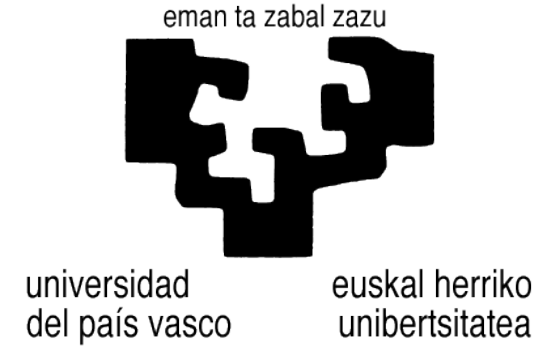

Facultad de Informática / Informatika Fakultatea

Proyecto Fin de Carrera:

Procedimiento de refutación para un conjunto de restricciones sobre documentos XML

Alumno: Javier Albors
Directora: Marisa Navarro

Donostia - San Sebastián, Julio de 2014

Abstract

We present a prototype that implements a set of logical rules to prove the satisfiability for a class of specifications on XML documents. Specifications are given by means of constraints built on Boolean XPath patterns. The main goal of this tool is to test if a given specification is satisfiable or not, showing the history of the execution. It can also be used to test if a given document is a model of a given specification and, as a subproduct, it allows to look for all the relations (monomorphisms) between two patterns or the result of doing some operations by combining patterns in different ways. The results of these operations are visually shown and therefore the tool makes these operations more understandable. The implementation of the algorithm has been written in Prolog but the prototype has a Java interface for an easy and friendly use.

Agradecimientos

Quiero agradecer a Marisa la paciencia y dedicación que ha aportado al proyecto, así como los consejos y sugerencias sin los cuales este proyecto no habría llegado a ser lo que es.

Gracias también a mi familia y a mis compañeros, por estar a mi lado y ayudarme todos estos años.

Por último, mencionar y agradecer al Gobierno Vasco por la Beca de Colaboración para el desarrollo de este proyecto, gracias a la cual he podido ampliar la dedicación al mismo.

Índice general

Abstract **I**

Agradecimientos **III**

Índice general **V**

Índice de figuras **IX**

Índice de tablas **XIII**

I Memoria **1**

1. Introducción **3**
1.1. Objetivos 3
1.2. Planificación 3
1.2.1. Alcance 4
1.2.2. Entregables 5
1.2.3. EDT 5
1.2.4. Gestión de Tiempos 6
1.2.5. Costes 8
1.2.6. Análisis de Calidad 8
1.2.7. Gestión de Comunicaciones 10
1.2.8. Gestión de Riesgos 10

2. Antecedentes **13**
2.1. Documento XML 13
2.2. Patrón 14
2.2.1. Monomorfismo 15
2.3. Constraints, Cláusulas y Especificaciones 16
2.4. Operaciones 17
2.4.1. $p_1 \otimes p_2$ 17
2.4.2. $p_1 \otimes_{c,m} q$ 18
2.5. Reglas 19

2.5.1. Regla R1 . . . 19
2.5.2. Regla R2 . . . 20
2.5.3. Regla R3 . . . 20
2.5.4. Subsunción . . . 21
2.5.5. Unfold1 . . . 22
2.5.6. Unfold2 . . . 22

3. Aplicación: Captura de Requisitos **25**
3.1. Requisitos . . . 25
3.2. Modelo de Casos de Uso . . . 25
3.2.1. Procedimiento de Refutación v1 . . . 26
3.2.2. Procedimiento de Refutación v2 . . . 27
3.2.3. Comprobar Especificación . . . 28
3.2.4. Monomorfismo . . . 28
3.2.5. Operar $(p_1 \otimes p_2)$ . . . 29
3.2.6. Operar con Común $(p_1 \otimes_{c,m} q)$ . . . 30
3.2.7. Consultar Cláusulas . . . 31
3.2.8. Consultar Constraints . . . 32

4. Aplicación: Diseño **33**
4.1. Arquitectura . . . 33
4.1.1. Puente Java-Prolog . . . 33
4.2. Modelo de Clases . . . 34
4.2.1. Capa de Aplicación . . . 34
4.2.2. Capa de Interfaz Gráfica . . . 39
4.2.3. Capa de Lógica . . . 53
4.2.4. Capa de Datos . . . 65

5. Aplicación: Implementación **69**
5.1. Patrones . . . 69
5.2. Constraints . . . 71
5.3. Cláusulas . . . 71
5.4. Monomorfismo . . . 72
5.4.1. Algoritmo . . . 73
5.4.2. Implementación . . . 74
5.4.3. Ejemplo . . . 75
5.5. Operar . . . 77
5.5.1. Algoritmo . . . 77
5.5.2. Implementación . . . 83
5.5.3. Ejemplo . . . 85
5.6. Operar Con Común . . . 91
5.6.1. Algoritmo . . . 91
5.6.2. Implementación . . . 93
5.6.3. Ejemplo . . . 93

5.7. Unfold 94
5.8. Subsunción 95
5.9. Reglas de Simplificación 97
5.10. Procedimiento de Refutación 98
5.10.1. Versión 1 98
5.10.2. Versión 2 99
5.11. Mejoras en la Implementación 100

II Anexos **101**

A. Manual de Configuración y Uso **103**
A.1. Requisitos 103
A.2. Configuración del Puente Java-Prolog 103
A.3. Manual de Uso 105
A.3.1. Pantalla Principal 106
A.3.2. Introducir Cláusulas 106
A.3.3. Resultado e Historial 107
A.3.4. Editar Patrones 109
A.3.5. Comprobar Especificación 111
A.3.6. Operación Monomorfismo 113
A.3.7. Operación $p_1 \otimes p_2$ 113
A.3.8. Operación $p_1 \otimes_{c,m} q$ 113
A.3.9. Menú Visualizar 115

B. Seguimiento y Control **117**
B.1. Retrasos 117
B.2. Tareas No Planificadas 117
B.3. Comparativa Planificación vs. Real 117
B.3.1. Gestión 118
B.3.2. Formación 118
B.3.3. Aplicación 118
B.3.4. Documentación 119
B.3.5. Resumen 119

Bibliografía **121**

Índice de figuras

1.1. Estructura de Descomposición de Trabajo 6
1.2. Diagrama de Gantt 9

2.1. Ejemplo de documento XML 14
2.2. Ejemplo de patrón 15
2.3. Monomorfismo de un patrón p en otro patrón q 16
2.4. Operación $p_1 \otimes p_2$ 18
2.5. Patrón caso general de la solución de la figura 2.4 18
2.6. Regla R1 20
2.7. Regla R2 20
2.8. Regla R3 21
2.9. Regla de Subsunción 21
2.10. Ejemplo de aplicación de $Unfold_1$ 22
2.11. Ejemplo de aplicación de $Unfold_2$ 23

3.1. Diagrama de secuencia de *Procedimiento de Refutación v1* . . 27
3.2. Diagrama de secuencia de *Procedimiento de Refutación v2* . . 27
3.3. Diagrama de secuencia de *Comprobar Especificación* 28
3.4. Diagrama de secuencia de *Monomorfismo* 29
3.5. Diagrama de secuencia de *Operar* 30
3.6. Diagrama de secuencia de *Operar con Común* 31
3.7. Diagrama de secuencia de *Consultar Cláusulas* 32
3.8. Diagrama de secuencia de *Consultar Constraints* 32

4.1. Arquitectura de la aplicación 34
4.2. Diagrama de la clase Aplicación 37
4.3. Diagrama de la clase AnalizadorPatrones 38
4.4. Diagrama de clases de la Capa de Aplicación 39
4.5. Panel *Editor* 40
4.6. Diagrama de la clase Editor 43
4.7. Diagrama de la clase CirculoNodo 45
4.8. Diagrama de la clase RelacionNodos 46
4.9. Panel *EditorCTParaTodo* 48
4.10. Diagrama de la clase EditorCTParaTodo 48

4.11. Ventana *PanelPrincipal* 49
4.12. Diagrama de la clase PanelPrincipal 52
4.13. Ventana *Historial* 53
4.14. Diagrama de clases de la Capa de Interfaz 54
4.15. Interacción de predicados de *Procedimiento de Refutación v1* 57
4.16. Interacción de predicados de *Procedimiento de Refutación v2* 58
4.17. Interacción de predicados de *Comprobar Especificación* 60
4.18. Interacción de predicados de *Monomorfismo* 62
4.19. Interacción de predicados de *Operar* 64
4.20. Interacción de predicados de *Operar Con Común* 66
4.21. Interacción de predicados de *Consultar Cláusulas* 66
4.22. Interacción de predicados de *Consultar Constraints* 67
4.23. Diagrama de clases de la Capa de Datos 68

5.1. Ejemplo de patrón 70
5.2. Ejemplo de constraint positivo en Prolog 72
5.3. Ejemplo de constraint negativo en Prolog 72
5.4. Ejemplo de constraint condicional en Prolog 72
5.5. Ejemplo de cláusula 72
5.6. Problema al no poder distinguir los nodos en el monomorfismo 74
5.7. Ejemplo monomorfismo: p y q 75
5.8. Ejemplo monomorfismo: p y q numerados 76
5.9. Ejemplo monomorfismo: raíces unidas 76
5.10. Ejemplo monomorfismo: monomorfismos posibles 76
5.11. Ejemplo monomorfismo: paso intermedio 77
5.12. Ejemplo monomorfismo: hijos directos de p unidos 77
5.13. Ejemplo monomorfismo: solución 78
5.14. Ejemplo monomorfismo: solución vista con la aplicación . . . 79
5.15. Ejemplo: nivel inicial 80
5.16. Ejemplo: unificar b y luego f 80
5.17. Ejemplo: unificar f y luego b 80
5.18. Fusión de nodos *hijo directo* 82
5.19. Fusión de nodos *hijo directo* y *descendiente* con mismo padre 82
5.20. Fusión de nodos / y // siendo éste ultimo hijo del ascendiente común mas cercano 83
5.21. Fusión de nodos / y // con diferente padre y ninguno hijo del ascendiente común más cercano 83
5.22. Fusión de nodos caso imposible 84
5.23. Fusión de nodos // y // diferente padre y ninguno hijo del ascendiente común más cercano 84
5.24. Operar: p_1 y p_2 84
5.25. Operar: $p_1 \otimes p_2$ 85
5.26. Operar: p_1, p_2 y $p_1 \otimes p_2$ 85
5.27. Patrones a operar: p y q 86

5.28. Ejemplo operar: Nivel 1 86
5.29. Ejemplo operar: Nivel 2 87
5.30. Ejemplo operar: patrones obtenidos de Nivel 2: b 88
5.31. Ejemplo operar: patrones obtenidos de Nivel 2: d 88
5.32. Ejemplo operar: patrones obtenidos de Nivel 2: f 89
5.33. Ejemplo operar: patrones obtenidos de Nivel 3: b y d 89
5.34. Ejemplo operar: patrones obtenidos de Nivel 3: b y f 90
5.35. Ejemplo operar: patrones obtenidos de Nivel 3: d y f 90
5.36. Ejemplo operar: patrones obtenidos de Nivel 3: f y c(3) 91
5.37. Ejemplo operar: solución 92
5.38. Constraints $\exists p_1$ y $\forall(c : p_2 \rightarrow q)$ 94
5.39. Nivel 1. Monomorfismo m con los nodos de q 94
5.40. Solución del ejemplo 95
5.41. Aplicación de la regla Unfold con $Starlength = 2$ 96

A.1. Creación de la variable de entorno SWI_HOME_DIR 104
A.2. Edición de la variable del entorno Path 105
A.3. Pantalla principal de la aplicación 106
A.4. Pantalla principal: introducir constraints 107
A.5. Pantalla principal: ejecutar el procedimiento de refutación . . 107
A.6. Resultado del procedimiento de refutación 108
A.7. Pantalla del historial 108
A.8. Editor de patones 109
A.9. Editor de constraints condicionales 111
A.10. Ejecutar el caso de uso Comprobar Especificación 112
A.11. Comprobar especificación con especificación vacía 112
A.12. Comprobar especificación con especificación actual 113
A.13. Pantalla de la operación monomorfismo 114
A.14. Pantalla de la operación $p_1 \otimes p_2$ 114
A.15. Pantalla de la operación $p_1 \otimes_{c,m} q$ 115
A.16. Menú Visualizar 115

Índice de tablas

1.1. Tiempo estimado para Gestión 7
1.2. Tiempo estimado para Formación 7
1.3. Tiempo estimado para Aplicación 7
1.4. Tiempo estimado para Documentación 7
1.5. Resumen de hitos 8

B.1. Comparativa de la actividad Gestión 118
B.2. Comparativa de la actividad Formación 118
B.3. Tiempo estimado para Aplicación 119
B.4. Tiempo estimado para Documentación 119

Parte I

Memoria

Capítulo 1

Introducción

En este capítulo se explicarán los objetivos del proyecto, así como todos los aspectos relacionados con la planificación inicial del mismo.

1.1. Objetivos

El proyecto tiene como objetivo principal la construcción de una herramienta que implemente un procedimiento de refutación para poder decidir la satisfacibilidad de una especificación sobre documentos XML. Para ello, es necesario un estudio y diseño previos. A continuación se muestra una lista con todos los objetivos establecidos para el proyecto:

- Estudio de una clase de especificaciones sobre documentos XML, definidas mediante conjuntos de restricciones ("constraints") construidos sobre patrones XPath, basándonos en los trabajos [2], [3] y [4].
- Estudio de las reglas lógicas necesarias para razonar sobre dichas especificaciones.
- Diseño de un procedimiento de refutación correcto, y a ser posible completo, que decida si una especificación dada es satisfacible o no lo es.
- Implementación del procedimiento de refutación.

1.2. Planificación

A continuación se hablará del alcance, los entregables, las tareas, la gestión de tiempos, los costes, la calidad, las comunicaciones y los riesgos del proyecto.

1.2.1. Alcance

En este apartado se indicará cuál es el alcance mínimo y la ampliación del alcance planteada.

1.2.1.1. Alcance Mínimo

El producto obtenido será una aplicación que permitirá introducir una especificación y comprobar si es satisfacible. Para ello, se dividirá la tarea en dos partes: procedimiento de refutación (que se implementará en Prolog) e interfaz gráfica (que se implementará en Java).

1.2.1.1.1. Procedimiento de Refutación

Se implementarán dos prototipos, el primero contendrá tan sólo las reglas R1, R2 y R3, además de la regla de subsunción. El segundo será una ampliación del primero, e incluirá las reglas Unfold. Las tareas a realizar en cada prototipo son, por tanto:

Prototipo1

- Decidir una representación para los patrones, constraints, cláusulas y demás elementos utilizados en el procedimiento.
- Implementar las operaciones monomorfismo, $p \otimes q$ (operar) y $p_1 \otimes_{c,m} q$ (operar con común).
- Implementar las reglas R1, R2 y R3.
- Implementar la regla subsunción.
- Diseñar e implementar un algoritmo de refutación que emplee las reglas anteriores para obtener el resultado.
- Opcionalmente:
    - Añadir un historial para poder consultar el orden de ejecución de las reglas.

Prototipo 2

- Implementar las reglas Unfold.
- Diseñar e implementar un nuevo algoritmo (sin modificar el del prototipo 1) que incluya también las reglas Unfold.
- Si en el prototipo 1 no se ha implementado el historial, se implementará para el prototipo 2.

1.2.1.1.2. Interfaz Gráfica

Las subtareas a realizar para esta tarea son:

- Analizar y configurar el puente Java-Prolog.
- Diseñar una distribución visual de las pantallas para facilitar el uso de la aplicación.
- Diseñar e implementar el editor de patrones, donde se podrán crear nodos, relaciones hijo/descendiente, eliminar nodos, seleccionar y mover los nodos, y copiar y pegar los nodos.
- Implementar el resto de las ventanas diseñadas: pantalla principal, pantalla del historial, etc.
- Incluir un sistema de guardado de ficheros para poder cargar rápidamente una especificación existente.
- Incluir herramientas para la utilización de las operaciones monomorfismo, operar y operar con común.

1.2.1.2. Ampliación del Alcance

Por el momento no se ha considerado la necesidad de ampliar el alcance. Se deja abierta, sin embargo, la posibilidad de ampliarla más adelante si se considera oportuno.

1.2.2. Entregables

El proyecto consta de los siguientes entregables:

1. Memoria del proyecto.
2. Aplicación ejecutable junto con todos los archivos necesarios para su ejecución.
3. Manual de configuración y uso de la aplicación.
4. Código fuente de la aplicación.

1.2.3. EDT

En la figura 1.1 se muestra la Estructura de Descomposición de Trabajo, donde se desglosan las tareas que van a componer el proyecto. El esquema se ha dividido en cuatro grupos de trabajo: Gestión, Formación, Aplicación y Documentación.

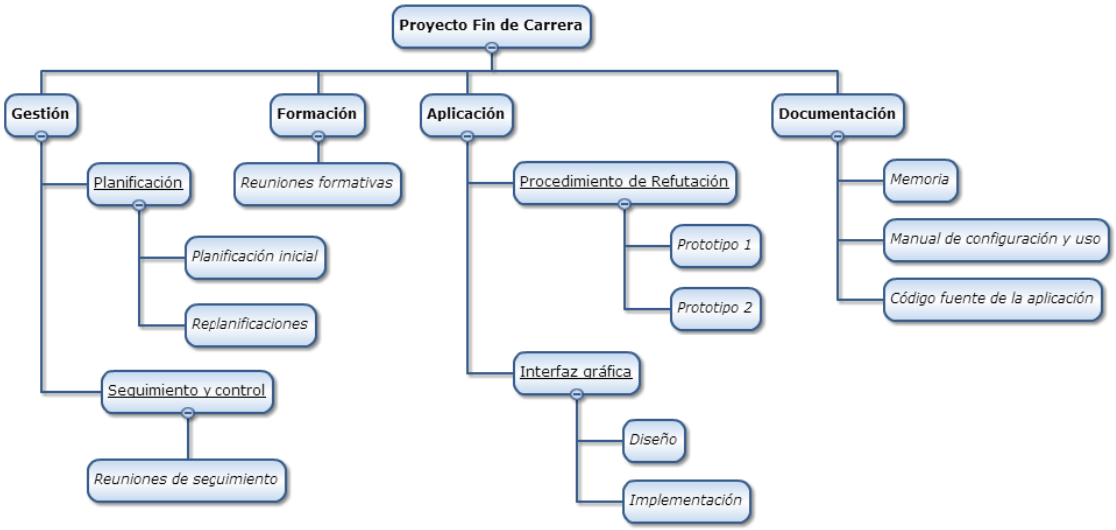

Figura 1.1: Estructura de Descomposición de Trabajo

1.2.4. Gestión de Tiempos

En esta sección se estima el tiempo que se empleará en la realización de las diferentes tareas del proyecto, se establece las fechas de los hitos y finalmente se incluye un cronograma (diagrama de Gantt) indicando las fechas previstas de realización de las tareas y de los hitos.

1.2.4.1. Tareas

El tiempo total que se ha planificado para el alcance mínimo es de 416 horas. A continuación, a modo de desglose, se mostrarán una serie de tablas donde se indicarán las horas que se emplearán y las fechas en las que se llevarán a cabo las diferentes tareas que aparecen en la Estructura de Descomposición de Trabajo (ver figura 1.1).

Gestión

En la tabla 1.1 se muestra la información de la tarea Gestión, donde se incluyen las tareas de Planificación y la de Seguimiento y control.

Formación

En la tabla 1.2 se muestra el tiempo dedicado a la tarea de Formación, constituido únicamente por la tarea de Reuniones formativas.

Aplicación

En la tabla 1.3 se muestra el tiempo planificado para la tarea Aplicación, compuesta por las tareas Procedimiento de refutación e Interfaz gráfica.

Gestión	Tiempo planificado
Planificación	
Planificación inicial	15h
Replanificaciones	5h
Seguimiento y control	
Reuniones de seguimiento	20h
Total	40h

Tabla 1.1: Tiempo estimado para Gestión

Formación	Tiempo planificado
Reuniones formativas	40h
Total	40h

Tabla 1.2: Tiempo estimado para Formación

Aplicación	Tiempo planificado
Procedimiento de refutación	
Prototipo 1	120h
Prototipo 2	60h
Interfaz gráfica	
Diseño	10h
Implementación	110h
Total	300h

Tabla 1.3: Tiempo estimado para Aplicación

Documentación

En la tabla 1.4 se indica el tiempo que se empleará para el desarrollo de la tarea Documentación, compuesta por las tareas Memoria, Manual de configuración y uso de la aplicación y Código fuente de la aplicación.

Documentación	Tiempo planificado
Memoria	30h
Manual de conf. y uso de la aplicación	5h
Código fuente de la aplicación	1h
Total	36h

Tabla 1.4: Tiempo estimado para Documentación

1.2.4.2. Hitos

A continuación se muestra en la tabla 1.5 un listado con los hitos y las fechas en las que están programados.

Hitos	Fecha planificada
Planificación	10/09/2013
Prototipo 1	29/11/2013
Prototipo 2	31/01/2014
Prototipo con interfaz	30/05/2014
Memoria	30/06/2014
Fin del Proyecto	01/07/2014

Tabla 1.5: Resumen de hitos

1.2.4.3. Diagrama de Gantt

En esta sección mostraremos un cronograma (figura 1.2) donde se indica la duración de cada tarea a lo largo del ciclo de vida del proyecto. Aunque el proyecto dura de septiembre a julio, cabe destacar que el mes de junio no será únicamente dedicado para la documentación, sino también como período de holgura para aquellas tareas que así lo precisen o en caso de que se decida hacer una ampliación de alcance.

1.2.5. Costes

Para la realización del proyecto se han estimado un total de 450 horas. Como ya hemos visto en el apartado 1.2.4.1, el tiempo total para la realización de las tareas es de 416. Por tanto, se posee un margen de 34 horas que se pueden emplear para alargar la duración de alguna tarea en caso de necesidad.

Por otro lado, se planea realizar las tareas utilizando herramientas de libre uso. De ese modo obtenemos un coste estimado de cero euros.

1.2.6. Análisis de Calidad

Se aspira a obtener un proyecto de calidad alta. A fin de poder lograr este objetivo, se han acordado las siguientes pautas:

1. Se realizarán **reuniones de seguimiento** cada quince días, permitiendo así redirigir el rumbo del proyecto en caso de haberse desviado del camino deseado.

2. Se fijan dos **hitos** principales: entrega de los prototipos 1 y 2 (ver tabla 1.5 para más información). Las fechas establecidas proporcionan la suficiente holgura como para poder admitir algún retraso en caso de que ocurriera algún incidente.

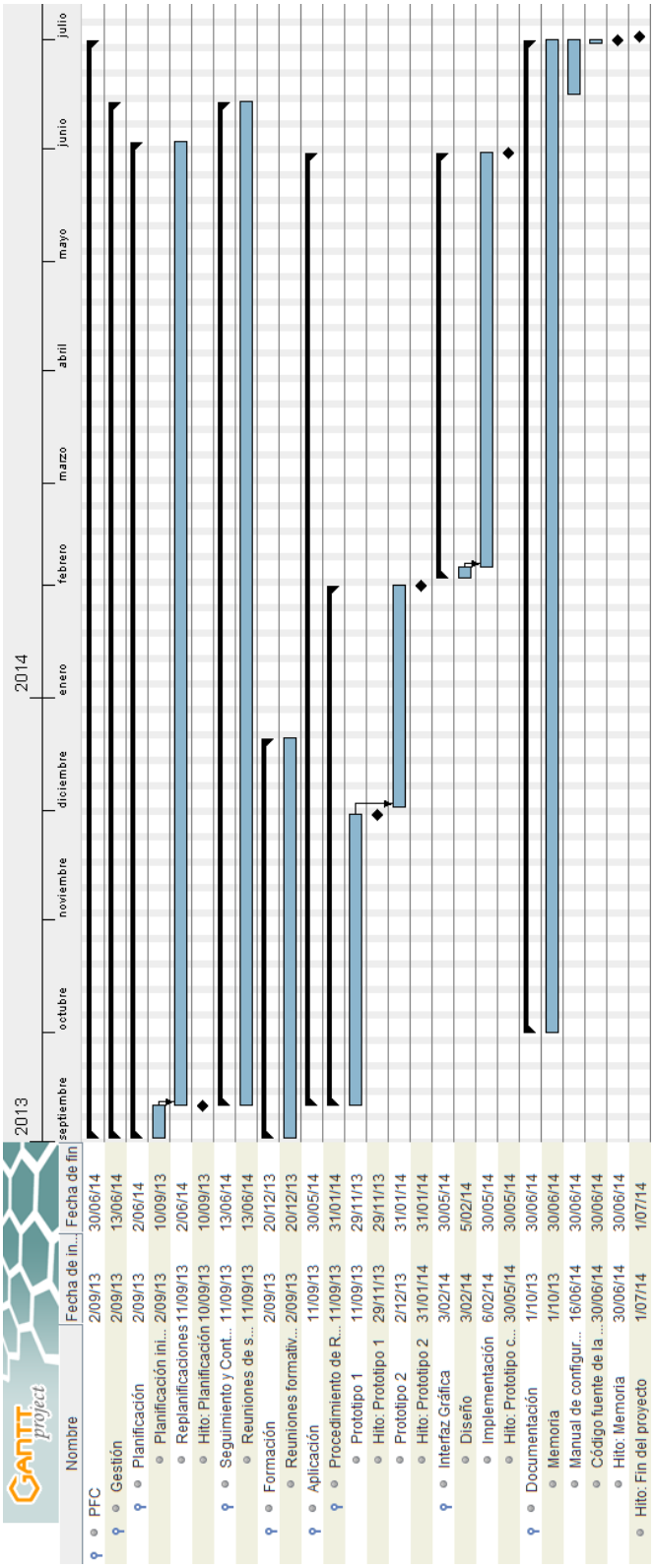

Figura 1.2: Diagrama de Gantt

1.2.7. Gestión de Comunicaciones

En esta sección estableceremos el plan de comunicaciones. Para ello, se identificarán a los interesados del proyecto y se explicarán los métodos de comunicación entre los mismos.

1.2.7.1. Interesados

Principalmente, los interesados de este proyecto son:

- Directora: Marisa Navarro
- Alumno: Javier Albors

1.2.7.2. Métodos de comunicación

La comunicación no presencial entre los dos interesados se hará mediante correo electrónico. Sin embargo, el método de comunicación principal será presencial, puesto que permite una comunicación mucho mayor y facilita llegar más rápidamente a una solución o a la resolución de alguna duda.

1.2.8. Gestión de Riesgos

A continuación se analizarán los diferentes riesgos que pueden suceder, indicando la gravedad con la que afectarían al proyecto, la probabilidad de que ocurran, las medidas previstas para la prevención de dichos riesgos y las medidas correctoras a seguir en caso de que sucedan.

Mala elección de las tecnologías a utilizar

- **Gravedad**: muy alta
- **Probabilidad**: baja
- **Prevención**: emplear los primeros días del proyecto a analizar si la tecnología a utilizar cumple con las necesidades del proyecto.
- **Corrección**: escoger nuevas tecnologías rápidamente para no afectar a la planificación.

Planificación inadecuada

- **Gravedad**: media
- **Probabilidad**: media
- **Prevención**: imposible.

- **Corrección**: replanificar aquello que sea necesario para obtener una buena planificación.

Pérdida de trabajo realizado

- **Gravedad**: muy alta
- **Probabilidad**: baja
- **Prevención**: realizar copias de seguridad cada semana y también cada vez que se haga un avance relevante. Además, utilizar también formato en papel para trabajar.
- **Corrección**: recuperar la última copia digital guardada y utilizar los apuntes en papel para recuperar el trabajo perdido.

Retrasos

- **Gravedad**: media
- **Probabilidad**: media
- **Prevención**: ajustarse a la planificación lo más posible.
- **Corrección**: replanificar lo que sea necesario para minimizar el impacto.

Enfermedad

- **Gravedad**: baja
- **Probabilidad**: baja
- **Prevención**: imposible.
- **Corrección**: seguir con la planificación acordada y, en caso de necesitarlo, emplear las horas sobrantes para la realización del trabajo atrasado.

Capítulo 2

Antecedentes

A fin de entender lo que se ha implementado en el proyecto es necesario comprender algunos conceptos previos. Éstos conceptos son: documento XML, patrón, constraint, cláusula, monomorfismo, $p_1 \otimes p_2$ (operar), $p_1 \otimes_{c,m} q$ (operar con común), las reglas R1, R2, R3, Subsunción y, por último, las reglas Unfold1 y Unfold2.

2.1. Documento XML

Consideramos un documento XML como un árbol cuyos nodos están etiquetados a partir de un alfabeto infinito Σ. Los símbolos de dicho alfabeto Σ pueden representar tanto las etiquetas de los elementos, las etiquetas de los atributos o cualquier valor que conlleve texto en un documento XML. A continuación explicaremos una serie de conjuntos y símbolos con los que trataremos de aquí en adelante.

T_Σ Dado un alfabeto Σ, denominamos T_Σ como el conjunto de todos los documentos que trabajan sobre el alfabeto Σ.

$Nodes(t)$ Dado un documento $t \in T_\Sigma$, definimos $Nodes(t)$ como el conjunto que contiene todos los nodos del documento t.

$Label(n)$ Dado un nodo $n \in Nodes(t)$, $Label(n)$ devuelve la etiqueta de dicho nodo n.

$Root(t)$ Dado un documento $t \in T_\Sigma$, la función $Root(t)$ devuelve el nodo raíz del documento t.

$Edges(t)$ Dado un documento $t \in T_\Sigma$, definimos $Edges(t)$ como el conjunto que contiene todos los arcos de t. Un arco es una relación entre dos nodos e

indica que un nodo es el padre de otro y se representa con la notación (x, y), donde $x, y \in Nodes(t)$ y significa que x es el padre de y.

$Edges^+(t)$ El conjunto $Edges^+(t)$ denota la clausura transitiva del conjunto $Edges(t)$. Cada elemento $(x, y) \in Edges^+(t)$ indica que existe un camino en t desde el nodo x hasta el nodo y.

En la figura 2.1 se puede ver un ejemplo de documento XML. A la izquierda se muestra el documento tal y como es. A la derecha, el modo en el que lo vamos a representar y a tratar en este proyecto.

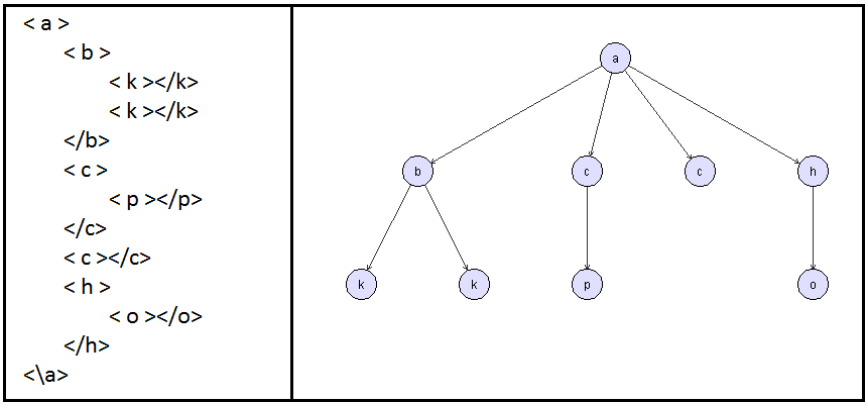

Figura 2.1: Ejemplo de documento XML

2.2. Patrón

Dado un alfabeto Σ, definimos un patrón como un árbol cuyos nodos tienen como etiqueta símbolos pertenecientes a $\Sigma \cup \{*\}$ y cuyos arcos pueden ser de dos tipos: de tipo / (que representa un *hijo directo*) o de tipo // (que representa un *descendiente*). La principal diferencia con los documentos es que estos últimos no pueden tener nodos con etiqueta "*"ni pueden tener arcos de tipo descendiente. A continuación mostraremos las diferencias entre los conjuntos definidos anteriormente para los documentos y los de los patrones.

P_Σ Análogamente a T_Σ para los documentos, P_Σ denota el conjunto de todos los patrones definidos sobre el alfabeto Σ.

$Nodes(p), Label(n), Root(p)$ Estos conjuntos representan lo mismo que en el caso de los documentos, salvo que p es un patrón y que $n \in Nodes(p)$.

$Edges(p)$ En el caso de los patrones $Edges(p)$ representa también el conjunto de arcos de p. Sin embargo, cabe destacar que el conjunto es la unión de otros dos: $Edges_{/}(p)$ y $Edges_{//}(p)$. El primero representa el conjunto de todos los arcos de tipo *hijo directo*, mientras que el segundo representa todos los arcos de tipo *descendiente*.

$Edges^{+}(p)$ Al igual que en los documentos, $Edges^{+}(p)$ denota la clausura transitiva de $Edges(p)$, es decir, $(x, y) \in Edges^{+}(p)$ indica que existe un camino en p desde el nodo x hasta el nodo y, independientemente del tipo de arcos (/ o //) que contenga dicho camino.

En la figura 2.2 se puede ver un ejemplo de patrón.

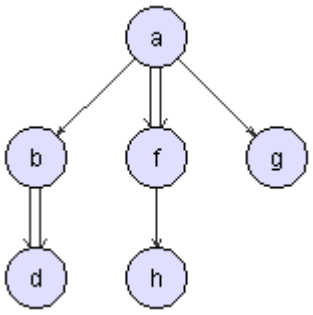

Figura 2.2: Ejemplo de patrón

2.2.1. Monomorfismo

En este apartado definiremos la noción de monomorfismo entre dos patrones y, como caso particular, la noción de monomorfismo entre un patrón y un documento. Este último se utilizará para comprobar qué documentos son modelo de un patrón concreto.

Dados dos patrones p y q, se define un homomorfismo como una función $h : Nodes(p) \to Nodes(q)$, cumpliéndose las siguientes condiciones:

- Preservación de las raíces:

$$h(Root(p)) = Root(q)$$

- Preservación de las etiquetas:

$$\forall n \in Nodes(p), Label(n) = * \vee Label(n) = Label(h(n))$$

- Preservación de las aristas *hijo*:

$$\forall (x, y) \in Edges_{/}(p), (h(x), h(y)) \in Edges_{/}(q)$$

- Preservación de las aristas *descendiente*:

$$\forall(x,y) \in Edges_{//}(p), (h(x),h(y)) \in Edges^{+}(q)$$

Dicho de otro modo, un homomorfismo indica una relación entre los nodos de un patrón p con los de otro patrón q. Respecto a los nodos, dicha relación debe cumplir que haya una **relación directa entre las raíces** y que **todo nodo n del patrón p debe tener una imagen en q** y además que **la etiqueta de dicho nodo n sea o bien '*' o bien la etiqueta del nodo imagen**. Respecto a las aristas debe cumplirse que, por un lado, si en p hay un nodo m que sea *hijo directo* de otro nodo n, **la imagen de m en q también será *hijo directo*** de la imagen de n; y, por otro lado, si en p hay un nodo m que sea *descendiente* de otro nodo n, **debe haber un camino desde la imagen de n hasta la imagen de m en q**.

En la figura 2.3 podemos ver dos diferentes soluciones de monomorfismo de un patrón p en otro patrón q. Para más información sobre el monomorfismo consultar el apartado 5.4.

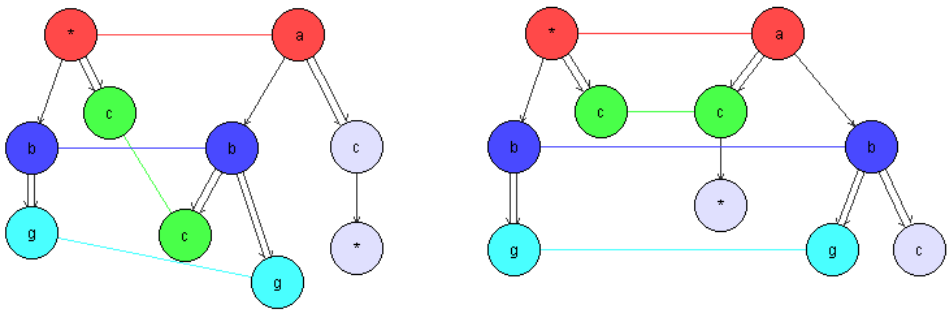

Figura 2.3: Monomorfismo de un patrón p en otro patrón q

Puesto que un documento podemos considerarlo como un patrón cuyas aristas son únicamente de tipo /, el homomorfismo entre un patrón y un documento es un caso particular de la definición anterior. De ahora en adelante trataremos tan sólo con homomorfismos *inyectivos*, los cuales son llamados *monomorfismos*. Por tanto, a partir de ahora emplearemos únicamente este término.

