Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Un procesador de expresiones epistémicas en programas lógicos

Estudiante: Javier Garea Cidre
Director: José Pedro Cabalar Fernández

A Coruña, 6 de septiembre de 2019.

A mi madre.

Agradecimientos

A mi tutor, Pedro Cabalar, por su inestimable ayuda en la realización de este trabajo.
A Jorge Fandiño y a Javier Romero, por sus consejos de implementación y optimización.
A mi madre, María José, por darme la vida y apoyarme en todas mis decisiones.
A mis hermanos, Nicolás y Santiago, por estar siempre ahí y no dejar que me rinda.
Y a Lara, por su cariño, por su paciencia y por llenar de felicidad mis días.

Resumen

En este proyecto se ha desarrollado la herramienta `eclingo` que calcula los modelos de un programa lógico con expresiones epistémicas. Estas expresiones suponen una ampliación del lenguaje declarativo *Answer Set Programming* (ASP), ampliamente usado en el área de Representación del Conocimiento en Inteligencia Artificial. En ASP, un problema de búsqueda se representa en términos de un programa lógico, y las soluciones al problema se obtienen a partir de los modelos (answer sets) del programa. Las expresiones epistémicas admitidas por `eclingo` permiten razonar sobre hechos que están presentes en todos los answer sets o en alguno de ellos, lo que permite razonamiento sobre incertidumbre y conocimiento parcial. La eficiencia de `eclingo` se ha evaluado a través de un estudio comparativo frente a otra herramienta de características similares, ofreciendo unos resultados muy positivos que la sitúan como una alternativa competitiva dentro del estado del arte.

Abstract

The developed tool, `eclingo`, computes the models of logic programs with epistemic expressions. These expressions represent an extension of the declarative language *Answer Set Programming*, widely used in the area of Knowledge Representation in Artificial Intelligence. In ASP, a search problem is represented in terms of a logic program, and solutions to the problem are obtained from the models (answer sets) of this program. The epistemic expressions accepted by `eclingo` allow reasoning about facts that are present in all answer sets or in some of them, which enables reasoning about uncertainty and partial knowledge. The efficiency of `eclingo` has been evaluated through a comparative study against another tool with similar characteristics, offering very positive results that place it as a competitive alternative within the state of the art.

Palabras clave:

- Programación lógica
- Representación del conocimiento
- Answer Set Programming
- Especificaciones epistémicas
- Planificación conformante

Keywords:

- Logic Programming
- Knowledge Representation
- Answer Set Programming
- Epistemic specifications
- Conformant planning

Índice general

1 Introducción **1**

1.1 Objetivos . . . 3

1.2 Estructura . . . 4

2 Fundamentos **5**

2.1 Answer Set Programming . . . 5

2.1.1 Conceptos básicos . . . 5

2.1.2 Programas con predicados . . . 8

2.1.3 Ampliaciones . . . 11

2.1.4 Aplicaciones . . . 11

2.1.5 Complejidad computacional . . . 14

2.2 Especificaciones epistémicas . . . 16

2.2.1 Semántica G91 . . . 18

2.2.2 Semántica K14 . . . 19

2.2.3 Aplicaciones . . . 20

2.2.4 Complejidad computacional . . . 24

2.3 `clingo` . . . 24

2.3.1 Lenguaje de entrada . . . 25

2.3.2 Multi-shot solving . . . 26

3 `eclingo` **29**

3.1 Módulo parser . . . 32

3.1.1 Preprocesado . . . 33

3.1.2 Definición de constantes . . . 34

3.1.3 Adición de reglas *choice* . . . 35

3.1.4 Adición de directivas de proyección . . . 36

3.1.5 Mejora . . . 38

3.2 Módulo solver . . . 39

3.2.1 Generación de candidatos . . . 39
3.2.2 Evaluación de candidatos . . . 39
3.2.3 Selección de resultados . . . 43

4 Evaluación **45**
4.1 Estado del arte . . . 45
4.1.1 `Wviews` . . . 45
4.1.2 `EP-ASP` . . . 45
4.2 Estudio comparativo . . . 46
4.2.1 Scholarship Eligibility Problem . . . 47
4.2.2 Yale Shooting Problem . . . 51

5 Metodología, seguimiento y coste **57**
5.1 Metodología . . . 57
5.2 Seguimiento . . . 58
5.3 Coste . . . 59

6 Conclusiones **63**
6.1 Trabajo futuro . . . 64
6.1.1 Nuevas mejoras . . . 64
6.1.2 Publicación . . . 64
6.1.3 Inclusión en el paquete de herramientas `Potassco` . . . 65

Bibliografía **67**

Índice de figuras

2.1 Proceso de cómputo de las soluciones de un programa ASP. 10
2.2 Solución al *Yale Shooting Problem* con el arma cargada en el estado inicial. . . 14
2.3 Solución al *Yale Shooting Problem* con el arma descargada en el estado inicial. . 14
2.4 Relaciones de inclusión entre las clases de complejidad más comunes. 15
2.5 Escenario del *Critical and Irrelevant Paths Problem*. 21
2.6 Planes ASP para el *Yale Shooting Problem* con el estado inicial del arma desconocido. 23
2.7 *Conformant plan* para el *Yale Shooting Problem* con el estado inicial del arma desconocido. 24
2.8 Proceso de cómputo de las soluciones de un programa ASP con `clingo`. . . 25

3.1 Arquitectura de `eclingo` 30
3.2 *yale.lp* 31
3.3 Formato de salida de `eclingo` 32
3.4 Fragmento de *solver.py* (1) 42
3.5 Fragmento de *solver.py* (2) 44
3.6 Fragmento de *solver.py* (3) 44

4.1 *eligible1.elps.elp* 48
4.2 *eligible.lp* 48
4.3 *eligible01.lp* 48
4.4 Representación semilogarítmica de los datos de la tabla 4.1. 50
4.5 *yale01.lp* 52
4.6 *yale1.txt* (1) 53
4.7 *yale1.txt* (2) 54
4.8 Representación semilogarítmica de los datos de la tabla 4.2. 56

5.1 Diagrama de Gantt (1) 60

5.2 Diagrama de Gantt (2) 61

Índice de tablas

4.1 Tabla de tiempos para ejecuciones del *Scholarship Elegibility Problem.* 49
4.2 Tabla de tiempos para ejecuciones del *Yale Shooting Problem.* 55

5.1 Coste estimado de los recursos humanos del proyecto. 59
5.2 Coste estimado de los recursos materiales del proyecto. 62

Capítulo 1

Introducción

En los últimos años, la Inteligencia Artificial (IA) [1] ha pasado de ser un sujeto de estudio a formar parte de nuestro entorno cotidiano en forma de pulseras inteligentes, aspiradoras automáticas, sistemas de recomendación de series y películas o asistentes de voz, entre otros. Este término, acuñado por McCarthy en 1959 [2], se ha vuelto tremendamente popular, apareciendo en numerosas publicaciones diariamente, tanto de propósito específico como de propósito general.

Una de las áreas de la Inteligencia Artificial, en la que esta dio sus primeros pasos, es la Representación del Conocimiento. Esta disciplina, íntimamente relacionada con el Razonamiento Automático, pretende formalizar el conocimiento de un agente sobre el mundo real de manera que un ordenador lo procese y pueda razonar sobre él. De esta forma, un sistema basado en conocimiento, suele contar con una serie de reglas expresadas en un lenguaje declarativo. Sobre estas reglas, efectúa procesos de inferencia con el objetivo de deducir nueva información o encontrar la solución a un problema.

El estudio y desarrollo de formalismos suficientemente expresivos para poder efectuar procesos de razonamiento complejos, es uno de los retos a los que se enfrenta la Representación del Conocimiento. En sus comienzos, los formalismos empleados eran la lógica proposicional y el cálculo de predicados. Sin embargo, con el avance en la investigación, estas lógicas no resultaron realmente apropiadas para formalizar conocimiento de sentido común, ya que este suele involucrar razonamiento *no monótono*. A grandes rasgos, el razonamiento no monótono es aquel que permite modificar el conocimiento ya adquirido ante la presencia de nueva información. Un ejemplo podría ser la frase *"normalmente, las aves vuelan"*. La propia palabra "normalmente" ya nos indica que esta regla no es absoluta y admite ciertas excepciones: por ejemplo, uno puede aprender más adelante que los pingüinos son una excepción a esta regla por defecto, y acomodar ese conocimiento. Esta capacidad es crucial para poder lograr una representación formal flexible y modular ya que, sin ella, estamos obligados a reformular

nuestra teoría cada vez que se produce o se conoce un pequeño cambio (como por ejemplo, una nueva excepción a la norma de las aves que vuelan) en el planteamiento del problema (McCarthy denominó a esta flexibilidad como *tolerancia a la elaboración*). En consecuencia, la investigación derivó en nuevos formalismos capaces de tratar con información por defecto. De las diversas aproximaciones, Answer Set Programming (de aquí en adelante, ASP) [3, 4] es probablemente el paradigma más exitoso de la Representación del Conocimiento.

ASP es un paradigma orientado al modelado de conocimiento y a la resolución declarativa de problemas de búsqueda, típicamente complejos. Se basa en la semántica de modelos estables, también conocida como *answer set semantics*, que toma nociones de la Lógica por Defecto y la Lógica Autoepistémica de Moore para el tratamiento de la negación por defecto. Su éxito se fundamenta en la expresividad de su lenguaje y en la eficiencia de sus herramientas de cálculo, los *solvers*. A pesar de la gran cantidad de aplicaciones prácticas que presenta, existen una serie de escenarios en los que el uso de ASP no resulta suficiente para la resolución del problema. Para solventar estas carencias, se están estudiando y desarrollando diversas ampliaciones del lenguaje con el objetivo de facilitar la representación de ciertos tipos de escenarios o aumentar su complejidad computacional para resolver problemas más complejos. Una de las ampliaciones del lenguaje de ASP que aumenta dicha complejidad computacional es la que permite el uso de *expresiones epistémicas*. Esta ampliación, propuesta por Michael Gelfond [5], consiste en añadir un nuevo operador modal mediante el cual es posible comprobar si un hecho forma parte de todos (o de alguno de) los *answer sets* de un programa, y derivar información a partir de ello. Esta característica permite representar el conocimiento de un agente dentro de un programa lógico, lo que hace posible razonar correctamente sobre información incompleta cuando nos encontramos ante múltiples interpretaciones (los *answer sets*) posibles.

Un ejemplo de este tipo de escenarios es el problema de *Conformant Planning*. En un problema de planificación clásico, los efectos de las acciones son deterministas y el estado inicial es totalmente conocido. Sin embargo, en un problema de *Conformant Planning*, estos conceptos pueden tomar un aspecto no determinista, lo que refleja la información parcial del agente. Un ejemplo tradicional de la literatura es el conocido como *Yale Shooting Problem* [6]. En este escenario, contamos con un arma y un objetivo a abatir (habitualmente, un pavo). Si conocemos el estado inicial del arma, cargada o descargada, es realmente sencillo obtener un plan para disparar al objetivo empleando ASP. El problema aparece cuando este estado es indeterminado, pues existen varias interpretaciones del problema: o bien el arma se encuentra cargada, o bien está descargada. Si pedimos una solución a ASP, este nos generará dos escenarios diferentes: en el primero el arma se encuentra cargada y por lo tanto apretamos el gatillo, en el segundo el arma está descargada y debemos cargar el arma antes de apretar

el gatillo. Sin embargo ninguna de estas soluciones es un *conformant plan*, pues ninguna garantiza el objetivo final independientemente del estado inicial: si tan sólo disparamos, el plan no funciona con un arma descargada; si empezamos intentando cargar, el plan no funciona con un arma ya cargada inicialmente. En realidad estas soluciones non son planes, sino explicaciones formadas por secuencias de acciones que, bajo ciertas circunstancias, llegan a la meta. Un plan generado bajo el lenguaje de especificaciones epistémicas se compondría de tres acciones: apretar el gatillo, lo que nos garantiza que el arma se queda descargada (y quizá ya hayamos abatido al pavo), cargar el arma y apretar el gatillo de nuevo, lo que nos asegurará que finalmente está muerto. Este plan garantiza la meta independientemente del no determinismo del estado inicial.

En este contexto, nacen las herramientas de cálculo de modelos de un programa lógico con expresiones epistémicas. A pesar del aumento en la complejidad computacional que supone dicha ampliación (de $\mathbf{\Sigma_2^P}$ en ASP normal a $\mathbf{\Sigma_3^P}$ en ASP epistémico), con el respectivo incremento en la capacidad de resolución de problemas, la realidad es que no existen muchas implementaciones de este tipo de herramientas.

Recientemente, esta carencia de herramientas de cálculo se ha hecho especialmente notable debido al creciente interés que presenta la comunidad ASP en esta ampliación. Este incremento de interés, viene propiciado por los resultados del trabajo en investigación en la teoría de especificaciones epistémicas de los últimos tiempos [7, 8, 9, 10, 11, 12, 13, 14, 15]. El desarrollo de una herramienta de este tipo, supone una gran oportunidad para contribuir a dicha investigación.

1.1 Objetivos

El objetivo principal del proyecto es el desarrollo de una herramienta capaz de procesar y resolver programas lógicos con expresiones epistémicas.

La herramienta debe permitir la especificación de expresiones epistémicas bajo una sintaxis sencilla y fácilmente entendible. La aplicación empleará el *solver* ASP `clingo` [16] como *backend*, usando las facilidades de esta herramienta para la ampliación de su sintaxis e intentando, en la medida de lo posible, mantener la mayor parte de construcciones sintácticas del propio lenguaje de `clingo`.

La eficiencia de la herramienta es también uno de los objetivos principales, pues la complejidad computacional de computar los modelos de un programa lógico epistémico es realmente alta (como se dijo antes, $\mathbf{\Sigma_3^P}$). Sin una implementación eficiente, la ampliación epistémica no tendrá un impacto práctico o será muy difícil de probar.

1.2 Estructura

De aquí en adelante, el documento se estructura de la siguiente manera:

Capítulo 2. Fundamentos. Se da una introducción a los fundamentos teóricos y tecnológicos en los que se basa el desarrollo del trabajo.

Capítulo 3. `eclingo`. Se describe la herramienta desarrollada y su arquitectura. Además, se detallan los módulos que la conforman, *parser* y *solver*.

Capítulo 4. Evaluación. Se exploran las diferentes herramientas de cálculo de modelos en programas lógicos epistémicos disponibles. Además, se presenta un estudio comparativo entre `eclingo` y el *solver* epistémico EP-ASP, anterior al desarrollo del presente trabajo.

Capítulo 5. Metodología, seguimiento y coste. Se presentan los aspectos relacionados con la gestión del proyecto.

Capítulo 6. Conclusiones. Se comentan las conclusiones extraídas del proyecto, así como las posibles líneas de trabajo futuro.

Capítulo 2

Fundamentos

Este capítulo introduce los fundamentos teóricos y técnicos más relevantes para el desarrollo del trabajo.

2.1 Answer Set Programming

Answer Set Programming (ASP) [3] es un paradigma orientado a la resolución declarativa de problemas en el área de la Representación del Conocimiento (KR). Su principal campo de aplicación es la resolución de problemas de búsqueda complejos.

ASP se basa en la semántica de modelos estables [17], también conocida como *answer set semantics*. Esta semántica se encuentra íntimamente ligada al razonamiento no monótono, pues toma nociones de la Lógica por Defecto y la Lógica Autoepistémica de Moore para el tratamiento de la negación por defecto.

2.1.1 Conceptos básicos

Un programa en ASP es un conjunto de reglas que siguen la siguiente estructura:

$$a_1, \ldots, a_h \leftarrow b_1, \ ..., \ b_m, \ not\ c_1, \ ..., \ not\ c_n \tag{2.1}$$

donde todas las a_i, b_j y c_k son átomos. La solución a un programa ASP es el conjunto formado por sus modelos estables o *answer sets*.

Un átomo es una proposición elemental que puede tomar el valor *cierto* (en inglés, *true*) o falso (en inglés, *false*). Para referirnos a un átomo o a su negación, hablamos de un literal.

Una regla es una implicación cuyo antecedente es el cuerpo, conjunción de los literales que se encuentran a la derecha de la conectiva $\leftarrow$, y cuyo consecuente es la cabeza, disyunción

de los átomos que se encuentran a la izquierda de la conectiva $\leftarrow$. Esta manera de representar la regla, en contraposición a la tradicional estructura $antecedente \rightarrow consecuente$, tiene como motivo facilitar la legibilidad de los programas, pues una regla se puede ver como una justificación para derivar que la cabeza es cierta si el cuerpo lo es. Por ejemplo, supongamos una caja llena de pelotas de color rojo y azul. La regla:

$$blue \leftarrow not\ red \tag{2.2}$$

quiere decir que *"si la pelota no es roja, entonces es azul"*.

Es importante remarcar el hecho de que not no representa la negación clásica ($\neg$), si no que indica *negación por defecto*. La negación por defecto consiste en considerar que algo es falso si no tenemos información para deducir que es cierto. Por lo tanto, una descripción más apropiada de lo que representa la regla (2.2) sería *"la pelota es azul si no tenemos evidencia de que sea roja"*. Para expresar negación clásica (o fuerte), se emplea el operador -.

Cuando una regla carece de cuerpo, podemos decir que la cabeza no necesita ninguna justificación. De este modo, podemos confirmar que la cabeza es incondicionalmente cierta. Este tipo de reglas se denominan *hechos* y normalmente omiten la conectiva $\leftarrow$. Por ejemplo, la regla:

$$ball \tag{2.3}$$

representa que el hecho $ball$ no necesita justificación y, en consecuencia, es cierto siempre. Se ve fácilmente si escribimos la regla completa:

$$ball \leftarrow \top \tag{2.4}$$

Si la regla tiene cuerpo pero no tiene cabeza, estamos hablando de una restricción o *constraint*. Este tipo de reglas representan una prohibición en el programa. Por ejemplo, la regla:

$$\leftarrow blue,\ red \tag{2.5}$$

representa que *"no puede ser que la pelota sea azul y roja a la vez"*. De nuevo, es muy fácil verlo si escribimos la regla completa:

$$\bot \leftarrow blue,\ red \tag{2.6}$$

Otro tipo especial de reglas, son las reglas *choice*. Estas reglas se caracterizan por el uso de llaves en la cabeza. Por ejemplo, la regla:

$$\{red\} \leftarrow ball \tag{2.7}$$

representa que *"si tenemos una pelota, generar un caso en que es roja y otro en que no"*. De esta manera, el programa formado por (2.3) y (2.7) da lugar a dos modelos estables:

$$\{ball\}$$
$$\{red,\ ball\}$$

El conjunto de modelos estables de un programa ASP viene determinado por los átomos que podemos derivar como ciertos. Sin embargo, podemos refinar este conjunto descartando modelos con el uso de *constraints*.

