Sistema de reescritura basado en soft sets para sistemas biológicos simbólicos modelados con Pathway Logic en Maude

Soft set based rewrite system for symbolic biological systems modeled with Pathway Logic in Maude

Trabajo de Fin de Máster
Curso 2020–2021

Autor
Rocío M. Santos Buitrago

Director
Adrián Riesco Rodríguez

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Sistema de reescritura basado en soft sets para sistemas biológicos simbólicos modelados con Pathway Logic en Maude

Soft set based rewrite system for symbolic biological systems modeled with Pathway Logic in Maude

Trabajo de Fin de Máster en Ingeniería Informática
Departamento de Sistemas Informáticos y Computación
(Software Systems and Computation)

Autor
Rocío M. Santos Buitrago

Director
Adrián Riesco Rodríguez

Convocatoria: *Febrero 2021*
Calificación: *XXXXXXXXXXX*

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

12 de enero de 2021

Autorización de difusión

El abajo firmante, matriculado en el Máster en Ingeniería en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Sistema de reescritura basado en soft sets para sistemas biológicos simbólicos modelados con Pathway Logic en Maude", realizado durante el curso académico 2020/2021 bajo la dirección de Adrián Riesco Rodríguez en el Departamento de de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Rocío M. Santos Buitrago

12 de junio de 2021

Dedicatoria

A mis padres, Beatriz y Gustavo, por su apoyo incondicional a lo largo de mi vida

Agradecimientos

Estudiar este máster ha sido un desafío importante en mi vida, que habría sido difícil de superar si no hubiera sido por el apoyo y el ánimo de mucha gente.

Quiero expresar mi agradecimiento, en primer lugar, a mi tutor, Adrián Riesco, por su guía, dedicación y apoyo en este trabajo de fin de máster: GRACIAS!!!

Estoy muy agradecida a todo el profesorado del máster, en especial en este último año, por su comprensión ante mis excepcionales circunstancias personales.

También quiero dar las gracias a Francisco J. López Fraguas, mi profesor de programación declarativa en el grado, y también tengo un agradecimiento especial a todos los miembros del grupo Maude en UCM. Tengo buenos recuerdos de la disposición y paciencia de Alberto Verdejo para resolver mis dudas. También muchas gracias a José Meseguer, como padre de Maude.

Por supuesto estaré siempre agradecida a Narciso Martí, por sus incontables y muy buenos consejos académicos y personales.

Por último, quiero agradecer a mi familia por su cariño y paciencia. A mis padres y mis abuelos por apoyar mis intereses académicos y por animarme a conseguir todos mis propósitos. A mis hermanos Bea, Marta, Patu y Gus, por estar siempre a mi lado y animarme en cada reto.

Resumen

Sistema de reescritura basado en soft sets para sistemas biológicos simbólicos modelados con Pathway Logic en Maude

Pathway Logic es una herramienta para tratar con sistemas biológicos simbólicos desarrollada en SRI International. Está basada en redes de Petri y en el lenguaje de reescritura Maude. Con esta herramienta se han desarrollado numerosos modelos de rutas de señalización celular. Estos modelos constituyen una base de conocimiento formal que contiene información sobre los cambios que ocurren en las proteínas dentro de una célula en respuesta a la exposición a ligandos/receptores, sustancias químicas o diversas tensiones. El objetivo principal de este trabajo consiste en desarrollar e implementar nuevas variantes de reescrituras basadas en soft sets en el contexto de Pathway Logic. La instrucción de reescritura estándar en Maude permite examinar los términos a través del árbol de reescrituras partiendo de un estado inicial. Para esa reescritura de términos, se proponen nuevas funciones de elección en el árbol de búsquedas con soft sets, que proporcionan ventajas sobre la toma de decisiones bajo información incompleta. La implementación propuesta se lleva a cabo con el lenguaje Maude y se utilizan las nuevas funcionalidades en el metaintérprete y la gestión de entrada/salida. Por último, este trabajo incluye un análisis de los resultados y del rendimiento del sistema propuesto respecto del sistema de reescritura estándar y de otros enfoques existentes en la literatura.

Palabras clave

Lógica de reescritura, Maude, Metalenguaje, Pathway Logic, Sistemas biológicos simbólicos, Soft sets, Información incompleta, Toma de decisiones, Sistema de búsqueda.

Abstract

Soft set based rewrite system for symbolic biological systems modeled with Pathway Logic in Maude

Pathway Logic is a tool for dealing with symbolic biological systems developed at SRI International. It is based on Petri nets and the Maude rewriting language. Numerous cellular signaling pathway models have been developed with this tool. These models constitute a formal knowledge base that contains information about the changes that occur in proteins within a cell in response to exposure to ligands/receptors, chemicals or various stresses. The main objective of this dissertation is to develop and implement new rewriting variants based on soft sets in the context of Pathway Logic. The standard rewrite instruction in Maude allows the examination of terms through the rewrite tree starting from an initial state. For this rewriting of terms, new choice functions are proposed in the search tree by using soft sets, which provide advantages over decision making under incomplete information. The proposed implementation is carried out with the Maude language and the new functionalities in the meta-interpreter and the input/output management are used. Finally, this dissertation includes an analysis of the results and performance of the proposed system with respect to the standard rewriting system and other existing approaches in the literature.

Keywords

Rewriting logic, Maude, Metalanguage, Pathway Logic, Symbolic biological systems, Soft sets, Incomplete information, Decision making, Search system.

Índice

Índice de figuras

Índice de tablas

Índice de listados

Capítulo 1

Introducción

"Bueno, algunas veces yo he creído hasta seis cosas imposibles antes del desayuno"
— Lewis Carroll, Alicia en el País de las Maravillas

Este capítulo comienza con una breve introducción al tema de este proyecto. A continuación se dedica una sección a los objetivos y al plan de trabajo. Por último, se describe la organización de la memoria.

1.1. Motivación

Muchos problemas de la vida real requieren el uso de datos imprecisos o desconocidos. Su análisis debe implicar la aplicación de principios matemáticos capaces de captar estas características. En el campo de la Inteligencia Artificial, la teoría de soft sets proporciona un marco de trabajo apropiado para la toma de decisiones en las situaciones de falta de información.

Pathway Logic es una herramienta diseñada para tratar con sistemas biológicos simbólicos. Con esta plataforma se han desarrollado numerosos modelos de rutas de señalización celular. Esta herramienta facilita la comprensión de los sistemas biológicos complejos y la verificación de hipótesis en el diseño de experimentos.

Una de las características del lenguaje Maude es su capacidad de poder expresar de forma natural una amplia gama de aplicaciones, por ejemplo, Pathway Logic. Las nuevas funcionalidades en el metaintérprete y la gestión de entrada/salida de la nueva versión permiten manejar de forma eficiente un entorno de ejecución dentro de Maude.

El sistema de reescritura estándar de Maude ofrece todos los términos alcanzables del árbol de búsqueda. En algunas aplicaciones es deseable elegir únicamente una opción entre todas las posibles. A partir de esta situación, en este proyecto se propone implementar un entorno de ejecución con diferentes estrategias en el sistema de reescritura que permitan escoger el mejor término reescrito. Teniendo en cuenta que la información disponible puede ser incompleta, se han definido las estrategias mediante la teoría de soft sets.

Con la intención de proporcionar una aplicación con utilidad real, la implementación de este sistema de reescritura se ha desarrollado con los modelos biológicos de Pathway Logic.

1.2. Objetivos

El objetivo principal de este proyecto consiste en desarrollar e implementar nuevas variantes de reescrituras basadas en soft sets en el contexto de Pathway Logic.

Las metas intermedias para alcanzar este objetivo se pueden especificar en:

1. Implementar una aplicación en Maude con su sistema de comandos propios.
2. Integrar la aplicación con Pathway Logic.
3. Definir estrategias para guiar la reescritura.

Los trabajos realizados se clasifican en las siguientes áreas:

- Lógica de reescritura.
- Soft computing, en especial la toma de decisiones bajo incertidumbre con soft sets.
- Modelización de sistemas biológicos basados en reglas.

1.3. Plan de trabajo

La realización del proyecto se ha apoyado en el plan de trabajo con el director. Las primeras reuniones se dedicaron a la concreción del tema y objetivos del proyecto. Después, cada dos semanas hemos tenido reuniones en las que se repasaban las tareas definidas en la reunión anterior. Por parte del director se realizaban correcciones y se proponían mejoras. En cada reunión se concretaba el trabajo a realizar durante las dos semanas siguientes.

Aparte de las reuniones ordinarias, las dudas puntuales sobre cualquier aspecto del trabajo se han resuelto por parte del director por correo electrónico. En todo momento, las respuestas han sido acertadas y las orientaciones valiosas.

Los hitos que se establecieron para alcanzar los objetivos son:

1. Búsqueda y definición inicial del tema del trabajo.
2. Profundizar en lenguaje Maude.
3. Investigación sobre Pathway Logic y soft sets.
4. Diseño e implementación de un prototipo básico de la aplicación.
5. Ampliación progresiva de las funcionalidades (tanto en los comandos como en las estrategias).
6. Realización de pruebas.
7. Redacción de la memoria del proyecto.

1.4. Organización de la memoria

La memoria está dividida en los siguientes capítulos:

- **Capítulo 1**: Introducción. Se introduce el tema del TFM, se describen los objetivos del trabajo y se detalla el plan de trabajo seguido para la consecución de los objetivos.
- **Capítulo 2**: Estado de la Cuestión. En este capítulo se introducen las áreas en las que se fundamenta el trabajo: el lenguaje de programación Maude, el metaintérprete de Maude, la teoría de soft sets y Pathway Logic.
- **Capítulo 3**: Descripción del Trabajo. En este capítulo se describe el trabajo realizado en el proyecto **PLSS**: **P**athway **L**ogic con **S**oft **S**ets.
- **Capítulo 4**: Análisis de Resultados. En este capítulo se incluyen algunos comentarios sobre el rendimiento de la aplicación y se realiza un análisis comparativo de los resultados obtenidos con los del uso estándar de la reescritura.
- **Capítulo 5**: Conclusiones y Trabajo Futuro. Por último, se han expuesto las conclusiones y las líneas para un trabajo futuro.

El código elaborado para este proyecto está disponible en el repositorio de GitHub `https://github.com/rsantosb/TFM-PLSS`, bajo la licencia MIT (`https://opensource.org/licenses/MIT`).

Capítulo 2

Estado de la cuestión

En este capítulo se introducen las cuatro áreas en las que se fundamenta el trabajo. La sección 2.1 trata del lenguaje de programación Maude. La sección 2.2 se refiere al metaintérprete de Maude. La sección 2.3 trata sobre la teoría de soft sets. Por último, la sección 2.4 introduce Pathway Logic, el entorno de trabajo que se ha utilizado en la modelización de los sistemas biológicos.

2.1. Maude

En esta sección se muestra la definición, una visión general de los objetivos, la filosofía de diseño, los fundamentos lógicos, las aplicaciones y la estructura general de Maude.

Maude es un lenguaje de programación para especificaciones formales mediante el uso de términos algebraicos. Se trata de un lenguaje interpretado que permite la verificación de propiedades y transformaciones sobre modelos y que permite también ejecutar la especificación como si fuera un prototipo (Clavel et al., 2020).

El diseño del lenguaje de Maude puede entenderse como un esfuerzo por maximizar simultáneamente tres dimensiones: sencillez, expresividad y rendimiento.

Una amplia gama de aplicaciones puede expresarse de forma natural: desde sistemas secuenciales deterministas hasta sistemas no deterministas altamente concurrentes; desde pequeñas aplicaciones hasta grandes sistemas; desde implementaciones concretas hasta especificaciones abstractas; pasando por frameworks lógicos en los que se pueden utilizar formalismos completos, otros lenguajes y otras lógicas.

Las implementaciones concretas deben producir un rendimiento del sistema competitivo con otros lenguajes de programación eficientes.

Los programas en Maude deben ser lo más simples posible y tener un significado claro. Las declaraciones de programación básicas de Maude son muy simples y fáciles de entender. Estas son ecuaciones y reglas, y en ambos casos tienen una semántica de reescritura simple donde las instancias del patrón del lado izquierdo se reemplazan por las instancias correspondientes del lado derecho.