2.3. Constraints, Cláusulas y Especificaciones

La idea principal de los **constraints** que vamos a utilizar en este proyecto es que representen restricciones sobre un documento XML, que deben ser cumplidas. Los diferentes tipos de constraint definidos son *positivo* ($\exists p$), *negativo* ($\neg\exists p$) y *condicional* ($\forall(c : p \rightarrow q)$). A continuación se da la definición formal de cuándo un documento t satisface cada tipo de constraint C, denotado $t \models C$:

- $t \models \exists p$ si existe un monomorfismo $m : p \to t$;
- $t \models \neg\exists p$ si no existe ningún monomorfismo $m : p \to t$;
- $t \models \forall(c : p \to q)$ si para cada monomorfismo $m : p \to t$ existe un monomorfismo $f : q \to t$ tal que $m = f \circ c$.

Definimos una **cláusula** como una disyunción finita de constraints. Formalmente, una cláusula α es una disyunción finita de literales $L_1 \vee L_2 \vee \ldots \vee L_n$, donde para cada $i \in \{1, \ldots, n\}$, el literal L_i es un constraint positivo, negativo o condicional. La disyunción vacía se conoce como *cláusula vacía* y se representa como *FALSE*.

Un documento $t \in T_\Sigma$ satisface una cláusula α, denotado $t \models \alpha$, si cumple:

- $t \models L_1 \vee L_2 \vee \ldots \vee L_n$ si $t \models L_i$ para algún $i \in \{1, \ldots, n\}$.

Las **especificaciones** que vamos a tratar en este proyecto consisten en un conjunto de cláusulas, representadas como las hemos definido antes. Dados un documento y una especificación, el documento satisface la especificación siempre que satisfaga todas y cada una de las cláusulas de la especificación. Sin embargo, puede que dicha especificación no pueda ser satisfecha nunca. Por ello, el objetivo de este proyecto es analizar la satisfacibilidad de una especificación y por tanto el procedimiento de refutación trabajará con especificaciones como datos de entrada.

2.4. Operaciones

A lo largo del proceso de refutación es necesaria la aplicación de tres operaciones: monomorfismo (vista antes), $p_1 \otimes p_2$ y $p_1 \otimes_{c,m} q$. Estas operaciones se utilizan principalmente en las diferentes reglas que componen el procedimiento de refutación. A continuación se explicarán las dos últimas.

2.4.1. $p_1 \otimes p_2$

Dados dos patrones p_1 y p_2, la operación $\otimes$ denota el conjunto de patrones que se pueden obtener mediante la combinación de p_1 y p_2. Es decir, si intentamos obtener todos los patrones que cumplen tanto p_1 como p_2 y eliminamos todos aquellos que sean casos particulares de otros más generales, entonces obtendríamos el conjunto $p_1 \otimes p_2$. A un nivel matemático, podemos definir la operación $p_1 \otimes p_2$ como el conjunto de patrones:

$$p_1 \otimes p_2 = \{s \in P_\Sigma \mid \exists monomorf.\ m_1 : p_1 \to s \wedge \exists monomorf.\ m_2 : p_2 \to s \\ \wedge\ Nodes(s) = m_1(Nodes(p_1)) \cup m_2(Nodes(p_2))\}$$

Debemos tener en cuenta que en el caso de la operación $p_1 \otimes p_2$ siempre va a haber solución excepto en un único caso: que las raíces de p_1 y de p_2 tengan etiquetas diferentes y ninguna de ellas sea asterisco.

En la figura 2.4 se puede ver un ejemplo de la operación $\otimes$. En la parte izquierda se muestran los dos patrones a operar. En la parte derecha, la solucion.

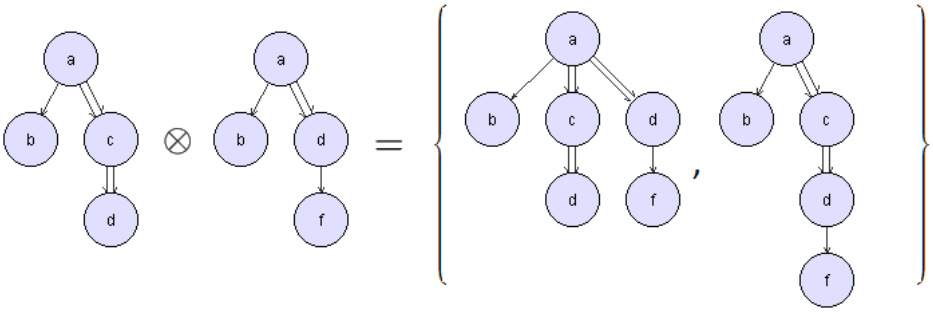

Figura 2.4: Operación $p_1 \otimes p_2$

Podemos encontrar otros patrones que también cumplan los patrones p_1 y p_2 de la figura 2.4, pero todos serían casos particulares de alguno de los dos obtenidos. Para saber si un patrón es un caso particular de otro podemos utilizar la operación monomorfismo. Si hay algún monomorfismo de p_1 en p_2, entonces p_2 es un caso particular de p_1. Por ejemplo, el patrón de la figura 2.5 cumpliría p_1 y p_2 pero es un caso particular de la solución, por lo que se elimina de la misma.

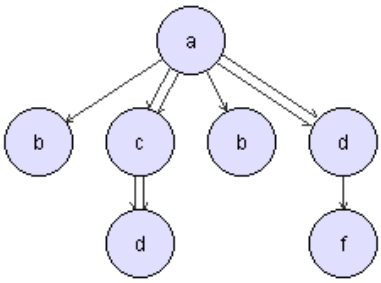

Figura 2.5: Patrón caso general de la solución de la figura 2.4

Para más información sobre la operación $\otimes$ consultar el apartado 5.5.

2.4.2. $p_1 \otimes_{c,m} q$

Antes de explicar esta operación cabe destacar que no se operan dos patrones. La entrada está formada por una función pre-árbol $c : p_2 \to q$ (donde p_2 es el pre-árbol de q), por un patrón p_1 y por una función monomorfismo

$m : p_2 \to p_1$ que *no es extensible a* $q \to p_1$. La salida, $p_1 \otimes_{c,m} q$, representa el conjunto de patrones que se pueden obtener combinando p_1 y q de todas las maneras posibles, pero manteniendo p_2 intacto. A nivel matemático, se define la operación $p_1 \otimes_{c,m} q$ como el conjunto de patrones:

$$p_1 \otimes_{c,m} q = \{s \in P_\Sigma \mid \exists monomorf.\ inc1 : p_1 \to s \wedge \exists monomorf.\ inc2 : q \to s \ /\ inc1 \circ m = inc2 \circ c\}$$

A diferencia de la operación $p_1 \otimes p_2$ (ver apartado 2.4.1), donde la solución podría ser el conjunto vacío, en esta operación siempre va a haber solución. Debido a que se parte de un monomorfismo $m : p_2 \to p_1$, la mínima solución contendría al patrón obtenido a partir de p_1 insertándole los nodos de q que no pertenecen a $Nodes(p_2)$. Concretamente, los patrones s pertenecientes a $p_1 \otimes_{c,m} q$ se obtienen añadiendo a p_1 todos los nodos y aristas de $q - c(p_2)$ en el lugar indicado por el monomorfismo m.

Para más información sobre esta operación, consultar el apartado 5,6.

2.5. Reglas

A continuación se explicarán las diferentes reglas que utiliza el procedimiento de refutación. Las reglas principales son R1, R2 y R3, cuya función básica es la de crear una nueva cláusula a partir de otras dos. La regla de Subsunción, pese a no ser necesaria, es muy útil ya que permite la eliminación de cláusulas redundantes. Por último, las reglas Unfold son necesarias para la completitud del procedimiento, pero se aplican de diferente modo a las anteriores.

Las reglas R1, R2 y R3 toman como entrada dos cláusulas. Si esas cláusulas cumplen ciertas condiciones (en cada regla son diferentes), la regla obtiene como resolvente una nueva cláusula, la cual se añade a la especificación actual obteniendo otra equivalente.

2.5.1. Regla R1

Las condiciones para que se pueda ejecutar la regla R1 son dos:

1. Que dadas dos cláusulas C_1 y C_2, uno de los constraints de C_1 sea del tipo $\exists p_1$ y uno de los constraints de C_2 sea del tipo $\neg\exists p_2$. El resto de la cláusulas de C_1 y C_2 se expresan, respectivamente, como Γ_1 y Γ_2.

2. Que una vez se cumpla la primera condición, exista un monomorfismo de p_2 en p_1. Esta condición se conoce como $cond_1$.

Si se cumplen las dos condiciones, se creará una nueva cláusula C_3 que contendrá la disyunción $\Gamma_1 \vee \Gamma_2$. En el caso de que tanto Γ_1 como Γ_2 sean

vacíos, se obtendrá una cláusula vacía, la cual representa a *FALSE*. En la figura 2.6 se muestra un esquema de la regla R1.

$$\frac{\exists p_1 \vee \Gamma_1 \quad \neg\exists p_2 \vee \Gamma_2}{\Gamma_1 \vee \Gamma_2} \quad \textbf{(R1)}$$

si existe un monomorfismo $m : p_2 \rightarrow p_1$

Figura 2.6: Regla R1

2.5.2. Regla R2

Las condiciones para que se pueda ejecutar la regla R2 son:

1. Que dadas dos cláusulas C_1 y C_2, uno de los constraints de C_1 y uno de los constraints de C_2 sean del tipo $\exists p$.
2. Que una vez se cumpla la primera condición, no exista un monomorfismo de p_2 en p_1 o de p_1 en p_2. Esta condición se conoce como $cond_2$ (no es necesaria pero, si se tiene en cuenta, se evita trabajo de más en el proceso de refutación).

Si se cumplen las dos condiciones, se creará una nueva cláusula C_3 que, a diferencia de la regla R1, no sólo contendrá la disyunción $\Gamma_1 \vee \Gamma_2$ sino también una disyunción de constraints positivos formados por el conjunto de patrones que cumplen tanto p_1 como p_2, esto es, $p_1 \otimes p_2$ (ver apartado 2.4.1). Para poder obtener *FALSE* mediante esta regla es necesario que Γ_1 y Γ_2 sean vacíos y que p_1 y p_2 tengan raíces diferentes (su operación dé un conjunto vacío). En la figura 2.7 se muestra un esquema de la regla R2.

$$\frac{\exists p_1 \vee \Gamma_1 \quad \exists p_2 \vee \Gamma_2}{(\bigvee_{p \in p_1 \otimes p_2} \exists p) \vee \Gamma_1 \vee \Gamma_2} \quad \textbf{(R2)}$$

si no existe ningún monomorfismo $m \ / \ m : p_2 \rightarrow p_1 \vee m : p_1 \rightarrow p_2$

Figura 2.7: Regla R2

2.5.3. Regla R3

Mientras que las dos reglas anteriores pueden encontrar el *FALSE*, la regla R3 crea siempre cláusulas no vacías. Las condiciones que se deben cumplir para que se pueda aplicar son:

1. Que dadas dos cláusulas C_1 y C_2, uno de los constraints de C_1 sea del tipo $\exists p_1$ y uno de los constraints de C_2 sea del tipo $\forall(c : p_2 \to q)$.

2. Que exista un monomorfismo $m : p_2 \to p_1$ que no pueda ser extendido a $f : q \to p_1$ / $f \circ c = m$.

Tras cumplirse las dos condiciones, se creará una nueva cláusula C_3 que estará compuesta por $\Gamma_1 \vee \Gamma_2$ y $p_1 \otimes_{c,m} q$ (ver apartado 2.4.2). Como ya se ha dicho, esta regla nunca obtendrá la cláusula *FALSE*. En la figura 2.8 se muestra un esquema de la regla R3.

$$\frac{\exists p_1 \vee \Gamma_1 \qquad \forall(c : p_2 \to q) \vee \Gamma_2}{(\bigvee_{p \in p_1 \otimes_{c,m} q} \exists p) \vee \Gamma_1 \vee \Gamma_2} \quad \textbf{(R3)}$$

si existe un monomorfismo $m : p_2 \to p_1$ no extensible a $f : q \to p_1$ / $f \circ c = m$

Figura 2.8: Regla R3

2.5.4. Subsunción

La regla de Subsunción se emplea para eliminar cláusulas redundantes. Puede ser utilizada en cualquier momento de la ejecución, pero es más eficiente si se utiliza lo antes posible. La idea principal de esta regla consiste en que, si deben cumplirse las cláusulas $C_1 = A \vee B$ y $C_2 = A \vee B \vee \Gamma$, entonces C_2 no es necesaria para el análisis de satisfacibilidad, ya que C_1 es más restrictivo. Más formalmente podemos definir la subsunción como que, dadas dos cláusulas C_1 y C_2, C_1 subsume a C_2 si $Mod(C_1) \subseteq Mod(C_2)$. En la figura 2.9 se muestra el esquema de la regla de subsunción.

A lo largo del proyecto, sin embargo, se ha analizado esta regla más detalladamente y dicho análisis ha resultado en tres reglas de subsunción que mejoran aún más la eficiencia del procedimiento de refutación. Dichas reglas se pueden consultar en el apartado 5.8.

$$\frac{\Gamma_1 \vee \Gamma_2 \qquad \Gamma_1}{\Gamma_1} \quad \textbf{(Subsunción)}$$

Figura 2.9: Regla de Subsunción

2.5.5. Unfold1

Debido a que el procedimiento de refutación no es completo con el conjunto de reglas anteriores, es necesario introducir dos reglas nuevas, las cuales se cree que completan el procedimiento: $Unfold_1$ y $Unfold_2$. El objetivo de estas reglas es transformar un patrón p en la disyunción de otros dos p' y p'' (manteniendo siempre la equivalencia entre los mismos) para poder activar alguna de las reglas (R1, R2 o R3) y continuar con el proceso de refutación.

Las reglas Unfold se usan cuando no se puede aplicar ninguna otra regla. En ese momento, reemplazan los constraints positivos de las cláusulas de la especificación que así lo requieran aplicando o bien la regla $Unfold_1$ o bien la regla $Unfold_2$. El modo en el que se decide si a un constraint se le debe aplicar estas reglas o no es fijándose en si su aplicación supondrá la activación de alguna de las reglas R1, R2 o R3. En caso afirmativo, se aplican las reglas Unfold.

La regla $Unfold_1$ transforma un patrón de la siguiente manera: para empezar, el patrón a transformar debe tener al menos un arco de tipo *descendiente*, ya que la expansión se realiza sobre dicho arco. Una vez localizado, se crean dos patrones. El primero, es el mismo patrón pero con dicho arco *descendiente* cambiado a *hijo directo*. En el segundo patrón, se sustituye el arco // por un *hijo directo* de etiqueta * que, a su vez, tiene un *descendiente* que será el descendiente original.

En la figura 2.10 se muestra un ejemplo de aplicación de esta regla. En la izquierda se muestra el patrón original, y en la derecha la disyunción equivalente generada mediante la regla $Unfold_1$.

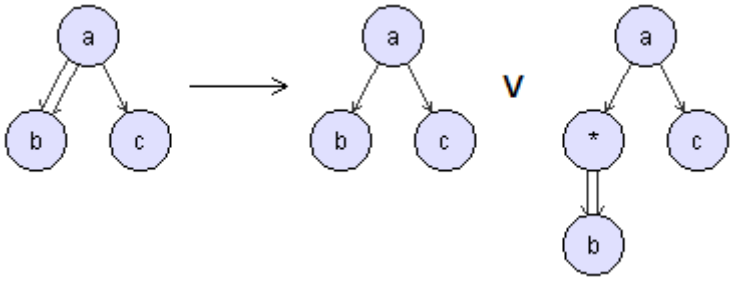

Figura 2.10: Ejemplo de aplicación de $Unfold_1$

2.5.6. Unfold2

Esta regla es muy similar a la anterior. Dado un patrón p con algún arco *descendiente*, lo transforma en la siguiente disyunción: el primer patrón es el mismo pero con arco *hijo directo*, y el segundo es el mismo pero en el lugar donde tenía el arco // se coloca un asterisco de tipo *descendiente* que tiene a su vez como *hijo directo* al descendiente del patrón original.

Para mayor aclaración, en la figura 2.11 se muestra un ejemplo de la aplicación de esta regla.

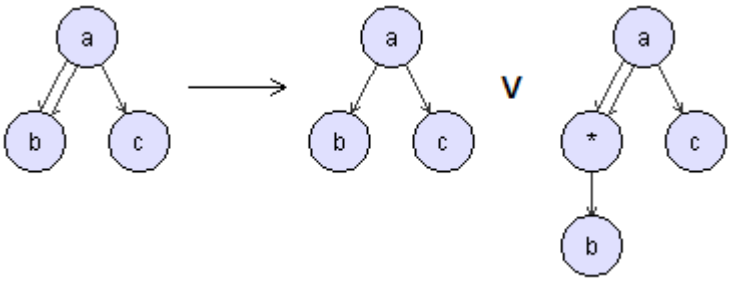

Figura 2.11: Ejemplo de aplicación de $Unfold_2$

Capítulo 3

Aplicación: Captura de Requisitos

En este capítulo se analizarán los diferentes requisitos y se obtendrá el consiguiente modelo de casos de uso, indicando la descripción, precondiciones, postcondiciones, referencias a los requisitos, escenario principal y diagrama de secuencia de los mismos. Para una explicación más extensa acerca del modo de uso de la aplicación, consultar el Manual de Uso en el apartado A.3.

3.1. Requisitos

Los diferentes requisitos que debe cumplir la aplicación son los siguientes:

1. Analizar satisfacibilidad de una especificación.
2. Comprobar si un documento dado es modelo de una especificación.
3. Realizar el monomorfismo entre dos patrones.
4. Operar dos patrones ($p_1 \otimes p_2$).
5. Operar con común un patrón y un condicional ($p_1 \otimes_{c,m} q$).
6. Consultar el historial.
7. Consultar cláusulas y constraints.

3.2. Modelo de Casos de Uso

A partir del conjunto de requisitos anterior se ha obtenido el siguiente modelo de casos de uso:

- Procedimiento de refutación v1.
- Procedimiento de refutación v2.
- Comprobar especificación.
- Monomorfismo.
- Operar ($p \otimes q$).
- Operar con común ($p_1 \otimes_{c,m} q$).
- Consultar cláusulas.
- Consultar constraints.

A continuación se detallarán uno a uno.

3.2.1. Procedimiento de Refutación v1

Descripción El usuario introduce un conjunto de cláusulas (especificación) y el sistema analiza su satisfacibilidad mediante el uso de las reglas R1, R2, R3 y Subsunción (ver apartado 2.5).

Precondiciones No se puede introducir una especificación vacía. Tampoco puede haber ninguna cláusula vacía.

Postcondiciones Se devuelve un valor booleano indicando la satisfacibilidad de la especificación y se genera un historial de ejecución.

Referencias Requisitos 1 y 6.

Escenario principal

- El usuario introduce la especificación y ejecuta el procedimiento de refutación.
- El sistema devuelve el resultado y le muestra al usuario el historial de la ejecución.
- El usuario hace uso, si quiere, de los casos de uso *Consultar Cláusulas* y *Consultar Constraints*.

Diagrama de secuencia En la figura 3.1 se muestra el diagrama de secuencia del sistema de este caso de uso.

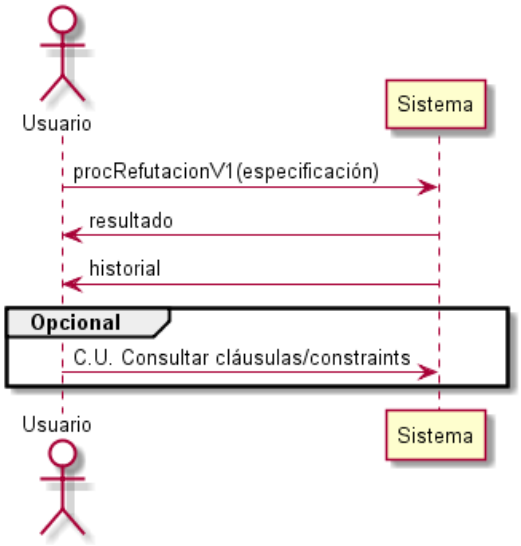

Figura 3.1: Diagrama de secuencia de *Procedimiento de Refutación v1*

3.2.2. Procedimiento de Refutación v2

Este caso de uso es exactamente igual al anterior pero incluyendo en la ejecución la aplicación de las reglas Unfold. En la figura 3.2 se muestra el diagrama de secuencia del sistema de este caso de uso.

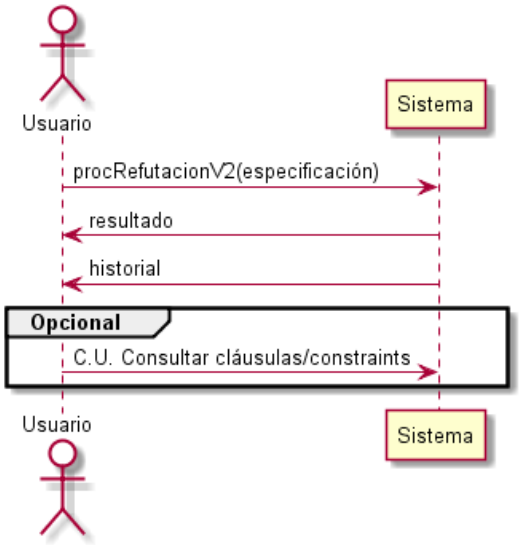

Figura 3.2: Diagrama de secuencia de *Procedimiento de Refutación v2*

3.2.3. Comprobar Especificación

Este caso de uso no se contempló en la planificación inicial. Sin embargo, debido a la y utilidad del mismo, se ha decidido incluirlo en el proyecto.

Descripción El usuario introduce un documento XML y un conjunto de cláusulas (especificación) y el sistema indica si el documento XML es un modelo de la especificación proporcionada.

Precondiciones El documento XML no debe ser vacío, así como tampoco la especificación. Tampoco puede haber ninguna cláusula vacía.

Postcondiciones Se devuelve un valor booleano indicando si el documento satisface la especificación o no.

Referencias Requisito 2.

Escenario principal

- El usuario introduce un documento XML y una especificación y llama al método *comprobar especificación*.
- El sistema devuelve un booleano con el resultado.

Diagrama de secuencia En la figura 3.3 se muestra el diagrama de secuencia del sistema de este caso de uso.

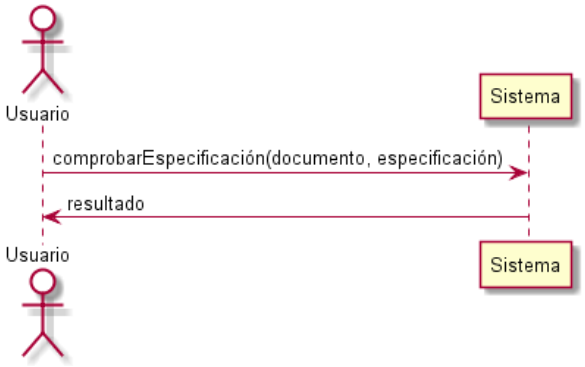

Figura 3.3: Diagrama de secuencia de *Comprobar Especificación*

3.2.4. Monomorfismo

Descripción El usuario introduce dos patrones, p y q, y el sistema calcula todos los monomorfismos de p en q.

Postcondiciones El sistema muestra todos los monomorfismos que existen de p en q.

Referencias Requisito 3.

Escenario principal

- El usuario introduce dos patrones y llama al método *monomorfismo*.
- El sistema muestra las diferentes soluciones.

Diagrama de secuencia En la figura 3.4 se muestra el diagrama de secuencia del sistema de este caso de uso.

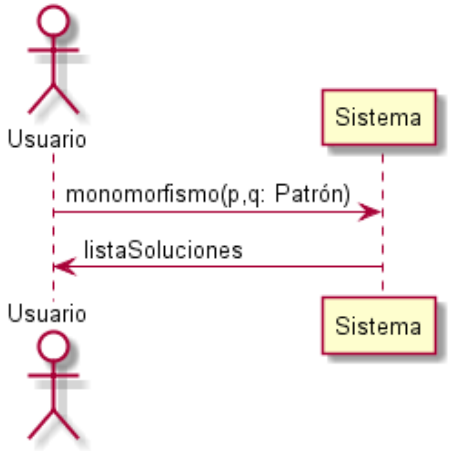

Figura 3.4: Diagrama de secuencia de *Monomorfismo*

3.2.5. Operar ($p_1 \otimes p_2$)

Descripción El usuario introduce dos patrones, p_1 y p_2, y el sistema opera ambos patrones y muestra la solución.

Postcondiciones Se muestra el resultado de operar p con q ó un mensaje de error en caso de que no exista una solución.

Referencias Requisito 4.

Escenario principal

- El usuario introduce dos patrones y llama al método *operar*.
- El sistema muestra la solución.

Diagrama de secuencia En la figura 3.5 se muestra el diagrama de secuencia del sistema de este caso de uso.

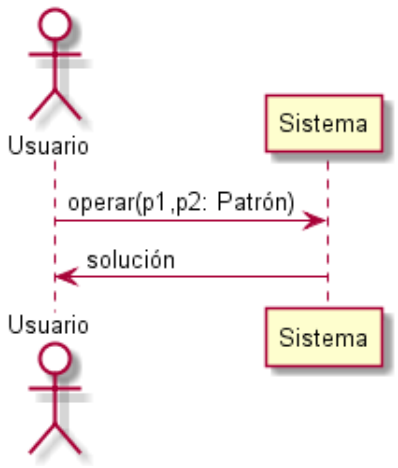

Figura 3.5: Diagrama de secuencia de *Operar*

3.2.6. Operar con Común ($p_1 \otimes_{c,m} q$)

Descripción Se introduce un constraint positivo p_1 y un constraint condicional $\forall(c : p_2 \to q)$ y el sistema calcula todos los monomorfismos $m : p_2 \to p_1$ no extensibles a $q \to p_1$ y para cada uno de ellos calcula la operación $p_1 \otimes_{c,m} q$.

Postcondiciones Se devuelven varias soluciones, una por cada monomorfismo de p_2 en p_1 no extensible a $q \to p_1$ ó un mensaje de error en caso de que no existe una solución.

Referencias Requisito 5.

Escenario principal

- El usuario introduce dos patrones y llama al método *generar pre-árbol*.
- El sistema calcula y muestra todas las posibles formas en que p_2 puede ser pre-árbol de q.
- El usuario selecciona una de las relaciones de pre-árbol.
- El sistema muestra una pantalla donde el usuario introducirá el patrón p_1.
- El usuario introduce el patrón p_1 y llama al método *operar con común*.

- El sistema calcula todos los monomorfismos de p_2 en p_1 no extensibles a $q \to p_1$, obtiene la solución para cada uno de ellos y las muestra.

Diagrama de secuencia En la figura 3.6 se muestra el diagrama de secuencia del sistema de este caso de uso.

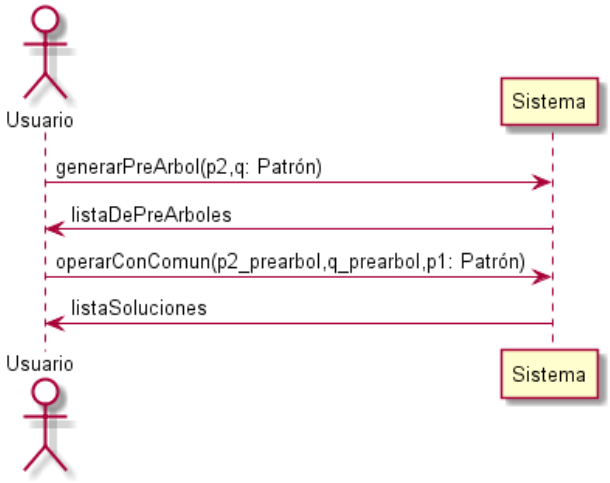

Figura 3.6: Diagrama de secuencia de *Operar con Común*

3.2.7. Consultar Cláusulas

Este caso de uso no es uno que se puede ejecutar en cualquier momento, sino tan sólo una vez se ha cargado el historial después de ejecutar los casos de uso *Procedimiento de Refutación v1* o *Procedimiento de Refutación v2*.

Descripción Se introduce el identificador de una cláusula y se mostrará el contenido de dicha cláusula.

Precondiciones La cláusula debe existir.

Postcondiciones Se visualiza la información referente a la cláusula.

Referencias Requisito 7.

Escenario principal

- El usuario introduce el identificador de una cláusula.
- El sistema devuelve la información de dicha cláusula.

Diagrama de secuencia En la figura 3.7 se muestra el diagrama de secuencia del sistema de este caso de uso.

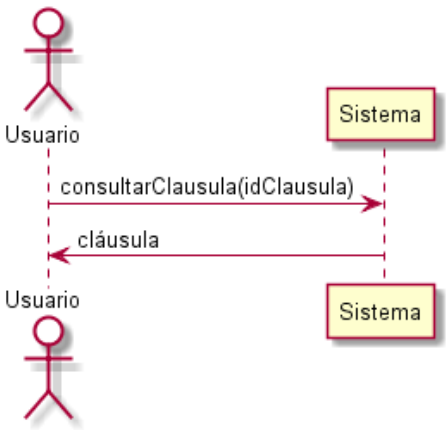

Figura 3.7: Diagrama de secuencia de *Consultar Cláusulas*

3.2.8. Consultar Constraints

Este caso de uso es exactamente igual al de *Consultar Cláusulas* pero introduciendo el identificador de un constraint y, por tanto, mostrando la información de dicho constraint. En la figura 3.8 se muestra el diagrama de secuencia del sistema de este caso de uso.

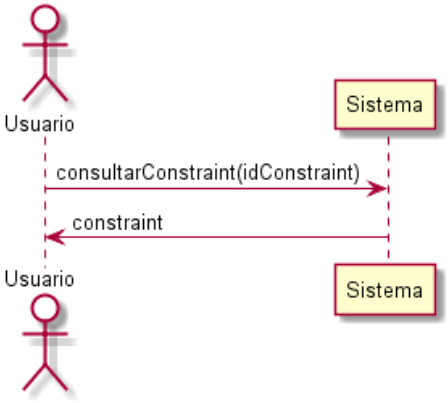

Figura 3.8: Diagrama de secuencia de *Consultar Constraints*

Capítulo 4

Aplicación: Diseño

En este capítulo se mostrará cómo es la estructura de la aplicación, tanto la arquitectura que se ha diseñado como el modelo de clases de cada capa de la arquitectura.

4.1. Arquitectura

La idea de la aplicación era seguir un modelo de arquitectura en tres capas: *Capa de Interfaz Gráfica*, *Capa de Lógica* y *Capa de Datos*. Sin embargo, surgen varios problemas. Por un lado, la capa de Datos es únicamente para guardar especificaciones, es decir, que durante el proceso de ejecución no se consulta ningún dato externo. Por tanto, la capa de Datos sólo tiene interacción con la capa de Interfaz Gráfica, ya que lo que se almacena en ficheros es la información que la capa de Interfaz envía a la capa de Lógica.

Por otro lado, la capa de Lógica se va a implementar en Prolog, mientras que el resto de las capas se implementarán en Java. Esto hace que entre la capa de Interfaz y la de Lógica se deberá usar un puente Java-Prolog. Por ello, se ha decidido crear una cuarta capa (*Aplicación*) que haga de mediadora entre las otras tres y tenga el rol de controladora, siendo esta capa la que utilice el puente Java-Prolog y traduzca los datos de un formato a otro.

En la figura 4.1 se puede ver un esquema de la arquitectura diseñada.

4.1.1. Puente Java-Prolog

El puente para conectar Java con Prolog que se ha utilizado es el que trae el propio SWI-Prolog: jpl. El modo de usarlo es muy sencillo: se importa la librería *jpl.jar* y se crea un objeto de tipo *Query*. En el constructor de dicho Query se indica un String con el comando que se escribiría en la interfaz de SWI-Prolog: *consult(patrones.pl)* y se ejecuta *q.open()* (siendo *q* la variable Query). Hecho eso, la conexión está ya en marcha. Para hacer cualquier

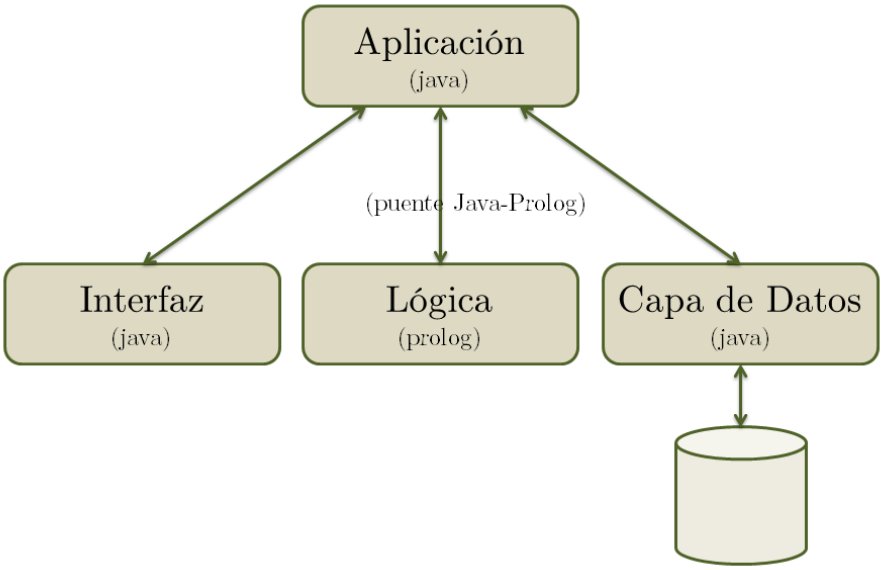

Figura 4.1: Arquitectura de la aplicación

consulta se realiza de la misma forma que en el inicio de conexión.

Sin embargo, lo anterior sólo funciona una vez el puente Java-Prolog quede configurado. Para saber cómo se configura consultar el apartado A.2.

4.2. Modelo de Clases

En este apartado se analizarán las diferentes clases que componen la aplicación. Para facilitar la comprensión de las mismas, se dividirán las capas en diferentes apartados y se explicarán una a una todas aquellas clases que sean medianamente relevantes para la capa de la arquitectura a la que pertenecen.

4.2.1. Capa de Aplicación

Como ya se ha indicado antes, esta capa es la encargada de mediar entre las capas de Interfaz, Lógica y Datos y además de controlar las mismas. Por tanto, contendrá métodos pertenecientes a las tres capas, actuando como *fachada* de las tres capas para las tres capas. Las clases que más explicación requieren de esta capa son *Aplicación* y *AnalizadorPatrones*.

Aplicación

Esta clase es, por un lado, la que contiene los métodos que las otras capas van a utilizar y, por otro, es la lanzadora de la aplicación. Por tanto, se ha diseñado para que sólo pueda haber una única instancia de esta clase.

Entre sus atributos tiene:

- un objeto **panelPrincipal** de tipo PanelPrincipal (perteneciente a la capa de Interfaz), que será la ventana principal de la aplicación. Esta ventana debe estar siempre abierta y, si se cierra, se cierra toda la aplicación. Por eso es necesario tener guardada la instancia de la pantalla principal.
- una lista de JFrame-s **listaVentanas** donde se guardarán todas las ventanas abiertas que no sean la pantalla principal. De este modo, si se intenta cerrar la pantalla principal y esta lista de ventanas no es vacía, se le advierte al usuario diciéndole que hay más ventanas abiertas.
- una instancia de AnalizadorPatrones **analizador**, el cual se usará para traducir las respuestas proporcionadas por Prolog a objetos Java con los que pueda trabajar la interfaz.
- un objeto con la instancia de la capa de Datos llamada **capaDatos**.

Los métodos principales que contiene la clase Aplicación son:

- *getInstance()*: devuelve la instancia de la clase Aplicación.
- *anadirVentana(frame)*: añade a la lista **listaVentanas** una nueva ventana que se ha abierto.
- *dispose(frame)*: se cierra la ventana que se pasa por parámetro. Si la ventana está en **listaVentanas**, se elimina también de la lista. Sino, si la ventana es la pantalla principal, se comprueba que no hay otras ventanas abiertas y se termina la ejecución. Si hubiera ventanas abiertas se pregunta al usuario si desea cerrar todas las ventanas y salir.
- *getAnalizador()*: devuelve el objeto **analizadorPatrones**.
- *nuevo()*: cierra la ventana principal actual y abre otra completamente nueva.
- *abrir(nombre)*: se cierra la pantalla principal actual y se abre la especificación guardada en el fichero cuyo nombre se pasa por parámetro. Si hay algún fallo al leer el fichero, se abre una pantalla principal nueva (sin cláusulas). Si no ha habido fallos, se abre una pantalla principal con la especificación cargada.