La semántica formal de answer sets se define del siguiente modo. Una *interpretación* proposicional I es un conjunto de átomos que son ciertos. El reducto de un programa lógico P con respecto a una interpretación I, escrito P^I, se define como el conjunto de reglas de la forma:

$$a_1, \ldots, a_h \leftarrow b_1, \ldots, b_m$$

para cada regla en P de la forma (2.1) que cumple que ningún c_i está incluido en I. Si la cabeza de la regla en P contiene una regla de elección del estilo:

$$\{a\} \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n$$

entonces el reducto P^I incluye la regla:

$$a \leftarrow b_1, \ldots, b_m$$

si ningún c_i pertenece a I y además $a \in I$. Una interpretación I es un *modelo estable* (o answer set) de un programa P si es modelo minimal (con respecto a inclusión de conjuntos) de entre los modelos clásicos de P^I. Por modelo clásico entendemos que se interpreta la coma a la izquierda de $\leftarrow$ como disyunción, a la derecha de $\leftarrow$ como conjunción, el not como negación y la propia $\leftarrow$ como implicación en lógica clásica.

Por ejemplo, el programa:

$$flies \leftarrow bird,\ not\ ab \tag{2.8}$$
$$ab \leftarrow bird,\ penguin \tag{2.9}$$
$$bird \tag{2.10}$$
$$penguin \tag{2.11}$$

tiene dos interpretaciones o modelos clásicos:

$$\{bird, penguin, ab, flies\}$$
$$\{bird, penguin, ab\}$$

El reducto del programa respecto a la primera interpretación es:

$$bird \tag{2.12}$$
$$ab \leftarrow bird,\ penguin \tag{2.13}$$
$$penguin \tag{2.14}$$

cuyo modelo minimal es:

$$\{bird, penguin, ab\}$$

Dado que la interpretación no coincide con el modelo minimal, esta interpretación no es un modelo estable del programa. En cambio, el reducto del programa respecto a la segunda interpretación es:

$$bird \tag{2.15}$$
$$ab \leftarrow bird,\ penguin \tag{2.16}$$
$$penguin \tag{2.17}$$

es decir, el mismo que el de la primera interpretación. Su modelo minimal es:

$$\{bird, penguin, ab\}$$

Este modelo minimal es la propia interpretación, por lo que dicha interpretación es un modelo estable del programa.

2.1.2 Programas con predicados

La versión proposicional de ASP, permite definir de manera sencilla su semántica. Sin embargo, lo que realmente convierte a ASP en un lenguaje potente en la resolución de problemas, es su versión con predicados. Esta versión, incluye predicados (o relaciones), constantes y variables en el lenguaje.

En este caso, una regla es una expresión con la siguiente estructura:

$$A_1, \ldots, A_h \leftarrow B_1,\ ...,\ B_m,\ not\ C_1,\ ...,\ not\ C_n$$

donde todas las A_i, B_j y C_k son fórmulas atómicas en el lenguaje que pueden contener argumentos con variables. Las variables en las reglas se entienden como cuantificadas universalmente de manera implícita y los conceptos de cabeza y cuerpo se definen igual que en la versión proposicional.

Un programa con variables se entiende como una abreviatura del programa *ground* (sin variables) en el que se reemplaza cada variable por cada posible valor constante del programa de todas las formas posibles. Por ejemplo, en el programa:

$$student(mary) \tag{2.18}$$
$$student(mike) \tag{2.19}$$
$$student(nancy) \tag{2.20}$$
$$person(X) \leftarrow student(X) \tag{2.21}$$

la última regla puede entenderse en realidad como una abreviatura de:

$$person(mary) \leftarrow student(mary) \tag{2.22}$$
$$person(mike) \leftarrow student(mike) \tag{2.23}$$
$$person(nancy) \leftarrow student(nancy) \tag{2.24}$$

La introducción de variables en las reglas del programa facilita el laborioso trabajo de codificación de un problema, pues permite representar grupos de términos englobados en un rango, como hemos visto arriba. La regla (2.21) representa que todos los estudiantes son personas, lo que hace que no sea necesario escribir una regla para cada estudiante.

La inclusión de las variables en el lenguaje de ASP, divide en dos fases el cómputo de los modelos estables. La primera fase se conoce como *grounding*. El *grounding* es el proceso por el que se instancian las variables de las reglas, dando lugar a un programa proposicional finito que representa al programa de entrada, es decir, un programa *ground*. La segunda fase se conoce como *solving*, el proceso por el que se computan los modelos estables de un programa *ground*. De esta manera, el *grounder*, la herramienta que ejecuta el grounding, produce la versión *ground* de un programa y el *solver*, la herramienta que ejecuta el *solving*, calcula sus modelos estables.

De esta manera, dado el programa:

$$player(1) \tag{2.25}$$
$$player(2) \tag{2.26}$$
$$\{winner(X)\} \leftarrow player(X) \tag{2.27}$$
$$\leftarrow winner(1),\ winner(2) \tag{2.28}$$

el proceso de *grounding* da como resultado:

$$player(1) \tag{2.29}$$
$$player(2) \tag{2.30}$$
$$\{winner(1)\} \tag{2.31}$$
$$\{winner(2)\} \tag{2.32}$$
$$\leftarrow winner(1),\ winner(2) \tag{2.33}$$

Si pasamos el resultado del *grounder* al *solver*, el resultado del proceso de *solving* es el conjunto formado por los 3 *answer sets* del programa:

$$\{player(1),\ player(2)\}$$
$$\{player(1),\ player(2),\ winner(1)\}$$
$$\{player(1),\ player(2),\ winner(2)\}$$

En la figura 2.1, se puede observar un diagrama del proceso de cómputo de las soluciones de un programa ASP.

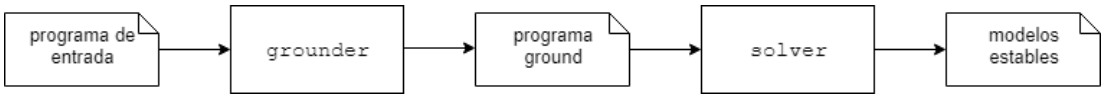

Figura 2.1: Proceso de cómputo de las soluciones de un programa ASP.

Una característica del lenguaje ASP similar a las variables, es el uso de rangos. Estos, se emplean para representar un conjunto de valores numéricos que se encuentran dentro de unos límites inferior y superior. Por ejemplo, el hecho:

$$dice(1..6) \tag{2.34}$$

representa al conjunto de reglas:

$$dice(1) \tag{2.35}$$
$$dice(2) \tag{2.36}$$
$$dice(3) \tag{2.37}$$
$$dice(4) \tag{2.38}$$
$$dice(5) \tag{2.39}$$
$$dice(6) \tag{2.40}$$

2.1.3 Ampliaciones

A pesar de la gran flexibilidad y expresividad del lenguaje ASP, todavía existen inconvenientes en su uso para ciertas áreas. Con el objetivo de superar estos inconvenientes, se han estado estudiando y desarrollando ampliaciones del lenguaje que aumentan su capacidad computacional y facilitan la representación de ciertos problemas.

Un gran ejemplo de ello, es el caso de la ampliación temporal de ASP [18], que introduce operadores de Lógica Lineal Temporal [19] en el lenguaje, haciendo mucho más sencilla la representación de dominios dinámicos.

Otras ampliaciones interesantes involucran la inclusión de *aggregates* [1], sentencias de optimización, restricciones numéricas, preferencias o expresiones epistémicas.

2.1.4 Aplicaciones

Uno de los pilares del éxito de ASP es la cantidad de aplicaciones prácticas que presenta. Estas aplicaciones pertenecen a una gran variedad de dominios, tanto académicos como industriales.

En el área industrial, entre otras aplicaciones, ASP se ha empleado en la configuración y reconfiguración de productos y servicios, en turismo electrónico y en el enrutado inteligente de llamadas telefónicas. Además, destaca su uso en el sistema de apoyo a la toma de decisiones del transbordador espacial [20].

Dentro del área de la IA, dados sus orígenes en KR y el razonamiento no monótono, ha destacado por su uso en la resolución de problemas clásicos como la diagnosis, la toma de decisiones o la planificación.

[1] Funciones sobre conjuntos.

Un ejemplo clásico y sencillo a la hora de hablar de planificación en ASP es el *Yale Shooting Problem* [6]. En este escenario, contamos con un arma y un objetivo a abatir. El estado de estos viene dado por dos variables temporales o *fluentes*: $loaded$, que nos indica si el arma se encuentra cargada, y $alive$, que nos indica si el objetivo está vivo. Además, existen dos acciones posibles: $load$, que carga el arma en caso de que esta esté descargada, y $pull\_trigger$, que aprieta el gatillo. Si el arma se encuentra cargada y se aprieta el gatillo, el arma se dispara y asumimos que el objetivo ha muerto. La solución al problema es obtener un plan para abatir al objetivo.

En ASP, los fluentes y las acciones pueden definirse a través de hechos del programa:

$$fluent(loaded) \tag{2.41}$$
$$fluent(alive) \tag{2.42}$$
$$action(load) \tag{2.43}$$
$$action(pull\_trigger) \tag{2.44}$$

Estas acciones, son ejecutables a lo largo de la longitud del plan, determinada por una constante $length$, y bajo unas circunstancias. Por ejemplo, mientras que el gatillo puede ser apretado en cualquier instante de tiempo, el arma debe estar descargada para poder cargarse. Dichas condiciones pueden representarse empleado las reglas:

$$stepless(0..length-1) \tag{2.45}$$
$$executable(pull\_trigger, T) \leftarrow stepless(T) \tag{2.46}$$
$$executable(load, T) \leftarrow -holds(loaded, T),\ stepless(T) \tag{2.47}$$

Estas acciones, tienen una serie de efectos. Por ejemplo, si el objetivo se encontraba vivo y apretamos el gatillo, el arma se dispara y el objetivo muere. Si el arma estaba cargada y se disparó, esta pasa a estar cargada. Si cargamos el arma, esta pasa a estar cargada. Una posible codificación de estos efectos podría ser:

$$-holds(alive, T+1) \leftarrow occurs(pull\_trigger, T),\ holds(loaded, T),\ stepless(T) \tag{2.48}$$
$$-holds(loaded, T+1) \leftarrow occurs(pull\_trigger, T),\ stepless(T) \tag{2.49}$$
$$holds(loaded, T+1) \leftarrow occurs(load, T),\ stepless(T) \tag{2.50}$$

Una característica importante que permite ASP es la especificación de la *ley de inercia*, imposible en lógica clásica ya que conlleva razonamiento no monótono. La ley de inercia dice que todo fluente se mantiene en su estado anterior si no hay evidencia de que nada haya cambiado.

En este caso, podríamos usar las siguientes reglas:

$$holds(F, S+1) \leftarrow fluent(F),\ stepless(S),\ holds(F, S),\ not\ {-}holds(F, S+1) \tag{2.51}$$

$${-}holds(F, S+1) \leftarrow fluent(F),\ stepless(S),\ {-}holds(F, S),\ not\ holds(F, S+1) \tag{2.52}$$

Para prohibir que dos acciones se ejecuten en el mismo instante de tiempo, podríamos usar una restricción como la siguiente:

$$\leftarrow occurs(A, S),\ occurs(B, S),\ A! = B \tag{2.53}$$

De manera similar a como sucede con la inercia, solemos suponer que en el estado inicial los fluentes toman el valor falso por defecto. Esto se representa a través de la siguiente regla:

$${-}holds(F, 0) \leftarrow not\ holds(F, 0),\ fluent(F) \tag{2.54}$$

La meta del plan consiste en que el objetivo esté muerto en el instante de tiempo final, siguiendo un plan en el que no se hayan empleado acciones no ejecutables. Esta meta se representa a través de las reglas:

$$goal \leftarrow {-}holds(alive, length) \tag{2.55}$$

$$\leftarrow not\ goal \tag{2.56}$$

Y finalmente tenemos que generar una serie de acciones posibles:

$$\{occurs(A, S)\} \leftarrow stepless(S),\ action(A) \tag{2.57}$$

$$\leftarrow occurs(A, S),\ not\ executable(A, S) \tag{2.58}$$

Un escenario de este problema puede consistir en suponer un estado inicial en el que el objetivo está vivo y el arma se encuentra cargada, es decir:

$$holds(alive, 0) \tag{2.59}$$

$$holds(loaded, 0) \tag{2.60}$$

Si establecemos que la constante $length$ toma el valor 1 y, por lo tanto, el plan tiene longitud 1, el plan generado por ASP se puede visualizar en la figura 2.2. Dicho plan consiste en apretar

el gatillo en el instante de tiempo 0, es decir:

$$occurs(pull\_trigger, 0)$$

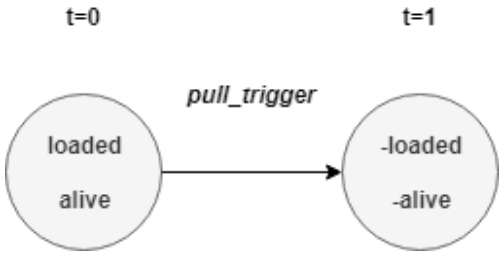

Figura 2.2: Solución al *Yale Shooting Problem* con el arma cargada en el estado inicial.

Si suponemos ahora un estado inicial en el que arma se encuentra descargada, la regla (2.60) pasaría a ser:

$$-holds(loaded, 0) \tag{2.61}$$

Si solicitamos a ASP un plan de longitud 1, no nos devolverá ningún modelo, pues no es posible alcanzar la meta en un paso.

Si solicitamos un plan de longitud 2, nos devolverá el plan de la figura 2.3, en el que cargamos el arma en el instante de tiempo 0 y apretamos el gatillo en el instante de tiempo 1, es decir, el modelo:

$$\{occurs(load, 0),\ occurs(pull\_trigger, 1)\}$$

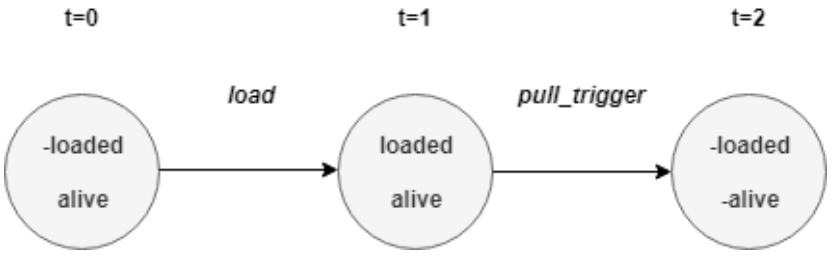

Figura 2.3: Solución al *Yale Shooting Problem* con el arma descargada en el estado inicial.

2.1.5 Complejidad computacional

Como ya hemos visto, ASP acepta disyunción en la cabeza de las reglas, ya que las comas en la cabeza representan la conectiva disyuntiva ($\vee$). Esto aumenta la capacidad de ASP para

resolver problemas, tanto por su expresividad como por la complejidad computacional de obtener los *answer sets* de un programa lógico disyuntivo.

Los problemas computacionales se catalogan dentro de una *clase de complejidad*. Las clases de complejidad engloban aquellos problemas con la misma complejidad computacional. Además, estas clases se relacionan a través de relaciones de inclusión, de manera que las clases de menor complejidad son subconjuntos de las de mayor complejidad. En la figura 2.4 [2], se pueden ver las relaciones de inclusión entre las clases de complejidad más comunes.

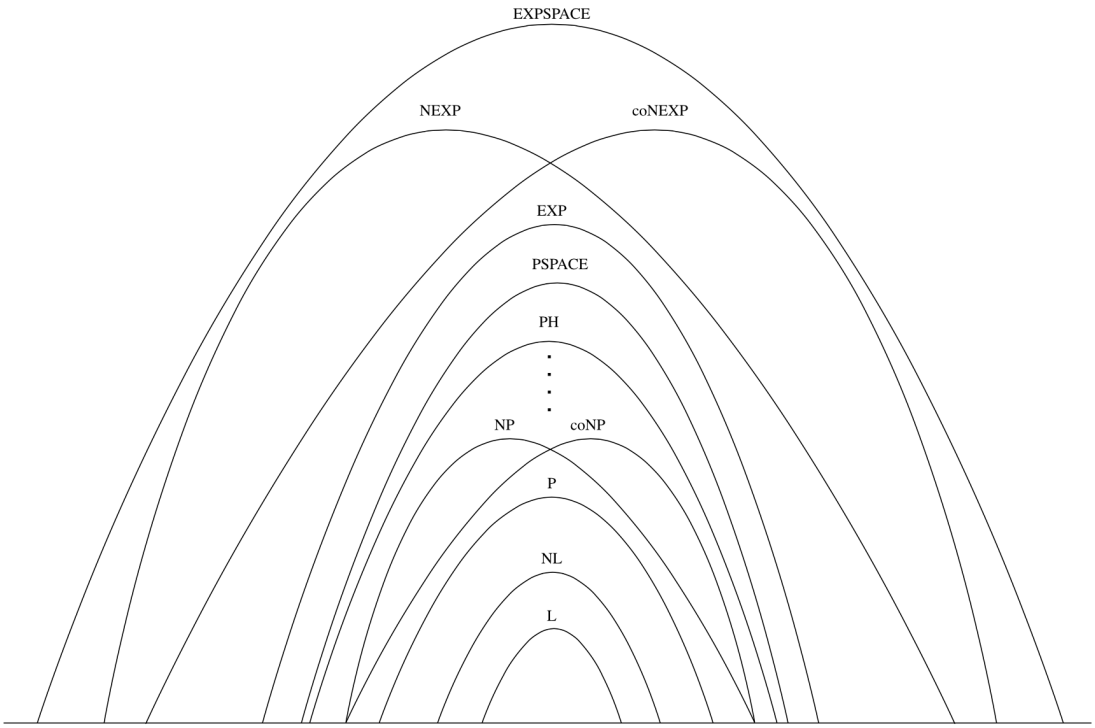

Figura 2.4: Relaciones de inclusión entre las clases de complejidad más comunes.

