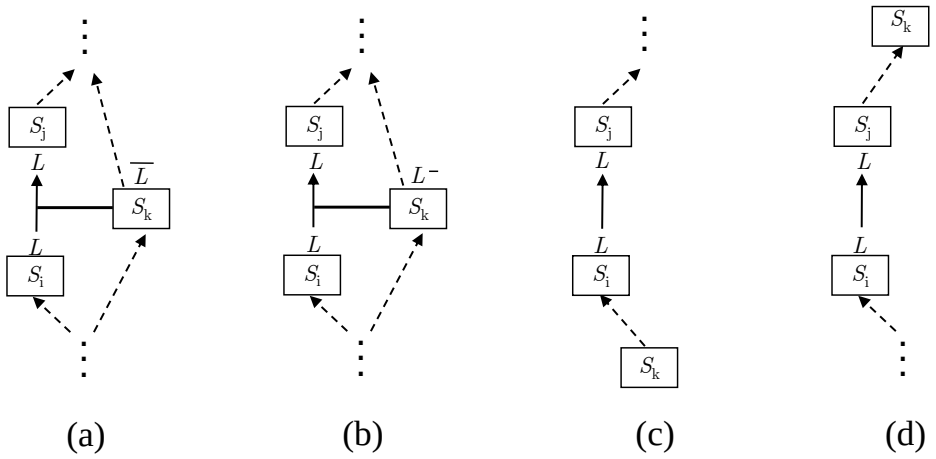

Figura 4.2: **(a)**, **(b)** *amenazas acción-acción*. La amenazas pueden ser resueltas por: **(c)** *retroceso* ó **(d)** *avance*.

Ejemplo 4.4 Sea $(\Psi_{4.4}, \Delta_{4.4}, \Gamma_{4.4}, Meta_{4.4}, \mathsf{C}_{4.4})$ un problema de planificación donde:

- $\Psi_{4.4} = \{c, q, r\}$
- $\Delta_{4.4} = \emptyset$
- $\Gamma_{4.4} = \left\{ \begin{array}{cc} \{p, \sim b\}, \{\}^- \xleftarrow{a_1} \{q\}, not\ \{\}, & \{p\}, \{b\}^- \xleftarrow{a_2} \{r\}, not\ \{\}, \\ \{b\}, \{\}^- \xleftarrow{a_3} \{c\}, not\ \{\}, & \{a\}, \{\}^- \xleftarrow{a_4} \{b\}, not\ \{\} \end{array} \right\}$
- $Meta_{4.4} = \{a, p\}$
- $\mathsf{C}_{4.4} = \emptyset$

La figura 4.3(a) muestra un plan posible al que puede arribar el planificador durante la búsqueda de una solución para el problema 4.4. Como puede observarse el paso de acción a_1 es una *amenaza acción-acción* para el vínculo causal $a_3 \xrightarrow{b} a_4$, dado que a_1 tiene $\sim b$ como efecto y puede ejecutarse entre a_3 y a_4. En este caso, las restricciones de orden permiten aplicar cualquiera de los dos métodos para resolver esta amenaza. Las figuras 4.3(b) y (c) muestran los planes que se obtienen al aplicar los métodos *retroceso* y *avance* respectivamente. Note que ambos planes son una solución para el problema.

La figura 4.3(d) muestra otro plan posible al que puede arribar el planificador durante la búsqueda. En este caso el planificador eligió la acción a_2 para lograr la meta p, la cual es una es una *amenaza acción-acción* para el vínculo causal $a_3 \xrightarrow{b} a_4$, dado que a_2

tiene $-b$ como efecto y puede ejecutarse entre a_3 y a_4. Nuevamente, las restricciones de orden permiten aplicar cualquiera de los dos métodos para resolver esta amenaza. Las figuras 4.3(e) y (f) muestran los planes que se obtienen al aplicar los métodos *retroceso* y *avance* respectivamente, los cuales son una solución para el problema.

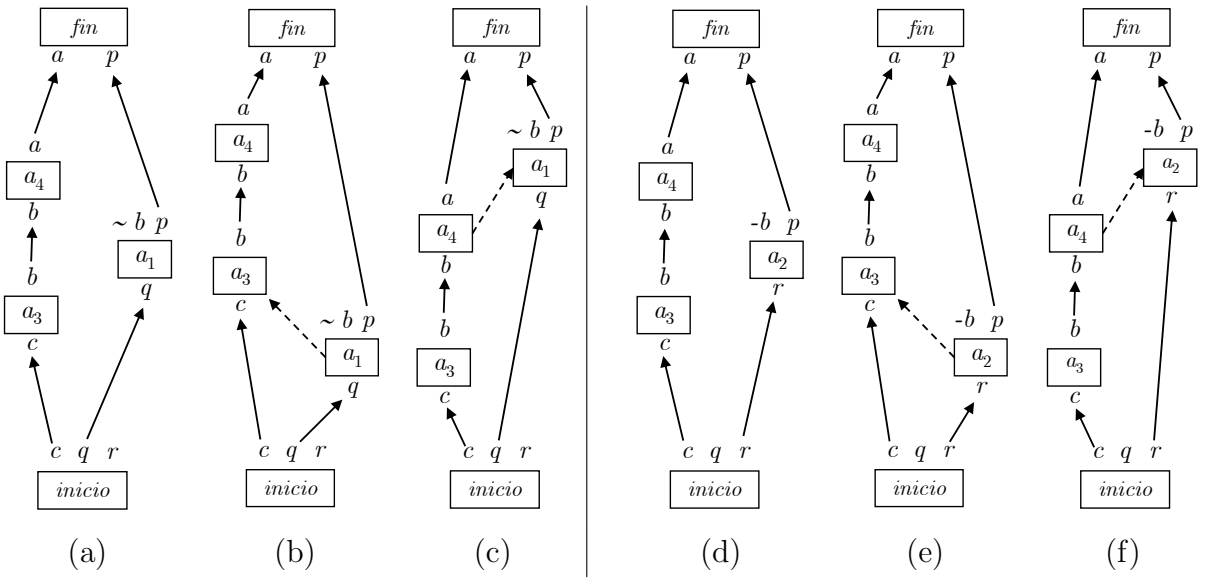

Figura 4.3: Resolviendo *amenazas acción-acción* para el ejemplo 4.4

4.3.2. Amenaza acción-base

Este tipo de amenaza se produce cuando un paso de acción tiene el efecto de borrar o negar un literal de la base de un paso de argumento. Considere las situaciones planteadas en la figuras 4.4(a) y 4.4(b). En ambos casos el paso de acción S_k amenaza el vínculo causal $S_i \xrightarrow{L} S_j$ donde S_j es un paso de argumento. Note que en la figura 4.4(a) el paso de acción S_k tiene $\overline{L}$ como efecto, mientras en la figura 4.4(b) S_k tiene $-L$ como efecto. A su vez, S_j se utiliza para sustentar una precondición de un paso de acción S_{ac} y las restricciones de orden permiten que S_k se ejecute entre S_i y S_{ac}. Si esto último ocurre, el argumento asociado a S_j no podrá construirse y S_{ac} no podrá ejecutarse.

Para identificar y definir este tipo de amenaza es necesario considerar el paso de acción soportado por un paso de argumento dentro de un plan, por este motivo se introducirán primero las siguientes definiciones.

Proposición 4.1 (vínculo de soporte único)
Sea $(\mathcal{AC}, \mathcal{AR}, \mathcal{O}, \mathcal{CL}, \mathcal{SL}, \mathcal{SG})$ un PPA y $S_{arg} \in \mathcal{AR}$ un paso de argumento entonces existe un único vínculo de soporte $S_{arg} \succ\!\!- S_{ac} \in \mathcal{SL}$.
Demostración: Directo por definición 4.6.

Definición 4.10 (paso de acción soportado)
Sea $(\mathcal{AC}, \mathcal{AR}, \mathcal{O}, \mathcal{CL}, \mathcal{SL}, \mathcal{SG})$ un PPA, $S_{arg} \in \mathcal{AR}$ un paso de argumento y $S_{arg} \succ\!\!- S_{ac} \in \mathcal{SL}$ su único vínculo de soporte asociado. La función *soportado* : $\mathcal{AR} \mapsto \mathcal{AC}$ se define como: $soportado(S_{arg})$=S_{ac}. Se dice que S_{ac} es el paso de acción soportado por S_{arg}.

Por ejemplo, en la figura 4.4 $soportado(S_j) = S_{ac}$.

Luego, una *amenaza acción-base* se define formalmente como sigue.

Definición 4.11 (amenaza acción-base)
Sea $(\mathcal{AC}, \mathcal{AR}, \mathcal{O}, \mathcal{CL}, \mathcal{SL}, \mathcal{SG})$ un PPA, sea $S_i \xrightarrow{L} S_j$ un vínculo causal en $\mathcal{CL}$ donde $(S_j, \cdot, \cdot, \cdot)$ es un paso de argumento en $\mathcal{AR}$ y S_{ac}=$soportado(S_j)$. Sea $(S_k, \cdot, \cdot, Ef_k)$ un paso de acción en $\mathcal{AC}$ ($S_k \neq S_i$, $S_k \neq S_{ac}$), se dice que S_k es una *amenaza acción-base* para $S_i \xrightarrow{L} S_j$ si:

- $\{S_i \prec S_k \prec S_{ac}\}$ *es consistente con* $\mathcal{O}$,
- $\overline{L} \in Ef_k$ ó $-L \in Ef_k$

Una *amenaza acción-base* puede resolverse por medio de uno de los siguientes *métodos*:

- *retroceso**: si $S_k \prec S_i$ *es consistente con* $\mathcal{O}$ entonces agregar $S_k \prec S_i$ al conjunto $\mathcal{O}$ (figura 4.4(c)).
- *avance**: si $S_{ac} \prec S_k$ *es consistente con* $\mathcal{O}$ entonces agregar $S_{ac} \prec S_k$ al conjunto $\mathcal{O}$, (figura 4.4(d)).

Ambos métodos impiden que S_k se ejecute entre S_i y S_{ac}, permitiendo que el argumento asociado al paso S_j pueda construirse para soportar la precondición del paso S_{ac}. Cualquiera de los métodos puede aplicarse siempre y cuando la restricción de orden que se agrega sea consistente con las restricciones de orden del plan.

Ejemplo 4.5 Sea $(\Psi_{4.5}, \Delta_{4.5}, \Gamma_{4.5}, Meta_{4.5}, \mathsf{C}_{4.5})$ un problema de planificación donde:

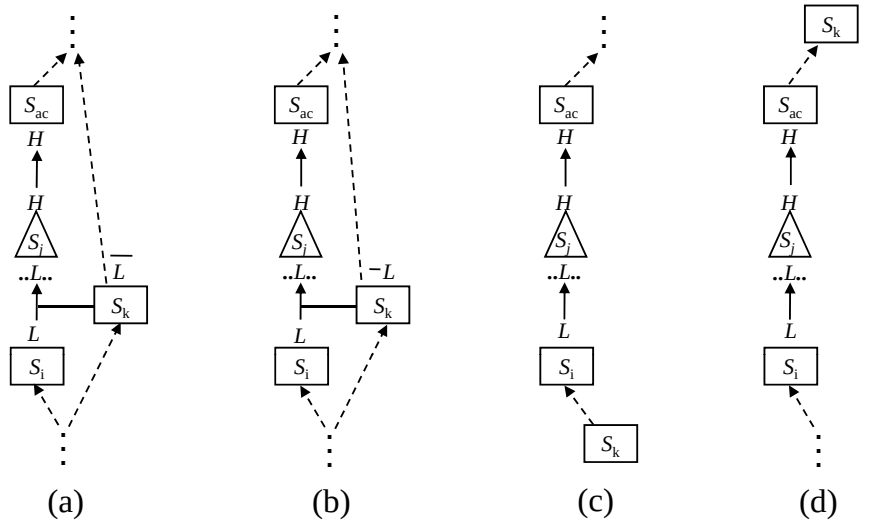

Figura 4.4: **(a)**,**(b)** *amenaza acción-base*. La amenaza puede ser resuelta por: **(c)** *retroceso** ó **(d)** *avance**.

- $\Psi_{4.5} = \{c, q, r\}$
- $\Delta_{4.5} = \{x \prec b\}$
- $\Gamma_{4.5} = \left\{ \begin{array}{cc} \{p, \sim b\}, \{\}^- \xleftarrow{a_1} \{q\}, not\ \{\}, & \{p\}, \{b\}^- \xleftarrow{a_2} \{r\}, not\ \{\}, \\ \{b\}, \{\}^- \xleftarrow{a_3} \{c\}, not\ \{\}, & \{a\}, \{\}^- \xleftarrow{a_5} \{x\}, not\ \{\} \end{array} \right\}$
- $Meta_{4.5} = \{a, p\}$
- $\mathsf{C}_{4.5} = \emptyset$

La figura 4.5(a) muestra un plan posible al que puede arribar el planificador durante la búsqueda de una solución para el problema 4.5. Como puede observarse el paso de acción a_1 es una *amenaza acción-base* para el vínculo causal $a_3 \xrightarrow{b} \mathcal{B}$, dado que a_1 tiene $\sim b$ como efecto y puede ejecutarse entre a_3 y a_5, el paso soportado por $\mathcal{B}$. En este caso, las restricciones de orden permiten aplicar cualquiera de los dos métodos para resolver esta amenaza. Las figuras 4.5(b) y (c) muestran los planes que se obtienen al aplicar los métodos *retroceso** y *avance** respectivamente. Note que ambos planes son una solución para el problema.

La figura 4.5(d) muestra otro plan posible al que puede arribar el planificador durante la búsqueda. En este caso el planificador eligió la acción a_2 para lograr la meta p, la cual es una es una *amenaza acción-base* para el vínculo causal $a_3 \xrightarrow{b} \mathcal{B}$, dado que a_2

tiene $-b$ como efecto y puede ejecutarse entre a_3 y a_5. Nuevamente, las restricciones de orden permiten aplicar cualquiera de los dos métodos para resolver esta amenaza. Las figuras 4.5(e) y (f) muestran los planes que se obtienen al aplicar los métodos *retroceso** y *avance** respectivamente, los cuales son una solución para el problema.

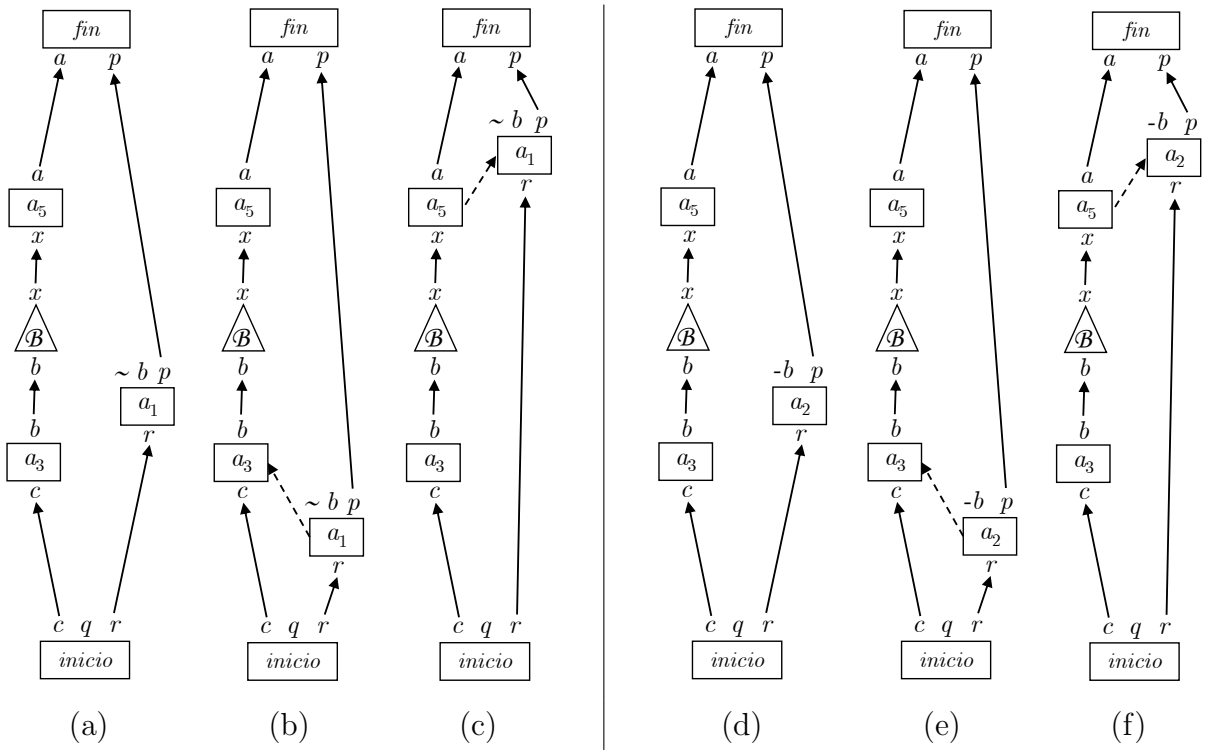

Figura 4.5: Resolviendo *amenazas acción-base* para el ejemplo 4.5

4.3.3. Amenaza acción-argumento

A diferencia de las acciones, los argumentos tienen una estructura interna que puede ser *amenazada* por los efectos de las acciones. Considere la figura 4.6(a). El paso de acción S_k tiene $\overline{N}$ como efecto y el literal N está presente en el argumento asociado a un paso de argumento S_i. A su vez, S_i se utiliza para sustentar una precondición de un paso de acción S_j ($S_j = soportado(S_i)$). Si el efecto $\overline{N}$ del paso S_k *se propaga hasta* S_j (definición 4.12), el argumento asociado a S_i no podrá construirse al momento de ejecutar S_j y por lo tanto éste no podrá ejecutarse.

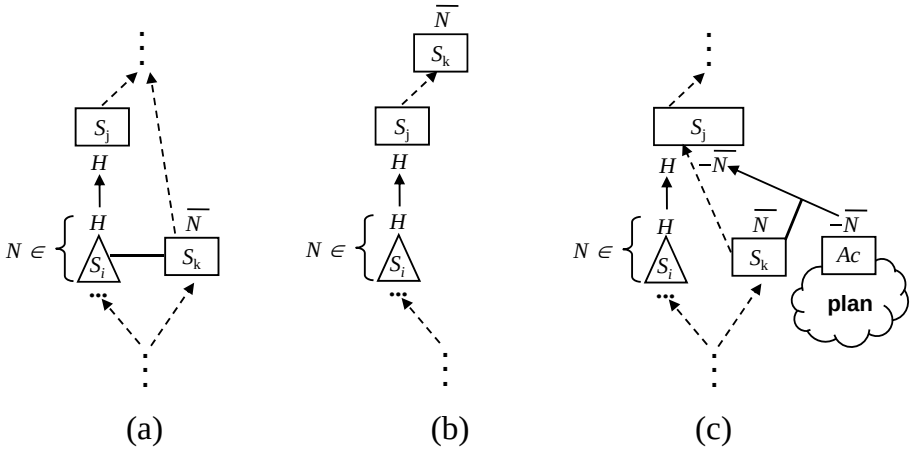

Figura 4.6: **(a)** *amenaza acción-argumento*. La amenaza puede ser resuelta por: **(b)** *avance** ó **(c)** *confrontación por eliminación*.

Para poder identificar y definir este tipo de amenaza, es necesario conocer los efectos que se propagan hasta un determinado paso de acción dentro de un plan. Por este motivo se introduce la siguiente definición.

Definición 4.12 (efectos propagados)
Sea $(\mathcal{AC}, \mathcal{AR}, \mathcal{O}, \mathcal{CL}, \mathcal{SL}, \mathcal{SG})$un PPA y $(S_j, \cdot, \cdot, \cdot)$ un paso de acción en $\mathcal{AC}$. El conjunto de *acciones-efectos propagados* hasta S_j, notado $AEP(S_j)$, se define como sigue:

$$AEP(S_j) = \{(A, e) |\ (A, \cdot, \cdot, Ef_A) \in \mathcal{AC}\ \wedge e \in Ef_A \wedge A \prec S_j \textit{ es consistente con } \mathcal{O} \wedge \nexists\ (D, \_, \_, Ef_D) \in \mathcal{AC} \wedge \{A \prec D, D \prec S_j\} \subset \mathcal{O}^+ \wedge (\overline{e} \in Ef_D \vee e^- \in Ef_D)\}$$

El conjunto de *efectos-propagados* hasta S_j, notado $EP(S_j)$, se define como sigue:

$$EP(S_j) = \{e |\ (A, e) \in AEP(S_j)\}$$

Un efecto de un paso de acción A se propagará hasta un paso de acción S_j dentro de un plan, salvo que otro paso de acción D intermedio $(A \prec D \prec S_j)$ lo elimine o lo niegue. Los efectos propagados hasta un paso de acción S_j dentro de un plan, son aquellos literales que podrían ser verdaderos al momento de ejecutar el paso S_j. Debido a que las restricciones de orden imponen un orden parcial sobre los pasos de un plan, el conjunto de literales que serán verdaderos al momento de ejecutar un paso de acción no es único y dependerá del orden total elegido al ejecutar el plan. Incluso el conjunto de efectos propagados podría ser inconsistente, como ocurre en el siguiente ejemplo.

Ejemplo 4.6 Considere el PPA que se muestra en la figura 4.7(a). Analicemos los efectos de cada paso de acción que puede ejecutarse antes de a_5:

- paso *inicio*
  - El efecto s se propaga hasta a_5 pues ningún paso de acción entre *inicio* y a_5 tiene $-s$ o $\sim s$ como efecto .
  - El efecto r no se propaga hasta a_5 porque el paso a_7 lo elimina, dado que tiene $-r$ como efecto y se ejecuta entre *inicio* y a_5.
  - El efecto q no se propaga hasta a_5 porque el paso a_6 lo niega, dado que tiene $\sim q$ como efecto y se ejecuta entre *inicio* y a_5.
- Todos los efectos de a_7 se propagan hasta a_5 dado que no existe ningún paso de acción que se ejecute entre a_7 y a_5.
- Todos los efectos de a_1 se propagan hasta a_5 dado que no existe ningún paso de acción que se ejecute entre a_1 y a_5.

Luego, el conjunto de *acciones-efectos propagados* hasta el paso a_5 es:

$$AEP(a_5) = \{(inicio, s), (a_7, y), (a_7, \sim c), (a_7, -r), (a_1, \sim b), (a_1, p), (a_1, c), (a_7, \sim q)\}$$

y el conjunto de *efectos propagados* hasta a_5 es:

$$EP(a_5) = \{s, y, \sim c, -r, \sim b, p, c, \sim q\}$$

Note que $EP(a_5)$ es inconsistente pues $\{c, \sim c\} \subset EP(a_5)$

Luego, una *amenaza acción-argumento* se define formalmente como sigue.

Definición 4.13 (amenaza acción-argumento)
Sea $(\mathcal{AC}, \mathcal{AR}, \mathcal{O}, \mathcal{CL}, \mathcal{SL}, \mathcal{SG})$un PPA, sea $(S_i, \mathcal{B}, \cdot)$ un paso de argumento en $\mathcal{AR}$ y sea $S_j = soportado(S_i)$. Sea (S_k, A, Pre_k, Ef_k) un paso de acción en $\mathcal{AC}$ $(S_k \neq S_j)$, se dice que S_k es una *amenaza acción-argumento* para S_i (figura 4.6(a)) si se cumplen las siguientes condiciones:

- $\overline{N} \in Ef_k$ y $N \in cabezas(\mathcal{B})$
- $(S_k, \overline{N}) \in AEP(S_j)$ (el efecto $\overline{N}$ de S_k se propaga hasta S_j)

Para resolver una *amenaza acción-argumento* es necesario impedir que el efecto $\overline{N}$ del paso S_k se propague hasta S_j, esto puede lograrse por uno de los siguientes métodos:

- *avance**: si $S_j \prec S_k$ *es consistente con* $\mathcal{O}$ entonces agregar $S_j \prec S_k$ al conjunto $\mathcal{O}$ (figura 4.6(b)).
- *confrontación por eliminación*: agregar $S_k \prec S_j$ al conjunto $\mathcal{O}$ y agregar $(-\overline{N}, S_j, ac, [\,])$ al conjunto $\mathcal{SG}$ (figura 4.6(c)).

El método *avance** es igual al definido para la *amenaza acción-base* en la sección 4.3.2 y consiste en agregar la restricción de orden $S_j \prec S_k$ al plan para forzar que el paso S_k se ejecute después de S_j. De esta forma el efecto $\overline{N}$ de S_k aparecerá después de S_j y el argumento asociado a S_i podrá construirse al momento de ejecutar S_j.

El método *confrontación por eliminación* consiste en quitar el efecto $\overline{N}$ antes del paso S_j. Para lograr esto se agrega $-\overline{N}$ como una nueva submeta del paso S_j del plan. Esta nueva submeta compromete al planificador a completar el plan para quitar el efecto $\overline{N}$ antes del paso S_j y de esta forma resolver la amenaza. La nueva submeta debe ser de tipo "*ac*", dado que el objetivo es quitar $\overline{N}$ y esto solo puede lograrse por medio de un paso de acción.

Observación 4.2 Note que el literal $N \in cabezas(\mathcal{B})$, luego $N \notin base(\mathcal{B})$ y por lo tanto N no es logrado por un paso de acción del plan. Esto impide aplicar el método *retroceso** (ver definición 4.11 y figura 4.4) para resolver una *amenaza acción-argumento*, ya que no existe un paso de acción en el plan que logre N y que permita agregar una restricción de orden para impedir que el efecto $\overline{N}$ del paso S_k se propague hasta S_j.

Ejemplo 4.7 Sea $(\Psi_{4.7}, \Delta_{4.7}, \Gamma_{4.7}, Meta_{4.7}, \mathsf{C}_{4.7})$ un problema de planificación donde: $\Psi_{4.7} = \{q, r, s\}$, $\Delta_{4.7} = \{(x \prec b), (b \prec y)\}$, $Meta_{4.7} = \{a, p\}$, $\mathsf{C}_{4.7} = \emptyset$ y

$$\Gamma_{4.7} = \left\{ \begin{array}{cc} \{b\}, \{\}^- \xleftarrow{a_3} \{c\}, not\ \{\}, & \{a\}, \{\}^- \xleftarrow{a_5} \{x\}, not\ \{\}, \\ \{p, \sim b, c, \sim q\}, \{\}^- \xleftarrow{a_6} \{q\}, not\ \{\}, & \{y, \sim c\}, \{r\}^- \xleftarrow{a_7} \{r\}, not\ \{\}, \\ \{\}, \{\sim b\}^- \xleftarrow{a_8} \{s\}, not\ \{\} & \end{array} \right\}$$

La figura 4.7(a) muestra un plan posible al que puede arribar el planificador durante la búsqueda de una solución para el problema 4.7. El paso de argumento $\mathcal{B} = \{(x \prec b), (b \prec y)\}$ sustenta la precondición x del paso de acción a_5. Como puede observarse el paso de acción a_6 es una *amenaza acción-argumento* para el paso de argumento

$\mathcal{B}$, dado que el efecto $\sim b$ de a_6 se propaga hasta el paso a_5 (ver ejemplo 4.6) impidiendo que el argumento $\mathcal{B}$ pueda construirse al momento de ejecutar a_5.

La figura 4.7(b) muestra el plan que se obtienen al aplicar el método *avance**, el cual soluciona el problema. El paso de acción a_6 es forzado a ejecutarse después del paso de acción a_5 agregando la restricción de orden $a_5 \prec a_6$ al plan. De esta forma el efecto $\sim b$ de a_6 aparecerá después de a_5 y el argumento $\mathcal{B}$ podrá construirse al momento de ejecutar S_5.

La figura 4.7(c) muestra el plan que se obtiene al aplicar el método *confrontación por eliminación*. Este método agrega la restricción de orden $a_6 \prec a_5$ y la submeta $(-\sim b, a_5, ac, [\,])$ al plan, la cual debe ser lograda por un paso de acción por ser de tipo "*ac*". La única acción disponible que tiene $-\sim b$ como efecto es la acción a_8, por lo tanto el planificador agrega el paso de acción a_8 junto con el vínculo causal $a_8 \xrightarrow{-\sim b} a_5$. La acción a_6 es una *amenaza acción-acción* para $a_8 \xrightarrow{-\sim b} a_5$ por lo tanto el planificador aplica el método *retroceso* para resolverla agregando la restricción de orden $a_6 \prec a_8$. Note que el método *avance* no puede aplicarse para resolver esta amenaza porque $a_6 \prec a_5$ pertenece a las restricciones de orden del plan. La precondición s del paso a_6 es un efecto del paso inicio, por lo tanto el planificador agrega el vínculo causal $inicio \xrightarrow{s} a_6$ para lograrla y obtiene un plan que soluciona el problema.

Observación 4.3 Como puede observarse en la figura 4.6(c) al aplicar el método *confrontación por eliminación* se introduce una *amenaza acción-acción* entre el paso S_k y el vínculo causal $Ac \xrightarrow{-\overline{N}} S_j$ que se agrega para lograr la nueva submeta. Esta amenaza será resuelta por el planificador al completar el plan. Si esta amenaza es resuelta por el método *avance* el planificador agregará la restricción $S_j \prec S_k$ al igual que el método *avance** y los pasos agregados para lograr la submeta $-\overline{N}$ serán redundantes. Por este motivo el método *confrontación por eliminación* agrega la restricción de orden $S_k \prec S_j$ la cual impide que el planificador aplique el método *avance* para resolver la *amenaza acción-acción* y deba aplicar el método *retroceso*.