- *guardarFichero(nombre,contC,contCT,cc)*: se guarda en un fichero la especificación de la pantalla principal.
- *abrirHistorial()*: abre una ventana nueva donde se muestra el historial de ejecución.
- *exportarHistorial(nombreFich,historial)*: se guarda en un fichero el historial de ejecución.
- *numerarPreArbol(pre,arbol)*: establece una o más relaciones de preárbol para dos patrones dados (conexión con Prolog).
- *hacerMonomorfismo(p1,p2)*: realiza el monomorfismo entre dos patrones dados (conexión con Prolog).
- *operar(p1,p2)*: realiza la operación $\otimes$ entre dos patrones dados (conexión con Prolog).
- *operarComun(p2,q,p1)*: realiza la operación $\otimes_{c,m}$ entre dos patrones dados (conexión con Prolog).
- *hacerGensymC(n)*: realiza n veces *"gensym(c,X)"* (conexión con Prolog).
- *hacerGensymCT(n)*: realiza n veces *"gensym(ct,X)"* (conexión con Prolog).
- *resetGensym()*: realiza el comando *"reset_gensym"* (conexión con Prolog).
- *aplicarVersion1(conjuntoClausulas)*: dada una especificación ejecuta la versión 1 del procedimiento de refutación (conexión con Prolog).
- *aplicarVersion2(conjuntoClausulas)*: dada una especificación ejecuta la versión 2 del procedimiento de refutación (conexión con Prolog).
- *cumpleEspecificacion(doc,conjuntoClausulas)*: dados un documento y una especificación comprueba si el documento satisface la especificación (conexión con Prolog).
- *historial()*: obtiene el historial y lo guarda en objetos Java (conexión con Prolog).
- *obtenerTodasClausulas()*: obtiene la información acerca de todas las cláusulas obtenidas en la ejecución del procedimiento de refutación (conexión con Prolog).
- *obtenerClausula(c)*: obtiene la información acerca de la cláusula cuyo identificador se pasa por parámetro (conexión con Prolog).

- *obtenerConstraint(ct)*: obtiene la información de un constraint cuyo identificador se pasa por parámetro (conexión con Prolog).

En la figura 4.2 se muestra el diagrama de esta clase.

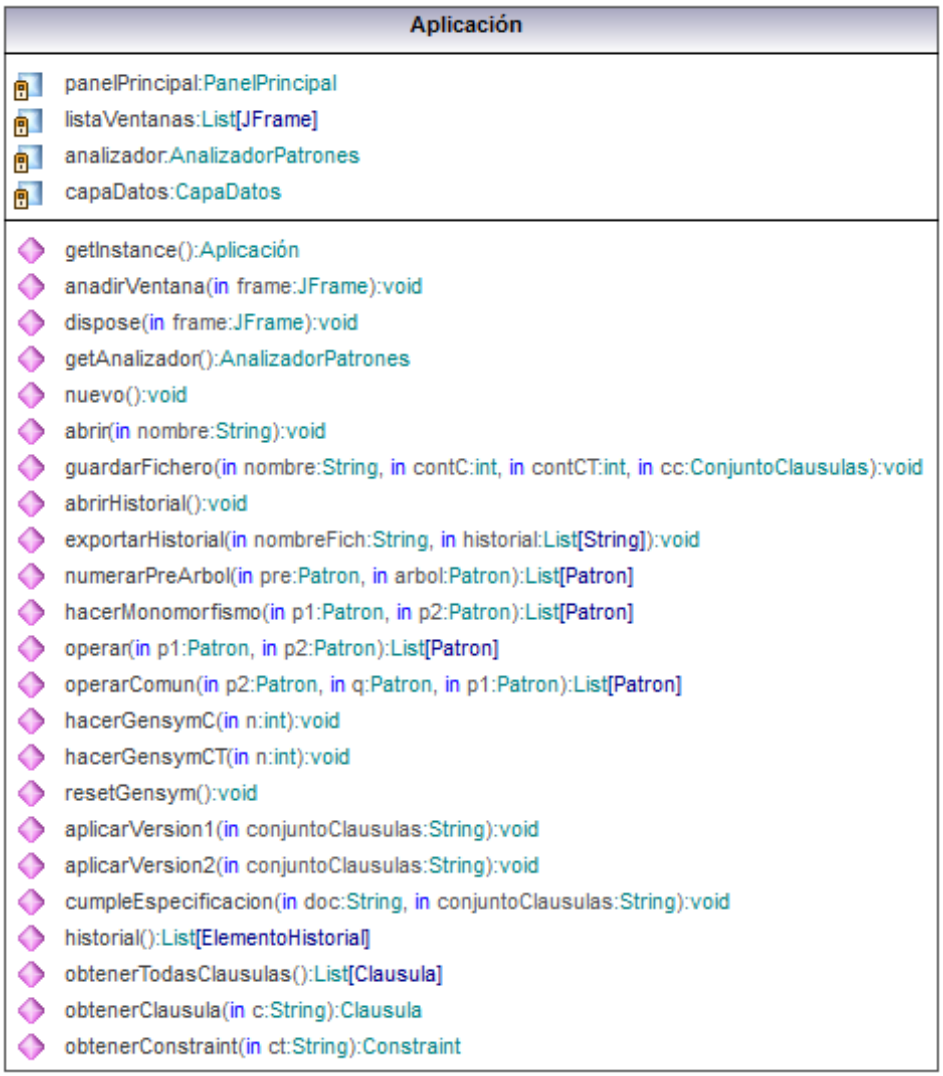

Figura 4.2: Diagrama de la clase Aplicación

AnalizadorPatrones

Esta clase se utiliza para traducir los patrones, constraints y cláusulas (entre otros) desde su notación en Prolog a su notación Java.

La clase AnalizadorPatrones no contiene ningún atributo que merezca la pena mencionar. Respecto a los métodos más relevantes, éstos son:

- *analizarPatron(p)*: dado un string que contiene la información de un patrón en notación Prolog, devuelve una estructura Java con la información de dicho patrón.
- *analizarConstraint(ct)*: dado un string que contiene la información de un constraint en notación Prolog, devuelve una estructura Java con la información de dicho constraint.
- *analizarClausula(c)*: dado un string que contiene la información de una cláusula en notación Prolog, devuelve una estructura Java con la información de dicha cláusula.

En la figura 4.3 se muestra el diagrama de esta clase.

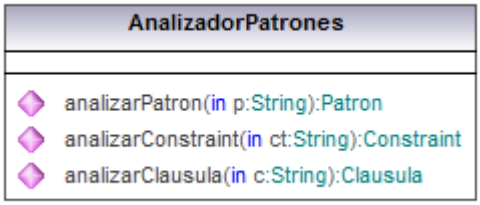

Figura 4.3: Diagrama de la clase AnalizadorPatrones

Otras Clases

En este apartado se mostrarán el resto de las clases de la *Capa de Aplicación* y seguidamente el diagrama de clases de la capa.

Patron Esta clase contendrá toda la información necesaria de un patrón. Los atributos de esta clase son **nodo** e **hijos**. Para más información acerca de los patrones y su notación en Prolog consultar el apartado 5.1.

Hijo Esta clase contendrá la información referente a los hijos de los patrones (ya sean *hijos directos* o *descendientes*). Sus atributos son **esHijo** y **patron**.

Nodo Esta clase contendrá la información referente a los nodos de los patrones. Sus atributos son **etiqueta**, **id**, **id2** y **esSimple**.

Constraint Esta clase contiene la información de un constraint. Sus atributos son **id**, **tipoConstraint**, **patronConstraint** y **preArbol**.

Clausula Esta clase guardará la información de una cláusula. Los atributos de esta clase son **id** y **constraints**.

ConjuntoClausulas Esta clase se utiliza únicamente para poder escribir una especificación en notación Prolog. Su único atributo es **clausulas** y su único método, además del constructor, es *toString*, que escribe con notación Prolog la especificación.

En la figura 4.4 se muestra el diagrama de clases de la Capa de Aplicación.

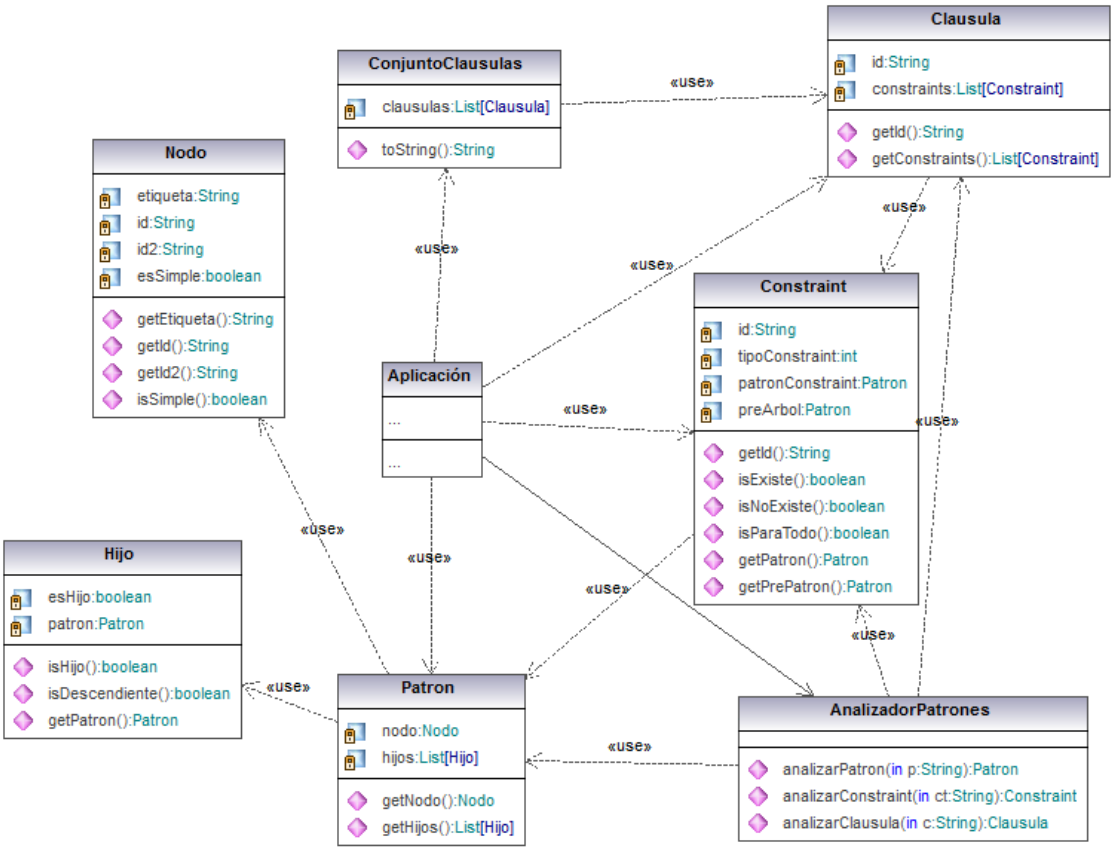

Figura 4.4: Diagrama de clases de la Capa de Aplicación

4.2.2. Capa de Interfaz Gráfica

Esta capa es la encargada de visualizar ventanas y de interactuar con el usuario, tanto mostrándole resultados como recibiendo los datos introducidos. Las clases más importantes de esta capa son *Editor*, *CirculoNodo*, *RelacionNodos*, *EditorCTParaTodo* y *PanelPrincipal*.

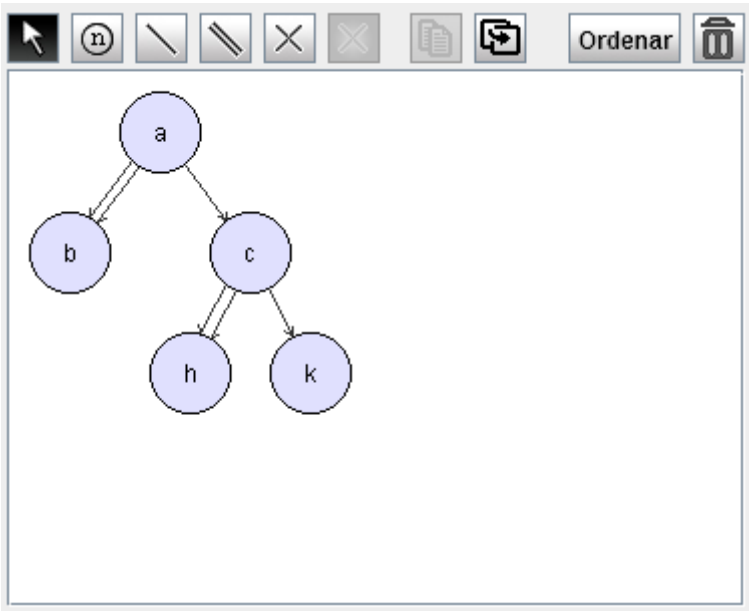

Figura 4.5: Panel *Editor*

Editor

Esta clase es una extensión de la clase JPanel. Por tanto, no es una ventana en sí misma, aunque para convertirla en ventana bastaría con colocar este panel en un JFrame. Se utiliza para la construcción visual de patrones, con la idea de que forme parte de todas aquellas ventanas en las que se necesite dibujar o introducir un patrón.

Como se puede observar en la figura 4.5, el editor se compone de varios botones. Los cinco de la izquierda (seleccionar, crear nodo, crear relación de hijo directo, crear relación de descendiente, eliminar) son los cinco estados en los que puede estar el editor. El siguiente es para eliminar tan sólo los nodos seleccionados. Los dos centrales son para copiar y pegar nodos en diferentes editores de la aplicación; y los dos botones de la derecha son para reordenar el patrón dibujado y para limpiar el editor.

Puesto que en la aplicación se utilizan los editores para introducir datos pero también para más cosas como por ejemplo mostrar únicamente datos sin permitir modificaciones o mostrar un monomorfismo, es posible cargar el editor de diferentes maneras, teniendo así diferentes efectos sobre su apariencia y su modo de funcionar. A continuación se explicará más detalladamente mediante el análisis de los atributos y de los métodos más relevantes de la clase.

Los atributos de los que se compone la clase Editor son:

- **width**: es un entero que guarda la anchura en píxeles que tendrán

visualmente los nodos de los patrones.

- **height**: es un entero que guarda la altura en píxeles que tendrán visualmente los nodos de los patrones. Dependiendo del valor que tenga, los nodos tendrán forma de elipse o de círculo. Puesto que es más cómodo visualizar círculos, se ha decidido que height tenga el mismo valor que **width**.

- **esNumerado**: es un valor booleano que indica si el patrón que se está dibujando (o mostrando) es numerado o no. Para más información acerca de los patrones y su representación consultar el apartado 5.1. Cabe decir que el editor siempre numera los patrones. Es este parámetro el que luego decide si se debe devolver un patrón numerado o un patrón normal.

- **ultimoIdent**: contiene el máximo identificador de nodos actual. Si se introduce algún nodo nuevo, éste tendrá un identificador de valor $ultimoIdent + 1$.

- **patrones**: este atributo contendrá los patrones dibujados en un formato que puede entender la Capa de Aplicación. Se habla de patrones en plural porque el editor permite dibujar varios patrones en un mismo editor.

- **nodos**: es una lista con los diferentes nodos que se han dibujado. Están en un formato sólo entendible por la clase Editor, por lo que para enviarlos a otras clases o capas es necesario generar los patrones dibujados en el atributo **patrones**.

- **relaciones**: contiene la relación padre-hijo (o padre-descendiente) entre cada par de nodos relacionados. Para poder generar los patrones en el atributo **patrones**, tan sólo es necesaria la información de este atributo y el de **nodos**.

- **lNodosSeleccion**: estando en el estado *selección*, es posible seleccionar varios nodos ya sea para moverlos, eliminarlos, etc. Se utiliza esta lista de nodos para guardar los nodos que actualmente están seleccionados.

- **area**: contiene el valor del ancho y alto del panel actual. Se utiliza para mostrar las barras de desplazamiento en caso de que el área actual del panel sea mayor que el espacio visual de la ventana.

- **modoLectura**: este atributo se utiliza para decirle al editor qué tipo de patrón (o patrones) está dibujando. Tiene cinco valores posibles: *normal*, *prearbol*, *monomorfismo*, *operar* y *operarcomun*. En el caso de los modos *prearbol*, *monomorfismo* y *operarcomun*, el editor estará en

modo *sólo lectura*. En el modo *normal* se podrá editar siempre y, en el modo *operar*, se podrá editar o no dependiendo del tipo de constructor con el que se haya creado el objeto Editor.

- **soloLectura**: es un valor booleano que indica si se puede editar o tan sólo visualizar el contenido del panel. Si está activado el modo *sólo lectura*, las únicas acciones que se podrán hacer son seleccionar y mover nodos, copiar y reordenar.
- **modo**: es el modo en el que está el editor en ese momento. Los diferentes modos en los que se puede encontrar son: *modoseleccion*, *modonodo*, *modohijo*, *mododescendiente*, *modoeliminar* y *modoespera*. Los cinco primeros estados se pueden seleccionar mediante los cinco botones de la izquierda que se pueden ver en la figura 4.5. El estado *modoespera* es el que se activa cuando hay que escribir la etiqueta de un nodo, el cual inhabilita todos los botones hasta que se termine de escribir y entonces retoma el estado anterior en el que se encontraba.

A continuación se explicarán los diferentes métodos de la clase Editor:

- *Editor()*: es el constructor por defecto. Crea un editor vacío con **modoLectura** = *modonormal*, **soloLectura** = *false* y **esNumerado** = *false*.
- *Editor(p,soloLectura)*: Se construye un editor en el que se visualiza el patrón p y con los atributos **modoLectura** = *modonormal*, **soloLectura** = *soloLectura (parámetro)* y por último **esNumerado** será cierto sólo si el patrón p también es numerado.
- *Editor(modo)*: se construye un editor vacío en modo **soloLectura** = *true*, **modoLectura** = *modo*, y **esNumerado** = *true*. Este constructor sólo se utiliza para establecer los modos *monomorfismo*, *prearbol*, *operarcomun* y el modo no-editable de *modooperar*. Para introducir patrones en este editor se deberá utilizar el método *setPatrones*.
- *Editor(grupoOperar)*: este constructor sólo se utiliza para crear el modo editable de *modooperar*. Es un editor normal (igual que si se hubiera llamado a *Editor()*), pero la única diferencia es que el color con el que se dibujan los nodos cambia dependiendo del valor de *grupoOperar*. Si *grupoOperar* es '1', los nodos serán azules. Si es '2', serán rojos y en cualquier otro caso serán del tono grisáceo por defecto.
- *setPatrones(p)*: se le pasa por parámetro un vector de patrones, los cuales se visualizarán independientemente de si el editor está en modo *sólo lectura* o no. En el caso de que hubiera previamente otros patrones visualizándose, se eliminan dichos patrones antes de visualizar los nuevos.

- *setPadre(padre,indice,frame)*: en algunos casos se puede sacar el editor de la ventana en la que se encuentra a una nueva ventana (mediante el botón *Detach*). Cuando se hace eso, es necesario indicarle al editor cuál es la ventana original en la que se encontraba en caso de que se quiera volver a colocarla donde estaba en un principio. Mediante este método se le indica al editor cuál era la ventana en la que se encontraba (parámetro *frame*), cuál era el panel concreto dentro de la ventana en el que se encontraba (parámetro *padre*) y qué posición ocupaba dentro del panel (parámetro *indice*).

- *eliminarTodo()*: realiza la misma acción que si se hubiera pulsado el botón de *eliminar todo*.

- *getPatron()*: genera los patrones dibujados en el atributo **patrones** y devuelve dicho atributo.

En la figura 4.6 se muestra el diagrama de esta clase.

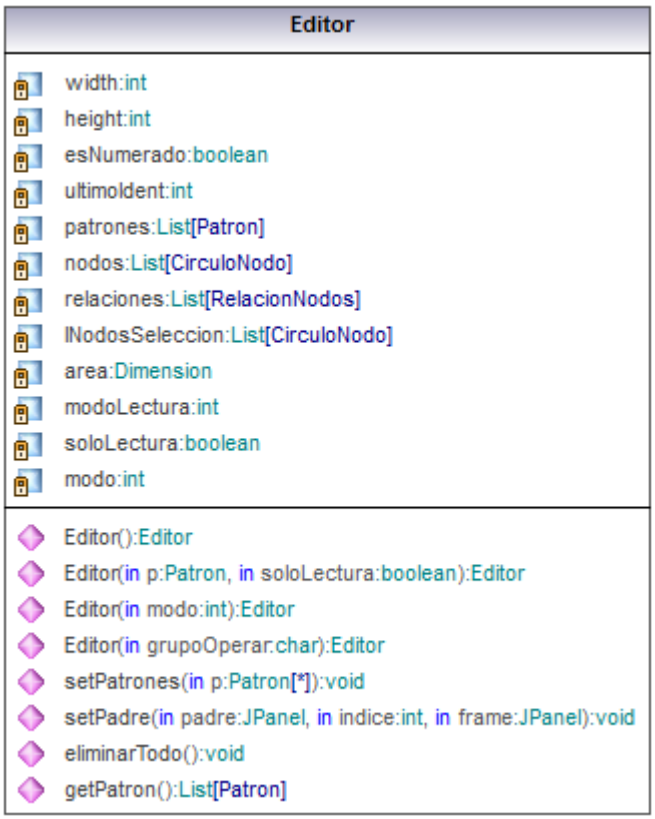

Figura 4.6: Diagrama de la clase Editor

CirculoNodo

Esta clase es una que sólo utiliza la clase *Editor* para guardar la información acerca de los nodos dibujados (excepcionalmente la clase *Aplicacion* también la utiliza pero tan sólo para almacenar los nodos copiados; en ningún momento realiza ninguna acción con ellos).

Los objetos de esta clase guardarán el contenido del nodo, así como también información acerca de la visualización del mismo: las coordenadas en las que se encuentra el nodo dentro del editor, el color por defecto del nodo y el color actual (un nodo puede ser de color azul por defecto pero, por estar seleccionado, podría estar dibujado en ese momento de color gris oscuro).

Los métodos más importantes de la clase CirculoNodo son:

- *CirculoNodo(x,y)*: es el constructor que utiliza el botón *crear nodo*. Crea un nodo sin etiqueta que se coloca en las coordenadas (x, y). Para introducir luego la etiqueta del nodo se utilizará el método *setLabel*.
- *CirculoNodo(n)*: este constructor sólo se utiliza cuando se intenta visualizar en un editor un patrón ya existente. Crea un nodo con etiqueta n y lo coloca en las coordenadas $(0, 0)$, que se recolocará automáticamente cuando se visualice el patrón entero.
- *CirculoNodo(n,x,y)*: este constructor se utiliza cuando se copian los nodos seleccionados. Para poder pegarlos con la misma distribución en la que se encuentran, es necesario guardar tanto el nodo como las coordenadas de los mismos.
- *setColor(c)*: se le indica al nodo que el color con el que se va a dibujar ahora es el color c.
- *saveColor()*: se guarda una copia del color actual del nodo, pudiendo así recuperar dicho color en cualquier momento.
- *restoreColor()*: recupera el color guardado y lo establece como color actual.
- *getColor()*: devuelve el color actual del nodo.
- *getX()/getY()*: devuelve respectivamente las coordenadas x e y del nodo.
- *getNodo()*: devuelve el nodo (no sólo la etiqueta) del elemento.
- *setLabel(label)*: cambia la etiqueta del nodo por *label*.
- *getEtiqueta()*: devuelve la etiqueta del elemento.

- *estaDentro(x,y)*: indica si el punto con coordenadas (x, y) está inscrito dentro de la circunferencia del nodo.

En la figura 4.7 se muestra el diagrama de esta clase.

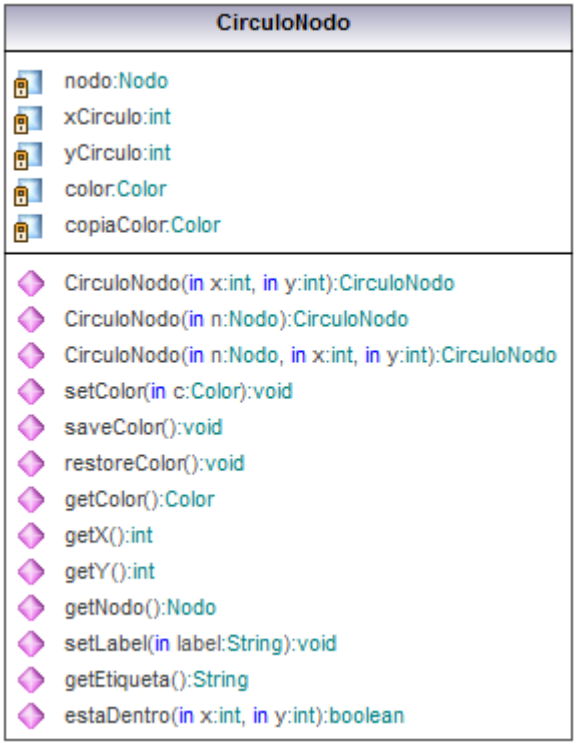

Figura 4.7: Diagrama de la clase CirculoNodo

RelacionNodos

Esta clase es la que guarda la información respecto a qué nodo es padre de otro dentro del editor.

Los atributos de la clase son:

- **padre**: este elemento representa al nodo padre de la relación.
- **hijo**: este elemento representa al nodo hijo (o descendiente) de la relación.
- **esHijo**: valor booleano que, si es *true*, indica que la relación es de *hijo directo* y, si es *false*, que la relación es de *descendiente*.

Los métodos que merecen la pena mencionar son los constructores de la clase:

- *RelacionNodos(padre,esHijo)*: este es el constructor principal. Se crea una relación en la que el padre es *padre* y el tipo de relación es la indicada por $esHijo$. Sin embargo, el nodo hijo aún no se indica. Esto es debido a que esta relación se crea cuando el usuario hace click sobre el padre durante el proceso de creación de un hijo. Una vez el usuario ha soltado el ratón encima de un nodo hijo, se establece el nodo hijo mediante el método *setHijo*.

- *RelacionNodos(padre,hijo,esHijo)*: este constructor se utiliza a la hora de copiar/pegar nodos o cuando se visualiza un patrón ya existente. Puesto que ya se sabe qué nodos forman parte de la relación, no es necesario realizar la asignación del hijo por separado.

En la figura 4.8 se muestra el diagrama de esta clase.

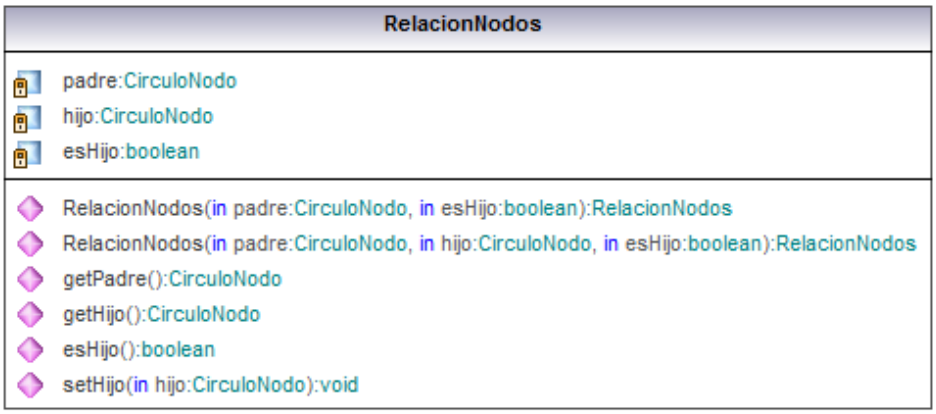

Figura 4.8: Diagrama de la clase RelacionNodos

EditorCTParaTodo

Esta clase es, al igual que la clase *editor*, una extensión de la clase JPanel y se utiliza principalmente para poder construir un constraint condicional. Sin embargo, su uso se ha extendido a las operaciones *monomorfismo*, *operar* y *operar con común*.

Como se puede apreciar en la figura 4.9, está compuesta por tres editores, dos superiores y uno inferior. El inferior siempre estará en modo *sólo lectura* y será para visualizar el constraint condicional o la solución de la operación deseada. En el caso del constraint condicional $\forall(c : p \to q)$, en el superior izquierdo se introducirá el patrón p y en el derecho el patrón q. Si la operación es el monomorfismo $m : p \to q$, se introducirá en el editor superior izquierdo el patrón p y en el derecho el patrón q. En el caso de la operación $p_1 \otimes$

p_2, puesto que es conmutativa, no importa qué patrón se introduzca en qué editor. Por último, el caso de la operación operar con común es más complicado y se explicará en otro apartado.

Los atributos de la clase *EditorCTParaTodo* son los siguientes:

- **editorPre**: es el editor superior izquierdo.
- **editorArbol**: es el editor superior derecho.
- **editorPreArbol**: es el editor inferior.
- **botonAceptar**: es el botón situado entre los editores superiores y el editor inferior. Es el encargado de obtener y mostrar la solución en el editor inferior. Dependiendo del modo en el que se haya creado la clase, su texto será *Generar pre-árbol*, *Generar monomorfismo*, *Operar* u *Operar con común*.
- **botonIzq**/**botonDer**: son los botones inferiores con forma de flechas. Cuando hay más de una solución disponible, estos botones se utilizan para cambiar la solución a mostrar.
- **modo**: indica el modo en el que se crea la clase. Los diferentes valores que puede tomar son *prearbol*, *monomorfismo*, *operar* y *operarcomun* (son los nombres de constantes enteras).
- **prearboles**: este atributo se usa para guardar la solución obtenida.

Los métodos principales de la clase son:

- *EditorCTParaTodo(modo)*: constructor por defecto. Crea los tres editores vacíos y establece el inferior en modo *sólo lectura*. El modo de la clase (*monomorfismo*, *prearbol*, etc.) viene indicada por el parámetro *modo*.
- *EditorCTParaTodo(pre,arbol,modo,soloLectura)*: este constructor sólo se utiliza en el procedimiento de refutación, para mostrar un constraint condicional que ya se había hallado previamente.
- *setParaTodo(p1,p2)*: este método sólo se utiliza para la operación *operar con común*. Como se va a explicar más adelante, en el editor superior izquierdo se mostrará el constraint condicional. Mediante este procedimiento se introduce dicho constraint en el editor.
- *getPatrones()*: devuelve el patrón (o patrones) que se muestra en el editor inferior.

En la figura 4.10 se muestra el diagrama de esta clase.

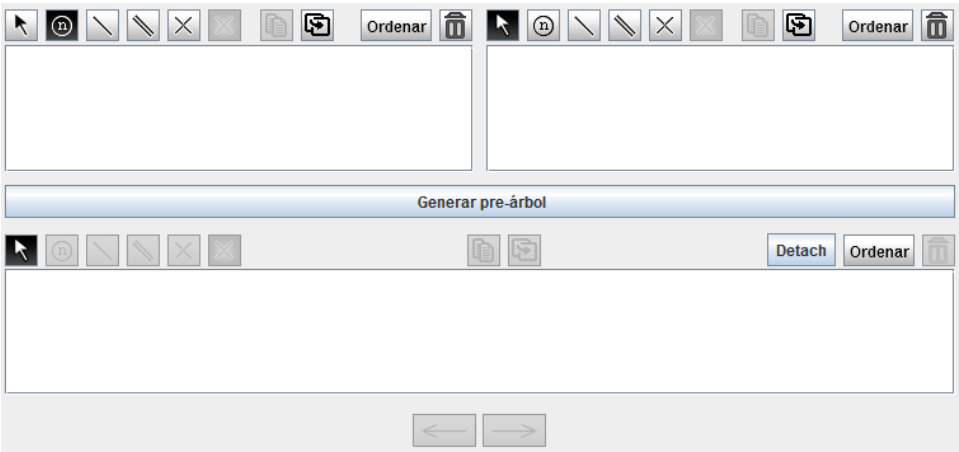

Figura 4.9: Panel *EditorCTParaTodo*

Figura 4.10: Diagrama de la clase EditorCTParaTodo

PanelPrincipal

Esta clase es una extensión de la clase JFrame, es decir, una ventana gráfica. Es la ventana principal de la interfaz y, como se puede ver en la figura 4.11, está compuesta por una barra de menú, un panel para las cláusulas, un panel para los constraints y un panel para el editor de los constraints. Aunque esta clase es la pantalla principal, se puede abrir de diferentes modos: *pantalla principal*, *comprobar especificación* e *historial*. A continuación explicaremos los atributos y métodos principales de la clase.

Los atributos principales son:

- **lClausulas**: este atributo guarda la información referente a las cláusulas que forman parte de la especificación.

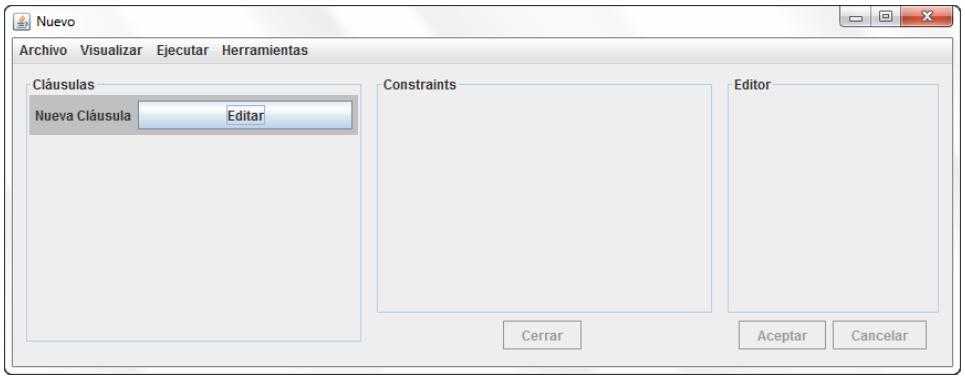

Figura 4.11: Ventana *PanelPrincipal*

- **lClausulasEliminadas**: este atributo guarda la información de las cláusulas que formaban parte de la especificación pero se han eliminado.
- **ultimaClausula**: es la cláusula de nombre *Nueva Cláusula* que aparece al final de la especificación. Al no ser en sí una cláusula, se guarda siempre aparte hasta que se termine de crear. Una vez se crea una cláusula, se genera una nueva *última cláusula*.
- **ultimoConstraint**: del mismo modo que con las cláusulas, en el panel de constraints siempre aparece al final un constraint de nombre *Nuevo Constraint*. Una vez se construye dicho constraint, se guarda dentro de la cláusula correspondiente y se genera un nuevo *último constraint*.
- **clausulaSeleccionada**: este atributo indica qué cláusula se ha seleccionado para editar o ver sus constraints. Si no hay ninguna cláusula seleccionada valdrá *null*.
- **constraintSeleccionado**: de modo análogo al atributo anterior, este atributo indica qué constraint se ha seleccionado.
- **lConstraintsHistorial**: este atributo sólo se utiliza cuando se crea un objeto de la clase *PanelPrincipal* en modo *historial*. Guardan la información de los constraints a visualizar (sin posibilidad de editarlos).
- **documento**: cuando se crea un objeto de esta clase en modo *comprobar especificación*, guarda en esta variable la información del documento introducido.
- **modo**: indica el modo en el que se ha creado el objeto de esta clase. Tiene tres valores posibles: *normal*, *lecturaclausulas* o *lecturaconstraints*. Los dos últimos son para el modo *historial*, en el que tan sólo se muestran cláusulas o constraints, sin permitir editarlos.

- **modoCumpleEspecificación**: este booleano indica si se está en el modo *comprobar especificación* o no.

- **nombreFichero**: este atributo guarda el nombre del fichero en del que se ha cargado la especificación actual y su valor se coloca como título de la ventana principal. Si la especificación se ha creado desde cero, su valor será "Nuevo".

- **modificado**: esta variable booleana indica si la especificación actual ha sufrido alguna modificación desde que se cargó. Si se ha modificado, el título de la ventana será el valor de **nombreFichero** más un asterisco, indicando así que la especificación está modificada. Además, en caso de que se cierre la ventana o se intente abrir otra especificación, el sistema preguntará al usuario si desea guardar los cambios antes de cerrar.

- **esNuevo**: esta variable indica si la especificación actual se ha creado desde cero o si se ha cargado desde algún archivo. Si se ha creado desde cero y el usuario pulsa el botón *guardar*, es necesario que el usuario indique el nombre del fichero en el que se va a guardar la especificación. Si se había cargado desde un fichero, se sobreescribirá dicho fichero.