La clase de complejidad **P** es la que se atribuye a los problemas que se pueden resolver en tiempo polinomial empleando una Máquina de Turing determinista. Un problema de complejidad **NP** se puede resolver en tiempo polinomial empleando una Máquina de Turing no determinista. Esto es equivalente a decir que un problema **NP** es verificable en tiempo polinomial, es decir, dado un candidato a solución, se puede verificar que dicho candidato es correcto en tiempo polinomial. Dentro de esta clase, podemos diferenciar otras dos clases, **P**, por la relación de inclusión antes mencionada, y **NP**-completo. Los problemas **NP**-completos

[2]Imagen recuperada del portal *Standford Encyclopedia of Philosophy*. https://plato.stanford.edu/entries/computational-complexity/

son aquellos en los que podemos codificar o traducir cualquier otro problema en la clase **NP** empleando un tiempo polinomial para dicha traducción.

El problema de hallar los *answer sets* de un programa lógico normal (sin disyunción) es **NP**-completo, es decir, igual de complejo que el clásico problema de satisfactibilidad booleana (SAT). En el caso de un programa lógico disyuntivo (el caso de ASP), el problema es $\mathbf{NP^{NP}}$-completo (o $\mathbf{\Sigma_2^P}$). Esto significa que aunque una subrutina de complejidad **NP** se resolviese en tiempo constante, el problema tendría complejidad **NP**.

2.2 Especificaciones epistémicas

El lenguaje de especificaciones epistémicas es una ampliación del lenguaje de ASP propuesto por Michael Gelfond en 1991 [5]. Esta ampliación viene motivada por la limitación de ASP para modelar conocimiento ante la presencia de información incompleta. Para ejemplificar esta limitación, Gelfond emplea el *Scholarship Elegibility Problem*. Este problema consiste en una colección de reglas

$$eligible(X) \leftarrow highGPA(X) \tag{2.62}$$

$$eligible(X) \leftarrow minority(X),\ fairGPA(X) \tag{2.63}$$

$$-eligible(X) \leftarrow -fairGPA(X) \tag{2.64}$$

$$interview(X) \leftarrow not\ eligible(X),\ not\ -eligible(X) \tag{2.65}$$

empleadas por una universidad para otorgar becas a sus estudiantes, representados por la variable X. Los predicados $fairGPA(X)$ y $highGPA(X)$ indican que el estudiante X ha tenido calificaciones normales o altas, respectivamente, mientras que $minority(X)$ indica que X pertenece a alguna minoría social que se pretende favorecer con la concesión de becas. Las tres primeras reglas deciden si un estudiante es o no elegible para recibir la beca, mientras que la cuarta regla se puede ver como la formalización de la frase: *"Los estudiantes cuya elegibilidad no pueda ser determinada por las tres primeras reglas deben ser entrevistados por el comité universitario."*

Supongamos que contamos con la regla

$$fairGPA(mike),\ highGPA(mike) \tag{2.66}$$

que representa que la nota promedio de $mike$ es normal o alta (recordemos que la coma en cabeza de regla es una disyunción). Teniendo en cuenta que no hay evidencia de que se encuentre en una situación de minoría, el programa formado por las reglas (2.62)-(2.66) tiene

dos *answer sets*:

$$\{highGPA(mike),\ eligible(mike)\}$$
$$\{fairGPA(mike),\ interview(mike)\}$$

Esto revela que el programa no puede establecer la elegibilidad de $mike$ y aun así no puede confirmar que se le deba entrevistar, ya que $interview(mike)$ solo es cierto en uno de los *answer sets*. La razón es que la regla (2.65) es demasiado débil para representar la frase deseada.

Para solucionar este problema, Gelfond define el lenguaje de especificaciones epistémicas como una ampliación del lenguaje ASP que permite el uso de los operadores modales $\boldsymbol{K}$ y $\boldsymbol{M}$ en el cuerpo de las reglas.

Dado un literal lit:

- La expresión $\boldsymbol{K}lit$ representa que *"se sabe que* lit *es cierto"*, o lo que es lo mismo, lit es cierto en todos los *answer sets* del programa.
- La expresión $\boldsymbol{M}lit$ representa que se *"se cree que* lit *es cierto"*, o lo que es lo mismo, lit es cierto en algún *answer set* del programa.

Estos operadores, mantienen las siguientes relaciones de equivalencia:

$$\boldsymbol{K}atom \equiv not\ \boldsymbol{M}\ not\ atom \tag{2.67}$$
$$not\ \boldsymbol{K}atom \equiv \boldsymbol{M}\ not\ atom \tag{2.68}$$
$$\boldsymbol{K}\ not\ atom \equiv not\ \boldsymbol{M}atom \tag{2.69}$$
$$not\ \boldsymbol{K}\ not\ atom \equiv \boldsymbol{M}atom \tag{2.70}$$

lo que puede reducir la sintaxis del lenguaje al uso de un único operador.

De esta manera, en el ejemplo anterior, la regla (2.65) pasaría a ser

$$interview(X) \leftarrow not\ \boldsymbol{K}eligible(X),\ not\ \boldsymbol{K}{-}eligible(X) \tag{2.71}$$

Este tipo de fórmulas que van precedidas por uno de los operadores modales, se denominan fórmulas subjetivas, mientras que las fórmulas que no van precedidas por un operador modal se denominan fórmulas objetivas. Un átomo no precedido por un operador modal se denomina átomo objetivo y un literal objetivo representa a un átomo objetivo o a su negación fuerte (o clásica).

En el lenguaje de especificaciones epistémicas, un programa epistémico (o especificación epistémica) es un conjunto de reglas con la siguiente estructura:

$$F \leftarrow G_1, \ ..., \ G_m$$

donde F es una fórmula objetiva y cada una de las G puede ser una fórmula objetiva o subjetiva.

Los modelos de un programa lógico epistémico son sus *world views*. Un *world view* es una colección de conjuntos de literales del lenguaje de especificaciones epistémicas. La definición de *world view* varía en función de la semántica empleada.

2.2.1 Semántica G91

La semántica propuesta por Gelfond se conoce como G91. En dicha semántica, se define un *world view* W como un conjunto de answer sets que cumple la propiedad que especificamos debajo. Empezamos diciendo que un conjunto de interpretaciones proposicionales W cumple el literal $\mathbf{K}lit$ si lit es cierto en todas las interpretaciones $I \in W$ y que cumple $not\ \mathbf{K}lit$ si lit es falso en alguna de las interpretaciones $I \in W$.

Sea W un conjunto de interpretaciones proposicionales y P una especificación epistémica (programa lógico con operadores epistémicos). El *reducto epistémico* P^W es el resultado de eliminar toda regla en la que alguno de sus literales subjetivos no se cumpla en W y de eliminar el resto de literales subjetivos de las reglas restantes. Es fácil ver que el resultado del reducto, P^W es un programa ASP no epistémico.

Definición 1 (Gelfond 1991 [5]) *Decimos que un conjunto de interpretaciones proposicionales W es un* world view *de una especificación epistémica P si W es precisamente el conjunto de answer sets de P^W.*

Por ejemplo, supongamos la especificación epistémica:

$$\{a\} \tag{2.72}$$

$$b \tag{2.73}$$

$$c \leftarrow \boldsymbol{M}a \tag{2.74}$$

Si suponemos que $\boldsymbol{M}a$ es cierto, el conjunto de interpretaciones del programa es:

$$\{\{a, b, c\}, \{b\}\}$$

En consecuencia, el reducto epistémico es el siguiente:

$$\{a\} \tag{2.75}$$
$$b \tag{2.76}$$
$$c \tag{2.77}$$

El conjunto de *answer sets* de este reducto es:

$$\{\{a, b, c\}, \{b, c\}\}$$

Dado que el conjunto de *answer sets* no coincide con el conjunto de interpretaciones, este conjunto de interpretaciones no es un *world view* de la especificación epistémica de entrada. En cambio, si suponemos que $\boldsymbol{M}a$ es falso, el conjunto de interpretaciones del programa es:

$$\{\{a, b\}, \{b\}\}$$

Por lo tanto, el reducto epistémico correspondiente es el siguiente:

$$\{a\} \tag{2.78}$$
$$b \tag{2.79}$$

El conjunto de *answer sets* de este reducto es:

$$\{\{a, b\}, \{b\}\}$$

Puesto que el conjunto de *answer sets* coincide con el conjunto de interpretaciones, dicho conjunto de interpretaciones es un *world view* de la especificación epistémica de entrada.

2.2.2 Semántica K14

En su tesis doctoral [21], Patrick Thor Kahl, definió una semántica alternativa a G91 para el cálculo de *world views* de un programa lógico epistémico, la semántica K14.

La principal diferencia que presenta dicha semántica con G91 es el proceso de generación del reducto epistémico. En este caso, sea W un conjunto de interpretaciones proposicionales y P una especificación epistémica (programa lógico con operadores epistémicos). El *reducto epistémico* P^W es el resultado de eliminar toda regla en la que alguno de sus literales subjetivos no se cumpla en W y de eliminar el resto de literales subjetivos de las reglas restantes, a excepción de los literales de la forma $\boldsymbol{K}lit$, que son reemplazados por lit, y los literales de la forma $not\ \boldsymbol{M}lit$, que son reemplazados por $not\ lit$.

A partir de este reducto epistémico, la definición de *world view* es la misma que en G91.

2.2.3 Aplicaciones

Como ya hemos visto, el lenguaje de especificaciones epistémicas nace con el objetivo de mejorar el razonamiento en programas lógicos con disyunción donde existe información incompleta. A través del *Scholarship Elegibility Problem*, Gelfond resaltó la utilidad de esta ampliación del lenguaje ASP para dichos escenarios. Con ello, en su publicación, también remarcó la posibilidad de utilizar el lenguaje para formalizar la *Closed World Assumption (CWA)* en bases de datos deductivas con disyunción o para representar las restricciones de integridad.

No obstante, además de las aplicaciones ya mostradas por ASP, que sufrirían un incremento de su capacidad computacional, destacamos aquellos problemas que radican en obtener información común a múltiples interpretaciones de un escenario. Dentro de esta categoría englobamos dos problemas de ejemplo: el *Critical and Irrelevant Paths Problem* y el problema de *Conformant Planning*.

Critical and Irrelevant Paths Problem

En este escenario, contamos con un grafo dirigido, definido por sus nodos y aristas, un nodo inicial y un nodo final. El objetivo del problema es identificar las aristas críticas e irrelevantes del grafo. Una arista crítica es aquella que, independientemente del camino a seguir para llegar al nodo meta, siempre se toma. Una arista irrelevante es aquella que, bajo la misma premisa que la arista crítica, nunca se toma. Por ejemplo, dado el grafo de la figura 2.5, si tomamos como nodo inicial al nodo 3 y como meta al nodo 1, la solución al problema consistiría en identificar como crítica a la arista (0,1) y como aristas irrelevantes a las aristas (1,9), (6,0), (6,1), (6,4), (7,9), (8,7) y (8,9).

Conformant Planning

La Planificación Automática, es el área de la Inteligencia Artificial que se encarga de la generación de estrategias para conseguir una meta en un entorno dinámico. En la versión clásica del problema de planificación se conoce totalmente el estado inicial y los efectos de las acciones son totalmente deterministas. No obstante, no todas las situaciones se encuentran bajo estas premisas, pues un agente puede encontrarse en escenarios con información incompleta. Bajo este contexto, se generan dos nuevos tipos de planificación: el *conditional planning* [22] y el *conformant planning*. Nosotros nos centraremos en el segundo.

El *conformant planning*, es una versión del problema clásico de planificación que consiste en obtener un plan que garantiza la meta independientemente del no determinismo del es-

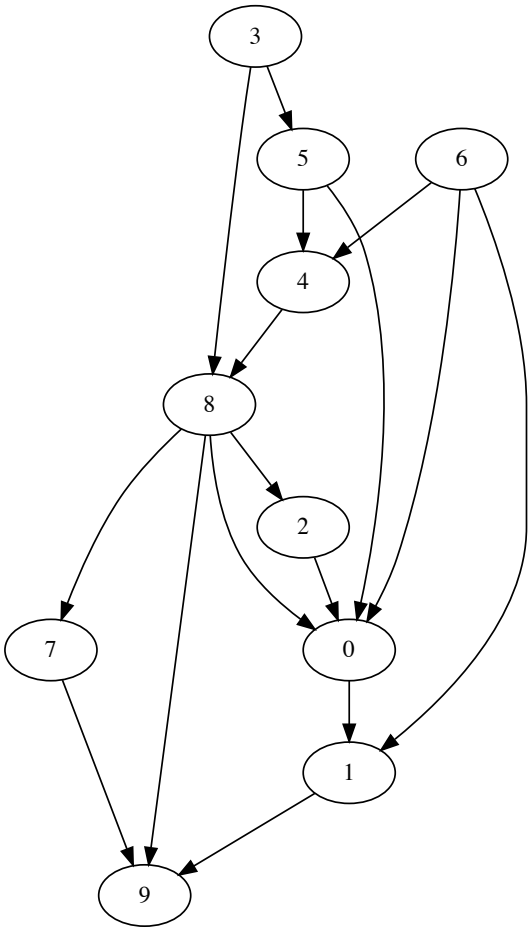

Figura 2.5: Escenario del *Critical and Irrelevant Paths Problem.*

cenario (estado inicial incompleto o acciones no deterministas). En este tipo de escenarios, la complejidad computacional de ASP no es suficiente, pues no es capaz de razonar sobre el conjunto de planes. En realidad, en un escenario de este estilo, ASP nos ofrece una serie de *answer sets* que se corresponden con distintas secuencias de acciones que nos llevan a la meta. No obstante, estas secuencias no se corresponden con un conjunto de planes, sino con un conjunto de explicaciones. Veamos un ejemplo sencillo a través del *Yale Shooting Problem* en su versión epistémica. Para ello, tenemos que realizar una serie de modificaciones en el programa formado por las reglas (2.41)-(2.58).

En primer lugar, modificamos la restricción que obliga a llegar a la meta, la regla (2.56), de manera que esta meta debe cumplirse sea cual sea el estado inicial:

$$\leftarrow \ not\ \boldsymbol{K} goal \tag{2.80}$$

El grupo de reglas que generan las acciones posibles, reglas (2.57)-(2.58), se cambian por las siguientes:

$$occurs(A,S) \ \leftarrow \ not\ \boldsymbol{K}\ not\ occurs(A,S),\ stepless(S),\ action(A) \tag{2.81}$$

$$occurs(S) \ \leftarrow \ \boldsymbol{K} occurs(A,S),\ action(A),\ stepless(S) \tag{2.82}$$

$$\leftarrow \ not\ occurs(S),\ stepless(S) \tag{2.83}$$

$$\leftarrow \ occurs(A,S),\ not\ \boldsymbol{K} executable(A,S) \tag{2.84}$$

La regla (2.81) es de tipo selección (*choice rule*) y permite generar dos casos posibles: o bien decidimos incluir la ejecución de la acción A en el instante S en todas las trazas alternativas (cada traza corresponde a un estado inicial distinto), o bien decidimos no incluir la ejecución de esa acción en ninguna de las trazas. Dicho de otro modo, una acción ejecutada en un plan se ejecuta del mismo modo en todas las posibles alternativas (si disparo el arma, lo hago tanto si en el estado inicial estaba cargada como si estaba descargada). La regla (2.82) usa el predicado auxiliar $occurs(S)$ (con un solo argumento) para registrar el hecho de que al menos una acción ha sido ejecutada en el estado S, mientras que la regla (2.83) se usa precisamente para garantizar que, en efecto, ejecutamos siempre al menos una acción en cada paso. Por último, la regla (2.84) prohíbe ejecutar una acción si esta no es ejecutable en todas las posibles alternativas contempladas. Por ejemplo, no podemos empezar cargando el arma si desconocemos el estado inicial, ya que no es posible cargar en la alternativa en la que el arma estaba ya cargada inicialmente.

Finalmente añadimos la versión epistémica de la regla que prohíbe las acciones concu-

rrentes:

$$\leftarrow \boldsymbol{K}occurs(A,S),\ \boldsymbol{K}occurs(B,S),\ action(A),\ action(B),\ stepless(S),\ A!=B \quad (2.85)$$

Si tomamos como estado inicial que el objetivo está vivo y el arma cargada, el plan generado por ASP y el *conformant plan* es idéntico. Se trata del plan de la figura 2.2.

Si cambiamos el estado inicial, y suponemos que el objetivo está vivo pero el arma se encuentra descargada, de nuevo, el plan generado por ASP y el *conformant plan* coinciden. El plan corresponde al de la figura 2.3.