Observación 4.4 Agregar N como una nueva submeta del paso S_j también es una alternativa para solucionar la *amenaza acción-argumento*, pero también tiene el efecto de alterar el argumento $\mathcal{B}$ asociado a S_i, dado que $N \in cabezas(\mathcal{B})$. Como se muestra el ejemplo 4.7 y las figuras 4.8(b) y 4.8(c) es preferible que el planificador evite esta amena-

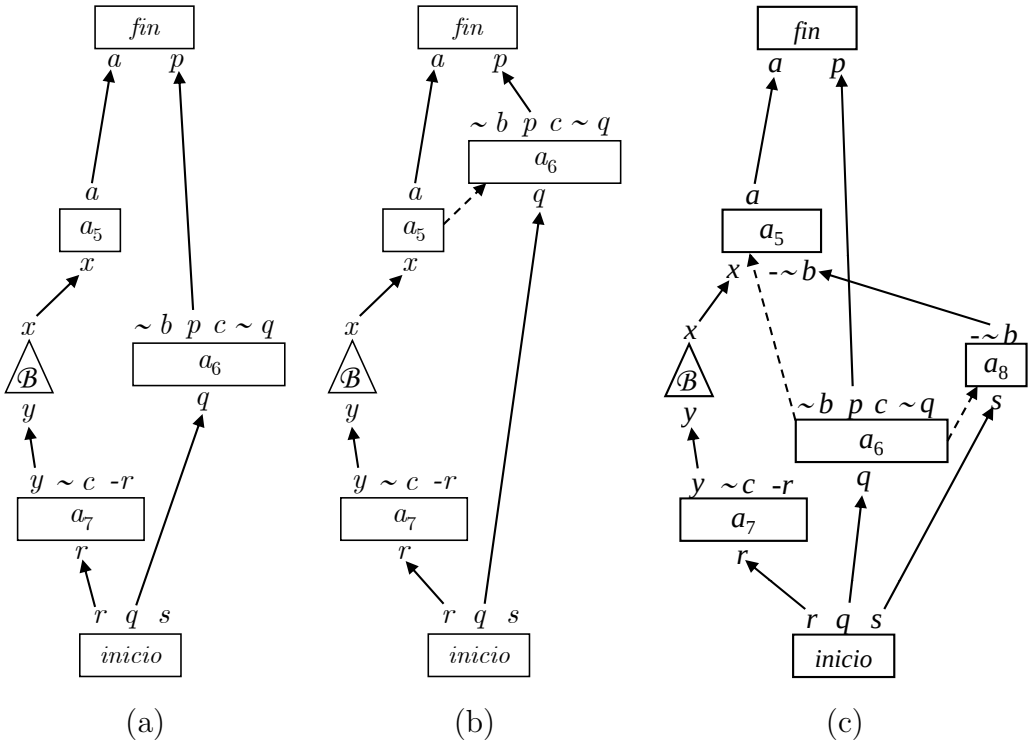

Figura 4.7: Resolviendo *amenazas acción-argumento* para el ejemplo 4.7

za desechando el plan y buscando (por *backtracking*) otra alternativa para el argumento asociado al paso S_i.

La figura 4.8(a) muestra el plan que se obtendría si se aplicara el método *avance* para resolver la amenaza *acción-acción* introducida al aplicar el método *confrontación por eliminación* (observación 4.3) para el ejemplo 4.7. Como puede observarse el paso de acción a_8 agregado para lograr la submeta $(-\sim b, a_5, ac, [\,])$ es totalmente irrelevante dado que la amenaza *acción-argumento* es resuelta al incorporar la restricción de orden $a_5 \prec a_6$. Esto resulta evidente al comparar el plan de la figura 4.8(a) con los planes de las figuras 4.7(b) y 4.7(c). Como puede observarse el plan de la figura 4.8(a) contiene ambos planes de las figuras 4.7(b) y 4.7(c).

La figura 4.8(b) muestra el plan que se obtendría si se agregara la submeta $(b, a_5, ac, [])$ al plan de la figura 4.7(a) para resolver la *amenaza acción-argumento* (observación 4.4). El paso de acción a_3 se agrega para lograr la submeta b del paso a_5 y esto evita que el efecto $\sim b$ se propague hasta a_5 lo cual permite que el argumento $\mathcal{B} = \{(x \prec b), (b \prec y)\}$ pueda construirse al momento de ejecutar a_5. Sin embargo, lograr la submeta b del paso

a_5 también permite construir el subargumento $\mathcal{D} = \{x \prec b\}$ de $\mathcal{B}$ al momento de ejecutar a_5. Luego, el paso de argumento $\mathcal{B}$ y el paso de acción a_7 son irrelevantes para el plan dado que la precondición x del paso a_5 es sustentada por $\mathcal{D}$. La figura 4.8(c) muestra el plan que obtiene el planificador si desecha el plan de la figura 4.7(a) y utiliza $\mathcal{D}$ como otra alternativa para sustentar la precondición x del paso a_5.

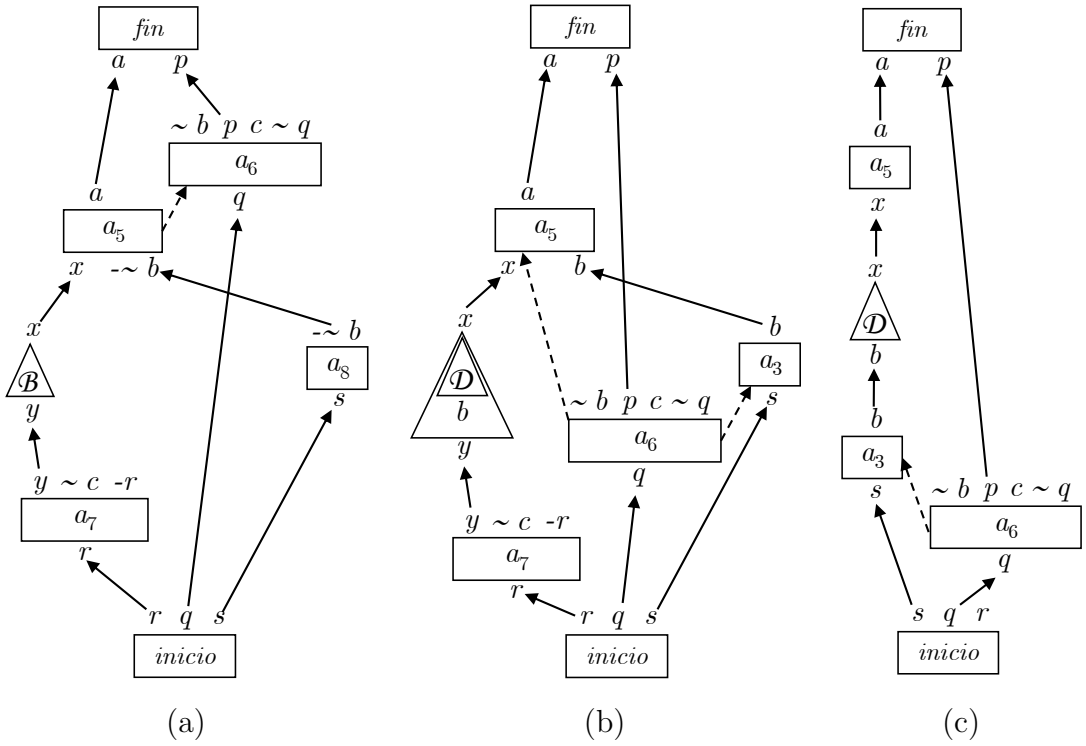

Figura 4.8: observaciones sobre la resolución de *amenazas acción-argumento* para el ejemplo 4.7

4.3.4. Amenaza acción-suposición

La figura 4.9(a) muestra otro tipo de amenaza que involucra acciones y argumentos. En este caso un paso de acción S_k, que tiene N como efecto, invalida la suposición *not* N del argumento asociado a un paso de argumento S_i. A su vez, S_i se utiliza para sustentar una precondición de un paso de acción S_j. Si el efecto N del paso S_k se propaga hasta S_j el argumento asociado a S_i no podrá construirse y S_j no podrá ejecutarse. Este tipo de amenaza recibirá el nombre de *amenaza acción-suposición* y se define formalmente a continuación.

Definición 4.14 (amenaza acción-suposición)
Sea $(\mathcal{AC}, \mathcal{AR}, \mathcal{O}, \mathcal{CL}, \mathcal{SL}, \mathcal{SG})$ un PPA sea $(S_i, \mathcal{B}, \cdot)$ un paso de argumento en $\mathcal{AR}$ y sea $S_j = soportado(S_i)$. Sea $(S_k, \cdot, \cdot, Ef_k)$ un paso de acción en $\mathcal{AC}$ $(S_k \neq S_j)$, se dice que S_k es una *amenaza acción-suposición* para S_i, si se cumplen las siguientes condiciones:

- $N \in Ef_k$ y $not\ N \in suposiciones(\mathcal{B})$
- $(S_k, N) \in AEP(S_j)$ (el efecto N de S_k se propaga hasta S_j).

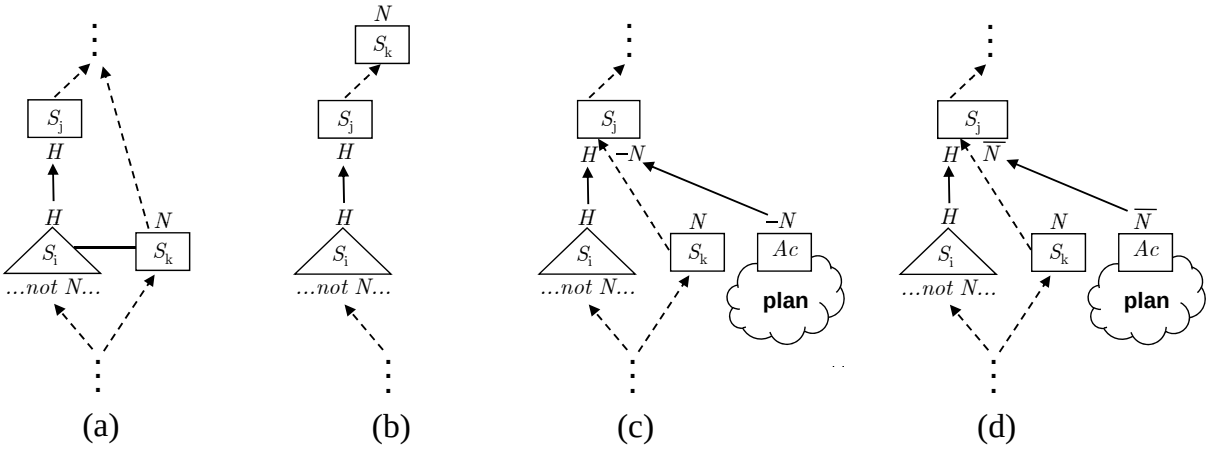

Figura 4.9: **(a)** *amenaza acción-suposición*. La amenaza puede ser resuelta por:**(b)** *avance**, **(c)** *confrontación por complemento* ó **(d)***confrontación por eliminación*.

Para resolver una *amenaza acción-suposición* es necesario impedir que el efecto N del paso S_k se propague hasta S_j, esto puede lograrse por uno de los siguientes métodos:

- *avance**: si $S_j \prec S_k$ *es consistente con* $\mathcal{O}$ agregar $S_j \prec S_k$ al conjunto $\mathcal{O}$ (figura 4.9(b)).
- *confrontación por complemento*: agregar $S_k \prec S_j$ al conjunto $\mathcal{O}$ y agregar $(\overline{N}, S_j, ac, [\,])$ al conjunto $\mathcal{SG}$ (figura 4.9(c)).
- *confrontación por eliminación*: agregar $S_k \prec S_j$ al conjunto $\mathcal{O}$ y agregar $(-N, S_j, ac, [\,])$ al conjunto $\mathcal{SG}$ (figura 4.9(d)).

Los métodos *avance** y *confrontación por eliminación* son iguales a los definidos para la *amenaza acción-suposición* en la sección 4.3.3.

El método *avance** consiste en agregar la restricción de orden $S_j \prec S_k$ al plan para forzar que el paso S_k se ejecute después de S_j. De esta forma el efecto N de S_k aparecerá después de S_j y el argumento asociado a S_i podrá construirse al momento de ejecutar S_j.

El método *confrontación por eliminación* consiste en quitar el efecto N antes del paso S_j. Para lograr esto se agrega $-N$ como una nueva submeta de tipo "*ac*" del paso S_j del plan. Esta nueva submeta compromete al planificador a completar el plan para quitar el efecto N antes del paso S_j y de esta forma resolver la amenaza. La submeta debe ser de tipo "*ac*", dado que el objetivo es quitar N y esto solo puede lograrse por medio de un paso de acción.

El método *confrontación por complemento* consiste en negar el efecto N antes del paso S_j. Para lograr esto se agrega $\overline{N}$ como una nueva submeta de tipo "*ac*" del paso S_j del plan. Esta nueva submeta compromete al planificador a completar el plan para negar el efecto N y de esta forma resolver la amenaza.

Ejemplo 4.8 Sea $(\Psi_{4.8}, \Delta_{4.8}, \Gamma_{4.8}, Meta_{4.8}, \mathsf{C}_{4.8})$ un problema de planificación donde: $\Psi_{4.8} = \{q, r, s\}$, $\Delta_{4.8} = \{x \prec not\ b\}$, $Meta_{4.8} = \{a, c\}$, $\mathsf{C}_{4.8} = \emptyset$ y

$$\Gamma_{4.8} = \left\{ \begin{array}{ll} \{p, \sim b\}, \{\}^- \xleftarrow{a_1} \{q\}, not\ \{\}, & \{p\}, \{b\}^- \xleftarrow{a_2} \{r\}, not\ \{\}, \\ \{a\}, \{\}^- \xleftarrow{a_5} \{x\}, not\ \{\}, & \{b, c\}, \{\}^- \xleftarrow{a_9} \{s\}, not\ \{\}, \end{array} \right\}$$

La figura 4.10(a) muestra un plan posible al que puede arribar el planificador durante la búsqueda de una solución para el problema 4.8. El paso de argumento $\mathcal{B} = \{x \prec not\ b\}$ sustenta la precondición x del paso de acción a_5. Como puede observarse el paso de acción a_9 es una *amenaza acción-suposición* para el paso de argumento $\mathcal{B}$, dado que el efecto b de a_6 se propaga hasta el paso a_5 impidiendo que el argumento $\mathcal{B}$ pueda construirse al momento de ejecutar a_5.

La figura 4.10(b) muestra el plan que se obtiene al aplicar el método *avance**, el cual soluciona el problema. El paso de acción a_9 es forzado a ejecutarse después del paso de acción a_5 incorporando la restricción de orden $a_5 \prec a_9$ al plan. De este forma el efecto b de a_9 aparecerá después de a_5 y el argumento $\mathcal{B}$ podrá construirse al momento de ejecutar S_5.

La figura 4.10(c) muestra el plan que se obtiene al aplicar el método *confrontación por eliminación*. Este método agrega la restricción de orden $a_9 \prec a_5$ y la submeta

$(-b, a_5, ac, [\,])$ al plan, la cual debe ser lograda por un paso de acción por ser de tipo "*ac*". La única acción disponible que tiene $-b$ como efecto es la acción a_2, por lo tanto el planificador agrega el paso de acción a_2 junto con el vínculo causal $a_2 \xrightarrow{-b} a_5$. La acción a_9 es una *amenaza acción-acción* para $a_2 \xrightarrow{-b} a_5$ por lo tanto el planificador aplica el método *retroceso* para resolverla agregando la restricción de orden $a_9 \prec a_2$. La precondición q del paso a_2 es un efecto del paso *inicio*, por lo tanto el planificador agrega el vínculo causal $inicio \xrightarrow{q} a_2$ para lograrla y obtiene un plan que soluciona el problema.

La figura 4.10(d) muestra el plan que se obtiene al aplicar el método *confrontación por complemento*. Este método agrega la restricción de orden $a_9 \prec a_5$ y la submeta $(\sim b, a_5, ac, [\,])$ al plan, la cual debe ser lograda por un paso de acción por ser de tipo "*ac*". La única acción disponible que tiene $\sim b$ como efecto es la acción a_1, por lo tanto el planificador agrega el paso de acción a_1 junto con el vínculo causal $a_1 \xrightarrow{\sim b} a_5$. La acción a_9 es una *amenaza acción-acción* para $a_1 \xrightarrow{\sim b} a_5$ por lo tanto el planificador aplica el método *retroceso* para resolverla agregando la restricción de orden $a_9 \prec a_1$. La precondición r del paso a_1 es un efecto del paso *inicio*, por lo tanto el planificador agrega el vínculo causal $inicio \xrightarrow{r} a_1$ para lograrla y obtiene un plan que soluciona el problema.

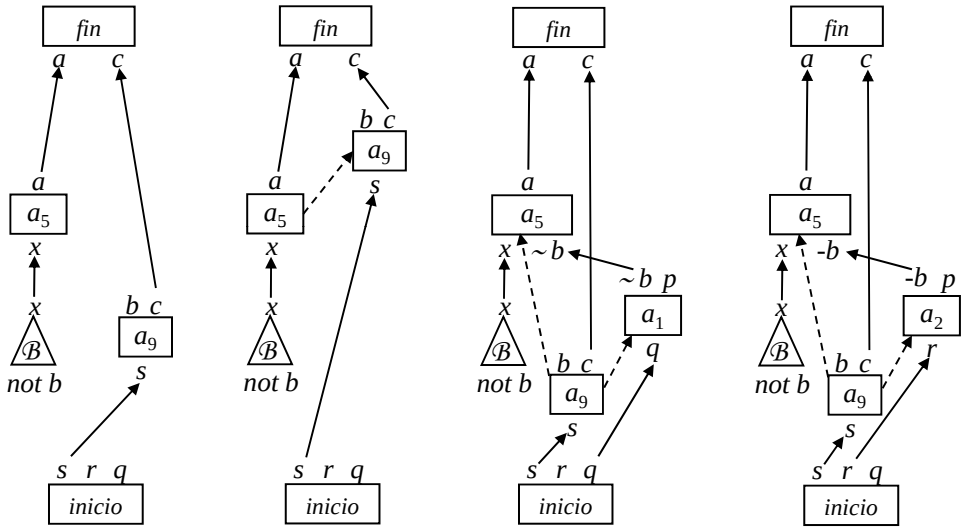

Figura 4.10: Resolviendo una *amenaza acción-suposición* para el ejemplo 4.8

4.3.5. Amenaza argumento-argumento

Para poder ejecutar una acción es necesario que todas sus precondiciones estén garantizadas. La incorporación de un paso de argumento a un plan para soportar la precondición de un paso de acción no es suficiente para garantizar dicha precondición, dado que el argumento asociado al paso de argumento podría estar derrotado por otro argumento.

La figura 4.11(a) muestra una situación de este tipo. Un paso de argumento S_i soporta la precondición H de un paso de acción S_j. Por otra parte, existe un *argumento potencial* $\mathcal{C}$ que es un derrotador para el argumento asociado a S_i. Si $\mathcal{C}$ derrota al argumento asociado a S_i antes de la ejecución de S_j, la precondición H no estará garantizada y S_j no podrá ejecutarse. Este tipo de amenaza recibirá el nombre de *amenaza argumento-argumento*.

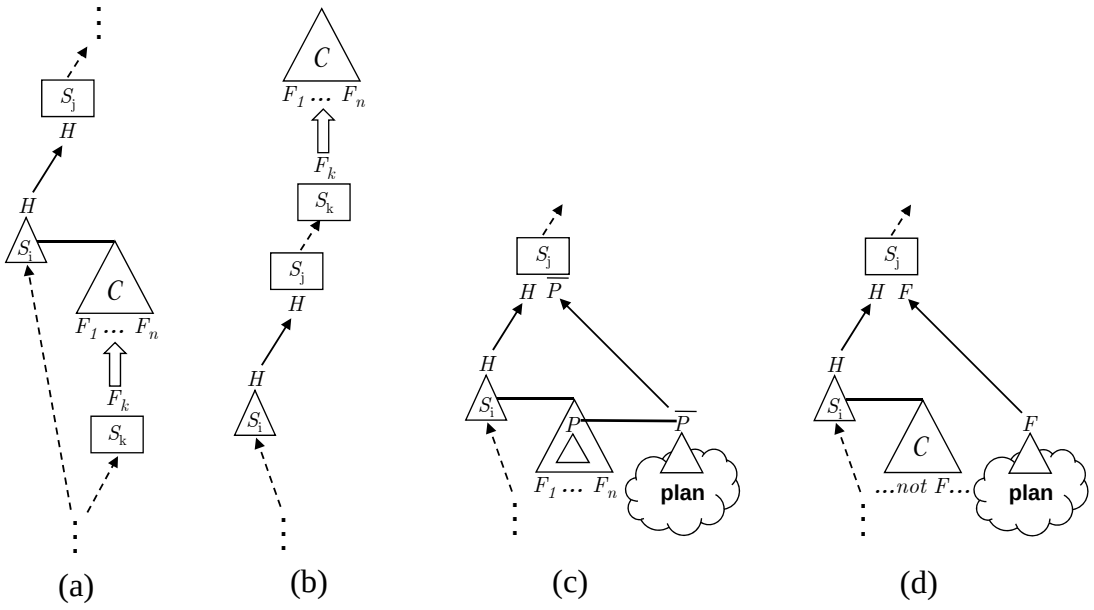

Figura 4.11: **(a)** *amenaza argumento-argumento*. La amenaza puede ser resuelta por: **(b)** *avanzar el derrotador*, **(c)** *atacar un punto interno* ó **(d)** *atacar una suposición*.

Es importante notar que el argumento $\mathcal{C}$ no necesariamente forma parte de un paso de argumento del plan, sino que puede ser cualquier argumento potencial construido a partir de Δ que *podría activarse antes* de S_j (ver definición 4.15).

Definición 4.15 (activación de argumentos potenciales en un plan)
Sea $(\mathcal{AC}, \mathcal{AR}, \mathcal{O}, \mathcal{CL}, \mathcal{SL}, \mathcal{SG})$ un PPA, $S_j \in \mathcal{AC}$ un paso de acción y $\mathcal{C} \subseteq \Delta$ un *argumento potencial* para Q. Se dice que $\mathcal{C}$ *podría activarse antes de* S_j si:

- $base(\mathcal{C}) \subseteq EP(S_j)$ y
- $\nexists\, not\, X \in suposiciones(\mathcal{C})$ tal que $X \in EP(S_j)$, $\overline{X} \notin EP(S_j)$ y $-X \notin EP(S_j)$

Luego, una *amenaza argumento-argumento* se define formalmente como sigue.

Definición 4.16 (amenaza argumento-argumento)
Sea $(\mathcal{AC}, \mathcal{AR}, \mathcal{O}, \mathcal{CL}, \mathcal{SL}, \mathcal{SG})$ un PPA, sea $(S_i, \mathcal{B}, \Lambda_{\mathcal{B}}) \in \mathcal{AR}$ un paso de argumento, donde $\Lambda_{\mathcal{B}} = [\langle \mathcal{A}_1, L_1\rangle, \ldots, \langle \mathcal{A}_n, L_n\rangle, \langle \mathcal{B}, H\rangle]$ y sea $S_j = soportado(S_i)$. Sea $\mathcal{C} \subseteq \Delta$ un *argumento potencial* para Q y sea (Ψ, Δ) un programa DeLP, donde:

$$\Psi = base(\mathcal{A}_1) \cup \cdots \cup base(\mathcal{A}_n) \cup base(\mathcal{B}) \cup base(\mathcal{C})$$

Se dice que $\mathcal{C}$ es una *amenaza argumento-argumento* para S_i, si se cumplen las siguientes condiciones:

- $\mathcal{C}$ es un derrotador para $\mathcal{B}$ en (Ψ, Δ) y $\mathcal{C}$ *podría activarse antes de* S_j,
- $[\langle \mathcal{A}_1, L_1\rangle, \ldots, \langle \mathcal{A}_n, L_n\rangle, \langle \mathcal{B}, H\rangle, \langle \mathcal{C}, Q\rangle]$ *es una línea de argumentación aceptable* en (Π, Δ),
- No existe $S \overset{R}{\succ\!\!-} S_j \in \mathcal{CL}$, tal que $(S, \mathcal{D}, \Lambda_{\mathcal{D}}) \in \mathcal{AR}$, $\mathcal{D}$ es un derrotador para $\mathcal{C}$ y $\Lambda_{\mathcal{D}} = [\langle \mathcal{A}_1, L_1\rangle, \ldots, \langle \mathcal{A}_n, L_n\rangle, \langle \mathcal{B}, H\rangle, \langle \mathcal{C}, Q\rangle, \langle \mathcal{D}, R\rangle]$ *es una línea de argumentación aceptable* (esto es, $\mathcal{C}$ no es derrotado por otro paso de argumento del plan)

Para asegurar que el argumento $\mathcal{B}$ asociado a S_i no esté derrotado es necesario construir un *árbol de dialéctica* que tiene $\mathcal{B}$ por raíz y donde todos los hijos (derrotadores) de $\mathcal{B}$ estén derrotados (marcados con "D"). Este árbol no puede construirse completamente al momento de incorporar S_i al plan, porque en ese momento es imposible conocer todos los derrotadores de $\mathcal{B}$ ya que dependen de los pasos de acción que se agreguen más adelante en el proceso de planificación. Por este motivo el árbol de dialéctica para $\mathcal{B}$ debe ser construido gradualmente a medida que se construye el plan.

A continuación se definen los métodos para resolver una *amenaza argumento-argumento*:

- *avanzar el derrotador*: Sea $F_k \in base(\mathcal{C})$. Para cada $(S_k, F_k) \in AEP(S_j)$, agregar $S_j \prec S_k$ al conjunto $\mathcal{O}$ (figura 4.11(b)).

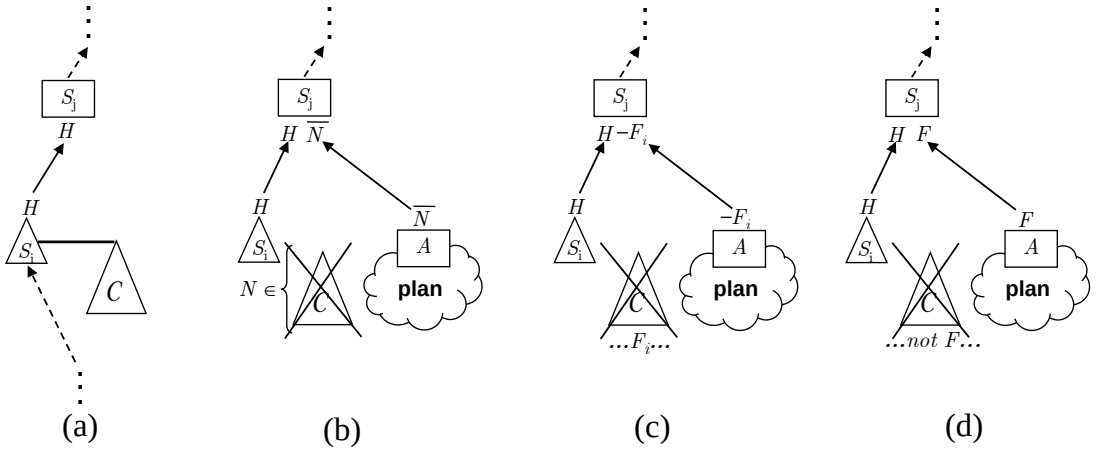

Figura 4.12: **(a)** *amenaza argumento-argumento*. La amenaza puede ser resuelta por: **(b)** *deshabilitar cualquier literal*, **(c)** *deshabilitar un literal de la base* ó **(d)** *deshabilitar una suposición*.

- *atacar un punto interno*: agregar $(\overline{P}, S_j, arg, [\langle \mathcal{A}_1, L_1 \rangle, \ldots, \langle \mathcal{A}_n, L_n \rangle, \langle \mathcal{B}, H \rangle, \langle \mathcal{C}, Q \rangle])$ al conjunto $\mathcal{SG}$, donde $P \in cabezas(\mathcal{C})$ (figura 4.11(c)).
- *atacar una suposición*: agregar $(F, S_j, arg, [\langle \mathcal{A}_1, L_1 \rangle, \ldots, \langle \mathcal{A}_n, L_n \rangle, \langle \mathcal{B}, H \rangle, \langle \mathcal{C}, Q \rangle])$ al conjunto $\mathcal{SG}$, donde *not* $F \in suposiciones(\mathcal{C})$ (figura 4.11(d)).
- *deshabilitar cualquier literal*: agregar $(\overline{N}, S_j, ac, [\,])$ al conjunto $\mathcal{SG}$, donde $N \in literales(\mathcal{C})$ (figura 4.12(b)).
- *deshabilitar un literal en la base*: agregar $(-F_i, S_j, ac, [\,])$ al conjunto $\mathcal{SG}$, donde $F_i \in base(\mathcal{C})$ (figura 4.12(c)).
- *deshabilitar una suposición*: agregar $(F, S_j, ac, [\,])$ al conjunto $\mathcal{SG}$, donde *not* $F \in suposiciones(\mathcal{C})$ (figura 4.12(d)).

El método *avanzar el derrotador* consiste en agregar una restricción de orden $S_j \prec S_k$ para cada paso de acción S_k que logra uno de los literales F_k en la base del derrotador $\mathcal{C}$. De esta forma $\mathcal{B}$ no será derrotado por $\mathcal{C}$ al momento de ejecutar S_j, porque $\mathcal{C}$ no podrá construirse hasta después de ejecutar S_j.

Los métodos *atacar un punto interno* y *atacar una suposición* consisten en introducir un nuevo paso de argumento para derrotar el derrotador $\mathcal{C}$ antes de ejecutar S_j atacando un punto interno P o una suposición *not* F de $\mathcal{C}$ respectivamente. Para lograr esto se introduce una nueva submeta del paso S_j, la cual debe ser de tipo "arg" para comprometer al planificador a lograrla por medio de un paso de argumento.

El método *deshabilitar cualquier literal* introduce $\overline{P}$ como una submeta de tipo "*arg*" del paso S_j, donde $P \in cabezas(\mathcal{C})$. Luego, el paso de argumento que se agregue para lograr $\overline{p}$ derrotará a $\mathcal{C}$ en el punto P antes de la ejecución de S_j.

La línea de argumentación $[\langle \mathcal{A}_1, L_1\rangle, \ldots, \langle \mathcal{A}_n, L_n\rangle, \langle \mathcal{B}, H\rangle, \langle \mathcal{C}, Q\rangle]$ es incluida en la submeta para que el planificador elija, para lograr $\overline{P}$, un argumento $\mathcal{D}_1$ tal que $[\langle \mathcal{A}_1, L_1\rangle, \ldots, \langle \mathcal{A}_n, L_n\rangle, \langle \mathcal{B}, H\rangle, \langle \mathcal{C}, Q\rangle, \langle \mathcal{D}_1, \overline{P}\rangle]$ constituya una *línea de argumentación aceptable*. De no ser así, el argumento elegido $\mathcal{D}_1$ invalidará la garantía de L_1.

El método *atacar una suposición* introduce F como una nueva submeta de tipo "*arg*" del paso S_j, donde *not* $F \in suposiciones(\mathcal{C})$. En este caso, el paso de argumento que se agregue para lograr F derrotará a $\mathcal{C}$ en la suposición *not* F antes de la ejecución de S_j.

Los métodos *deshabilitar cualquier literal*, *deshabilitar un literal en la base* y *deshabilitar una suposición* consisten en introducir un paso de acción al plan para quitar o agregar un literal y de esta forma evitar que el derrotador $\mathcal{C}$ pueda construirse en el momento de ejecutar S_j. Esto se logra introduciendo una nueva submeta del paso S_j, la cual debe ser de tipo "*ac*" para comprometer al planificador a lograrla utilizando un paso de acción.