Maude tiene una librería de módulos predefinidos que se cargan por defecto en el sistema al inicio de cada sesión. Estos módulos son `BOOL`, `INT`, `QID`, `QID-LIST`, `LOOP-MODE`,

`CONFIGURATION` y `META-LEVEL` (Clavel et al., 2020).

- El módulo `BOOL` define los valores booleanos `true` y `false` y los operadores booleanos usuales (e.g., igualdad, desigualdad, conjunción, disyunción, negación, etc.)
- El módulo `INT` proporciona el tipo `Int` de los números enteros, el tipo `NzInt` de los enteros distintos de cero y las operaciones aritméticas habituales para trabajar con los enteros.
- El módulo `QID` proporciona el tipo `Qid` de los identificadores con comilla, junto con operadores para estos identificadores, tales como la concatenación, la indexación de un identificador por un entero o la eliminación del primer carácter después de la comilla.

  También resulta útil disponer de un tipo de datos de listas de estos identificadores con comilla. El módulo `QID-LIST` extiende el módulo `QID` con el tipo `QidList` de las listas de identificadores con comilla.
- El módulo `LOOP-MODE` fue la herramienta principal para crear aplicaciones interactivas en las versiones anteriores de Maude. Aunque `LOOP-MODE` ha quedado obsoleto, se incluye en la versión actual a efectos de compatibilidad con versiones anteriores. La funcionalidad de `STD-STREAM`, que es más general y flexible, permite gestionar las entradas/salidas y mantener el estado persistente del entorno lingüístico o de la herramienta.
- El módulo `CONFIGURATION` proporciona tipos básicos y constructores para modelar sistemas basados en objetos. Una configuración es una sopa o conjunto múltiple de objetos y mensajes que representa un posible estado del sistema. Las clases necesarias para describir un sistema de objetos son `Object`, `Msg` y `Configuration`.
- El módulo `META-LEVEL` se comenta en la sección 2.2.

Un programa de Maude que contiene solo ecuaciones se denomina *módulo funcional*, es decir, define una o más funciones mediante ecuaciones, utilizadas como reglas de simplificación.

Por ejemplo, si construimos listas de identificadores (que son secuencias de caracteres que comienzan con el carácter "`'`" y pertenecen a la clase `Qid` con un operador "`.`" definido con notación infija:

```
op nil : -> List .
op _._ : Qid List -> List .
```

Entonces, por medio de operadores y ecuaciones, podemos definir una función de longitud de una lista y un predicado de pertenencia de un elemento a una lista:

```
vars I J : Qid .
var L : List .

op length : List -> Nat .
eq length(nil) = 0 .
eq length(I . L) = s length(L) .

op _in_ : Qid List -> Bool .
eq I in nil = false .
eq I in J . L = (I == J) or (I in L) .
```

donde: `s_` denota la función sucesora en números naturales, `_==_` es el predicado de igualdad en identificadores y `_or_` es la disyunción habitual en valores booleanos.

Las ecuaciones se especifican en Maude con la palabra clave `eq` y terminan con un punto. Se utilizan de izquierda a derecha como reglas de simplificación de ecuaciones.

Por ejemplo, si queremos evaluar la expresión:

```
length('a . 'b . 'c . nil)
```

podemos aplicar la segunda ecuación de `length` para simplificar la expresión tres veces, y luego aplicar la primera ecuación una vez para obtener el valor final `s s s 0` (es decir, 3):

```
length('a . 'b . 'c . nil)
= s length('b . 'c . nil)
= s s length('c . nil)
= s s s length(nil)
= s s s 0
```

Esta es la utilización estándar de *reemplazo de iguales por iguales* de ecuaciones en álgebra elemental y tiene una semántica muy clara y simple en la lógica de ecuaciones. El reemplazo de iguales por iguales se realiza aquí solo de izquierda a derecha y luego se invoca a la simplificación ecuacional o reescritura de ecuaciones.

Por supuesto, las ecuaciones de nuestro programa deben tener buenas propiedades como *reglas de simplificación* en el sentido de que su resultado final debe existir y ser único. Este es de hecho el caso de las dos definiciones funcionales anteriores.

En Maude, las ecuaciones pueden ser condicionales; es decir, solo se aplican si se cumple una determinada condición. Por ejemplo, podemos simplificar una fracción a su forma irreducible usando la ecuación condicional:

```
vars I J : NzInt .
ceq J / I = quot(J, gcd(J, I)) / quot(I, gcd(J, I))
    if gcd(J, I) > s 0 .
```

donde `ceq` es la palabra clave de Maude que introduce ecuaciones condicionales, `NzInt` es la clase para identificar los números enteros distintos de cero, y donde asumimos que las operaciones de cociente de enteros (`quot`) y máximo común divisor (`mcd`) ya han sido definidas por sus ecuaciones correspondientes.

Un programa de Maude que contiene reglas y posiblemente ecuaciones se denomina *módulo de sistema*. Las reglas también se calculan reescribiendo de izquierda a derecha, es decir, como las reglas de reescritura, pero no son ecuaciones; en cambio, se entienden como reglas de transición local en un sistema posiblemente concurrente.

Por ejemplo, consideremos un sistema bancario distribuido en el que visualizamos los objetos de la cuenta en una *sopa*, es decir, en un conjunto múltiple o bolsa de objetos y mensajes. Dichos objetos y mensajes pueden *flotar* en la sopa distribuida y pueden interactuar localmente entre sí de acuerdo con reglas de reescritura específicas.

De esta forma, podemos representar una cuenta bancaria como una estructura similar a un registro con el nombre o etiqueta del objeto, su nombre de clase cuenta (`Account`) y un atributo para el balance (`bal`) que se corresponde con un número natural. Los siguientes son dos objetos de cuentas diferentes en esta notación:

```
< 'A-001 : Account | bal : 200 >
< 'A-002 : Account | bal : 150 >
```

Las cuentas se pueden actualizar al recibir diferentes mensajes y cambiar su estado en consecuencia. Por ejemplo, podemos tener mensajes de débito y crédito, como:

```
credit('A-002, 50)
debit('A-001, 25)
```

Podemos pensar en la sopa como la yuxtaposición de objetos y mensajes con sintaxis vacía. Por ejemplo, los dos objetos y dos mensajes anteriores forman la siguiente sopa:

```
< 'A-001 : Account | bal : 200 >
< 'A-002 : Account | bal : 150 >
credit('A-002, 50)
debit('A-001, 25)
```

En una sopa, el orden de los objetos y mensajes es irrelevante. Las reglas de interacción local para cuentas de crédito y débito se expresan en Maude mediante las reglas de reescritura:

```
var I : Qid .
vars N M : Nat .

rl < I : Account | bal : M > credit(I, N)
=> < I : Account | bal : (M + N) > .

crl < I : Account | bal : M > debit(I, N)
=> < I : Account | bal : (M - N) >
    if M >= N .
```

donde las reglas se introducen con la palabra clave `rl` y las reglas condicionales con la palabra clave `crl`, como la regla anterior para débito que requiere que la cuenta tenga fondos suficientes.

Es importante tener en cuenta que estas reglas no son ecuaciones, son reglas de transición locales de un sistema bancario distribuido. Se pueden aplicar simultáneamente a diferentes fragmentos de la sopa. Por ejemplo, aplicando ambas reglas a la sopa anterior obtenemos un nuevo estado distribuido:

```
< 'A-001 : Account | bal : 175 >
< 'A-002 : Account | bal : 200 >
```

Hay que tener en cuenta que la reescritura realizada es una reescritura de varios conjuntos, de modo que, independientemente de dónde se coloquen los objetos de la cuenta y los mensajes en la sopa, siempre se pueden unir y reescribir si se aplica una regla.

En Maude, esta especificación corresponde a la parte ecuacional del programa, donde declaramos que el operador de unión de conjuntos múltiples con sintaxis vacía satisface las ecuaciones de asociatividad y commutatividad:

```
X (Y Z) = (X Y) Z
X Y = Y X
```

Esto no se hace dando explícitamente las ecuaciones anteriores. En su lugar, se declara el operador de unión de conjuntos múltiples con los atributos de ecuación `assoc` y `comm`, donde `Configuration` denota los conjuntos múltiples o sopas de objetos y mensajes:

```
op __ : Configuration Configuration -> Configuration [assoc comm] .
```

Luego, Maude utiliza esta información para generar un algoritmo de coincidencia de conjuntos múltiples (*multiset matching algorithm*), donde el operador de unión de múltiples conjuntos se empareja módulo asociatividad y commutatividad. Por tanto, un programa que contiene estas reglas de reescritura es intuitivamente muy simple y tiene una semántica de reescritura sencilla.

Por supuesto, los sistemas especificados por estas reglas pueden ser altamente concurrentes y no deterministas; es decir, a diferencia de las ecuaciones, no se supone que todas las secuencias de reescritura puedan conducir al mismo resultado.

Por ejemplo, dependiendo del orden en que se hayan tramitado los mensajes de débito o crédito, la cuenta bancaria puede terminar en estados bastante diferentes, ya que la regla de débito solo se puede aplicar si el saldo de la cuenta es lo suficientemente grande.

Además, algunos sistemas pueden no tener estados finales: su objetivo puede ser participar continuamente en interacciones con su entorno como sistemas reactivos.

2.1.1. Reflexión en Maude

Una característica importante de Maude es la reflexión. Intuitivamente, significa que los programas de Maude se pueden metarrepresentar como datos, que luego se pueden manipular y transformar mediante funciones apropiadas (Clavel et al., 2020).

También significa que existe una conexión causal sistemática entre los propios módulos de Maude y sus metarrepresentaciones, en el sentido de que primero podemos realizar un cálculo en un módulo y luego metarrepresentar su resultado, o equivalentemente podemos primero metarrepresentar el módulo y su estado inicial y luego realizar todo el cálculo en el metanivel.

El proceso de metarrepresentación en sí mismo puede repetirse dando lugar a una torre de reflexión muy útil. Gracias a la lógica de la semántica de Maude, esta torre es una forma precisa de reflexión lógica con una semántica bien definida.

Dado que las reglas de reescritura de un módulo de sistema pueden ser altamente no deterministas, puede haber muchas formas posibles de aplicarlas, lo que lleva a resultados bastante diferentes. En un sistema de objetos distribuidos, esto puede ser solo parte de la vida: siempre que se respeten algunos supuestos de equidad, cualquier ejecución simultánea puede ser aceptable.

La ejecución secuencial de Maude admite dos estrategias diferentes de ejecución *justa* de forma integrada a través de sus comandos `rewrite` y `frewrite`. Si queremos utilizar una estrategia diferente para una aplicación determinada, entonces debemos ejecutar los módulos de Maude en el metanivel con estrategias internas definidas por el usuario.

Las estrategias internas se pueden definir reescribiendo reglas en un módulo de metanivel que pueden guiar la aplicación, posiblemente no determinista, de las reglas en el

módulo dado a nivel de objeto. Este proceso puede repetirse en la torre de reflexión, es decir, podríamos definir meta-estrategias, meta-meta-estrategias, etc.

2.2. Metaintérprete y META-LEVEL en Maude

Conceptualmente, un metaintérprete es un objeto externo que es un intérprete independiente de Maude, con bases de datos de módulos y vistas, que envía y recibe mensajes. El módulo META-INTERPRETER del archivo meta-interpreter.maude contiene los comandos y mensajes de respuesta que cubren casi la totalidad del intérprete Maude.

Por ejemplo, puede construirse un metaintérprete para insertar o mostrar módulos y vistas, o realizar cálculos en un módulo con nombre.

Como respuesta, el metaintérprete responde con mensajes de reconocimiento de las operaciones realizadas o que contienen resultados. Los metaintérpretes pueden crearse y destruirse según sea necesario, y como un metaintérprete es un intérprete completo de Maude, puede albergar a los propios metaintérpretes y así sucesivamente en una torre de reflexión.

Además, el módulo funcional original de META-LEVEL puede utilizarse por sí mismo desde el interior de un metaintérprete después de ser insertado.

Internamente, las tripas de la implementación del intérprete Maude están encapsuladas en una clase C++ llamada Interpreter y el intérprete de alto nivel con el que se interactúa en la línea de comandos es una instancia de esta clase junto con una pequeña cantidad de código de cola que le permite comunicarse a través de los flujos de entrada/salida estándar. Los metaintérpretes también son instancias de esta clase, con una pequeña cantidad de código adjunto que les permite intercambiar mensajes con un contexto de ejecución de reescritura orientado a objetos. Actualmente, tanto el intérprete a nivel de objeto como cualquier metaintérprete existente ejecutan todos el mismo proceso en un solo hilo, y el flujo de control se gestiona a través del mecanismo de reescritura orientado a objetos.