El problema aparece cuando desconocemos el estado del fluente $loaded$. Si calculamos los *answer sets* del problema codificado en ASP, obtendremos dos posibles "planes". Tal y como se puede ver en la figura 2.6, el primer "plan" consiste en suponer que nos encontramos en el primer escenario alternativo del problema y, por lo tanto, el plan es el de la figura 2.2. El segundo "plan", parte de la base de que el arma está descargada y, en consecuencia, nos encontramos en el segundo escenario. Este segundo "plan" es el de la figura 2.3. El conjunto de *answer sets* generados es:

$$\{\{occurs(pull\_trigger,0)\},\ \{occurs(load,0),\ occurs(pull\_trigger,1)\}\}$$

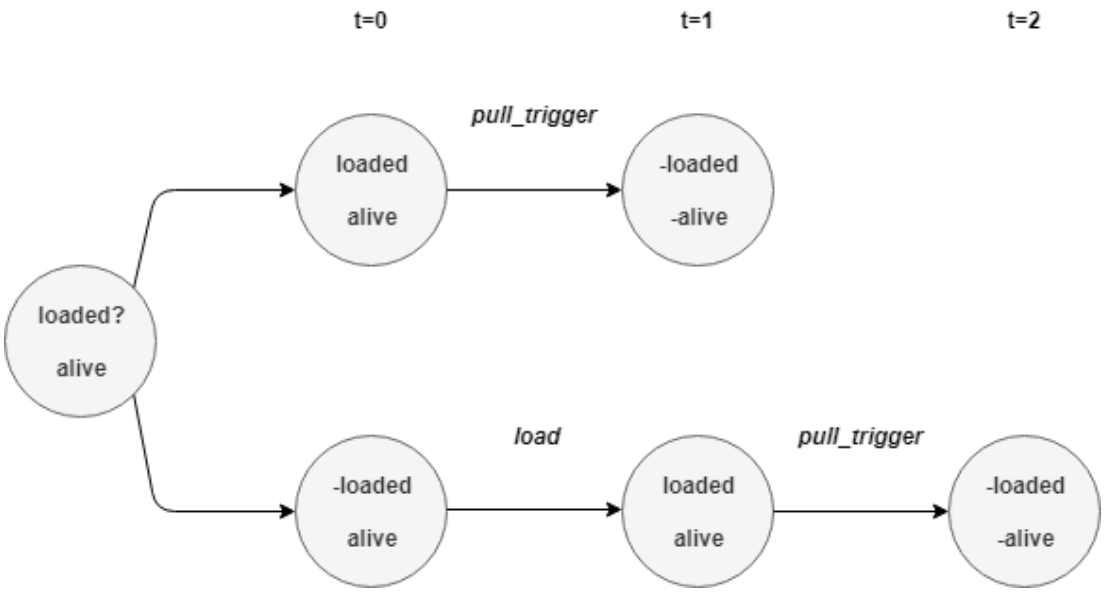

Figura 2.6: Planes ASP para el *Yale Shooting Problem* con el estado inicial del arma desconocido.

Sin embargo, si calculamos el *conformant plan* de este problema, este es único. En la figura 2.7, se puede observar la secuencia de acciones que garantiza la muerte del objetivo:

1. Apretamos el gatillo. Si se encontraba cargada, habremos descargado el arma y matado al objetivo. En caso contrario, no sucede nada. En cualquier caso garantizamos que el arma queda descargada.
2. Cargamos el arma. El estado del fluente $alive$ continúa desconocido.
3. Apretamos el gatillo de nuevo. Si el objetivo no había muerto la primera vez que apretamos el gatillo, muere en este momento. En este punto se garantiza que el valor del fluente $alive$ es falso.

Esto es equivalente al modelo:

$$\{\boldsymbol{K}occurs(pull\_trigger, 0),\ \boldsymbol{K}occurs(load, 1),\ \boldsymbol{K}occurs(pull\_trigger, 2)\}$$

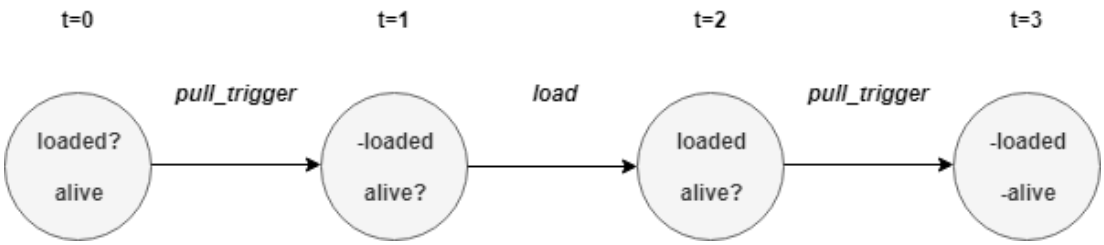

Figura 2.7: *Conformant plan* para el *Yale Shooting Problem* con el estado inicial del arma desconocido.

2.2.4 Complejidad computacional

El problema de calcular los modelos de un programa lógico epistémico, tiene clase de complejidad $\boldsymbol{\Sigma_3^P}$, tal y como así lo demuestra Truszczynski en [14]. Esto significa que si una llamada a una subrutina de complejidad $\boldsymbol{\Sigma_2^P}$ se realizase en tiempo constante, el problema sería de complejidad **NP**.

2.3 clingo

`clingo` [16] es una de las herramientas de la "Potsdam Answer Set Solving Collection" (Potassco). Está escrita en el lenguaje de programación C++ y publicada bajo la licencia GNU General Public License. En el momento de redacción de este documento, su versión más reciente es la 5.3.0.

Se trata de un sistema que integra el *grounder* `gringo` con el *solver* `clasp`. En la figura 2.8, se puede observar el diagrama que corresponde al proceso de cómputo de las soluciones de un programa ASP empleando `clingo`.

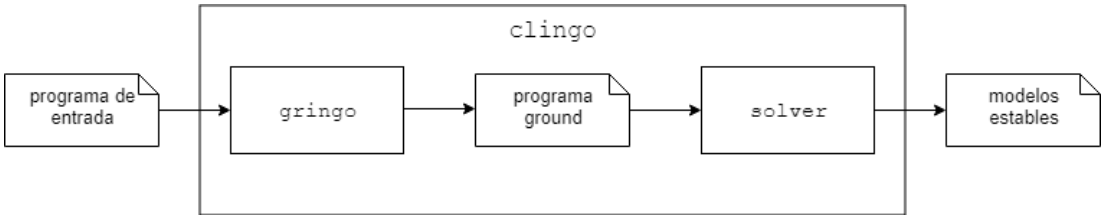

Figura 2.8: Proceso de cómputo de las soluciones de un programa ASP con `clingo`.

Uno de los motivos que hace que `clingo` sea uno de los sistemas ASP más populares en la actualidad, es la eficiencia de su *solver* `clasp`. Esta eficiencia ha sido demostrada con numerosos podios en las competiciones de *solvers* más relevantes como *ASP competition* o *Configurable SAT Solver Challenge*.

El segundo motivo de su éxito, es su lenguaje de entrada, que ofrece características tan expresivas como el uso de *aggregates*, reglas de cardinalidad o sentencias de optimización.

Finalmente, el tercer motivo que hace que `clingo` sea interesante, es su flexibilidad en los procesos de *grounding* y *solving*. `clingo` cuenta con librerías de integración para los lenguajes de scripting Lua y Python, lo que ofrece control total sobre el flujo de ejecución del sistema. Esta característica se conoce como *multi-shot solving* [23].

2.3.1 Lenguaje de entrada

Un programa lógico en `clingo`, se construye a partir de una serie de términos. Los términos más básicos son los enteros, las constantes y las variables. Las funciones o predicados, se componen de otros términos. El lenguaje de `clingo` también acepta un tipo especial de variables, las variables anónimas. Estas variables, que se representan con el símbolo _, son similares a una variable normal, con la diferencia de que cada ocurrencia de estas variables se trata como una nueva variable.

Los tres tipos principales de reglas se definen como:

$$A_0 \ :- \ L_1, \ ..., \ L_n. \tag{2.86}$$

$$A_0. \tag{2.87}$$

$$:- \ L_1, \ ..., \ L_n. \tag{2.88}$$

tal que (2.86) es una regla estándar, (2.87) es un hecho y (2.88) es una restricción. A_0 es un átomo A tal que A es una función o constante y los L_i son literales de la forma A o $not\ A$. El conjunto de literales $\{L_1,\ ...,\ L_n\}$ se conoce como el cuerpo de la regla. La negación clásica o fuerte se denota por el símbolo $-$, la disyunción en la cabeza con la coma (,) y el símbolo :- representa la implicación.

Estas reglas, deben de cumplir la propiedad de *safety*. Esta propiedad consiste en que todas las variables que aparecen en dicha regla deben aparecer en algún literal positivo del cuerpo. Si una variable aparece en un predicado positivo en el cuerpo, entonces decimos que este predicado liga la variable y, por lo tanto, se vuelve *safe*. Por ejemplo, la regla:

$$a(X)\ :-\ not\ b(X). \tag{2.89}$$

no es *safe*, pues la variable X no aparece en ningún literal positivo del cuerpo de dicha regla. Si modificamos el cuerpo de la regla, de manera que añadimos un predicado positivo que incluye dicha variable:

$$a(X)\ :-\ not\ b(X),\ dom(X). \tag{2.90}$$

entonces la regla pasaría a ser *safe*. Si una variable es *unsafe*, `gringo`, el *grounder* de `clingo`, no es capaz de determinar su valor.

Adicionalmente, `clingo` acepta los siguientes predicados de comparación: $==$ (igual que), $!=$ (distinto de), $<$ (menor que), $<=$ (menor o igual que), $>$ (mayor que) y $>=$ (mayor o igual que). Además, permite el uso de funciones aritméticas, asignaciones, intervalos, condiciones, *aggregates* o sentencias de optimización. Entre las sentencias que tratan sobre un programa, destaca el uso de predicados de ocultación (directivas $show$ y $hide$) y las definiciones de constantes, entre otras.

2.3.2 Multi-shot solving

El *multi-shot solving*, cambia el paradigma de funcionamiento de los sistemas de cómputo ASP. En lugar de lanzar una ejecución secuencial que empieza, computa los *answer sets* y termina, estos procesos permanecen accesibles y mantienen la información con el objetivo de volver a ejecutar los procesos de *grounding* y *solving* si fuese necesario. De esta manera, se dota a estos procesos de estructuras con un estado interno que puede ser manipulado a través de una serie de operaciones. Estas operaciones permiten añadir programas, ejecutar el proceso de *grounding* o solving o establecer el valor de verdad de *external atoms* entre otras cosas. Los *external atoms* son un tipo especial de átomos cuyo valor de verdad no viene dado por el programa lógico en el que aparecen. Este valor de verdad se declara como externo, por lo que `clingo` no puede derivar su valor de certeza ni suponerlo falso por defecto. De

este modo, evita la simplificación de reglas en el proceso de *grounding*, lo que permite activar y desactivar reglas a medida que avanza el proceso de razonamiento cambiando el valor de verdad de estos *external atoms*.

Estas características, hacen de `clingo` un sistema ideal para funcionar como *backend* en sistemas de razonamiento complejo, como en el caso de problemas de optimización, o en entornos interactivos, como en robótica.

Capítulo 3

eclingo

La herramienta desarrollada, `eclingo`, calcula los modelos de un programa lógico epistémico. Se ha escrito sobre el lenguaje de programación `Python` (en su versión 3.6.9) y hace uso del sistema ASP `clingo` (en su versión 5.3.0) como *backend*.

Su arquitectura es un sencillo *pipe-and-filter* con dos filtros (véase figura 3.1), el módulo $parse$ y el módulo $solve$. Este *pipe-and-filter* cuenta con una sencilla interfaz de línea de comandos (CLI) que se encarga de procesar los argumentos de entrada y mostrar la salida del sistema.

El argumento principal de `eclingo` es la ruta a los ficheros que conforman el programa de entrada. Este programa de entrada consiste en un conjunto de reglas ASP en formato `clingo` cuyo cuerpo puede incluir literales subjetivos. Dichos literales subjetivos se representan utilizando la siguiente notación:

$$\begin{aligned}
\mathbf{K}\ atom &\equiv \&k\{\ atom\ \} \\
\mathbf{K}\ not\ atom &\equiv \&k\{\ \sim atom\ \} \\
\mathbf{K}\ -atom &\equiv \&k\{\ -atom\ \} \\
\mathbf{K}\ not\ -atom &\equiv \&k\{\ \sim -atom\ \}
\end{aligned}$$

donde el símbolo ~ representa la negación por defecto, el símbolo - representa la negación fuerte (o clásica) y $atom$ es un átomo objetivo.

El operador modal $\boldsymbol{M}$, no se incluye en la representación ya que este se puede definir a partir del operador $\boldsymbol{K}$ siguiendo la regla (2.70) y, por lo tanto, su inclusión no aumenta la capacidad computacional de la herramienta.

Además, `eclingo` admite directivas $show$ en el programa de entrada, sentencias que especifican qué literales subjetivos deben aparecer en los modelos resultado. Estas siguen la

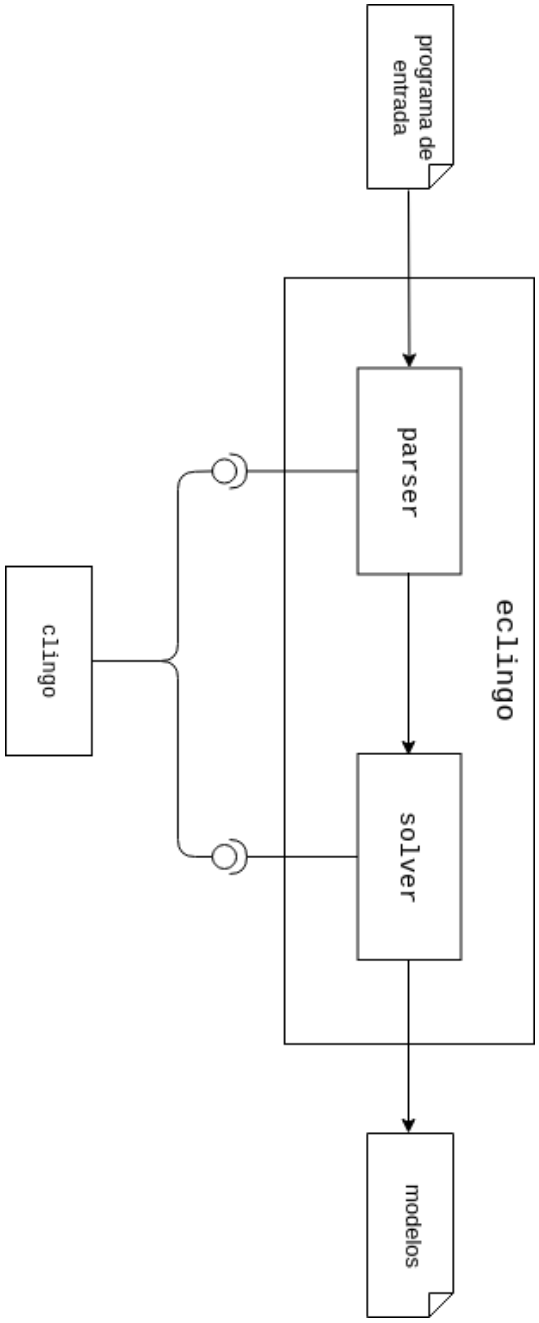

Figura 3.1: Arquitectura de `eclingo`

estructura:

$$\#show\ atom/arity.$$

donde $atom$ es un átomo objetivo y $arity$ es su aridad [1]. Con esta definición, dada la directiva:

$$\#show\ p/1.$$

se mostrarán en los *answer sets* del programa aquellos literales subjetivos que toman el valor cierto cuya estructura es $\boldsymbol{K}p(X)$, donde X es cualquier argumento.

Por ejemplo, una codificación del escenario base del *Yale Shooting Problem*, mostrando solo las acciones ejecutadas, en `eclingo` sería la que podemos ver en la figura 3.2.

```
stepless(0..length-1).

%%% Inertial fluents are fluents
fluent(F) :- inertial(F).

%%%% inertia %%%%
holds(F,S+1) :- inertial(F), stepless(S), holds(F,S),not
    -holds(F,S+1).
-holds(F,S+1):- inertial(F), stepless(S),-holds(F,S),not
    holds(F,S+1).

%%% non-concurrent actions
:- occurs(A,S), occurs(B,S), A!=B.

%%%% initial state: fluents false by default
-holds(F,0):-not  holds(F,0),fluent(F).

%%%%% goal  %%%%%%%
goal :- -holds(alive,length), not imposs.

%%%% epistemic %%%%%%%
imposs :- occurs(A,S), not executable(A,S).
:- not &k{goal}.

occurs(A,S) :- not &k{ ~ occurs(A,S) }, stepless(S), action(A).

occurs(S) :- &k{occurs(A,S)}, action(A), stepless(S).
:- not occurs(S), stepless(S).

:- &k{occurs(A,S)}, &k{occurs(B,S)}, action(A), action(B),
    stepless(S), A!=B.
```

Figura 3.2: *yale.lp*

[1] Número de argumentos de una función o predicado.

La salida mostrada por la CLI consiste en los modelos del programa de entrada. Los modelos producidos consisten en un conjunto de literales subjetivos que se calculan a partir de los *world views* del programa. Si el programa no incluye directivas $show$, este conjunto está formado por los literales subjetivos de cada *world view* calculado. Por ejemplo, si el programa contiene el literal subjetivo $\mathbf{K}c$ y tenemos el *world view*:

$$\{\, \{\, a,\ b,\ c,\ d\,\},\ \{a,\ c\}\,\}$$

en el que c siempre es cierto, el resultado producido sería:

$$\{\boldsymbol{K}c\}$$

El formato de esta salida se pude ver en la figura 3.3, que muestra el resultado del modelo que hemos comentado. En este formato, se muestra cada modelo como una *answer* del programa. Si el programa tiene algún modelo, se muestra el mensaje **SATISFIABLE**, mientras que en caso de no tener modelos, el mensaje mostrado es **UNSATISFIABLE**. La salida va acompañada de un valor que indica el tiempo, expresado en segundos, empleado por `eclingo` para procesar y resolver el programa de entrada.

```
Answer: 1
&k{ c }
SATISFIABLE

Elapsed time: 0.001238 s
```

Figura 3.3: Formato de salida de `eclingo`

Si el programa incluye directivas $show$, los modelos estarán formados por los literales subjetivos especificados cuyo valor de verdad sea *true* en cada *world view*.

A continuación, se describen los módulos de `eclingo` y sus procedimientos en las secciones 3.1 y 3.2.

3.1 Módulo parser

Es el primer filtro del *pipe-and-filter*. Se encarga de analizar sintácticamente el programa de entrada y traducirlo a una representación auxiliar soportada por `clingo`, el sistema empleado como *backend*. Esta funcionalidad se invoca a través de su única función pública $parse$.