El método *deshabilitar cualquier literal* introduce $\overline{N}$ como una submeta del paso S_j, donde $N \in literales(\mathcal{C})$. Luego, el paso de acción que se agregue para lograr $\overline{N}$ impedirá que el derrotador $\mathcal{C}$ pueda construirse al momento de ejecutar S_j.

El método *deshabilitar un literal de la base* introduce $-F_i$ como una nueva submeta del paso S_j, donde $F_i \in base(\mathcal{C})$. En este caso, el paso de acción que se agregue para lograr $-F_i$ impedirá que $\mathcal{C}$ pueda construirse al momento de ejecutar S_j, porque eliminará el literal F_i presente en la base de $\mathcal{C}$.

Por último, el método *deshabilitar una suposición* introduce F como una nueva submeta del paso S_j, donde *not* $F \in suposiciones(\mathcal{C})$. En este caso, el paso de acción que se agregue para lograr F impedirá que $\mathcal{C}$ pueda construirse al momento de ejecutar S_j, porque invalidará la suposición *not* F.

A continuación se presenta un ejemplo que muestra como se resuelve una *amenaza argumento-argumento* aplicando cada uno de los métodos definidos.

Ejemplo 4.9 Sea $(\Psi_{4.9}, \Delta_{4.9}, \Gamma_{4.9}, Meta_{4.9}, \mathsf{C}_{4.9})$ un problema de planificación donde: $\Psi_{4.9} = \{q, r, s, z\}$, $Meta_{4.9} = \{a, c\}$, $\mathsf{C}_{4.9} = \emptyset$,

$$\Delta_{4.9} = \left\{ \begin{array}{c} (x -\!\prec y),\ (y -\!\prec z), \\ (\sim y -\!\prec d),\ (d -\!\prec b, not\ e), \\ (\sim d -\!\prec b, not\ e, f),\ (e -\!\prec g) \end{array} \right\}$$

$$\Gamma_{4.9} = \left\{ \begin{array}{ll} \{p\}, \{b\}^- \xleftarrow{a_2} \{r\}, not\ \{\}, & \{a\}, \{\}^- \xleftarrow{a_5} \{x\}, not\ \{\}, \\ \{b, c\}, \{\}^- \xleftarrow{a_9} \{s\}, not\ \{\}, & \{f\}, \{\}^- \xleftarrow{a_{10}} \{q\}, not\ \{\}, \\ \{g\}, \{\}^- \xleftarrow{a_{11}} \{q\}, not\ \{\}, & \{\sim d\}, \{\}^- \xleftarrow{a_{12}} \{r\}, not\ \{\} \\ \{e\}, \{\}^- \xleftarrow{a_{13}} \{r\}, not\ \{\} & \end{array} \right\}$$

La figura 4.13(a) muestra un plan posible al que puede arribar el planificador durante la búsqueda de una solución para el problema 4.9. El paso de argumento $(S_{arg}, \mathcal{B}, [\langle \mathcal{B}, x \rangle])$ (etiquetado con $\mathcal{B}$ en la figura) donde $\mathcal{B}$=$\{(x -\!\prec y),\ (y -\!\prec z)\}$, se agregó al plan para sustentar la precondición x del paso de acción a_5. Por otra parte, el argumento potencial $\mathcal{C}$=$\{(\sim y -\!\prec d),\ (d -\!\prec b, not\ e)\} \subseteq \Delta$ es una *amenaza argumento-argumento* para S_{arg} ya que:

- $\mathcal{C}$ es un derrotador para $\mathcal{B}$ en $(\Pi_{4.9}, \Delta_{4.9})$, donde $\Pi_{4.9} = base(\mathcal{B}) \cup base(\mathcal{C}) = \{z, b\}$
- $[\langle \mathcal{B}, x \rangle, \langle \mathcal{C}, \sim y \rangle]$ *es una línea de argumentación aceptable* en $(\Pi_{4.9}, \Delta_{4.9})$
- $\mathcal{C}$ *podría activarse antes de* a_5 ya que:
  - $base(\mathcal{C}) = \{b\} \subseteq EP(a_5) = \{b, c, s, q, r, z\}$.
  - $suposiciones(\mathcal{C}) = \{not\ e\}$ y $e \notin EP(a_5) = \{b, c, s, q, r, z\}$.
- $\mathcal{C}$ no es derrotado por otro paso de argumento distinto de S_{arg}

La figura 4.13(b) muestra el plan que se obtiene al aplicar el método *avanzar el derrotador*. Note que el paso de acción a_9 es el único paso que logra el literal $b \in base(\mathcal{C})$. Luego, el paso de acción a_9 es forzado a ejecutarse después del paso de acción a_5 incorporando la restricción de orden $a_5 \prec a_9$ al plan. De esta forma $\mathcal{B}$ no será derrotado por $\mathcal{C}$ al momento de ejecutar a_5 porque $\mathcal{C}$ no podrá construirse hasta después de ejecutar a_9.

La figura 4.13(c) muestra el plan que se obtiene al aplicar el método *atacar un punto interno*. En este caso el punto elegido para atacar es el literal $d \in cabezas(\mathcal{C}) = \{d, \sim y\}$, por lo tanto se agrega la submeta $(\sim d, a_5, arg, [\langle \mathcal{B}, x \rangle, \langle \mathcal{C}, \sim y \rangle])$ al plan. Esta submeta debe ser lograda por un paso de argumento por ser de tipo "arg". A partir de $\Delta_{4.9}$ se puede construir un único argumento potencial $\mathcal{D}_1$=$\{(\sim d -\!\prec b, not\ e, f)\}$ para $\sim d$, donde

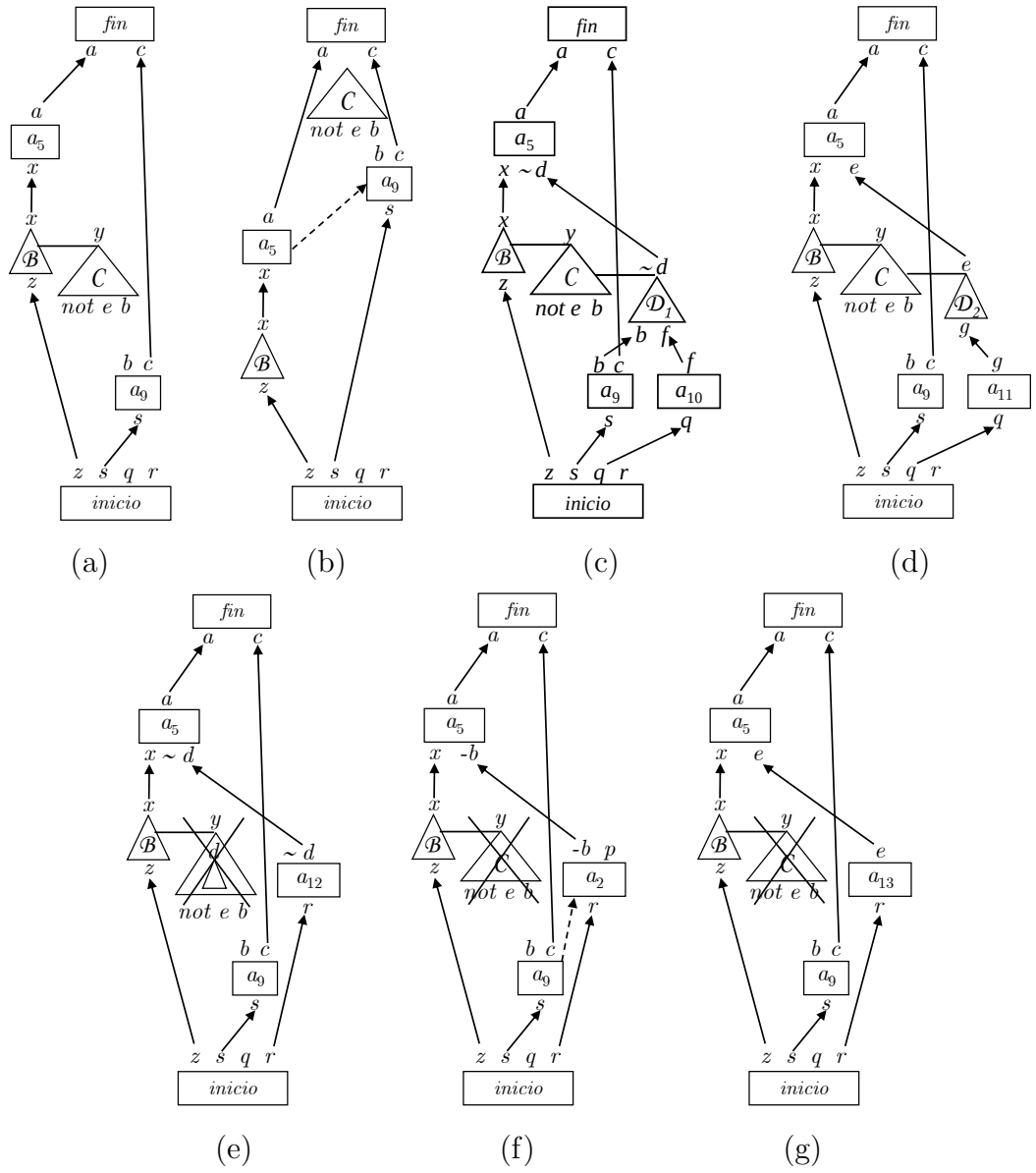

Figura 4.13: Resolviendo *amenazas argumento-argumento* para el ejemplo 4.9

$[\langle \mathcal{B}, x\rangle, \langle \mathcal{C}, \sim y\rangle, \langle \mathcal{D}_1, \sim d\rangle]$ es un línea de argumentación aceptable. Luego, el planificador agrega un nuevo paso de argumento $(S_1, \mathcal{D}_1, [\langle \mathcal{B}, x\rangle, \langle \mathcal{C}, \sim y\rangle, \langle \mathcal{D}_1, \sim d\rangle])$ junto con el vínculo de soporte $S_1 \overset{\sim d}{\succ\!\!-} a_5$. De esta forma, el argumento $\mathcal{D}_1$ derrota a $\mathcal{C}$ antes de la ejecución de a_5 resolviendo la amenaza. Los literales presentes en $base(\mathcal{D}_1) = \{b, f\}$ se convierten en nuevas submetas del plan que deben ser logradas por pasos de acción. Dado que el literal d es un efecto del paso a_9, simplemente se agrega el vínculo causal $a_9 \xrightarrow{b} S_1$. Para lograr f, el planificador agrega un nuevo paso de acción a_{10} y el vínculo causal $a_{10} \xrightarrow{f} S_1$. La

precondición q del paso a_{10} es un efecto del paso *inicio*, por lo tanto el planificador agrega el vínculo causal $inicio \xrightarrow{q} a_{10}$ para lograrla y obtiene un plan que soluciona el problema.

La figura 4.13(d) muestra el plan que se obtiene al aplicar el método *atacar una suposición*. Existe una única suposición $not\, e \in ensuposiciones(\mathcal{C})$ por lo tanto se agrega la submeta $(e, a_5, arg, [\langle \mathcal{B}, x\rangle, \langle \mathcal{C}, \sim y\rangle])$ al plan. A partir de $\Delta_{4.9}$ se puede construir un único argumento potencial $\mathcal{D}_2$=$\{(e \prec g)\}$ para e, donde $[\langle \mathcal{B}, x\rangle, \langle \mathcal{C}, \sim y\rangle, \langle \mathcal{D}_2, \sim d\rangle]$ es una línea de argumentación aceptable. Por lo tanto, el planificador agrega un nuevo paso de argumento $(S_2, \mathcal{D}_2, [\langle \mathcal{B}, x\rangle, \langle \mathcal{C}, \sim y\rangle, \langle \mathcal{D}_2, e\rangle])$ junto con el vínculo de soporte $S_1 \overset{e}{\succ} a_5$. Luego, para lograr el literal $g \in base(\mathcal{D}_2) = g$, el planificador agrega el paso de acción a_{11} junto con el vínculo causal $a_{11} \xrightarrow{g} S_2$. La precondición q del paso a_{11} es un efecto del paso *inicio*, por lo tanto el planificador agrega el vínculo causal $inicio \xrightarrow{q} a_{11}$ para lograrla y obtiene un plan que soluciona el problema.

La figura 4.13(e) muestra el plan que se obtiene al aplicar el método *deshabilitar cualquier literal*. En este caso el planificador elige deshabilitar el literal $d \in literales(\mathcal{C}) = \{\sim y, b, d\}$ y por lo tanto agrega la submeta $(\sim d, a_5, ac, [\,])$, la cual debe ser lograda por un paso de acción por ser de tipo "*ac*". La única acción disponible en $\Gamma_{4.9}$ que tiene $\sim d$ como efecto es a_{12}, luego se agrega el paso de acción a_{12} junto con el vínculo causal $a_{12} \xrightarrow{\sim b} a_5$. De esta forma el argumento $\mathcal{C}$ no podrá construirse para derrotar a $\mathcal{B}$ al momento de ejecutar a_5. La precondición r del paso a_{12} es un efecto de *inicio*, por lo tanto el planificador agrega el vínculo causal $inicio \xrightarrow{r} a_{12}$ para lograrla y obtiene un plan que soluciona el problema.

La figura 4.13(f) muestra el plan que se obtiene al aplicar el método *deshabilitar un literal en la base*. Existe un único literal $b \in base(\mathcal{C})$ por lo tanto se agrega la submeta $(-d, a_5, ac, [\,])$. La única acción disponible en $\Gamma_{4.9}$ que tiene $-d$ como efecto es a_2, luego el planificador agrega el paso de acción a_2 junto con el vínculo causal $a_2 \xrightarrow{-b} a_5$. Note que a_9 es una *amenaza acción-acción* para $a_2 \xrightarrow{-b} a_5$, que en este caso el planificador resuelve aplicando el método *retroceso*, para lo cual agrega $a_9 \prec a_2$. De esta forma, el literal b no se propagará hasta a_5 y $\mathcal{C}$ no podrá construirse para derrotar a $\mathcal{B}$ al momento de ejecutar a_5. La precondición r del paso a_2 es un efecto de *inicio*, luego el planificador agrega el vínculo causal $inicio \xrightarrow{r} a_2$ para lograrla y obtiene un plan que soluciona el problema.

La figura 4.13(g) muestra el plan que se obtiene al aplicar el método *deshabilitar un suposición*. Existe una única suposición $note \in suposiciones(\mathcal{C})$ por lo tanto el planificador agrega la submeta $(e, a_5, ac, [\,])$. La única acción disponible en $\Gamma_{4.9}$ que tiene e como efecto es la acción a_{13}, luego el planificador agrega el paso de acción a_{13} junto con el vínculo

causal $a_{13} \xrightarrow{e} a_5$. De esta forma el argumento $\mathcal{C}$ no podrá construirse para derrotar a $\mathcal{B}$ al momento de ejecutar a_5. La precondición r del paso a_{13} es un efecto del paso inicio, por lo tanto el planificador agrega el vínculo causal $inicio \xrightarrow{r} a_{13}$ para lograrla y obtiene un plan que soluciona el problema.

4.4. Planificación de Orden Parcial Argumentativa

En esta sección se presentará una extensión del algoritmo de POP que implementa un nuevo método de planificación definido en esta tesis que se denominará APOP, por las siglas en inglés de: *Argumentative Partial Order Planning*. Este algoritmo permite resolver problemas de planificación definidos utilizando DAKAR como formalismo de representación y la idea central del mismo consiste en realizar una búsqueda sobre el espacio de *planes parciales argumentativos* (definición 4.6).

Al igual que en POP, los problemas de planificación se representan dentro del algoritmo de APOP mediante un plan especial denominado *plan argumentativo nulo* el cual constituye el nodo de partida de la búsqueda y se define de la siguiente forma:

Definición 4.17 (plan argumentativo nulo) Un *plan argumentativo nulo* o *plan argumentativo inicial* para un problema de planificación DAKAR $(\Psi, \Delta, \Gamma, Meta, R)$, notado $plan\_nulo(\Psi, \Delta, \Gamma, Meta, R)$, es un plan $(\mathcal{AC}, \mathcal{AR}, \mathcal{O}, \mathcal{CL}, \mathcal{SL}, \mathcal{SG})$, donde:

- $\mathcal{AC} = \{(inicio, \cdot, \emptyset, \Psi), (fin, \cdot, Meta, \emptyset)\}$
- $\mathcal{AR} = \emptyset$
- $\mathcal{O} = \{inicio \prec fin\}$
- $\mathcal{CL} = \emptyset$
- $\mathcal{SL} = \emptyset$
- $\mathcal{SG} = \{(M, fin, ac\_arg, [\,]) \mid M \in Meta\}$

Ejemplo 4.10 El plan P(a) definido en el ejemplo 4.3, mostrado en la figura 4.1(a), corresponde al *plan argumentativo nulo* para el problema definido en el ejemplo 4.1.

A través de las figuras 4.14, 4.15, 4.16, 4.17, 4.18, 4.19 se presenta un esquema del algoritmo de APOP que sólo considera acciones totalmente instanciadas y permite

resolver problemas de planificación $(\Psi, \Delta, \Gamma, Meta, R)$ que no contienen *restricciones*, esto es, $\mathsf{C} = \emptyset$ y los esquemas de acción en Γ no tienen restricciones.

Como se demostró en el teorema 3.1, cualquier problema de planificación P se puede transformar en un problema de planificación P' que no contiene restricciones, de forma que toda solución para P también es una solución para P' y viceversa.

El algoritmo está definido utilizando un pseudocódigo que combina sentencias de lenguajes imperativos, fórmulas matemáticas y en algunos casos lenguaje natural. En particular, los símbolos $\downarrow$ y $\uparrow$ sobre los parámetros de las funciones o procedimientos, identifican parámetros de entrada y salida respectivamente. Las sentencias **choose** y **fail** se utilizan para describir el no-determinismo. La primitiva **choose** permite al algoritmo elegir entre diferentes alternativas y mantener un registro de las alternativas pendientes. Cuando el algoritmo encuentra una sentencia **fail**, el control vuelve al punto del algoritmo en donde se realizó la última elección y las opciones pendientes son consideradas.

La función principal `APOP`, recibe a través de los parámetros de entrada la descripción de un problema de planificación $(\Psi, \Delta, \Gamma, Meta, R)$ sin restricciones y devuelve, si es que existe, un plan parcial argumentativo $Plan$ que soluciona el problema. Esta función comienza construyendo un *plan nulo* para el problema (función `Construir_Plan_Nulo`) y luego intenta completarlo a través del procedimiento recursivo `Completar_Plan`. En cada instancia, este procedimiento selecciona una submeta $(P, S, Tipo, \Lambda_p)$ del plan, elige un paso para lograrla y lo agrega al plan a través del procedimiento `Elegir_Paso`. Cuando se agrega un paso al plan, el algoritmo debe verificar si se producen nuevas *amenazas* y elegir un método para resolverlas (procedimiento `Resolver_Amenazas`). Si en algún momento el algoritmo falla (esto es, ejecuta una sentencia `fail`) al elegir un paso para lograr una submeta o al resolver una amenaza, volverá al último punto de elección. Cabe destacar que, si bien el orden en que el algoritmo considera las submetas puede influir en el resultado de la búsqueda, cada una de las submetas debe ser considerada en algún momento para asegurar la completitud del mismo. Luego, la selección de las submetas no es un punto de elección.

El procedimiento `Elegir_Paso` (ver figura 4.15) se encarga de elegir un paso y agregarlo al $Plan$ para lograr una submeta $(P, S_j, Tipo, [\langle \mathcal{A}_1, L_1 \rangle, \ldots, \langle \mathcal{A}_n, L_n \rangle])$. Para esto debe considerar acciones y argumentos, por eso construye dos conjuntos: `Pasos_Ac` y `Pasos_Arg`.

El conjunto `Pasos_Ac` (ver figura 4.15, sentencia `(1)`) está formado por todos los pasos de acción que tienen P como efecto. Estos pasos pueden estar creados a partir de

```
function APOP(↓Ψ, ↓Δ, ↓Meta, ↓Γ): Plan;
begin
   Plan:=Construir_Plan_Nulo(Ψ,Meta);
   Completar_Plan(Plan,Δ, Γ);
end

function Construir_Plan_Nulo(↓Ψ, ↓Meta): Plan;
begin
    Plan.AC:= {(inicio, _, ∅, Ψ), (fin, _, Meta, ∅)}; //pasos de acción
    Plan.AR:= ∅;  //pasos de argumento
    Plan.O:= {inicio ≺ fin}; //restricciones de orden
    Plan.SG:= {(G, fin, ac_arg,[ ])| G ∈ Meta} //submetas
    Plan.CL:= ∅; //vinculos causales
    Plan.SL:= ∅; //vinculos de soporte
end

procedure Completar_Plan(↓↑Plan, ↓Δ, ↓Γ);
begin
    if Plan.SG = ∅ then return Plan;
    Sea (P, S, Tipo, Λ_P) ∈ Plan.SG;
    Plan.SG:= Plan.SG \ {(P, S, Tipo, Λ_P)};
    Elegir_Paso(Plan, Δ, Γ, (P, S, Tipo, Λ_P));
    Resolver_Amenazas(Plan, Δ);
    Completar_Plan(Plan,Δ,Γ);
end
```

Figura 4.14: Algoritmo APOP

las acciones en Γ, o pueden ser pasos de acción pertenecientes al plan que puedan aparecer antes del paso S_j.

El conjunto `Pasos_Arg` (ver figura 4.15, sentencia `(2)`) está formado por todos los pasos de argumento que se pueden formar con los argumentos potenciales que se pueden construir a partir de Δ para sustentar P. Esto es, el conjunto `Pasos_Arg` para una submeta $(P, S_j, Tipo, [\langle \mathcal{A}_1, L_1\rangle, \ldots, \langle \mathcal{A}_n, L_n\rangle])$ contiene pasos de la forma $(S_{arg}, \mathcal{B}, \Lambda_{S_{arg}})$ donde:

1. $\mathcal{B} \subseteq \Delta$ es un *argumento potencial* para P y
2. $\Lambda_{S_{arg}} = [\langle \mathcal{A}_1, L_1\rangle, \ldots, \langle \mathcal{A}_n, L_n\rangle, \langle \mathcal{B}, P\rangle]$ es una *línea de argumentación aceptable* (ver definición 2.33) considerando el programa DeLP (Π, Δ) donde:

$$\Pi = base(\mathcal{A}_1) \cup \cdots \cup base(\mathcal{A}_n) \cup base(\mathcal{B})$$

procedure Elegir_Paso($\overset{\downarrow\uparrow}{Plan}$, $\overset{\downarrow}{\Delta}$, $\overset{\downarrow}{\Gamma}$, $\overset{\downarrow}{SM}$)
begin
Sea $SM = (P,\ S_j,\ Tipo,\ [\langle \mathcal{A}_1, L_1\rangle,\ \ldots,\ \langle \mathcal{A}_n, L_n\rangle])$;
(1)Pasos_Ac:= $\{(S_{ac}, A, Pre, Ef)$ | $P \in Ef$, S_{ac} es un paso nuevo creado a partir de $A \in \Gamma$ ó
$(S_{ac}, A, Pre, Ef) \in Plan.\mathcal{AC}$ y $S_{ac} \prec S_j$ *es consistente con* $Plan.\mathcal{O}$ $\}$;
(2)Pasos_Arg:= $\{(S_{arg}, \mathcal{B}, \Lambda_{S_{arg}})$ | $\mathcal{B}$ es un *argumento potencial* de Δ para P y
$\Lambda_{S_{arg}}$=$[\langle \mathcal{A}_1, L_1\rangle,\ \ldots,\ \langle \mathcal{A}_n, L_n\rangle,\ \langle \mathcal{B}, P\rangle]$ *es una línea de arg. aceptable*
en (Π, Δ), donde $\Pi = base(\mathcal{A}_1) \cup \cdots \cup\ base(\mathcal{A}_n) \cup\ base(\mathcal{B})\}$;
(3)**case** $Tipo$ **of**
ac_arg: Pasos:= Pasos_Ac $\cup$ Pasos_Arg;
ac: Pasos:= Pasos_Ac;
arg: Pasos:= Pasos_Arg;
endcase;
(4)**if** Pasos= $\emptyset$ **then fail**;
(5)**choose** S **from** Pasos;
(6)**if** $S = (S_i, A_i, Pre_i, Ef_i) \in$ Pasos_Ac **then**
begin
$Plan.\mathcal{CL}$:= Plan.$\mathcal{CL} \cup \{S_i \xrightarrow{P} S_j\}$;
(6.1)**if** $(P,\ S_j,\ Tipo,\ \Lambda_P) \in Plan.\mathcal{AR}$ **then** // S_j *es un paso de argumento*
begin
Ac:= $soportado(S_j)$;
$Plan.\mathcal{O}$:= Plan.$\mathcal{O} \cup \{S_i \prec Ac\}$;
end
(6.2)**else** // S_j *es un paso de acción*
$Plan.\mathcal{O}$:= Plan.$\mathcal{O} \cup \{S_i \prec S_j\}$;
(6.3)**if** $S \notin Plan.\mathcal{AC}$ **then** // S *es un paso de acción nuevo*
begin
$Plan.\mathcal{AC}$:= $Plan.\mathcal{AC} \cup (S_i, A_i, Pre_i, Ef_i)$
$Plan.\mathcal{O}$:= $Plan.\mathcal{O} \cup \{START \prec S_i,\ S_i \prec FINISH\}$;
$Plan.\mathcal{SG}$:= $Plan.\mathcal{SG}$: $\cup$ {(G, S_i, *ac_arg*,[ ])| $G \in Pre_i$)};
end
end
(7)**if** $S = (S_i, \mathcal{B}, \Lambda_i) \in$ Arg_Steps **then**
begin
$Plan.\mathcal{AR}$:= $Plan.\mathcal{AR} \cup \{(S_i, \mathcal{B}, \Lambda_i)\}$;
$Plan.\mathcal{SL}$:= $Plan.\mathcal{SL} \cup \{S_i \overset{P}{\succ} S_j\}$;
$Plan.\mathcal{SG}$:= $Plan.\mathcal{SG} \cup \{$(G, S_i, *ac*,[ ])|$G \in$ Base($\mathcal{B}$)$\}$;
end
end *//Elegir_Paso(...)*

Figura 4.15: Algoritmo APOP: Elección de Pasos

La condición 2 es necesaria en el caso que la submeta $(P, S_j, Tipo, \Lambda_P)$ haya sido agregada al plan por los métodos *atacar un punto interno* o *atacar una suposición* para resolver una *amenaza argumento-argumento* (ver sección 4.3.5). Esta condición asegura que el planificador sólo considere para lograr P, argumentos que sean adecuados en el contexto de la línea de argumentación Λ_P

Una vez construidos los conjuntos $Pasos\_Ac$ y $Pasos\_Arg$ el algoritmo selecciona el conjunto de pasos adecuados para lograr la submeta $(P, S_j, Tipo, \Lambda_P)$ considerando el valor de $Tipo$ (ver sentencia `(3)`). Si $Tipo = ac\_arg$ la submeta puede ser lograda por pasos de acción y por pasos de argumento, por lo tanto `Pasos:= Pasos_Ac` $\cup$ `Pasos_Arg`. Si $Tipo = ac$ la submeta puede ser lograda solo por acciones, entonces `Pasos:= Pasos_Ac`. Si $Tipo = arg$ la submeta puede ser lograda solo por argumentos, luego `Pasos:= Pasos_Arg`.

Observación 4.5 Es claro que no es necesario construir el conjunto `Pasos_Arg` cuando $Tipo = ac$, como tampoco es necesario construir el conjunto `Pasos_Ac` cuando $Tipo = arg$. Para mantener el algoritmo lo más claro posible, siempre se calculan ambos conjuntos.

Si el conjunto `Pasos` es vacío (esto es, no existen pasos para lograr $(P, S_j, Tipo, \Lambda_P)$) el algoritmo falla y vuelve al último punto de elección (ver sentencia `(4)`). Sino, elige un paso S_i del conjunto (ver sentencia `(5)`) y lo agrega al plan junto con las restricciones de orden, submetas y vínculos causales o de soporte correspondientes. Si el algoritmo falla mas adelante, volverá a este punto y elegirá otro paso para lograr la submeta.

Si el paso S_i elegido para lograr $(P, S_j, Tipo, \Lambda_P)$ es un paso de acción (ver sentencia `(6)`), entonces se extenderá el plan de la siguiente forma:

- Se agregará el vínculo causal $S_i \xrightarrow{P} S_j$
- Si S_j es un paso de argumento se agregará la restricción de orden $S_i \prec Ac$, donde Ac es el paso de acción soportado por S_j (sentencia `(6.1)`). Esto se debe a que (como se explicó anteriormente) solo es posible establecer un orden entre los pasos de acción del plan y S_i debe aparecer antes de Ac para que el argumento asociado S_j pueda construirse para soportar Ac. Si S_j es un paso de acción se agregará solamente la restricción de orden $S_i \prec S_j$ (sentencia `(6.2)`).
- Si el paso elegido (S_i, A, Pre_i, Ef_i) para lograr la submeta no es un paso presente en el plan (sentencia `(6.3)`), entonces es incorporado al mismo después de $START$ y

antes $FINISH$. Cada literal $G \in Pre_i$ es agregado al plan como una nueva submeta de la forma $(G, S_i, ac\_arg, [])$. Por tratarse de la precondición de una acción, la línea de argumentación es vacía y el tipo de la submeta es "ac_arg", lo cual permite que pueda ser lograda tanto por acciones como por argumentos.

Si el $(S_i, \mathcal{B}, \Lambda_i)$ elegido para lograr $(P, S_j, Tipo, \Lambda_P)$ es un paso de argumento (ver sentencia `(7)`), entonces es agregado al plan junto con el vínculo de soporte $S_i \overset{P}{\succ\!\!-} S_j$. Cada literal $G \in Base(\mathcal{B})$ es agregado como una nueva submeta de la forma $(G, S_i, ac, [])$. Por tratarse de los literales de la base de un argumento, la línea de argumentación es vacía y el tipo de la submeta es "ac", lo cual permite que solo pueda ser lograda por acciones.