La metarrepresentación de términos, módulos y puntos de vista se comparte con el módulo funcional META-LEVEL. La API para los metaintérpretes definida en el módulo META-INTERPRETER incluye varios tipos y constructores, un identificador de objeto integrado interpreterManager y una gran colección de comandos y mensajes de respuesta.

En el módulo META-INTERPRETER todos los mensajes siguen el formato estándar de los mensajes Maude, donde los dos primeros argumentos los identificadores de objeto del objetivo y del remitente. El identificador de objeto interpreterManager se refiere a un objeto externo especial que se encarga de crear nuevos metaintérpretes en el contexto de ejecución actual. Estos metaintérpretes tienen identificadores de objeto de la forma interpreter(n) para cualquier número natural n.

El módulo META-LEVEL tiene varias funciones de descenso incorporadas que proporcionan formas útiles y eficientes de reducir los cálculos de metanivel a nivel de objeto, de ahí la relación entre ellos.

Este módulo es puramente funcional, esto se debe a que todas sus funciones de descenso son deterministas, aunque puedan manipular entidades intrínsecamente no deterministas como las teorías de reescritura.

La mayoría de las operaciones de este módulo son parciales, es decir, una función cuya aplicación a algunos argumentos tiene un tipo, pero no siempre, por lo que debe considerarse indefinida para esos argumentos. En lo que respecta a estos, no se trata de errores,

sino de excepciones o resultados fuera de los límites para los que existe una semántica definida.

Para representar este tipo de operaciones se utiliza la flecha (`~>`) en la declaración del operador correspondiente.

Dado que la naturaleza de `META-LEVEL` es puramente funcional, aunque sea muy potente, significa que no tiene noción de estado.

Los metaintérpretes de Maude permiten tipos de interacciones reflexivas muy versátiles en las que los intérpretes de Maude están encapsulados como objetos externos y pueden interactuar reflexivamente tanto con otros intérpretes como con otros objetos externos, incluido el usuario.

Por lo tanto, las aplicaciones reflexivas en las que la interacción del usuario implican un cambio de estado es esencial que requieran el uso de `META-LEVEL` para apoyar dicha interacción.

En el módulo `META-LEVEL` se introducen una serie de operaciones para proporcionar los cálculos de las reducciones. Dependiendo de su uso se pueden utilizar, por ejemplo, para moverse entre nivel de objeto y metanivel, para la simplificación y la reescritura, para la aplicación de reglas y para poder realizar matching, búsquedas, o reescritura usando estrategias.

En las siguientes subsecciones se describen algunas de las operaciones que se utilizan en el presente trabajo.

2.2.1. Operación `metaParse`

La operación `metaParse` refleja el comando `parse` de Maude.

```
op metaParse : Module QidList Type? ~> ResultPair? [special (...)] .
```

Esta función analiza si la lista de tokens (`QidList`) se corresponde con el tipo (`Type?`), respecto del módulo recibido en el primer argumento. La constante `anyType` permite cualquier miembro.

Si `metaParse` tiene éxito, la función devuelve la metarrepresentación del término analizado con su clase o tipo correspondiente. En caso contrario, devuelve:

- `noParse(`n`)` cuando no hubo análisis. El valor n es el índice del primer token incorrecto (contando desde 0), o el número de tokens en el caso de un final de entrada inesperado. Es del tipo `ResultPair?`.
- `ambiguity(`$r1$`,`$r2$`)` cuando existen varios análisis. Los valores $r1$ y $r2$ son los pares de resultados correspondientes a dos análisis distintos.

A continuación, se muestran dos ejemplos de esta operación con el módulo `VENDING-MACHINE` tomados del manual de Maude (Clavel et al., 2020):

```
Maude>  reduce in META-LEVEL : metaParse(upModule('VENDING-MACHINE, false), '
    $, 'Coin) .
result ResultPair: {'$.Coin,'Coin}
```

```
Maude> reduce in META-LEVEL : metaParse(upModule('VENDING-MACHINE, false), '$
    'q 'd 'q, 'Coin) .
result ResultPair?: noParse(2)
```

En el primer ejemplo, identifica el token `$` como un elemento del tipo `Coin`. Sin embargo, en el segundo ejemplo, la función `metaParse` nos devuelve `noParse(2)` porque el tercer elemento `d` no pertenece al tipo `Coin`.

2.2.2. Operaciones `upTerm` y `downTerm`

Estas funciones son fundamentales en el uso de la reflexión con Maude:

- La función `upTerm` toma un término `t` y devuelve la metarrepresentación de su forma canónica.
- La función `downTerm` toma la metarrepresentación de un término `t` como primer argumento y un término `t'` como segundo argumento. La función devuelve:
  - La forma canónica de `t`, si `t` es un término del mismo tipo que `t'`.
  - En caso contrario, devuelve la forma canónica de `t'`.

Esta es la sintaxis de los comandos `upTerm` y `downTerm`:

```
op upTerm : Universal -> Term [poly (1) special (...)] .
op downTerm : Term Universal -> Universal [poly (2 0) special (...)] .
```

A continuación se muestra un ejemplo de estas funciones basado en un sencillo módulo funcional `UP-DOWN-TEST`:

```
fmod UP-DOWN-TEST is
 protecting META-LEVEL .
 sort Foo .
 ops a b c d : -> Foo .
 op f : Foo Foo -> Foo .
 op error : -> [Foo] .
 eq c = d .
endfm
```

En primer lugar, se utiliza la función `upTerm` para obtener la metarrepresentación del término `f(a, f(b, c))` en el módulo `UP-DOWN-TEST`:

```
Maude> reduce in UP-DOWN-TEST : upTerm(f(a, f(b, c))) .
result GroundTerm: 'f['a.Foo,'f['b.Foo,'d.Foo]]
```

En este ejemplo se observa que el argumento dado ha sido reducido antes de obtener su metarrepresentación, en concreto, el subtérmino `c` se ha convertido en `d`.

En el siguiente ejemplo, la función `downTerm` recupera el término `f(a, f(b, c))` a partir de su metarrepresentación:

```
Maude> reduce in UP-DOWN-TEST : downTerm('f['a.Foo,'f['b.Foo,'c.Foo]], error).
result Foo: f(a, f(b, d))
```

Por último, se muestra que ocurre con el resultado de `downTerm` cuando su primer argumento no corresponde a la metarrepresentación de un término en el módulo:

```
Maude> reduce in UP-DOWN-TEST : downTerm('f['a.Foo,'f['b.Foo,'e.Foo]], error).
Advisory: could not find a constant e of sort Foo in meta-module UP-DOWN-TEST.
result [Foo]: error
```

Se ha utilizado la constante `e` en el término metarrepresentado, que no se corresponde con una constante declarada en el módulo.

Existen algunas otras funciones para moverse en la torre de reflexión, entre los niveles de objeto y metanivel (por ejemplo, `upModule`, `upView`).

2.2.3. Operación `metaReduce`

La operación parcial `metaReduce` se encarga de reducir un término con la simplificación ecuacional. Su sintaxis es la siguiente:

```
op metaReduce : Module Term ~> ResultPair [special (...)] .
```

Toma como argumentos la metarrepresentación de un módulo `R` y la metarrepresentación de un término `t`. Cuando `t` es un término de `R`, `metaReduce(R,t)` devuelve la metarrepresentación de la forma canónica de `t`, con el uso de las ecuaciones del módulo `R`, junto con la metarrepresentación de su correspondiente clase. La estrategia de reducción que utiliza `metaReduce` es similar a la empleada por el comando `reduce`.

Cuando alguno de los argumentos que recibe no es correcto, la operación `metaReduce` es indefinida, es decir, el término no se reduce y no se evalúa a un término de tipo `ResultPair`.

2.2.4. Operación `metaSearch`

La operación parcial `metaSearch` realiza una búsqueda en la que los términos se pueden reescribir con reglas, que en nuestro programa `PLSS` serán del tipo soft reglas. La estrategia de búsqueda utilizada por `metaSearch` coincide con la del comando `search` a nivel de objeto.

La operación `metaSearch` toma como argumentos la metarrepresentación de un módulo (`Module`), la metarrepresentación del término inicial de la búsqueda (`Term`), la metarrepresentación del patrón a buscar (`Term`), la metarrepresentación de una condición a satisfacer (`Condition`), la metarrepresentación del tipo de búsqueda a realizar (`Qid`), un valor (`Bound`) y un número natural (`Nat`):

```
op metaSearch : Module Term Term Condition Qid Bound Nat ~> ResultTriple? [
    special (...)] .
```

Los valores `Qid` permiten definir los tipos de búsqueda con los argumentos similares a los tipos de búsqueda empleados por el comando `search`:

- `'*` para una búsqueda que implique cero o más reescrituras (`=>*`).
- `'+` para una búsqueda que consta de una o más reescrituras (`=>+`).

- `'!` para una búsqueda que solo coincide con formas canónicas (`=>!`).

El argumento `Bound` indica la profundidad máxima de la búsqueda, y el argumento `Nat` es el número de solución.

2.2.5. Operación `metaNarrowingSearch`

La función `metaNarrowingSearch` se encarga de realizar la búsqueda narrowing o con estrechamiento, en el cual se analizan todos los cómputos de un conjunto de estados gracias a la descripción de los mismos mediante técnicas simbólicas (Clavel et al., 2020; Escobar et al., 2005). La sintaxis de esta función es:

```
op metaNarrowingSearch : Module Term Term Qid Bound Qid Nat ->
    NarrowingSearchResult? [special ...] .
```

Los argumentos de entrada en `metaNarrowingSearch` son:

- La metarrepresentación del módulo (`Module`); después el término inicial (primer `Term`) o estado inicial del que se parte y el término del patrón a alcanzar (segundo `Term`).
- El tipo de búsqueda (`Qid`), en nuestro caso es `'*`. A continuación, `Bound` representa la longitud máxima de narrowing, `unbounded` para nuestro programa;
- El siguiente argumento `Qid` puede ser `'none` o `'match`. La constante `'none` significa que se aplica una reducción estándar sin ningún plegado (como con el comando `vu-narrow`). Cuando se utiliza la constante `'match`, se aplica la reducción de plegado (como con el comando `fvu-narrow`).
- Por último, el argumento `Nat` se corresponde con la solución elegida. De esta forma, se pueden proporcionar todas las soluciones de forma secuencial, como hacen otros comandos de metanivel en Maude.

La función devuelve la solución del tipo `NarrowingSearchResult` cuando la búsqueda es exitosa. Si no hubo resultado, la función devuelve `failure`.

2.2.6. Ejemplo de las muñecas rusas

Con este breve ejemplo del módulo `RUSSIAN-DOLLS` tomado de Clavel et al. (2020) se comprueba la flexibilidad y generalidad de los metaintérpretes. Este realiza un cálculo en un metaintérprete que a su vez existe en una torre de metaintérpretes anidada a una profundidad definible por el usuario. Tan solo requiere dos ecuaciones y dos reglas.

Listing 2.1: Módulo `RUSSIAN-DOLLS`

```
mod RUSSIAN-DOLLS is
  extending META-INTERPRETER .

  op me : -> Oid .
  op User : -> Cid .
  op depth:_ : Nat -> Attribute .
  op computation:_ : Term -> Attribute .
```

```
    vars X Y Z : Oid .
    var AS : AttributeSet .
    var N : Nat .
    var T : Term .

    op newMetaState : Nat Term -> Term .
    eq newMetaState(0, T) = T .
    eq newMetaState(s N, T)
      = upTerm( <>
          < me : User | depth: N, computation: T >
          createInterpreter(interpreterManager, me, none)) .

    rl < X : User | AS >
       createdInterpreter(X, Y, Z)
    => < X : User | AS >
       insertModule(Z, X, upModule('RUSSIAN-DOLLS, true)) .

    rl < X : User | depth: N, computation: T, AS >
       insertedModule(X, Y)
    => < X : User | AS >
       erewriteTerm(Y, X, unbounded, 1, 'RUSSIAN-DOLLS, newMetaState(N, T)) .
endm
```

El estado visible del cálculo reside en un objeto de identificador `me` de la clase `Oid` y `User` de la clase `Cid`. El objeto tiene dos valores en sus respectivos atributos: la profundidad (`depth`) del metaintérprete, que se registra como `Nat`, con `0` como nivel superior, y el cómputo (`computation`) a realizar, que se registra como `Term`.