- **contadorC**: guarda el valor del máximo identificador de cláusula generado. Cuando se crea una nueva cláusula, se incrementa este valor y se le asigna dicho identificador. Cuando se ejecute el procedimiento de refutación se llamará al método *gensymC* de la clase *Aplicación* pasándole como parámetro este mismo atributo.

- **contadorCT**: del mismo modo que la variable anterior con las cláusulas, este atributo guarda el valor del máximo identificador de constraint generado. Al ejecutar el procedimiento de refutación se llamará al método *gensymCT* de la clase *Aplicación* pasándole como argumento este atributo.

Los métodos más relevantes de la clase son:

- *PanelPrincipal(modoEspecificacion)*: este constructor es el constructor principal. El parámetro *modoEspecificacion* indica si vamos a abrir la pantalla principal o el caso de uso *comprobar especificación*. Si es *false*, se creará una pantalla principal con una especificación vacía.

- *PanelPrincipal(listaClausulas,contC,contCT,nFich)*: este constructor se utiliza para abrir la pantalla principal cargando una especificación contenida en un fichero.

- *PanelPrincipal(contC,contCT,lC,lCE)*: este constructor se utiliza únicamente para cargar la pantalla del caso de uso *comprobar especificación*. La diferencia de este constructor con el primero es que el primero cargaba el caso de uso con una especificación vacía, mientras que éste carga la pantalla del caso de uso *comprobar especificación* con una especificación no vacía.

- *PanelPrincipal(listaClausulas)*: este constructor se utiliza durante la consulta del historial. Cuando se llama al caso de uso *consultar cláusulas*, se llama a este constructor pasándole las cláusulas a mostrar. Al ser una consulta, en ningún momento se podrán modificar las cláusulas por lo que estará en modo *sólo lectura*.

- *PanelPrincipal(lConstraints)*: al igual que el constructor anterior, éste también se usa durante la consulta del historial. Se llama a este constructor cuando se quiere consultar uno o varios constraints (caso de uso *consultar constraints*).

- *repintarPaneles()*: cada vez que hay un cambio en la pantalla (porque se ha eliminado una cláusula, etc.) se llama a este método para que se visualicen los cambios producidos.

- *setClausulaSeleccionada(c)*: muestra el contenido de la cláusula *c*, estableciéndola como la cláusula seleccionada.

- *setConstraintSeleccionado(ct)*: muestra el constraint *ct*, estableciéndolo como el constraint seleccionado.

- *crearUltimaClausula()*: cuando se genera una nueva cláusula, se llama a este método y éste vuelve a construir la cláusula *Nueva Cláusula*, que es la que permite la creación de nuevas cláusulas en la pantalla principal.

- *crearUltimoConstraint()*: del mismo modo que el método anterior, construye un constraint del tipo *Nuevo Constraint*, que es con el que se podrá generar un nuevo constraint.

- *setModificado()*: indica que se ha realizado alguna modificación y se establece el atributo **modificado** a *true*.

En la figura 4.12 se muestra el diagrama de esta clase.

Otras clases

En este apartado se expondrán de forma más resumida el resto de las clases de la *Capa de Interfaz* y se mostrará el diagrama de clases de la capa completa.

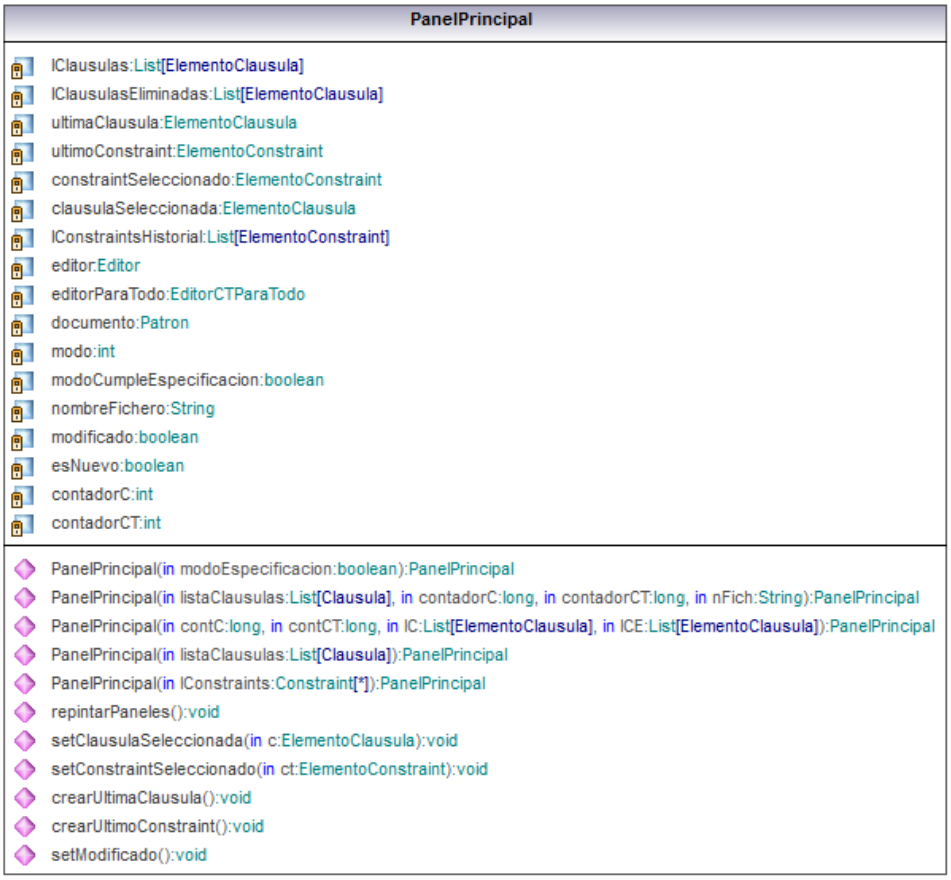

Figura 4.12: Diagrama de la clase PanelPrincipal

ElementoClausula Esta clase contiene toda la información necesaria para la visualización de una cláusula en la pantalla principal: una etiqueta con el identificador de la cláusula, el botón *editar*, etc. Tiene dos constructores: *ElementoClausula(pp)*, que crea una cláusula vacía (se usa para crear la *última cláusula*), y *ElementoClausula(c,pp,eliminado)*, que crea una cláusula con datos (se usa cuando se carga una especificación desde fichero). Con *generarClausula* se actualiza el objeto.

ElementoConstraint Esta clase es parecida a la anterior pero para constraints en lugar de cláusulas. Cuando se genera un constraint, éste llama a la cláusula en la que se va a introducir y ejecuta el método *generarClausula*.

ElementoHistorial Esta clase se utiliza para guardar toda la información necesaria de un paso del historial: número de paso, acción realizada, cláusulas involucradas, etc.

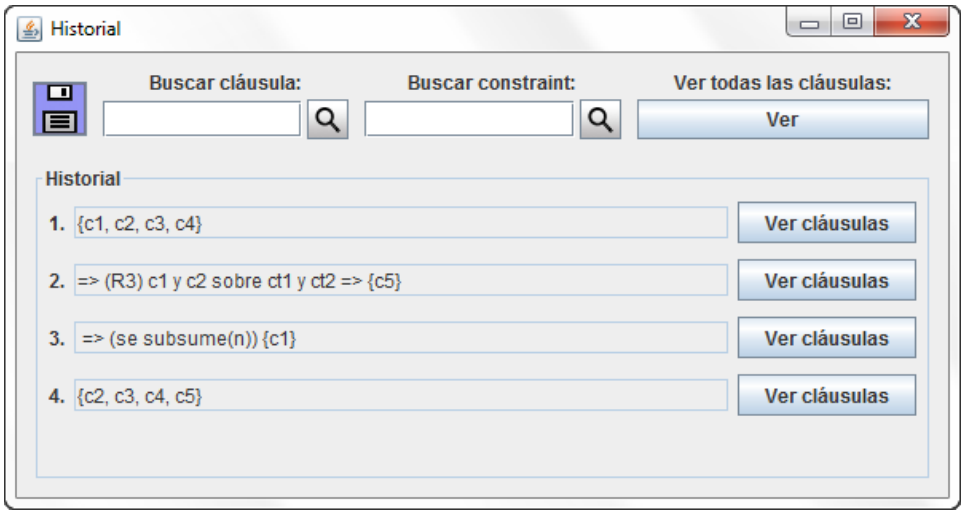

Figura 4.13: Ventana *Historial*

HistorialFrame Esta clase es una extensión de la clase JFrame, cuya apariencia se puede ver en la figura 4.13. Se utiliza para mostrar el historial y poder consultar cláusulas y constraints.

OperarComun Esta clase es también una extensión de la clase JFrame. Se utiliza para emplear la herramienta *operar con común*. Consta de dos objetos de la clase *EditorCTParaTodo* y de dos botones, *siguiente* y *anterior*, además de una etiqueta en la que se incluirán instrucciones de uso de la herramienta. El primer editor se crea con el modo *prearbol* y se usa para introducir el constraint condicional. El segundo editor se crea con el modo *operarcomun* y se usa para introducir el patrón p_1 y realizar la operación $p_1 \otimes_{c,m} q$.

En la figura 4.14 se muestra el diagrama de clases de la Capa de Interfaz.

4.2.3. Capa de Lógica

Esta capa es la encargada de procesar los datos de entrada y obtener todas las soluciones. Está escrita en Prolog y por ello este apartado no será como los de las otras capas. En este apartado expondremos los predicados más importantes para cada caso de uso y se mostrará un diagrama mostrando la interacción entre los predicados en cada caso de uso. Para más información acerca de la implementación de esta capa, consultar el capítulo 5.

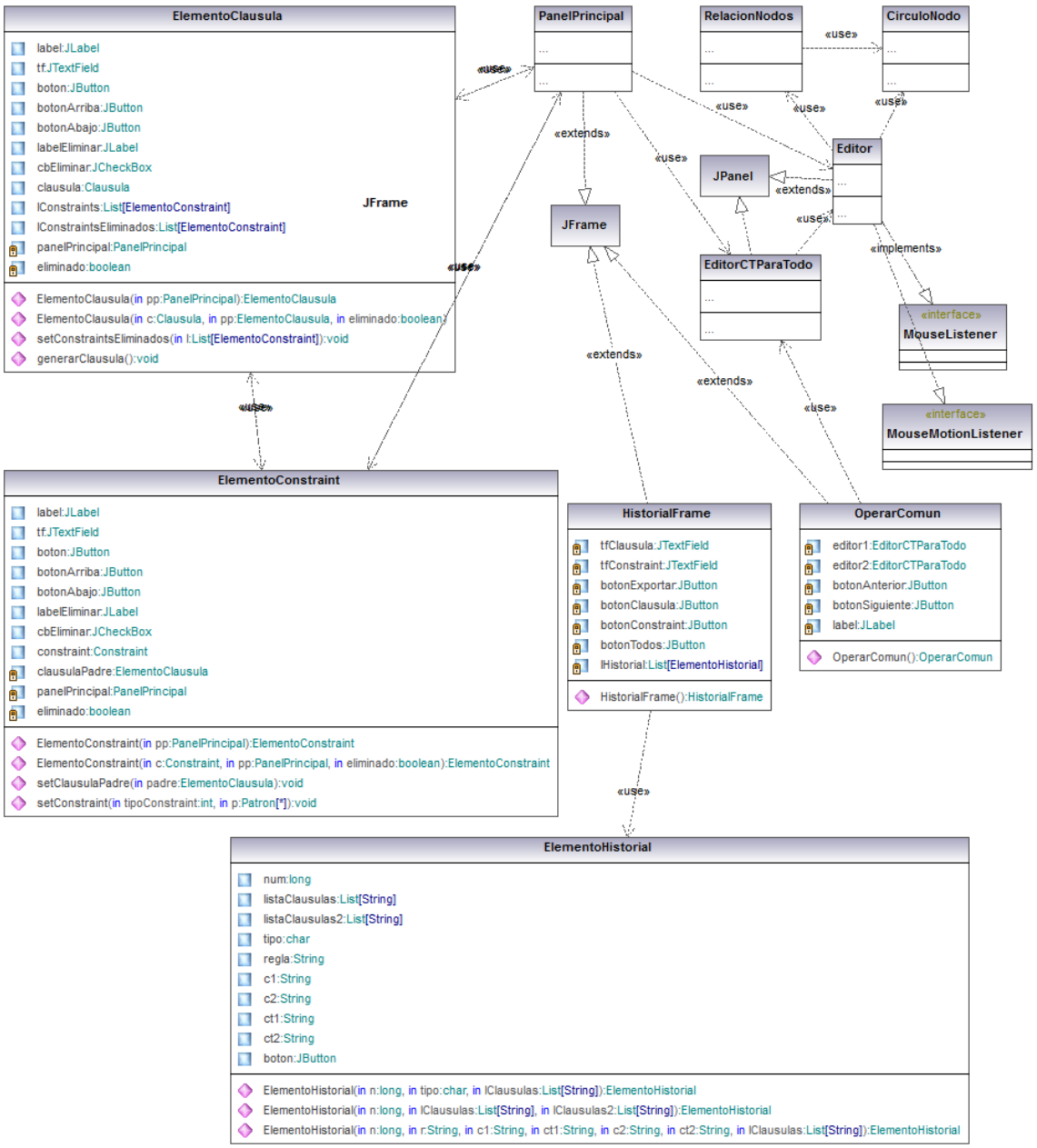

Figura 4.14: Diagrama de clases de la Capa de Interfaz

Procedimiento de Refutación v1

Este caso de uso cuenta con un total de 146 predicados, incluidos los predicados de los diferentes casos de uso que utiliza (*monomorfismo*, *operar* y *operar con común*). El predicado principal de este caso de uso es *pro-*

cRefV1/2. Recibe como primer parámetro una lista de cláusulas y devuelve en el segundo parámetro el resultado (*true* o *false*). Los predicados más relevantes son:

- *procRefV1*: como ya se ha indicado antes, este predicado es el principal del caso de uso. Dada una especificación de entrada, aplica todas las reglas posibles entre ellas y obtiene dos grupos de cláusulas: las que ya se han aplicado todas las reglas entre ellas, S, y las que todavía no se han aplicado ninguna regla con nadie, Sp. Luego, irá aplicando las reglas entre las cláusulas de S y Sp por un lado, y las de Sp consigo mismas por otro.
- *resolventes1*: este predicado se utiliza para aplicar todas las reglas posibles entre S y Sp.
- *resolventes2*: este predicado se utiliza para aplicar todas las reglas posibles entre Sp consimo mismo.
- *resolver*: dadas dos cláusulas y dos constraints pertenecientes a las mismas, determina qué regla habría que aplicar y, si se puede, la aplica.
- *cond1*/*cond2*: son los métodos que deciden si las reglas R1 y R2 se pueden aplicar o no.
- *regla1*/*regla2*/*regla3*: aplican, respectivamente, las reglas R1, R2 y R3.
- *subsumciones*: aplica la regla de subsunción entre diferentes conjuntos de cláusulas.
- *autosubsumption*: aplica la regla de subsunción entre todas las cláusulas de un mismo conjunto de cláusulas.
- *anadirClausulas*: guarda en memoria los constraints que componen una cláusula dada para poder ser consultada en el caso de uso *consultar cláusula*.
- *anadirConstraints*: guarda en memoria un constraint dado, pudiendo ser consultado en el caso de uso *consultar constraint*.
- *anadirAHistorialLista*/*anadirAHistorialListaSiNoVacia*/ *anadirAHistorialSubsumcion*/*anadirAHistorialRegla*/*anadirAHisorialFalse*: guardan en memoria toda la información referente al historial: conjunto de cláusulas actual, cláusulas subsumidas, reglas aplicadas y si se ha encontrado el *false*.
- *hayMonomorfismo*: es el predicado principal del caso de uso *monomorfismo*.

- *operar*: es el predicado principal del caso de uso *operar*.

- *operarConComun*: es el predicado principal del caso de uso *operar con común*.

En la figura 4.15 se muestra el diagrama de interacción de los predicados de este caso de uso. A pesar de la poca claridad del mismo, sirve para mostrar la complejidad de implementación del caso de uso.

Procedimiento de Refutación v2

Este caso de uso es el que más procedimientos utiliza, ya que usa los del caso de uso anterior más otros 46, formando un total de 192 predicados. El método principal es *procRefV2/2*. Al igual que en el caso de uso anterior, recibe como parámetro de entrada un conjunto de cláusulas y devuelve como segundo parámetro el resultado, *true* o *false*. Los predicados más importantes de este caso de uso son:

- *procRefV2*: ejecuta el procedimiento de refutación versión 2. Llama primero a la versión 1 y, si no se encuentra *false*, aplica las reglas Unfold y repite el proceso. Continúa hasta que se encuentre *false* o hasta que las reglas Unfold no se puedan aplicar.

- *guardarNoExistesYParaTodos*: este procedimiento almacena en memoria todos los constraints negativos y condicionales para poder ser accesibles rápidamente cuando haya que decidir si aplicar las reglas Unfold a un constraint positivo o no. Esto es debido a que las condiciones que debe cumplir un constraint positivo para que se le apliquen las reglas Unfold son tres: que tenga al menos un arco de tipo *descendiente*, que no se le haya aplicado anteriormente al constraint las reglas Unfold, y que haya algún monomorfismo desde algún constraint negativo o condicional[1] hacia el constraint positivo. Es por eso que se necesita tener accesibles dichos constraints.

- *decidirUnfold/bucleUnfold*: estos predicados son los que deciden si el procedimiento de refutación debe terminar o no. En *decidirUnfold* se comprueba si se ha obtenido el *false* y, en caso afirmativo, finaliza devolviendo *false*. Si se ha obtenido *true*, selecciona los constraints positivos a los que se les va a hacer Unfold y llama a *bucleUnfold*, que comprueba si la lista de estos constraints es vacía, en cuyo caso finalizaría devolviendo *true*. Si no, aplica las reglas Unfold.

[1] los constraints negativos y condicionales que se guardan en memoria y se usan en las reglas Unfold están ligeramente modificados. Se les cambian todas las cadenas de asteriscos que contengan por un único arco descendiente.

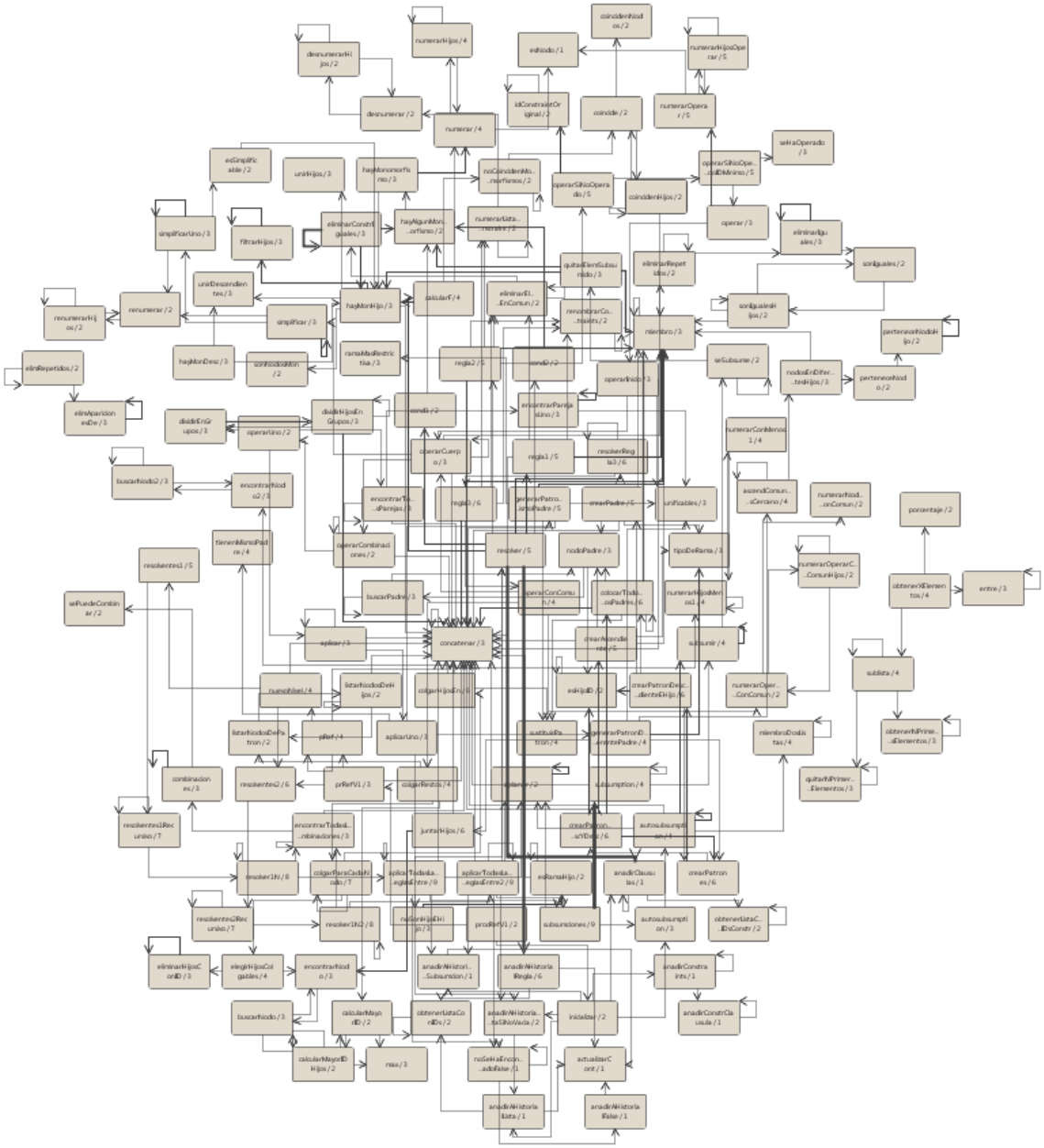

Figura 4.15: Interacción de predicados de *Procedimiento de Refutación v1*

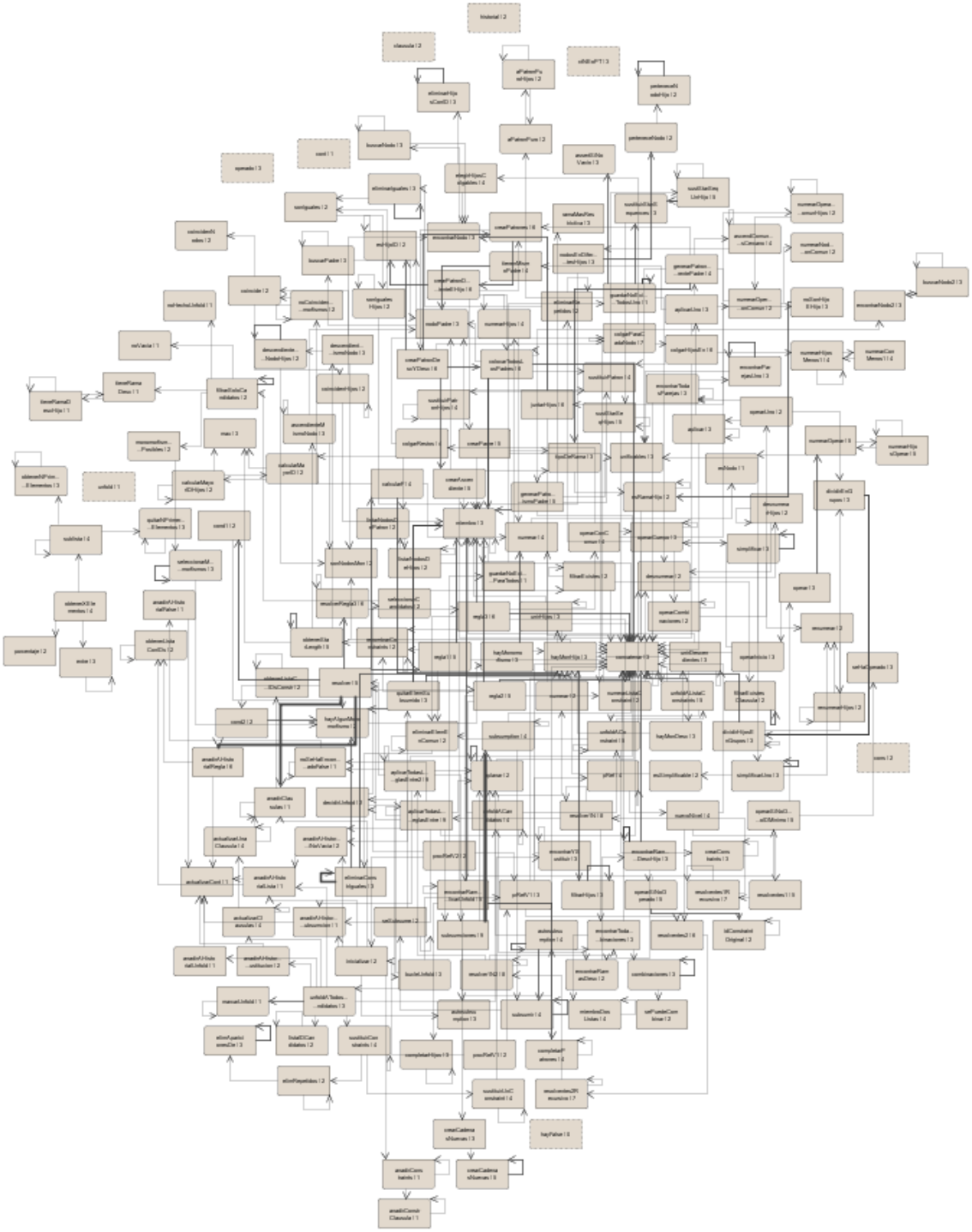

Figura 4.16: Interacción de predicados de *Procedimiento de Refutación v2*

- *seleccionarCandidatos*: este predicado es el que escoge la lista de constraints positivos a los que se les va a aplicar las reglas Unfold.

- *unfoldATodosCandidatos*: mediante este predicado se les aplica las reglas Unfold a todos los candidatos obtenidos en el método anterior.

- *sustituirConstraints*: una vez se han aplicado las reglas Unfold, este método sustituye los constraints originales por las correspondientes disyunciones de constraints obtenidos con las reglas Unfold.

- *anadirAHistorialUnfold/anadirAHistorialSustitucion*: estos dos métodos completan el historial del progama. El primer predicado guarda la información acerca de a qué constraints se les ha aplicado las reglas Unfold. El segundo guarda la información de qué cláusulas contenían a los constraints a los que se les ha hecho Unfold y por tanto han sido modificadas.

En la figura 4.16 se muestra el diagrama de interacción de los predicados de este caso de uso.

Comprobar Especificación

Este caso de uso se compone de 23 predicados. Su predicado principal es *cumpleEspecificacion/3*, que recibe como primer parámetro un conjunto de cláusulas, como segundo parámetro un documento XML y devuelve como tercer parámetro el resultado de la comprobación, *true* si el documento es un modelo de la especificación y *false* en caso contrario. Los predicados más importantes de este caso de uso son dos:

- *cumpleEspecificacion*: ejecuta el caso de uso *Comprobar Especificación*. Comprueba para cada cláusula dada en la especificación, si el documento dado satisface dicha cláusula. Si satisface todas, devuelve *true*. Si no, devuelve *false*.

- *cumpleConstraint*: indica si un documento dado satisface un constraint o no. Si el constraint es positivo, el documento lo satisface si hay algún monomorfismo del constraint en el documento. Si es un constraint negativo, el documento lo satisface si no hay ningún monomorfismo del constraint en el documento. Por último, si es un constraint condicional $\forall(c : p \rightarrow q)$, se satisface si no hay ningún monomorfismo de p en el documento o, si lo hay, que todos los monomorfismos de p en el documento sean extensibles a $q \rightarrow$ documento.

En la figura 4.17 se muestra el diagrama de interacción de los predicados de este caso de uso.

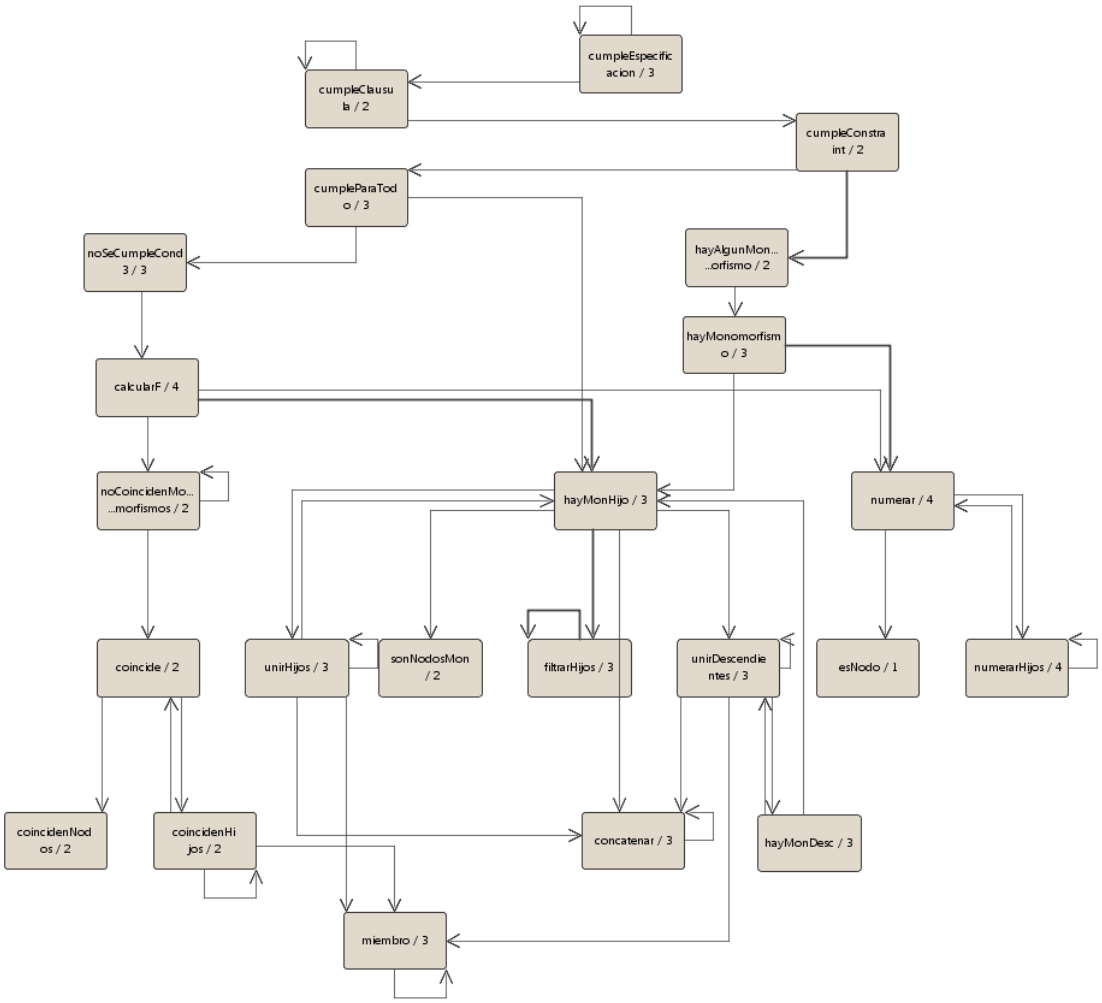

Figura 4.17: Interacción de predicados de *Comprobar Especificación*

Monomorfismo

El predicado principal de este caso de uso es *hayMonomorfismo/3*, que recibe como primer parámetro un patrón p, como segundo parámetro un patrón q y devuelve en el tercer parámetro un monomorfismo $m : p \to q$. Puede devolver más de una solución si existiera más de un monomorfismo posible de p en q. El caso de uso cuenta con un total de 12 predicados, siendo los más importantes:

- *hayMonomorfismo*: el método principal del caso de uso. Transforma los patrones de entrada ligeramente para poder trabajar con la operación monomorfismo y llama al método *hayMonHijo*.
- *hayMonHijo*: recibe dos patrones e intenta realizar el monomorfismo entre los dos. Si las raíces no se pueden unificar, para. Si se pueden unificar, los unifica y divide la tarea en dos pasos: unir los *hijos directos* y después los *descendientes*.
- *unirHijos*: este predicado se encarga de unir todos los *hijos directos* de un nodo de p con los *hijos directos* de un nodo de q. Si esto no se puede hacer, el procedimiento para.
- *unirDescendientes*: este predicado se encarga de unir todos los hijos *descendientes* de un nodo de p con algún nodo hijo o descendiente de un nodo de q. Al igual que en el predicado anterior, si esto no se puede hacer, el procedimiento para.
- *hayMonDesc*: dados dos patrones p_1 y p_2, intenta realizar el monomorfismo entre los dos. Sin embargo, la diferencia entre *hayMonHijo* y este método es que éste devuelve el monomorfismo entre p_1 y p_2 o también, si lo hubiera, el monomorfismo entre p_1 y alguno de los hijos de p_2.

En la figura 4.18 se muestra el diagrama de interacción de los predicados de este caso de uso.

Operar

Este caso de uso cuenta con un total de 60 predicados, siendo el principal *operar/3*, que recibe como primer parámetro un patrón p_1, como segundo parámetro un patrón p_2 y devuelve en el tercero el resultado de operar $p_1 \otimes p_2$. Los métodos principales son:

- *operar*: dados dos patrones p_1 y p_2, los transforma a otro formato para poder trabajar con esta operación y llama al método *operarInicio*.

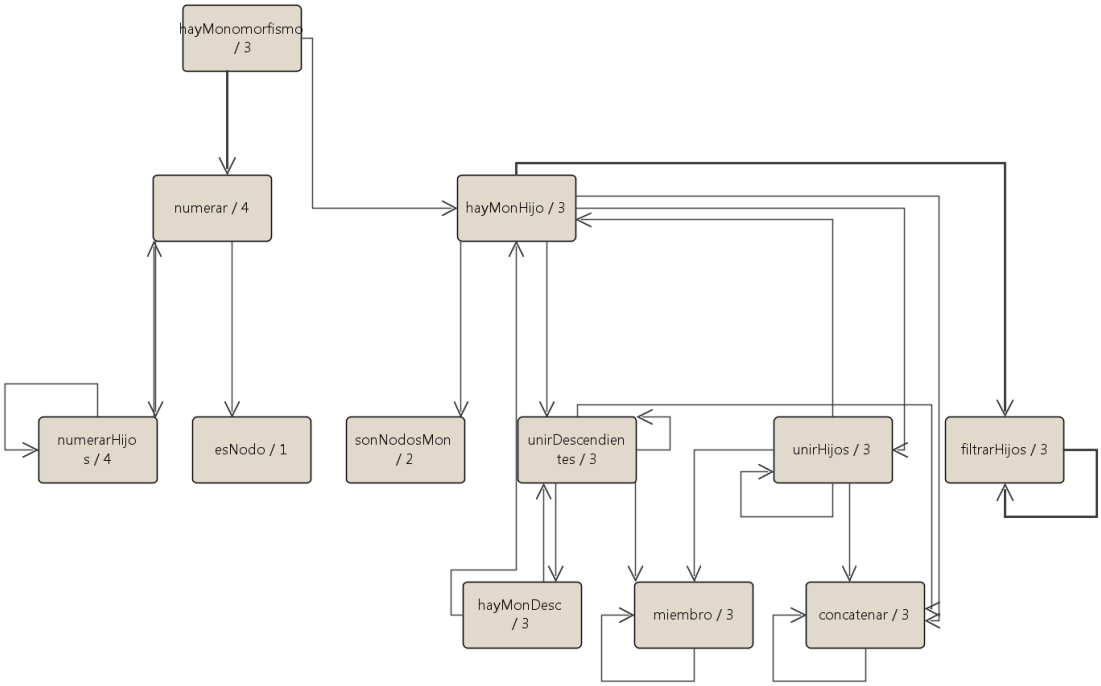

Figura 4.18: Interacción de predicados de *Monomorfismo*

- *operarInicio*: fusiona las raíces de los dos patrones de entrada formando un único nodo raíz cuyos hijos son los hijos de ambas raíces y llama al método *operarCuerpo*.

- *operarCuerpo*: este predicado es el cuerpo de la iteración. Va obteniendo nuevos patrones hasta que no se genere ninguno más y entonces finaliza.

- *generarPatronMismoPadre*: fusiona dos nodos que tienen al mismo padre. El modo de actuar en este caso es unificar sus hijos y colocar el arco más restrictivo[2] como relación entre el padre y el nodo fusionado.