El procedimiento `Resolver_Amenazas` (ver figura 4.16) se encarga de detectar y resolver los diferentes tipos de amenazas que pueden aparecer en el plan.

```
procedure Resolver_Amenazas(Plan↓↑, Δ↓)
begin
   Resolver_Amenazas_Acción_Acción(Plan);
   Resolver_Amenazas_Acción_Base(Plan);
   Resolver_Amenazas_Acción_Argumento(Plan);
   Resolver_Amenazas_Acción_Suposición(Plan);
   Resolver_Amenazas_Argumento_Argumento(Plan, Δ);
end;
```

Figura 4.16: Algoritmo APOP: Resolución de amenazas

Cada uno de los procedimientos invocados dentro de `Resolver_Amenazas` corresponde a un tipo de amenaza de las definidas en la sección 4.3, y se encarga de detectar y resolver la amenaza correspondiente aplicando alguno de los métodos definidos para la misma. Dado que existe más de un método para resolver cada tipo de amenaza, la aplicación de un método particular constituye un punto de elección. Si el algoritmo falla después de aplicar un método, intentará resolver la amenaza aplicando otro. Si el algoritmo falla después de aplicar todos los métodos posibles para resolver una amenaza, volverá al último punto de elección.

```
procedure Resolver_Amenazas_Acción_Acción(Plan↓↑);
begin
 for each S_i -P-> S_j ∈ {A -•-> B | A -•-> B ∈ Plan.CL, (A,·,·,·) ∈ Plan.AC } do
   for each (S_k,·,·,·) ∈ {(S,·,·,Ef) | (S,·,·,Ef) ∈ Plan.AC ∧
                          S_i≺S≺S_j  es consistente con Plan.O ∧
                          (P̄ ∈ Ef ∨ P⁻ ∈ Ef)} do
       choose
            retroceso: //figura 4.2(c)
               if S_k≺S_i  es consistente con Plan.O then
                  Plan.O:= Plan.O ∪ {S_k≺S_i};
               else fail;
            avance: //figura 4.2(d)
               if S_j≺S_k  es consistente con Plan.O then
                  Plan.O:= Plan.O ∪ {S_j≺S_k};
               else fail;
       end;
end;

procedure  Resolver_Amenazas_Acción_Base(Plan↓↑);
begin
 for each S_i -P-> S_j ∈ {A -•-> B | A -•-> B ∈ Plan.CL, (A,·,·) ∈ Plan.AR } do
  begin
    S_Ac:= soportado(S_j); //Definición 4.10
    for each (S_k,·,·,·) ∈ {(S,·,·,Ef) | (S,·,·,Ef) ∈ Plan.AC ∧
                          S_i≺S≺S_Ac  es consistente con Plan.O ∧
                          (P̄ ∈ Ef ∨ −P ∈ Ef)} do
       choose
            retroceso*: //figura 4.4(c)
               if S_k≺S_i  es consistente con Plan.O then
                  Plan.O:= Plan.O ∪ {S_k≺S_i};
               else fail;
            avance*: //figura 4.4(d)
               if S_Ac≺S_k  es consistente con Plan.O then
                  Plan.O:= Plan.O ∪ {S_Ac≺S_k};
               else fail;
        end;
  end;
end;
```

Figura 4.17: Algoritmo APOP: Resolución de amenazas *acción-acción* y *acción-base*

```
procedure Resolver_Amenazas_Acción_Argumento(Plan^{↓↑});
begin
 for each S_i ≻^H S_j ∈ Plan.SL do
   begin
     Sea (S_i, B, Λ_B) ∈ Plan.AR
     for each (S_k, N̄) ∈ {(A, N̄)| (A, N̄) ∈ AEP(S_j) ∧ N ∈ cabezas(B)} do
     // AEP(S_j) = acciones-efectos propagados hasta S_j (definición 4.12)
       choose
          avance*: //figura 4.6(b)
            if S_j≺S_k es consistente con Plan.O then
               Plan.O:= Plan.O ∪ {S_j≺S_k};
            else fail;
          confrontación por eliminación: //figura 4.6(c)
             Plan.O:= Plan.O ∪ {S_k≺S_j};
             Plan.SG:= Plan.SG ∪ (−N, S_j, ac,[ ]);
       end;
   end;
end;

procedure Resolver_Amenazas_Acción_Suposición(Plan^{↓↑});
begin
 for each S_i ≻^H S_j ∈ Plan.SL do
  begin
     Sea (S_i, B, Λ_B) ∈ Plan.AR
     for each (S_k, N) ∈ {(A, N)| (A, N) ∈ AEP(S_j) ∧ not N ∈ suposiciones(B)} do
       choose
          avance*: //figura 4.9(b)
            if S_j≺S_k es consistente con Plan.O then
               Plan.O:= Plan.O ∪ {S_j≺S_k};
            else fail;
          confrontación por complemento: //figura 4.9(c)
             Plan.O:= Plan.O ∪ {S_k≺S_j};
             Plan.SG:= Plan.SG ∪ (N̄, S_j, ac,[ ]);
          confrontación por eliminación: //figura 4.9(d)
             Plan.O:= Plan.O ∪ {S_k≺S_j};
             Plan.SG:= Plan.SG ∪ (−N, S_j, ac,[ ]);
       end;
  end;
end;
```

Figura 4.18: Algoritmo APOP: Res. de amenazas *acción-argumento* y *acción-suposición*

```
procedure Resolver_Amenazas_Argumento_Argumento(Plan↓↑, Δ↓);
begin
(1)for each S_i ≻^H S_j ∈ Plan.SL do
   begin
     Sea (S_i, B, [⟨A_1, L_1⟩, ..., ⟨A_n, L_n⟩, ⟨B, H⟩]) ∈ Plan.AR
  (2)for each C ∈ {C| C es un argumento potencial de Δ y
                      es una amenaza argumento-argumento para S_i} do
       choose
           avanzar el derrotador:  //figura 4.11(b)
              choose F from base(C)
              for each A ∈ {A|(A, F) ∈ AEP(S_j)} do
                 if S_j≺A  es consistente con Plan.O then
                    Plan.O := Plan.O ∪ {S_j≺A};
                 else fail;
           atacar un punto interno: //figura 4.11(c)
              choose P from cabezas(C);
              Plan.SG := Plan.SG ∪ (P̄, S_j, arg, [⟨A_1, L_1⟩, ..., ⟨A_n, L_n⟩, ⟨B, b⟩, ⟨C, q⟩]);
           atacar una suposición: //figura 4.11(d)
              choose not P from suposiciones(C);
              Plan.SG := Plan.SG ∪ (P, S_j, arg, [⟨A_1, L_1⟩, ..., ⟨A_n, L_n⟩, ⟨B, b⟩, ⟨C, q⟩]);
           deshabilitar cualquier literal: //figura 4.12(b)
              choose N from literales(C) \ literales(B);
              Plan.SG := Plan.SG ∪ (N̄, S_j, ac, [ ]);
           deshabilitar un literal de la base: //figura4.12(c)
              choose F from base(C) \ base(B);
              Plan.SG := Plan.SG ∪ (−F, S_j, ac, [ ]);
           deshabilitar una suposición: //figura 4.12(d)
              choose not P from suposiciones(C);
              Plan.SG := Plan.SG ∪ (P, S_j, ac, [ ]);
       end;
   end;
end;
```

Figura 4.19: Algorimo APOP: Resolución de amenazas *argumento-argumento*

Observación 4.6 Para detectar y resolver las amenazas acción-acción en la figura 4.17, el procedimiento analiza cada paso de acción para cada vínculo causal del plan. La detección de las amenazas acción-acción puede hacerse más eficientemente de manera incremental:

- Cuando se agrega un vínculo causal al plan sólo se analizan las acciones para ver si

lo amenazan.

- Cuando se agrega una acción al plan sólo se analizan los vínculos causales para ver si son amenazados por la acción.

En general, algo similar ocurre en cada uno de los procedimientos encargados de detectar y resolver los diferentes tipos de amenazas. El algoritmo presentado intenta una especificación clara del procedimiento, esto en algunos casos conspira contra la eficiencia del mismo.

4.5. Resumen

En este capítulo se definió un nuevo método de planificación denominado *Planificación de Orden Parcial Argumentativa*, que permite resolver problemas definidos en el formalismo de representación DAKAR. Este método combina técnicas de Planificación de Orden Parcial con Argumentación Rebatible, lo cual permite utilizar el conocimiento de un dominio de planificación para razonar rebatiblemente durante la construcción de un plan.

En la sección 4.2 se definió una nueva estructura de plan que combina acciones y argumentos denominada *Plan Parcial Argumentativo*. Luego, en la sección 4.3, se extendió la noción de *amenaza* para considerar los nuevos tipos de interferencias que surgen en este tipo de planes y se definieron nuevos métodos para resolver cada una de ellas. En base a los conceptos y métodos definidos, en la sección 4.4 se presentó el algoritmo APOP, una extensión al algoritmo tradicional de POP que incorpora argumentación rebatible para construir *planes parciales argumentativos* para resolver problemas de planificación DAKAR.

Capítulo 5

Herramientas implementadas

Durante el desarrollo de esta tesis se implementó un planificador en base a los algoritmos propuestos, y se desarrolló una herramienta de visualización que permite observar el proceso de planificación y los planes obtenidos. En este capítulo se describirán algunos detalles de la implementación, funcionamiento y uso de las herramientas implementadas.

Dado que el planificador es un prototipo implementado como prueba de concepto no se realizaron, pruebas de velocidad ni comparaciones con otros planificadores.

5.1. Extendiendo el algoritmo APOP

En esta sección se introducirá EAPOP(*Expanded* APOP), un algoritmo basado en APOP que fue utilizado para la implementación de un planificador desarrollado como complemento de esta tesis. Este nuevo algoritmo combina una búsqueda iterativa con un proceso de expansión similar al utilizado por el método de planificación de *graphplan* [BF97]. Como se verá más adelante, esta combinación tiene las siguientes ventajas:

- reduce el espacio de búsqueda.
- permite calcular de manera general la longitud mínima de la solución, lo cual permite evitar ciclos de búsqueda infructuosos.
- posibilita determinar cuando un problema de planificación no tiene solución.

Siguiendo las ideas de *graphplan*, el algoritmo EAPOP alterna iterativamente entre dos fases: una *fase de expansión* y una *fase de búsqueda*. Durante la fase de expansión, en lugar de extender un grafo, se extienden 2 conjuntos: un conjunto de acciones totalmente instanciadas Acc y un conjunto de argumentos Arg. Luego, en la fase de búsqueda, se invoca al algoritmo APOP imponiendo un límite sobre la longitud del plan, y restringiendo el espacio de la misma a aquellos planes que se pueden construir con las acciones y argumentos presentes en los conjuntos Acc y Arg respectivamente.

En cada iteración los conjuntos son expandidos a partir de los conjuntos de la iteración anterior. A continuación se define formalmente el contenido de los conjuntos generados en la fase de expansión para cada iteración.

Definición 5.1 (Conjuntos generados en la fase de expansión) Sea $(\Psi, \Delta, \Gamma, Meta, R)$ un problema de planificación sin restricciones. Los conjuntos Acc_1 y Arg_1 correspondientes a la iteración 1, se definen de la siguiente forma:

- $Arg_1 = \{\mathcal{B} |\ \mathcal{B}$ es un argumento construido a partir del programa $(\Delta, \Psi)\}$
- $Acc_1 = \{A |\ A \in \Gamma$ y A es aplicable a partir del estado inicial $\Psi\}$

Sea Acc_i y Arg_i los conjuntos correspondientes a la i-esima iteración. Luego, los conjuntos Acc_{i+1} y Arg_{i+1} correspondientes a la iteración $i+1$ se definen como sigue:

- $Arg_{i+1} = Arg_i \cup \{\mathcal{B} |\ \mathcal{B}$ es un argumento potencial construido a partir Δ, donde $base(\mathcal{B}) \subseteq \Psi \cup efectos(Acc_i)\}$;
- $Acc_{i+1} = Acc_i \cup \{A |\ A \in \Gamma$ y $precondiciones(A) \subseteq \Psi \cup efectos(Acc_i) \cup conlusiones(Arg_{i+1})\}$;

Note que los efectos de las acciones en Acc_i permiten generar nuevos argumentos que se incorporan a Arg_{i+1}. A su vez, las conclusiones de los argumentos en Arg_{i+1} junto con los efectos de las acciones en Acc_i permiten generar nuevas acciones para el conjunto Acc_{i+1}. De esta forma, los conjuntos contienen las acciones que se podrían ejecutar y los argumentos que se podrían construir encadenando acciones y argumentos a partir del estado inicial.

A continuación se presenta un ejemplo que muestra como se modifican los conjuntos durante la fase de expansión para un problema de planificación proposicional.

Ejemplo 5.1 Sea $(\Delta_{5.1}, \Gamma_{5.1})$ un dominio de planificación donde

$$\Delta_{5.1} = \left\{ \begin{array}{c} a \prec b, \\ b \prec c, \\ c \prec d, e, \\ q \prec s, \\ s \prec t \end{array} \right\}$$

$$\Gamma_{5.1} = \left\{ \begin{array}{l} \{b\}, \{\}^- \xleftarrow{a_1} \{p\}, not\ \{\}, \\ \{c\}, \{\}^- \xleftarrow{a_2} \{q\}, not\ \{\}, \\ \{d\}, \{\}^- \xleftarrow{a_3} \{r\}, not\ \{\}, \\ \{e\}, \{\}^- \xleftarrow{a_4} \{s\}, not\ \{\}, \end{array} \right\}$$

Consideremos el problema de planificación $(\Psi_{5.1}, \Delta_{5.1}, \Gamma_{5.1}, Meta_{5.1}, \mathsf{C}_{5.1})$ donde: $\Psi_{5.1} = \{t\}$, $Meta_{5.1} = \{a\}$ y $\mathsf{C}_{5.1} = \{\}$.

Inicialmente, a partir del programa $(\Delta_{5.1}, \Psi_{5.1})$ se puede construir el siguiente conjunto de argumentos:

$$Arg_1 = \left\{ \begin{array}{l} \mathcal{B}_1 = \langle \{(s \prec t)\}, s \rangle \\ \mathcal{B}_2 = \langle \{(q \prec s), (s \prec t)\}, q \rangle \end{array} \right\}$$

y se genera el conjunto $Acc_1 = \{a_2, a_4\}$ con las acciones aplicables a partir de $\Psi_{5.1}$

Para generar el conjunto de argumentos Arg_2, en la siguiente iteración se consideran los efectos de las acciones en Acc_1 y los literales en $\Psi_{5.1}$. A partir de los literales en $efectos(Acc_1) \cup \Psi_{5.1} = \{t, c, e\}$ se puede generar el siguiente conjunto de argumentos:

$$Arg_2 = \left\{ \begin{array}{l} \mathcal{B}_1 = \langle \{(s \prec t)\}, s \rangle \\ \mathcal{B}_2 = \langle \{(q \prec s), (s \prec t)\}, q \rangle \\ \mathcal{B}_3 = \langle \{(b \prec c)\}, b \rangle \\ \mathcal{A}_1 = \langle \{(a \prec b), (b \prec c)\}, a \rangle \end{array} \right\}$$

Note que $base(\mathcal{A}_1) = base(\mathcal{B}_3) = \{c\} \subseteq efectos(Acc_1) \cup \Psi_{5.1}$.

Para generar el conjunto de acciones Acc_2 se consideran los literales en $conclusiones(Arg_2) \cup efectos(Acc_1) \cup \Psi_{5.1} = \{t, c, e, s, q, b, a\}$. Dado que $precondiciones(a_1) = \{p\} \not\subseteq \{t, c, e, s, q, b, a\}$ y $precondiciones(a_3) = \{r\} \not\subseteq \{t, c, e, s, q, b, a\}$ no es posible incorporar nuevas acciones a Acc_2 y por lo tanto $Acc_2 = Acc_1 = \{a_2, a_4\}$.

La figura 5.1 muestra gráficamente el proceso realizado en la fase de expansión y su relación con los *planes parciales argumentativos* (PPA). Los triángulos representan argumentos y los rectángulos acciones. Una flecha entre un argumento de un conjunto Arg_i y una acción de un conjunto Acc_i indica que la conclusión del argumento aparece en las precondiciones de la acción y puede verse como un vínculo de soporte. Una flecha entre una acción de un conjunto Acc_i y un argumento en Arg_{i+1} (o una acción en el conjunto Acc_{i+1}) indica que un efecto de la acción aparece en la base del argumento (o en las precondiciones de otra acción respectivamente) y puede verse como un vínculo causal. Los argumentos y acciones que aparecen en gris corresponden a los argumentos y acciones generados en la iteración 1.

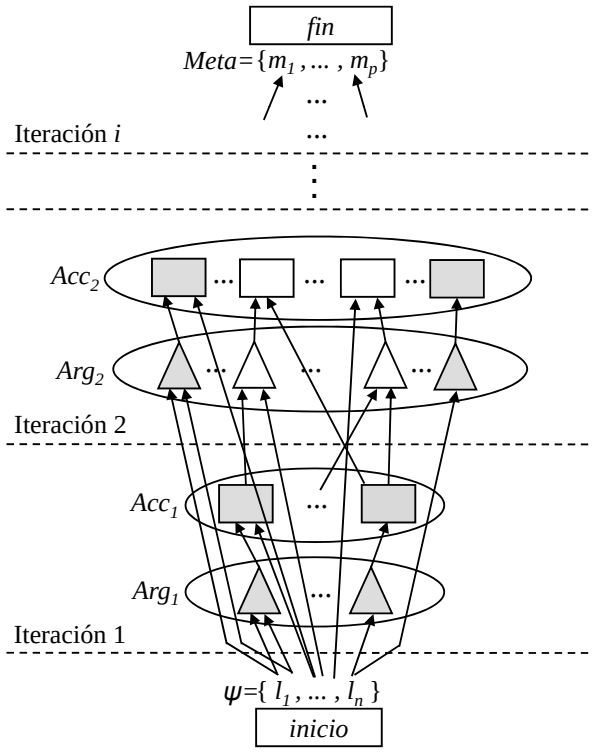

Figura 5.1: Fase de expansión y su relación con los PPA

La fase de búsqueda se iniciará en una iteración i, sólo si cada una de las metas del problema aparece en las conclusiones de los argumentos en Arg_i o en los efectos de las acciones en Acc_i (esto es, $Meta \subseteq conclusiones(Arg_i) \cup efectos(Acc_i)$). Esta es una condición necesaria pero insuficiente para garantizar que a partir de los argumentos en Arg_i y acciones en Acc_i se puede construir un PPA para solucionar el problema.

Por otra parte, el número de iteración puede utilizarse como una cota mínima sobre la longitud del plan que soluciona el problema. Note que para que una acción aparezca en Acc_i, sus precondiciones deben aparecer en los efectos de las acciones en Acc_{i-1} o en las conclusiones de los argumentos en Arg_{i-1}. Es decir que la ejecución de una acción en Acc_i depende de las acciones en Acc_{i-1} y de los argumentos en Arg_{i-1}. Como puede observarse en la figura 5.1, el plan deberá tener como mínimo una longitud i para que se logren las metas del problema y el paso fin pueda ejecutarse.

Otra de las ventajas de la fase de expansión es que permite reducir el espacio de búsqueda, ya que para la construcción del plan sólo se consideran las acciones y argumentos presentes en los conjuntos expandidos. Si un argumento no aparece en el conjunto Arg_i o una acción no aparece en Acc_i significa que en un plan de longitud i no será posible construir el argumento o ejecutar la acción, porque alguno de los literales en la base del argumento o en la precondición de la acción no tendrá soporte. Por lo tanto, los argumentos y acciones que es posible generar en el dominio de planificación pero que no forman parte de los conjuntos expandidos, pueden ser descartados para la construcción del plan como se muestra en el siguiente ejemplo.

Ejemplo 5.2 Considere el problema de planificación $(\Psi_{5.1}, \Delta_{5.1}, \Gamma_{5.1}, Meta_{5.1}, \mathsf{C}_{5.1})$ del ejemplo 5.1. El algoritmo EAPOP deberá expandir los conjuntos hasta la iteración 2 para lograr la condición necesaria para iniciar la búsqueda ($Meta_{5.1} \subseteq conclusiones(Arg) \cup efectos(Acc)$). Observe que a, la única meta del problema, es la conclusión del argumento $\mathcal{A}_1$ que aparece en el conjunto Arg_2. Luego, el algoritmo APOP es invocado con los conjuntos

$$Acc = \{a_2, a_4\} \ , \ Arg = \left\{ \begin{array}{l} \mathcal{B}_1 = \langle \{(s \prec t)\}, s \rangle \\ \mathcal{B}_2 = \langle \{(q \prec s), (s \prec t)\}, q \rangle \\ \mathcal{B}_3 = \langle \{(b \prec c)\}, b \rangle \\ \mathcal{A}_1 = \langle \{(a \prec b), (b \prec c)\}, a \rangle \end{array} \right\}$$

y considerando 2 como límite en la longitud del plan. Con estas restricciones sólo es posible construir el plan que se muestra en la figura 5.2(a), el cual es una solución para el problema.

Si no se realizara la fase de expansión, el algoritmo debería considerar todos los argumentos potenciales y acciones posibles que se pueden generar en el dominio para lograr cada submeta. Por ejemplo, para lograr la meta a se deberían considerar todos los argumentos potenciales para a que se pueden construir a partir de $\Delta_{5.1}$, los cuales son:

$\mathcal{A}_1 = \langle\{(a \prec b), (b \prec c)\}, a\rangle$
$\mathcal{A}_2 = \langle\{(a \prec b)\}, a\rangle$
$\mathcal{A}_3 = \langle\{(a \prec b), (b \prec c), (c \prec d, e)\}, a\rangle$

Como puede observarse en la figura 5.2(b) utilizando el argumento $\mathcal{A}_2$ no es posible obtener un plan para solucionar el problema porque no existe ningún argumento o acción para lograr la precondición p del paso de acción a_1. La figura 5.2(c) muestra que utilizando el argumento $\mathcal{A}_3$ tampoco es posible obtener un plan, porque no existe ningún argumento o acción para lograr la precondición r del paso de acción a_3.

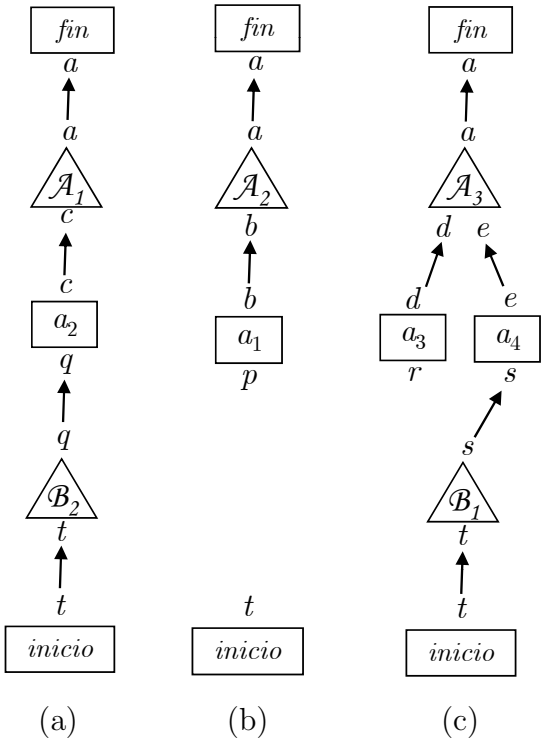

Figura 5.2: Planes parciales para el problema del ejemplo 5.1

La figura 5.3 muestra un esquema del algoritmo EAPOP, que utiliza como primitiva el procedimiento APOP. Este último debe ser modificado levemente para considerar los conjuntos de argumentos y acciones expandidos, y para considerar un límite en la búsqueda. Primero, note que el procedimiento APOP es invocado con los conjuntos Arg y Acc en lugar de Δ y Γ respectivamente. De esta forma el algoritmo considerará los argumentos y acciones generadas por la fase de expansión, al momento de elegir un paso para lograr una submeta (ver figura 4.15, sentencias`(1)` y `(2)`), o para detectar los argumentos que constituyan una *amenaza* para otros argumentos (ver figura 4.19, sentencia `(2)`).

```
function EAPOP(↓Ψ, ↓Δ, ↓Meta, ↓Γ): Plan;
begin
   Arg := {A | A es un argumento construido a partir de (Δ, Ψ)};
   Acc := {Ac | Ac ∈ Γ y Ac es aplicable a partir de Ψ };
   punto_fijo := false; Cota := 1;   Max := 1; Terminar := false;
   while not Terminar do
     if Meta ⊆ conclusiones(Arg) ∪ efectos(Acc) then
        choose
           éxito:
              Plan := APOP(Ψ, Arg, Meta, Acc);
              Terminar := true;
           fracaso: Plan := no_plan;
        end;
     else
        if punto_fijo then //no existe plan posible
          begin
            Plan := no_plan;
            Terminar := true;
          end;
     if Plan = no_plan and not Terminar then
        if Max < Cota and punto_fijo then //no existe plan posible
            Terminar := true;
        else
           if not punto_fijo then
             begin
               Nuevo_Arg := Arg ∪ {A | A es un argumentos potencial a partir de Δ,
                                       donde base(A) ⊆ Ψ ∪ efectos(Acc)};
               Nuevo_Acc := Arg ∪ {Ac | Ac ∈ Γ y precondiciones(Ac) ⊆ Ψ ∪
                                       efectos(Acc) ∪ conclusiones(Nuevo_Arg)};
               punto_fijo :=(Nuevo_Arg = Arg) and (Nuevo_Acc = Acc);
               Arg := Nuevo_Arg;
               Acc := Nuevo_Acc;
             end;
             Cota := Cota + 1;
end;
```

Figura 5.3: Algoritmo EAPOP

Para que el algoritmo considere un límite en la búsqueda, se utilizan dos variables globales $Cota$ y Max. La variable $Cota$ mantiene el límite actual para la búsqueda y la variable Max contiene la cota máxima alcanzada al realizar la última búsqueda infruc-

tuosa de un plan. En este caso, la medida utilizada para limitar la búsqueda es la longitud del plan, aunque cada implementación particular del algoritmo podría considerar otras medidas (por ejemplo, la cantidad de pasos del plan).

Cada vez que el procedimiento `APOP` modifica el plan (esto es, agrega un nuevo paso, restricción de orden, vínculo causal o de soporte) debe actualizar Max y, en caso de que el plan exceda la $Cota$ suministrada, debe fallar para que se inicie una nueva fase de expansión y se incremente la $Cota$. Para lograr esto solo es necesario extender el procedimiento `Completar_Plan` (presentado en el capitulo 4) como se muestra en la figura 5.4.

```
procedure Completar_Plan(Plan↓↑, Δ↓, Γ↓);
begin
    if Plan.SG = ∅ then return Plan;
    Sea (p, S, Tipo, Λ_p) ∈ Plan.SG;
    Plan.SG:= Plan.SG \ {(p, S, Tipo, Λ_p)};
    Elegir_Paso(Plan, Δ, Γ, (p, S, Tipo, Λ_p));
    Resolver_Amenazas(Plan, Δ);
    if longitud(Plan) > Max then  Max:= longitud(Plan);
    if longitud(Plan) > Cota then fail;
    Completar_Plan(Plan,Δ,Γ);
end
```

Figura 5.4: Extensión del proc. `Completar_Plan` para considerar una cota en la búsqueda

Como se mencionó anteriormente, los conjuntos Arg y Acc son expandidos hasta que se cumple la condición necesaria para la existencia de un plan (esto es, $Meta \subseteq conclusiones(Arg) \cup efectos(Acc)$) donde se invoca el algoritmo APOP para encontrarlo. Si el proceso de expansión llega a un *punto fijo* (esto es, los conjuntos no se modificaron de una iteración a la siguiente) y la condición $Meta \subseteq conclusiones(Arg) \cup efectos(Acc)$ no se cumple, se puede asegurar que no se cumplirá nunca y por lo tanto no existe un plan para solucionar el problema.

Cuando el procedimiento APOP falla en encontrar un plan, también es posible determinar si se debe a que no existe un plan para solucionar el problema. Para esto se deben cumplir dos condiciones:

1. que la cota máxima alcanzada durante la búsqueda infructuosa sea menor que la cota suministrada, y
2. que el proceso de expansión haya llegado a un *punto fijo*.

La primera condición indica que la falla en la búsqueda no se produjo por que la cota en la longitud del plan era insuficiente, dado que esta nunca se alcanzó. La segunda condición indica que ya se consideraron todos los argumentos y acciones posibles para solucionar el problema (no se pueden generar más). Por lo tanto, si se incrementa la cota y se invoca nuevamente el algoritmo APOP se obtendrá el mismo resultado.

Observación 5.1 Para garantizar que el proceso de expansión llegue a un punto fijo, tanto las reglas en Δ como los esquemas de acción en Γ no deben contener funciones. Como se muestra en siguiente el ejemplo, si se permite el uso funciones, los conjuntos expandidos pueden crecer infinitamente.

Considere un problema de planificación $(\Delta_{5.1}, \Psi_{5.1}, \Gamma_{5.1}, \dots)$, donde
$\Delta_{5.1} = \{q(s(Y)) \prec q(Y)\}$,
$\Psi_{5.1} = \{q(b), p(a)\}$ y
$\Gamma_{5.1} = \{\{p(f(X)\}, \{\}^- \overset{n(X)}{\longleftarrow} \{p(X)\}, not\ \{\}\}$.
El conjunto de argumentos generado en la primera iteración de la fase de expansión

$$Arg_1 = \left\{ \begin{array}{c} q(s(b)) \prec q(b) \\ q(s(s(b))) \prec q(s(b)) \\ \dots \\ q(s(s(...s(b)))) \prec q(s(...s(b))) \end{array} \right\}$$

es infinito, dado que existe una cantidad infinita de argumentos que se pueden generar a partir del programa $(\Delta_{5.1}, \Psi_{5.1})$.

Por otra parte, el conjunto de acciones crece infinitamente:

$$Acc_1 = \left\{ \{p(f(a))\}, \{\}^- \overset{n(a)}{\longleftarrow} \{p(a)\}, not\ \{\} \right\}$$

$$Acc_2 = \left\{ \begin{array}{c} \{p(f(a))\}, \{\}^- \overset{n(a)}{\longleftarrow} \{p(a)\}, not\ \{\} \\ \{p(f(f(a)))\}, \{\}^- \overset{n(f(a))}{\longleftarrow} \{p(f(a))\}, not\ \{\} \end{array} \right\}$$

$$\dots$$

$$Acc_n = \left\{ \begin{array}{c} \{p(f(a))\}, \{\}^- \overset{n(a)}{\longleftarrow} \{p(a)\}, not\ \{\} \\ \{p(f(f(a)))\}, \{\}^- \overset{n(f(a))}{\longleftarrow} \{p(f(a))\}, not\ \{\} \\ \dots \\ \{p(f(f(...f(a))))\}, \{\}^- \overset{n((f(...f(a))}{\longleftarrow} \{p(f(...f(a)))\}, not\ \{\} \end{array} \right\}$$

Note que, utilizando el efecto de una acción generada a partir del esquema de acción $\{p(f(X)\}, \{\}^{-} \overset{n(X)}{\longleftarrow} \{p(X)\}, not\ \{\}$, se puede generar una nueva acción a partir del mismo esquema, y así sucesivamente.