El operador `newMetaState` toma una profundidad y un metatérmino para evaluar. Si la profundidad es cero, entonces simplemente devuelve el metatérmino como el nuevo `metastate`; de lo contrario, se crea una nueva configuración, que consiste en un portal (necesario para reescribir con objetos externos, para localizar dónde salen y entran en la configuración los mensajes intercambiados con los objetos externos), el objeto visible por el usuario que contiene la profundidad disminuida y el cálculo, y un mensaje dirigido al objeto externo `interpreterManager`, solicitando la creación de un nuevo metaintérprete. Esta configuración se eleva al metanivel utilizando el operador `upTerm` importado del metanivel funcional.

La primera regla del módulo `RUSSIAN-DOLLS` maneja el mensaje: `createInterpreter` del `interpreterManager`, que recibe como argumento el identificador de objeto del metaintérprete recién creado. Utiliza la función `upModule` para elevar su propio módulo, `RUSSIAN-DOLLS`, al metanivel y envía una petición para insertar este meta-módulo en el nuevo metaintérprete.

La segunda regla maneja el mensaje `insertedModule` del nuevo metaintérprete. Llama al operador `newMetaState` para crear un nuevo metaestado y luego envía una solicitud al nuevo metaintérprete para realizar un número no acotado de reescrituras, con soporte de objetos externos y una reescritura por ubicación por recorrido en la copia de metanivel del módulo `RUSSIAN-DOLLS` que se acaba de insertar.

Comenzamos el cálculo con el comando de `erewrite` en una configuración que consiste en un portal, un objeto de usuario y un mensaje `createInterpreter`.

En este caso el objeto de usuario tiene como atributos la profundidad `0` y el cómputo a evaluar de la metarrepresentación de `2 + 2`:

```
Maude> erewrite
  <>
  < me : User | depth: 0, computation: ('_+_['s_^2['0.Zero], 's_^2['0.Zero]])>
```

```
  createInterpreter(interpreterManager, me, none) .

result Configuration:
<>
< me : User | none >
erewroteTerm(me, interpreter(0), 1, 's_^4['0.Zero], 'NzNat)
```

Como se puede ver en `result Configuration`, el resultado es `Zero` que es la evaluación de la meta-representación del cálculo directamente en un metaintérprete, sin anidar.

Para la siguiente ejecución realizaremos el mismo cálculo pero cambiaremos la profundidad a `1`:

```
Maude> erewrite
  <>
  < me : User | depth: 1, computation: ('_+_ ['s_^2['0.Zero],'s_^2['0.Zero]])>
  createInterpreter(interpreterManager, me, none) .

result Configuration:
<>
< me : User | none >
erewroteTerm(me, interpreter(0), 5,
    '__['<>.Portal,
        '<_:_|_>['me.Oid,'User.Cid,'none.AttributeSet],
        'erewroteTerm['me.Oid,'interpreter['0.Zero],'s_['0.Zero],
          '_`[_`]['`'s_^4.Sort,'`'0.Zero.Constant],'`'NzNat.Sort]], '
              Configuration)
```

Pasando a una profundidad de `1` resulta un cambio y se transforma en un metaintérprete anidado. El mensaje de respuesta de nivel superior `erewroteTerm(...)` contiene un resultado que es una metaconfiguración, que contiene el metamensaje de respuesta del metaintérprete interno.

2.3. Soft sets

Muchos problemas de la vida real requieren el uso de datos imprecisos o inciertos. Su análisis debe implicar la aplicación de principios matemáticos capaces de captar estas características. La teoría de conjuntos difusos (fuzzy sets) supuso un cambio paradigmático en las matemáticas al permitir un grado de pertenencia parcial. Existe una vasta literatura sobre los conjuntos difusos y sus aplicaciones desde la publicación del artículo de Zadeh (1965).

De las generalizaciones de los fuzzy sets nos interesa especialmente la aplicación de la teoría de los conjuntos blandos (soft sets) y sus extensiones a los problemas de la toma de decisiones. Los soft sets fueron introducidos por Molodtsov (1999). Algunas referencias relevantes del desarrollo de su teoría se deben a: Aktaş y Çağman (2007), Alcantud (2016a,b), Ali et al. (2009) y Maji et al. (2003). Ali et al. (2015) definen soft sets ordenados en red para situaciones en las que existe algún orden entre los elementos del conjunto de parámetros. Qin et al. (2013) combinan los conjuntos de intervalos y los soft sets y Zhang (2014) estudia los interval soft sets y sus aplicaciones. Maji et al. (2001) introducen los fuzzy soft sets. Wang et al. (2014) introducen los hesitant fuzzy soft sets. Han et al. (2014), Qin et al. (2011), y Zou y Xiao (2008) se ocupan de los soft sets incompletos. También hay interesantes modelos híbridos en la literatura reciente.

2.3.1. Soft sets incompletos

Se adopta la descripción y terminología habitual para los soft sets y sus extensiones: el conjunto U denota el universo de objetos y el conjunto E denota el conjunto universal de parámetros.

Definición 1 (Molodtsov (1999)) *Un par (F, A) es un soft set sobre U cuando $A \subseteq E$ y $F : A \longrightarrow \mathcal{P}(U)$, donde $\mathcal{P}(U)$ denota el conjunto de todos los subconjuntos de U.*

Un soft set sobre U es considerado como una familia de subconjuntos parametrizados del universo U, siendo el conjunto A los parámetros. Para cada parámetro $e \in A$, $F(e)$ es el subconjunto de U aproximado por e o el conjunto de elementos e-aproximados del soft set. Muchos investigadores han desarrollado esta noción y definen otros conceptos relacionados (Maji et al., 2003; Feng y Li, 2013). Para modelar situaciones cada vez más generales, se ha propuesto la siguiente definición de soft sets incompletos.

Definición 2 (Han et al. (2014)) *Un par (F, A) es un soft set incompleto sobre U cuando $A \subseteq E$ y $F : A \longrightarrow \{0, 1, *\}^U$, donde $\{0, 1, *\}^U$ es el conjunto de todas las funciones de U a $\{0, 1, *\}$.*

Obviamente, todo soft set puede considerarse un soft set incompleto. El símbolo $*$ en la definición 2 permite capturar la *falta de información*: cuando $F(e)(u) = *$ interpretamos que se desconoce si u pertenece al subconjunto de U aproximado por e. Como en el caso de los soft sets, cuando $F(e)(u) = 1$ (resp., $F(e)(u) = 0$), interpretamos que u pertenece (resp., no pertenece) al subconjunto de U aproximado por e.

Cuando los conjuntos U y A son finitos, los soft sets y los soft sets incompletos pueden representarse por matrices o en forma tabular. Las filas se corresponden con objetos en

U y las columnas se corresponden con parámetros en A. En el caso de un soft set, estas representaciones son binarias (es decir, todos los elementos o celdas son 0 ó 1).

El siguiente ejemplo de la práctica real ilustra un soft set incompleto. Después lo utilizamos para explicar los fundamentos de toma de decisiones en términos prácticos.

Ejemplo 1 *Sea* $U = \{h_1, h_2, h_3\}$ *el universo de casas y* $E_0 = \{e_1, e_2, e_3, e_4\}$ *el conjunto de parámetros (atributos o características de la casa). La siguiente información define un soft set incompleto* (F_0, E_0)*:*

(a) $h_1 \in F_0(e_1) \cap F_0(e_3)$ *y* $h_1 \notin F_0(e_4)$*, pero se desconoce si* $h_1 \in F_0(e_2)$ *o no.*

(b) $h_2 \in F_0(e_2)$ *y* $h_2 \notin F_0(e_3) \cup F_0(e_4)$*, pero se desconoce si* $h_2 \in F_0(e_1)$ *o no.*

(c) $h_3 \in F_0(e_1) \cap F_0(e_4)$ *and* $h_3 \notin F_0(e_2) \cup F_0(e_3)$*.*

(d) $h_4 \notin F_0(e_1) \cup F_0(e_2) \cup F_0(e_4)$*, pero se desconoce si* $h_4 \in F_0(e_3)$ *o no.*

La tabla 2.1 captura la información que define (F_0, E_0). De esta forma, se obtiene la representación tabular del soft set incompleto (F_0, E_0).

	e_1	e_2	e_3	e_4
h_1	1	$*$	1	0
h_2	$*$	1	0	0
h_3	1	0	0	1
h_4	0	0	$*$	0

Tabla 2.1: Representación tabular del soft set incompleto (F_0, E_0) definido en el ejemplo 1.

2.3.2. Toma de decisiones con soft sets

Maji et al. (2002) fueron pioneros en la toma de decisiones basadas en soft sets. Establecieron el criterio de que un objeto puede ser seleccionado si maximiza el valor de elección del problema. En relación con esto, Zou y Xiao (2008) argumentaron que en el proceso de recopilación de datos puede haber datos desconocidos, imprecisos o inexistentes. Por lo tanto, se deben tener en cuenta los soft sets estándar bajo información incompleta, lo que exige la inspección de soft sets incompletos.

Cuando un soft set (F, A) se representa en forma matricial a través de la matriz $(t_{ij})_{k\times l}$, donde k y l son los cardinales de U y A respectivamente, entonces el valor de elección (o *choice value*) de un objeto $h_i \in U$ es $c_i = \sum_j t_{ij}$. Se hace una elección adecuada cuando el objeto seleccionado h_k verifica $c_k = \max_i c_i$. En otras palabras, los objetos que maximizan el valor de elección son los resultados satisfactorios de este problema de decisión.

En lo que respecta a la toma de decisiones incompleta basada en soft sets, los enfoques más utilizados son los de Zou y Xiao (2008), Qin et al. (2011), Han et al. (2014) y Alcantud y Santos-García (2017). Examinemos las ideas de sus métodos:

(a) Zou y Xiao (2008) iniciaron el análisis de soft sets y fuzzy soft sets bajo información incompleta. En el primer caso, proponen calcular todos los valores de elección posibles para cada objeto, y luego calcular sus respectivos valores de decisión d_i por el método

del promedio ponderado. Para ello, el peso de cada valor de elección posible se calcula con la información completa existente. En particular, proponen algunos indicadores sencillos que pueden utilizarse eventualmente para priorizar las alternativas, a saber, $c_{i(0)}$ (el valor de elección si se supone que todos los datos que faltan son 0), $c_{i(1)}$ (el valor de elección si se supone que todos los datos que faltan son 1) y d_{i-p} (el valor de elección corresponde a $(c_{i(0)} + c_{i(1)})/2$).

(b) Inspirándose en el enfoque de análisis de datos de Zou y Xiao, Qin et al. (2011) proponen una nueva forma de completar los datos que faltan en un soft set incompleto. Para ello, introducen la relación entre los parámetros. Así pues, dan prioridad a la asociación entre parámetros antes que a la probabilidad de que aparezcan objetos en $F(e_i)$. De esta manera, adjuntan un soft set completo con cualquier soft set incompleto. Sin embargo, el procedimiento de Qin et al. (2011) presupone que hay asociaciones entre algunos de los parámetros. En su propuesta, cuando no se alcanza un umbral dado exógenamente, los datos se rellenan según el enfoque de Zou y Xiao.

Qin et al. indican que su procedimiento puede utilizarse para implementar aplicaciones que impliquen soft sets incompletos, pero no hacen ninguna declaración explícita en cuanto a la toma de decisiones. No obstante, parece apropiado complementar su procedimiento de llenado con una priorización de los objetos de acuerdo con sus valores de elección Q_i, como es frecuente en la toma de decisiones basada en los soft sets.

(c) Han et al. (2014) explican que su método es bueno cuando los objetos en U están relacionados entre sí. Estos autores desarrollan y comparan varios criterios de obtención para la toma de decisiones de soft sets incompletos que se generan por intersección restringida.

(d) Por último, Alcantud y Santos-García (2017) proponen considerar todas las formas posibles de "completar" un soft set incompleto y tomar la decisión a partir de estas soluciones completadas. Este sistema es apropiado cuando no se tiene información a priori sobre la relación entre los objetos y los parámetros.