Los argumentos de entrada a la función son:

- Una lista de rutas a los ficheros que conforman el programa epistémico.
- Una lista de constantes a definir en el programa.
- Un valor booleano que indica la semántica sobre la que se computarán los *world views*.
- Un valor entero que indica la versión de mejora empleada en el cómputo.

El procedimiento que ejecuta la función se divide en cinco fases: preprocesado, definición de constantes, adición de reglas *choice*, adición de directivas de proyección y mejora.

3.1.1 Preprocesado

Se comienza instanciando dos objetos $Control$ de la librería *clingo*, que servirán como interfaz con `clingo` durante el proceso de cómputo de los *world views*. El primero se empleará como generador de candidatos a modelo, de aquí en adelante $candidates\_gen$, y el segundo como evaluador de estos candidatos, de aquí en adelante $candidates\_test$. En el método de inicialización se especifican las opciones 0 y $--project$ para $candidates\_gen$ y la opción 0 para $candidates\_test$.

- La opción 0 indica que las llamadas al método $solve$ devolverán todos los modelos del programa almacenado en el estado del objeto $Control$.
- La opción $--project$ configura al objeto para proyectar el resultado de las llamadas a $solve$ sobre los átomos especificados a través de una serie de directivas en el programa.

El siguiente paso consiste en analizar sintácticamente los ficheros que conforman el programa de entrada. Durante este análisis se reemplazan los literales subjetivos de entrada por literales objetivos auxiliares, con el objetivo de que `clingo` acepte el programa como entrada. Este reemplazo viene determinado por la semántica que se indica como parámetro. En el

caso de la semántica G91, la política de reemplazo es la siguiente:

$$\&k\{\ atom\ \} \longmapsto not\ not\ aux\_atom \tag{3.1}$$
$$not\ \&k\{\ atom\ \} \longmapsto not\ aux\_atom \tag{3.2}$$
$$\&k\{\ \sim atom\ \} \longmapsto aux\_not\_atom \tag{3.3}$$
$$not\ \&k\{\ \sim atom\ \} \longmapsto not\ aux\_not\_atom \tag{3.4}$$
$$\&k\{\ -atom\ \} \longmapsto aux\_sn\_atom \tag{3.5}$$
$$not\ \&k\{\ -atom\ \} \longmapsto not\ aux\_sn\_atom \tag{3.6}$$
$$\&k\{\ \sim -atom\ \} \longmapsto aux\_not\_sn\_atom \tag{3.7}$$
$$not\ \&k\{\ \sim -atom\ \} \longmapsto not\ aux\_not\_sn\_atom \tag{3.8}$$

Para la semántica K14, el reemplazo es el mismo para todos los casos, a excepción de (3.1) que pasa a ser:

$$\&k\{\ atom\ \} \longmapsto aux\_atom,\ atom \tag{3.9}$$

Este análisis sintáctico, se realiza a través de la función $parse\_program$ de la librería $clingo$, que recibe una expresión en formato `clingo` y devuelve su árbol sintáctico. Este árbol sintáctico se pasa a una función privada que se encarga de traducir y cargar las reglas en $candidates\_gen$ y $candidates\_test$ a la vez que almacena las directivas $show$ en una lista que llamaremos $show\_signatures$.

Una vez finalizado el preprocesado, se procede a definir las constantes especificadas por parámetro si las hubiese.

3.1.2 Definición de constantes

Si se reciben como parámetro, se procede a definir en el programa preprocesado las constantes especificadas. Para ello, se añaden al programa sentencias con la estructura:

$$\#const\ constant.$$

En ellas, $constant$ es una cadena de texto con el formato:

$$< id >=< term >$$

donde $< id >$ es el identificador de la constante y $< term >$ es el valor que toma. Por ejemplo, si queremos definir la constante $longitud$ con un valor de 10, la sentencia correspondiente es:

$$\#const\ longitud = 10.$$

A continuación, con el objetivo de generar distintos *world views* a partir de los valores de verdad de los literales subjetivos del programa, se añade una regla *choice* al programa por cada átomo auxiliar.

3.1.3 Adición de reglas *choice*

A través del método $backend$ de un objeto $Control$, podemos extender el programa lógico almacenado en el estado de dicho objeto. Este método nos devuelve un objeto $backend$ cuyo método add recibe una regla *ground* en formato ASPIF [2] como parámetro y la añade al programa. En este formato, cada literal del programa recibe un código numérico que lo identifica. Esta asignación se realiza durante el proceso de *grounding*, por lo que es necesario ejecutarlo para obtener los valores numéricos correspondientes.

No obstante, no podemos realizar el *grounding* del programa actual directamente, pues este proceso realiza una serie de simplificaciones en el programa en base a su estructura sintáctica. El objetivo de estas simplificaciones es reducir el espacio de búsqueda en el proceso de *solving*.

En concreto, la modificación que nos afecta, es aquella que consiste en eliminar del programa aquellas reglas que contienen en el cuerpo un átomo que no aparece en la cabeza de ninguna regla, es decir, que no puede evaluarse como cierto. Por ejemplo, dado el programa:

$$a. \tag{3.10}$$

$$b\ :-\ a. \tag{3.11}$$

$$c\ :-\ e. \tag{3.12}$$

$$d\ :-\ not\ e. \tag{3.13}$$

podemos fácilmente deducir que, puesto que no aparece en la cabeza de ninguna regla, e toma el valor falso por defecto. En consecuencia, esto significa que:

$$e \leftarrow \bot$$

en todo el programa. Así, las reglas que contienen e en su cuerpo, como (3.12) se eliminan,

[2] Formato empleado internamente por `clingo` para el cálculo de los *answer sets* de un programa lógico.

pues una conjunción de fórmulas toma el valor falso si uno de las fórmulas es falsa. Del mismo modo, $not\ e$ se vuelve cierto, por lo que podemos convertir la regla (3.13) en un hecho d.

Dado que los literales subjetivos del programa de entrada solo aparecen en el cuerpo de las reglas, los átomos auxiliares que los representan nunca aparecen en cabeza y, por lo tanto, al ejecutar el *grounding* serían simplificados.

Para evitar esta simplificación, se añade una regla por cada átomo auxiliar que protege a este durante el proceso de *grounding*. La cabeza de la regla es el átomo auxiliar en cuestión y su cuerpo está formado por:

- El literal objetivo al que hace referencia el literal subjetivo.
- Un *external atom* que se empleará para desactivar la regla una vez finalizado el proceso.

Además, si el átomo auxiliar representa un literal subjetivo que indica negación de un literal objetivo, esta regla también incluye el cuerpo positivo de la regla en la que aparece el literal subjetivo. El objetivo de esto es convertir la regla en una regla *safe*.

A continuación se invoca el método $ground$ de los objetos $Control$. Acto seguido, para cada uno de estos objetos, se obtienen los códigos ASPIF asociados a los átomos auxiliares observando el atributo $symbolic\_atoms$ de cada objeto. Con el código correspondiente, se añade una regla *choice* con la siguiente estructura:

$$\{\ aux\_atom\ \}.$$

donde aux_atom es el respectivo átomo auxiliar.

Además, durante la observación del atributo $symbolic\_atoms$, se construye un diccionario [3] $epistemic\_atoms$ donde las claves son los átomos auxiliares del programa y sus valores los átomos objetivos a los que hacen referencia.

Para finalizar esta fase del procedimiento, se llama al método $cleanup$ de los objetos $Control$. Este método falsea los *external atoms* del programa y, en consecuencia, elimina las reglas auxiliares.

3.1.4 Adición de directivas de proyección

En este punto del flujo de ejecución, los objetos $Control$ contienen un programa ASP estándar, es decir, sin expresiones epistémicas. Una llamada al método $solve$ de estos objetos

[3] Estructura de datos clave-valor y tipo de dato en `Python`.

nos devolvería los modelos estables de este programa, que equivalen a conjuntos de creencias candidatos del programa epistémico de entrada. Sin embargo, lo que realmente nos interesa conocer es el valor de verdad de los literales subjetivos de cada *world view* candidato.

Una opción para conseguir este resultado podría consistir en agrupar los conjuntos de creencias por el valor de los literales subjetivos y así conformar los respectivos *world views*. De cada *world view* se obtendría un modelo candidato eliminando sus átomos objetivos.

Sin embargo, dado que $candidates\_gen$ se ha instanciado con la opción $--project$, se pueden proyectar los modelos resultado de la llamada al método $solve$ sobre los átomos auxiliares. De esta manera, el proceso anterior lo realiza `clingo`, devolviéndonos directamente los modelos candidatos. Por ejemplo, dado el programa lógico:

$$\{a\}. \tag{3.14}$$

$$\{b\}. \tag{3.15}$$

obtenemos 4 modelos estables:

$$\{\}$$
$$\{a\}$$
$$\{b\}$$
$$\{a,\ b\}$$

Si proyectamos el resultado sobre el átomo a, `clingo` nos devolverá dos modelos, $\{\}$ y $\{a\}$, que se diferencian únicamente en el valor de verdad de a, el átomo sobre el que se proyectó.

En consecuencia, añadimos al programa almacenado en $candidates\_gen$ directivas de proyección para cada átomo auxiliar siguiendo la estructura:

$$\#project\ aux\_atom/arity.$$
$$\#show\ aux\_atom/arity.$$

donde aux_atom es el respectivo átomo auxiliar y $arity$ su aridad. Las directivas $project$, indican que la proyección se efectúa sobre los átomos auxiliares, y las directivas $show$, indican que se han de mostrar estos mismos átomos auxiliares. Estas directivas $show$, ocultan los átomos objetivos.

Antes de finalizar el procedimiento, en caso haber sido indicado, se añade un conjunto de reglas que mejoran la eficiencia de la herramienta.

3.1.5 Mejora

Si el argumento de la función así lo indica, se añaden una serie de reglas que filtran el conjunto de modelos candidato. Estas reglas siguen la siguiente estructura:

$$:- aux\_lit,\ not\ lit.$$

donde aux_lit es un átomo auxiliar y $not\ lit$ es la negación por defecto del literal objetivo al que hace referencia el respectivo literal subjetivo.

El objetivo de esta regla es eliminar aquellos candidatos a modelo en los que el valor de verdad del átomo auxiliar y el literal objetivo al que hace referencia son inconsistentes. Por ejemplo, dado el programa lógico de entrada:

$$\{a\}. \tag{3.16}$$
$$b\ :-\ \&k\{a\}. \tag{3.17}$$

el programa resultado de la traducción es:

$$\{a\}. \tag{3.18}$$
$$b\ :-\ aux\_a. \tag{3.19}$$
$$\{aux\_a\}. \tag{3.20}$$

Si calculamos los modelos estables de este programa obtenemos 4 modelos:

$$\{\}$$
$$\{a\}$$
$$\{aux\_a,\ b\}$$
$$\{a,\ aux\_a,\ b\}$$

Como se puede observar, el modelo $\{aux\_a,\ b\}$ es inconsistente, pues está representando que *"a es cierto en todos los modelos"* en un modelo en el que a es falso.

Aquí, termina el procedimiento de la función *parse*, devolviendo como salida los objetos $candidates\_gen$ y $candidates\_test$, el diccionario $epistemic\_atoms$ y la lista de directivas $show\_signatures$. Esta salida, junto con el número máximo de modelos a computar especificado en la entrada a `eclingo`, conforma el conjunto de argumentos de entrada que recibe el segundo filtro del *pipe-and-filter*, el módulo $solver$.

3.2 Módulo solver

Es el segundo y último filtro del *pipe-and-filter*. Se encarga de calcular los modelos del programa epistémico de entrada haciendo uso de `clingo` como *backend*. Su funcionalidad se invoca a través de su única función pública $solve$.

Los argumentos de entrada a la función son:

- El objeto $Control$ que se empleará para generar los candidatos, $candidates\_gen$.
- El objeto $Control$ que se empleará para evaluar los candidatos, $candidates\_test$.
- El diccionario $epistemic\_atoms$.
- La lista de directivas $show\_signatures$.
- Un valor entero que indica el número máximo de modelos a computar.

El procedimiento que ejecuta la función se divide en tres pasos: generación de candidatos, evaluación de candidatos y selección de resultados.

3.2.1 Generación de candidatos

El procedimiento comienza con la generación de los candidatos a modelo. Para ello, se ejecuta el método $solve$ de $candidates\_gen$. Cada modelo resultado, está conformado únicamente por átomos auxiliares, siendo cada uno una configuración distinta de sus valores de verdad y, por lo tanto, representando a un *world view*.

A continuación, se evalúa cada candidato con el objetivo de eliminar del conjunto de candidatos aquellos que no cumplan las condiciones que representan sus literales subjetivos.

3.2.2 Evaluación de candidatos

Dado un modelo candidato, se construyen cuatro conjuntos a partir de los átomos auxiliares del programa traducción, que son las claves del diccionario $epistemic\_atoms$, y de los átomos auxiliares que pertenecen al candidato.

- Conjunto k_lits. Lo conforman los átomos auxiliares, que no representan negación de un literal objetivo, que pertenecen al modelo candidato.
- Conjunto k_not_lits. Lo conforman los átomos auxiliares, que representan negación de un literal objetivo, que pertenecen al modelo candidato.

- Conjunto not_k_lits. Lo conforman los átomos auxiliares, que no representan negación de un literal objetivo, que no pertenecen al modelo candidato.

- Conjunto $not\_k\_not\_lits$. Lo conforman los átomos auxiliares, que representan negación de un literal objetivo, que no pertenecen al modelo candidato.

Además, con estos conjuntos, se construye una lista que denominaremos $assumptions$ con una tupla por cada átomo auxiliar del diccionario $epistemic\_atoms$. Estas tuplas están compuestas por el átomo auxiliar y un valor booleano que indica si este aparece o no en el modelo candidato. De esta manera, los átomos auxiliares de los conjuntos k_lits y k_not_lits conforman tuplas donde el valor booleano es cierto, mientras que los átomos auxiliares de los conjuntos not_k_lits y $not\_k\_not\_lits$ conforman tuplas donde el valor booleano es falso. Esta lista de tuplas puede pasarse como parámetro a una llamada al método $solve$ para forzar el valor de verdad de cada átomo auxiliar al valor que lo acompaña en la tupla.

Con esta lista de tuplas y los conjuntos construidos, se procede a evaluar el candidato. Para que el candidato se considere válido, este debe cumplir cuatro condiciones:

1. Los literales objetivos a los que hacen referencia los literales subjetivos, cuyo átomo auxiliar pertenezca al conjunto k_lits, deben pertenecer a las *cautious consequences* del programa.

2. Los literales objetivos a los que hacen referencia los literales subjetivos, cuyo átomo auxiliar pertenezca al conjunto not_k_lits, no pueden pertenecer a las *cautious consequences* del programa.

3. Los literales objetivos a los que hacen referencia los literales subjetivos, cuyo átomo auxiliar pertenezca al conjunto k_not_lits, no pueden pertenecer a las *brave consequences* del programa.

4. Los literales objetivos a los que hacen referencia los literales subjetivos, cuyo átomo auxiliar pertenezca al conjunto k_lits, deben pertenecer a las *brave consequences* del programa.

Para realizar estas comprobaciones, se cambia la configuración del objeto $candidates\_test$, seleccionando el modo de enumeración *cautious* y *brave* para el cálculo de las *cautious consequences* y de las *brave consequences* del programa respectivamente.

Las *cautious consequences* de un programa lógico son el conjunto intersección de todos los *answer sets* del programa. Una llamada al método *solve* de un objeto $Control$ con el modo de

enumeración *cautious*, devuelve un objeto iterable cuyos resultados son modelos que convergen a las *cautious consequences* del programa. De esta manera, cada nuevo modelo resultado es un subconjunto del anterior. Por ejemplo, dado el programa:

$$\{a\}. \tag{3.21}$$

$$\{b\}. \tag{3.22}$$

$$c\ . \tag{3.23}$$

una posible serie de modelos que convergen a las *cautious consequences* sería:

1. $\{a,\ b,\ c\}$
2. $\{b,\ c\}$
3. $\{c\}$

donde el conjunto (3) corresponde a las *cautious consequences* del programa.

Las *brave consequences* de un programa lógico son el conjunto unión de todos los *answer sets* del programa. Una llamada al método *solve* de un objeto $Control$ con el modo de enumeración *brave*, devuelve un objeto iterable cuyos resultados son modelos que convergen a las *brave consequences* del programa. De esta manera, cada nuevo modelo resultado es un superconjunto del anterior. Por ejemplo, dado el programa:

$$\{a\}. \tag{3.24}$$

$$\{b\}. \tag{3.25}$$

$$c\ . \tag{3.26}$$

una posible serie de modelos que convergen a las *brave consequences* sería:

1. $\{c\}$
2. $\{b,\ c\}$
3. $\{a,\ b,\ c\}$

donde el conjunto (3) corresponde a las *brave consequences* del programa.

Estos iterables, permiten realizar ciertas comprobaciones de manera iterativa. Por ejemplo, si un átomo no pertenece a un superconjunto de las *cautious consequences*, no pertenecerá a dichas *cautious consequences*. De igual forma, si un átomo pertenece a un subconjunto de las *brave consequences*, pertenecerá a dichas *brave consequences*. Así, si un candidato no cumple

una de las condiciones antes de obtener el modelo final, se descarta automáticamente sin necesidad de finalizar el cálculo en curso. El objetivo de esto, es reducir el tiempo de ejecución de `eclingo`.