5.2. Implementación de un planificador para APOP

En base al algoritmo EAPOP presentado en la sección anterior, durante el desarrollo de esta tesis se implementó un prototipo de planificador, junto con una herramienta de visualización que permite observar el proceso de planificación y los planes obtenidos. La figura 5.5 muestra un esquema de la arquitectura del prototipo implementado. La implementación se realizó completamente en SWI-Prolog y consiste de cuatro módulos: el *Planificador*, el *módulo DeLP*, el *Generador de Grafos* y el *Visualizador*.

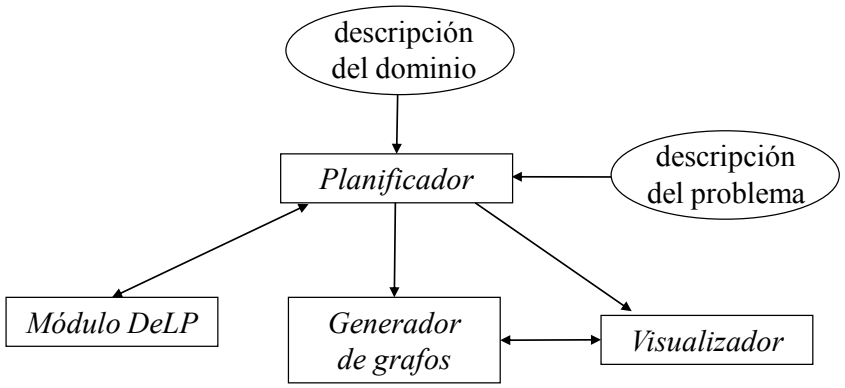

Figura 5.5: Arquitectura del prototipo implementado

El *Planificador* controla los otros tres módulos y, a través de una implementación del algoritmo EAPOP, se encarga de obtener *planes parciales argumentativos* para un problema definido sobre un dominio de planificación DAKAR. Durante la construcción de un plan, el *Planificador* utiliza al *módulo DeLP* para construir argumentos potenciales, obtener los derrotadores para un argumento y verificar si una línea de argumentación es aceptable. El *Generador de Grafos* se encarga de traducir los planes obtenidos por el planificador, en un grafo más adecuado para ser mostrado por el *Visualizador*. Este último modulo, provee un interfaz gráfica para que el usuario pueda observar los planes y fue implementado utilizando las herramientas gráficas XPCE provistas por SWI-prolog.

A continuación se presentará un ejemplo para mostrar el uso y funcionamiento de estas herramientas. Para esto consideraremos el dominio de planificación $(\Delta_{3.2}, \Gamma_{3.3})$ definido en el ejemplo 3.4, sobre un agente encargado de la limpieza de una oficina.

El *Planificador* recibe la descripción de un dominio a través de un archivo de texto que contiene la definición de los esquemas de acción y las reglas rebatibles. La figura 5.6 muestra el contenido del archivo *limpieza.pl* que define el dominio $(\Delta_{3.2}, \Gamma_{3.3})$.

```
%------------------------Definición de acciones Γ3.3---------------------------
%def_action(Nombre, Efectos_pos, Efectos_neg, Precondiciones, Restricciones).

def_action(subir_pers,[pers_alta],[],[∼pers_alta],[]).
def_action(bajar_pers,[∼pers_alta],[],[pers_alta],[]).
def_action(encender(I),[enc(I)],[],[int(I),∼enc(I)],[not peligroso(I)]).
def_action(apagar(I),[∼enc(I)],[],[int(I),enc(I)],[not peligroso(I)]).
def_action(cambiar(L,I,R),[con(I,R)],[-con(I,L)],[rep(R),con(I,L),∼enc(I)],[]).
def_action(limpiar,[oficina_limpia],[],[luz],[]).

%------------------------Conocimiento rebatible Δ3.2----------------------------
luz -< luz_natural.
luz -< luz_artificial.
luz_natural -< pers_alta, es_dia.
luz_artificial -< enc(L), lamp(L).
peligroso(I) -< int(I), mojado(I).
peligroso(I) -< int(I), roto(I).
∼peligroso(I) -< int(I), ∼electricidad.
enc(L) -< con(I,L), enc(I).
∼enc(L) -< lamp(L), ∼electricidad.
∼enc(L) -< lamp(L), rota(L).
rep(L) -< lamp(L), not en_uso(L).
en_uso(L) -< con(_,L).
∼rep(L) -< lamp(L), rota(L).
%------------------------------------------------------------------------------
```

Figura 5.6: Archivo *limpieza.pl* con la descripción del dominio de planificación $(\Delta_{3.2}, \Gamma_{3.3})$

Para invocar el planificador sobre un problema particular es necesario consultar en el interprete prolog el archivo *apop.pl* que contiene el módulo *Planificador* y luego

consultar el archivo *limpieza.pl* con la descripción del dominio. A partir del predicado `find_plan(I,M,R)` se pueden obtener planes para un problema definido sobre un dominio consultado previamente. Los argumentos `I`, `M` y `R` son listas que contienen los literales presentes en el estado inicial, las metas y las restricciones del problema respectivamente. La figura 5.7 muestra como invocar el planificador desde el intérprete Prolog para el problema $(\Psi, \Delta_{3.2}, \Gamma_{3.3}, Meta, \mathsf{C})$ definido sobre el dominio $(\Delta_{3.2}, \Gamma_{3.3})$, donde:

- $\Psi = \{\sim pers\_alta, int(i\_1), \sim enc(i\_1), lamp(l\_1), rota(l\_1), con(i\_1, l\_1), lamp(l\_2)\}$
- $Meta = \{oficina\_limpia\}$
- $\mathsf{C} = \{not\ luz\_artificial\}$

En el estado inicial del problema, la persiana de la oficina se encuentra baja, hay dos lámparas l_1 y l_2, y existe un solo interruptor i_1 que se encuentra apagado y conectado a la lámpara l_1 que está rota. La meta del problema consiste en dejar la oficina limpia, con la restricción de que no quede la luz encendida.

```
?- consult(apop).
yes
?- consult(limpieza).
yes
?- find_plan([~pers_alta,int(i_1),~enc(i_1),lamp(l_1),rota(l_1),con(i_1,l_1),
              lamp(l_2)], [oficina_limpia], [not luz_artificial]).
```

Figura 5.7: Invocando el planificador desde el intérprete Prolog

La figura 5.8 muestra el contenido de los conjuntos *Acc* y *Arg* que son generados por el planificador para el problema $(\Psi, \Delta_{3.2}, \Gamma_{3.3}, Meta, \mathsf{C})$ antes de iniciar la búsqueda de un plan. El planificador elimina todas las restricciones del problema siguiendo la transformación definida en sección 3.4.1, es por esto que las acciones presentes en *Acc* no contienen restricciones. Por ejemplo, en la acción $apagar(i\_1)$ la restricción $not\ peligroso(i\_1)$ es reemplazada por la precondición p_c_3, y se agrega la regla $p\_c\_3 \prec not\ peligroso(i\_1)$ con la cual se construye el argumento `arg5` del conjunto *Arg*. A su vez, también se elimina la restricción del problema $not\ luz\_artificial$, agregándose una nueva meta p_c_1 y la regla $p\_c\_1 \prec not\ luz\_artificial$ con la cual se construye el argumento `arg4`.

En las figura 5.9 se puede observar la interfaz gráfica del *Visualizador* donde se muestra un plan obtenido para el problema $(\Psi, \Delta_{3.2}, \Gamma_{3.3}, Meta, \mathsf{C})$. La notación gráfica utilizada

$$Acc = \left\{ \begin{array}{c} \{\sim pers\_alta\}, \{\}^- \overset{bajar\_pers}{\longleftarrow} \{pers\_alta\}, not\ \{\} \\ \{\sim enc(i\_1)\}, \{\}^- \overset{apagar(i\_1)}{\longleftarrow} \{int(i\_1), enc(i\_1), p\_c\_3\}, not\ \{\} \\ \{con(i\_1, l\_1)\}, \{con(i\_1, l\_2)\}^- \overset{cambiar(l\_2,i\_1,l\_1)}{\longleftarrow} \{rep(l\_1), con(i\_1, l\_2), \sim enc(i\_1)\}, not\ \{\} \\ \{oficina\_limpia\}, \{\}^- \overset{limpiar}{\longleftarrow} \{luz\}, not\ \{\} \\ \{pers\_alta\}, \{\}^- \overset{subir\_pers}{\longleftarrow} \{\sim pers\_alta\}, not\ \{\} \\ \{enc(i\_1)\}, \{\}^- \overset{encender(i\_1)}{\longleftarrow} \{int(i\_1), \sim enc(i\_1), p\_c\_2\}, not\ \{\} \\ \{con(i\_1, l\_2)\}, \{con(i\_1, l\_1)\}^- \overset{cambiar(l\_1,i\_1,l\_2)}{\longleftarrow} \{rep(l\_2), con(i\_1, l\_1), \sim enc(i\_1)\}, not\ \{\} \end{array} \right\}$$

$$Arg = \left\{ \begin{array}{rl} \texttt{arg0} = & \langle \{(luz \prec luz\_artificial), (luz\_artificial \prec enc(l\_1), lamp(l\_1)), \\ & (enc(l\_1) \prec con(i\_1, l\_1), enc(i\_1))\}, luz \rangle \\ \texttt{arg1} = & \langle \{(luz \prec luz\_artificial), (luz\_artificial \prec enc(l\_2), lamp(l\_2)), \\ & (enc(l\_2) \prec con(i\_1, l\_2), enc(i\_1))\}, luz \rangle \\ & \langle \{(luz\_artificial \prec enc(l\_1), lamp(l\_1)), \\ & (enc(l\_1) \prec con(i\_1, l\_1), enc(i\_1))\}, luz\_artificial \rangle \\ \texttt{d4} = & \langle \{(luz\_artificial \prec enc(l\_2), lamp(l\_2)), \\ & (enc(l\_2) \prec con(i\_1, l\_2), enc(i\_1))\}, luz\_artificial \rangle \\ & \langle \{(enc(l\_1) \prec con(i\_1, l\_1), enc(i\_1))\}, enc(l\_1) \rangle \\ & \langle \{(enc(l\_2) \prec con(i\_1, l\_2), enc(i\_1))\}, enc(l\_2) \rangle \\ \texttt{d0} = & \langle \{(\sim enc(l\_1) \prec lamp(l\_1), rota(l\_1))\}, \sim enc(l\_1) \rangle \\ & \langle \{(rep(l\_1) \prec lamp(l\_1), not\ en\_uso(l\_1))\}, rep(l\_1) \rangle \\ \texttt{arg3} = & \langle \{(rep(l\_2) \prec lamp(l\_2), not\ en\_uso(l\_2))\}, rep(l\_2) \rangle \\ & \langle \{(en\_uso(l\_1) \prec con(i\_1, l\_1))\}, en\_uso(l\_1) \rangle \\ & \langle \{(en\_uso(l\_2) \prec con(i\_1, l\_2))\}, en\_uso(l\_2) \rangle \\ & \langle \{(\sim rep(l\_1) \prec lamp(l\_1), rota(l\_1))\}, \sim rep(l\_1) \rangle \\ \texttt{arg4} = & \langle \{(p\_c\_1 \prec not\ luz\_artificial)\}, p\_c\_1 \rangle \\ \texttt{arg2} = & \langle \{(p\_c\_2 \prec not\ peligroso(i\_1))\}, p\_c\_2 \rangle \\ \texttt{arg5} = & \langle \{(p\_c\_3 \prec not\ peligroso(i\_1))\}, p\_c\_3 \rangle \end{array} \right\}$$

Figura 5.8: Conjuntos Acc y Arg generados para el problema $(\Psi, \Delta_{3.2}, \Gamma_{3.3}, Meta, \mathsf{C})$

por el *Visualizador* difiere mínimamente con la notación utilizada en el resto de la tesis. En esta notación simplificada, los efectos de los pasos de acción y la conclusión de los pasos de argumento no aparecen dibujados sobre el paso. Los vínculos causales y de soporte relacionan un literal directamente con el paso que lo sustenta. Por otra parte, solo se

muestran aquellos literales que están sustentados por algún vínculo causal o de soporte.

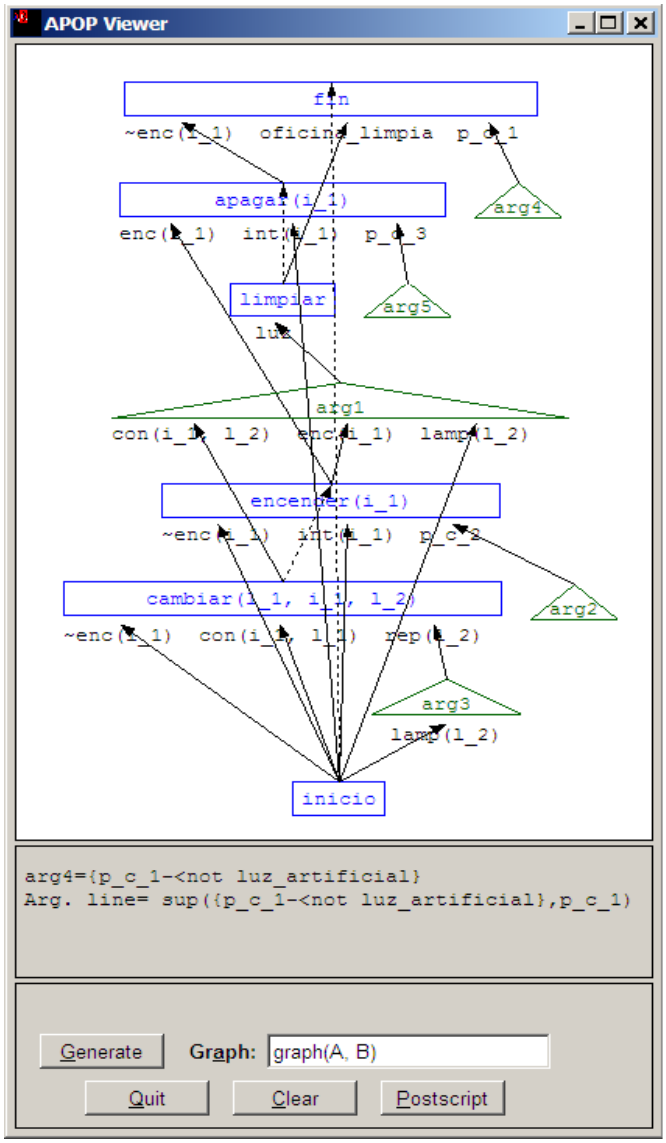

Figura 5.9: Interfaz gráfica del Visualizador

El *Planificador* provee diferentes modos ejecución que permiten monitorear y controlar diferentes aspectos del proceso de planificación, como: la fase de expansión, la elección de pasos del plan y la resolución de amenazas. Además, el *Generador de Grafos* y el *Vizualizador* pueden ser invocados por el *Planificador* en cualquier instancia del proceso de planificación (aun cuando el plan este incompleto), lo cual permite visualizar la construcción de un plan paso por paso.

En las figuras 5.10, 5.11, 5.12, 5.13 y 5.14 se puede observar una secuencia con alguno de los planes que son mostrados por el *Vizualizador* durante la búsqueda de un plan para el problema $(\Psi, \Delta_{3.2}, \Gamma_{3.3}, Meta, \mathsf{C})$. El planificador comienza construyendo el plan inicial que se muestra en la figura 5.10(a). Luego selecciona del conjunto *Acc* la acción *limpiar* para lograr la meta $oficina\_limpia$, obteniendo el plan de la figura 5.10(b).

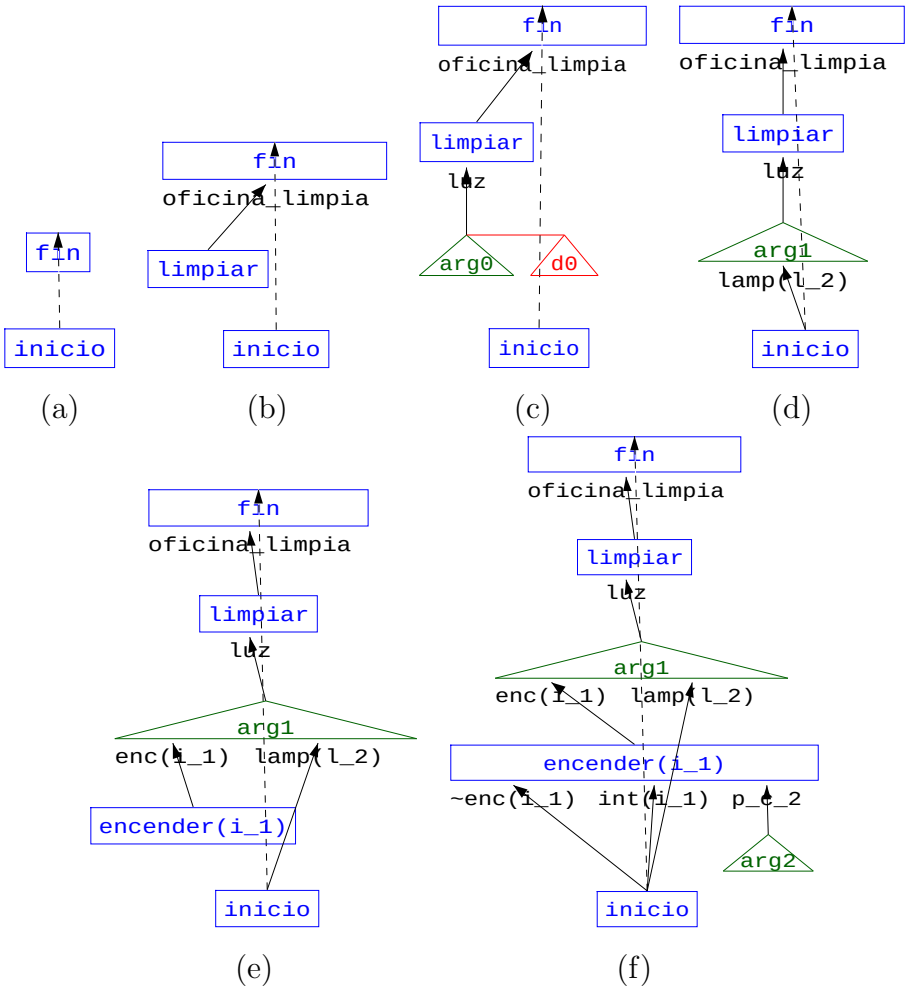

Figura 5.10: Traza de la construcción de un plan: Parte I

Para sustentar la precondición *luz* del paso *limpiar* existen dos argumentos en el conjunto *Arg*:

$$\begin{aligned}
\texttt{arg0} = \; & \langle\{(luz \prec luz_artificial), (luz_artificial \prec enc(l_1), lamp(l_1)), \\
& (enc(l_1) \prec con(i_1, l_1), enc(i_1))\}, luz\rangle \\
\texttt{arg1} = \; & \langle\{(luz \prec luz_artificial), (luz_artificial \prec enc(l_2), lamp(l_2)), \\
& (enc(l_2) \prec con(i_1, l_2), enc(i_1))\}, luz\rangle
\end{aligned}$$

Al elegir el argumento `arg0` se obtiene el plan de la figura 5.10(c), donde existe una *amenaza argumento-argumento* producida por la activación del argumento `d0`=$\langle\{(\sim enc(l\_1) \prec lamp(l\_1), rota(l\_1))\}, \sim enc(l\_1)\rangle$ que ataca al argumento `arg0`. El planificador no puede resolver esta amenaza, por lo tanto selecciona el argumento `arg1` para sustentar la precondición *luz* y obtiene el plan que se muestra en la figura 5.10(d) que no presenta amenazas. Luego, se selecciona la acción $encender(i_1)$ para sustentar el literal $enc(i_1)$ en la base de `arg1`, obteniendo el plan que se muestra en la figura 5.10(e). Las precondiciones $\sim enc(i\_1)$ y $int(i\_1)$ del paso $encender(i\_1)$ son logradas por la acción *inicio* por lo tanto el planificador agrega los vínculos causales correspondientes. Para lograr la precondición `p_c_2` del paso $encender(i\_1)$ se elige el argumento `arg2`=$\langle\{(p\_c\_2 \prec not\ peligroso(i\_1))\}, p\_c\_2\rangle$ del conjunto Arg obteniendo el plan que se muestra en la figura 5.10(f).

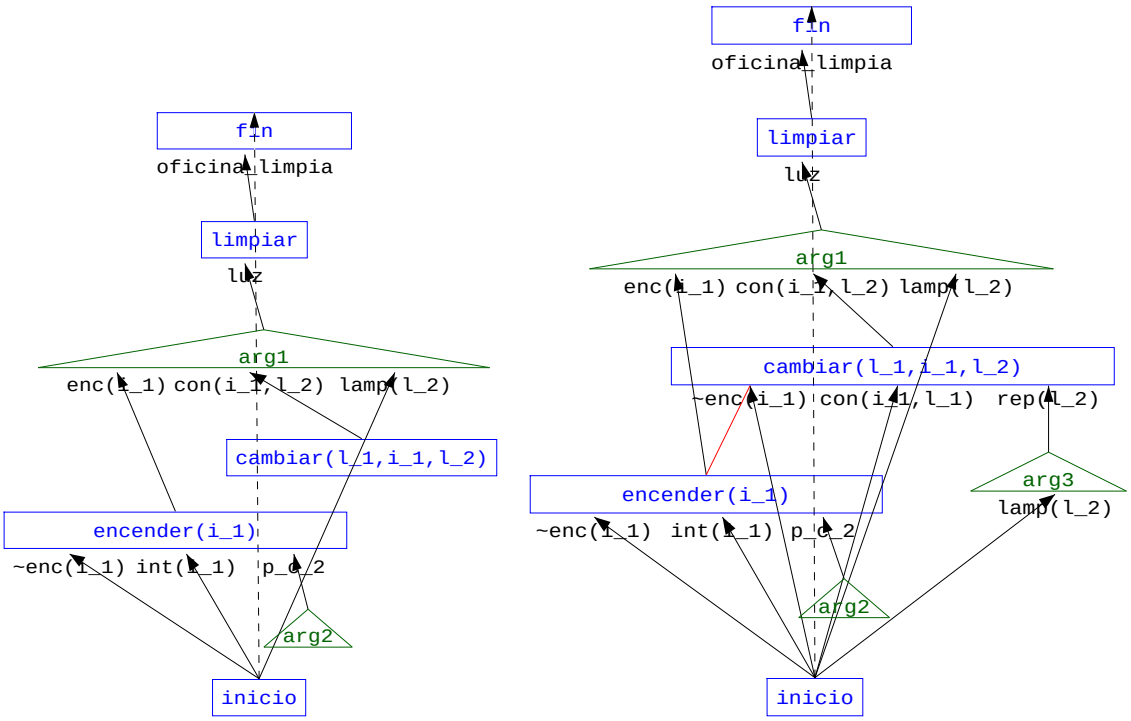

Figura 5.11: Traza de la construcción de un plan: Parte II

Para sustentar el literal `con(i_1,l_2)` en la base del argumento `arg1` el planificador elige acción $cambiar(l\_1, i\_1, l\_2)$ obteniendo el plan de la figura 5.11(g). Luego, agrega el argumento `arg3`=$\langle\{(rep(l\_2) \prec lamp(l\_2), not\ en\_uso(l\_2))\}, rep(l\_2)\rangle$ para sustentar la precondición rep(l_2) del paso `cambiar(l_1,i_1,l_2)` y los vínculos

causales correspondientes para sustentar las precondiciones ∼enc(i_1) y con(i_1,l_1) obteniendo el plan de la figura 5.11(h). Note que el el paso encender(i_1) tiene enc(i_1) como efecto y por lo tanto constituye una *amenaza acción-acción* para el vínculo causal inicio $\overset{\sim enc(i\_1)}{\longrightarrow}$ cambiar(l_1, i_1, l_2). Par resolver esta amenaza el planificador aplica el *método avance*, incorporando la restricción de orden encender(i_1) $\prec$ cambiar(l_1,i_1,l_2) y obtiene el plan que se muestra en la figura 5.12(i)

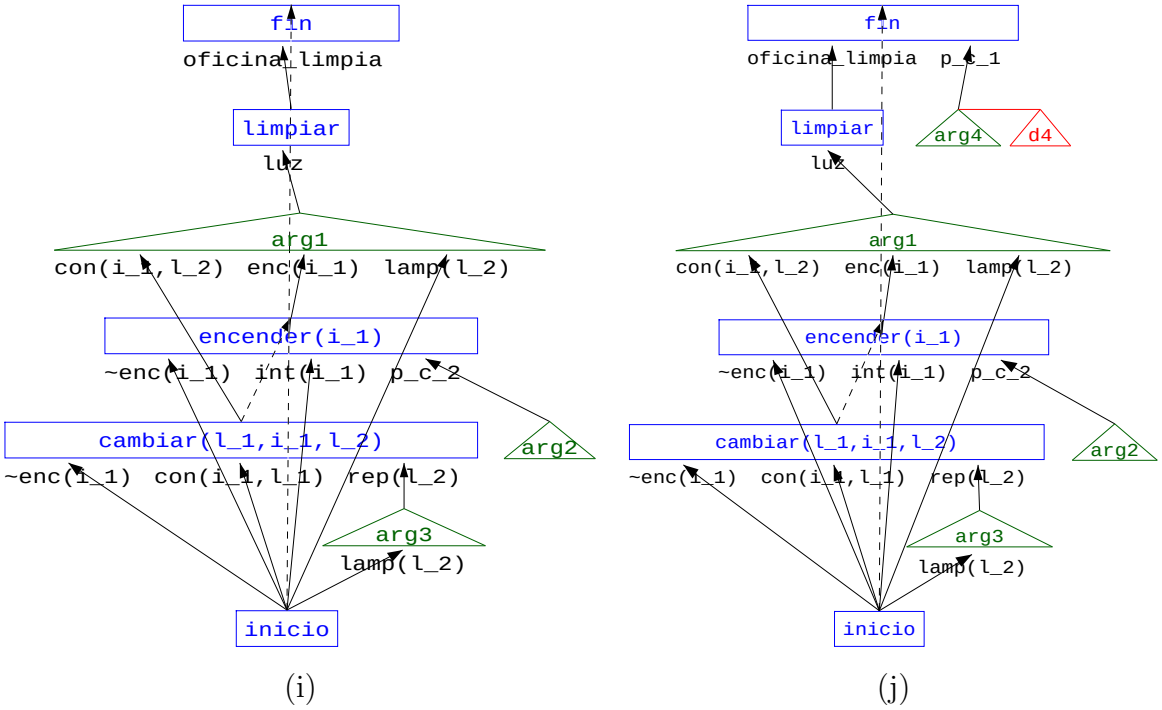

Figura 5.12: Traza de la construcción de un plan: Parte III

Todavía está pendiente la meta p_c_1 del paso fin. Para lograr esta meta el planificador incorpora el argumento arg4=$\langle\{(p\_c\_1 \prec not\ luz\_artificial)\}, p\_c\_1\rangle$ obteniendo el plan de la figura 5.12(j), el cual presenta una *amenaza argumento-argumento* dado que el argumento

$$\begin{aligned} \texttt{d4} = \quad & \langle\{(luz\_artificial \prec enc(l\_2), lamp(l\_2)), \\ & (enc(l\_2) \prec con(i\_1, l\_2), enc(i\_1))\}, luz\_artificial\rangle \end{aligned}$$

ataca a la suposición *not luz_artificial* del argumento arg4. Para solucionar esta amenaza el planificador elige deshabilitar el literal enc(i_1) agregando la meta ∼enc(i_1) al paso fin. Luego, incorpora la acción apagar(i_1) para lograr ∼enc(i_1) y obtiene el plan de la figura 5.13(k). Note que el paso encender(i_1) es una *amenaza acción-acción* para el vínculo causal apagar(i_1) $\overset{\sim enc}{\longrightarrow}$ *fin*. Luego, el planificador aplica el método *retroceso*,

incorporando la restricción de orden `encender(i_1)`$\prec$`apagar(i_1)` y obtiene el plan que se muestra en la figura 5.13(l).

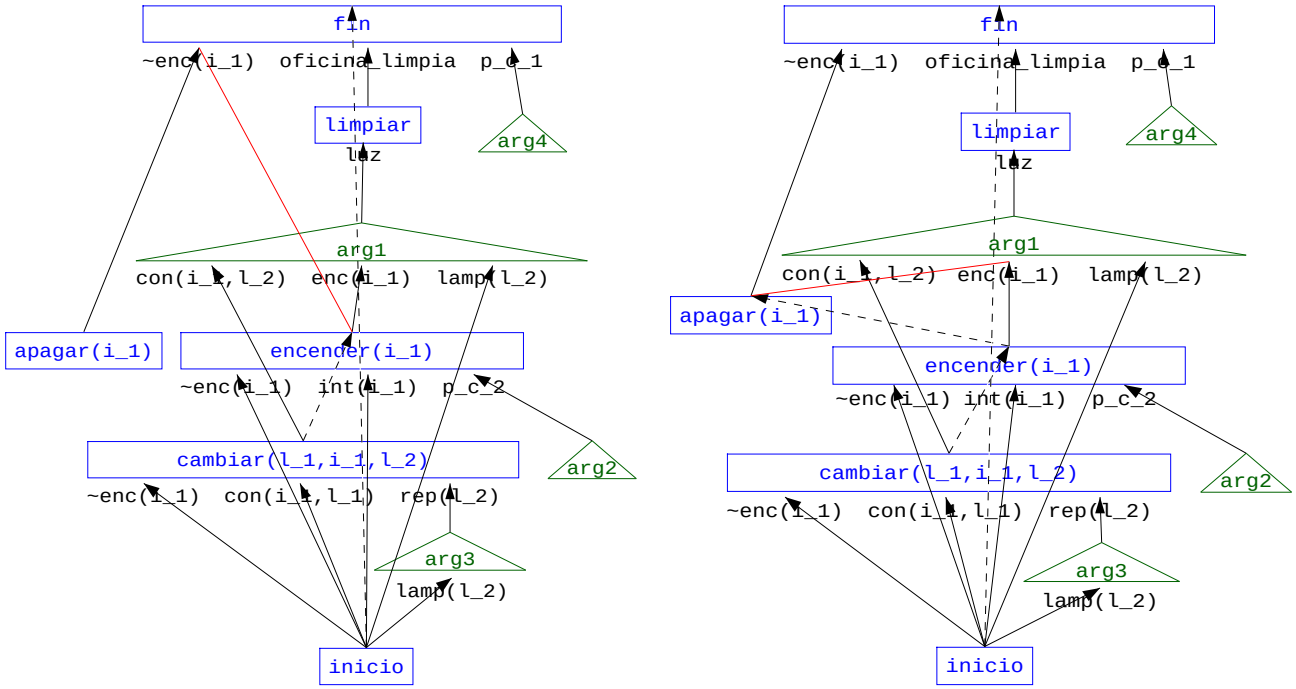

Figura 5.13: Traza de la construcción de un plan: Parte IV

En el plan de la figura 5.13(l), el paso `apagar(i_1)` es una *amenaza acción-base* para el vínculo de soporte `encender(i_1)` $\overset{\texttt{enc(i\_1)}}{\succ\!\!-}$ `arg1`, ya que `apagar(i_1)` tiene $\sim$`enc(i_1)` como efecto. Para solucionar esta amenaza el planificador aplica el *método avance**, incorporando la restricción de orden `limpiar`$\prec$ `apagar(i_1)` y obtiene el plan de la figura 5.13(m).