Para explicar la idea de este método, la aplicamos sobre el ejemplo 1 y obtenemos los valores de elección s_i de cada opción. Si se tienen en cuenta todas las posibilidades, en última instancia uno de las cuatro tablas representadas en la tabla 2.2 contiene la información completa que se necesita para tomar la decisión. En esta tabla hemos eliminado el objeto h_4 debido al cribado de dominación previo que se explica a continuación. Como no sabemos cuál será la correcta, debemos asumir que todas estas tablas son equiparables según el principio de indiferencia de Laplace. Por lo tanto, es sensato calcular qué objetos deben ser seleccionados de acuerdo con la toma de decisiones en cada uno de estos casos, y luego seleccionar el objeto que sea óptimo en la mayoría de los casos.

El investigador puede llevar a cabo una operación de cribado antes de seleccionar una opción final. En primer lugar, calculamos el valor máximo c_0 de todos los valores de elección $c_{j(0)}$ a través de las opciones u_j. Si este valor es estrictamente mayor que el valor de elección $c_{k(1)}$ de una alternativa u_k, esta alternativa puede ser eliminada de la matriz/tabla inicial. La razón es que si se supone que todos los datos que faltan para el u_k son 1, hay otra opción i que verifica que cuando todos los datos que faltan para u_i se suponen que son 0, la opción i sigue teniendo un valor de elección mayor que la opción k. Este argumento sugiere la siguiente definición de dominancia entre opciones.

C_{v_1} matrix

	e_1	e_2	e_3	e_4	c_i
h_1	1	0	1	0	2
h_2	0	1	0	0	1
h_3	1	0	0	1	2

C_{v_2} matrix

	e_1	e_2	e_3	e_4	c_i
h_1	1	0	1	0	2
h_2	1	1	0	0	2
h_3	1	0	0	1	2

C_{v_3} matrix

	e_1	e_2	e_3	e_4	c_i
h_1	1	1	1	0	3
h_2	0	1	0	0	1
h_3	1	0	0	1	2

C_{v_4} matrix

	e_1	e_2	e_3	e_4	c_i
h_1	1	1	1	0	3
h_2	1	1	0	0	2
h_3	1	0	0	1	2

Tabla 2.2: Las cuatro tablas completadas para el soft set incompleto (F_0, E_0) según el paso 4 de nuestro algoritmo, con los respectivos valores de elección para cada alternativa.

Definición 3 (Alcantud y Santos-García (2017)) *Sea (F, A) un soft set incompleto sobre U. Una opción i domina una opción k cuando $c_{k(1)} < c_{i(0)}$.*

Claramente, si empleamos cualquier solución basada en el valor de la elección, podemos descartar libremente las opciones dominadas. Por ejemplo, si utilizamos d_j, $c_{j(0)}$, $c_{j(1)}$ ó d_{j-p} como indicador de cualquier opción j, la opción k no puede maximizar el indicador seleccionado cuando la opción i lo domina. Esta simplificación es básicamente intrascendente en el caso de las soluciones de Zou y Xiao, sin embargo, podemos aplicarla en otros algoritmos computacionalmente costosos para reducir los cálculos.

Siguiendo con el ejemplo 1, observamos que las casas 1 y 3 dominan la casa 4, por lo que h_4 se elimina y queda una tabla de 3×4. En dicha tabla recortada tenemos $w = 2$, y enumeramos las celdas con valor $*$ como $((1,2),(2,1))$.

Por cada $v \in \{0,1\}^w = \{v_1 = (0,0), v_2 = (0,1), v_3 = (1,0), v_4 = (1,1)\}$ surge una tabla factible completada. Estas cuatro tablas están representadas en la tabla 2.2, junto con los valores de elección de las casas de cada tabla. Observamos que h_1 adjunta el valor de elección más alto en todas estas cuatro tablas, h_2 adjunta el valor de elección más alto sólo en C_{v_2}, y h_3 adjunta el valor de elección más alto exactamente en C_{v_1} y C_{v_2}.

	s_i	d_i	d_{i-p}	$c_{i(0)}$	$c_{i(1)}$	Q_i
h_1	1,00	2,50	2,50	2	3	3
h_2	0,25	2,00	1,50	1	2	1
h_3	0,50	2,00	2,00	2	2	2
h_4	0	0,33	0,5	0	1	1
Óptimo	$\{h_1\}$	$\{h_1\}$	$\{h_1\}$	$\{h_1, h_3\}$	$\{h_1\}$	$\{h_1\}$

Tabla 2.3: Soluciones para el problema representado por (F_0, E_0) en ejemplo 1 según varios indicadores.

La tabla 2.3 contiene los indicadores de las propuestas de solución que hemos mencionado aplicados al ejemplo 1. También se representan las alternativas óptimas para cada procedimiento. Además, observe que el hecho de que las casas 1 y 3 dominen la casa 4 se deriva de la tabla 2.3, al comparar el máximo de la columna $c_{i(0)}$—que se alcanza en 1 y 3—y los valores de la columna $c_{i(1)}$ que son estrictamente menores que dicho máximo.

2.4. Sistemas biológicos simbólicos y Pathway Logic

El lenguaje Maude proporciona numerosas herramientas de análisis para teorías de reescritura: cálculo de reescrituras, búsqueda en amplitud, verificación de modelos o model checking en la lógica temporal lineal, demostrador inductivo de teoremas y muchos otros. Con la utilización de estas funcionalidades es posible estudiar el comportamiento de los sistema biológicos, para comprobar si es posible alcanzar un cierto estado desde el estado inicial y analizar si el sistema verifica algunas propiedades temporales.

La idea de transición entre estados permite modelar los sistemas biológicos con la lógica de reescritura de una manera muy natural: mientras que las células son un conjunto de multiconjuntos que representan las diferentes componentes que aparecen en una célula real, las reacciones bioquímicas se representan por medio de reglas de reescritura (Bernardo et al., 2008).

Pathway Logic es una técnica de análisis cualitativo basada en la lógica de reescritura (Talcott, 2008). Pathway Logic se utiliza para modelar y analizar procesos biológicos, como la transducción de señales, las redes metabólicas o la señalización de células del sistema inmunológico. Los modelos de Pathway Logic se representan y analizan con la utilización del sistema Maude (Clavel et al., 2007; Talcott, 2016). Los modelos pueden analizarse directamente por ejecución, búsqueda y verificación de modelos (Talcott y Dill, 2006; Talcott et al., 2003; Talcott y Knapp, 2017). Actualmente las capacidades de Pathway Logic incluyen:

1. Modelos con diferentes niveles de detalle. Las moléculas biológicas, sus estados, sus localizaciones y sus roles en los procesos moleculares o celulares pueden ser modelados con diferentes niveles de abstracción. Por ejemplo, una proteína de señalización compleja puede ser modelada ya sea de acuerdo a un estado general, a sus modificaciones postraduccionales o como un conjunto de dominios funcionales de la proteína y sus interacciones internas o externas (Eker et al., 2002b).

2. Rutas generadas dinámicamente con el uso de búsquedas y verificación de modelos. Dada una especificación de un sistema concurrente, esta se puede: ejecutar para encontrar un comportamiento posible; utilizar la búsqueda para comprobar si se puede alcanzar un estado que cumpla una condición determinada; o la verificación del modelo para ver si se satisface una propiedad temporal; y, cuando no se cumple la propiedad, se puede mostrar el contraejemplo (Knapp et al., 2005).

3. Transformación a redes de Petri para análisis y visualización. Pathway Logic Assistant es un software Java que implementa una herramienta gráfica de Pathway Logic. Pathway Logic Assistant proporciona una representación visual interactiva de los modelos de Pathway Logic y, entre otras, facilita las siguientes tareas: muestra la red de reacciones de señalización para una determinada placa de preparación (o dish); formula y envía consultas para encontrar y comparar rutas; o calcula y muestra la subred descendente de una o más proteínas (Talcott y Dill, 2005). Dado un estado inicial, Pathway Logic Assistant selecciona las reglas correspondientes del conjunto de reglas y representa la red de reacciones resultante como una red de Petri. De esta forma, se consigue una representación gráfica natural y se consiguen algoritmos eficientes para responder a las consultas.

2.4.1. Modelos en Pathway Logic: ecuaciones

Para ilustrar cómo Pathway Logic puede tratar las rutas de señalización, se muestra a continuación un modelo abreviado de transducción de señales intracelulares. Una base de conocimiento formal contendrá la información sobre los cambios que ocurren en las proteínas dentro de una célula en respuesta a la exposición a ligandos receptores, sustancias químicas y otros elementos.

TGFB1 (Factor de crecimiento transformante beta 1) es uno de los modelos implementados en Pathway Logic. El modelo TGFB1 contiene un total de 57 reglas y 968 datums. La evidencia experimental de cada regla se suministra en forma de datums. Cada datum representa el resultado de un experimento publicado en una revista especializada (Talcott, 2008). Las reglas y las evidencias forman parte del modelo de estímulos que se puede descargar del sitio web de Pathway Logic (`http://pl.csl.sri.com`).

La figura 2.1 muestra la complejidad de las reacciones que tienen lugar en la ruta de señalización TGF-β.

Los modelos de Pathway Logic están estructurados en cuatro capas: clases y operaciones, componentes, reglas y consultas. En primer lugar, la capa de clases y operaciones declara las principales relaciones de clases y subordinación, constituye el análogo lógico de la ontología. De esta forma, analizaremos de arriba a abajo los aspectos involucrados en un modelo de Pathway Logic. Las placas de preparación o dishes se definen como envoltorios de múltiples conjuntos o términos de la clase sopa (`Soup`). Los elementos de una sopa son

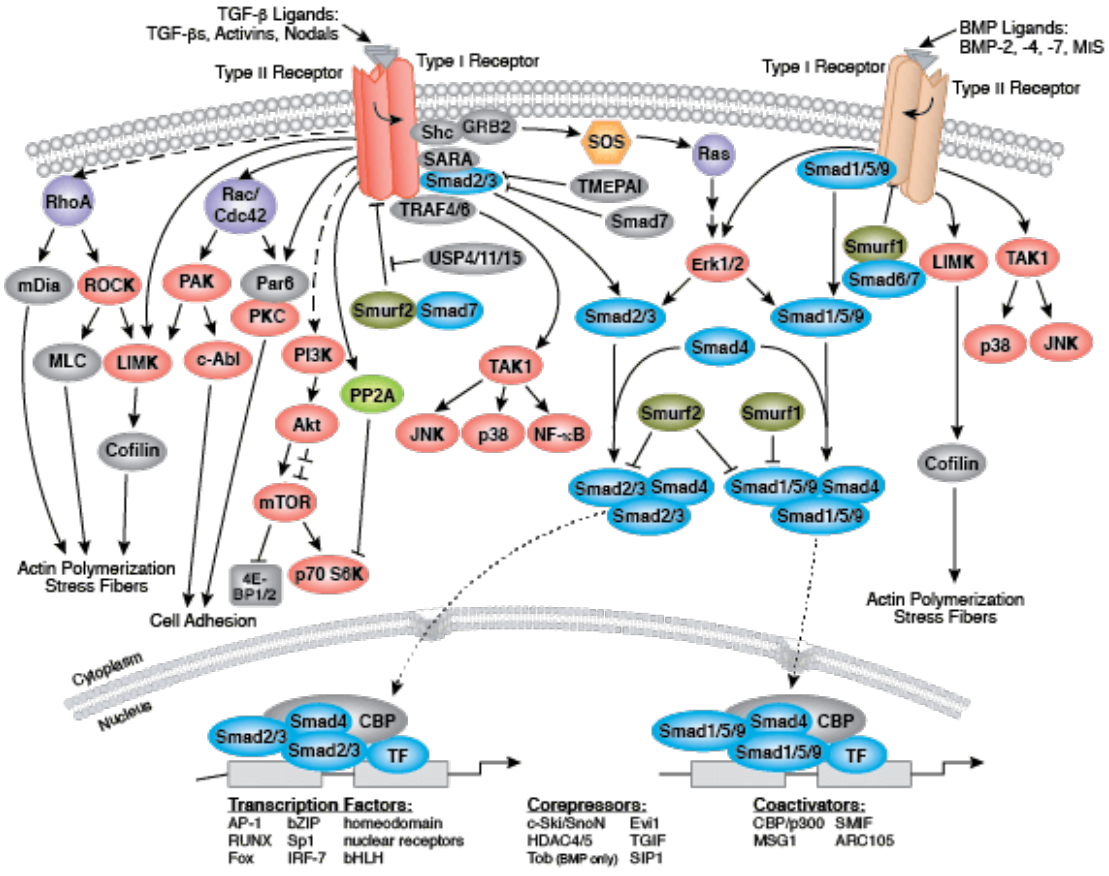

Figura 2.1: Ruta de señalización TGF-β. Fuente: Cell Signaling Technology.