Si la unión de los conjuntos k_lits y not_k_lits no es el conjunto vacío, se llama al método $solve$ de $candidates\_test$. Esta llamada lleva como parámetro la lista de tuplas $assumptions$. Además, la configuración del objeto $Control$ tiene activado el modo de enumeración *cautious*. A continuación, se comprueba la condición 1 de manera iterativa, pues si alguno de los átomos correspondientes no pertenece a un superconjunto de las *cautious consequences*, se puede descartar el candidato antes de finalizar el cómputo. La condición 2 solo se comprueba sobre el modelo final. El código que implementa estas comprobaciones, se puede ver en la figura 3.4.

```
test = True
if k_lits | not_k_lits:
    candidates_test.configuration.solve.enum_mode = 'cautious'
    with candidates_test.solve(yield_=True,
                               assumptions=assumptions) \
                                as candidates_test_handle:
        for cautious_model in candidates_test_handle:
            for epistemic in k_lits:
                atom = epistemic_atoms.get(epistemic)
                if atom not in cautious_model.symbols(atoms=True):
                    test = False
                    break
        if test:
            for epistemic in not_k_lits:
                atom = epistemic_atoms.get(epistemic)
                if atom in cautious_model.symbols(atoms=True):
                    test = False
                    break
```

Figura 3.4: Fragmento de $solver.py$ (1)

Acto seguido, si el candidato ha superado las pruebas y la unión de los conjuntos not_k_lits y $not\_k\_not\_lits$ no es el conjunto vacío, se llama al método $solve$ de $candidates\_test$. Esta llamada lleva como parámetro la lista de tuplas $assumptions$. Además, la configuración del objeto $Control$ tiene activado el modo de enumeración *brave*. La condición 3 se comprueba de manera iterativa, pues si alguno de los átomos correspondientes pertenece a un subconjunto de las *brave consequences* se puede descartar antes de finalizar el cómputo. La condición 4 solo se comprueba sobre el modelo final. La implementación de estas dos comprobaciones se puede ver en la figura 3.5:

Si el candidato supera todas las comprobaciones, pasa a la fase de selección de resultados.

3.2.3 Selección de resultados

En esta fase, se seleccionan los datos del candidato a mostrar en la salida de la herramienta. Estos datos vienen determinados por la lista $show\_signatures$. Los elementos de la lista $show\_signatures$ son triplas que siguen la siguiente estructura:

$$(name,\ arity,\ positive)$$

donde $name$ indica el nombre del átomo objetivo al que hace referencia el literal subjetivo que se pretende mostrar, $arity$ es su aridad y positive es valor booleano que indica el signo del átomo, cierto si es positivo y falso si tiene negación fuerte.

Si esta lista es vacía, significa que el programa no tenía ninguna directiva que especificase qué literales subjetivos se deberían mostrar, por lo que la acción por defecto consiste en devolver el candidato. En caso contrario, se realiza una búsqueda de los átomos objetivos en el programa a través del método $by\_signature$ del atributo $symbolic\_atoms$ de $candidates\_gen$. De esta manera, pasando como parámetro una tripla como las que contiene la lista $show\_signatures$, obtendremos los átomos *ground* del programa que cumplan esa estructura. A este conjunto de átomos lo llamaremos $show\_atoms$.

Con el conjunto $show\_atoms$ construido, se calculan las *cautious consequences* del programa pasando la lista de tuplas `assumptions` de nuevo como parámetro y se descartan aquellos átomos del conjunto que no pertenezcan a este modelo. Los átomos que superen el filtrado, conforman el modelo que se devuelve como salida. Para devolver estos modelos, se hace uso de la instrucción $yield$ de `Python`, que permite devolver una serie de resultados de uno en uno, en lugar de tener que almacenarlos en una estructura de datos y devolver un único valor. De esta manera, la CLI de `eclingo` puede ir mostrando los resultados a medida que se calculan.

Cada vez que se devuelve un modelo, se incrementa en uno el contador de modelos calculados. Si no quedan más candidatos por calcular o se ha calculado el número de modelos especificado por parámetro, la función finaliza.

El código correspondiente a esta fase del procedimiento, se puede ver en la figura 3.6. En la línea 16 se llama a la función privada $\_show\_result$, que recibe como parámetro una lista de átomos objetivos. El resultado de esta llamada, es una lista con los literales subjetivos con la estructura:

$$\boldsymbol{K}atom$$

donde $atom$ es un átomo objetivo de la lista que se pasa como parámetro.

```
if test and (k_not_lits | not_k_not_lits):
    candidates_test.configuration.solve.enum_mode = 'brave'
    with candidates_test.solve(yield_=True,
                               assumptions=assumptions) \
                                as candidates_test_handle:
        for brave_model in candidates_test_handle:
            for epistemic in k_not_lits:
                atom = epistemic_atoms.get(epistemic)
                if atom in brave_model.symbols(atoms=True):
                    test = False
                    break
        if test:
            for epistemic in not_k_not_lits:
                atom = epistemic_atoms.get(epistemic)
                if atom not in brave_model.symbols(atoms=True):
                    test = False
                    break
```

Figura 3.5: Fragmento de *solver.py* (2)

```
model_count += 1
if show_signatures:
    show_atoms = []
    for (name, arity, positive) in show_signatures:
        show_atoms += [atom.symbol for atom in
                       candidates_gen.symbolic_atoms.by_signature(
                         name, arity, positive)]

    candidates_test.configuration.solve.enum_mode = 'cautious'
    with candidates_test.solve(yield_=True,
                               assumptions=assumptions) \
                                as candidates_test_handle:
        *_, cautious_model = candidates_test_handle
        k_atoms = [atom for atom in show_atoms
                   if atom in cautious_model.symbols(atoms=True)]

    yield _show_result(k_atoms)

else:
    yield model.symbols(shown=True)

if model_count == models:
    break
```

Figura 3.6: Fragmento de *solver.py* (3)

Capítulo 4

Evaluación

En este capítulo se exploran las distintas alternativas a `eclingo` y se presentan los experimentos realizados para la evaluación de la herramienta desarrollada.

4.1 Estado del arte

En el momento de redacción de este documento, se han encontrado dos sistemas de resolución de programas lógicos epistémicos disponibles bajo repositorios públicos: `Wviews` [24] y EP-ASP [12].

4.1.1 `Wviews`

Es una herramienta de cálculo de *world views* bajo la semántica G94 [25]. Está escrita en el lenguaje de programación `Python` en su versión 2.x y emplea la herramienta de cálculo ASP DLV [26] como *backend*.

No se ha realizado ninguna comparativa entre `eclingo` y esta herramienta debido a las diferencias entre semánticas y entre *backends*, lo que daría lugar a una comparativa poco realista.

4.1.2 `EP-ASP`

Es un sistema de cómputo de modelos en programas lógico epistémicos que emplea `clingo` (en su versión 4.5.3) como *backend*. Está escrita en el lenguaje de programación `Python` en su versión 2.x. Permite calcular estos modelos bajo las semánticas K14 y SE16 [11]. Además de la selección de semánticas, ofrece opciones para realizar un preprocesado del programa y para emplear o no una heurística en los problemas de planificación.

La sintaxis de entrada a EP-ASP es sustancialmente diferente a la empleada por eclingo. De hecho, los programas de entrada se obtienen a partir de una traducción de los programas en formato .elps con el uso de la herramienta ELPS [27]. Además, para los problemas de *conformant planning*, los programas de entrada deben de seguir una estructura concreta especificada en una guía que acompaña al *solver*. La dificultad que presenta manipular esta representación, genera complicaciones para el uso de EP-ASP.

En la sección 4.2, se muestra una comparativa entre eclingo y EP-ASP. Para esta comparativa se han empleado como entrada para EP-ASP los programas ya traducidos por ELPS, puesto que no se contaba con los ficheros fuente en el formato anterior a la traducción.

4.2 Estudio comparativo

Se ha realizado un estudio comparativo entre EP-ASP y eclingo, con el objetivo de medir la eficiencia de la herramienta desarrollada. Para ello, se han medido los tiempos de ambas herramientas resolviendo dos problemas bien conocidos en la literatura de especificaciones epistémicas: el *Scholarship Elegibility Problem* y el *Yale Shooting Problem*.

Para que dicha comparativa sea lo más realista posible, lo ideal sería medir los tiempos de las herramientas sobre la misma máquina. No obstante, dado que ambas herramientas comparten las dependencias, pero en versiones diferentes, se hace realmente difícil que estas herramientas coexistan en la misma máquina sin producir conflictos. En concreto, EP-ASP emplea la versión 4.5.3 de clingo y la versión 2.x de Python. En cuanto a eclingo, la versión de clingo empleada es la 5.3.0 y la versión de Python es la 3.x.

Para superar este problema, se ha recurrido al uso de dos servidores Amazon Web Services (AWS) [1], con el objetivo de que las herramientas se ejecuten en máquinas con las mismas especificaciones, y así la comparativa sea lo más justa posible.

Las especificaciones técnicas de los servidores son:

- **OS**: Ubuntu Server 18.04 LTS (HVM) 64-Bit(x86)
- **CPU**: Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz
- **Memory**: 1 GiB

Los escenarios de entrada se han tomado del repositorio de EP-ASP, siendo los escenarios empleados en eclingo una adaptación de estos.

[1] https://aws.amazon.com/

4.2.1 Scholarship Eligibility Problem

Se han ejecutado 25 instancias del problema con un *timeout* de 300s (5min.) en cada herramienta. En la tabla 4.1, se pueden ver en detalles los resultados de la evaluación. Se han medido los tiempo de ejecución de ambas herramientas en sus diferentes opciones para calcular un modelo de cada instancia. Las casillas de la tabla que contienen el símbolo -, corresponden a ejecuciones que no han logrado terminar antes del *timeout*. Para cada instancia, se señala en negrita el mejor tiempo medido.

En el caso de `EP-ASP`, las opción pre indica si se han de computar o no las *brave* y *cautious consequences* del programa como preprocesado. Para cada una de estas opciones, se han calculado los tiempos con cada una de las semánticas soportadas por la aplicación.

Para `eclingo`, se han calculado los tiempos para cada semántica soportada en ambas versiones de la herramienta, la versión base (base) y la versión con mejora (op1).

A partir de los datos de la tabla 4.1, se ha construido la gráfica 4.4. Este diagrama de líneas, consiste en una representación semilogarítmica de los tiempos de ejecución de las herramientas. En él, el eje de ordenadas tiene escala logarítmica en base 10 y el eje de abscisas tiene una escala categórica formada por las diferentes instancias del problema. Se ha elegido la representación semilogarítmica debido a que la mayor parte de los datos se concentran en valores muy bajos (entre 0 y 1), pero existen valores muy altos que afectarían a una escala lineal. En dicha gráfica, se han representado únicamente los valores obtenidos, de manera que las casillas que corresponden a ejecuciones que no superan el *timeout* no han podido ser dibujadas.

Como se puede observar en el diagrama, las líneas de la herramienta `eclingo` en la versión con mejora son las que ofrecen mejores resultados en la gran mayoría de casos. En concreto, la línea que corresponde a la semántica G91 ha obtenido el tiempo más bajo en 12 de 25 instancias del problema, seguida muy de cerca de la semántica K14 de la misma herramienta y versión.

El formato de los programas de entrada a `EP-ASP` se puede observar en la figura 4.1, que corresponde al primer escenario del problema.

En cuanto a `eclingo`, la codificación del problema es la que se ve en la figura 4.2. Esta codificación va acompañada del código de cada instancia individual, como por ejemplo el de la figura 4.3.

```
% nat,s1

eligible(X_G):-highGPA(X_G),s1(X_G).
eligible(X_G):-minority(X_G),fairGPA(X_G),s1(X_G).
-eligible(X_G):--fairGPA(X_G),-highGPA(X_G),s1(X_G).
interview(X_G):--k_eligible(X_G),-k_0eligible(X_G),s1(X_G).
highGPA(mike)|fairGPA(mike).
s1(mike).
k1_eligible(X_G):-not k0_eligible(X_G), s1(X_G).
k0_eligible(X_G):-not k1_eligible(X_G), s1(X_G).
k0_eligible(X_G):-k1_0eligible(X_G).
k0_eligible(X_G):-m1_0eligible(X_G).
k0_eligible(X_G):-m0_eligible(X_G).
-k_eligible(X_G):-k0_eligible(X_G).
-k_eligible(X_G):-k1_eligible(X_G), not eligible(X_G).
k1_0eligible(X_G):-not k0_0eligible(X_G), s1(X_G).
k0_0eligible(X_G):-not k1_0eligible(X_G), s1(X_G).
k0_0eligible(X_G):-k1_eligible(X_G).
k0_0eligible(X_G):-m1_eligible(X_G).
k0_0eligible(X_G):-m0_0eligible(X_G).
-k_0eligible(X_G):-k0_0eligible(X_G).
-k_0eligible(X_G):-k1_0eligible(X_G), not -eligible(X_G).
```

Figura 4.1: *eligible*1.*elps.elp*

```
eligible(X) :- highGPA(X), student(X).
eligible(X) :- minority(X), fairGPA(X), student(X).
-eligible(X) :- -fairGPA(X), -highGPA(X), student(X).
interview(X) :- not &k{ eligible(X) }, not &k{ -eligible(X) },
    student(X).
```

Figura 4.2: *eligible.lp*

```
highGPA(mike), fairGPA(mike).
student(mike).
```

Figura 4.3: *eligible*01.*lp*

	EP-ASP				eclingo			
	pre=0		pre=1		base		op1	
	K14	SE16	K14	SE16	G91	K14	G91	K14
eligible01.lp	0.0133	0.0080	0.0112	0.0085	0.0033	0.0033	0.0033	**0.0032**
eligible02.lp	0.0133	0.0681	0.0076	0.0104	**0.0033**	**0.0033**	**0.0033**	0.0034
eligible03.lp	0.0189	0.6540	0.0084	0.0114	**0.0035**	0.0037	0.0036	**0.0035**
eligible04.lp	1.0669	6.7914	0.0092	0.0116	0.0045	**0.0045**	0.0047	0.0046
eligible05.lp	4.9430	46.4682	0.0144	0.0165	**0.0043**	**0.0043**	0.0045	0.0046
eligible06.lp	5.5972	38.0919	0.1943	0.0235	0.0067	0.0065	**0.0054**	0.0055
eligible07.lp	16.0136	-	0.0204	0.0269	0.0100	0.0101	0.0088	**0.0086**
eligible08.lp	45.8858	-	0.0215	0.0285	0.0150	0.0146	**0.0133**	0.0135
eligible09.lp	-	-	0.0261	0.0386	**0.0154**	0.0160	0.0195	0.0188
eligible10.lp	-	-	2.5225	9.6790	0.0876	0.0889	**0.0387**	0.0394
eligible11.lp	-	-	14.0898	10.5925	0.1554	0.1510	0.0850	**0.0847**
eligible12.lp	-	-	51.9861	12.3694	0.4441	0.4501	0.1575	**0.1557**
eligible13.lp	-	-	1.4259	13.2225	0.8498	0.8710	**0.3810**	0.4031
eligible14.lp	-	-	15.2031	14.2496	0.8823	0.8909	**0.3700**	0.3756
eligible15.lp	-	-	**0.0691**	16.4215	0.8996	0.9236	0.3777	0.3905
eligible16.lp	-	-	**0.0664**	28.6030	0.8106	0.8120	0.3895	0.4299
eligible17.lp	-	-	-	52.7815	2.5711	2.6801	**0.4713**	0.5187
eligible18.lp	-	-	-	23.6137	3.8668	3.9600	**0.4822**	0.4930
eligible19.lp	-	-	-	41.6403	6.2125	6.2680	**1.1669**	1.1927
eligible20.lp	-	-	-	46.8596	6.2640	6.2740	**1.3779**	1.3887
eligible21.lp	-	-	-	34.7780	10.5929	10.5703	**2.5157**	2.6422
eligible22.lp	-	-	-	40.5958	23.4731	23.5783	7.2683	**7.2127**
eligible23.lp	-	-	-	36.8165	75.2119	75.2193	**12.4937**	12.8398
eligible24.lp	-	-	-	41.9281	77.8251	76.9534	12.9330	**11.2606**
eligible25.lp	-	-	-	43.1458	38.5276	38.7441	31.6274	**31.4977**

- La herramienta no fue capaz de terminar la ejecución dentro del *timeout* (300s).

Tabla 4.1: Tabla de tiempos para ejecuciones del *Scholarship Elegibility Problem*.

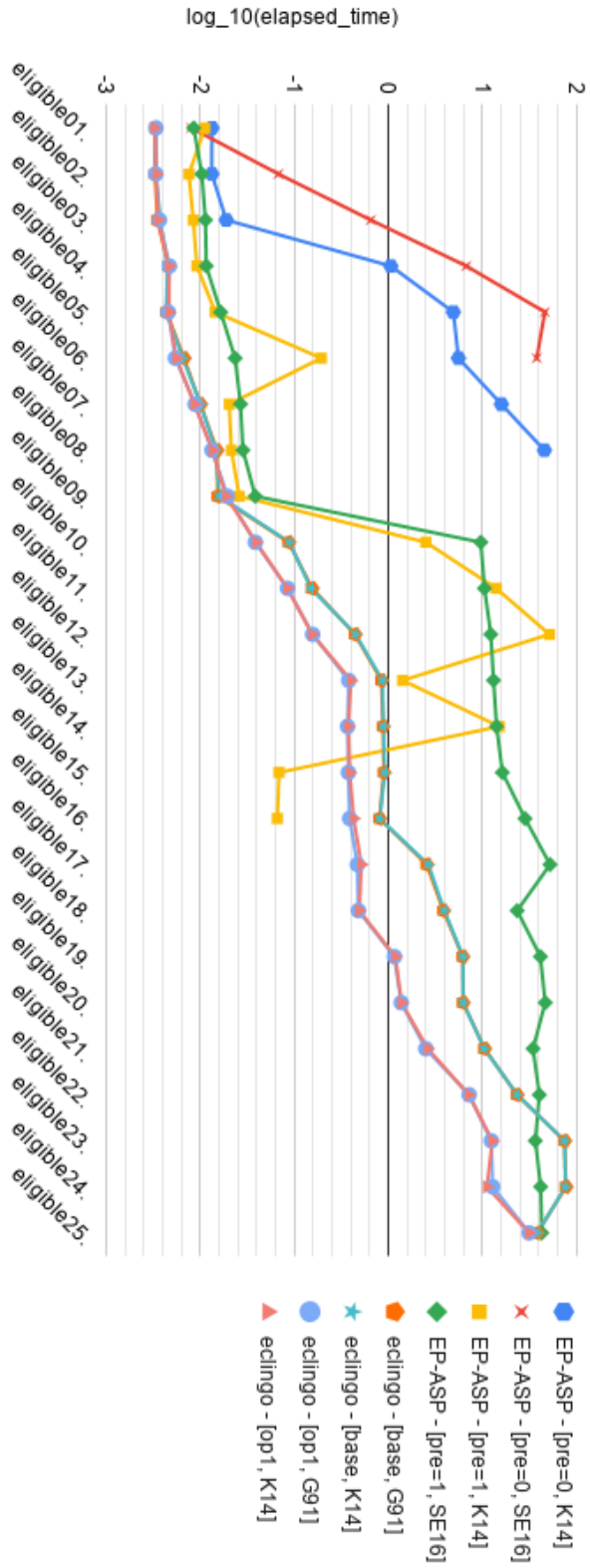

Figura 4.4: Representación semilogarítmica de los datos de la tabla 4.1.