- *generarPatronDiferentePadre*: fusiona dos nodos que no tienen el mismo padre. El modo de actuar depende de los arcos que tenga cada nodo. Si las dos son de *hijo directo*, no tiene solución. Si una es del tipo *hijo directo* y la otra *descendiente*, se fusionan colocando como padre al padre del *hijo directo*, y si las dos son del tipo *descendiente*, se forman dos posibles soluciones: una en la que el padre del nodo fusionado es el padre de uno de los nodos, y otra en la que el padre es el padre del otro nodo.

En la figura 4.19 se muestra el diagrama de interacción de los predicados de este caso de uso.

Operar Con Común

El predicado principal de este caso de uso es *operarConComun/4*. Siendo los constraints que se van a operar $\forall(c : p_2 \rightarrow q)$ y $\exists p_1$, el procedimiento recibe como primer parámetro el patrón p_2, como segundo parámetro el patrón q, como tercero un monomorfismo m de p_2 en p_1 no extensible a $q \rightarrow p_1$, y devuelve en el cuarto parámetro el resultado. El total de predicados con los que cuenta el caso de uso es 78, siendo los más relevantes:

- *operarConComun*: es el método principal, que recibe un constraint condicional $\forall(c : p_2 \rightarrow q)$ y un monomorfismo $m : p_2 \rightarrow p_1$. El monomorfismo se representa mediante el patrón p_1 añadiendo a sus nodos la información de con qué nodos de p_2 se han unificado. Este método obtiene el primer elemento del conjunto $p_1 \otimes_{c,m} q$ y llama al método *operarCuerpo* (del caso de uso *Operar*) para calcular el resto de elementos de dicho conjunto.

- *colgarRestos*: este método es el encargado de colocar los nodos de q que no aparecen en p_2 como hijos de los nodos correspondientes en el monomorfismo de sus nodos padres.

[2] el arco / es más restrictivo que el arco //

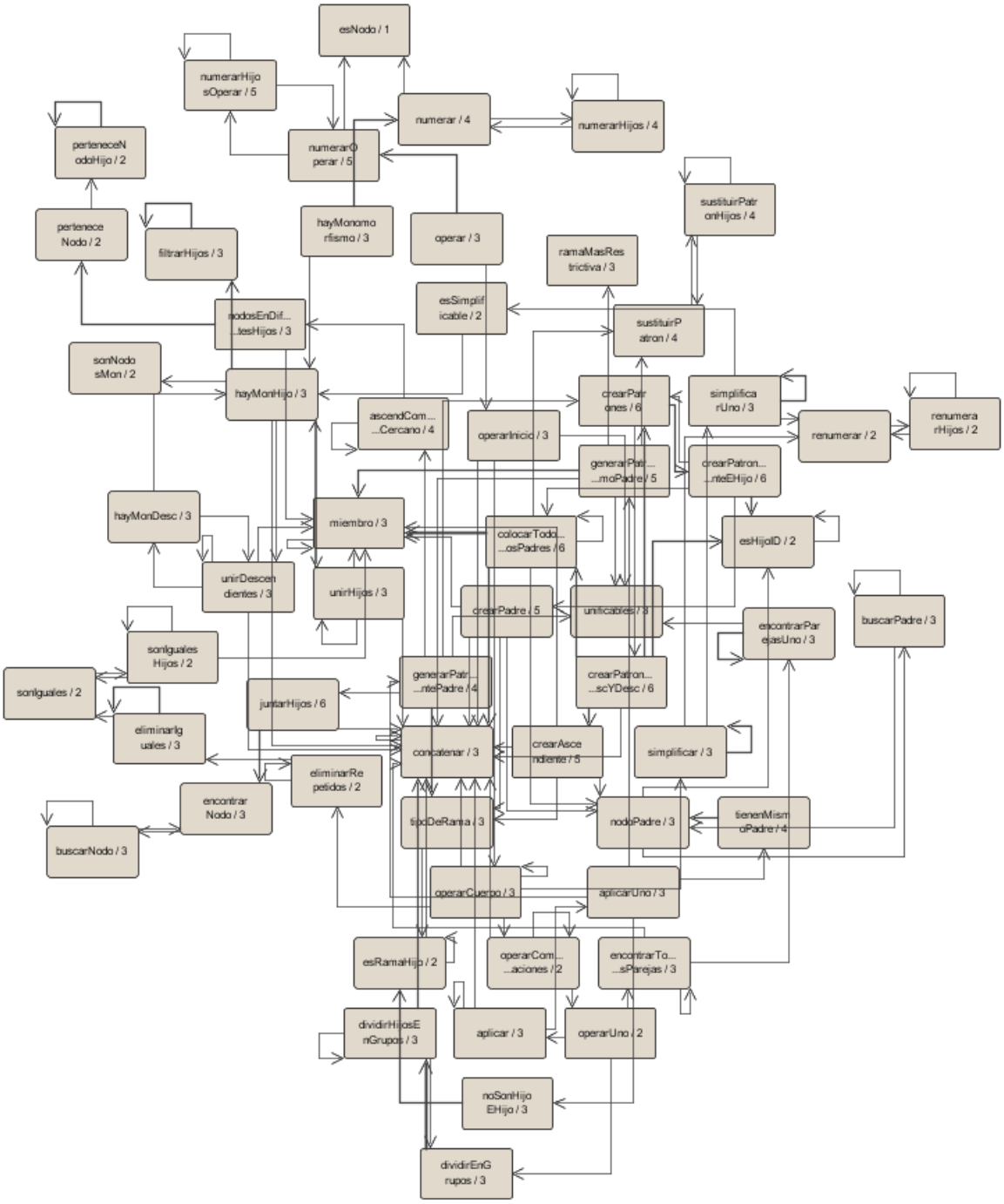

Figura 4.19: Interacción de predicados de *Operar*

En la figura 4.20 se muestra el diagrama de interacción de los predicados de este caso de uso.

Consultar Cláusulas

Este caso de uso emplea un único predicado estático (*constraint/2*)y dos dinámicos (*clausula/2* y *cons/2*). Durante el proceso de ejecución del procedimiento de refutación se habrán guardado cláusulas dinámicamente. El predicado dinámico *clausula/2* se puede usar de dos maneras:

1. Pasándole como primer parámetro el identificador de una cláusula. De ese modo devuelve en el segundo parámetro una lista con los identificadores de los constraints que forman la cláusula. Una vez obtenidos, llama al método *constraint* para obtener el constraint correspondiente a cada identificador.

2. Pasándole como primer parámetro una variable libre. Así, el predicado devolvería todas las cláusulas existentes junto con los identificadores de los constraints que las componen.

En la figura 4.21 se muestra el diagrama de interacción de los predicados de este caso de uso.

Consultar Constraints

Este último caso de uso es el más simple de todos, utilizando un único predicado estático y uno dinámico. El principal es *constraint/2*, al que se le pasa en el primer parámetro el identificador de un constraint y devuelve en el segundo parámetro el constraint. Este predicado utiliza el predicado dinámico *cons/2*, donde se han guardado los diferentes constraints a lo largo de la ejecución del procedimiento de refutación.

En la figura 4.22 se muestra el diagrama de interacción de los predicados de este caso de uso.

4.2.4. Capa de Datos

Esta capa es la encargada de guardar especificaciones de la capa de interfaz en ficheros y viceversa. Está compuesta por una única clase, *CapaDatos*, y su diagrama se puede ver en la figura 4.23.

Los atributos de esta clase son:

- **br**: variable de tipo *BufferedReader* que se utiliza para leer ficheros.

- **bw**: variable de tipo *BufferedWriter* que se utiliza para escribir en ficheros.

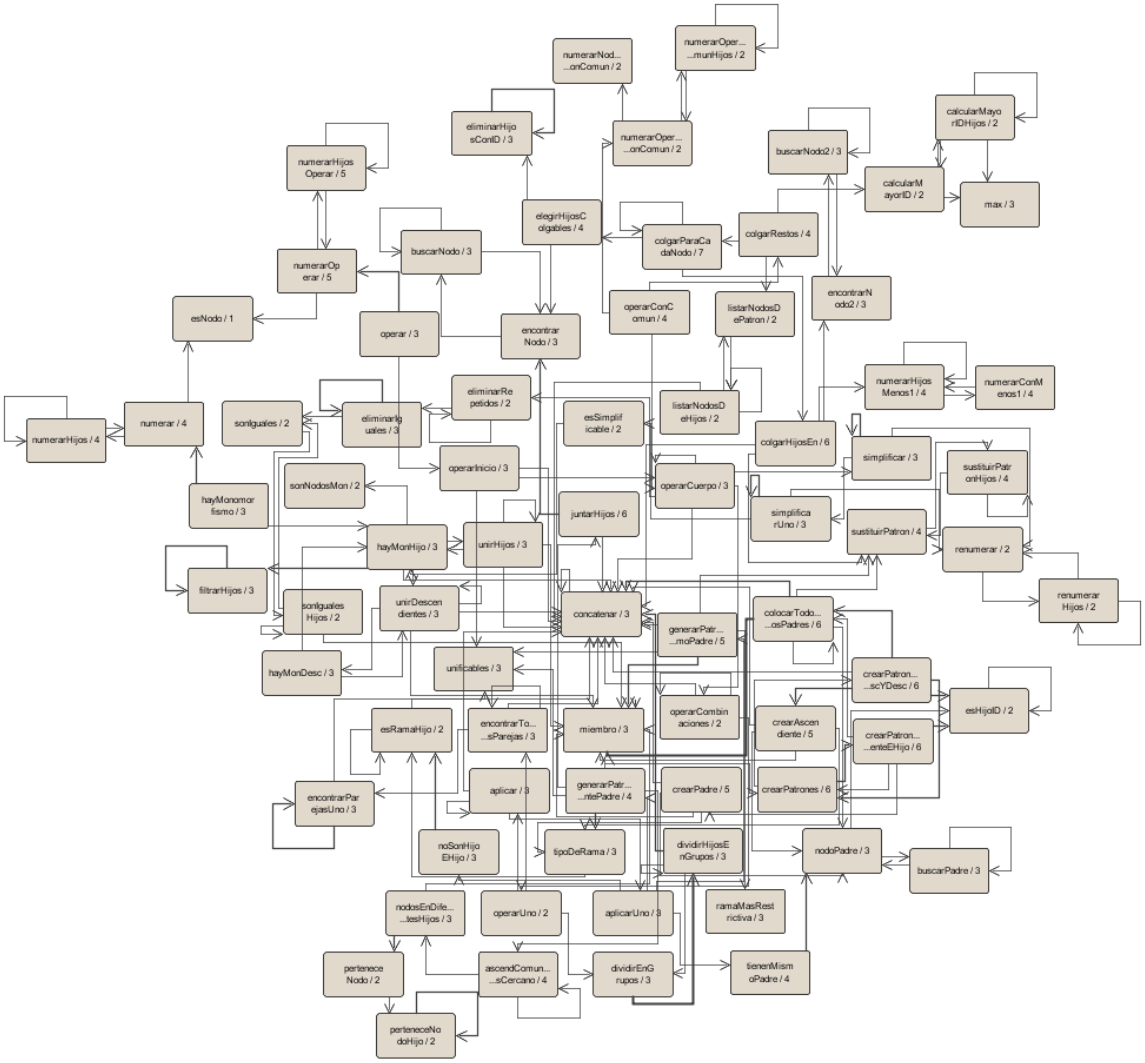

Figura 4.20: Interacción de predicados de *Operar Con Común*

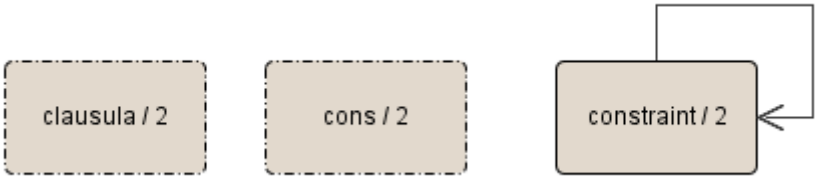

Figura 4.21: Interacción de predicados de *Consultar Cláusulas*

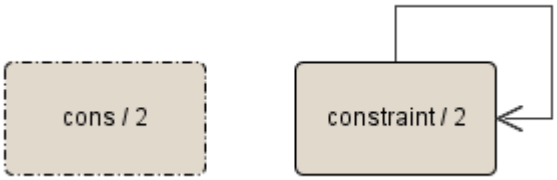

Figura 4.22: Interacción de predicados de *Consultar Constraints*

- **contadorC**: variable que se utiliza para saber, al cargar una especificación desde fichero, el número de cláusulas que hay y poder saber qué identificador asignarle a una nueva cláusula.
- **contadorCT**: al igual que **contadorC** pero con constraints, se utiliza para conocer el siguiente identificador de constraint a asignar.
- **l**: variable que almacena el conjunto de cláusulas cargadas desde fichero.

Los métodos más importantes son:

- *getInstance*(): al igual que la clase *Aplicacion*, sólo puede haber una única instancia de esta clase. De ese modo, este método devuelve la única instancia que existe de esta clase.
- *leerFichero(nombre)*: carga un conjunto de cláusulas desde el fichero *nombre*.
- *escribirFichero(nombre,contadorC,contadorCT,cc)*: guarda una especificación dada en el fichero *nombre*.
- *escribirHistorial(nombre,s)*: guarda en el fichero *nombre* el historial s.
- *existe(nombre)*: devuelve *true* si el fichero *nombre* existe y *false* en caso contrario.

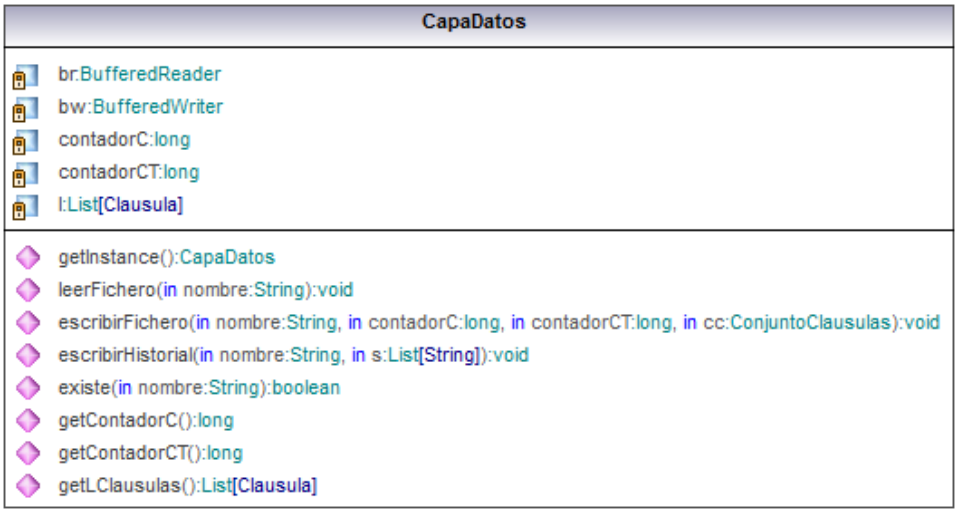

Figura 4.23: Diagrama de clases de la Capa de Datos

Capítulo 5

Aplicación: Implementación

En este capítulo se explicará cómo se han implementado en Prolog las diferentes estructuras de datos y operaciones de las reglas R1, R2 y R3, indicando en el caso de estas últimas un ejemplo en el que se pueda comprobar cómo funciona el algoritmo diseñado para las mismas. También se detalla cómo se han implementado las reglas de Subsunción, Simplificación y Unfold. Por último, se indica el algoritmo general del procedimiento de refutación. La implementación completa consta de alrededor de 1300 líneas de Prolog y unas 4000 líneas de Java.

5.1. Patrones

Si queremos utilizar patrones en Prolog, debemos poder tener información tanto de la etiqueta de los nodos como de las relaciones entre ellos. Por ello, la manera en la que se han representado los patrones ha sido:

$$p(Nodo, Hijos)$$

Donde *Nodo* contiene la etiqueta de la raíz del patrón e *Hijos* contiene información sobre los sub-patrones hijos de la raíz. La sintaxis completa se puede obtener mediante la siguiente gramática:

Patrón → **Nodo**
| **p (Nodo ,** [Hijos] **)**
Hijos → **h (** Rama **,** Patrón **)** RestoHijos
| ε
RestoHijos → **,** **h (** Rama **,** Patrón **)** RestoHijos
| ε
Rama → **'/'** | **'//'**

Nodo es del tipo de datos *atomic*, es decir, cualquier combinación de letras o números o cualquier combinación de caracteres colocados entre co-

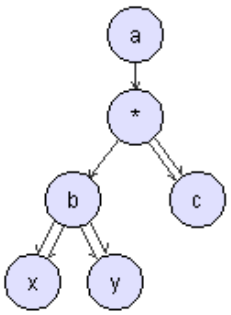

Figura 5.1: Ejemplo de patrón

millas simples. Por tanto, si queremos escribir el patrón de la figura 5.1, se podría escribir:

$$p(a, [h(/, p(*, [h(/, p(b, [h(//, x), h(//, y)])), h(//, c)]))])$$

Aunque los nodos hoja también se pueden escribir como $p(Nodo, [])$.

A partir de aquí distinguiremos dos tipos de patrón: el patrón *normal* y el patrón *puro*. Consideraremos el normal a cualquier forma de expresar un patrón mediante la gramática arriba indicada. El puro, por otro lado, es igual que el normal, sólo que los nodos hoja deben de ser siempre de la forma $p(Nodo, [])$, es decir, la transición $Patrón \rightarrow Nodo$ en la gramática no se admite.

El patrón de la figura 5.1 en modo *puro* se escribiría como:

$$p(a, [h(/, p(*, [h(/, p(b, [h(//, \mathbf{p(x,[])}), h(//, \mathbf{p(y,[])})])), h(//, \mathbf{p(c,[])})])])$$

La diferencia entre patrón normal y patrón puro es que todos los métodos funcionan bien con patrones puros, mientras que hay alguno que no funciona del todo utilizando patrones normales. Por ello, cuando se indica que se admite patrón normal, quiere decir que se admite tanto normal como puro, pero cuando se indica que se admite patrón puro, sólo se admite el patrón puro.

Existe un tercer tipo de patrón, llamado *numerado*. La idea de este patrón es guardar más información aparte de las meras etiquetas en los nodos. Por ello, en el lugar donde está el nodo se colocará una lista de elementos, entre las cuales estará la etiqueta. Respecto al resto de información a guardar, ésta consiste en un identificador único para cada nodo[1] y un segundo identificador cuyo uso depende de la operación que se esté realizando. Por lo tanto, el patrón *numerado* se escribe:

$$p([Nodo, Id, Id2], Hijos)$$

[1]El identificador es único dentro del patrón. Puede haber varios nodos de diferentes patrones con el mismo identificador.

En el apartado 5.4.2 se explicará la razón por la que se necesitó crear el tipo de patrón *numerado*.

5.2. Constraints

La idea para expresar un constraint es parecido al patrón. Tendremos una estructura cuyo funtor es *ct*. Los datos que contiene son, por un lado, la información sobre el constraint, es decir, si es de tipo $\exists$, $\neg\exists$ o $\forall$ junto con el patrón o los patrones necesarios; y por otro lado un identificador único de constraint.

El identificador se necesita para poder distinguir rápidamente un constraint de cualquier otro. Esto se usa tanto para buscar constraints concretos tras la finalización del procedimiento de refutación (caso de uso *Consultar constraints*), como en la ejecución de las reglas R1, R2 y R3, donde ellas reciben como datos dos cláusulas y dos identificadores de constraints, y tienen que ejecutar la regla sobre esos constraints. Si no se usaran identificadores, el modo de buscar un constraint podría ser más complicado e ineficiente, por lo que añadiendo un simple identificador simplifica las cosas.

La sintaxis de un constraint es, por tanto:

$$ct(Identificador, Elemento)$$

Donde *Identificador* se obtiene dinámicamente llamando al predicado *gensym(ct,Identificador)* y *Elemento* es una de las siguientes tres opciones:

- e(P)
- ne(P)
- v(X,Q)

La expresión *e(P)* representa $\exists p$, donde p es un patrón normal; *ne(P)* representa $\neg\exists p$, donde p también es un patrón normal. Por último, *v(X,Q)* representa $\forall(c : x \rightarrow q)$. Para poder expresar la función c (relación de pre-árbol entre x y q), en vez de utilizar patrones *normales* se utilizan patrones *numerados* (en el apartado 5.4.2 se explicarán en detalle). Puesto que x es pre-árbol de q, se tiene que cumplir que los nodos que tengan en común estén numerados con el mismo identificador Id. Respecto al Id2 que tienen todos los nodos numerados, puesto que no se va a utilizar se coloca un cero.

En las figuras 5.2, 5.3 y 5.4 se pueden ver un ejemplo para cada uno de los tres constraints.

5.3. Cláusulas

De nuevo, las cláusulas se expresan mediante una estructura cuyo funtor es *c*. Guardaremos, como con los constraints, un identificador para que

$$\exists \begin{pmatrix} a \\ | \\ b \end{pmatrix} \text{ es } ct\Big(ct3, e\big(p(a,[h(/,b)])\big)\Big)$$

Figura 5.2: Ejemplo de constraint positivo en Prolog

$$\neg\exists(c) \text{ es } ct\Big(ct1, ne\big(p(c)\big)\Big)$$

Figura 5.3: Ejemplo de constraint negativo en Prolog

$$\forall \begin{pmatrix} a & & a \\ c\colon | & \rightarrow & / \ \backslash\backslash \\ b & & b \quad c \end{pmatrix} \text{ es } ct(ct6, v(p([a,1,0],[h(/,p([b,2,0],[])]),\\ p([a,1,0],[h(/,p([b,2,0],[])),h(//,p([c,3,0],[]))])))$$

Figura 5.4: Ejemplo de constraint condicional en Prolog

podamos referirnos a una cláusula concreta sin que pueda dar lugar a ambigüedades.

La sintaxis de una cláusula es:

$$c(Identificador, ListaConstraints)$$

Donde $Identificador$ se obtiene dinámicamente llamando al predicado *gensym(c,Identificador)* y $ListaConstraints$ es una lista no vacía de constraints.

Como ejemplo, supongamos que tenemos la cláusula de la figura 5.5. Ésta se escribiría en Prolog:

$$C1 = c(c1, [ct(ct1, e(p)), ct(ct2, ne(p(q, [h(/, r)])))])$$

$$\mathsf{C1} = \exists \mathsf{p} \vee \neg\exists \begin{pmatrix} q \\ | \\ r \end{pmatrix}$$

Figura 5.5: Ejemplo de cláusula

5.4. Monomorfismo

En este apartado se explicará el algoritmo de la operación *monomorfismo* y su implementación, además de un ejemplo completo.

5.4.1. Algoritmo

Explicamos primero la idea de cómo hallar un monomorfismo de un patrón p en otro patrón q.

El diseño debe satisfacer eficientemente las cuatro condiciones que deben cumplirse en el monomorfismo: preservación de las raíces, preservación de las etiquetas, preservación de los caminos *hijo* y preservación de los caminos *descendiente*. Lo primero que hay que comprobar es que **la raíz del patrón p pueda unirse con la raíz del patrón q**. Para ello nos fijamos en si ambos nodos tienen la misma etiqueta o si al menos la etiqueta de la raíz de p es un asterisco.

Si se cumple lo anterior, entonces hay que **unir**[2] **todos los hijos(tanto *hijos directos* como *descendientes*) de la raíz de p con nodos de q**. Para ello tenemos en cuenta la tercera condición, la de preservación de los caminos *hijo*. Según esto, si un nodo de p tiene un *hijo directo* h, entonces su nodo imagen en q también tiene un *hijo directo* que además es la imagen de h. Por lo tanto, si la raíz de p tiene *hijos directos* y *descendientes*, **debemos encontrar los nodos imagen de los *directos* antes que los de los *descendientes***, y además las imágenes de los *hijos directos* han de ser también *hijos directos* de la raíz de q.

Una vez unidos todos los *hijos directos* de p con los de q, hay que unir los hijos *descendientes* de p con los restantes de q. **Los nodos de q que ya se hayan unido con algún *hijo directo* de p no se tendrán en cuenta**. El problema de los descendientes es que sus imágenes podrían estar en cualquier parte del patrón q, ya sea en los hijos de la propia raíz o en nodos descendientes de los hijos.

Para hacer este último paso **se selecciona uno de los nodos descendientes de p y se busca un nodo en q que tenga la misma etiqueta que él** y que no se haya unido antes con nadie (en el caso de que el nodo de p sea un asterisco, hay que mirar con cada nodo de q). Una vez encontrado, **los unimos como lo hemos hecho con los *hijos directos***: obteniendo el monomorfismo de los correspondientes sub-árboles. En caso de no haberse podido unir, se busca otro nodo diferente y se vuelve a intentar. Si se ha podido unir, **se repite este paso para todos los nodos descendientes de p restantes**.

La especificación del algoritmo obtenido es el siguiente:

Entrada: dos patrones, p y q.
Salida: una lista de monomorfismos, L.
Postcondición: cada elemento de L representa un monomorfismo de p en q diferente. La forma de representación de cada elemento de L es el

[2]Nota: al hablar de unir dos nodos, nos referimos al hecho de unir dos nodos y también sus hijos, es decir, como si halláramos el monomorfismo entre dos (sub)árboles cuyas raíces son los dos nodos a unir.

propio patrón q cuyos nodos pueden estar marcados o no. Si están marcados, contienen la información del nodo de p con el que se han unido. Si no están marcados, significa que ningún nodo de p se ha unido a ellos. Si L es vacía, significa que no existe ningún monomorfismo de p en q.

5.4.2. Implementación

A la hora de implementar el monomorfismo en Prolog, lo primero es determinar la forma de representar un monomorfismo. Al igual que en el algoritmo, **se empleará el propio patrón q con marcas en los nodos**.

Como ya hemos visto, un patrón en Prolog se escribe de la siguiente manera:

$$p(Nodo, Hijos)$$

La idea es que en el lugar de *Nodo* haya más información acerca de dicho nodo aparte de su etiqueta. Pero, ¿qué información necesitamos? Es necesario saber con qué nodo de p se ha unido, por lo que podemos tener una lista de dos elementos en los que el primero es la etiqueta del propio nodo y el segundo la etiqueta del nodo de p con el que se ha unido:

$$p([Nodo, NodoP], Hijos)$$

Sin embargo, esto presenta dos problemas: el primero, ¿qué colocamos en el lugar de *NodoP* cuando es un nodo que no se ha unido con ningún otro? Y el segundo, si en p hubiera dos nodos con la misma etiqueta, ¿cómo los diferenciamos? Visualmente se puede ver este problema en la figura 5.6.

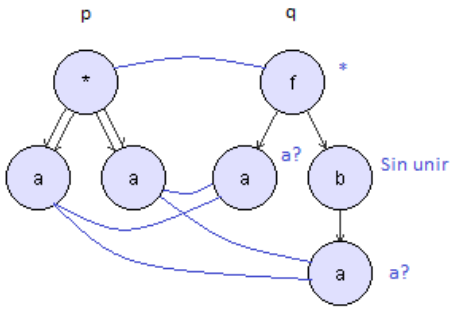

Figura 5.6: Problema al no poder distinguir los nodos en el monomorfismo

La solución es que en vez de guardar la etiqueta del nodo, la cual puede estar repetida, se guarde un identificador que haga referencia a un único nodo. Para ello necesitamos que los nodos estén previamente identificados. Finalmente, la información a guardar por un nodo será su etiqueta, su identificador único dentro del patrón, y por último el identificador del nodo del

otro patrón con el que se ha unido. Si utilizamos números como identificadores y suponemos que el mínimo identificador será el 1, podemos utilizar el número 0 para expresar que un nodo no se ha unido con nadie.

El nuevo patrón quedará así:

$$p([Nodo, Id, Id2], Hijos)$$

Con estas nuevas formas de expresar patrones surge un nuevo problema, y es que antes *Nodo* era una cadena de caracteres y ahora es una lista de tres elementos. Esa es la razón por la que hay que crear el tipo de patrón *numerado*. En este caso, el patrón **numerado para monomorfismo**.

Puesto que el procedimiento de refutación tratará con patrones no numerados, para poder ejecutar el método de monomorfismo habrá que traducir de patrón no numerado a numerado. Una vez hecho eso ya se puede seguir con el algoritmo.

5.4.3. Ejemplo

Sean p y q respectivamente los patrones de la figura 5.7, lo primero que hay que hacer es numerarlos con la notación de monomorfismo, mostrados en la figura 5.8.

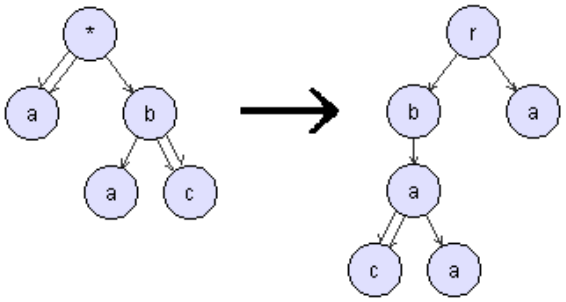

Figura 5.7: Ejemplo monomorfismo: p y q

Lo siguiente es comprobar que las raíces se puedan unir. Puesto que la raíz de p es un asterisco, sí que se puede. Por lo tanto, el segundo identificador de la raíz de q pasa a tener el identificador de la raíz de p, tal y como se muestra en la figura 5.9.

Ahora se dividen los hijos de la raíz de p en dos grupos: los *hijos directos* y los *descendientes*. En los hijos directos sólo hay un nodo, el [b,3,0]. Hay que comprobar si hay algún monomorfismo de ese nodo a alguno de los hijos directos de la raíz de q. Hay dos posibilidades, que se muestran en la figura 5.10. El caso 2 no tiene solución, pero el caso 1 sí. Las raíces se pueden unir, por lo que el nodo [b,2,0] del patrón q pasará a ser [b,2,3]. De nuevo, hay que dividir los hijos del nodo [b,3,0] del patrón p en *hijos directos* y *descendientes*. Su único hijo directo es [a,4,0] y hay que comprobar si se

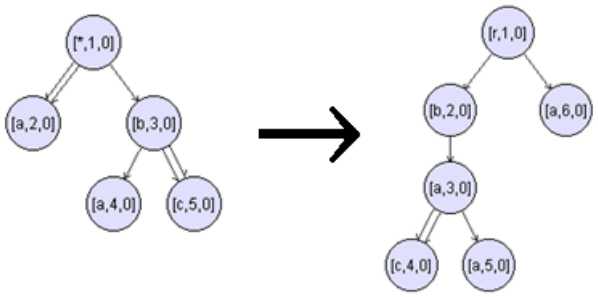

Figura 5.8: Ejemplo monomorfismo: p y q numerados

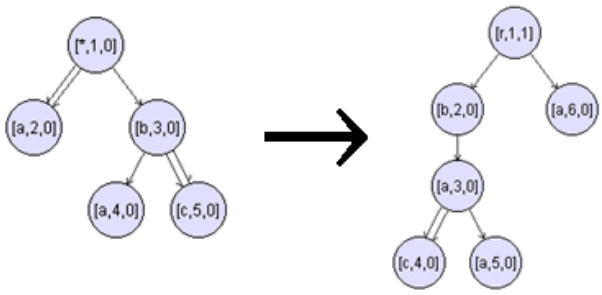

Figura 5.9: Ejemplo monomorfismo: raíces unidas

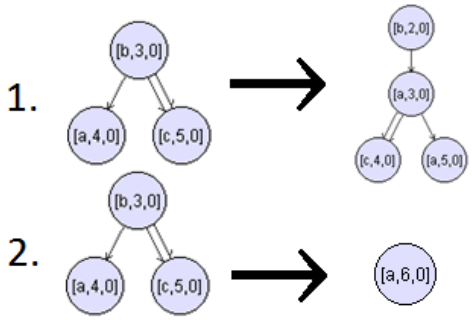

Figura 5.10: Ejemplo monomorfismo: monomorfismos posibles

puede unir con el hijo de [b,2,3]. Las raíces se pueden unir pero [a,4,0] ya no tiene hijos por lo que los unimos y volveríamos atrás. El patrón actual resultante se muestra en la figura 5.11.

Ahora que se han tratado los hijos directos de [b,3,0], sólo queda unir el descendiente [c,5,0] con algún nodo no marcado del patrón de la derecha. El único nodo con el que se puede unir es [c,4,0]. Como [c,5,0] no tiene hijos, el monomorfismo se puede realizar y por tanto el nodo de q quedaría [c,4,5]. Tras finalizar con estos patrones hay que volver atrás. El monomorfismo

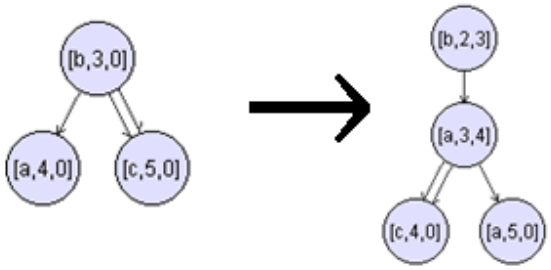

Figura 5.11: Ejemplo monomorfismo: paso intermedio

ahora mismo se encuentra en el estado de la figura 5.12.

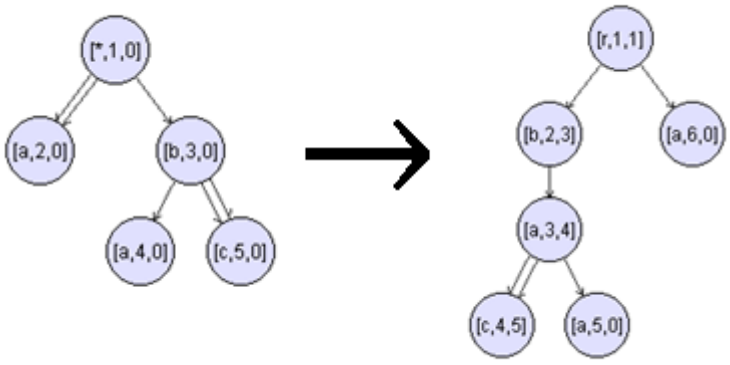

Figura 5.12: Ejemplo monomorfismo: hijos directos de p unidos

Por último, ya sólo queda unir el descendiente de p [a,2,0] con algún nodo libre de q. Hay dos posibilidades, con [a,5,0] y con [a,6,0]. Las dos son posibles, por lo que habrá dos resultados finales, mostrados en la figura 5.13. Gracias a la aplicación implementada, podemos ver las soluciones representadas como en la figura 5.14.

5.5. Operar

En este apartado explicaremos el algoritmo de la operación $p_1 \otimes p_2$, así como su implementación y un ejemplo.

5.5.1. Algoritmo

Supondremos que se trata de la operación $p_1 \otimes p_2$. También trataremos con el concepto de *fusionar* dos nodos, que consiste en que desaparecen

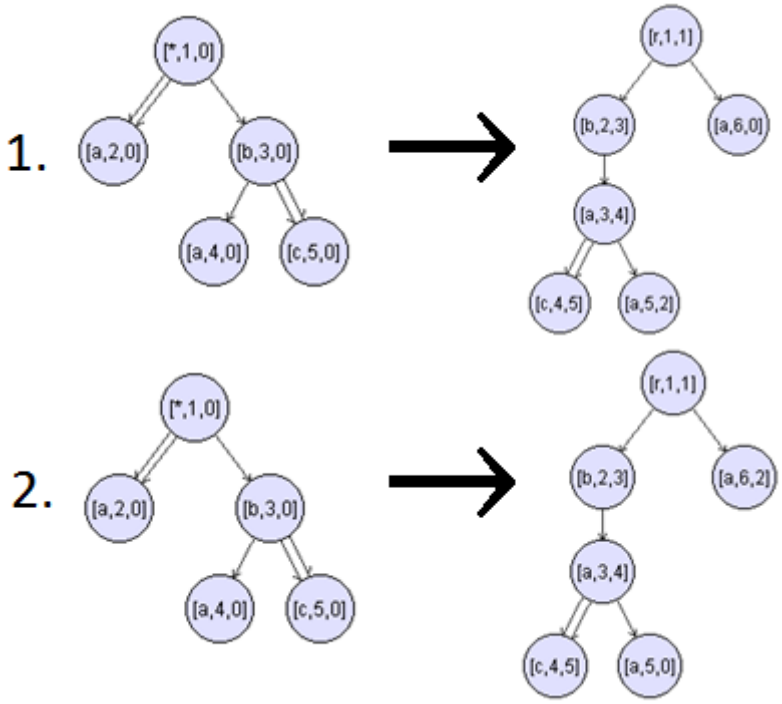

Figura 5.13: Ejemplo monomorfismo: solución

esos dos nodos como tales y los convertimos en un mismo nodo, por lo que **todos los hijos que tengan cada uno de los dos nodos ahora los tendrá el *fusionado*** y, respecto a sus padres, **el camino hasta la raíz estará compuesto por una cadena de los diferentes nodos padre** de los dos nodos originales. Más adelante se explica mejor este proceso.