las diferentes partes o localizaciones de la célula con sus respectivos contenidos. En Maude, el operador de PD se define como:

```
op PD : Soup -> Dish [ctor] .
```

De esta manera, el operador de PD se aplica a un conjunto múltiple de partes de la célula y se obtiene un elemento de la clase Dish. Por ejemplo, el siguiente código en Maude define una placa con dos localizaciones: el núcleo (NUc) que contiene las proteínas Rb1, Myc, y Tp53; y la membrana celular (CLm) que contiene las proteínas EgfR y PIP2:

```
PD( {NUc | Rb1 Myc Tp53}  {CLm | EgfR PIP2} ) .
```

En un segundo nivel, se define la sopa de localizaciones:

Listing 2.2: Definición en Maude de la sopa de localizaciones

```
sorts MtSoup Soup .
subsort MtSoup < Soup .
op empty : -> MtSoup [ctor] .
op __ : Soup Soup -> Soup [ctor assoc comm id: empty] .
```

Es decir, se definen las clases de Dish y MtSoup y además MtSoup es una subclase de la clase Soup. La sopa vacía se define con el operador constante empty. Por último, el operador __ define un conjunto múltiple de entidades no ordenadas (en términos matemáticos, una lista asociativa y conmutativa de elementos cuyo elemento neutro es la lista vacía empty). Un ejemplo de un término de la clase Soup es:

```
{NUc | Rb1 Myc Tp53}  {CLc | Erks Erk1}  {CLm | EgfR PIP2}
```

donde hay un conjunto de tres localizaciones (NUc, CLc y CLm) con sus respectivos contenidos. En cuanto a la definición de cada localización, se utiliza la clase Location para especificar los elementos en las diferentes localizaciones de la célula:

```
op {_|_} : LocName Soup -> Location [ctor] .
```

El operador {_|_} tiene dos argumentos: el identificador de la localización y su contenido (es decir, una sopa de elementos como proteínas, sustancias químicas y genes). Los diferentes elementos pueden estar contenidos en diferentes partes o ubicaciones de la célula: fuera de la célula (XOut), dentro o a través de la membrana celular (CLm), adheridos al interior de la membrana celular (CLi), en el citoplasma (CLc) y en el núcleo (NUc). Podemos indicar que el núcleo NUc contiene el gen Tp53-gene (la transcripción del gen está on) y las proteínas Rb1, Myc, Tp53 y NProteasome:

```
{NUc | [Tp53-gene - on] Rb1 Myc Tp53 NProteasome}
```

En la definición del operador {_|_}, hemos visto que se recibe un término del tipo Soup como segundo argumento. En este caso, esta sopa es una lista del contenido de esa parte o localización de la célula. Finalmente, señalamos que cada uno de los elementos de una sopa puede tener modificaciones. Por ejemplo, el término [Rac1 - GDP] indica que la proteína Rac1 se une al guanosín difosfato (GDP).

La clase Modification se utiliza para representar una modificación de la proteína post-

traducción (por ejemplo, activación, unión, fosforilación). Las modificaciones en Maude se aplican utilizando el operador `[_-_]`.

```
op [_-_] : Protein ModSet -> Protein  [right id: none ] .
```

Las modificaciones son un conjunto de modificaciones individuales que pertenecen a la clase `ModSet`. Un conjunto de modificaciones se define en Maude de forma análoga a las sopas definidas anteriormente:

Listing 2.3: Conjunto de modificaciones en Maude

```
sorts Site  Modification ModSet .
subsort Modification < ModSet .
op none : -> ModSet .
op __ : ModSet ModSet -> ModSet [assoc comm id: none] .
```

Hay numerosas modificaciones posibles: `acetyl!site` (acetilado en un sitio específico), `act` (activado), `degraded` (degradado), `dimer` (dimerizado), `GDP` (ligado al GDP), `GTP` (ligado al GTP), `K48ubiq` (ligado covalentemente a la ubiquitina polimerizada mediante enlaces K48), `K63ubiq` (ligado covalentemente a la ubiquitina polimerizada mediante enlaces K63), `p50` (un producto de separación de 50kD), `phos` (fosforilado), `phos!` (fosforilado en un sitio específico), `sumo` (sumoilado), `ubiq` (ubiquitado), `Yphos` (fosforilado en tirosina), `off` (no transcribe el ARNm) y `on` (transcribe el ARNm) (Talcott, 2016).

En el lenguaje Maude, cada una de estas modificaciones se define de esta manera:

Listing 2.4: Definición de las modificaciones en Maude

```
op act : -> ACT [ctor metadata "((description \"activated\") (abbrev +))"] .
op phos : -> AAMOD [ctor metadata "((term \"phosphorylated residue\")
    (abbrev p))"] .
op ubiq : -> AAMOD [ctor metadata "((term \"ubiquitinylated on lysine\")
    (abbrev ub))"] .
op GDP : -> SMBIND [ctor metadata "((term \"Guanosine 5'-diphosphate\")
    (abbrev GDP))"] .
op GTP : -> SMBIND [ctor metadata "((term \"Guanosine 5'-triphosphate\")
    (abbrev GTP))"] .
```

donde `ACT`, `AAMOD`, y `SMBIND` son subclases de la clase `Modification`. En las opciones del operador, una palabra clave `metadata` permite incluir meta-información adicional sobre un modificador.

La figura 2.2 muestra una representación esquemática de una célula muy simple. Diferentes elementos aparecen en diferentes partes o localizaciones de la célula: fuera de la célula (`XOut`), dentro o a través de la membrana celular (`CLm`), adheridos al interior de la membrana celular (`CLi`), en el citoplasma (`CLc`) y en el núcleo (`NUc`). Se representan algunas proteínas: el factor de crecimiento epidérmico (`Egf`), la cinasa PI3 (`Pi3k`), la cinasa activadora ERK 1 (`Mek1`), etc. Algunas componentes aparecen con distintos modificadores: activación (`act`), fosforilación sobre la tirosina (`Yphos`), y unión al GDP (`GDP`). Esta célula se representada en Maude con el siguiente `SmallDish`:

Listing 2.5: Placa de preparación `SmallDish`

```
eq SmallDish =
  PD( {XOut | Egf}  {CLi | Pi3k [Cdc42 - GDP]}
```

```
{NUc | Rb1 Myc Tp53}
{CLc | [Mek1 - act] [Ilk - act] Erks Erk1}
{CLm | EgfR PIP2 [Gab1 - Yphos]}) .
```

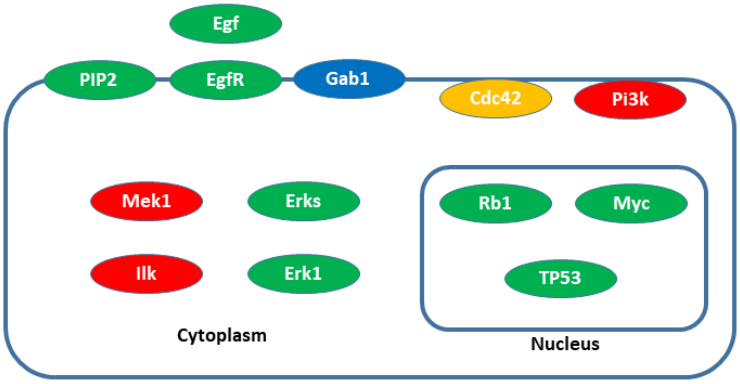

Figura 2.2: Representación esquemática de una célula. Las proteínas activadas están marcadas en rojo, las fosforiladas en azul y las unidas a GDP en amarillo. Las proteínas que no tienen modificaciones se muestran en verde.

Con la ayuda de Pathway Logic Assistant, la figura 2.3 muestra la representación neta de Petri de la ruta de señalización. Los rectángulos son transiciones (reacciones bioquímicas) y los óvalos son ocurrencias (entidades biológicas) en las que las ocurrencias iniciales son más oscuras. Los reactivos de una regla son las ocurrencias conectadas a la regla por flechas desde la ocurrencia a la regla. Los productos de una regla son las ocurrencias conectadas a la regla por flechas desde la regla hasta la ocurrencia. Las flechas punteadas indican una ocurrencia que es tanto de entrada como de salida. Por ejemplo, observamos en esta figura que la proteína `Jak1` (en el citoplasma) y la proteína transmembrana `Gp130` (en la localización `GP130C`) intervienen como reactivos en la reacción/regla `1229c`. El resultado de esta reacción es que la proteína `Gp130` no cambia y `Jak1` se mueve desde el citoplasma a la ubicación de `GP130C`.

Por último, detallamos la codificación completa de un ejemplo específico de la placa de preparación `Tgfb1Dish`:

Listing 2.6: Dish `Tgfb1Dish`

```
op Tgfb1Dish : -> Dish .
eq Tgfb1Dish = PD( {XOut | Tgfb1} {Tgfb1RC | TgfbR1 TgfbR2} {CLo | empty}
  {CLm | empty} {CLi | [Cdc42 - GDP] [Hras - GDP] [Rac1 - GDP] }
  {CLc | Abl1 Akt1 Atf2 Erks Fak1 Jnks Mekk1 Mlk3 P38s Pak2 Pml Smad2 Smad3
      Smad4 Smurf1 Smurf2 Tab1 Tab2 Tab3 Tak1 Traf6 Zfyve16}
  {NUc | Ctdsp1 Ets1 Smad7 Cdc6-gene Cdkn1a-gene Cdkn2b-gene Col1a1-gene
      Col3a1-gene  Ctgf-gene Fn1-gene Mmp2-gene Pai1-gene Smad6-gene Smad7-
      gene Tgfb1-gene Timp1-gene Cst6-gene Dst-gene Mmp9-gene Mylk-gene Pthlh
      -gene Gfi1-gene Csrp2-gene RoRc-gene}) .
```

En este dish se define un estado inicial (llamado `Tgfb1Dish`) con varias localizaciones y elementos:

- el exterior (localización `XOut`) que contiene el factor de crecimiento transformante beta1 (`Tgfb1`);

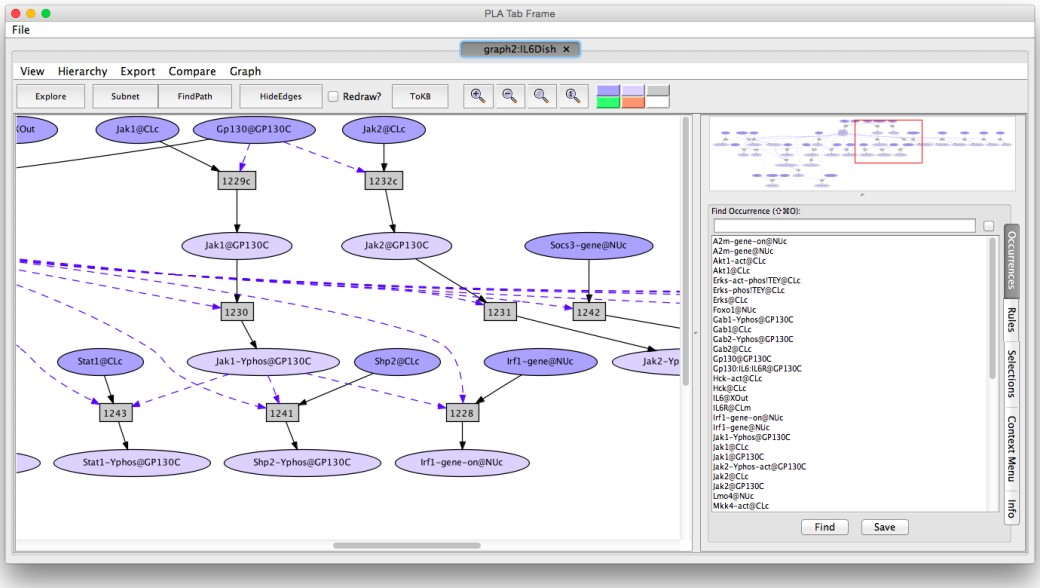

Figura 2.3: Vista general de una ruta de señalización con Pathway Logic Assistant.