4.2.2 Yale Shooting Problem

En este caso, se han ejecutado 12 instancias del problema con un *timeout* de 300s (5min.) en cada herramienta. En la tabla 4.2, se pueden ver en detalle los resultados de la evaluación. Se han medido los tiempo de ejecución de ambas herramientas en sus diferentes opciones para calcular un modelo de cada instancia. Las casillas de la tabla que contienen el símbolo -, corresponden a ejecuciones que no han logrado terminar antes del *timeout*, las que contienen el símbolo * , corresponden a instancias cuyo código no estaba disponible, y las que contienen el símbolo □, corresponden a instancias que no tienen solución. Para cada instancia, se señala en negrita el mejor tiempo medido.

En el caso de EP-ASP, las opción pre indica si se han de computar o no las *brave* y *cautious consequences* del programa como preprocesado y la opción $heuristic$ especifica si se emplea o no el modo heurístico de la herramienta. Para cada una de estas opciones, se han calculado los tiempos con cada una de las semánticas soportadas por la aplicación.

Para `eclingo`, se han calculado los tiempos para cada semántica soportada en ambas versiones de la herramienta, la versión base (base) y la versión con mejora (op1).

A partir de los datos de la tabla 4.2, se ha construido la gráfica 4.8. Al igual que en el problema anterior, este diagrama de líneas consiste en una representación semilogarítmica de los tiempos de ejecución de las herramientas. En él, el eje de ordenadas tiene escala logarítmica en base 10 y el eje de abscisas tiene una escala categórica formada por las diferentes instancias del problema. Se ha elegido la representación semilogarítmica debido a que la mayor parte de los datos se concentran en valores muy bajos (entre 0 y 1), pero existen valores muy altos que afectarían a una escala lineal. En dicha gráfica, se han representado únicamente los valores obtenidos, de manera que las casillas que corresponden a ejecuciones que no superan el *timeout* no han podido ser dibujadas. De la misma forma, se han omitido las instancias $yale07.lp$ y $yale13.lp$ por no disponer del código fuente y no tener solución, respectivamente. Además, para facilitar la legibilidad, se han omitido las líneas correspondientes a la versión no heurística de EP-ASP, puesto que eran las que ofrecían un peor resultado.

Como se puede observar en el diagrama, la línea que corresponde a la semántica G91 de la herramienta `eclingo` en la versión con mejora ha obtenido el tiempo más bajo en 6 de 11 instancias del problema, seguida muy de cerca por la línea de la semántica K14 de la misma herramienta y versión. Estas dos líneas son las únicas que tienen datos para todos los escenarios.

El escenario base en formato `eclingo` es el mostrado en el capítulo 3 (figura 3.2). Esta codificación va acompañada del código de cada instancia individual, como por ejemplo el de

la figura 4.5.

En cuanto al formato de entrada de EP-ASP siguen la estructura del programa conformado por el código de las figuras 4.6 y 4.7, que corresponde con la primera instancia del problema.

```
%%%%%%%  fluents   %%%%%%%%%
inertial(loaded).
inertial(alive).

%%%%%%%  actions   %%%%%%%%
action(load).
action(pull_trigger).

%%%%%%  executable  %%%%%%%
executable(pull_trigger,T):-  stepless(T).
executable(load,T) :- -holds(loaded,T),stepless(T).

%%%%%  effects   %%%%%%%
-holds(alive,T+1):-occurs(pull_trigger,T),holds(loaded,T),stepless(T).
-holds(loaded,T+1):-occurs(pull_trigger,T),stepless(T).
holds(loaded,T+1):-occurs(load,T),stepless(T).

%%%% initial state %%%%%
holds(alive,0).
holds(loaded,0).
```

Figura 4.5: $yale01.lp$

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%initial states%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#program initial.
#external _heuristic.
holds(alive,0).
holds(loaded,0).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%problem description%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#program problem.

step(0..len):- _heuristic.
step(0..length) :- not _heuristic.


max(len) :- _heuristic.
1{max(len); max(length)}1.
max(length) :- not _heuristic.

stepless(0..L-1) :- max(L).


%%%%%%%  fluents   %%%%%%%%%
fluent(loaded).
fluent(alive).
inertial(loaded).
inertial(alive).

%%%%%%%  actions   %%%%%%%%
action(load).
action(pull_trigger).


%%%%%%  executable  %%%%%%%
executable(pull_trigger,T):-  stepless(T).
executable(load,T) :- -holds(loaded,T),stepless(T).

%%%%%  effects   %%%%%%%

-holds(alive,T+1):-occurs(pull_trigger,T),holds(loaded,T),
    stepless(T).
-holds(loaded,T+1):-occurs(pull_trigger,T),stepless(T).
holds(loaded,T+1):-occurs(load,T),stepless(T).
```

Figura 4.6: $yale1.txt$ (1)

```
%%%%% occurs  %%%%%%%
1{occurs(A,S) : action(A), executable(A,S)} 1 :- stepless(S),
    _heuristic.

%%%%% goal  %%%%%%%
goal :- -holds(alive,LEN), -holds(is_impossible,LEN), max(LEN).

%%%% epistemic %%%%%%%

fluent(is_impossible).
inertial(is_impossible).

holds(is_impossible,S):- step(S), action(A), occurs(A, S), not
    executable(A, S).

%-holds(is_impossible, 0).

-holds(F,0):-not  holds(F,0),fluent(F).

holds(F,S+1):-fluent(F),inertial(F), stepless(S),holds(F,S),not
    -holds(F,S+1).

-holds(F,S+1):-fluent(F),inertial(F), stepless(S),-holds(F,S),not
    holds(F,S+1).

:-goal,-k_goal.
:-not  m_goal.

k1_goal:-not k0_goal.
k0_goal:-not k1_goal.

k0_goal:-m0_goal.
-k_goal:-k0_goal.
-k_goal:-k1_goal, not goal.
m1_goal:-not m0_goal.
m0_goal:-not m1_goal.
m1_goal:-k1_goal.

m_goal:-m1_goal.
m_goal:-m0_goal, goal.

m1_occurs(A,S):-not m0_occurs(A,S), stepless(S), action(A).
m0_occurs(A,S):-not m1_occurs(A,S), stepless(S), action(A).

m_occurs(A,S):-m1_occurs(A,S), stepless(S), action(A).
m_occurs(A,S):-m0_occurs(A,S), not not occurs(A,S), stepless(S),
    action(A).

occurs(A,S):-m_occurs(A,S), stepless(S),action(A).