Podemos empezar indicando que, si las raíces de p_1 y de p_2 no se pueden fusionar, entonces devolvemos un conjunto vacío. Si, por el contrario, sí se pueden fusionar, los fusionamos y damos comienzo al algoritmo.

La idea es que partimos de un patrón con **tres tipos de nodos: los nodos de p_1, los nodos de p_2 y los nodos fusionados**, los cuales no se pueden fusionar ya con ningún otro. El algoritmo empieza con un patrón cuya raíz es un nodo fusionado que contiene sub-árboles que son, o bien de p_1, o bien de p_2. Trataremos de ir fusionando los nodos de p_1 con los de p_2 hasta que no se puedan fusionar más. Hay que tener en cuenta que no se pueden fusionar dos nodos que provienen del mismo patrón (ya sean de p_1 o de p_2) y que, una vez fusionados, ni ellos ni sus ascendientes podrán fusionarse con nadie más.

El algoritmo consistirá en ir obteniendo nuevos patrones por niveles. El primer nivel consta de un único patrón (el que consiste en la fusión de las raíces de p_1 y de p_2). Para obtener el segundo nivel, **busca-**

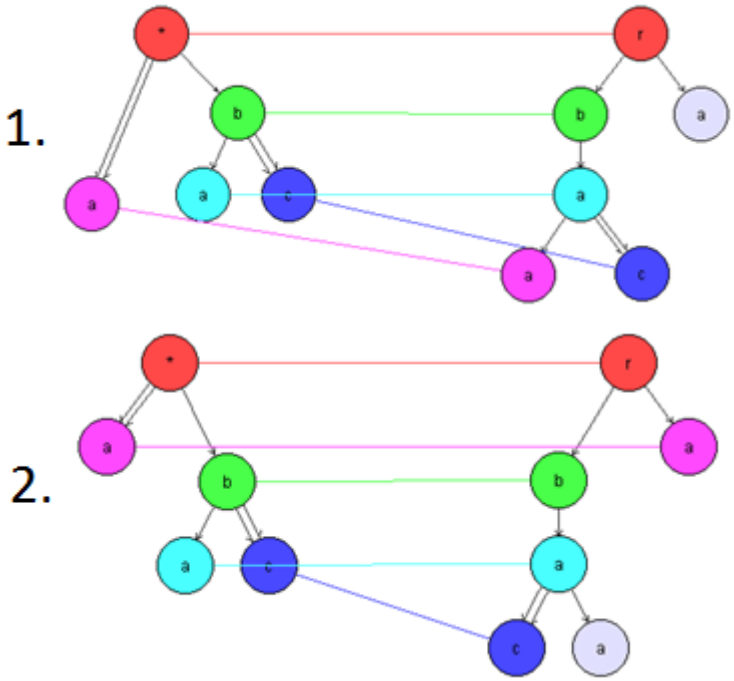

Figura 5.14: Ejemplo monomorfismo: solución vista con la aplicación

remos todas las posibles parejas de nodos de p_1 y de p_2 que se puedan fusionar y obtendremos tantos patrones como parejas hayamos hecho. (Es posible que dos nodos se *puedan* fusionar pero que al intentar unificar su camino hasta la raíz esto no sea posible y por tanto ese patrón no se obtenga). Una vez obtenido el segundo nivel, calcularemos el tercero haciendo lo mismo que hemos hecho para el segundo, pero para cada uno de los patrones que componen el segundo nivel.

Hallado ya el tercer nivel, los anteriores ya no los necesitaremos para obtener los siguientes, por lo que podemos simplificarlos. El concepto de *simplificar* es el hecho de **eliminar los casos particulares de los casos generales**. Además, también hay que considerar un caso más: es posible que se llegue a un mismo patrón mediante dos o más caminos diferentes. En ese caso, **deberíamos eliminar los elementos repetidos para no obtener soluciones redundantes**. Podemos entenderlo mejor en el siguiente ejemplo[3]:

Partiendo del nivel de la figura 5.15, de entre todas las diferentes parejas que se pueden hacer, vamos a fijarnos en las etiquetas b y f. Si primero fusionamos b y luego fusionamos f, nos quedará como en el de la figura 5.16, y,

[3]Nota: los nodos de color azul representan a los nodos de p_1 y los nodos de color rojo a los de p_2. Los grises son los fusionados.

si por el contrario fusionáramos primero f y luego b, nos quedaría como en el de la figura 5.17. Podemos observar que en ambos casos obtenemos el mismo patrón, por lo que hay que considerar la posibilidad de que obtengamos repetidos.

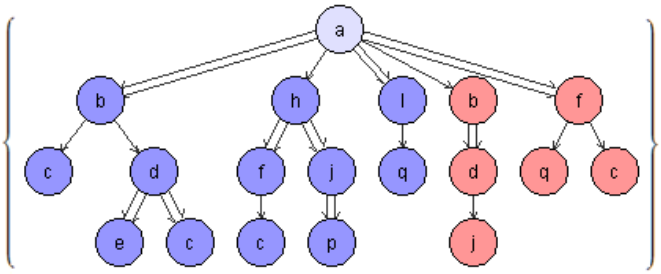

Figura 5.15: Ejemplo: nivel inicial

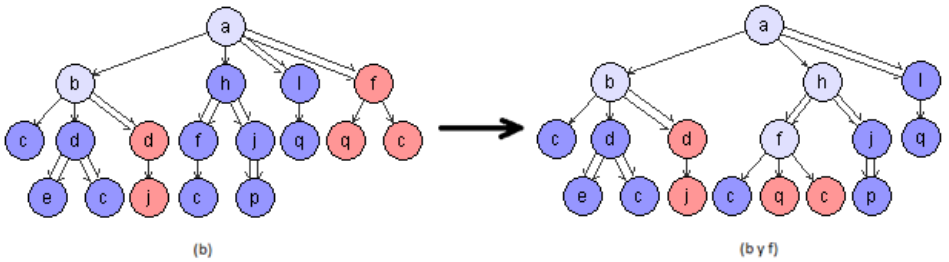

Figura 5.16: Ejemplo: unificar b y luego f

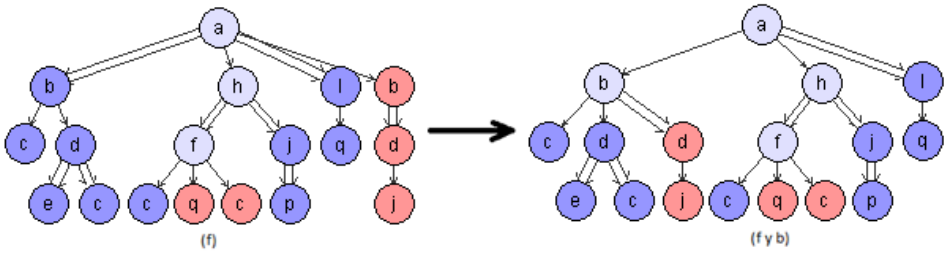

Figura 5.17: Ejemplo: unificar f y luego b

Por tanto, el **esquema general del algoritmo** sería algo así:

```
if sePuedenFusionar(raíz(p1),raíz(p2)) then
    aux := fusionar(raíz(p1),raíz(p2));
    nivelesAnteriores := listaVacía();
    nivelActual := nuevaLista(aux);
    while not esVacía(nivelActual) do
        nivelesAnteriores := concatenar(nivelesAnteriores,nivelActual);
```

```
        nivelActual := calcularNuevoNivel(nivelActual);
        nivelesAnteriores := simplificar(nivelesAnteriores);
        nivelActual := eliminarRepetidos(nivelActual);
    end while
    return nivelesAnteriores;
else
    return listaVacía();
end if
```

Ahora vamos a centrarnos en la creación del nuevo nivel. Partimos de una lista de patrones (el nivel actual). El proceso es: para cada patrón del nivel actual **calculamos todas las parejas de nodos posibles** (recordemos que las parejas de nodos están compuestas por un nodo de p_1 y otro de p_2 que podrían fusionarse, es decir, que tienen la misma etiqueta o que uno de ellos es un asterisco) **e intentamos fusionar cada una de ellas**. Se creará un nuevo patrón por cada pareja satisfactoriamente fusionada. Todos los patrones nuevos obtenidos formarán el nuevo nivel.

Por último, sólo queda analizar la manera de fusionar los nodos y obtener el nuevo arco. A la hora de fusionar los nodos hay que fijarse en cuál será la etiqueta que tendrá el nodo *fusionado*. Si ambos nodos tienen la misma etiqueta, se colocará ésa. Si no, si uno de ellos es un asterisco, entonces se pondrá la etiqueta que no sea asterisco. Una vez colocada la etiqueta, se colocan como hijos del nuevo nodo tanto los hijos del nodo de p_1 como los hijos del nodo de p_2.

El camino a la raíz es más complicado. Todos los nodos deben cumplir las relaciones hijo/descendiente que tuvieran en el patrón original, por lo que **si alguno de los nodos a fusionar tiene una relación de *hijo directo* con su padre, ese padre debe colocarse obligatoriamente** como padre del nuevo nodo. Podemos deducir que, si ambos nodos a fusionar son hijos directos, entonces no es posible colocar un padre al nuevo nodo, ya que deberíamos colocar a ambos padres y sólo puede serlo uno. Hay una excepción y es que los dos nodos a fusionar sean dos hijos directos del mismo padre. En ese caso sí se puede.

Por tanto, tenemos tres posibilidades: que los dos nodos sean hijos directos, que uno sea hijo directo y otro descendiente, y que los dos sean descendientes:

- Los dos hijos directos: Como ya hemos dicho, salvo que sean hijos del mismo padre, no tiene solución. Si son del mismo padre, se coloca dicho padre y se termina (figura 5.18).

- Uno es hijo directo y otro descendiente: Si tienen el mismo padre, se coloca dicho padre y se termina (figura 5.19). Si no, si el descendiente es hijo del *ascendiente común más cercano entre los dos*, se colocan todos los ascendientes del hijo directo hasta llegar al ascendiente común y

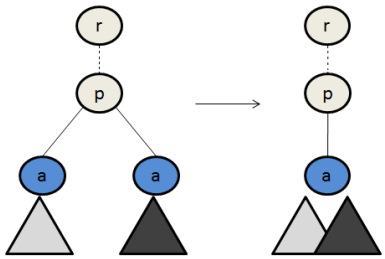

Figura 5.18: Fusión de nodos *hijo directo*

se termina (figura 5.20). Si no, si el hijo directo tampoco es hijo del ascendiente común más cercano entre los dos, se coloca como padre al padre del hijo directo y se analizan los padres del nodo descendiente y del nodo padre colocado (figura 5.21). Este caso puede fallar si el hijo directo es hijo del ascendiente común y el descendiente no lo es (figura 5.22).

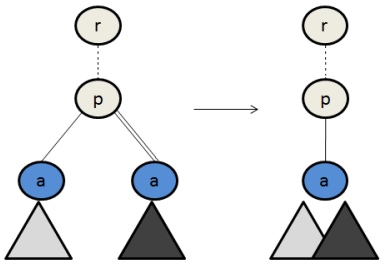

Figura 5.19: Fusión de nodos *hijo directo* y *descendiente* con mismo padre

- Los dos son descendientes: Si tienen el mismo padre, se coloca dicho padre y se termina (como en la figura 5.19 pero teniendo ambos arco descendiente). Si no, si uno de ellos es hijo del *ascendiente común más cercano entre los dos*, se colocan todos los ascendientes del otro nodo hasta llegar al ascendiente común y se termina (como en la figura 5.20 pero teniendo ambos arco descendiente). Si no, se crean dos patrones: en el primero se coloca como padre al padre del nodo de p_1 y en el segundo al padre del nodo de p_2 (figura 5.23). Se realiza una nueva iteración para cada uno de ellos por separado y se devuelven las soluciones satisfactorias obtenidas por ambos caminos.

Una vez ha terminado el punto anterior, es decir, una vez se han fusionado los nodos hasta llegar al ascendiente común más cercano entre ellos, el resto del patrón no varía. Sin embargo, para evitar trabajo innecesario **vamos a colocar a todos los ascendientes** (hasta llegar a la raíz) **el estado de *nodo fusionado***. El motivo de hacer esto se basa en que, como hemos visto antes, si tenemos para fusionar los nodos a y los nodos b,

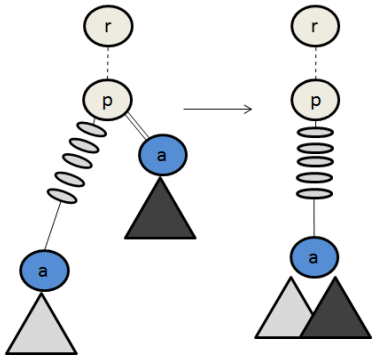

Figura 5.20: Fusión de nodos / y // siendo éste ultimo hijo del ascendiente común mas cercano

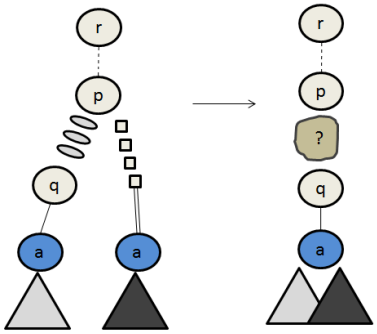

Figura 5.21: Fusión de nodos / y // con diferente padre y ninguno hijo del ascendiente común más cercano

no importa el orden en el que lo hagamos que cuando se hayan fusionado tanto los a como los b obtendremos el mismo patrón. Por eso, al fusionar los ascendientes es cierto que estamos quitando posibles soluciones al patrón, pero sabemos que si el orden se hubiera hecho al revés, entonces se habrían hallado. Puesto que nuestro algoritmo calcula todas las opciones, sabemos que esa solución la vamos a encontrar y nos evitamos así simplificaciones posteriores.

5.5.2. Implementación

La forma de representar la operación $\otimes$ en Prolog es, al igual que con el monomorfismo, mediante un *patrón numerado*. Recordemos que un patrón numerado consistía en que en su nodo, además de la etiqueta guardábamos dos identificadores: uno el que identificaba al propio nodo dentro del patrón y el otro lo utilizábamos para diferentes funciones. En el monomorfismo se utilizaba para indicar con qué nodo se había unido, y en la operación $\otimes$ lo

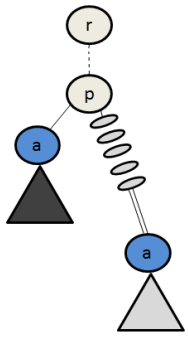

Figura 5.22: Fusión de nodos caso imposible

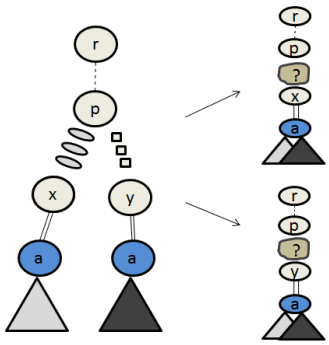

Figura 5.23: Fusión de nodos // y // diferente padre y ninguno hijo del ascendiente común más cercano

utilizaremos para indicar si es un nodo fusionado (para el cuál utilizaremos el número 0), si es un nodo del patrón p_1 (número 1) o si es un nodo del patrón p_2 (número 2).

Poniéndolo en un ejemplo, supongamos que queremos obtener $p_1 \otimes p_2$, siendo p_1 y p_2 respectivamente (como antes, utilizaremos el color azul para indicar los nodos de p_1 y el rojo para los nodos de p_2), los de la figura 5.24. Tras operar el resultado es el de la figura 5.25.

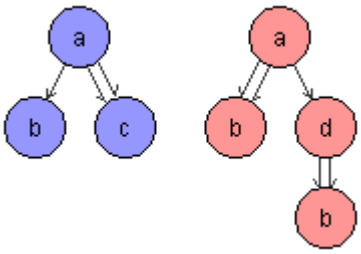

Figura 5.24: Operar: p_1 y p_2

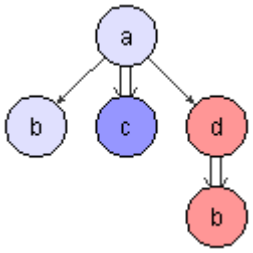

Figura 5.25: Operar: $p_1 \otimes p_2$

En Prolog, los patrones anteriores se indicarían de la siguiente manera:

- p_1 es $p([a,1,1],[h(/,p([b,2,1],[])),h(//,p([c,3,1],[]))])$
- p_2 es $p([a,1,2],[h(//,p([b,2,2],[])),h(/,p([d,3,2],[h(//,p([b,4,2],[]))]))])$
- $p_1 \otimes p_2$ es $p([a,1,0],[h(/,p([b,2,0],[])),h(//,p([c,3,1],[])),h(/,p([d,4,2],[h(//,p([b,5,2],[]))]))])$

O, visualmente, como aparecen en la figura 5.26.

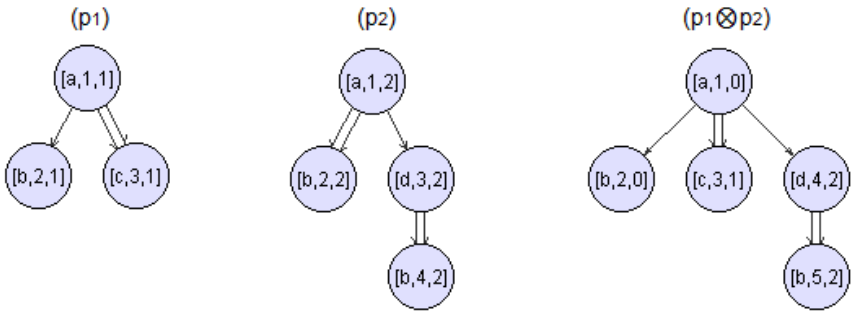

Figura 5.26: Operar: p_1, p_2 y $p_1 \otimes p_2$

5.5.3. Ejemplo

Vamos a ver ahora todo el proceso que llevaría realizar la operación $\otimes$ entre los patrones de la figura 5.27. El primer paso sería tratar de crear el primer nivel, compuesto por la fusión de las raíces. Puesto que ambas raíces son a, pueden fusionarse, por lo que el primer nivel será el de la figura 5.28.

Ahora hay que buscar todas las parejas de nodos a fusionar. Las posibles parejas son: b, c(1), c(2), c(3), d, f, j y q. Para cada una de ellas intentaremos fusionarlas, dando como resultado los patrones de la figura 5.29. Para las parejas c(1), c(3), j y q no había solución posible. Así, el nivel 2 lo componen los 4 patrones de la figura anterior. No hay repetidos, por lo que dejamos el

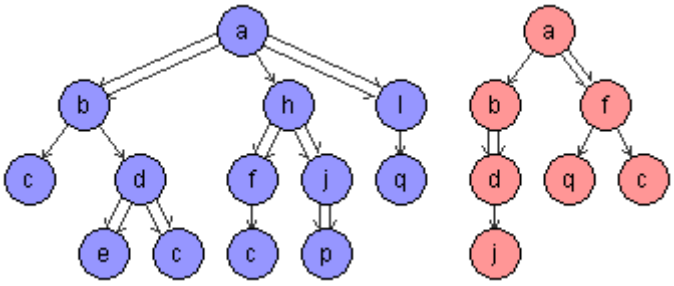

Figura 5.27: Patrones a operar: p y q

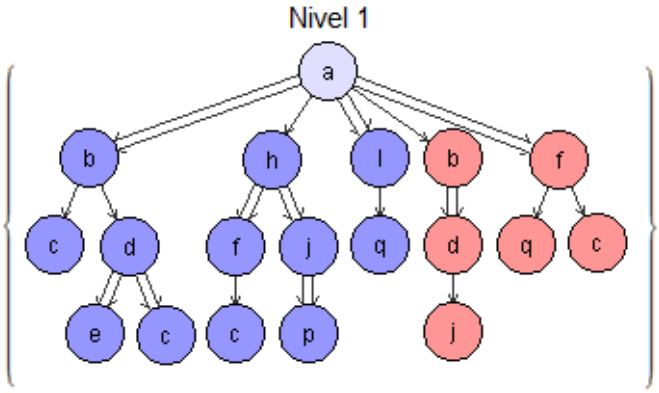

Figura 5.28: Ejemplo operar: Nivel 1

nivel como está. Ahora, para cada uno de los patrones del nivel actual, se calcularán todas las posibles parejas de nodos e intentaremos fusionarlas:

- A partir del patrón *Nivel 2: b* se obtienen las siguientes posibles parejas: c(1), c(2), c(3), d, f, j y q. Las soluciones obtenidas son las de la figura 5.30. Para las parejas c(1), c(3), j y q no había solución posible.
- A partir del patrón *Nivel 2: d* se obtienen las siguientes posibles parejas: c(1), c(2), c(3), f, j y q. Las soluciones obtenidas son las de la figura 5.31. Para las parejas c(1), c(3), j y q no había solución posible.
- A partir del patrón *Nivel 2: c(2)* se obtienen las siguientes posibles parejas: j y q. Sin embargo, no se obtiene ninguna solución.
- A partir del patrón *Nivel 2: f* se obtienen las siguientes posibles parejas: b, c(1), c(2), c(3), d, j y q. Las soluciones obtenidas son las de la figura 5.32. Para las parejas c(1), j y q no había solución posible.

Con esto hemos terminado de calcular el nivel 3. Ahora debemos hacer dos cosas. La primera, intentar simplificar los niveles anteriores al actual.

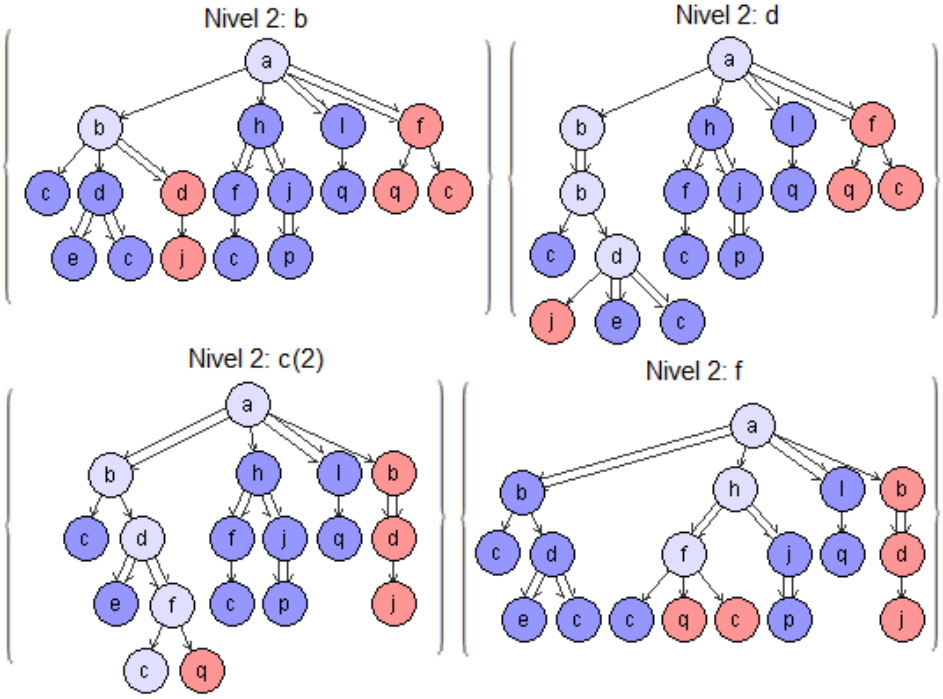

Figura 5.29: Ejemplo operar: Nivel 2

En este caso no ocurre ninguna simplificación, por lo que los niveles 1 y 2 se quedan como estaban. Lo segundo que tenemos que hacer es eliminar los patrones repetidos del nivel 3. Observemos que tenemos dos patrones repetidos: *b y f* se repite en *f y b*; y *d y f* se repite en *f y d*. De cada grupo anterior debemos dejar sólo uno de ellos. Eliminamos, por ejemplo, el patrón *f y b* así como el patrón *f y d*. Ahora calcularemos el siguiente nivel:

- A partir del patrón *Nivel 3: b y d* se obtienen las siguientes posibles parejas: c(1), c(2), c(3), f, j y q. Las soluciones obtenidas son las de la figura 5.33. Para las parejas c(1), c(3), j y q no había solución posible.
- A partir del patrón *Nivel 3: b y c(2)* se obtienen las siguientes posibles parejas: j y q. Sin embargo, no se obtiene ninguna solución.
- A partir del patrón *Nivel 3: b y f* se obtienen las siguientes posibles parejas: d, c(1), c(2), c(3), j y q. Las soluciones obtenidas son las de la figura 5.34. Para las parejas c(1), c(2), j y q no había solución posible.
- A partir del patrón *Nivel 3: d y c(2)* se obtienen las siguientes posibles parejas: j y q. Sin embargo, no se obtiene ninguna solución.
- A partir del patrón *Nivel 3: d y f* se obtienen las siguientes posibles

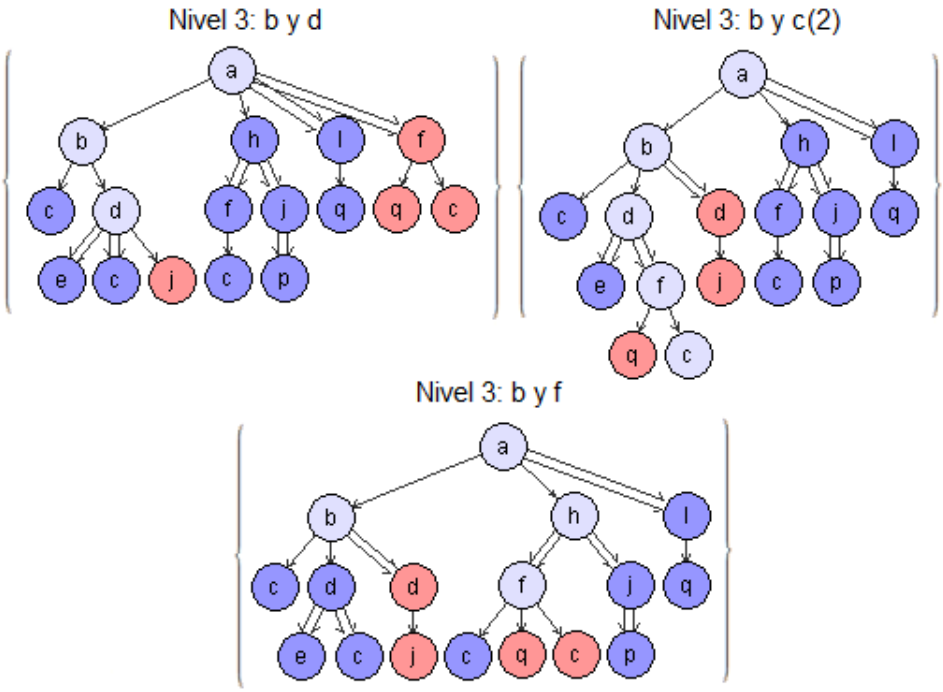

Figura 5.30: Ejemplo operar: patrones obtenidos de Nivel 2: b

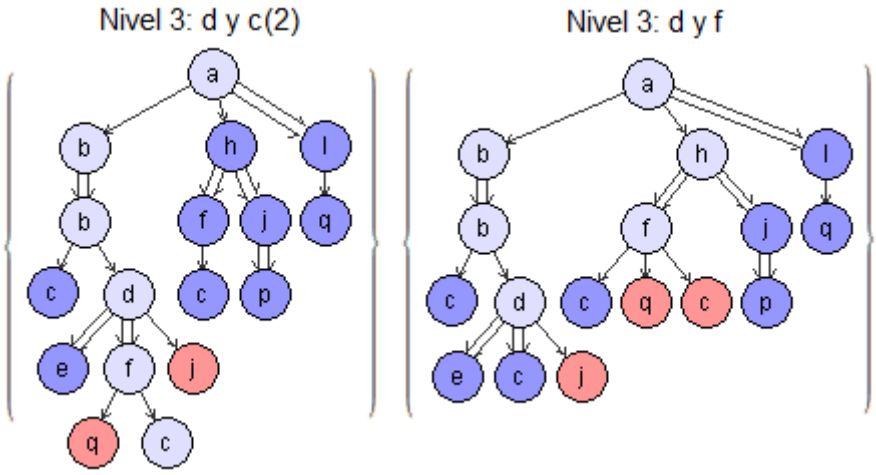

Figura 5.31: Ejemplo operar: patrones obtenidos de Nivel 2: d

parejas: c(1), c(2), c(3), j y q. Las soluciones obtenidas son las de la figura 5.35. Para las parejas c(1), c(2), j y q no había solución posible.

- A partir del patrón *Nivel 3: f y c(2)* se obtienen las siguientes posibles parejas: j y q. Sin embargo, no se obtiene ninguna solución.

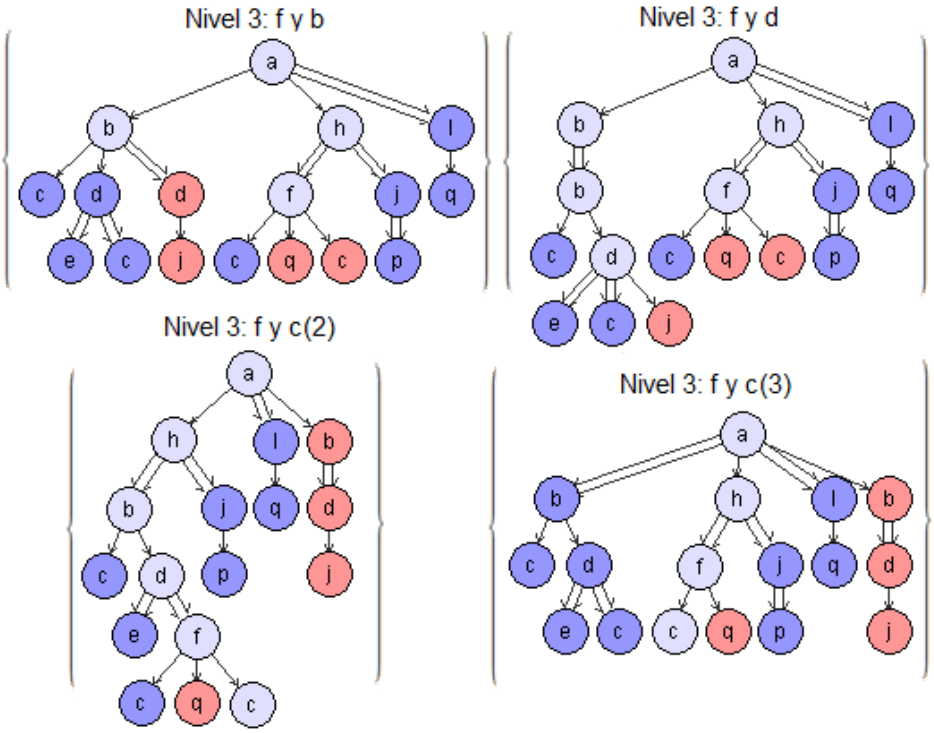

Figura 5.32: Ejemplo operar: patrones obtenidos de Nivel 2: f

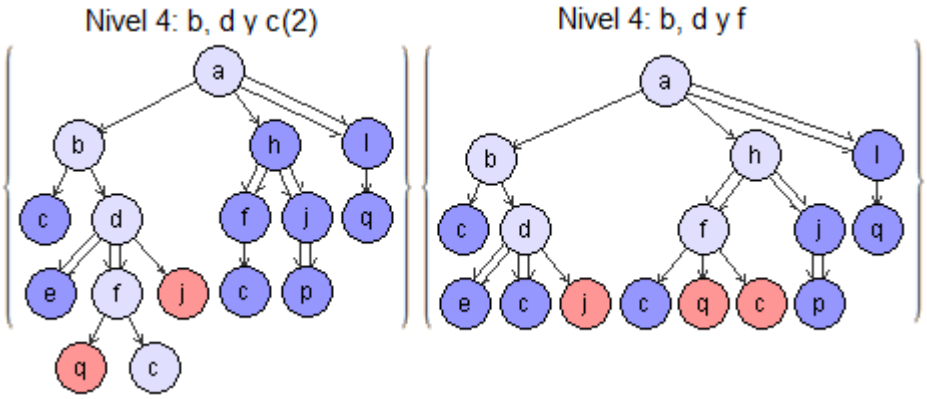

Figura 5.33: Ejemplo operar: patrones obtenidos de Nivel 3: b y d

- A partir del patrón *Nivel 3: f y c(3)* se obtienen las siguientes posibles parejas: b, d, j y q. Las soluciones obtenidas son las de la figura 5.36. Para las parejas j y q no había solución posible.

Tras calcular estos patrones habremos terminado el nivel 4. De nuevo, hay que hacer dos pasos: eliminar repetidos del nivel 4 y simplificar los niveles

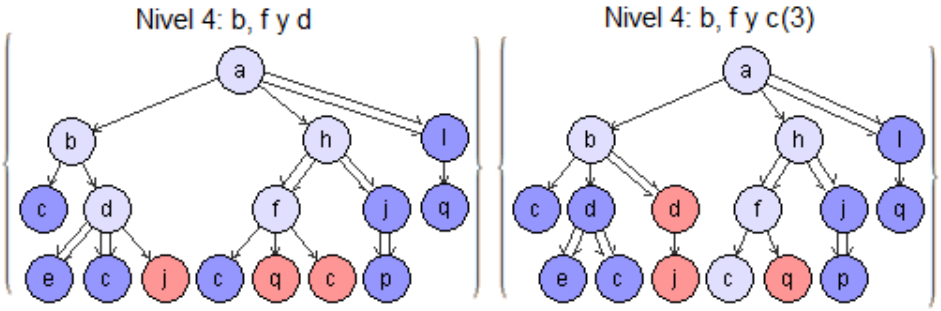

Figura 5.34: Ejemplo operar: patrones obtenidos de Nivel 3: b y f

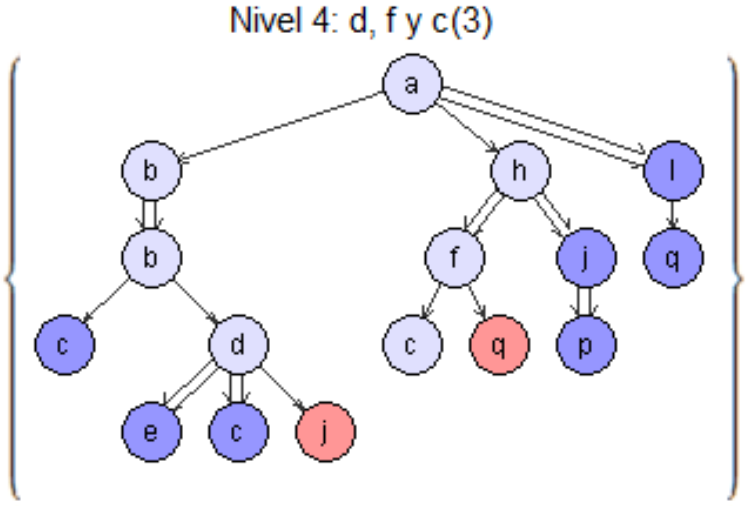

Figura 5.35: Ejemplo operar: patrones obtenidos de Nivel 3: d y f

anteriores al actual. Respecto a los repetidos del nivel actual, tenemos las siguientes repeticiones:

- *b, d y f / b, f y d*
- *b, f y c(3) / f, c(3) y b*
- *d, f y c(3) / f, c(3) y d*

Eliminamos, por ejemplo, *b, f y d*, *f, c(3) y b* y *f, c(3) y d*. Respecto a las simplificaciones de los niveles anteriores, hay un monomorfismo de *Nivel 3: f y c(3)* en *Nivel 2: f* y de *Nivel 3: f y c(3)* en *Nivel 3: f y c(2)*, por lo que eliminamos *Nivel 2: f* y *Nivel 3: f y c(2)*. El proceso continúa hasta que se obtiene la solución final, mostrada en la figura 5.37.