Para finalizar, se agrega el argumento `arg5`$=\langle\{(p\_c\_3 \prec \textit{not peligroso}(i\_1))\}, p\_c\_3\rangle$ para sustentar la precondición `p_c_3` del paso `apagar(i_1)` y se incorporan los vínculos causales `encender(i_1)` $\overset{\texttt{enc(i\_1)}}{\longrightarrow}$ `apagar(i_1)` y `inicio` $\overset{\texttt{int(i\_1)}}{\longrightarrow}$ `apagar(i_1)` para lograr el resto de las precondiciones. De esta forma, se obtiene el plan de la figura 5.13(n) que soluciona el problema.

Recuerde que, según la definición 4.8, un *plan parcial argumentativo* P es una solución para un problema de planificación DAKAR, si cada secuencia de acciones obtenida de un orden topológico de P es una solución para el problema. En este caso, el plan de la figura 5.13(n) tiene un solo orden topológico del cual se obtiene la siguiente secuencia

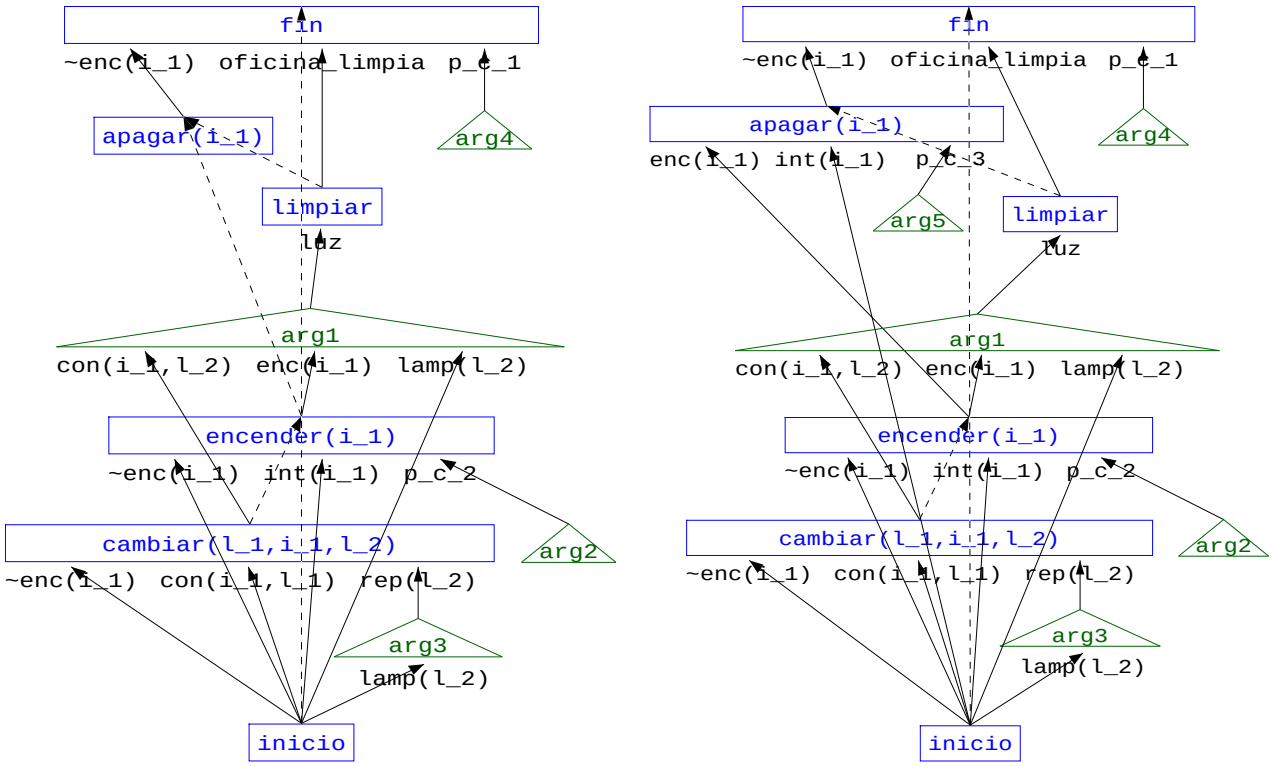

Figura 5.14: Traza de la construcción de un plan: Parte final

aplicable de acciones:

$$S = [cambiar(l\_1, i\_1, l\_2),\ encender(i\_1),\ limpiar,\ apagar(i\_1)]$$

Al ejecutar la secuencia S en el estado inicial del problema Ψ, se obtiene el estado

$$\begin{aligned}\Psi^{\overleftarrow{S}} = \quad & \{\sim pers\_alta, int(i\_1), \sim enc(i\_1), lamp(l\_1), rota(l\_1),\\ & con(i\_1, l\_2), lamp(l\_2), oficina\_limpia\}\end{aligned}$$

donde se cumple tanto la meta $oficina\_limpia$ como la restricción $not\, luz\_artificial$ del problema $(\Psi, \Delta_{3.2}, \Gamma_{3.3}, Meta, \mathsf{C})$. Note que $oficina\_limpia$ está garantizado a partir del programa $(\Psi^{\overleftarrow{S}}, \Delta_{3.2})$ porque $oficina\_limpia \in \Psi^{\overleftarrow{S}}$, y $luz\_artificial$ no está garantizado a partir de $(\Psi^{\overleftarrow{S}}, \Delta_{3.2})$, ya que no es posible construir un argumento para $luz\_artificial$ porque $\sim enc(i\_1) \in \Psi^{\overleftarrow{S}}$.

Capítulo 6

Trabajos Relacionados

En este capítulo se describen algunos de los lenguajes y formalismos de representación de acciones, dominios y problemas de planificación existentes en la literatura. El objetivo es presentar sus principales características para realizar una comparación con el formalismo de representación DAKAR. Dado que el objetivo principal de esta tesis no es mejorar la eficiencia de los métodos de planificación existentes, no se realizarán comparaciones desde este punto de vista. Además, se describirán algunas propuestas existentes que combinan argumentación y planificación y, en este sentido, se compararán con el enfoque propuesto en esta tesis.

6.1. Introducción

Los problemas de planificación siempre han sido de gran interés para el área de Inteligencia Artificial, y a lo largo de las últimas décadas se han desarrollado numerosos enfoques para representar y resolver estos problemas. En general estos enfoques pueden dividirse en dos grandes grupos: los enfoques *deductivos* y los enfoques *procedurales*. Los enfoques deductivos utilizan la lógica para modelar y resolver problemas de planificación, mientras que los enfoques procedurales son un híbrido entre la computación procedural y la lógica.

Si bien los primeros enfoques propuestos fueron deductivos [Gre69, MH69], perdieron interés rápidamente, principalmente por su ineficiencia y su incapacidad de resolver el conocido *problema del marco* (en inglés, *frame problem*), esto es, el problema de especificar

que hechos no cambian cuando se ejecuta una acción. Como alternativa para resolver este problema, la mayoría de los enfoques surgidos posteriormente fueron procedurales. Por otra parte, el problema del marco fue uno de los primeros ejemplos de razonamiento *default* que motivaron el desarrollo de las lógicas no-monótonas, que luego impulsaron el desarrollo de nuevos enfoques deductivos.

El primero de los enfoques procedurales fue STRIPS [FN71], extendido luego por el lenguaje ADL [Ped89], para incorporar efectos condicionales y cuantificación universal sobre las precondiciones y efectos de las acciones. Por otra parte, se desarrollaron numerosos métodos y algoritmos para resolver problemas de planificación definidos en estos lenguajes, entre los que se destacan POP [PW92] y *Graphplan* [BF97]. En el año 1998, en el marco de la competencia de planificación desarrollada en la conferencia AIPS'98 (Artificial Inteligent Planning Systems), surge el lenguaje PDDL (Planning Domain Definition Languaje) [GHK$^+$98] como una estandarización del lenguaje ADL. El objetivo era contar con un lenguaje común para poder evaluar empíricamente el desempeño de diferentes planificadores y desarrollar un conjunto de problemas estándar. Este lenguaje fue desarrollado, y es extendido constantemente, por los grupos de investigación más activos del área de planificación e incorpora características de diferentes sistemas de planificación.

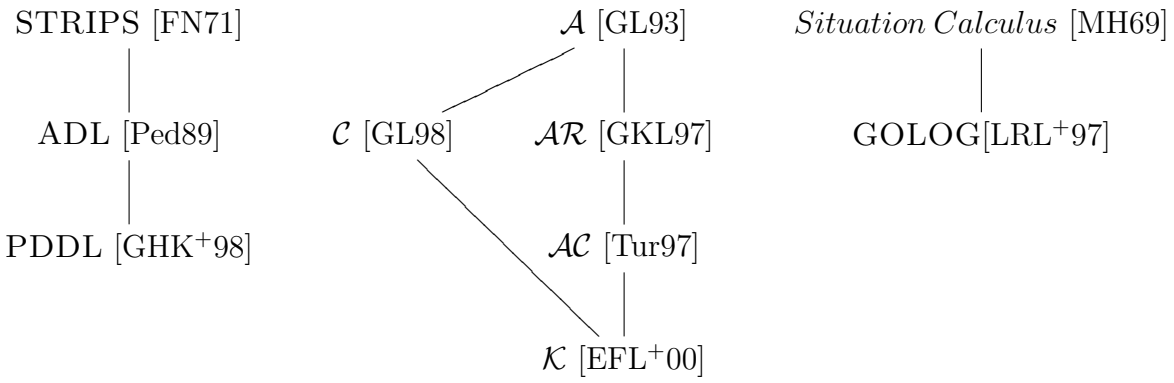

Figura 6.1: Evolución de los lenguajes y formalismos de representación de acciones

En los últimos años los enfoques deductivos han cobrado interés nuevamente, surgiendo diferentes líneas de trabajo, entre las cuales se destacan:

- Distintas soluciones al *problema del marco* permitieron el surgimiento de nuevos enfoques deductivos basados en el *Cálculo de Situaciones* (*Situation Calculus* [MH69]), como el lenguaje de programación en lógica para entornos dinámicos GOLOG [LRL$^+$97].

- El uso de la lógica y la programación en lógica para representar conocimiento y razonar sobre las acciones y el cambio, dió origen a una familia de lenguajes de representación de alto nivel basados en lógica. El primero de estos lenguajes, denominado $\mathcal{A}$, fue definido en [GL93] donde se describe un método para traducir un dominio de planificación descripto en $\mathcal{A}$ en un programa lógico extendido [GL91] y se prueba que esta traducción es sensata con respecto a la semántica de $\mathcal{A}$. Siguiendo esta metodología, el lenguaje $\mathcal{A}$ ha sido extendido dando origen a nuevos lenguajes de alto nivel para formalizar acciones en diferentes formalismos lógicos. En [KL94, GKL97] se define el lenguaje $\mathcal{AR}$, una extensión de $\mathcal{A}$ que permite describir condiciones de imposibilidad (esto es, cuando una acción no puede ser ejecutada), efectos indirectos de las acciones y formas simples de no determinismo. Además, se define una traducción de $\mathcal{AR}$ en un formalismo basado en *circunscripción* [McC80]. El lenguaje $\mathcal{AR}$ fue extendido luego por el lenguaje $\mathcal{C}$ [GL98] y el lenguaje $\mathcal{AC}$ definido en [Tur97], donde se presenta una traducción sensata y completa de $\mathcal{AC}$ a la *lógica default* [Rei80]. Una de las extensiones más completas y que combina características de todos estos lenguajes es el lenguaje $\mathcal{K}$, definido en [EFL$^+$00]. Una de las características distintivas de $\mathcal{K}$ es que permite utilizar negación por falla, lo cual resulta adecuado para representar problemas de planificación con conocimiento incompleto. El lenguaje $\mathcal{K}$ fue implementado sobre el sistema de programación en lógica disyuntiva `DLV` [ELM$^+$98], dando origen al sistema de planificación `DLV`$^{\mathcal{K}}$ [EFL$^+$03]. Esta implementación se logra a través de una transformación que permite traducir un problema de planificación P descripto en $\mathcal{K}$, en un programa lógico disyuntivo $lp(P)$, de forma que los conjuntos de respuesta (en inglés, *answer sets*) de $lp(P)$ contienen la solución de P.

- En [KS92] se propuso formular un problema de planificación como un problema de satisfabilidad de una formula lógica. Esto dió origen a Blackbox [KS99], un eficiente sistema de planificación que reduce un problema de planificación del tipo STRIPS (especificado en PDDL) a un problema de satisfabilidad, que luego resuelve utilizando varios métodos de resolución de satisfabilidad (en inglés, *satisfability solvers*). Siguiendo este mismo espíritu, se desarrollaron métodos para reducir a un problema de satisfabilidad, problemas de planificación descriptos en otros lenguajes de representación existentes. Por ejemplo, en [MT98] se implementa el *cálculo de situaciones* y en [Giu00] se implementa el lenguaje $\mathcal{C}$.

Para realizar una comparación de DAKAR con otros lenguajes y formalismos de representación, se eligió un representante de cada enfoque. En la sección 6.2 se describe el lenguaje PDDL, por ser el más completo y expresivo de los enfoques procedurales. Luego, en la sección 6.3 se introduce el lenguaje $\mathcal{K}$ que reúne características de muchos lenguajes que siguen el enfoque deductivo. En la sección 6.4 se realizará una comparación basada principalmente en aquellos aspectos que son relevantes en DAKAR como aporte de esta tesis.

Por otra parte, ninguna de las propuestas mencionadas anteriormente ni los lenguajes de representación existentes considera el uso de argumentación. De hecho, muy pocos investigadores han abordado el tema de incorporar argumentación al proceso de planificación y las propuestas existentes difieren considerablemente con el enfoque propuesto en esta tesis. En la sección 6.5 se describirán brevemente alguna de estas propuestas.

6.2. Lenguaje PDDL

Desde la aparición del lenguaje STRIPS en los años setenta, se han introducido diferentes extensiones con el objetivo de mejorar la eficiencia y la expresividad, e incrementar el alcance de los planificadores a un conjunto más grande de dominios de planificación. El lenguaje PDDL (Planning Domain Definition Languaje) [GHK+98] fue introducido en 1998 para su utilización dentro de la competencia de planificación desarrollada en la conferencia AIPS'98 (Artificial Inteligent Planning Systems). El objetivo era contar con un lenguaje común para poder evaluar empíricamente el desempeño de diferentes planificadores y desarrollar un conjunto de problemas standard.

Este lenguaje fue desarrollado conjuntamente por los grupos de investigación más activos del área de planificación e incorpora características de diferentes sistemas de planificación. Por lo tanto, PDDL puede verse como un compromiso entre diferentes características y representaciones sintácticas presentes en los sistemas de planificación actuales más importantes.

En principio PDDL es una estandarización del lenguaje STRIPS, sin embargo su poder expresivo se extiende aún más. Las extensiones más relevantes son:

- Estructuras de tipos para los objetos del dominio, variables y parámetros de las acciones

- Restricciones de igualdad sobre los parámetros de las acciones y las variables
- Efectos condicionales en las acciones
- Precondiciones disyuntivas y negativas
- Cuantificación universal sobre los efectos y precondiciones de las acciones

Estas extensiones son básicamente las propuestas en el lenguaje ADL [Ped89]. La mayoría de los planificadores modernos están basados en STRIPS y soportan las primeras tres extensiones. Mientras que las primeras cuatro extensiones principalmente mejoran la efectividad en la construcción de un plan, la última extensión permite expandir la clase de dominios para la cual se pueden construir planes.

La sintaxis de PDDL está inspirada en lenguaje funcional Lisp, por lo tanto la estructura del lenguaje es una lista de expresiones parentizadas. El lenguaje permite definir de manera separada la descripción de las acciones, que caracterizan el comportamiento del dominio de planificación, de la descripción de los objetos, el estado inicial y las metas que caracterizan un problema en particular. De esta forma, un problema de planificación se describe juntado la descripción del dominio con la descripción del problema.

Para introducir brevemente la estructura del lenguaje, se presentará a continuación un ejemplo del mundo de bloques definido en PDDL.

Ejemplo 6.1 Considere el dominio del *mundo de bloques* descripto en el ejemplo 2.1. La figura 6.2 muestra como se define este dominio en PDDL.

La definición comienza dando un nombre al dominio, en este caso `mundo_de_bloques`. El campo `:requirements` especifica qué requerimientos o características expresivas (por ejemplo: *efectos condicionales*, *cuantificación universal*) tiene el dominio para que cada planificador pueda determinar rápidamente si puede manejarlo. En este caso, el requerimiento `:strips` indica que la definición del dominio sólo utiliza las caracteristicas expresivas convencionales del lenguaje de representación STRIPS.

El campo `:predicates` permite definir los predicados básicos que se utilizan en la descripción del dominio. La definición de un predicado básico consta de un nombre y una lista de variables, que se distinguen por estar precedidas por el símbolo "`?`". Los predicados se utilizan para decribir propiedades o relaciones entre los objetos del dominio.

```
(define (domain mundo_de_bloques)
  (:requirements :strips)
  (:predicates (sobre_mesa ?x)
               (sobre ?x ?y)
               (libre ?x)
  )
  (:action apilar
     :parameters (?x ?y)
     :precondition (and (sobre_mesa ?x) (libre ?x) (libre ?y))
     :effect (and (not (libre ?y)) (not (sobre_mesa ?x)) (sobre ?x ?y))
  )
  (:action desapilar
     :parameters (?x ?y)
     :precondition (and (sobre ?x ?y) (libre ?x))
     :effect (and (not (sobre ?x ?y)) (sobre_mesa ?x) (libre ?y))
  )
)
```

Figura 6.2: Definición del mundo de bloques en PDDL

Por ejemplo, el predicado `(sobre ?x ?y)` se utiliza para representar que un bloque `x` se encuentra sobre un bloque `y`.

Por último se definen las acciones disponibles en el dominio. El campo `:action` permite definir el nombre de la acción y el campo `:parameters` permite definir los parámetros de la acción. Los campos `:precondition` y `:effect` permiten definir las precondiciones y efectos de la acción respectivamente. Por tratarse de un dominio con las caracteristicas expresivas de STRIPS, las precondiciones y los efectos se definen simplemente mediante una conjunción de literales.

Una vez definido un dominio de planificación se pueden definir problemas particulares sobre ese dominio.

Ejemplo 6.2 Considere el problema definido en el ejemplo 2.5, que consiste en construir una pila con tres bloques que se encuentran inicialmente sobre la mesa. La figura 6.3 muestra como definir este problema en PDDL, utilizando el dominio definido en la figura 6.2.

```
(define (problem torre3)
   (:domain mundo_de_bloques)
   (:objects a b c)
   (:init (libre a) (libre b) (libre c)
          (sobre_mesa a) (sobre_mesa b) (sobre_mesa c)
   )
   (:goal (and (sobre b a) (sobre c b)))
)
```

Figura 6.3: Definición de un problema de planificación del mundo de bloques en PDDL

Al igual que un dominio, un problema de planificación recibe un nombre, en este caso `torre3`. El campo `:domain` especifica el nombre del dominio de planificación sobre el cual se define el problema. El campo `:objects` menciona los objetos presentes en el problema. En este caso, existen tres objetos (`a`,`b` y `c`) que representan los bloques. Como veremos más adelante, es posible definir tipos para los objetos. Si no se especifica ningún tipo sobre un objeto, se asume que es de un tipo predefinido llamado `object`. El campo `:init` describe el estado inicial del mundo, mencionando los literales que son verdaderos en dicho estado. Por último, el campo `:goal` define las metas del problema por medio de una conjunción de literales.

En lo que sigue de esta sección se describirá más detalladamente la estructura sintáctica y las características más relevantes del lenguaje PDDL. Dado que este lenguaje está en constante cambio y evolución, no se presentará en esta tesis una descripción completa del mismo.

6.2.1. Dominios de planificación en PDDL

La definición de un dominio de planificación en PDDL tiene la siguiente estructura:

```
<domain> ::= ( define ( domain <name>)
             [( :requirements <require-key>+)]
             [( :types <typed list (name)>)]
             [( :constants <typed list (name)>)]
             [( :predicates <atomic formula>+)]
```

```
[( :functions <atomic function>+)]
[( :constraints <con-GD>)]
<structure-def>*)
```

(domain `<name>`**)** define el nombre del dominio. El nombre del dominio (como cualquier aparición de `<name>`) consiste de una cadena de caracteres que comienza con un letra y puede contener letras, dígitos y los símbolos "-" y "_".

El campo **(:requirements** `<require-key>`$^+$**)** permite especificar que requerimientos o características expresivas tiene el dominio para que cada planificador pueda determinar rápidamente si puede manejarlo. Los requerimientos se especifican utilizando palabras reservadas predefinidas, esto es: `<require-key> ::=` **strips | typing** ...

El campo **(:types** `<typed list (name)>`**)** permite definir nuevos tipos presentes en el dominio. La lista tipada (`<typed list (name)>`) se usa para declarar el tipo de una lista de entidades.

```
<typed list (x)> ::= x*
<typed list (x)> ::= x+ - <type> <typed list(x)>
<primitive-type> ::= <name>
<type> ::= (either <primitive-type>+)
<type> ::= <primitive-type>
```

Los tipos están precedidos por el símbolo "-" y cualquier otro elemento de la lista es declarado del primer tipo que sigue a continuación, o del tipo predefinido `object` si no hay ninguno. Por ejemplo: **:types** `entero real - number bloque`, define tres nuevos tipos. Los primeros dos son subclases de `number` y el último es una subclase de `object`. Esto es, cada entero es un número, cada real es un número y cada bloque es un objeto.

Un tipo primitivo es simplemente un nombre (`<name>`). Además existen tipos compuestos de la forma **(either** $\mathtt{t}_1 \ldots \mathtt{t}_k$**)**, que representa la unión de los tipos $\mathtt{t}_1$ a $\mathtt{t}_k$.

El campo **(:constants** `<typed list (name)>`**)** tiene la misma sintaxis que el campo **:types**, pero el significado es diferente. Es este caso los nombres declaran nuevas constantes del dominio, que se asocian a un tipo de los mencionados, de la misma forma que se indicó anteriormente.

El campo **(:predicates** `<atomic formula>+`**)** permite definir predicados, denominados *básicos*, que se utilizarán en la descripción del dominio para describir propiedades o

relaciones entre los objetos del mismo. La definición de un predicado consta de un nombre y una lista tipada de variables, que se distinguen por estar precedidas por el símbolo "?".

```
<atomic formula> ::= (<predicate> <typed list (variable)>)
<predicate> ::= <name>
<variable> ::= ?<name>
```

El campo (**:functions** `<atomic function>+`) permite declarar funciones cuyo valor es de tipo numérico (`number`). La definición de una función es similar a la de un predicado:

```
<atomic function> ::= (<function-symbol> <typed list (variable)>)
<function-symbol> ::= <name>
```

Las funciones permiten definir acciones con efectos que usan operadores de asignación y precondiciones aritméticas.

Los campos restantes de la definición del dominio:

```
<structure-def> ::= <action-def>
<structure-def> ::= <durative-action-def>
<structure-def> ::= <derived-def>
```

permiten definir las acciones y los predicados derivados, y serán desarrollados en las siguientes secciones.

6.2.2. Acciones en PDDL

Las acciones se representan básicamente como los operadores en el lenguaje STRIPS, mediante una descripción de sus *precondiciones* y *efectos*. La definición de una acción en PDDL tiene la siguiente estructura:

```
<action-def> ::= (:action <name>
                  :parameters (<typed list (variable)>)
                  [:precondition <pre-GD>]
                  [:effect <effect>])
```

La definición de una acción comienza dando un nombre a la misma a través de `:action <name>`. El campo `:parameters` permite definir los parámetros de la acción y especificar su tipo, a través de una lista tipada de variables. Los parámetros deben ser instanciados con objetos presentes en un problema de planificación particular para poder aplicar la acción. Para poder instanciar una acción, los tipos de los objetos deben coincidir con los tipos de los parámetros.

El campo **:precondition** permite especificar las precondiciones de la acción a través de la descripción de una meta, que debe ser satisfecha en un estado para que la acción pueda ser ejecutada. Note que este campo es opcional, si la definición de una acción no especifica ninguna precondición entonces siempre es ejecutable.

La definición de las precondiciones consiste básicamente en la descripción de una meta `<GD>` (*goal description*), con la posibilidad de definir *restricciones blandas* (en inglés *soft constraints*) sobre precondiciones que no sean disyuntivas.

```
<pre-GD> ::= <GD>
<pre-GD> ::= (and <pre-GD> <pre-GD>+)
<pre-GD> ::= (forall (<typed list(variable)>) <pre-GD>)
<pre-GD> ::= <pref-GD>
<pref-GD> ::=(preference [<name>] <GD>)
```

Una meta etiquetada con **preference** indica que la meta no necesariamente debe ser verdadera para satisfacer la precondición correspondiente. El nombre asignado `<name>` puede utilizarse para definir métricas de calidad del plan (ver `<metric-espec>` en la sección 6.2.4). En general, un plan se considera mejor cuando más preferencias satisface.

La descripción de una meta (`<GD>`) en PDDL es bastante expresiva. Es posible especificar una sentencia lógica de primer orden arbitraria (incluyendo cuantificadores anidados) que no contenga funciones, con la posibilidad de utilizar negación por falla (**not** `<GD>`).

```
<GD> ::= <atomic formula(term)>
<GD> ::= <literal(term)>
<GD> ::= (and <GD>*)
<GD> ::= (or <GD>*)
<GD> ::= (not <GD>)
<GD> ::= (imply <GD> <GD>)
```

<GD> ::= (**exists** (<typed list(variable)>) <GD>)
<GD> ::= (**forall** (<typed list(variable)>) <GD>)
<GD> ::= <f-comp>

<literal(t)> ::= <atomic formula(t)>
<literal(t)> ::= (**not** <atomic formula(t)>)
<atomic formula(t)> ::= (<predicate> t^*)
<term> ::= <name>
<term> ::= <variable>

Por medio de <f-comp> se pueden comparar expresiones aritméticas usando los operadores de comparación habituales.

<f-comp> ::= (<binary-comp> <f-exp> <f-exp>)
<binary-comp> ::= > | < | = | >= | <=

El campo **:effect** permite definir los efectos de la acción, esto es, permite describir de que forma se modifica un estado del mundo cuando la acción es ejecutada.

<effect> ::= (**and** <c-effect> <c-effect>$^+$)
<effect> ::= <c-effect>
<c-effect> ::= (**forall** (<typed list (variable)>*) <effect>)
<c-effect> ::= (**when** <GD> <cond-effect>)
<c-effect> ::= <p-effect>
<p-effect> ::= (<assign-op> <f-head> <f-exp>)
<p-effect> ::= <literal(term)>
<cond-effect> ::= (**and** <p-effect> <p-effect>$^+$)
<cond-effect> ::= <p-effect>
<assign-op> ::= **assign | scale-up | scale-down | increase | decrease**

Todas las variables presentes en la definición de una acción deben aparecer en la lista de parámetros de la acción o deben estar ligadas a un cuantificador (**exists** o **forall**). Todas variables libres que aparezcan en el campo **:effect** deben aparecer ligadas a un cuantificador en el campo **:precondition**.

La semántica de la ejecución de una acción es similar a la de una acción STRIPS, esto es, las acciones describen la transición de un estado a otro en el espacio de estados.

A continuación se describirá de manera simplificada como se aplican los efectos de una acción a un estado S para producir un estado S'. Una definición más detallada de la semántica de una acción PDDL puede encontrarse en [FL03].

Definición 6.1 (Sustitución de variables en PDDL) Sea D un dominio de planificación y P un problema de planificación (ver sección 6.2.4) en PDDL. Una *sustitución* σ es un conjunto de elementos de la forma $\sigma = \{$`(?var`$_1$`, obj`$_1$`)`,..., `(?var`$_n$`, obj`$_n$`)`$\}$ donde:

- `?var`$_i$ es una variable definida en D,
- `obj`$_i$ es un objeto definido en P del mismo tipo que `?var`$_i$ y
- `?var`$_i \neq$`?var`$_j$ si $i \neq j$

Definición 6.2 (Aplicación de una sustitución a una acción en PDDL) La aplicación de una sustitución σ a una acción a, denotado $\sigma(a)$, se obtiene a partir de a reemplazando todas las apariciones de cada parámetro `?var` de a por el objeto `obj`, donde `(?var, obj)` $\in \sigma$.

Definición 6.3 (Aplicación de una sustitución a un efecto en PDDL) La aplicación de una sustitución σ a un efecto `<effect>` denotado σ(`<effect>`), se obtiene reemplazando todas las apariciones de cada variable libre `?var` presente en `<effect>` por el objeto `obj`, donde `(?var, obj)` $\in \sigma$.

Dada un acción a, cada sustitución σ de parámetros por objetos produce una instancia particular $\sigma(a)$ de la acción a. Si un estado S satisface la precondición de $\sigma(a)$, la ejecución de $\sigma(a)$ en S producirá un nuevo estado S' que resulta de aplicar los efectos de $\sigma(a)$ en S de la siguiente forma:

1. Los efectos atómicos de la forma `<literal(term)>` son aplicados:
   - *agregando* `<literal(term)>` a S, si `<literal(term)>` es una *fórmula atómica positiva*
   - *quitando* `<atomic formula(term)>` de S, si `<literal(term)>` es una *fórmula atómica negativa* de la forma **not** `<atomic formula(term)>`.
2. Los *efectos condicionales* de la forma **(when** `<GD> <cond-effect>`**)** son aplicados a S, aplicando `<cond-effect>` a S sólo si S satisface `<GD>`.

3. Los efectos de la forma (**forall** (<typed list (variable)>*) <effect>) son aplicados a S encontrando todas las posibles sustituciones σ_i de las variables en <typed list (variable)> por objetos, luego aplicando $\sigma \cup \sigma_i$(<effect>) a S.