%m1_occurs(A,S) :- occurs(A,S),  action(A), step(S).
```

Figura 4.7: $yale1.txt$ (2)

	EP-ASP								eclingo			
	heuristic=0				heuristic=1							
	pre=0		pre=1		pre=0		pre=1		base		op1	
	K14	SE16	K14	SE16	K14	SE16	K14	SE16	G91	K14	G91	K14
yale01.lp	0.0102	0.0132	0.0078	0.0110	0.0095	0.0133	0.0105	0.0145	0.0070	0.0073	**0.0069**	**0.0069**
yale02.lp	0.0075	0.0108	0.0090	0.0116	0.0106	0.0144	0.0121	0.0162	0.0074	0.0075	**0.0072**	0.0077
yale03.lp	0.0083	0.0119	0.0199	0.0329	0.0267	0.0430	0.0159	0.0204	0.0189	0.0150	**0.0082**	0.0085
yale04.lp	0.0092	0.0133	0.0110	0.0149	0.0135	0.0183	0.0152	0.0199	0.0298	0.0177	**0.0087**	0.0088
yale05.lp	0.0628	0.1177	0.0223	0.0319	0.0954	0.1764	0.0241	0.0302	58.4462	29.9206	**0.0131**	0.0134
yale06.lp	0.1114	0.2138	8.7664	5.9623	9.8776	16.5054	**0.0378**	0.0463	-	-	0.0515	0.0507
yale07.lp	*	*	*	*	*	*	*	*	*	*	*	*
yale08.lp	0.0173	0.0231	0.0211	0.0274	0.0242	0.0311	0.0282	0.0356	-	-	**0.0167**	0.0169
yale09.lp	**0.0227**	0.0258	0.0768	0.1225	0.0702	0.1079	222.6543	16.6545	-	-	0.4377	0.1899
yale10.lp	-	-	-	-	29.6763	42.9821	0.0769	0.1027	-	-	**0.0753**	0.2929
yale11.lp	-	-	-	-	0.2438	0.3962	-	-	-	-	0.0863	**0.0834**
yale12.lp	-	-	-	-	-	-	1.8496	**0.1820**	-	-	1.0662	1.0873
yale13.lp	□	□	□	□	□	□	□	□	□	□	□	□

* No se disponía del código fuente de la instancia.
- La herramienta no fue capaz de terminar la ejecución dentro del *timeout* (300s).
□ El problema no tiene solución.

Tabla 4.2: Tabla de tiempos para ejecuciones del *Yale Shooting Problem*.

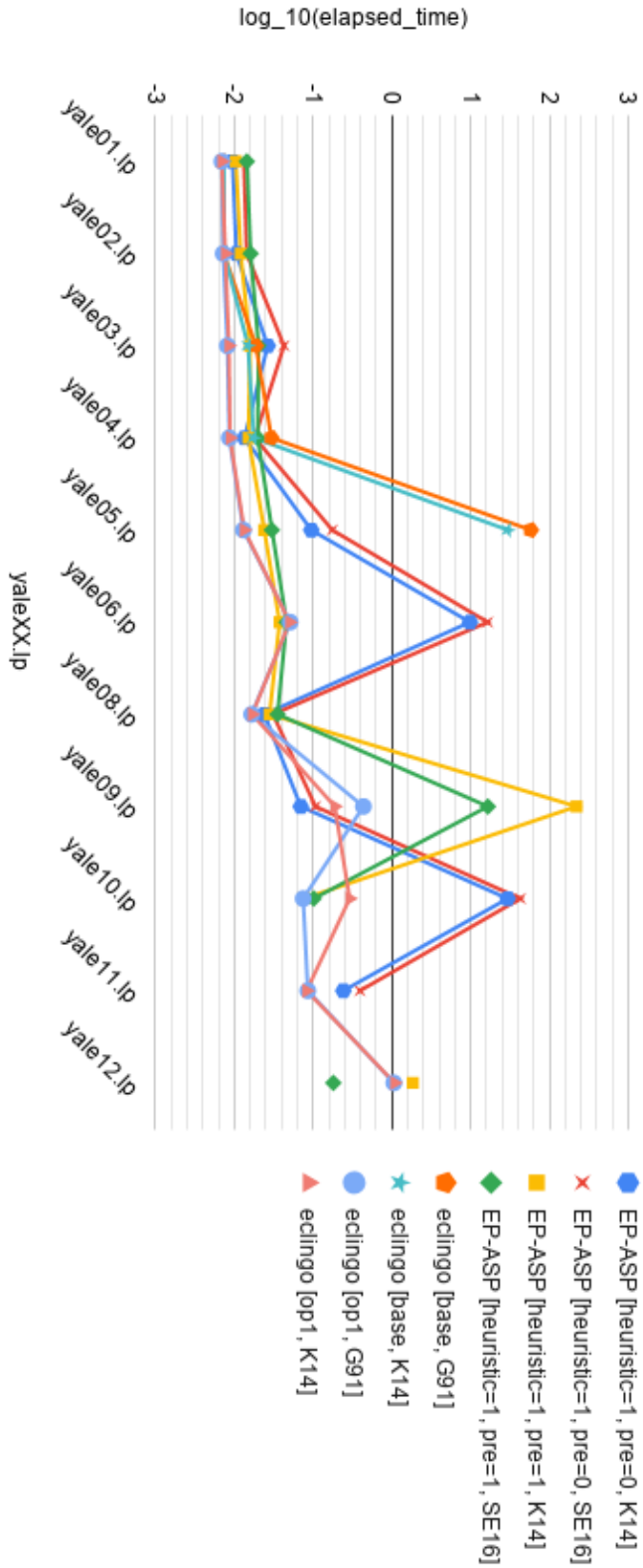

Figura 4.8: Representación semilogarítmica de los datos de la tabla 4.2.

Capítulo 5

Metodología, seguimiento y coste

En este capítulo, detallamos los aspectos relacionados con la gestión del proyecto desarrollado.

5.1 Metodología

La metodología empleada en el desarrollo del proyecto se puede encuadrar dentro de una aproximación ágil, incremental e iterativa. Esta aproximación tomó como referencia las metodologías ágiles *scrum* y *extreme programming*. Mas concretamente:

- ***Scrum:*** se han tomado los conceptos de agenda de producto y agenda de sprint, así como las reuniones de planificación de sprint y revisión de sprint.

- ***Extreme programming:*** se han dedicado fases del proyecto a la revisión y refactorización de código, con el objetivo de mantener un código limpio y sencillo. Además se han adoptado las técnicas de desarrollo dirigido por pruebas e integración continua.

Dado que no se conocía el alcance del proyecto, se decidió realizar el trabajo en una serie de iteraciones o *sprints* de corta duración (entre 1 y 2 semanas). Además, las iteraciones dedicadas al desarrollo tendrían un carácter incremental, por lo que cada una de ellas daría lugar a una versión funcional de la aplicación.

El proyecto partió de una reunión inicial en la que se definieron los objetivos principales del trabajo, dando así lugar a la primera versión de la agenda de producto. Al comienzo de cada nueva iteración se realizó una nueva reunión con el objetivo de revisar el trabajo realizado en la iteración pasada (revisión de *sprint*) y definir los objetivos de la nueva iteración (planificación de *sprint*), modificando la agenda de producto en los casos necesarios.

El desarrollo dirigido por pruebas ha sido clave en la realización del proyecto, ya que se partía de una batería de casos de prueba y sus soluciones para el desarrollo de la aplicación. Además, dado el carácter iterativo del desarrollo, se ha empleado el software de control de versiones `git` [1]. De esta manera, en caso de ser necesario, se podría volver a una versión anterior del software. Para combinar el desarrollo dirigido por pruebas con el software de control de versiones, se ha empleado la herramienta de integración continua `Travis CI` [2]. Así, con la subida de una nueva versión al repositorio, se ejecutan de manera automática las pruebas software, permitiendo llevar un control de la integridad de la aplicación entre incrementos.

Las pruebas software realizadas se clasifican en dos tipos:

- Pruebas de caja blanca o estructurales. Se ha empleado la herramienta de prueba `pylint` [3] para buscar defectos de diseño en la implementación de la aplicación.
- Pruebas de caja negra o funcionales. Se ha empleado la herramienta de prueba `pytest` [4] para validar la aplicación a través de una batería de casos de prueba.

No se han tomado en cuenta los aspectos de gestión de equipo de las metodologías de referencia, pues el trabajo fin de grado ha sido realizado únicamente por el alumno y su director.

5.2 Seguimiento

En las figuras 5.1 y 5.2 se puede visualizar el diagrama de Gantt del proyecto, donde se describen las tareas realizadas en cada una de las iteraciones. A continuación se muestran los detalles más relevantes de cada una de estas iteraciones.

Iteración 1 (29/04/2019 - 12/05/2019): El comienzo de la iteración está marcado por la reunión inicial. Se realiza un repaso de ASP.

Iteración 2 (13/05/2019 - 26/05/2019): Se proponen una serie de pequeños experimentos para familiarizarse con la libería *clingo* para `Python`.

Iteración 3 (27/05/2019 - 09/06/2019): Se realiza un estudio de la teoría de Especificaciones Epistémicas y se realizan nuevos experimentos.

Iteración 4 (10/06/2019 - 23/06/2019): Se continúa el estudio de Especificaciones Epistémicas y se continúan los experimentos.

[1] https://github.com/git/git
[2] https://travis-ci.org/
[3] https://github.com/PyCQA/pylint/
[4] https://github.com/pytest-dev/pytest

Iteración 5 (24/06/2019 - 07/07/2019): Se comienza con el desarrollo de la herramienta. Se recolecta una batería de casos de prueba sencillos y se implementa una aproximación a la semántica G91 que puede procesarlos.

Iteración 6 (08/07/2019 - 21/07/2019): Se refina la aproximación. Además, se implementa una mejora que reduce la cantidad de falsos candidatos generados.

Iteración 7 (22/07/2019 - 04/08/2019): Se incrementa la batería de casos de prueba con instancias del *Scolarship Elegibility Problem* y se implementa el soporte a la negación fuerte en los programas de entrada. También se actualiza el método de preprocesado.

Iteración 8 (05/08/2019 - 18/08/2019): Se implementa el soporte a la semántica K14 y se añaden más instancias del *Scholarship Elegibility Problem* a la batería de pruebas. Además, se comienza con la redacción de este documento.

Iteración 9 (19/08/2019 - 30/08/2019): Se incrementa la batería de casos de prueba con instancias del *Yale Shooting Problem* y del *Irrelevant Paths Problem*. También se da soporte a las directivas *show* en los programas de entrada y se implementa la funcionalidad de pasar constantes al programa como argumento.

Iteración 10 (31/08/2019 - 06/09/2019): Se evalúa el rendimiento de la herramienta y se finaliza la redacción de este documento.

5.3 Coste

Para este proyecto se han realizado 10 iteraciones. Por parte del alumno se estima un trabajo de 18 horas por semana. Dadas las fechas expuestas en la sección 5.2, se estiman unas 264,85 horas de trabajo total por parte del alumno. El rol del alumno es el de analista-programador, lo que supone un coste de 9€ por hora de trabajo. En cuanto al director, se le estima un trabajo de 1.5 horas por iteración, con un rol de jefe de proyecto que supone un coste de 20€ por hora trabajada. En total, el coste estimado de los recursos humanos del proyecto es de 2.967,65€. En la tabla 5.1 se pueden visualizar los detalles del cálculo.

Recurso	Horas de trabajo (h)	Coste/hora (€/h)	Coste total (€)
Jefe de proyecto	13,5	20	270
Analista-programador	300,85	9	2.707,65
Coste total de recursos (€)			2.967,65

Tabla 5.1: Coste estimado de los recursos humanos del proyecto.

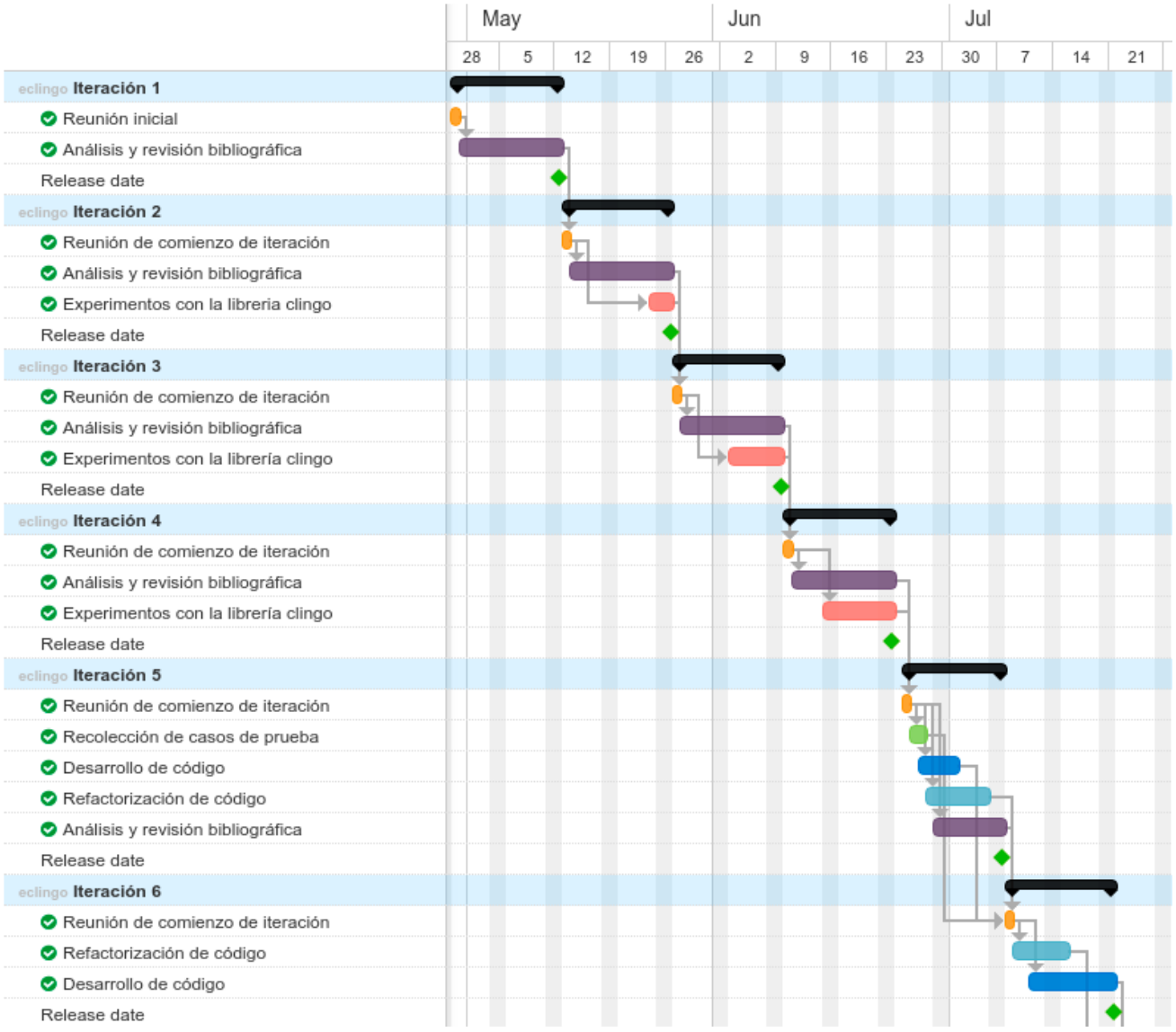

Figura 5.1: Diagrama de Gantt (1)

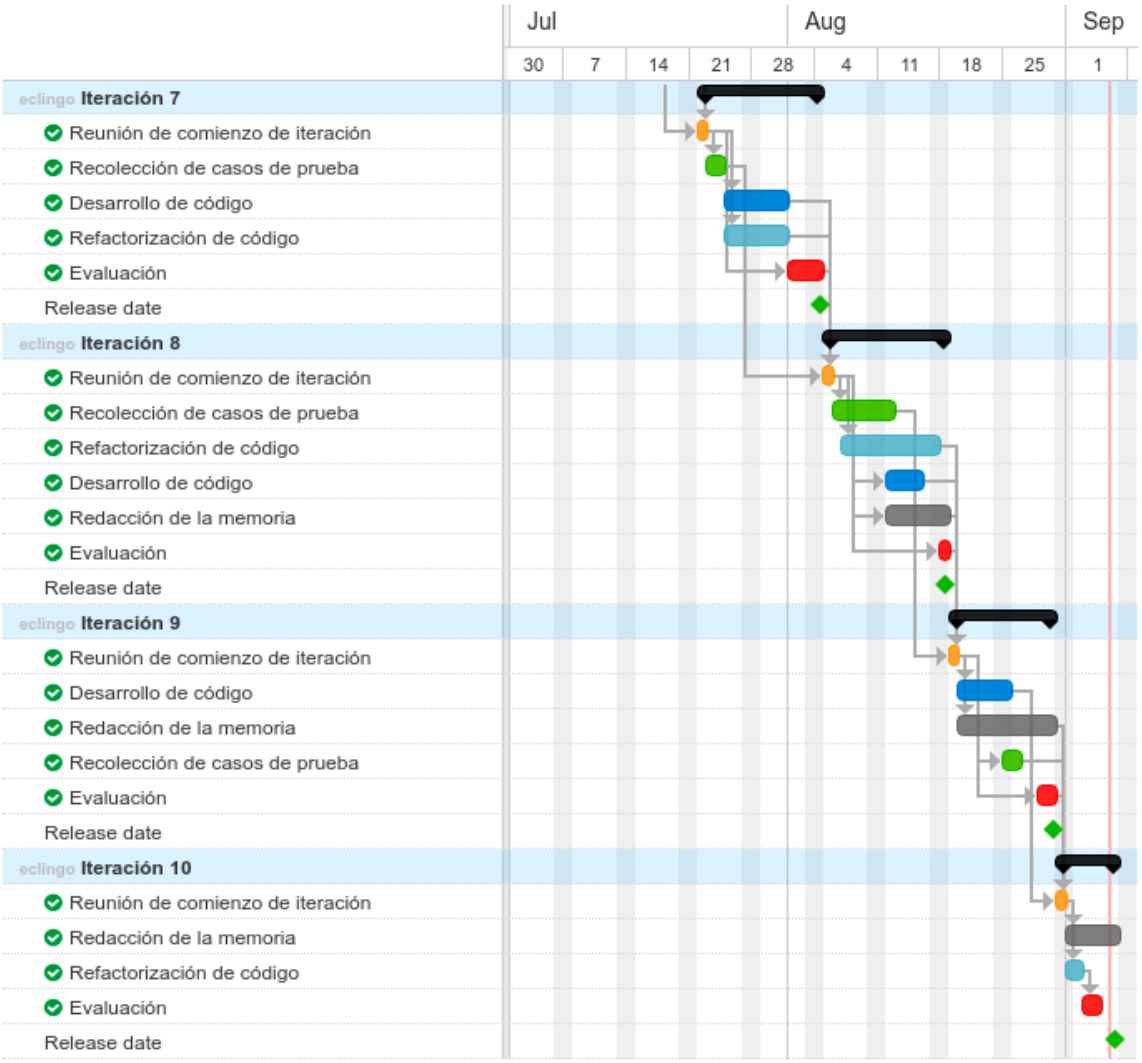

Figura 5.2: Diagrama de Gantt (2)

Dado que las herramientas software utilizadas son gratuitas y no se ha necesitado ningún *hardware* adicional para el desarrollo de la herramienta, el coste total de los recursos materiales involucrados en el desarrollo del proyecto es de 0€. En la tabla 5.2 se puede ver el desglose del cálculo.

Recurso	**Coste total (€)**
`clingo`	0
Intérprete de `Python`	0
`pylint`	0
`pytest`	0
`git`	0
`Travis CI`	0
Servidor AWS 1	0*
Servidor AWS 2	0*
Coste total de recursos (€)	0

*Coste del servicio con el plan AWS Educate.

Tabla 5.2: Coste estimado de los recursos materiales del proyecto.

Capítulo 6

Conclusiones

En este trabajo se ha desarrollado `eclingo`, una herramienta de cálculo de modelos de programas lógicos con expresiones epistémicas. La aplicación emplea la herramienta `clingo`, en su versión 5.3.0, como *backend* y se ha escrito sobre el lenguaje de programación *Python* en su versión 3.6.9.

El uso de `clingo`, ha facilitado la definición de una sintaxis para los programas de entrada a la `eclingo`, pues ha sido suficiente con una pequeña modificación de la sintaxis de `clingo`. Esta modificación permite al programador definir literales subjetivos en el cuerpo de las reglas.

Para el desarrollo de la herramienta, se ha reunido una batería de casos de prueba, que se compone de una serie de escenarios englobados en cuatro categorías diferentes. Además, se han diseñado e implementado los módulos que conforman la aplicación: el módulo $parser$, que analiza sintácticamente el programa de entrada, y el módulo $solver$, que computa sus modelos.

La interfaz entre el usuario y el sistema es una sencilla interfaz de linea de comandos que recibe los argumentos de entrada e imprime los resultados. Además del programa de entrada, `eclingo` permite una serie de argumentos opcionales que configuran el comportamiento del sistema. A través de estos argumentos es posible especificar el número máximo de modelos a computar, así como seleccionar entre las semánticas G91 y K14 para el cómputo de dichos modelos. De manera adicional, dichos argumentos también permiten la definición de constantes en el programa, una opción realmente útil para escenarios de planificación, o seleccionar la versión de mejora del sistema, lo que permite controlar la variación de la eficiencia del sistema entre las versiones de la implementación.

El balance global del trabajo es realmente positivo. El motivo principal es que se han superado los objetivos propuestos al comienzo del proyecto:

1. La herramienta calcula correctamente los modelos de un programa epistémico bajo las semánticas G91 y K14.

2. La sintaxis de entrada es realmente sencilla y muy similar a la de `clingo`, pues no es más que una pequeña modificación de esta.

3. Los resultados de la evaluación realizada han demostrado su eficiencia. Dichos resultados reflejan una mejora frente a EP-ASP, una de las herramientas con mayor relevancia en el estado del arte.

Además, no se han detectado defectos o errores inesperados en el comportamiento de la herramienta, lo que parece estar fuertemente motivado por el uso de prácticas como el desarrollo dirigido por pruebas o la integración continua.

Durante el desarrollo del proyecto solo se ha descartado una funcionalidad, que consistía en una segunda mejora de la herramienta. Esta mejora, reduciría teóricamente el tiempo de ejecución de la herramienta en escenarios concretos. Sin embargo, la mejora incrementaba sustancialmente el tiempo de cómputo en el resto de escenarios por lo que, ante la falta de escenarios de prueba específicos, se decidió no integrarla en el sistema.

6.1 Trabajo futuro

6.1.1 Nuevas mejoras

Tras los buenos resultados obtenidos, se pretende continuar con el desarrollo de `eclingo`, reuniendo nuevos escenarios de prueba e incorporando nuevas mejoras con el objetivo de aumentar aun más la eficiencia y opciones del sistema.

Durante la fase final del proyecto, se ha detectado una de estas posibles mejoras. Sin embargo, esta precisaba de un análisis de la estructura del programa lógico de entrada, lo que conllevaba un cambio muy sustancial en la implementación del sistema. Por ello se determinó que dicha modificación sobrepasaba el alcance del Trabajo Fin de Grado y, por lo tanto, se desarrollaría *a posteriori*.

6.1.2 Publicación

Los resultados de la evaluación también reflejan el impacto del trabajo desarrollado en el estado del arte, pues la evaluación realizada se hizo enfrentando a EP-ASP. Esta herramienta fue presentada en la conferencia más relevante de Inteligencia Artificial, la *International Joint Conference on Artificial Intelligence* (IJCAI).

En consecuencia, se pretende realizar una evaluación mas exhaustiva frente a esta y otras herramientas y contribuir a la comunidad publicando los resultados obtenidos.

6.1.3 Inclusión en el paquete de herramientas `Potassco`

El desarrollo de esta herramienta ha generado un creciente interés entre los desarrolladores de la colección de herramientas ASP más popular de la comunidad, el grupo de herramientas `Potassco`. Se pretende realizar una colaboración con dichos desarrolladores para la implementación en `eclingo` de las mejoras mencionadas anteriormente, con el objetivo de liberar la herramienta desarrollada como un proyecto de software libre dentro de dicha colección.

Bibliografía

[1] S. J. Russell and P. Norvig, *Artificial Intelligence - A Modern Approach, Third International Edition.* Pearson Education, 2010.

[2] J. McCarthy, "Programs with common sense," in *Proceedings of the Teddington Conference on the Mechanization of Thought Processes.* London: Her Majesty's Stationary Office, 1959, pp. 75–91.

[3] G. Brewka, T. Eiter, and M. Truszczynski, "Answer set programming at a glance." *Commun. ACM*, vol. 54, no. 12, pp. 92–103, 2011.

[4] E. Erdem, M. Gelfond, and N. Leone, "Applications of answer set programming." *AI Magazine*, vol. 37, no. 3, pp. 53–68, 2016.

[5] M. Gelfond, "Strong introspection." in *Proceedings of the Ninth National Conference on Artificial Intelligence - Volume 1*, ser. AAAI'91. AAAI Press, 1991, pp. 386–391.

[6] S. Hanks and D. V. McDermott, "Nonmonotonic logic and temporal projection." *Artif. Intell.*, vol. 33, no. 3, pp. 379–412, 1987.

[7] P. Cabalar, J. Fandinno, and L. F. del Cerro, "Splitting epistemic logic programs." in *Logic Programming and Nonmonotonic Reasoning - 15th International Conference, LPNMR 2019, Philadelphia, PA, USA, June 3-7, 2019, Proceedings*, 2019, pp. 120–133.

[8] P. Cabalar, J. Fandinno, and L. F. del Cerro, "Founded world views with autoepistemic equilibrium logic," in *Logic Programming and Nonmonotonic Reasoning - 15th International Conference, LPNMR 2019, Philadelphia, PA, USA, June 3-7, 2019, Proceedings*, 2019, pp. 134–147.

[9] P. T. Kahl and A. P. Leclerc, "Epistemic logic programs with world view constraints," in *Technical Communications of the 34th International Conference on Logic Programming, ICLP 2018, July 14-17, 2018, Oxford, United Kingdom*, 2018, pp. 1:1–1:17.

[10] A. P. Leclerc and P. T. Kahl, "A survey of advances in epistemic logic program solvers," *CoRR*, vol. abs/1809.07141, 2018.

[11] Y. Shen and T. Eiter, "Evaluating epistemic negation in answer set programming." *Artif. Intell.*, vol. 237, pp. 115–135, 2016.

[12] T. C. Son, T. Le, P. Kahl, and A. Leclerc, "On computing world views of epistemic logic programs." in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 2017, pp. 1269–1275.

[13] P. Kahl, R. Watson, E. Balai, M. Gelfond, and Y. Zhang, "The language of epistemic specifications (refined) including a prototype solver," *Journal of Logic and Computation*, p. exv065, 09 2015.

[14] M. Truszczynski, "Revisiting epistemic specifications." in *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, 2011, pp. 315–333.

[15] M. Gelfond, "New semantics for epistemic specifications," in *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, 2011, pp. 260–265.

[16] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele, "A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`." [En línea]. Disponible en: http://potassco.sourceforge.net.

[17] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming." in *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*, 1988, pp. 1070–1080.

[18] P. Cabalar, R. Kaminski, P. Morkisch, and T. Schaub, "telingo = ASP + time," in *Logic Programming and Nonmonotonic Reasoning - 15th International Conference, LPNMR 2019, Philadelphia, PA, USA, June 3-7, 2019, Proceedings*, 2019, pp. 256–269.

[19] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems - specification.* Springer, 1992.

[20] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry, "An A prolog decision support system for the space shuttle," in *Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st Intl. ASP'01 Workshop, Stanford, CA, USA, March 26-28, 2001*, 2001.

[21] P. Kahl, "Refining the semantics for epistemic logic programming." Ph.D. dissertation, Texas Tech University, 2014.

[22] P. Cabalar, J. Fandinno, and L. F. del Cerro, "Dynamic epistemic logic with asp updates: Application to conditional planning." in *Proc. of the 10th Intl. Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'19) Philadeplphia, PA, USA, June 3rd, 2019*, 2019.

[23] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Multi-shot ASP solving with clingo." *TPLP*, vol. 19, no. 1, pp. 27–82, 2019.

[24] M. Kelly, "A worldview solver for epistemic logic programs." Honour's Thesis, University of Western Sydney, 2007.

[25] M. Gelfond, "Logic programming and reasoning with incomplete information." *Ann. Math. Artif. Intell.*, vol. 12, no. 1-2, pp. 89–116, 1994.

[26] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, "The DLV system for knowledge representation and reasoning," *ACM Trans. Comput. Log.*, vol. 7, no. 3, pp. 499–562, 2006.

[27] E. Balai and P. Kahl, "Epistemic logic programs with sorts." in *Proc. 7th Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2014)*, 2014.