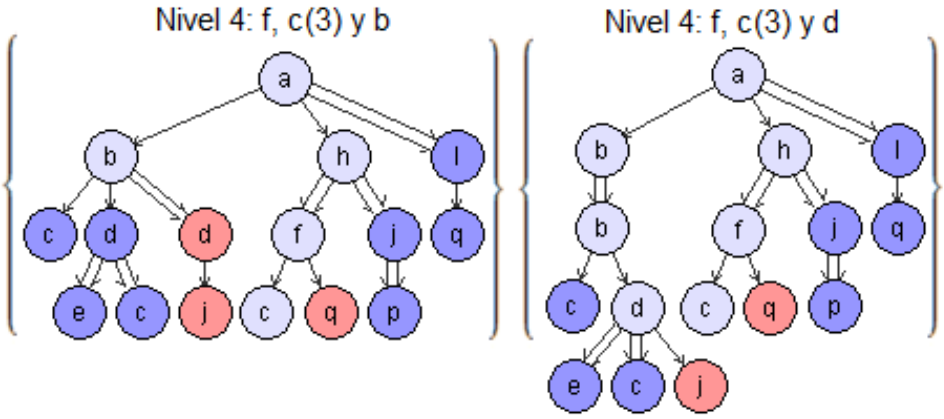

Figura 5.36: Ejemplo operar: patrones obtenidos de Nivel 3: f y c(3)

5.6. Operar Con Común

En este apartado se explica el algoritmo de la operación $p_1 \otimes_{c,m} q$, su implementación y un ejemplo.

5.6.1. Algoritmo

Obtener el algoritmo para esta operación, una vez obtenido el de $p_1 \otimes p_2$, es bastante sencillo. Suponemos que partimos de un constraint condicional $\forall(c : p_2 \to q)$ y un constraint positivo $\exists p_1$.

Lo primero es calcular todos los monomorfismos de p_2 en p_1 no extensibles a $q \to p_1$. Para hacer esto, se calculan todos los monomorfismos $m_1 : p_2 \to p_1$. Si no hubiera ninguno, la operación finalizaría y no se devolvería ninguna solución. Si hay al menos uno, para cada uno de ellos se hace lo siguiente: se calculan todos los monomorfismos $m_2 : q \to p_1$. Si no existe ninguno, se guarda el monomorfismo m_1 actual como un *monomorfismo de* p_2 *en* p_1 *no extensible a* $q \to p_1$. Si existiera al menos uno, habría que comprobar si alguno de ellos *coincide* con el monomorfismo m_1. Si no *coincide* ninguno, guardamos el monomorfismo m_1 actual como un *monomorfismo de* p_2 *en* p_1 *no extensible a* $q \to p_1$. Sino, el monomorfismo m_1 actual no sirve.

El concepto de *coincidir* es el siguiente: dados los monomorfismos $m_1 : p_2 \to p_1$ y $m_2 : q \to p_1$, y la relación de pre-arbol $c : p_2 \to q$, si todos los nodos de m_1[4] que se han unido con un nodo de p_2 tienen las mismas marcas que esos mismos nodos en m_2[5], se considera que m_1 y m_2 *coinciden*.

[4]Recordemos que al ser p_2 pre-árbol de q los nodos comunes se numeran con el mismo identificador; y que un monomorfismo $m : p \to q$ lo representamos como el propio patrón q cuyos nodos pueden estar marcados, indicando el nodo de p con el que se han unido.

[5]m_1 y m_2 tienen exactamente los mismos nodos ya que son monormofismos sobre p_1, por lo que lo único en lo que se diferencian es en las marcas de sus nodos.

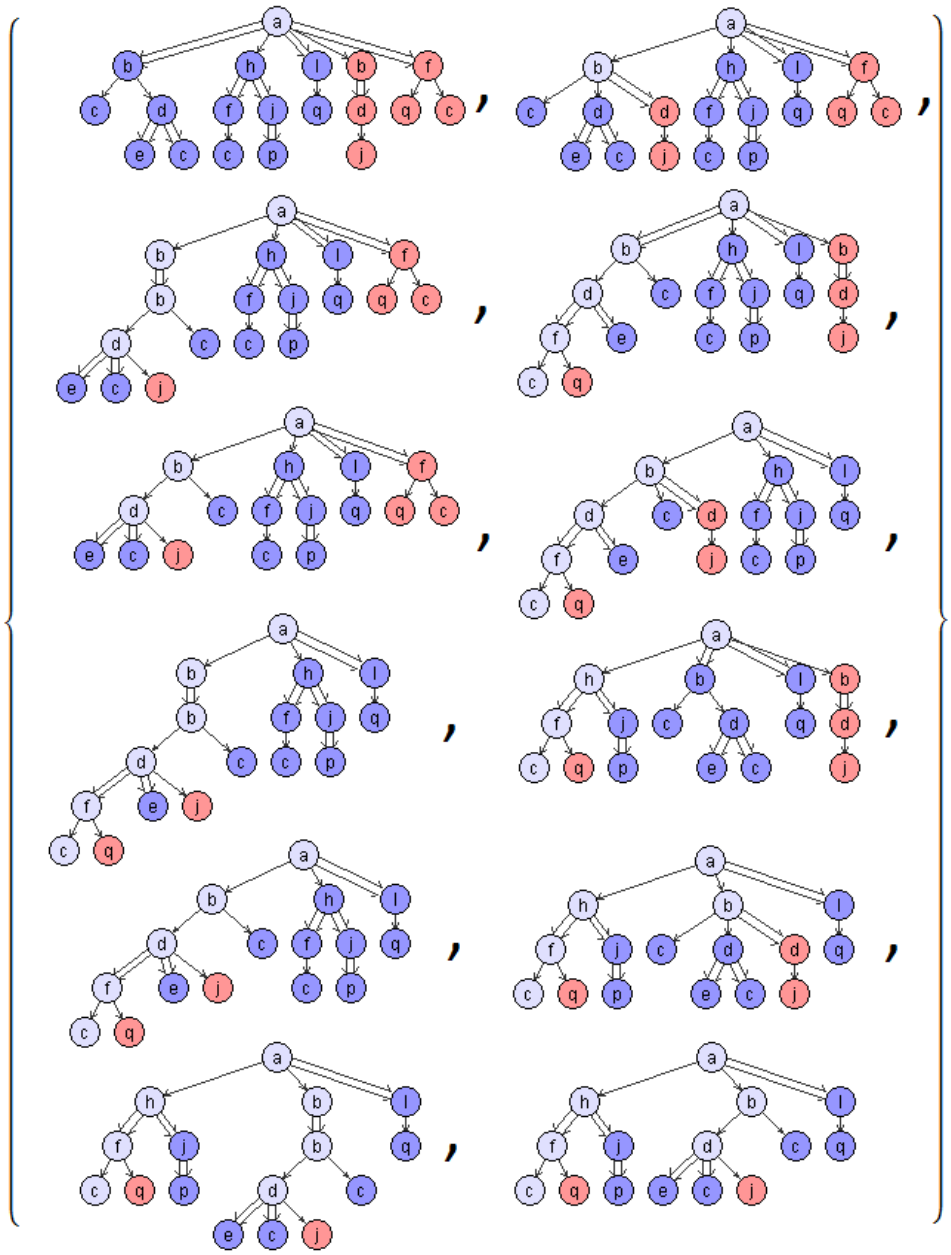

Figura 5.37: Ejemplo operar: solución

Una vez tenemos los monomorfismos m_1 no extensibles, sólo queda colocar, para cada uno, los nodos de q no pertenecientes a p_2 en su lugar correspondiente y aplicar el algoritmo de la operación $p_1 \otimes p_2$. Para colocar los nodos, se obtiene lo primero de todo una lista L con todos los nodos de p_2. Después, para cada uno de ellos se hace lo siguiente: se busca en q ese

mismo nodo (el cual existe, ya que p_2 es pre-árbol de q) y se guardan en una lista L_2 todos los sub-árboles hijos de ese nodo, de donde se eliminan todos los sub-árboles cuyo nodo raíz esté ya en la lista L. Ahora se busca el nodo actual en el monomorfismo m_1 y se colocan como hijos todos los sub-árboles de $L2$. Una vez realizado esto para todos los nodos, se aplica el algoritmo de la operación $p_1 \otimes p_2$ a partir del primer nivel, es decir, que el patrón obtenido al colocar los nodos de q en m_1 será el que constituya el primer nivel, por lo que se empezará obteniendo el segundo nivel.

Por último, hay que indicar que, puesto que en la operación $p'_1 \otimes p'_2$ se diferencian los **nodos de p'_1, los de p'_2 y los *fusionados***, necesitamos indicar también cómo son los nodos del patrón de nuestro nivel 1. Por lo tanto, los nodos pertenecientes a L serán los **nodos *fusionados***. El resto de los nodos que pertenecían a q (es decir, los que se han *colocado* en m_1) serán los **nodos pertenecientes a p'_2** y, por último, el resto de los nodos de m_1 serán los **nodos pertenecientes a p'_1**.

5.6.2. Implementación

El modo de representar el resultado es, al igual que en la operación $p_1 \otimes p_2$, mediante un patrón numerado. Sin embargo, durante el proceso de esta operación hay que aplicar un proceso de renumeración, ya que se parte de un **patrón numerado para monomorfismo**, y se termina en un **patrón numerado para la operación** $\otimes$.

La renumeración comienza cuando se colocan los nodos de q en el monomorfismo m_1. Los nodos nuevos (los de q) se colocan en m_1 con el identificador que corresponda, y con un $Id2$ de valor -1. Una vez colocados todos los nodos, se procede a la renumeración: los nodos con $Id2 > 0$ (los *fusionados*) pasan a tener $Id = 0$, los nodos con $Id2 = -1$ (los de q *colocados* en m_1) pasan a tener $Id2 = 2$, y los nodos con $Id2 = 0$ (el resto de nodos de m_1) pasan a tener $Id2 = 1$.

5.6.3. Ejemplo

En este apartado veremos un ejemplo sencillo del proceso a realizar en la operación $p_1 \otimes_{c,m} q$, siendo los constraints $\exists p_1$ y $\forall(c : p_2 \to q)$ los que se muestran en la figura 5.38.

Lo primero es encontrar un monomorfismo no extensible. En este caso, se puede ver que existe un único monomorfismo m de p_2 en p_1 y que no existe ningún monomorfismo de q en p_1, por lo que m es no-extensible. Ahora se colocan los nodos de q restantes en m, cuyo resultado se muestra en la figura 5.39 (los nodos grises son los *fusionados*, los rojos son los de q y los azules son los de m que no se han fusionado con ninguno de p_2).

Por último, sólo queda aplicar el algoritmo de la operación $\otimes$, suponiendo que el patrón de la figura 5.39 constituye el nivel 1. El resultado final es el

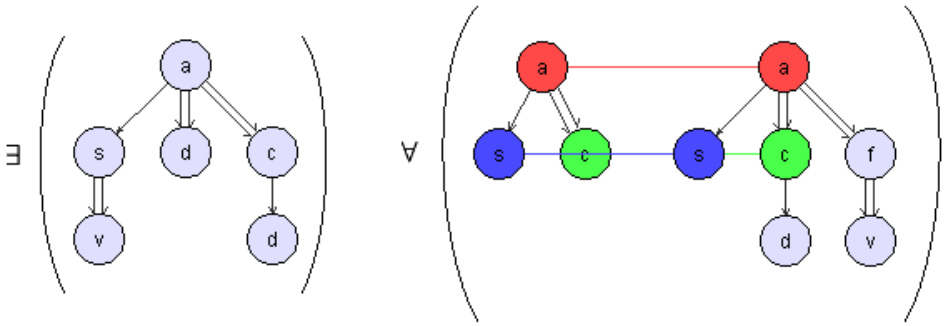

Figura 5.38: Constraints $\exists p_1$ y $\forall(c : p_2 \rightarrow q)$

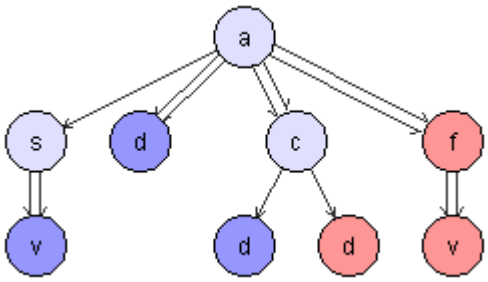

Figura 5.39: Nivel 1. Monomorfismo m con los nodos de q

de la figura 5.40.

5.7. Unfold

A lo largo del proyecto se ha hecho un análisis en profundidad de las reglas Unfold, que ha resultado en la obtención de una única regla equivalente, en lugar de las dos que previamente se habían concebido. El objetivo de las reglas Unfold era la de sustituir un constraint positivo que no activa ninguna regla, por la disyunción equivalente de otros dos constraints positivos. Estos nuevos constraints podrían activar alguna regla, pero también podría ser que hubiera que volver a aplicar las reglas Unfold.

Por ello, el objetivo del análisis realizado era encontrar una regla que aplicara las reglas Unfold en un sólo paso, obteniendo una disyunción de constraints equivalente a cualquier otra solución obtenida a través de la aplicación de las reglas *Unfold1* o *Unfold2*, las cuales producían diferentes soluciones dependiendo del orden de aplicación de las mismas. Finalmente, la regla Unfold obtenida realiza lo siguiente sobre un constraint positivo al que se le va a aplicar esta regla:

1. Se obtiene el valor de *Starlength* para el constraint, esto es, de entre

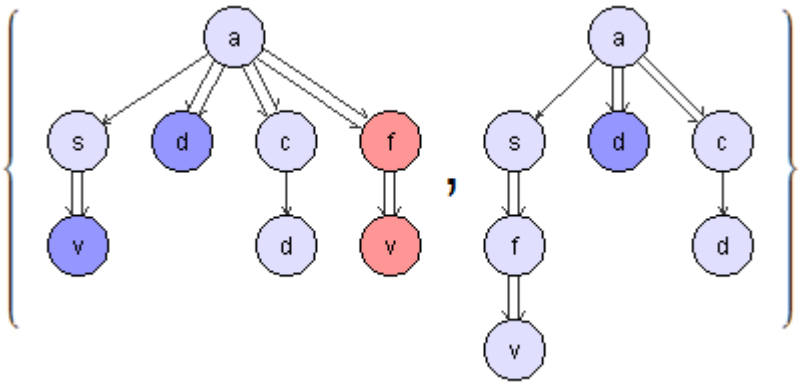

Figura 5.40: Solución del ejemplo

todas las cadenas de asteriscos[6] que aparecen en los constraints negativos y condicionales, se obtiene el máximo valor de la longitud de dichas cadenas.

2. Luego, se selecciona un arco *descendiente* (a, b) del constraint y se expande, obteniendo $2xStarlength + 1$ patrones. El primero es el propio patrón pero sustituyendo el arco *descendiente* por el de *hijo directo*. El segundo, sustituyendo el arco *descendiente* por un *hijo directo* de etiqueta asterisco y luego colocando b como *hijo directo*. El tercero, colocando un *hijo directo* asterisco, que a su vez tiene un *hijo directo* asterisco que tiene a b como *hijo directo*, y así sucesivamente hasta el penúltimo patrón. El último patrón es diferente: se colocan $Starlength$ asteriscos (cada uno como *hijo directo* del anterior), luego un arco *descendiente*, y luego otros $Starlength$ asteriscos. Este proceso se repite con el resto de arcos *descendientes* del constraint original, en cada patrón obtenido en la disyunción; mientras que el arco // del último patrón que proviene de expandir (a, b) se marca para no ser expandido más.

   En la figura 5.41 se muestra un ejemplo para la expansión de un patrón con $Starlength = 2$.

5.8. Subsunción

En principio, la regla de la subsunción consistía en que, si hay una cláusula $C_1 = A \vee B$ y otra $C_2 = A \vee B \vee \Gamma$, la cláusula C_2 era reduntante[7], por lo

[6] Una cadena de asteriscos es cualquier sub-patrón en el que su raíz tenga un único hijo (que además de etiqueta sea un asterisco), que a su vez sólo tenga un único hijo (también asterisco), etc.

[7] Ya que todos los modelos de C_1 son también modelos de C_2.

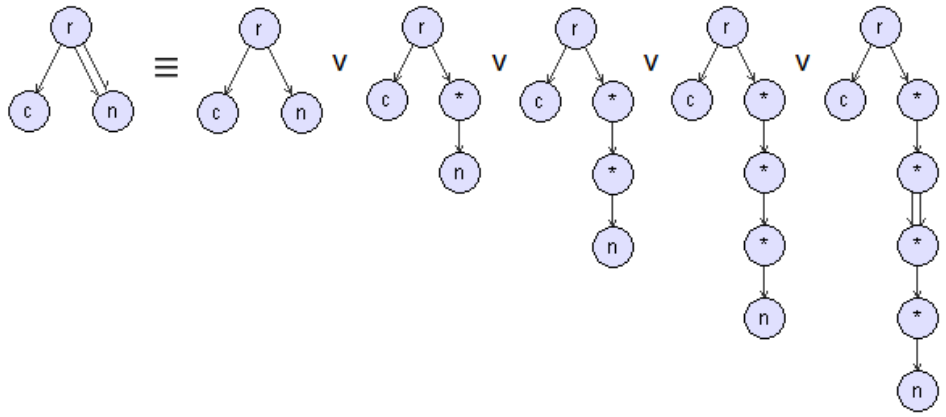

Figura 5.41: Aplicación de la regla Unfold con $Starlength = 2$

que podría ser subsumida por C_1. Sin embargo, para que una cláusula subsuma a otra no es necesario que sus constraints sean exactamente iguales a algunos o todos los de otra cláusula. Hemos visto con el monomorfismo que, si existe un monomorfismo de un patrón p en otro q, significa que todos los modelos de q son también modelos de p, con lo que se podría extender a la subsunción. Esto da lugar a tres nuevas reglas de subsunción, mejorando considerablemente la eficiencia del procedimiento de refutación:

- **S2**. Esta regla se encarga de comparar los constraints positivos entre dos cláusulas. Dados dos constraints $\exists p$ y $\exists q$, si existe un monomorfismo de p en q, significa que $Mod(\exists q) \subseteq Mod(\exists p)$. Por lo tanto, el esquema de la regla S2 es:

$$\frac{\exists q \vee \Gamma \qquad \exists p \vee \Gamma}{\exists q \vee \Gamma} \quad \textbf{(S2)}$$

si existe un monomorfismo $m : p \to q$

- **S3**. Esta regla compara los constraints negativos. Dados dos constraints $\neg\exists p$ y $\neg\exists q$, si existe un monomorfismo de p en q, significa que $Mod(\neg\exists p) \subseteq Mod(\neg\exists q)$. El esquema de la regla S3 es:

$$\frac{\neg\exists p \vee \Gamma \qquad \neg\exists q \vee \Gamma}{\neg\exists p \vee \Gamma} \quad \textbf{(S3)}$$

si existe un monomorfismo $m : p \to q$

- **S4**. Esta regla compara los constraints condicionales. Dados dos constraints $\forall(c_1 : x_1 \to q_1)$ y $\forall(c_2 : x_2 \to q_2)$, sucederá que $Mod(\forall(c_1 :$

$x_1 \to q_1)) \subseteq Mod(\forall(c_2 : x_2 \to q_2))$ si existe un monomorfismo de x_1 en x_2 y de q_2 en q_1. Por lo tanto, el esquema de la regla S4 es:

$$\frac{\forall(c_1 : x_1 \to q_1) \vee \Gamma \qquad \forall(c_2 : x_2 \to q_2) \vee \Gamma}{\forall(c_1 : x_1 \to q_1) \vee \Gamma} \textbf{ (S4)}$$

si existen dos monomorfismos $f : x_1 \to x_2$ y $g : q_2 \to q_1$

5.9. Reglas de Simplificación

Al igual que en la conjunción de cláusulas el monomorfismo ha contribuido a mejorar la subsunción de cláusulas, también se puede utilizar para mejorar las reglas de simplificación en las disyunciones de constraints ya que, mientras que en la conjunción de A y B se elimina B si $Mod(A) \subseteq Mod(B)$, en la disyunción ocurre lo contrario: se elimina B si $Mod(B) \subseteq Mod(A)$. De ese modo, obtenemos tres reglas de simplificación:

- **Sim2**. Esta regla se encarga de simplificar dos constraints positivos. Dados dos constraints $\exists p$ y $\exists q$, si existe un monomorfismo de p en q, significa que $Mod(\exists q) \subseteq Mod(\exists p)$, dando como resultado la siguiente regla:

$$\frac{\exists p \vee \exists q \vee \Gamma}{\exists p \vee \Gamma} \textbf{ (Sim2)}$$

si existe un monomorfismo $m : p \to q$

- **Sim3**. Esta regla simplifica dos constraints negativos. Dados dos constraints $\neg\exists p$ y $\neg\exists q$, si existe un monomorfismo de p en q, entonces $Mod(\neg\exists p) \subseteq Mod(\neg\exists q)$:

$$\frac{\neg\exists p \vee \neg\exists q \vee \Gamma}{\neg\exists q \vee \Gamma} \textbf{ (Sim3)}$$

si existe un monomorfismo $m : p \to q$

- **Sim4**. Por último, esta regla simplifica dos constraints condicionales. Dados dos constraints $\forall(c_1 : x_1 \to q_1)$ y $\forall(c_2 : x_2 \to q_2)$, si existe un monomorfismo de x_1 en x_2 y de q_2 en q_1, entonces $Mod(\forall(c_1 : x_1 \to q_1)) \subseteq Mod(\forall(c_2 : x_2 \to q_2))$. Por lo tanto, el esquema de la regla es:

$$\frac{\forall(c_1 : x_1 \to q_1) \vee \forall(c_2 : x_2 \to q_2) \vee \Gamma}{\forall(c_2 : x_2 \to q_2) \vee \Gamma} \quad \textbf{(Sim4)}$$

si existen dos monomorfismos $f : x_1 \to x_2$ and $g : q_2 \to q_1$

5.10. Procedimiento de Refutación

En este apartado explicaremos el algoritmo del procedimiento de refutación. Puesto que existen dos versiones, se expondrán los dos algoritmos por separado.

5.10.1. Versión 1

Lo primero que se hace es añadir las cláusulas iniciales al historial. Después, se intenta aplicar las reglas de subsunción para reducir el conjunto de cláusulas inicial. Si se subsume alguna cláusula, se añaden al historial las cláusulas subsumidas. Una vez simplificada la especificación, se aplican todas las cláusulas posibles entre ellas (este proceso se explica más adelante) y se guardan las nuevas cláusulas obtenidas en una lista de cláusulas Sp separadas de las de la especificación original S. Ahora el bucle principal da comienzo, siguiendo los siguientes pasos:

1. Si se ha encontrado *false*, el bucle para y el procedimiento termina devolviendo *false*.

2. Si el conjunto de cláusulas nuevas Sp es vacío, significa que no se pueden aplicar más reglas, por lo que el bucle para y el procedimiento termina devolviendo *true*.

3. Si no, se aplican las reglas en dos partes. Primero, se aplican todas las reglas posibles entre las cláusulas de S y las de Sp, es decir, todas las reglas posibles entre dos cláusulas de las cuales una de ellas pertenece al conjunto S y la otra a Sp, y se guardan las nuevas cláusulas generadas en una lista nueva A. Luego, se aplican todas las reglas posibles entre las cláusulas de Sp, es decir, todas las reglas posibles entre dos cláusulas pertenecientes ambas al conjunto Sp, guardando las cláusulas nuevas también en la lista A. Una vez aplicadas todas las reglas, sustituimos S por la concatenación de los conjuntos S y Sp, y Sp por la lista A. Después, se realiza una nueva iteración del bucle, yendo al paso 1.

El proceso de aplicar las reglas apenas varía ya sea sobre una cláusula de S y otra de Sp o si es sobre dos cláusulas del conjunto Sp. En este último caso, las cláusulas de S tan sólo se usan en las reglas de subsunción. Por lo demás, el proceso consiste en lo siguiente: se buscan todas las parejas

de constraints pertenecientes uno a una cláusula de S y otro a una de Sp (o las dos de Sp si es el otro caso), que cumplan que al menos uno de ellos sea un constraint positivo[8]. Luego, para cada pareja de constraints se intentará aplicar la regla correspondiente.

- Si la regla es R1, se comprobará que exista un monomorfismo del constraint negativo al positivo. Si existe, se aplica la regla R1 y se obtiene una nueva cláusula C. Si C es la cláusula *FALSE*, se indica y el procedimiento para. Si no, se aplican las reglas de simplificación sobre C y, después, las reglas de subsunción entre todas las cláusulas (independientemente de que pertenezcan a S, Sp o a la lista A). Si alguna cláusula de los conjuntos S o Sp se subsume por C, se actualiza el conjunto correspondiente eliminando dicha cláusula. Si C no se ha subsumido, se añade a la lista A.

- Si la regla es R2, se comprobará que no exista ningún monomorfismo de un constraint al otro. Si esto se cumple, se aplica la regla R2 y se obtiene una nueva cláusula C. Al igual que en el caso anterior, si C es la cláusula *FALSE*, se indica y el procedimiento para, mientras que si no era *FALSE*, se aplicarían las reglas de simplificación y subsunción.

- Si la regla es R3, se calcularán primero todos los monomorfismos de p_2 en p_1 no extensibles a $q \rightarrow p_1$. Para cada monomorfismo resultante se aplicará la regla R3 utilizando dicho monomorfismo en la operación $p_1 \otimes_{c,m} q$. De nuevo, una vez aplicada esta regla para cada monomorfismo se aplicarán las reglas de simplificación y subsunción.

5.10.2. Versión 2

Esta versión comienza también añadiendo las cláusulas iniciales al historial y aplicando las reglas de subsunción. A continuación, ejecuta la versión 1. Si el resultado es *false*, el procedimiento finaliza devolviendo *false*. Si el resultado es *true*, guarda en memoria dinámica (para un acceso posterior mucho más rápido) todos los patrones de los constraints negativos o premisas de constraints condicionales a los que se les ha sustituido todas las cadenas de asteriscos por un arco *descendiente*. Después comienza el bucle principal, siguiendo los siguientes pasos:

1. Si se ha encontrado *false*, el bucle para y el procedimiento termina devolviendo *false*.

2. Si no, se seleccionan todos los constraints positivos $\exists q$ que cumplan las tres condiciones siguientes: que no se le haya aplicado previamente

[8] Las tres reglas R1, R2 y R3 tienen en común que uno de los dos constraints sobre los que se aplican es siempre un constraint positivo.

la regla Unfold, que tenga al menos un arco *descendiente* y que exista algún monomorfismo $m : p \to q$, donde p es un constraint negativo o premisa de un constraint condicional que se ha guardado previamente. Ahora pueden ocurrir dos cosas:

- Si no existe ningún constraint positivo que cumpla las tres condiciones, el bucle para y el procedimiento termina devolviendo *true*.
- Si no, como se ha explicado en la sección 5.7, se aplica la regla Unfold a todos los constraints que cumplen las tres condiciones y se ejecuta la versión 1 del procedimiento de refutación. Después, se realiza una nueva iteración del bucle, volviendo al paso 1.

En resumen, dada una especificación como entrada al algoritmo, el resultado final es uno de los tres siguientes:

a) Si el procedimiento finaliza con *false*, la especificación es insatisfacible.

b) Si el procedimiento finaliza con *true*, la especificación es satisfacible y tiene un modelo finito.

c) Si el procedimiento no finaliza, entonces la especificación es satisfacible y sólo tiene modelos infinitos.

5.11. Mejoras en la Implementación

Queda pendiente la realización de varias mejoras en la aplicación. En este apartado expondremos una breve explicación de las mismas.

- Inhabilitar la posibilidad de crear arcos *descendientes*, así como nodos de etiqueta '*', cuando se está editanto un documento XML.
- Posibilidad de guardar documentos XML y poder cargarlos en la aplicación.
- Inclusión de un botón *Seleccionar todo*, que permita seleccionar todos los nodos del patrón automáticamente.
- Mejora de las reglas de subsunción y simplificación. Hasta ahora se había mejorado la regla principal de subsunción para que, dadas las cláusulas $C_1 = \exists q \vee \Gamma$ y $C_2 = \exists p \vee \Gamma$, C_1 subsuma a C_2 si existe un monomorfismo de p en q (o las condiciones correspondientes para los constraints negativos y condicionales). La mejora de la regla se centraría en no restringir el resto de las cláusulas C_1 y C_2 a que sean Γ, sino Γ_1 y Γ_2 que a su vez Γ_1 subsuma a Γ_2.

Parte II

Anexos

Anexo A

Manual de Configuración y Uso

En este apéndice se explicará todo lo relacionado con el despliegue y uso de la aplicación.

A.1. Requisitos

Esta aplicación es portable, por lo que no incorpora ningún archivo de instalación. Sin embargo, para su correcto funcionamiento sí necesita que varios programas ajenos a la propia aplicación estén previamente instalados. Todos los requisitos de la aplicación son los siguientes:

- Sistema Operativo Windows 7/XP. Es posible que con otras versiones de Windows la aplicación funcione correctamente. Sin embargo, no se han realizado pruebas en otras versiones de Windows ni en Sistemas Operativos diferentes.
- SWI-Prolog. Se deberá tener instalado el programa SWI-Prolog, mediante el cual se ejecutará la lógica de la aplicación. Se puede descargar desde `http://www.swi-prolog.org/Download.html`.
- Java. Se deberá tener la versión Java 1.7 (jre7). Al igual que con el Sistema Operativo, no se sabe si con otras versiones de Java funcionará o no.

A.2. Configuración del Puente Java-Prolog

Una vez SWI-Prolog está instalado, la configuración del puente Java-Prolog es muy sencilla. Sólo consta de dos pasos:

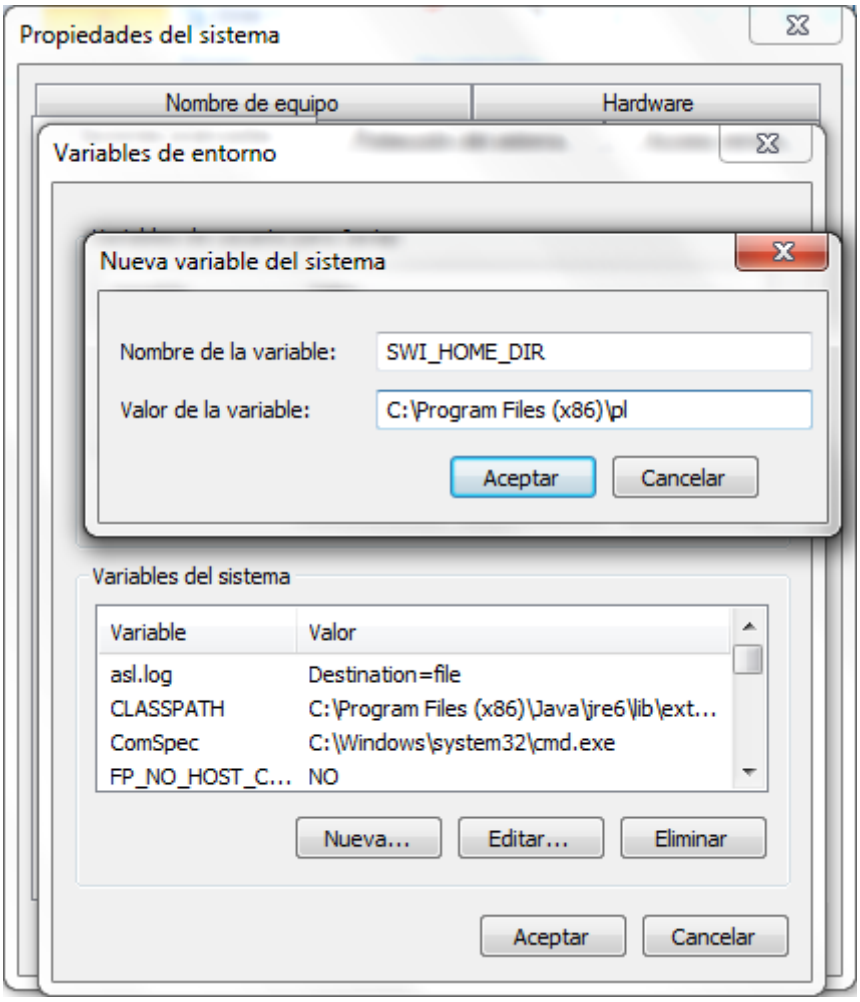

Figura A.1: Creación de la variable de entorno SWI_HOME_DIR

1. **Crear una variable del entorno SWI_HOME_DIR**. Para ello, ir a *Panel de Control→Sistema y Seguridad→Sistema* (o sino *Propiedades del sistema*). Una vez ahí seleccionar *Configuración avanzada del sistema* y hacer click en el botón «*Variables de entorno...*». Seleccionar el botón «*Nueva...*» en el panel *Variables del sistema* e introducir los siguientes datos:

   - *Nombre de la variable*: SWI_HOME_DIR
   - *Valor de la variable*: «Directorio de instalación SWI-Prolog»

   En la figura A.1 se muestra una imagen donde se puede ver la pantalla de creación de la variable.

2. **Modificar la variable del entorno Path**. En la misma pantalla

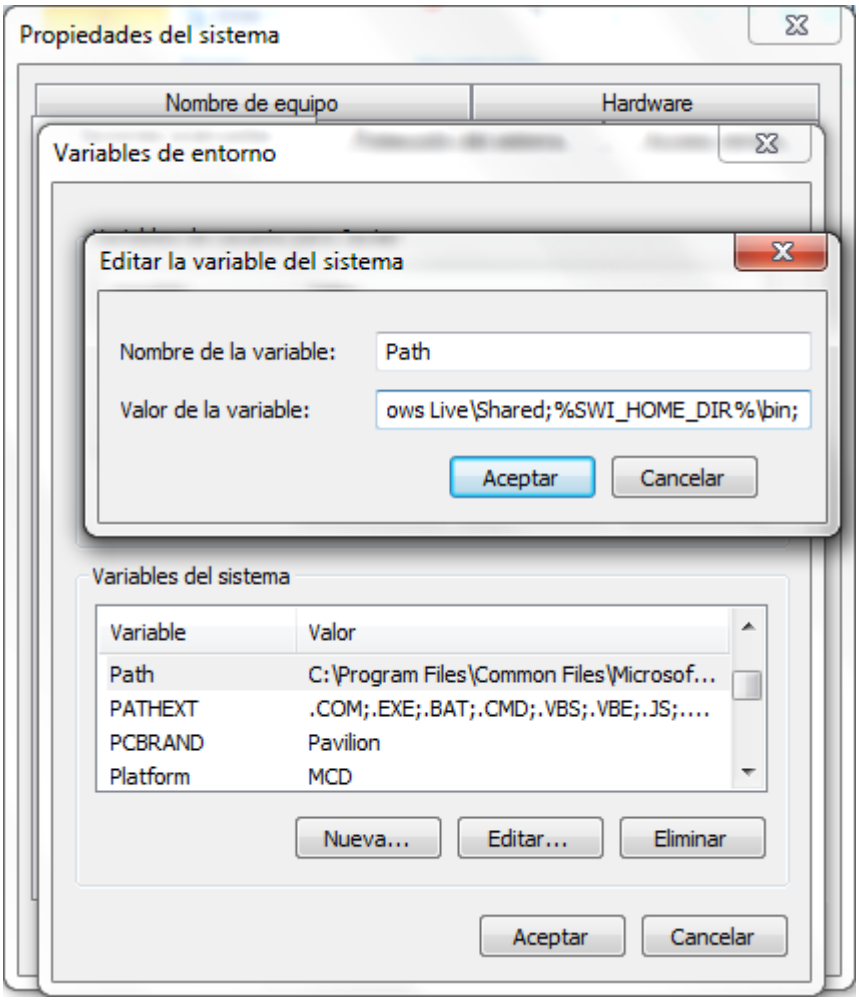

Figura A.2: Edición de la variable del entorno Path

que el paso anterior («Variables de entorno...») buscar la variable del sistema *Path*, seleccionarla y hacer click en el botón «Editar...». Indicar como valor de la variable: %SWI_HOME_DIR%\bin. En el caso de que ya hubiera un valor en la variable, no borrar el contenido: añadir un ';' y luego el valor %SWI_HOME_DIR%\bin. En la figura A.2 se muestra la pantalla de edición de la variable Path.

A.3. Manual de Uso

En este apartado se explicará cómo usar todas las ventanas de la aplicación.