- la localización `Tgfb1RC` que contiene el receptor beta del factor de crecimiento transformante I y II (`TgfbR1` y `TgfbR2`);
- la localización `CLo`, que contiene los elementos adheridos al exterior de la membrana de plasma, está vacía;
- la membrana (localización con la etiqueta `CLm`) también está vacía;
- el interior de la membrana (localización con la etiqueta `CLi`) contiene tres proteínas unidas a GDP: `Cdc42`, `Hras` y `Rac1`;
- el citoplasma (localización con la etiqueta `CLc`) contiene las proteínas `Abl1`, `Akt1`, `Atf2`, `Erks`, etc.; y
- el núcleo (localización con la etiqueta `NUc`) contiene varios genes (`Smad7`, `Tgfb1`, `Cst6`, etc.) y proteínas (`Ctdsp1`, `Ets1`, etc.).

2.4.2. Modelos en Pathway Logic: reglas de reescritura

Las reglas de reescritura describen el comportamiento de las proteínas y otros componentes dependiendo de los estados de modificación y los contextos biológicos. Cada regla representa un paso en un proceso biológico como las reacciones metabólicas o las reacciones de señalización intracelular o intercelular (Eker et al., 2002a, 2003; Santos-Buitrago et al., 2017; Talcott, 2006).

El conjunto de reglas de transición se construye a partir de los hallazgos experimentales publicados en revistas prestigiosas. Nakao et al. (1997) determinan el comportamiento de las señales del TGF-β desde la membrana al núcleo a través de los receptores de serina/-treonina cinasa y sus efectores posteriores, denominados proteínas SMAD.

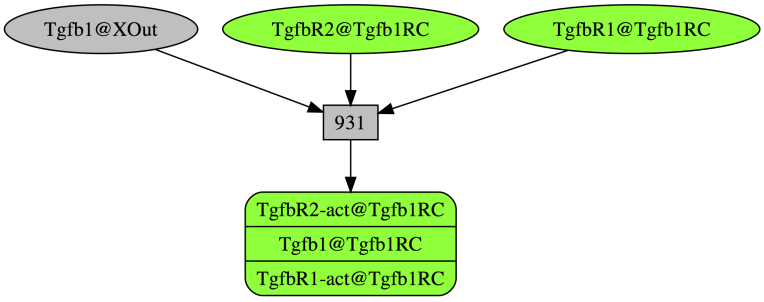

Figura 2.4: Representación esquemática de la regla de reescritura `931.TgfbR1.TgfbR2.by.Tgfb1` en Pathway Logic Assistant.

Esta regla de reescritura 931 establece: *en presencia del receptor I del factor de crecimiento transformante beta* ***`Tgfb1`*** *en el exterior de la célula (****`XOut`****), los receptores* ***`TgfbR1`*** *y* ***`TgfbR2`*** *se activan (****`TgfbR1-act`*** *y* ***`TgfbR2-act`****) y se unen entre sí y a* ***`Tgfb1`*** *(****`[TgfbR1 - act] : [TgfbR2 - act] : Tgfb1`****).* En la sintaxis de Maude, este proceso de señalización se expresa mediante la siguiente regla de reescritura:

Listing 2.7: Regla de reescritura `931.TgfbR1.TgfbR2.by.Tgfb1`

```
rl [931.TgfbR1.TgfbR2.by.Tgfb1]:
   {XOut | xout Tgfb1 }
   {Tgfb1RC | tgfb1rc TgfbR1 TgfbR2 }
=> {XOut | xout }
   {Tgfb1RC | tgfb1rc
   ([TgfbR1 - act] : [TgfbR2 - act] : Tgfb1) } .
```

Capítulo 3

Descripción del trabajo

En este capítulo se describe el trabajo realizado en el proyecto **PLSS**: Pathway Logic con Soft Sets.

Como se ha descrito en la sección 2.4, Pathway Logic es una herramienta diseñada para tratar con sistemas biológicos simbólicos desarrollada en SRI International (Talcott, 2008). Está basada en las redes de Petri y en el lenguaje de reescritura Maude.

Pathway Logic dispone de varios modelos con diferentes niveles de detalle y permite la generación dinámica de rutas de señalización mediante búsquedas y verificación de modelos. Este entorno de modelización permite además la transformación de sus modelos a redes de Petri para análisis y visualización. La herramienta de visualización se llama *Pathway Logic Assistant* (Talcott y Dill, 2005). Esta herramienta gráfica permite diferentes vistas de las rutas de señalización generadas dinámicamente.

Se han desarrollado numerosos modelos de rutas de señalización celular con esta herramienta. El modelo STM7 es una base de conocimiento formal que contiene información sobre los cambios que ocurren en las proteínas dentro de una célula en respuesta a la exposición a ligandos/receptores, sustancias químicas o diversas tensiones. Para simplificar, STM7 se divide en 32 mapas, cada uno de los cuales representa un estímulo. Estos mapas se llaman platos o dishes porque describen un estado inicial que corresponde al estado de las células en un plato de cultivo al comienzo de un experimento, más un estímulo. En nuestro caso, se utiliza el mapa que corresponde al factor de crecimiento transformante beta 1 (Tgfb1).

Este trabajo pretende extender el modelo STM7, de forma que permita trabajar con datos imprecisos o desconocidos. Según se ha descrito en la sección 2.3, la teoría de conjuntos difusos (fuzzy sets) supuso un cambio paradigmático en las matemáticas al permitir un grado de pertenencia parcial (Zadeh, 1965). En nuestro proyecto se emplea la teoría de los conjuntos blandos (soft sets), una generalización de los fuzzy sets que ha demostrado ser una herramienta útil para el problema de la toma de decisiones bajo situaciones de imprecisión o falta de información.

Con una base de conocimiento de Pathway Logic, en este trabajo se pretende implementar una variante de reescritura basada en soft sets. El comando `search` estándar en Maude permite realizar búsquedas a través del árbol de reescrituras partiendo de un estado inicial. La implementación propuesta se realizará con ayuda del metaintérprete de Maude y

proporcionará las ventajas de los soft sets sobre la toma de decisiones bajo información incompleta. En las secciones 2.1 y 2.2, se comentaron brevemente los aspectos fundamentales del lenguaje de reescritura Maude y su metaintérprete (Clavel et al., 2020).

La implementación de este proyecto se puede dividir en los siguientes bloques:

1. Importación y adaptación de los módulos de Tgfb1 procedentes del modelo STM7 de Pathway Logic.
2. Implementación en Maude de los soft sets y desarrollo de los operadores y funciones asociadas.
3. Definición de *softdish*, que será un plato o dish de Pathway Logic con unos atributos que permitirán establecer una elección entre las distintas reglas de ejecución basadas en soft sets.
4. Elaboración de un entorno de ejecución para PLSS utilizando la gestión de la entrada/salida y el metaintérprete.
5. Implementación con el metaintérprete de los comandos de simplificación con ecuaciones y reglas *soft*.
6. Implementación con el metaintérprete de otros comandos: carga de ficheros de usuario con comandos, sistema de ayuda y salida del programa.
7. Integración de todas las partes en el proyecto PLSS.

En las secciones que vienen a continuación se desarrollan cada uno de los bloques enumerados[1]. Después se muestra el procedimiento de instalación y la ejecución del programa con algunos ejemplos.

3.1. Módulos de Pathway Logic

En este apartado se aborda la importación y adaptación de los módulos de Tgfb1 en el modelo STM7 de Pathway Logic.

De acuerdo con la sección 2.4, en el módulo `QQ` se definen la sintaxis y la semántica del modelo Tgfb1:

Listing 3.1: Módulo `QQ`

```
mod QQ is
 inc ALLRULES .
 inc ALLOPS .
 inc SOFT-SET-FUN .
 inc TGFB1DISH .
endm
```

[1]El código completo del proyecto está disponible en `https://github.com/rsantosb/TFM-PLSS`. En la presente memoria se incluyen únicamente los fragmentos de código necesarios para la explicación del diseño y desarrollo del programa.

Este módulo importa los siguientes módulos: (1) `ALLOPS` que especifica los operadores que definen los elementos y componentes de la célula; (2) `TGFB1DISH` que establece el dish o estado inicial en este modelo; (3) `ALLRULES` que incluye todas las reglas de reescritura que gobiernan la dinámica de la célula; y (4) `SOFT-SET-FUN` que define las estrategias de decisión posibles con el soft set utilizado.

El módulo `ALLOPS` a su vez incluye los módulos en los que se definen las constantes para los químicos, genes y proteínas que pueden existir en la célula y también sus modificaciones (`MODIFICATIONOPS`). El listado 3.2 muestra el código de este módulo.

El módulo `ALLOPS` también importa el módulo `LOCATIONOPS` donde se definen las localizaciones de la célula. Por ejemplo, se define la constante `NUc` que representa el núcleo de la célula. El último módulo que incluye es `THEOPS`, en el cual se establece toda la sintaxis de la célula.

Listing 3.2: Módulo `ALLOPS`

```
fmod ALLOPS is
 inc CHEMICALOPS .
 inc GENEOPS .
 inc LOCATIONOPS .
 inc MODIFICATIONOPS .
 inc PROTEINOPS .
 inc SIGNATUREOPS .
 inc SITEOPS .
 inc STRESSOPS .
 inc THEOPS .
endfm
```

En el módulo `THEOPS` y los módulos que se incluyen sucesivamente, se definen las clases y operadores que nos permiten modelizar la célula. Se define el constructor de un dish (`Dish`) con el operador `PD`, cuyo argumento es la sopa de localizaciones. Cada una de las localizaciones se construye con el operador `{_|_}`, donde el primer argumento es el identificador o nombre de la localización y el segundo argumento es la sopa (`Soup`) de sus componentes.

```
op PD : Soup -> Dish [ctor] .
op {_|_} : LocName Soup -> Location [ctor format (n d d d d d)] .
```

La sopa se construye con el operador `__`, de forma que una sopa es un conjunto asociativo y commutativo de elementos (`Things`) tales como genes, proteínas modificadas, compuestos, etc. (ver listado 3.3).

Listing 3.3: Módulo `SOUP`

```
fmod SOUP is
 pr THING .

 sort MtSoup .
 op empty : -> MtSoup [ctor] .

 sort Soup .
 subsort MtSoup < Soup .
 op __ : Soup Soup -> Soup [ctor assoc comm id: empty] .
 op __ : MtSoup MtSoup -> MtSoup [ditto] .

 *** Soup sorts
 sort Things .  *** soup of things
```

```
 subsort MtSoup < Things .
 subsorts Thing < Things < Soup .
 op __ : Things Things -> Things [ditto] .

 sort Location .
 sort Locations .  *** soup of locations
 subsort MtSoup < Locations .
 subsorts Location < Locations < Soup .
 op __ : Locations Locations -> Locations [ditto] .
endfm
```

En el módulo `MODIFICATION` se incluye la sintaxis para las modificaciones. Se construye con el operador `[_-_]`, donde su primer argumento es el elemento a modificar (por ejemplo, una proteína) y el segundo argumento es el conjunto de modificadores que afectan a ese elemento. Estos elementos se definen también como una sopa o multiconjunto asociativo y commutativo de modificadores.

```
fmod MODIFICATION is
 pr SOUP .

 sorts Site  Modification ModSet .
 subsort Modification < ModSet .

 op none : -> ModSet .
 op __ : ModSet ModSet -> ModSet [assoc comm id: none] .

 op [_-_] : Protein   ModSet -> Protein   [right id: none ] .
 op [_-_] : Chemical  ModSet -> Chemical  [right id: none ] .
 op [_-_] : Gene      ModSet -> Gene      [right id: none ] .
endfm
```

A continuación, el módulo `TGFB1DISH` define el dish o estado inicial en el modelo Tgfb1 de STM7. El listado 3.4 contiene la definición de la constante `Tgfb1Dish` de la clase `Dish` como un conjunto de localizaciones con sus contenidos. Por ejemplo, se puede comprobar que la membrana de la célula está vacía (`{CLm | empty}`); que el citoplasma (`CLc`) contiene, entre otras, las proteínas `Abl1`, `Akt1` y `Atf2`; y que en el interior de la membrana celular (`CLi`) está presente la proteína `Hras` con la modificación `GDP` (`[Hras - GDP]`):

Listing 3.4: Módulo `TGFB1DISH`

```
mod TGFB1DISH is
inc ALLOPS .

op Tgfb1Dish : -> Dish .
eq Tgfb1Dish = PD(
 {XOut | Tgfb1 } {Tgfb1RC | TgfbR1 TgfbR2  } {CLo | empty } {CLm | empty }
 {CLi | [Cdc42 - GDP] [Hras - GDP] [Rac1 - GDP] }
 {CLc | Abl1 Akt1 Atf2 Erks Fak1 Jnks Mekk1 Mlk3 P38s Pak2 Pml Smad2 Smad3
     Smad4 Smurf1 Smurf2 Tab1 Tab2 Tab3 Tak1 Traf6 Zfyve16 }
 {NUc | Ctdsp1 Ets1 Smad7 Cdc6-gene Cdkn1a-gene Cdkn2b-gene Col1a1-gene
     Col3a1-gene Ctgf-gene Fn1-gene Mmp2-gene Pai1-gene Smad6-gene Smad7-gene
     Tgfb1-gene Timp1-gene Cst6-gene Dst-gene Mmp9-gene Mylk-gene Pthlh-gene
     Gfi1-gene Csrp2-gene RoRc-gene } ) .

endm
```

El módulo `ALLRULES` incluye todas las reglas de reescritura que gobiernan la dinámica de la célula. El significado de estas reglas se explicó con detalle en la seccion 2.4. En el

listado 3.5 se muestran dos de sus reglas.