4. Los efectos de la forma (<assign-op> <f-head> <f-exp>) son aplicados a S cambiando el valor de la función <f-head> por el valor resultante de evaluar la expresión <f-exp>, teniendo en cuenta el operador de asignación <assign-op>. En la figura 6.6 se define una acción `verter` que permite pasar el contenido de un jarra a otra y utiliza este tipo de efectos.

5. Los efectos de la forma (**and** <c-effect> <c-effect>$^+$) son aplicados a S, aplicando cada uno de los efectos <c-effect> a S.

A continuación se presentan unos ejemplos de acciones que muestran diferentes tipos de efectos.

Ejemplo 6.3 La figura 6.4 muestra una definición alternativa a la planteada en la figura del 6.2 para el mundo de bloques, que utiliza *tipos, restricciones de igualdad y efectos condicionales*. En este caso la mesa se modela como una constante que siempre está libre y existe una única acción `mover` que permite mover un bloque `?x` desde un bloque hacia otro, desde la mesa hacia un bloque o desde un bloque hacia la mesa. Estas alternativas son modeladas a través de efectos condicionales. Note que sólo el parámetro `?x` debe ser de tipo `bloque` y los efectos de la acción varían dependiendo del tipo de `?y` e `?z` :

- si `?y` es un bloque (**not** `(= ?y Mesa)`) entonces `?y` estará libre luego de mover `?x` desde `?y` hacia `?z`
- si `?z` es un bloque (**not** `(= ?z Mesa)`) entonces `?z` no estará libre luego de mover `?x` desde `?y` hacia `?z`.

Ejemplo 6.4 En la figura 6.5 se presenta una acción para el dominio del mundo de bloques definido en la figura 6.4 que utiliza cuantificación universal en los efectos. Esta acción permite mover hacia la mesa, *todos* los bloques libres `?x` que se encuentren sobre otro bloque `?y`.

Ejemplo 6.5 La figura 6.6 muestra un dominio de planificación sencillo que utiliza funciones para modelar una acción que permite verter el contenido de una jarra a otra. Las

```
(define (domain mundo_de_bloques)
  (:requirements :strips :typing :equality :conditional-effects)
  (:types bloque)
  (:constants Mesa)
  (:predicates (sobre ?x ?y)
               (libre ?x)
  )
  (:action mover
     :parameters (?x -bloque ?y ?z)
     :precondition (and (sobre ?x ?y) (libre ?x) (libre ?z)
                        (not (= ?x ?y)) (not (= ?y ?z)) (not (= ?z ?x)))
     :effect (and (not (sobre ?x ?y)) (sobre ?x ?z)
                  (when (not (= ?y Mesa))(libre ?y))
                  (when (not (= ?z Mesa))(not (libre ?z)))
             )
  )
)
```

Figura 6.4: Definición del mundo de bloques en PDDL usando tipos, restricciones de igualdad y efectos condicionales

```
 (:action topes_a_la_mesa
  :parameters ()
  :precondition ()
  :effect (forall (?x ?y -bloque)
                  (when (and (clear ?x)(on ?x ?y))
                         (and not (on ?x ?y) (on ?x Mesa))
                  )
          )
 )
```

Figura 6.5: Una acción del mundo de bloques con cuantificación universal

funciones `(cantidad ?j - jarra)` y `(capacidad ?j - jarra)` permiten representar la cantidad de líquido que contiene una jarra y su capacidad respectivamente. La acción `verter` permite pasar el contenido de un jarra `j1` a otra jarra `j2` siempre y cuando la capacidad de la jarra `j2` menos su contenido (esto es, la capacidad restante) sea mayor que el contenido de la jarra `j1`.

```
(define (domain liquidos)
 (:requirements :typing :fluents)
 (:types jarra)
 (:functions
     (cantidad ?j - jarra)
     (capacidad ?j - jarra)
 )
 (:action verter
    :parameters (?j1 ?j2 - jarra)
    :precondition (>= (- (capacidad ?j2) (cantidad ?j2)) (cantidad ?j1))
    :effect (and (assign (cantidad ?j1) 0)
                 (increase (cantidad ?j2) (cantidad ?j1)))
 )
)
```

Figura 6.6: Dominio del ejemplo 6.5

6.2.3. Predicados derivados

Como se vió en la sección 6.2.1 la definición de un dominio de planificación PDDL permite definir *predicados básicos*, para describir relaciones entre los objetos del dominio. El valor de verdad de un predicado básico depende del estado particular del mundo. Los predicados presentes en la descripción de un estado se consideran verdaderos y los predicados que no aparecen se consideran falsos. Por otra parte, los efectos de las acciones modifican el estado del mundo agregando o quitanto predicados básicos y por lo tanto modifican el valor de verdad de los mismos.

Además de los predicados básicos, PDDL permite definir *predicados derivados*. Estos predicados no son afectados por los efectos de las acciones, sino que su valor de verdad en un estado se infiere de otros predicados a través de ciertas reglas. La definición de un predicado derivado cuenta de una o más reglas con la siguiente estructura:

```
<derived-def> ::= (:derived <predicate> <typed list (variable)> <GD>)
<predicate> ::= <name>
```

Luego, en la definición de un dominio de planificación PDDL se pueden distinguir dos conjuntos disjuntos de símbolos de predicados $\mathcal{B}$ y $\mathcal{D}$, que representan los predicados básicos y derivados respectivamente. Los símbolos presentes en $\mathcal{D}$ no pueden aparecer en

la descripción del estado inicial ni en los efectos de las acciones, sólo pueden aparecer en las precondiciones de las acciones y en la descripción de una meta `<GD>`.

Las reglas que definen los predicados derivados son esencialmente sentencias de *programas lógicos estratificados* [ABW88]. En una regla de la forma `(:`**derived** $(d? \vec{x})(f? \vec{x})$`)`, $d \in \mathcal{D}$ y f es una fórmula de primer orden construida con símbolos presentes en $\mathcal{B} \cup \mathcal{D}$, cuyas variables libres son las mencionadas en el vector $\vec{x}$.

Intuitivamente, una regla `(:`**derived** $(d? \vec{x})(f? \vec{x})$`)` significa que cuando $(f? \vec{x})$ es verdadero con ciertos argumentos en un estado, se puede derivar que $(d? \vec{x})$ es verdadero con los mismos argumentos en dicho estado. A diferencia de las implicaciones lógicas tradicionales, estas derivaciones no pueden ser contrapuestas (la negación de f no puede derivarse de la negación de d) y los consecuentes de las reglas no pueden ser predicados negados. Todos los predicados que no son derivados como verdaderos se asumen falsos (*Hipótesis de mundo cerrado*).

Con fin el obtener una semántica simple para los predicados derivados, se impuso la restricción de que el conjunto de reglas debe estar *estratificado*. La idea básica de la estratificación consiste en evitar que un predicado dependa recursivamente de su negación: un grupo de predicados derivados estará definido en función de los predicados básicos (posiblemente negados) o en función de ellos mismos (recursivamente) pero sin usar negación. Luego, un grupo más abstracto de predicados estará definido en función de los predicados anteriores (posiblemente negados) o en función de ellos mismos sin usar negación, y así sucesivamente.

Ejemplo 6.6 Consideremos nuevamente el dominio del mundo de bloques definido en 6.4. En este dominio se definen dos predicados básicos: `(libre ?x)`, para representar que un bloque `?x` está libre, y `(sobre ?x ?y)` para representar que un bloque `?x` está sobre un bloque `y`. Por medio de un predicado derivado `(encima ?x ?y)` podemos definir recursivamente la clausura transitiva de `(sobre ?x ?y)` de la siguiente forma:

```
(:derived (encima ?x ?y)
      (or (sobre ?x ?y)
          (exists (?z) (and  (sobre ?x ?z) (encima ?z ?y)))
      )
)
```

Es decir, un bloque `?x` *está encima de* un bloque `?y` si el bloque `?x` se encuentra sobre un bloque `?y`, o `?x` está sobre otro bloque `?z`, y `?z` *está encima de* `?y`.

También es posible definir el predicado básico `(libre ?x)` como un predicado derivado en función del predicado básico `(sobre ?x ?y)` de la siguiente forma:

```
(:derived (libre ?x)
      (forall (?y) (not  (sobre ?y ?x)))
)
```

Esto es, Un bloque `?x` está libre si no hay ningún bloque sobre él.

Los predicados derivados aumentan el poder expresivo del lenguaje [THN05], en el sentido que permiten representar dominios del mundo real de una forma más concisa y elegante. En general, estos dominios requieren definir condiciones complejas, que son más fáciles de construir jerárquicamente a partir de condiciones elementales de los estados del mundo. Sin predicados derivados, estas condiciones deben modelarse a través de las precondiciones y los efectos de las acciones, los cuales crecen considerablemente y se vuelven poco claros. La situación empeora más aún cuando se quiere introducir una nueva acción al dominio. En [THN05] se demuestra que modelar los predicados derivados a través de las acciones produce una explosión exponencial en el tamaño de la descripción del dominio ó en la longitud del plan más corto.

6.2.4. Problemas de planificación en PDDL

Una vez definido un dominio de planificación en PDDL se pueden definir problemas particulares sobre ese dominio. La definición de un problema tiene la siguiente estructura:

```
<problem> ::= (define (problem <name>)
              (:domain <name>)
              [:requirements <require-key>]
              [(:objects <typed list (name)>]
              (:init <init-el>*)
              (:goal <pre-GD>)
              [(:constraints <pref-con-GD>)]
              [(:metric <optimization> <metric-f-exp>>]
```

```
 [(:length [(serial <number>)][(:parallel <number>]
)
```

La definición comienza dando un nombre al problema y luego en el campo **:domain** se especifica el nombre del dominio de planificación sobre el cual se define el problema.

El campo **:requirements**, al igual que en la definición de un dominio, permite especificar los requerimientos expresivos que tiene el problema. En general un problema hereda los requerimentos del dominio sobre el cual se define. Sin embargo, hay casos en los que el estado inicial o la meta tienen requerimentos expresivos distintos a los mencionados en el dominio del problema.

El campo **:objects** permite definir los objetos presentes en el problema y el tipo de cada uno. Si no se especifica ningún tipo sobre un objeto, se asume que es de un tipo predefinido llamado **object**. Estos objetos no deben estar declarados en el campo **:constants** del dominio sobre el cual se define el problema.

El estado inicial del problema se define en el campo `(:init <init-el>*)`, donde:

```
<init-el> ::= <atomic formula(name)>
<init-el> ::=(= <f-head> <number>)
<init-el> ::=(at <number> <literal(name)>)
<f-head> ::= (<function-symbol> <term>*)
<f-head> ::= <function-symbol>
```

La definición del estado inicial consiste en una lista de literales fijos positivos y funciones con su correspondiente valor. Los literales que aparecen en la definición del estado se consideran verdaderos y los literales que no son mencionados se asumen falsos. Mediante expresiones de la forma `(= <f-head> <number>)` se establece el valor de las funciones en el estado inicial.

Las expresiones de la forma `(at <number> <literal(name)>)` permiten especificar *literales iniciales temporizados* (en inglés, *timed initial literales*), esto es, literales que se que harán verdaderos o falsos en determinado instante de tiempo `<number>` relativo al estado inicial (esto es, instante de tiempo 0). Estos literales son dados a conocer al planificador con anticipación y no dependen de las acciones que éste elija. Luego, los literales temporizados pueden verse como eventos externos incondicionales y deterministas.

El campo **:goal** permite definir las metas del problema de de igual forma que las precondiciones de una acción a través de `<pre-GD>`. (ver sección 6.2.2).

El campo (**:constraints** `<pref-con-GD>`) permite definir restricciones que debe cumplir un plan para solucionar el problema. Esto es, condiciones que deben cumplirse por todos los estados visitados durante la ejecución del plan. Estas condiciones son expresadas a través de operadores modales aplicados a formulas de primer orden sobre predicados del problema de planificación (`<GD>`).

El campo (**:metric** `<optimization> <metric-f-exp>`) permite definir métricas de calidad para los planes, donde `<metric-f-exp>` es una expresión aritmética y `<optimization>` indica si la calidad del plan depende de minimizar o maximizar el resultado de dicha expresión.

Como se vió en la sección 6.2.2, una meta etiquetada de la forma:

preference `<pref-name> <GD>`,

indica que la meta `<GD>` no necesariamente debe ser satisfecha, pero un plan se considerará mejor si logra satisfacerla. La penalización por la violación de una preferencia se define usando la expresión:

`<metric-f-exp> ::=` (**is-violated** `<pref-name>`),

donde `<pref-name>` puede estar asociado a una o más preferencias. Esta expresión recibe un valor igual al número de preferencias con ese nombre que *no* son satisfechas por el plan.

Por último, el campo (**:length** `[`(**serial** `<number>`)`][`(**:parallel** `<number>`) permite declarar la existencia de un plan de cierta longitud que soluciona el problema. Esto resulta útil para aquellos planificadores que buscan la solución en función de la longitud.

6.3. Lenguaje $\mathcal{K}$

El uso de la lógica y la programación en lógica para representar conocimiento y razonar sobre las acciones y el cambio, dió origen a una familia de lenguajes de representación de alto nivel basados en lógica [GL93, KL94, Tur97, GKL97, GL98]. El lenguaje $\mathcal{K}$[EFL$^+$00]

combina características de todos estos lenguajes, siendo el más expresivo y completo de esta familia.

El lenguaje fue nombrado $\mathcal{K}$ porque permite describir transiciones entre *estados de conocimiento* (en inglés, *Knowledge states*), en contraste con otros lenguajes de esta familia que describen transiciones entre *estados del mundo*. Los estados del mundo son caracterizados por el valor de verdad de un conjunto de fluentes (en inglés *fluents*), esto es, predicados que describen propiedades del dominio y cuyo valor puede variar de un estado a otro. En un estado del mundo el valor de cada fluente es verdadero o es falso. El lenguaje $\mathcal{K}$ en cambio, adopta tres valores de verdad para los fluentes: falso, verdadero o desconocido. Esta característica, en combinación con el uso de negación por falla hacen que resulte adecuado para modelar problemas de planificación con información incompleta.

6.3.1. Sintaxis del lenguaje

La signatura del lenguaje $\mathcal{K}$ consiste de tres conjuntos disjuntos y no vacíos $\mathbf{A}$,$\mathbf{F}$,$\mathbf{T}$ de nombres de acciones, fluentes y tipos respectivamente. Un átomo de acción (respectivamente de fluente, de tipo) se define como $p(X_1, \ldots, X_n)$ donde $p \in \mathbf{A}$ (respectivamente $p \in \mathbf{F}$, $p \in \mathbf{T}$) y $X_1, \ldots, X_n$ son identificadores de variables o constantes. Como es usual, las variables se distinguen de las constantes por que las primeras comienzan con un letra mayúscula.

Un literal de acción (respectivamente de fluente o de tipo) es un átomo a de acción, (respectivamente de fluente o de tipo), o su *negación* $\neg a$, donde "$\neg$" es el símbolo de negación fuerte (también representado por "$-$"). Si l es un literal $\neg.l$ denota su complemento con respecto a la negación fuerte, esto es, $\neg.l = \neg a$ si $l = a$ y $\neg.l = a$ si $l = \neg\, a$, donde a es un átomo. Dado un conjunto de literales L, se utilizaran L^+ y L^- para distinguir los subconjuntos de literales positivos y negativos de L respectivamente. Además, $\mathcal{L}_{\mathbf{A}}$,$\mathcal{L}_{\mathbf{F}}$ y $\mathcal{L}_{\mathbf{T}}$ denotan los conjuntos de todos los literales de acciones, de fluentes y de tipos respectivamente, siendo $\mathcal{L} = \mathcal{L}^+_{\mathbf{A}} \cup \mathcal{L}_{\mathbf{F}} \cup \mathcal{L}_{\mathbf{T}}$ el conjunto de todos los literales [1].

Para describir un dominio de planificación, el lenguaje $\mathcal{K}$ cuenta básicamente con tres tipos de sentencias: *declaración de acciones o fluentes*, *reglas causales* y *condiciones de ejecución*.

[1] Note que sólo se permiten literales de acciones positivos.

Una *declaración de acción* (respectivamente de fluente) es un expresión de la forma:

$$p(X_1, \ldots, X_n) \textbf{ requires } t_1, \ldots, t_n \quad (1)$$

donde $p \in \mathbf{A}$ (resp. $p \in \mathbf{F}$), $X_1, \ldots, X_n$ son variables, $t_1, \ldots, t_n \in \mathcal{L}_\mathbf{T}$ son literales de tipo y cada X_i aparece en $t_1, \ldots, t_n$.

Este tipo de expresión permite definir características estáticas de las acciones y fluentes, como la cantidad y el tipo de sus argumentos. Por ejemplo, para el dominio del mundo de bloques se podría definir la regla

$$mover(X, Y) \textbf{ requires } bloque(X), bloque(Y)$$

para declarar que la acción de *mover* tiene dos argumentos que son de tipo *bloque*.

Una *regla causal* es una expresión de la forma:

$$\textbf{caused } f \textbf{ if } b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_l \textbf{ after } a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n \quad (2)$$

donde $f \in \mathcal{L}_\mathbf{F} \cup \{false\}$, $b_1, \ldots, b_l \in \mathcal{L}_\mathbf{F} \cup \mathcal{L}_\mathbf{T}$ y $a_1, \ldots, a_n \in \mathcal{L}$. Las reglas con $n = 0$ se denominan *estáticas*, sino se denominan *dinámicas*. Si $l = 0$ la palabra **if** pueden omitirse y de la misma forma puede omitirse **caused** cuando $n = 0$. Para hacer referencia a las distintas partes de una regla causal r se utilizará la siguiente notación: $h(r) = f$, $post^+(r) = \{b_1, \ldots, b_k\}$ $post^-(r) = \{b_{k+1}, \ldots, b_l\}$, $pre^+(r) = \{a_1, \ldots, a_m\}$ $pre^-(r) = \{a_{m+1}, \ldots, a_n\}$.

Intuitivamente una regla de la forma **caused** f **if** B **after** A, representa lo siguiente: *"si B es verdadero en el estado actual y A es verdadero en el estado previo entonces f es verdadero en el estado actual"*. Por ejemplo,

$$\textbf{caused } sobre(X, Y) \textbf{ after } mover(X, Y)$$

indica que si la acción $mover(X, Y)$ se ejecutó en el estado previo entonces $\neg sobre(X, Y)$ es verdadero en el estado actual.

Note que en una regla de la forma **caused** f **if** B **after** A, A puede contener literales de acciones, de fluentes y de tipos. Esto permite representar efectos condicionales, ya que la misma acción puede aparecer en varias reglas causales con distintos efectos acompañada de diferentes literales de fluentes y de tipos. Por ejemplo,

$$\textbf{caused } \neg sobre(X, Z) \textbf{ after } mover(X, Y), sobre(X, Z)$$

indica que si la acción $mover(X, Y)$ se ejecutó y $sobre(X, Z)$ era verdadero en el estado previo entonces $\neg sobre(X, Z)$ es verdadero en el estado actual.

En particular, una regla estática de la forma **caused** f **if** B significa que f es verdadero en cualquier estado donde B es verdadero. Por ejemplo,

$$\textbf{caused } ocupado(X) \textbf{ if } sobre(Y, X), bloque(X), bloque(Y)$$

indica que un bloque X esta ocupado, si hay otro bloque Y sobre X.

Si bien las reglas causales estáticas alcanzan a todos los estados, es posible definir reglas causales estáticas solo para el estado inicial precediéndolas con la palabra **initially**. Estas reglas se denominan *restricciones de estado inicial*. Por ejemplo, **initially** $sobre(a, b)$ indica que $sobre(a, b)$ es verdadero en el estado inicial.

Una *condición de ejecución* es una expresión de la forma:

$$\textbf{executable } a \textbf{ if } b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_l \quad (3)$$

donde $a \in \mathcal{L}_{\mathbf{A}}$ y $b_1, \ldots, b_l \in \mathcal{L}$. Para referenciar las distintas partes de una condición de ejecución e se utilizarán $h(e) = a$, $pre^+(r) = \{b_1, \ldots, b_k\}$ y $pre^-(r) = \{b_{k+1}, \ldots, b_l\}$.

Esta sentencia especifica las condiciones que deben cumplirse en un estado para que una acción pueda ejecutarse. Por ejemplo,

$$\textbf{executable } mover(X, Y) \textbf{ if } not\ ocupado(X), not\ ocupado(Y)$$

indica que la acción $mover(X, Y)$ puede ejecutarse si los bloques X e Y no están ocupados.

En particular, si $l = 0$ (no tiene condiciones) significa que la acción puede ejecutarse siempre y la palabra **if** puede omitirse. Note que las condiciones de ejecución pueden contener literales de acciones, lo cual permite modelar *acciones dependientes*, esto es acciones tales que para poder ejecutarse dependen de la ejecución de otras.

Observación 6.1 Todas las sentencias deben cumplir con la siguiente restricción sintáctica: todas las variables que aparecen en literales de tipos con negación por falla (esto es, precedidos por *not*) deben aparecer en un literal que no sea un literal de tipo con negación por falla. Esta restricción es similar a la noción de *seguridad* (*safety*) de los programas lógicos.

Una *descripción de acciones* en $\mathcal{K}$ es un par $\langle D, R\rangle$ donde D es un conjunto finito de declaraciones de acciones y fluentes, y R es un conjunto finito de reglas causales, restricciones de estado inicial y condiciones de ejecución. Luego, un *dominio de planificación* es un par $PD = \langle \Pi, AD\rangle$, donde Π es un programa Datalog [CGT89] estratificado (*stratified Datalog program*) que describe conocimiento estático (esto es, predicados que no son afectados por la acciones) y AD es una descripción de acciones. Por ejemplo, la figura 6.7 muestra un dominio de planificación $PD_b = \langle \Pi_b, \langle D_b, R_b\rangle\rangle$ para el mundo de bloques.

$$\Pi_b = \left\{ \begin{array}{l} bloque(a). \\ bloque(b). \\ lugar(mesa). \\ lugar(B) :- bloque(B). \end{array} \right\}$$

$$D_b = \left\{ \begin{array}{l} sobre(B, L) \textbf{ requires } bloque(B), lugar(L). \\ ocupado(B) \textbf{ requires } bloque(B). \\ mover(B, L) \textbf{ requires } bloque(B), lugar(L). \end{array} \right\}$$

$$R_b = \left\{ \begin{array}{l} \textbf{initially } sobre(a, mesa). \\ \textbf{initially } sobre(b, mesa). \\ \textbf{initially } sobre(c, mesa). \\ \textbf{caused } ocupado(X) \textbf{ if } sobre(Y, X), bloque(X), bloque(Y). \\ \textbf{caused } sobre(X, Y) \textbf{ after } mover(X, Y). \\ \textbf{caused } \neg sobre(X, Z) \textbf{ after } mover(X, Y), sobre(X, Z). \\ \textbf{executable } apilar(X, Y) \textbf{ if } not\ ocupado(X), not\ ocupado(Y). \\ \textbf{inertial } \text{sobre(B,L)}. \end{array} \right\}$$

Figura 6.7: Un dominio de planificación $PD_b = \langle \Pi_b, \langle D_b, R_b\rangle\rangle$ para el mundo de bloques

Sobre un dominio de planificación PD se puede definir un *problema de planificación* concreto como un par $\langle PD, q\rangle$ donde q es una consulta de la forma:

$$g_1, \ldots, g_m, not\ g_{m+1}, \ldots, not\ g_n?(i)$$

donde $g_1, \ldots, g_n$ son literales fijos de fluentes, $i \geq 0$ indica la longitud de plan. Por ejemplo, un problema para el dominio del mundo bloques PD_b se define como $\langle PD_b, q_b\rangle$ donde $q_b = sobre(c, b), sobre(b, a), sobre(a, mesa)?$.

La semántica de $\mathcal{K}$ se define para dominios de panificación fijos, esto es, que no contienen variables. Para cualquier dominio de planificación $PD = \langle \Pi, \langle D, R \rangle \rangle$ es posible obtener una *instancia fija tipada* reemplazando Π y R por sus versiones fijas Π' y R', manteniendo en R' aquellas reglas que contienen literales de acciones y de fluentes *legales*, esto es, que respetan las restricciones de tipos impuestas por las declaraciones en D. En lo que sigue, se considerarán solo dominios de planificación fijos. Para cualquier otro dominio de planificación los conceptos respectivos son definidos a través de su instancia fija tipada.

6.3.2. Semántica

Intuitivamente, un dominio de planificación PD definido en $\mathcal{K}$ describe un diagrama de transición cuyos nodos representan los posibles estados del dominio y cuyos arcos representan las acciones que llevan de un estado a otro. Los caminos del diagrama corresponden a todas las trayectorias posibles del dominio.

Consideremos la instancia fija tipada $PD = \langle \Pi, \langle D, R \rangle \rangle$ de un dominio de planificación donde M es el único conjunto de respuesta de Π. Un *estado* se define como un conjunto consistente de literales de fluentes presentes en PD. Un estado S_0 es una *estado inicial legal* para PD si S_0 es el conjunto mínimo (con respecto a la inclusión) tal que para cada restricción de estado inicial y cada regla estática c, si $post^+(c) \subseteq s_0 \cup M$ entonces $h(c) \in S_0$.

Un conjunto de acciones $A \subseteq \mathcal{L}^+_{\mathbf{A}}$ es *ejecutable* en un estado s si para cada $a \in A$ existe una condición de ejecución $e \in R$ tal que:

- $h(e) = a$,
- $pre(e)^+ \cap (\mathcal{L}_{\mathbf{F}} \cup \mathcal{L}_{\mathbf{T}}) \subseteq (s \cup M)$,
- $pre(e)^+ \cap \mathcal{L}^+_{\mathbf{A}} \subseteq A$ y
- $pre(e)^- \cap (s \cup A \cup M) = \emptyset$

Una *transición de estado* en PD, es una tupla $\langle s, A, s' \rangle$, donde s y s' son estados y A es un conjunto de literales de acción fijos legales presentes en PD. Una transición $\langle s, A, s' \rangle$ *es legal* en PD si A es ejecutable en s y s' es el mínimo conjunto consistente que satisface

todas las reglas causales con respecto a $s \cup A \cup M$. Esto es, para cada regla causal $r \in R$, si se cumplen las siguientes condiciones

- $pre(r)^+ \cap (\mathcal{L}_{\mathbf{F}} \cup \mathcal{L}_{\mathbf{T}}) \subseteq (s \cup M)$,
- $pre(r)^+ \cap \mathcal{L}^+_{\mathbf{A}} \subseteq A$,
- $pre(r)^- \cap (s \cup A \cup M) = \emptyset$,
- $post(r)^+ \subseteq (s' \cup M)$ y
- $post(r)^- \cap (s' \cup M) = \emptyset$

entonces $h(r) \neq false$ y $h(r) \in s'$.

A través del concepto de transición de estado, se puede formalizar el concepto de plan como una secuencia posible de transiciones de estado que lleve desde el estado inicial a un estado que satisfaga la meta.

Una secuencia de transiciones de estado $\langle\langle s_0, A_1, s_1\rangle, \langle s_1, A_2, s_2\rangle, \ldots, \langle s_{n-1}, A_n, s_n\rangle\rangle$ es una *trayectoria* para PD, si s_0 es un estado inicial legal y $\langle s_{i-1}, A, s_i\rangle$, es una *transición legal* en PD, para $1 \leq i \leq n$.

Dado un problema de planificación $\langle PD, q\rangle$ con $q = g_1, \ldots, g_m, not\, g_{m+1}, \ldots, not\, g_n?(i)$, una secuencia de conjuntos de acciones $\langle A_1, \ldots, A_i\rangle$ es un *plan optimista* para $\langle PD, q\rangle$, si existe una trayectoria $\langle\langle s_0, A_1, s_1\rangle, \langle s_1, A_2, s_2\rangle, \ldots, \langle s_{i-1}, A_i, s_i\rangle\rangle$ para PD tal que s_i satisface la meta, esto es, $\{g_1, \ldots, g_m\} \subseteq s_i$ y $\{g_{m+1}, \ldots, g_n\} \cap s_i = \emptyset$.

Esta noción de plan corresponde a la noción habitual de plan de la literatura. El término *optimista* se debe a que en dominios de planificación que proveen información incompleta del estado inicial o tienen acciones con efectos no-deterministas (esto es, producen transiciones de estados alternativas), la ejecución de este tipo de planes no garantiza que la meta sea alcanzada. Un plan que siempre garantiza la concreción de la meta (aun ante la presencia de incertidumbre) se denomina *plan seguro*, también conocido en la literatura como *conformant plan*.

Un plan optimista $\langle A_1, \ldots, A_n\rangle$ es un *plan seguro* si para cada estado inicial legal S_0 y para cada trayectoria $\langle\langle s_0, A_1, s_1\rangle, \ldots, \langle s_{j-1}, A_{j+1}, s_j\rangle\rangle$, con $0 \leq j \leq n$, se verifica que:

- si $j = n$ entonces s_j satisface la meta

- si $j < n$ entonces A_{j+1} es ejecutable en s_j, esto es, existe un transición legal $\langle s_j, A_j, s_{j+1}\rangle$

6.3.3. Sintaxis extendida

Si bien el lenguaje presentado en la sección 6.3.1 es completo y permite una representación compacta, su sintaxis ha sido extendida a través de la definición de algunos "macros" o "atajos notacionales" para que resulte más declarativo y amigable para el usuario.

A continuación se describirán algunos macros que permiten una representación más concisa y declarativa de algunos conceptos que se utilizan frecuentemente. En lo que sigue se utilizará $a \in \mathcal{L}^{+}_{\mathbf{A}}$ para denotar un átomo de acción, $f \in \mathcal{L}_{\mathbf{F}}$ para denotar un literal de fluente, B es una secuencia (posiblemente vacía) de la forma $b_1, \ldots, b_k, not\, b_{k+1}, \ldots, not\, b_l$, con $b_i \in \mathcal{L}_{\mathbf{F}} \cup \mathcal{L}_{\mathbf{T}}$ y A es una secuencia (posiblemente vacía) de la forma $a_1, \ldots, a_m, not\, a_{m+1}, \ldots, not\, a_n$, con $a_j \in \mathcal{L}$.