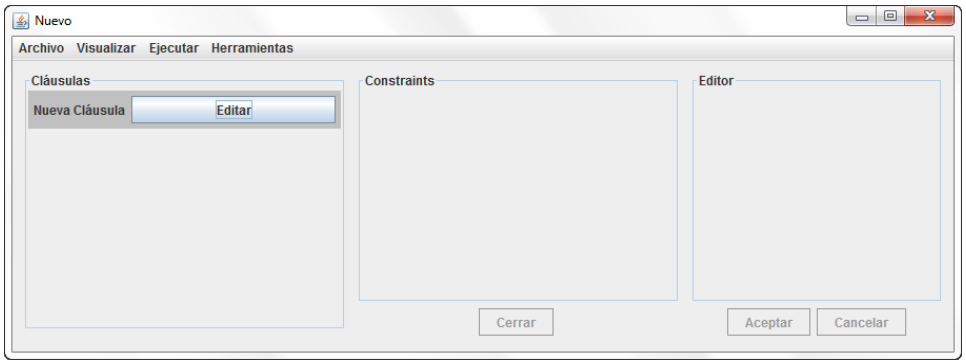

Figura A.3: Pantalla principal de la aplicación

A.3.1. Pantalla Principal

Al ejecutar la aplicación se abrirá la pantalla principal, la cual no se puede cerrar o sino finalizará la ejecución. La pantalla (ver figura A.3) se compone de: barra de menú, panel de cláusulas, panel de constraints y panel del editor.

A.3.2. Introducir Cláusulas

En el panel de cláusulas aparecerán todas las cláusulas creadas, ya estén eliminadas (por lo que no se incluirán en la especificación) o activas. Siempre habrá una fila de color gris en la que ponga *Nueva Cláusula*, la cual se usa para crear nuevas cláusulas. Si se selecciona el botón *Editar* de dicha "Nueva Cláusula", el color de la cláusula pasará a ser de color azul (el color de las cláusulas seleccionadas) y se mostrará el panel de los constraints.

El panel de los constraints funciona de la misma manera que el panel de cláusulas: aparecen tanto los activos como los eliminados y siempre va a haber una fila en la que ponga *Nuevo Constraint*. Seleccionando el botón *Editar* de *Nuevo Constraint*, se dibujará éste de color azul y se mostrará el editor de patrones.

Este último panel está compuesto por dos elementos: los botones que eligen el tipo de constraint a construir y el propio editor. Si el constraint seleccionado es positivo o negativo, el editor será individual. Si el constraint es condicional, el editor estará dividido en tres editores. Hay que tener en cuenta que, mientras que en los paneles de cláusulas y de constraints cualquier cambio que se produzca (eliminar cláusulas, cambiar el orden, etc.) se guarda automáticamente, los cambios producidos en el panel del editor deberán ser guardados haciendo click en el botón *Aceptar*. En el siguiente apartado se explicará cómo se editan los patrones.

Una vez dibujado un constraint en el panel del editor (ver figura A.4), se selecciona el botón *Aceptar* para guardar el constraint. Esto hace que, en el

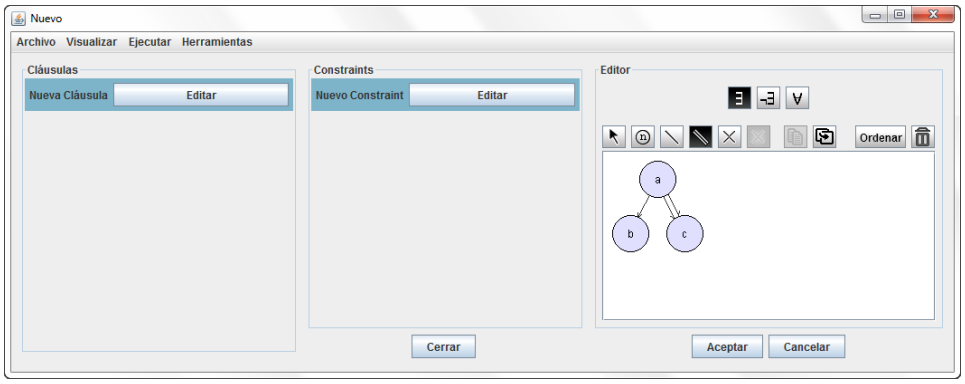

Figura A.4: Pantalla principal: introducir constraints

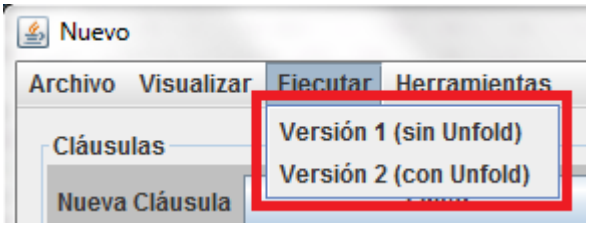

Figura A.5: Pantalla principal: ejecutar el procedimiento de refutación

panel de constraints, el *Nuevo Constraint* pase a llamarse del modo "CTX", donde X es un número positivo, y se cree una nueva fila de nombre *Nuevo Constraint*.

Una vez se han creado todas las cláusulas, para ejecutar el procedimiento de refutación tan sólo hay que ir al menú *Ejecutar* y seleccionar una de las dos opciones: *Versión 1* o *Versión 2*, tal y como se muestra en la figura A.5.

A.3.3. Resultado e Historial

Una vez terminado el procedimiento de refutación, el sistema devolverá el resultado del mismo, indicando además el tiempo de ejecución (ver figura A.6). Tras cerrar el mensaje, se cargará la pantalla del historial (ver figura A.7).

Las acciones que se pueden hacer en la pantalla del historial son: leer el historial, consultar cáusulas, consultar constraints y exportar el historial.

Consultar cláusulas se puede hacer de tres maneras. La primera, mediante el campo de búsqueda *Buscar cláusula* en la parte superior izquierda de la pantalla. Se debe introducir el identificador de una cláusula (por ejemplo c5) y pulsar intro o hacer click sobre el botón con forma de lupa. La segunda manera de consultar cláusulas es mediante el botón *Ver* en la esquina superior derecha. Ese botón cargará todas las cláusulas existentes por lo que no

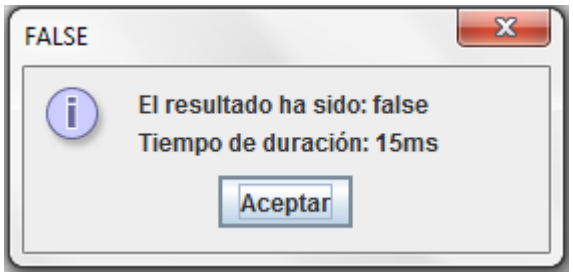

Figura A.6: Resultado del procedimiento de refutación

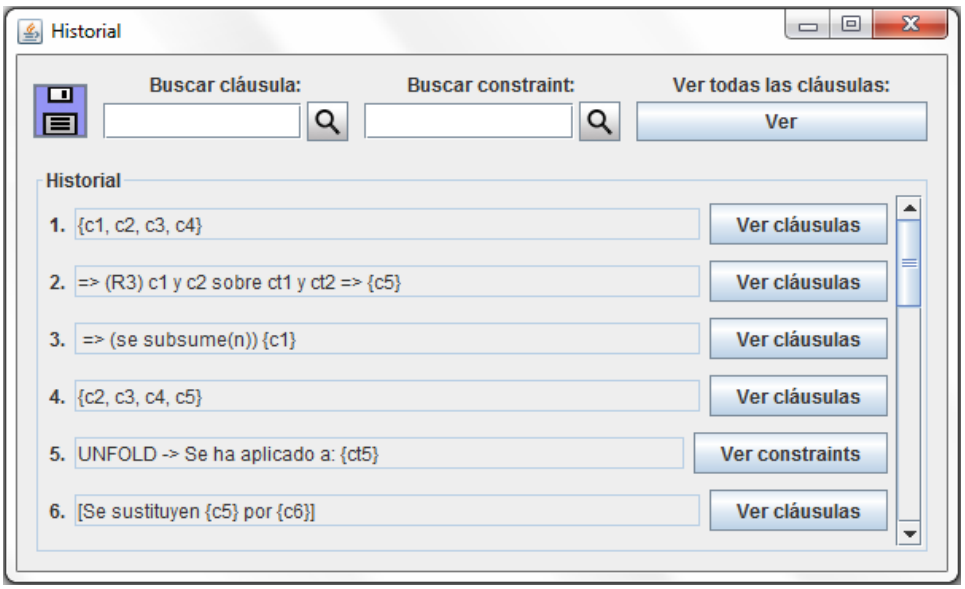

Figura A.7: Pantalla del historial

es necesario indicar ningún identificador. El último método para consultar cláusulas es mediante los botones *Ver cláusulas* de los diferentes pasos del historial. Esos botones cargan las cláusulas nombradas en su propio paso del historial. Así, si seleccionáramos el botón *Ver cláusulas* del paso 2 de la figura A.7, se cargarían las cláusulas c1, c2 y c5.

Para consultar constraints, se puede hacer de dos maneras. La primera, mediante el campo de búsqueda *Buscar constraint* en la parte superior. Este campo de búsqueda funciona como el de buscar cláusulas: se introduce el identificador de un constraint (como por ejemplo ct1) y se pulsa intro o se hace click sobre el botón con forma de lupa. La segunda manera de consultar constraints es mediante los botones *Ver constraints* de los pasos del historial en el que se haya aplicado la regla Unfold.

Por último, para exportar el historial, tan sólo hay que hacer click sobre el icono de guardar en la esquina superior izquierda, donde se introducirá el

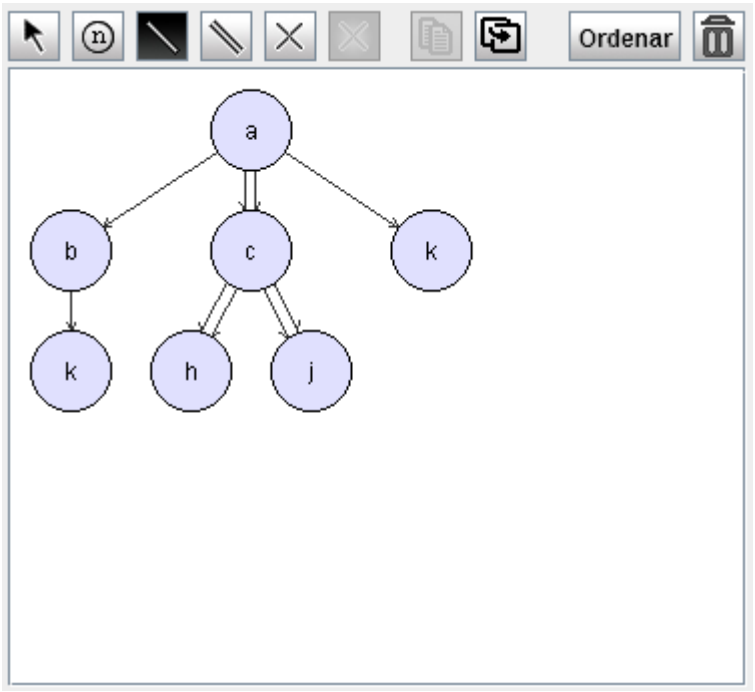

Figura A.8: Editor de patones

nombre del fichero en el que se quiere exportar el historial y se pulsará el botón *guardar*.

A.3.4. Editar Patrones

Los patrones se construyen en el editor (ver figura A.8). Los botones del editor son, de izquierda a derecha: *seleccionar*, *crear nodo*, *crear hijo directo*, *crear descendiente*, *eliminar*, *eliminar selección*, *copiar*, *pegar*, *ordenar* y *eliminar todo*. A continuación se explicarán uno a uno.

Seleccionar Haciendo click sobre este botón se activa el modo *selección*. En este modo, se pueden mover los nodos del editor haciendo click izquierdo sobre ellos y arrastrándolos. También se pueden mover varios nodos seleccionándolos primero y luego haciendo click izquierdo sobre alguno de los nodos seleccionados y arrastrando el ratón. Para seleccionar más de un nodo a la vez hay que hacer click izquierdo sobre el fondo del editor y arrastrar el ratón. Se formará un cuadro de selección y todos los nodos que estén dentro de ese cuadro quedarán seleccionados. Por último, si se hace click derecho sobre un nodo, se le podrá cambiar la etiqueta. Si al final no se desea cambiar la etiqueta, hay que dejar el campo de texto vacío y pulsar intro.

Crear Nodo Este botón activará el modo *nodo*. Haciendo click izquierdo en cualquier parte del editor se creará un nodo al que hay que introducir una etiqueta no vacía[1].

Crear Hijo Directo Este botón activa el modo *hijo*. Para crear una relación de tipo *hijo directo* entre dos nodos hay que hacer click izquierdo sobre el nodo que será el padre y luego, sin soltar el ratón, arrastrarlo hasta el nodo que será el hijo.

Crear Descendiente Este botón funciona exactamente igual que el botón *crear hijo directo*, salvo que crea una relación de tipo *descendiente*.

Eliminar Este botón es el último botón de estados. Establece el modo *eliminar*. Para eliminar un nodo, hay que hacer click izquierdo sobre un nodo (sin mover el ratón). Si se quiere eliminar una relación entre dos nodos, hay que hacer click izquierdo sobre uno de los nodos que participan en la relación a eliminar, y arrastrar el ratón (sin soltarlo) hasta el otro nodo de la relación.

Eliminar Selección Este botón se puede pulsar siempre que haya varios nodos seleccionados. Si se pulsa este botón, se eliminarán de golpe todos los nodos seleccionados en el modo *selección*.

Copiar Funciona del mismo modo que el botón anterior: cuando hay nodos seleccionados. Sin embargo, su función es copiar los nodos (junto con sus relaciones) seleccionados para poder pegarlos en cualquier editor de la aplicación.

Pegar Este botón sólo funciona cuando previamente en la aplicación se ha copiado algún nodo con el botón *Copiar*. Si hay nodos copiados, al pulsar este botón se introducirán en el editor dichos nodos con la misma posición en la que se han copiado.

Ordenar Haciendo click en este botón se reordenará el contenido del editor poniendo los patrones de un modo organizado.

Eliminar Todo Este botón limpiará el editor y lo dejará sin ningún nodo.

A.3.4.1. Editor ParaTodo

Este editor (ver figura A.9) es el que se utilizará para construir los constraints condicionales. Se compone de tres editores: dos superiores editables

[1] La etiqueta no puede contener el carácter comilla simple '

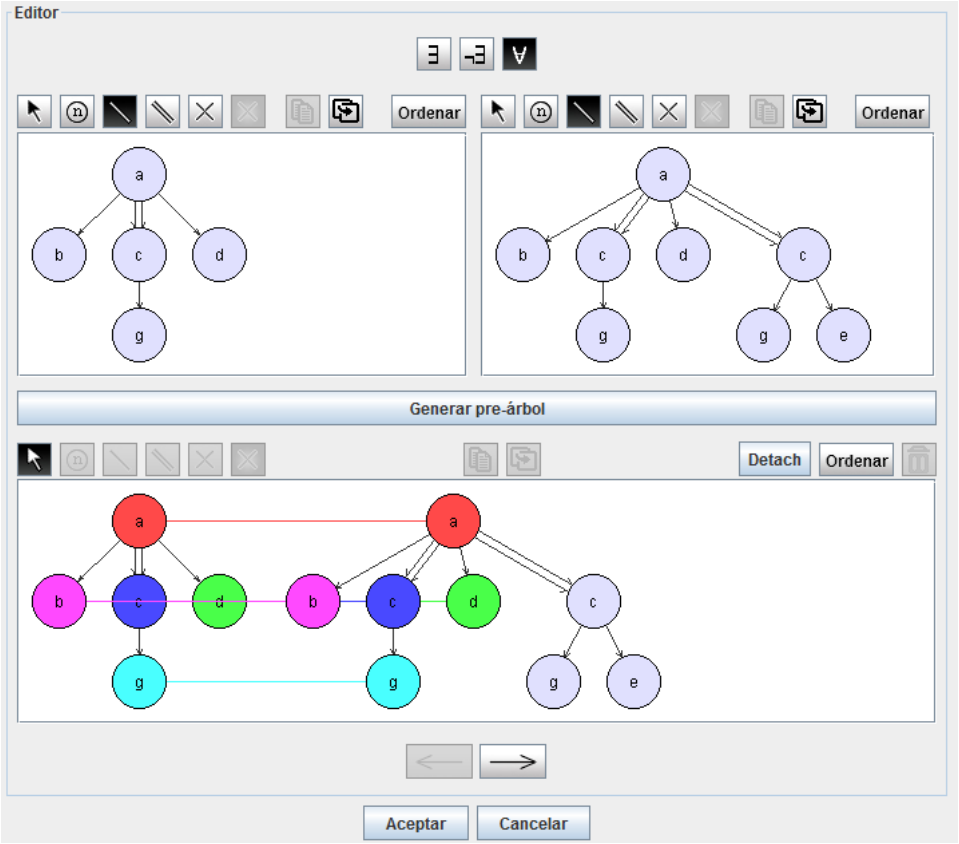

Figura A.9: Editor de constraints condicionales

y uno inferior en modo *sólo lectura*.

Suponiendo que estamos introduciendo el constraint $\forall(c : p \rightarrow q)$, en el editor superior izquierdo se introducirá el patrón p, y en el derecho el patrón q. Una vez introducidos se pulsará el botón *Generar pre-árbol* y el sistema calculará todos los posibles modos de colocar p como pre-árbol de q. En el editor inferior sólo se mostrará una única solución. Si se desea cambiar de solución, habrá que pulsar los botones con forma de flecha. Una vez se ha encontrado la relación de pre-árbol deseada, se pulsará el botón *Aceptar*.

A.3.5. Comprobar Especificación

Hay dos maneras de ejecutar este caso de uso: mediante la opción *Comprobar especificación* o mediante *Comprobar especificación actual*, ambas en el menú *Herramientas*, tal y como se puede ver en la figura A.10. A continuación se mostrarán las diferencias entre las dos:

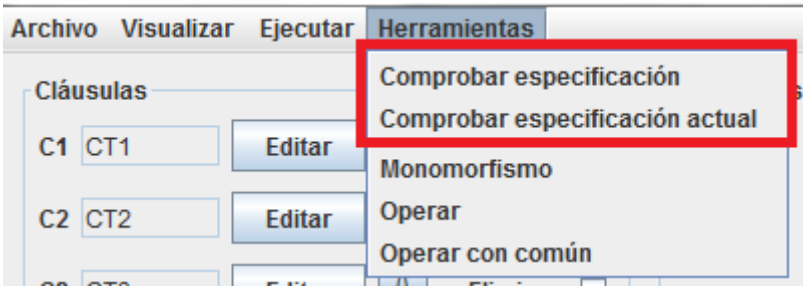

Figura A.10: Ejecutar el caso de uso Comprobar Especificación

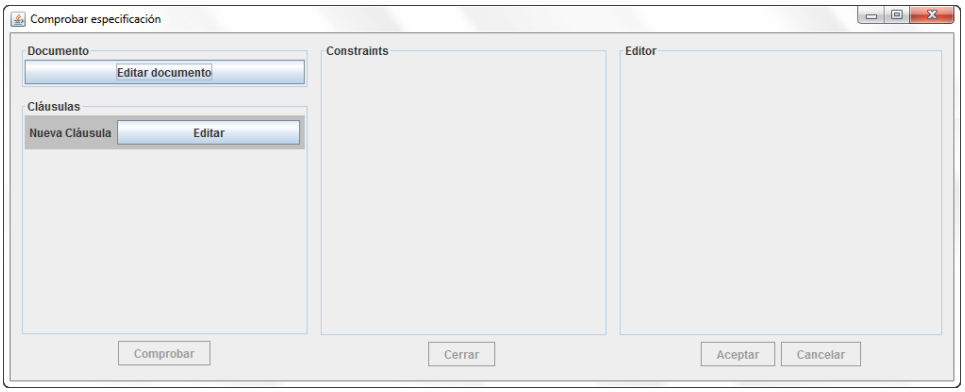

Figura A.11: Comprobar especificación con especificación vacía

Comprobar Especificación Esta opción cargará una pantalla de este caso de uso sin ninguna especificación guardada (ver figura A.11). Por lo tanto, habrá que introducir la especificación del mismo modo que en el procedimiento de refutación, y también el documento XML pulsando sobre el botón *Editar documento*. Para introducir el documento se utilizará el mismo editor que para los constraints.

Comprobar Especificación Actual Esta opción cargará una pantalla de este caso de uso pero con la misma especificación que se tenga en la pantalla principal. En la figura A.12 se muestra un ejemplo de esta opción: se ha cargado la especificación que había en la pantalla principal. Hay que tener en cuenta que la especificación de esta pantalla es independiente a la de la pantalla principal por lo que, si se modifica algo (añadiendo cláusulas, por ejemplo), esos cambios no se verán reflejados en la pantalla principal y viceversa.

Una vez introducida la especificación y el documento, se pulsará el boton *Comprobar* situado en la esquina inferior izquierda y el sistema devolverá el resultado: *true* si el documento es un modelo de la especificación, y *false* en

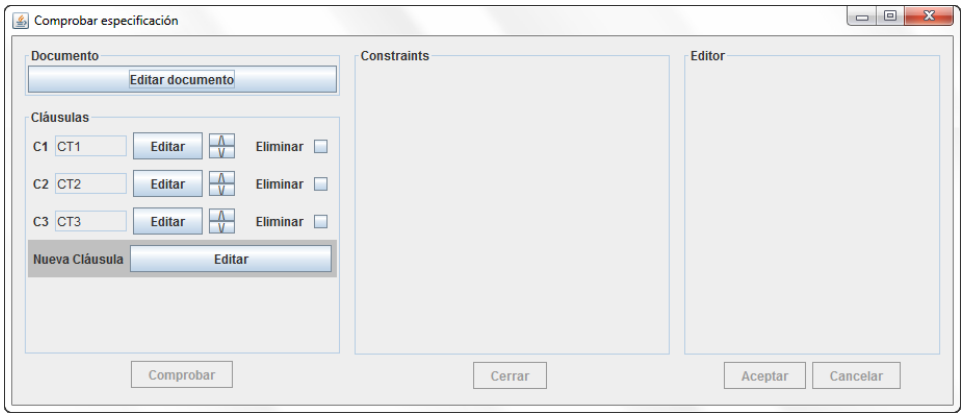

Figura A.12: Comprobar especificación con especificación actual

caso contrario.

A.3.6. Operación Monomorfismo

El modo de acceder a esta operación es seleccionando la opción *Monomorfismo* del menú *Herramientas*. La ventana que abre (figura A.13) es una muy parecida a aquella en la que se introducen los constraints condicionales. Si estamos calculando el monomorfismo de p_1 en p_2, en el editor superior izquierdo introduciremos el patrón p_1 y en el derecho el patrón p_2. A continuación se pulsará el botón *Generar monomorfismo* y el sistema calculará todos los monomorfismos posibles. Mediante los botones con forma de flecha se podrá cambiar de solución.

A.3.7. Operación $p_1 \otimes p_2$

Seleccionando la opción *Operar* en el menú *Herramientas* se abrirá la ventana de la figura A.14. En el editor superior izquierdo se introducirá el patrón p_1 y en el superior derecho el patrón p_2. Después se hará click sobre el botón *Operar* y el sistema calculará la solución. En este caso, todos los patrones que forman parte de la solución se mostrarán en el mismo editor, por lo que no hay que pulsar los botones con forma de flecha.

A.3.8. Operación $p_1 \otimes_{c,m} q$

Para poder utilizar esta herramienta, se seleccionará la opción *Operar con común* en el menú *Herramientas*. Esta herramienta consta de dos ventanas. La primera es en la que se introducirá el constraint condicional $p_2 \rightarrow q$ de la forma explicada en el apartado A.3.4.1. Después se pulsará el botón *Siguiente* y se pasará a la ventana de la figura A.15, donde en el editor superior izquierdo aparecerá el constraint condicional introducido en la ventana

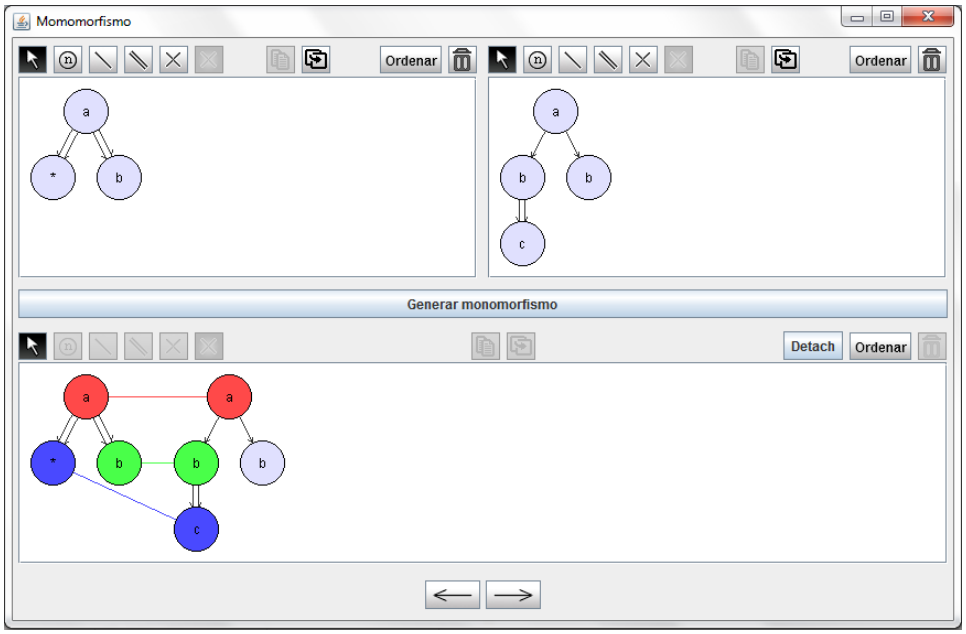

Figura A.13: Pantalla de la operación monomorfismo

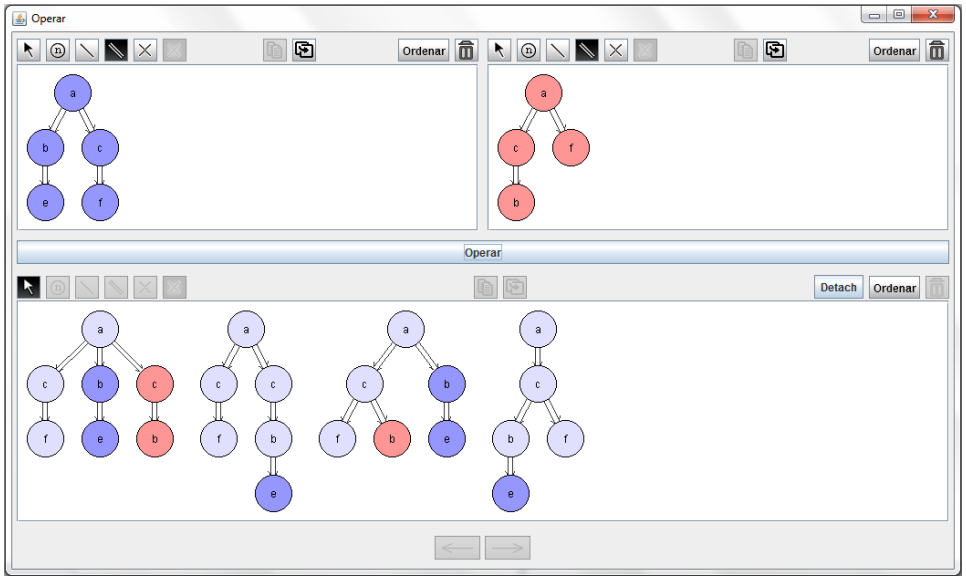

Figura A.14: Pantalla de la operación $p_1 \otimes p_2$

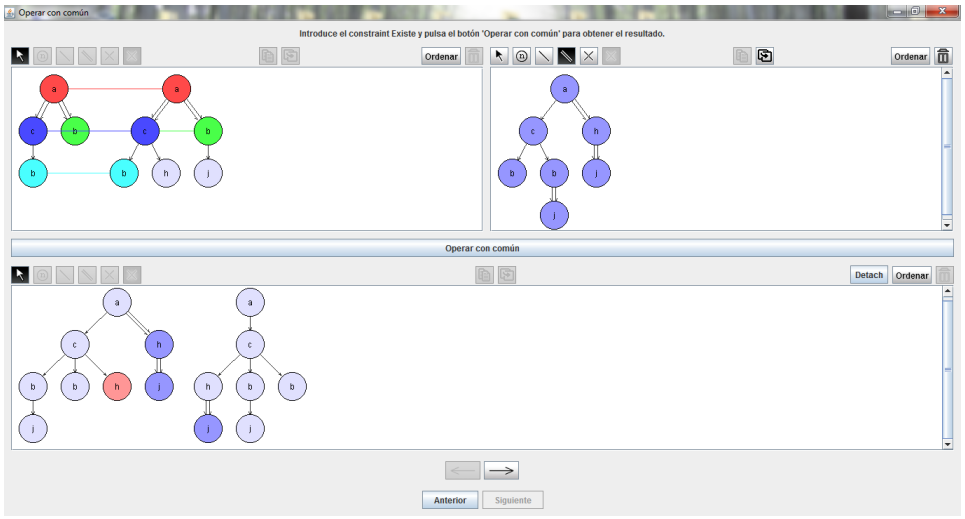

Figura A.15: Pantalla de la operación $p_1 \otimes_{c,m} q$

anterior, y los otros dos editores estarán vacíos. En el editor superior derecho se introducirá el patrón p_1 y, tras pulsar el botón *Operar con común*, se mostrarán las soluciones.

A.3.9. Menú Visualizar

Esta opción de la barra de menú (figura A.16) permite introducir un patrón, constraint o cláusula indicando dicho elemento con la notación de Prolog. El sistema analiza y traduce la notación de Prolog a Java y muestra al usuario el elemento introducido.

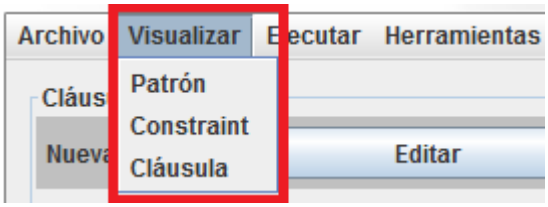

Figura A.16: Menú Visualizar

Anexo B

Seguimiento y Control

En este apéndice se comparará la planificación inicial con la verdadera trayectoria del proyecto. Además, en los casos en los que haya habido grandes diferencias, se explicarán las razones por las que han ocurrido dichas diferencias.

B.1. Retrasos

La mayor parte de los retrasos producidos en este proyecto han sido debido a la gran cantidad de trabajo ajeno al proyecto que ha aparecido. También hubo un importante retraso durante la implementación de las reglas Unfold, debido a que no se tuvo en cuenta en la planificación la complejidad de las mismas.

Sin embargo, a pesar de los retrasos, hubo algunas tareas que se hicieron en menor tiempo del planificado, por lo que los retrasos no tuvieron un impacto tan negativo como el que podrían haber tenido.

B.2. Tareas No Planificadas

El proyecto ha completado con éxito todas las tareas contempladas en la planificación inicial. Por ello, se decidió analizar una posible ampliación del alcance, resultando en una única tarea: el caso de uso *Comprobar Especificación*. Se decidió incluir esta tarea debido a que todos los procedimientos que utiliza ya estaban implementados, por lo que las ventajas que aportaría su inclusión en el proyecto eran muy grandes frente a sus prácticamente nulas desventajas.

B.3. Comparativa Planificación vs. Real

En este apartado se compararán las diferentes actividades planificadas con la verdadera inversión de tiempo en la realización de las mismas.

B.3.1. Gestión

En la tabla B.1 se puede ver la comparativa entre la planificación de la actividad **Gestión** y su realización.

Gestión	T. planificado	T. real	Desviación
Planificación			
Planificación inicial	15h	15h	0
Replanificaciones	5h	2h	-3h
Seguimiento y control			
Reuniones de seguimiento	20h	40h	+20h
Total	40h	57h	+17h

Tabla B.1: Comparativa de la actividad Gestión

La principal desviación de esta actividad ha sido en las reuniones de seguimiento. Debido a la complejidad de las reglas Unfold, en los meses de enero-abril hubo un aumento en la cantidad de reuniones de seguimiento. Sin embargo, no se considera que haya tenido un impacto negativo, ya que esas reuniones han contribuido a la buena finalización del proyecto.

B.3.2. Formación

En la tabla B.2 se puede ver la comparativa de la actividad **Formación**.

Formación	T. planificado	T. real	Desviación
Reuniones formativas	40h	40h	0
Total	40h	40h	0

Tabla B.2: Comparativa de la actividad Formación

Como se puede apreciar, no ha habido ninguna desviación en esta actividad.

B.3.3. Aplicación

En la tabla B.3 se puede ver la comparativa entre la planificación de la actividad **Aplicación** y el tiempo real dedicado a la misma.

La gran desviación de esta actividad ha sido el desarrollo del prototipo 2, el que incluía las reglas Unfold. En la planificación inicial no se había tenido en cuenta la posible dificultad que podían acarrear y se habían planificado muy pocas horas. Las tareas realizadas durante esta ampliación de tiempo incluyen no sólo la implementación del prototipo, sino también el análisis de las reglas Unfold, habiendo obtenido unas reglas equivalentes a las originales pero que simplifican mucho más la implementación del prototipo, por lo que

Aplicación	T. planificado	T. real	Desviación
Procedimiento de refutación			
Prototipo 1	120h	100h	-20h
Prototipo 2	60h	180h	+120h
Interfaz gráfica			
Diseño	10h	10h	0
Implementación	110h	90h	-20
Total	300h	380h	+80h

Tabla B.3: Tiempo estimado para Aplicación

el impacto negativo que ha supuesto este retraso viene compensado por la mejora cualitativa del prototipo.

Por otro lado, ha habido una ganancia de tiempo en las tareas *Prototipo 1* e *Implementación*. La primera, planificada para terminar a finales de noviembre, se terminó a mediados de octubre. La segunda, debido a los retrasos ocasionados por el prototipo 2, empezó a finales de marzo en lugar de a principios de febrero. Por lo tanto, aunque se dedicaron finalmente veinte horas menos para el desarrollo del mismo, se dedicaron con menos descanso entre las mismas para poder terminar en el plazo previsto, lo cual se logró.

B.3.4. Documentación

La tabla B.4 muestra la comparación entre el tiempo planificado y el tiempo real empleado para la actividad **Documentación**.

Documentación	T. planificado	T. real	Desviación
Memoria	30h	50h	+20h
Manual de conf. y uso	5h	5h	0
Código fuente de la aplicación	1h	1h	0
Total	36h	56h	+20h

Tabla B.4: Tiempo estimado para Documentación

B.3.5. Resumen

El total de 416 horas planificadas se han convertido en 533 horas invertidas. Teniendo en cuenta que se disponía de 34 horas extra que se iban a dedicar a retrasos, en lugar de un aumento de 117 horas, se puede considerar que la cantidad de horas de más que se ha necesitado ha sido de 83.

Bibliografía

[1] Clocksin, W. F., Mellish, C.S. Programming in Prolog. Berlin; Springer-Verlag (2003).

[2] Miklau, G., and Suciu, D. *Containment and equivalence for a fragment of XPath*, Journal of the ACM, Vol. 51, N.1, (2004) 2-45.

[3] Navarro, M., and Orejas, F. *Proving Satisfiability of Constraint Specifications on XML Documents* , Proceedings of X Jornadas sobre Programación y Lenguajes, CEDI2010, Valencia (2010).

[4] Orejas, F., Ehrig, H., and Prange, U. *A Logic of Graph Constraints*, Fundamental Approaches to Software Engineering, 11th Int. Conference, FASE 2008. LNCS 4961 (2008) 179-198.

[5] WORLD WIDE WEB CONSORTIUM. 2007. *XML path language (XPath) 2.0.*

[6] Configuración del Puente Java-Prolog `http://www.swi-prolog.org/packages/jpl/installation.html`

[7] Diagramas de clases mediante *Altova* `http://www.altova.com/es/`

[8] Dropbox: Almacenamiento en la nube `https://www.dropbox.com/`

[9] Entorno de programación Eclipse `http://www.eclipse.org/`

[10] Manual de LaTeX `http://es.wikibooks.org/wiki/Manual_de_LaTeX`

[11] Programación en Java: Arrastrar gráficos `http://www.chuidiang.com/java/graficos/arrastrar-grafico/arrastrar-grafico.php`

[12] Programación en Java: Explorador de archivos `http://docs.oracle.com/javase/7/docs/api/javax/swing/JFileChooser.html`

[13] Programación en Java: Imágenes `http://lefunes.wordpress.com/2009/01/29/cargando-imagenes-desde-java/`

[14] Programación en Java: Layouts http://docs.oracle.com/javase/tutorial/uiswing/layout/using.html

[15] Programación en Java: Scroll http://docs.oracle.com/javase/tutorial/uiswing/components/scrollpane.html#sizing

[16] Prolog en Eclipse http://sewiki.iai.uni-bonn.de/research/pdt/docs/download