- **Inercia**. El lenguaje $\mathcal{K}$ da la posibilidad de declarar la propiedad de *"inercia"* para algunos fluentes, lo que significa que estos fluentes mantendrán su valor de verdad de un estado a otro en una transición, a menos que sean afectados explícitamente por una acción. Esta situación es conocida habitualmente como el *"problema del marco"* [MH69]. La mayoría de los lenguajes de representación adoptan la propiedad de inercia por defecto para todos fluentes que son efectos de acciones. En $\mathcal{K}$, se define el siguiente macro para declarar la propiedad de inercia para un fluente f

$$\textbf{inertial } f \textbf{ if } B \textbf{ after } A \;\Leftrightarrow\; \textbf{caused } f \textbf{ if } not\, \neg.f, B \textbf{ after } f, A$$

- **Defaults**. Un valor por defecto para un fluente f puede expresarse por medio de

$$\textbf{default } f \;\Leftrightarrow\; \textbf{caused } f \textbf{ if } not\, \neg.f$$

  el cual tendrá efecto a menos que exista evidencia para el valor opuesto de f, producido por alguna otra regla causal.

- **Fluentes totales**. El siguiente macro resulta útil para razonar con conocimiento incompleto pero total y permite representar acciones con efectos no deterministas.

$$\textbf{total } f \textbf{ if } B \textbf{ after } A \quad \Leftrightarrow \quad \begin{array}{l} \textbf{caused } f \textbf{ if } not\ \neg f, B \textbf{ after } A \\ \textbf{caused } \neg f \textbf{ if } not\ f, B \textbf{ after } A \end{array}$$

Una expresión de este tipo indica que tanto f como $\neg f$ pueden ser ciertos (no simultáneamente) si B es verdadero en el estado actual, A es verdadero en el estado previo y no hay evidencia en el estado actual que indique lo contrario.

- **Restricciones de estado**. A través del siguiente macro es posible definir restricciones para estados y su correspondiente estado previo.

$$\textbf{forbidden } B \textbf{ after } A \ \Leftrightarrow\ \textbf{caused } false \textbf{ if } B \textbf{ after } A$$

Una regla de este tipo indica que B no puede ser verdadero en ningún estado si A es verdadero en el estado previo.

- **Restricciones de ejecución**. El siguiente macro permite especificar cuando una acción no puede ejecutarse.

$$\textbf{nonexecutable } a \textbf{ if } B \ \Leftrightarrow\ \textbf{caused } false \textbf{ after } a, B$$

Esta expresión es opuesta **executable** a **if** ... y por la forma en que está definida constituye una restricción más fuerte.

En todos los macros tanto B como A pueden ser vacíos, en cuyo caso **if** B o **after** A pueden omitirse respectivamente.

6.4. Comparación de los formalismos

En esta sección se realizará una comparación de DAKAR con los lenguajes PDDL y $\mathcal{K}$, basada principalmente en un aspecto común a todos los lenguajes como es el uso de reglas para representar conocimiento en un dominio de planificación.

La característica distintiva de DAKAR es el uso de reglas rebatibles para representar conocimiento tentativo acerca del dominio. Los lenguajes PDDL y $\mathcal{K}$ también permiten representar conocimiento sobre el dominio, pero utilizan diferentes tipos de reglas con diferentes semánticas.

El lenguaje PDDL permite definir predicados derivados a través de un conjunto de reglas estratificado de la forma `(:derived` $(d? \vec{x})(f? \vec{x})$`)`, con las siguientes restricciones sintácticas:

- d es un predicado derivado y no puede estar negado
- f es una fórmula de primer orden construida con predicados básicos y derivados, cuyas variables libres son las mencionadas en el vector $\vec{x}$.
- d no puede aparecer en la descripción del estado inicial ni en los efectos de las acciones, sólo puede aparecer en las precondiciones de las acciones y en la descripción de una meta.

La tercera restricción asegura que las acciones no puedan afectar a los predicados derivados directamente. Esto permite obtener una semántica simple para los predicados derivados y evitar situaciones que son difíciles de tratar por los planificadores. Por ejemplo, si existe una regla `(:derived (r)(and p q))` y una acción que tiene `not r` como efecto se ejecuta en un estado donde `p` y `q` son verdaderos, no es claro cual debería ser el resultado.

La reglas rebatibles utilizadas en DAKAR no tienen estas restricciones. En una regla rebatible de la forma $L \prec B$, L es un literal (esto es, un átomo a o un átomo negado $\sim a$) que puede aparecer en el estado inicial y en cualquier parte de la descripción de una acción, incluidos los efectos positivos y negativos. Más aún, el efecto de una acción puede ser el complemento de L.

El conjunto de reglas utilizadas en PDDL para definir los predicados derivados constituye básicamente un *programa lógico estratificado* [ABW88], que permite derivar nuevos predicados a partir de los predicados básicos presentes en un estado del mundo. El valor de verdad de un predicado en un estado sólo puede ser verdadero o falso. Si un predicado pertenece al estado o puede ser derivado entonces es verdadero, en otro caso se asume falso (Hipótesis de mundo cerrado).

El conjunto Δ de reglas rebatibles de un dominio de planificación DAKAR, junto con la descripción de un estado del mundo Ψ, constituye un *programa lógico rebatible* (Ψ,Δ) a partir del cual se pueden garantizar nuevos literales. La noción de garantía es diferente a la noción de derivación de los programas lógicos. Para determinar si un literal está garantizado, no solo debe poder derivarse a partir del programa DeLP sino que además es necesario realizar un proceso de dialéctica que analice los argumentos a favor

y en contra del literal. Si L es un literal que representa una consulta para un programa DeLP, la respuesta a L puede ser

- SI: si L está garantizado
- NO: si $\overline{L}$ está garantizado
- INDECISO: si tanto L como $\overline{L}$ no están garantizados
- DESCONOCIDO: si L no pertenece a la signatura del programa.

Los predicados derivados de PDDL básicamente permiten representar condiciones que pueden definirse jerárquicamente en función de otros predicados básicos o derivados. Las reglas rebatibles permiten definir condiciones mucho más complejas que involucren razonamiento rebatible.

En el lenguaje $\mathcal{K}$ [EFL$^+$00] existen dos formas para representar conocimiento en un dominio de planificación $PD = \langle \Pi, AD \rangle$. Por un lado, el conjunto Π es un programa Datalog [CGT89] estratificado que, al igual que los predicados derivados de PDDL, permite describir conocimiento estático (esto es, predicados que no son afectados por la acciones). A diferencia de las reglas que definen los predicados derivados de PDDL, el cuerpo C de un regla p:-C en Π no puede contener nombres de fluentes definidos en AD. Por lo tanto, el programa Π solo puede utilizarse para definir los objetos y tipos del dominio, y establecer relaciones entre ellos.

Por otra parte, las reglas causales presentes en AD permiten definir conocimiento dinámico del dominio, ya que describen como se modifica el valor de los fluentes. Las reglas causales dinámicas definen como son afectados los fluentes por la ejecución de acciones y por los valores de otros fluentes. Las reglas causales estáticas, son un caso particular de las anteriores, y definen el valor de un fluente en un estado a partir de los valores de otros fluentes en el mismo estado.

En este sentido, las reglas causales estáticas en $\mathcal{K}$ tienen el mismo uso que las reglas rebatibles en DAKAR, incluso su estructura sintáctica es muy similar. Al igual que una regla rebatible, la cabeza f de una regla causal estática:

$$\textbf{caused } f \textbf{ if } b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_l$$

es un literal (esto esun átomo a o un átomo negado $\neg a$, donde $\neg$ representa la negación fuerte) y el cuerpo de la regla es un conjunto de literales, posiblemente negados utilizando negación por falla (not).

Sin embargo, desde el punto de vista semántico estas reglas son diferentes. Intuitivamente, una regla rebatible $f \prec B$ (en el contexto de un dominio de planificación DAKAR) expresa que *"razones para creer en B en un estado proveen razones para creer en f en dicho estado"* lo que implícitamente introduce un análisis dialéctico, mientras que una regla causal estática **caused** f **if** B significa que *"f es verdadero en cualquier estado donde B es verdadero"*.

En general, las reglas causales siguen una semántica equivalente a la semántica de conjuntos de respuesta (en inglés, *answer set semantics* [GL90]). Una comparación exhaustiva de la semántica de DeLP con la semántica de conjuntos de respuesta, está fuera del alcance de esta tesis y se refiere al lector a [GS04, TKI08] para una comparación de DeLP con otros formalismos que proveen una expresividad comparable.

Tanto las reglas causales estáticas como las reglas rebatibles permiten el uso de negación fuerte. Sin embargo, el uso de estas reglas para representar información tentativa o contradictoria es muy distinto. La negación fuerte no puede utilizarse libremente en las reglas causales estáticas, porque se podrían derivar literales complementarios produciendo estados inconsistentes. Para evitar este problema, el diseñador del dominio debe definir explícitamente las condiciones que impiden que una regla se aplique, utilizando la negación por falla. Las reglas rebatibles en cambio, permiten representar información contradictoria de manera mas general y abstracta. El proceso de dialéctica es el encargado de comparar la información y decidir cual es la conclusión que está garantizada.

Por ejemplo, consideremos el dominio definido en el ejemplo 4.1 sobre un agente encargado de la limpieza de una oficina. Según el conocimiento del dominio, un interruptor mojado resulta peligroso salvo que no haya electricidad. Esta información puede representarse por medio de las siguientes reglas rebatibles:

$$peligroso(I) \prec interruptor(I), mojado(I).$$
$$\sim peligroso(I) \prec interruptor(I), mojado(I), \sim electricidad.$$

Utilizando estas reglas, en un estado donde un interruptor i_1 se encuentra mojado y no hay electricidad se pueden construir los siguientes argumentos:

$\mathcal{A}_1 = \langle peligroso(i_1), peligroso(i_1) \prec interruptor(i_1), mojado(i_1) \rangle$
$\mathcal{A}_2 = \langle \sim peligroso(i_1), \sim peligroso(i_1) \prec interruptor(i_1), mojado(i_1), \sim electricidad \rangle$

Si se utiliza especificidad generalizada, resulta que $\mathcal{A}_2 \succ \mathcal{A}_1$ y por lo tanto un análisis dialéctico concluirá que $\sim peligroso(i_1)$ está garantizado.

La misma información puede representarse en $\mathcal{K}$ por medio de las siguientes reglas causales estáticas:

caused $peligroso(I)$ **if** $interruptor(I), mojado(I), not\ \neg peligroso(I)$
caused $\neg peligroso(I)$ **if** $interruptor(I), mojado(I), \neg electricidad$

Nótese que el diseñador del dominio debe incluir explícitamente la condición $not\ \neg peligroso(I)$ (esto es, $peligroso(I)$ puede ser cierto) en la primer regla, para evitar que tanto $peligroso(i_1)$ como $\neg peligroso(i_1)$ puedan derivarse en un estado donde un interruptor i_1 está mojado no hay electricidad.

Como se mencionó anteriormente, las reglas causales estáticas son un caso especial de las reglas causales dinámicas. Estas reglas permiten definir como son afectados los fluentes por la ejecución de acciones y por los valores de otros fluentes. Básicamente, estas reglas se utilizan para definir los efectos condicionales de las acciones, pero presentan algunas características que no están presentes en los otros formalismos. Consideremos una regla causal dinámica:

caused f **if** B **after** A

Dado que A puede contener literales de acciones, además de fluentes y de tipos, esto permite especificar efectos condicionados a la ejecución simultánea de varias acciones. Otra característica particular es que B permite especificar una condición que debe cumplirse en el estado resultante para que el fluente f sea verdadero en dicho estado.

Dado que la cabeza (f) de la regla puede ser $false$, es posible definir *restricciones* (esto es, condiciones que no puedan ocurrir nunca) que permiten acotar el diagrama de transición de estados definido por el dominio. Además, dado que un dominio definido en $\mathcal{K}$ es implementado mediante una traducción a un programa lógico disyuntivo `DLV` [ELM+98], las restricciones permiten al diseñador sacar provecho del cálculo de los conjuntos de respuesta subyacente obteniendo una implementación más eficiente.

Estas características pueden combinarse libremente en una o más reglas y permiten definir algunos conceptos interesantes como los vistos en la sección 6.3.3.

Existen otras características presentes en los lenguajes PDDL y $\mathcal{K}$ que no se han incorporado a DAKAR por estar fuera del alcance de los objetivos de esta tesis. Algunas de éstas características son:

- Restricciones y estructuras de tipos para los objetos del dominio, variables y parámetros de las acciones.
- Restricciones de igualdad sobre los parámetros de las acciones y las variables
- Precondiciones disyuntivas en las acciones
- Efectos condicionales en las acciones
- Cuantificación universal sobre los efectos y precondiciones de acciones (solo en PDDL)

Las primeras cuatro características están orientadas principalmente a mejorar la eficiencia del proceso de planificación más que la expresividad del lenguaje.

Las primeras dos características no son difíciles de incorporar a DAKAR, y pueden ser implementadas eficientemente por tratarse de restricciones estáticas, que podrían verificarse durante la etapa de expansión del algoritmo EAPOP.

Las precondiciones disyuntivas podrían modelarse en DAKAR a través del conocimiento del dominio. En general, una acción a cuya precondición contenga de una disyunción de la forma $p_1 \vee p_2 \cdots \vee p_n$ puede representarse en DAKAR incluyendo un literal nuevo p en las precondiciones de a, e incorporando las reglas $p \prec p_1$, $p \prec p_2$, $\ldots$, $p \prec p_n$ a la base de conocimiento del dominio.

Las últimas dos características requieren un análisis más profundo en DAKAR, porque involucran aspectos dinámicos de la interacción entre argumentos y acciones. La incorporación de estas y otras características a DAKAR podría realizarse como trabajo futuro.

6.5. Argumentación y planificación

En [Pol98, Amg03, HvdT04, RA06] se presentan diferentes propuestas que combinan de alguna forma argumentación y planificación. La característica común a todos estos trabajos es que los planes son considerados argumentos para lograr determinadas metas y el análisis argumentativo se realiza sobre los planes ya construidos. El enfoque presentado en esta tesis es totalmente diferente, ya que los argumentos son piezas del plan que se utilizan

para sustentar submetas y el análisis argumentativo se realiza durante la construcción del mismo.

En [Pol98] se propone un planificador que realiza esencialmente el mismo tipo de búsqueda que el algoritmo de POP, pero razonando rebatiblemente sobre los planes. Este planificador realiza un búsqueda regresiva sobre el espacio de planes parciales (compuestos solo por acciones), planificando separadamente para cada una de las metas de una conjunción de metas y luego combinando estos planes en un solo plan que satisface todas las metas. Durante la construcción de los planes no se verifica la existencia de amenazas (threats) y se *infiere rebatiblemente* que el plan combinado es una solución al problema. Un derrotador para esta inferencia rebatible consiste en descubrir que el plan contiene una amenaza, en cuyo caso el plan es modificado agregando una restricción de orden utilizando los métodos convencionales de POP (*promotion* y *demotion*).

En [Amg03] se propone un marco argumentativo para agentes que siguen el modelo BDI. Este trabajo fue extendido luego en [RA06], donde se introducen diferentes instancias del marco argumentativo abstracto de Dung que se utilizan para argumentar sobre las creencias del agente, generar deseos consistentes y adoptar planes para lograr estos deseos. Aunque estos trabajos relacionan argumentación y planes, difieren considerablemente del formalismo propuesto en esta tesis, principalmente porque los planes no son construidos a partir de aciones, sino que se obtienen de combinar planes de una base de planes preexistentes. Como expresan claramente los autores en [Amg03], no están interesados en la forma en que los planes son generados. Un plan es considerado como un *"argumento instrumental"* para lograr un determinado deseo y se define una noción de conflicto entre planes. Luego, un análisis argumentativo permite determinar que planes elegir de un conjunto de planes conflictivos para lograr los deseos del agente.

El trabajo presentado en [HvdT04] es una extensión al marco argumentativo presentado en [Amg03], donde se incorpora un procedimiento para generar metas a partir de los deseos del agente y resolver conflictos entre las metas utilizando argumentación. Al igual que en [Amg03, RA06] el tratamiento de la construcción de planes es muy superficial y la argumentación se utiliza para resolver conflictos entre planes.

Capítulo 7

Conclusiones

Esta tesis involucra principalmente dos áreas de la inteligencia artificial como son la argumentación rebatible y la planificación. Su principal aporte consiste en la definición de un formalismo que combina acciones y argumentación rebatible para representar dominios y problemas de planificación, y la definición de un nuevo método de planificación para resolver problemas definidos en este nuevo formalismo.

En el capítulo 3 se definió DAKAR (Defeasible Argumentation for Knowledge and Action Representation), un formalismo de representación de acciones y dominios de planificación que permite definir conocimiento del dominio utilizando reglas rebatibles de DeLP. Este conocimiento se utiliza para razonar acerca de las precondiciones, restricciones y efectos de las acciones, permitiendo obtener conclusiones en base a un análisis dialéctico que evalúa argumentos a favor y en contra, construidos utilizando las reglas rebatibles. De esta manera, las consecuencias producidas por la ejecución de una acción en un estado no se limitan a los efectos explícitos de la misma, sino a todas las conclusiones que puedan garantizarse mediante el análisis dialéctico a partir del estado resultante y las reglas rebatibles disponibles.

A diferencia de otros formalismos de representación, DAKAR permite distinguir entre efectos que se quitan de la descripción de un estado y efectos que se agregan negados. Otra característica distintiva de DAKAR es que permite utilizar tanto la negación fuerte como la negación por falla. Estas características combinadas con la Programación en Lógica Extendida resultan adecuadas para representar dominios con información incompleta y contradictoria.

En base al formalismo de representación DAKAR, en el capitulo 4 se definió un nuevo método de planificación denominado APOP (*Argumentative Partial Order Planning*) que combina Argumentación Rebatible con técnicas de Planificación de Orden Parcial (POP). La incorporación de Argumentación Rebatible permite utilizar el conocimiento del dominio para razonar rebatiblemente durante la construcción de un plan.

Para poder incorporar argumentación rebatible al proceso de planificación, se definió una nueva estructura de plan que combina acciones y argumentos llamada *plan parcial argumentativo* (PPA). En este contexto, se analizaron detalladamente todas las posibles interferencias que pueden surgir de la interacción entre acciones y argumentos dentro de un PPA, se definieron nuevos tipos de amenazas y nuevos métodos para resolver cada una de ellas.

En base a los conceptos y métodos definidos en el capítulo 4, se desarrolló un algoritmo de planificación para APOP. Este algoritmo es una extensión al algoritmo tradicional de POP que incorpora argumentación rebatible para construir *planes parciales argumentativos* para resolver problemas de planificación DAKAR.

En el capítulo 5 se presentó EAPOP (*Expanded* APOP), un algoritmo basado en APOP que fue utilizado para la implementación de un prototipo de planificador desarrollado como complemento de esta tesis. Este nuevo algoritmo combina una búsqueda iterativa con un proceso de expansión obteniendo las siguientes ventajas:

- reduce el espacio de búsqueda.
- permite calcular de manera general la longitud mínima de la solución, lo cual permite evitar ciclos de búsqueda infructuosos.
- posibilita determinar cuando un problema de planificación no tiene solución.

Finalmente, en el capítulo 6 se presentó un resumen de los diferentes enfoques y formalismos que han sido propuestos en la literatura para representar y resolver problemas de planificación. En particular, se describieron más detalladamente los lenguajes PDDL y $\mathcal{K}$, por ser los más completos y expresivos, y se realizó una comparación con DAKAR basada principalmente en los aspectos expresivos.

Por otra parte, se describieron algunas de las propuestas existentes que incorporan algún tipo de argumentación al proceso de planificación. La característica común a estas propuestas, es que los planes son considerados argumentos para lograr determinadas

metas y el análisis argumentativo se realiza sobre los planes ya construidos. El enfoque presentado en esta tesis es totalmente diferente, ya que los argumentos son piezas del plan que se utilizan para sustentar submetas y el análisis argumentativo se realiza durante la construcción del mismo.

7.1. Trabajo futuro

Existen nuevas líneas de trabajo que se desprenden del trabajo realizado en esta tesis y que serán consideradas en un trabajo futuro.

Una de estas líneas está relacionada a extensiones del formalismo DAKAR desde el punto de vista expresivo y las correspondientes modificaciones al método APOP para considerar estas extensiones. En este sentido, se plantea extender el formalismo para incorporar algunas características presentes en otros lenguajes de representación como:

- Restricciones y estructuras de tipos para los objetos del dominio, variables y parámetros de las acciones.
- Restricciones de igualdad sobre las variables y los parámetros de las acciones.
- Efectos condicionales en las acciones.
- Cuantificación universal sobre los efectos y precondiciones de acciones.

Las primeras dos características no son difíciles de incorporar a DAKAR, y pueden ser implementadas eficientemente por tratarse de restricciones estáticas, que podrían verificarse durante la etapa de expansión del algoritmo EAPOP. Las últimas dos características requieren un análisis más profundo en DAKAR, porque involucran aspectos dinámicos de la interacción entre argumentos y acciones.

Otra línea de trabajo a explorar es la combinación de otros métodos de planificación con argumentación rebatible para resolver problemas de planificación definidos en DAKAR. Durante el desarrollo de esta tesis se analizó la posibilidad de extender el método de planificación de Graphplan, y surgieron ideas promisorias que deberían ser investigadas en profundidad.

La capacidad de representar conocimiento a través de la programación en lógica rebatible, hace de DAKAR un formalismo adecuado para modelar dominios de planificación

con información incompleta y contradictoria. Esta característica resulta interesante para modelar y resolver problemas de *conformant planning*, donde la información sobre el estado inicial y los efectos de las acciones es incompleta, pero los planes obtenidos deben garantizar que la meta sea alcanzada siempre, independientemente del estado del mundo. En este sentido, se plantea evaluar las capacidades del formalismo y del método de planificación APOP para modelar y resolver este tipo de problemas de planificación, así como las posibles modificaciones y extensiones.

Bibliografía

[ABW88] Apt, K. R., Blair, H. A., and Walker, A. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann, 1988, pp. 89–148.

[Amg03] Amgoud, L. A formal framework for handling conflicting desires. In *Proceedings of ECSQARU, vlume 2711 of LNCS* (2003), pp. 552–563.

[BF97] Blum, A., and Furst., M. Fast planning through planning graph Analysis. *Artificial intelligence 90*, 1-2 (1997), 281–300.

[CCS01] Capobianco, M., Chesñevar, C. I., and Simari, G. R. An argumentative Formalism for Implementing Rational Agents. In *Proceedings of the II Workshop en Agentes y Sistemas Inteligentes (WASI), VII CACIC* (October 2001), Universidad Nacional de la Patagonia San Juan Bosco, El Calafate, Argentina, pp. 1051–1062.

[CGT89] Ceri, S., Gottlob, G., and Tanca, L. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng. 1*, 1 (1989), 146–166.

[EFL+00] Eiter, T., Faber, W., Leone, N., Pfeifer, G., and Polleres, A. Planning under incomplete knowledge. In *Computational Logic* (2000), pp. 807–821.

[EFL+03] Eiter, T., Faber, W., Leone, N., Pfeifer, G., and Polleres, A. A logic programming approach to knowledge-state planning, ii: The dlv$^{\mathrm{k}}$ system. *Artif. Intell. 144*, 1-2 (2003), 157–211.

[ELM+98] Eiter, T., Leone, N., Mateis, C., Pfeifer, G., and Scarcello, F. The kr system dlv: Progress report, comparisons and benchmarks. In *KR* (1998), pp. 406–417.

[FEGS08] Ferretti, E., Errecalde, M., García, A. J., and Simari, G. R. Decision rules and arguments in defeasible decision making. In *COMMA* (2008), pp. 171–182.

[FL03] Fox, M., and Long, D. Pddl2.1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR) 20* (2003), 61–124.

[FN71] Fikes, R., and Nilsson, N. J. Strips: A new approach to the application of theorem proving to problem solving. *Artif. Intell. 2*, 3/4 (1971), 189–208.

[GGS08] García, D. R., García, A. J., and Simari, G. R. Defeasible reasoning and partial order planning. In *FoIKS* (2008), pp. 311–328.

[GHK+98] Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., and Wilkins, D. Pddl—the planning domain definition language, 1998.

[Giu00] Giunchiglia, E. Planning as satisfiability with expressive action languages: Concurrency, constraints and nondeterminism. In *KR* (2000), pp. 657–666.

[GKL97] Giunchiglia, E., Kartha, G. N., and Lifschitz, V. Representing action: Indeterminacy and ramifications. *Artificial Intelligence 95* (1997), 409–443.

[GL90] Gelfond, M., and Lifschitz, V. Logic programs with classical negation. In *ICLP* (1990), D. Warren and P. Szeredi, Eds., MIT Press, pp. 579–597.

[GL91] Gelfond, M., and Lifschitz, V. Classical negation in logic programs and disjunctive databases. *New Generation Comput. 9*, 3/4 (1991), 365–386.

[GL93] Gelfond, M., and Lifschitz, V. Representing action and change by logic programs. *J. Log. Program. 17*, 2/3&4 (1993), 301–321.

[GL98] Giunchiglia, E., and Lifschitz, V. An action language based on causal explanation: preliminary report. In *Proceedings of National Conference on Artificial Intelligence (AAAI)* (1998), AAAI Press, pp. 623–630.

[Gre69] Green, C. C. Application of theorem proving to problem solving. In *IJCAI* (1969), pp. 219–240.

[GS04] García, A. J., and Simari, G. R. Defeasible logic programming: An argumentative approach. *Theory and Practice of Logic Programming 4*, 1 (2004), 95–138.

[GSG07] Garcia, D. R., Simari, G. R., and Garcia, A. J. Planning and Defeasible Reasoning. In *Proceedings of the Sixth Intl. Joint Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS 07).* (2007), pp. 856–858.

[HvdT04] Hulstijn, J., and van der Torre, L. Combining goal generation and Planning in and argumentation framework. In *In Proceedings of the 10th International Workshop on Non-Monotonic Reasoning (NMR2004)* (2004). ISBN. 92-990021-0-X.

[KL94] Kartha, G. N., and Lifschitz, V. Actions with indirect effects (preliminary report). In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)* (1994), pp. 341–350.

[KS92] Kautz, H. A., and Selman, B. Planning as satisfiability. In *ECAI* (1992), pp. 359–363.

[KS99] Kautz, H. A., and Selman, B. Unifying sat-based and graph-based planning. In *IJCAI* (1999), pp. 318–325.

[Lif96] Lifschitz, V. Foundations of logic programs. In *Principles of Knowledge Representation*, G. Brewka, Ed. CSLI Pub., 1996, pp. 69–128.

Es un trabajo muy bueno donde se condensa y ordena los últimos avances en programación en lógica. Se define primeramente los *programas básicos* con "negación clásica" y un operador de concecuencia. Luego define el cálculo SLD para programas básicos. En segundo lugar define a los programas normales (sin negación). Luego introduce la negación por falla, presenta la semántica de conjuntos de respuestas, de [GL90], presenta los programas *tight* y estratificados, y define el cálculo SLDNF. A continuación define los programas con reglas esquemáticas (con variables) y extiende el cálculo SLDNF para trabajar con variables. Finalmente introduce los programas disyuntivos. interesante: Answer set: pp 23, esquemas: PP 39.

[Lou87] Loui, R. P. Defeat Among Arguments: A System of Defeasible Inference. *Computational Intelligence 3*, 3 (1987), 100–106.

Este trabajo presenta un sistema de razonamiento no-monotónico con reglas rebatibles. En este sistema ante la aparición de múltiples extensiones, se realiza un ordenamiento utilizando consideraciones sintácticas. El sistema trata la derrota de la forma en que es tomada por los filósofos en epistomología. En [Lou87], una regla rebatible se denota "$a \succ b$", y se interpreta como "a es una razón rebatible para b". El sistema formal está definido a partir de una base de datos $\langle EK, R\rangle$, donde EK es conocimiento evidencial, y R el conjunto de reglas rebatibles. Las reglas rebatibles permiten formar argumentos.

[LRL+97] Levesque, H. J., Reiter, R., Lespérance, Y., Lin, F., and Scherl, R. B. Golog: A logic programming language for dynamic domains. *J. Log. Program. 31*, 1-3 (1997), 59–83.

[McC80] McCarthy, J. Circumscription - a form of non-monotonic reasoning. *Artif. Intell. 13*, 1-2 (1980), 27–39.

[MH69] McCarthy, J., and Hayes, P. J. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*, B. Meltzer and D. Michie, Eds. Edinburgh University Press, 1969, pp. 463–502. reprinted in McC90.

en este trabajo se introduce situation calculus.

[MT98] McCain, N., and Turner, H. Satisfiability planning with causal theories. In *KR* (1998), pp. 212–223.

[Ped89] Pednault, E. P. D. Adl: Exploring the middle ground between strips and the situation calculus. In *KR* (1989), pp. 324–332.

[Pol98] Pollock, J. Defeasible Planning. In *Integrating Planning, Scheduling, and Execution in Dynamic and Uncertain Environments AIPS Workshop.* (1998), Ralph Bergmann and Alexander Kott, Cochairs.

[Poo85] Poole, D. L. On the Comparison of Theories: Preferring the Most Specific Explanation. In *Proc. 9th IJCAI* (1985), IJCAI, pp. 144–147.

[PW92] Penberthy, J., and Weld, D. S. UCPOP: A Sound, Complete, Partial Order Planner for ADL. In *In Proc. of the 3rd. Int. Conf. on Principles of Knowledge Representation and Resoning, 113-124.* (1992).

[RA06] Rahwan, I., and Amgoud, L. An argumentation-based approach for practical reasoning. In *Proc. AAMAS* (2006), pp. 347–354.

[Rei80] Reiter, R. A logic for default reasoning. *Artif. Intell. 13*, 1-2 (1980), 81–132.

[SCG94] Simari, G. R., Chesñevar, C. I., and García, A. J. The role of dialectics in defeasible argumentation. In *XIV International Conference of the Chilenean Computer Science Society* (November 1994).

En este trabajo se trata por primera vez el problema de líneas de argumentación falaces. Se introduce el concepto de árbol de dialéctica aceptable, y marcado de un árbol de dialéctica. Este trabajo dió origen al estudio de las falacias que luego fue extendido en muchos aspectos en trabajos posteriores, e incluso en esta tesis..

[SL92] Simari, G. R., and Loui, R. P. A Mathematical Treatment of Defeasible Reasoning and its Implementation. *Artificial Intelligence 53* (1992), 125–157.

[THN05] Thiébaux, S., Hoffmann, J., and Nebel, B. In defense of pddl axioms. *Artif. Intell. 168*, 1-2 (2005), 38–69.

[TKI08] Thimm, M., and Kern-Isberner, G. On the relationship of defeasible argumentation and answer set programming. In *COMMA* (2008), pp. 393–404.

[Tur97] Turner, H. Representing actions in logic programs and default theories: A situation calculus approach. *J. Log. Program. 31*, 1-3 (1997), 245–298.

[Wel94] Weld, D. S. An introduction to least commitment planning. *AI Magazine 15*, 4 (1994), 27–61.