Listing 3.5: Reglas en el módulo `ALLRULES`

```
rl[931.TgfbR1.TgfbR2.by.Tgfb1]:
  {XOut    | xout Tgfb1                                    }
  {Tgfb1RC | tgfb1rc  TgfbR1 TgfbR2                        }
  =>
  {XOut    | xout                                         }
  {Tgfb1RC | tgfb1rc ([TgfbR1 - act] : [TgfbR2 - act] : Tgfb1) } .

rl[1719.Abl1.irt.Tgfb1]:
  {Tgfb1RC | tgfb1rc ([TgfbR1 - act] : [TgfbR2 - act] : Tgfb1) }
  {CLc     | clc      [Fak1 - fak1mods] Abl1             }
  =>
  {Tgfb1RC | tgfb1rc ([TgfbR1 - act] : [TgfbR2 - act] : Tgfb1) }
  {CLc     | clc      [Fak1 - fak1mods] [Abl1 - act]     } .
```

Por último, el módulo `SOFT-SET-FUN` define las estrategias de decisión posibles con el soft set utilizado, que se comenta en la siguiente sección.

3.2. Soft sets y funciones de elección

En la sección 2.3 se han descrito los fundamentos teóricos de los soft sets. Ahora se comenta la implementación en Maude de los soft sets y el desarrollo de los operadores y funciones asociados.

Un soft set se puede representar como una matriz cuyos elementos son ceros, unos y asteriscos. En nuestro caso, las filas corresponden con todos los posibles términos en los que se puede reescribir nuestro término inicial y las columnas a cada uno de los atributos que se consideran. Por tanto, desde el punto de vista de la codificación, la implementación de los soft sets consiste en definir matrices. En el módulo `MATRIX` se definen la fila de una matriz como una listas de valores y la matriz como una lista de filas:

```
fmod MATRIX is
 pr VALUE .
 sorts Row SoftSet .
 subsort Value < Row < SoftSet .

 op mtRow : -> Row [ctor] .
 op _,_ : Row Row -> Row [ctor assoc id: mtRow] .

 op mt : -> SoftSet [ctor] .
 op __ : SoftSet SoftSet -> SoftSet [ctor assoc id: mt] .
endfm
```

Después se definen las funciones que calculan los valores de elección (*choice values*). Los valores de elección asignan un valor numérico (natural o racional) a cada posible término reescrito (o término alcanzable por reescritura) y el *mejor* término alcanzable será el que tenga un mayor valor. Existen varias formas de definir estos valores de elección. Según se explica en la sección 2.3, uno de los valores de elección es $c_{i(0)}$, que es el valor de elección si se supone que todos los datos que faltan son ceros. A continuación se detalla el código Maude para $c_{i(0)}$, con un fragmento del módulo `PREDEF-VALUE-FUNCTIONS` en el que se define el funcionamiento de las funciones `undefZero` y su función auxiliar `compUndefZero` (ver listado 3.6).

Listing 3.6: Funciones `undefZero` y su función auxiliar `compUndefZero`

```
*** MAX CHOICE VALUE FUNCTION BY ZEROS
*** Replaces * by 0 and adds up all values
op undefZero : SoftSet -> Nat .
eq undefZero(M) = compUndefZero(M, 0, 0, 0) .

*** Current - Best value - Selected Row
op compUndefZero : SoftSet Nat Nat Nat -> Nat .
eq compUndefZero(mt, C, BV, S) = S .
ceq compUndefZero(R M, C, BV, S) =
              if N >= BV
              then compUndefZero(M, s(C), N, C)
              else compUndefZero(M, s(C), BV, S)
              fi
 if N := addUndefZero(R) .

op addUndefZero : Row -> Nat .
eq addUndefZero(mtRow) = 0 .
eq addUndefZero((0, R)) = addUndefZero(R) .
eq addUndefZero((1, R)) = s(addUndefZero(R)) .
eq addUndefZero((*, R)) = addUndefZero(R) .
```

De forma análoga se codifican las funciones `undefOne` y `undefSemi` que calculan los mejores valores de elección cuando se sustituyen los asteriscos por 1 o por 1/2.

Las dos nuevas funciones propuestas definen también el valor de elección para una fila como la suma de los valores asignados a cada elemento de la fila. Los elementos 0 y 1 suman su propio valor. Sin embargo, en el caso de los asteriscos, las nuevas funciones tomarán como referencia la distribución de los valores desconocidos en su fila o columna, respectivamente.

De esta forma, la estrategia `undefWRow` asigna el siguiente valor al grupo de elementos desconocidos de la fila i:

$$\frac{\sum_{i=0}^{N} i \cdot \binom{N}{i}}{2^N}$$

donde N es el número de asteriscos en la fila. Es decir, la función `undefWRow` asigna al grupo de asteriscos un valor que depende del número de elementos desconocidos en esa fila N. Para todas las posibles combinaciones de sustituciones de ceros y unos en el valor desconocido (es decir, 2^N), se calcula el número de casos en los que los unos suman un valor i, que es $\binom{N}{i}$. Después se multiplica por el valor i correspondiente y se suman, de forma que obtenemos el valor esperado para el grupo de elementos desconocidos de la fila de acuerdo con su distribución en la fila de la matriz.

Dicho de otra forma, el choice value en la estrategia `undefWRow` para la fila i es:

$$\mathrm{cv}(i) = N_1 \cdot \frac{\sum_{i=0}^{N} i \cdot \binom{N}{i}}{2^N}$$

donde N_1 es el número de unos en la fila y N es el número de asteriscos en la fila.

Por último, la nueva estrategia `undefWCol` asigna el valor a un elemento desconocido de la fila i y columna j basándose en la distribución de valores desconocidos en el atributo correspondiente (es decir, en la columna). El choice value que se asigna a cada fila será también la suma de los valores asignados a cada elemento. Este valor de cada elemento es igual a cero y uno para cada cero y uno, respectivamente. En el caso de un dato desconocido, entonces se le asigna $\frac{N_1}{N_1+N_0}$, donde N_0 y N_1 son respectivamente el número de ceros y unos

en la columna, siempre que $N_1 + N_0 \neq 0$, es decir, que todos los elementos de la columna no sean desconocidos. En caso contrario, se asigna el valor 1/2. Dicho de otra forma, en la estrategia `undefWCol` asignamos a cada asterisco la proporción de unos dentro de los valores conocidos en su columna.

3.3. Definición y reescritura de *softdish*

El elemento fundamental en PLSS es el *softdish*. Siguiendo el artículo de Santos-Buitrago et al. (2019), se define un *softdish* como un plato o dish de Pathway Logic junto con unos atributos que permitirán establecer una elección entre las distintas reglas de ejecución basadas en soft sets.

Para realizar la reescritura de un softdish, se necesita construir la matriz del soft set asociado buscando todos los términos alcanzables desde el término inicial con el valor de sus atributos y, después, se elige aplicar el término cuyo valor de elección sea mayor.

En la implementación he realizado dos versiones de reescritura (`rewStrat` y `rewNStrat`), que se explican con detalle en las siguientes subsecciones.

3.3.1. Operador `rewStrat`

En esta primera versión de reescritura, la implementación de la reescritura de un softdish se realiza con el operador `rewStrat`, que a su vez se apoya en el operador auxiliar `next`, que es el que calcula el siguiente término reescrito (ver Listado 3.7).

Listing 3.7: Operadores `rewStrat` y `next`

```
op rewStrat : Module Term TermList Qid -> Term .
ceq rewStrat(M, T, ATTS, Q) = rewStrat(M, T', ATTS, Q)
 if T' := next(M, T, ATTS, Q) .
eq rewStrat(M, T, ATTS, Q) = T [owise] .

op next : Module Term TermList Qid ~> Term .
ceq next(M, T, ATTS, Q) = T'
 if TL := allReachableTerms(M, T) /\
    TL =/= empty /\
    MX := matrixFromTerms(TL, ATTS) /\
    N := computeValue(MX, Q) /\
    T' := TL [N] .
```

Como se observa en el código, el operador `next` recibe como argumentos un módulo, un término y unos atributos. El operador `next` se apoya en las siguientes funciones:

- `allReachableTerms`: Calcula todos los términos alcanzables desde el término actual.
- `matrixFromTerms`: Construye la matriz asociada al soft set.
- `computeValue`: Calcula la fila (regla de reescritura) correspondiente con el mayor valor de decisión (con la función de choice value que se haya escogido).

La función `allReachableTerms` se implementa en el listado 3.8 haciendo uso de las funciones `metaReduce` y `metaSearch`.

Listing 3.8: Función `allReachableTerms`

```
op allReachableTerms : Module Term -> TermList .
ceq allReachableTerms(M, T) = allReachableTerms(M, T, V, 0, empty)
 if Ty := getType(metaReduce(M, T)) /\
    V := qid("V:" + string(Ty)) .

op allReachableTerms : Module Term Variable Nat TermList -> TermList .
ceq allReachableTerms(M, T, V, N, TL) = TL
 if metaSearch(M, T, V, nil, '+, 1, N) == failure .
ceq allReachableTerms(M, T, V, N, TL) =
    allReachableTerms(M, T, V, s(N), (TL, T'))
 if {T', Ty, SB} := metaSearch(M, T, V, nil, '+, 1, N) .
```

La función `matrixFromTerms` construye la matriz asociada al soft set. Para cada término de la lista de términos alcanzables, `matrixFromTerms` construirá una fila en la matriz. La función auxiliar `rowFromTerm` recibe cada término de la lista y los atributos, para formar la fila correspondiente en la matriz:

```
op matrixFromTerms : TermList TermList -> SoftSet .
eq matrixFromTerms(empty, ATTS) = mt .
eq matrixFromTerms((T, TL), ATTS) =
    rowFromTerm(T, ATTS) matrixFromTerms(TL, ATTS) .
```

La función `computeValue` recibe como argumento la matriz del soft set y el nombre de la función de decisión que se desea utilizar:

```
op computeValue : SoftSet Qid -> Nat .
eq computeValue(MX,'undefZero) = undefZero(MX) .
eq computeValue(MX,'undefOne)  = undefOne(MX) .
eq computeValue(MX,'undefSemi) = undefSemi(MX) .
eq computeValue(MX,'undefWRow) = undefWRow(MX) .
eq computeValue(MX,'undefWCol) = undefWCol(MX) .
```

3.3.2. Operador `rewNStrat`

En esta subsección se presenta la implementación del operador `rewNStrat` que realiza un tipo de reescritura de un softdish basado en narrowing.

Su función principal es `rewNStrat` que, junto con su auxiliar `nextN`, se encargan de realizar la reescritura[2]. Las funciones `matrixFromTerms` y `computeValue`, definidas en la subsección anterior para el operador `rewStrat`, se utilizan también en el nuevo operador `rewNStrat`.

La nueva función `nextN` implementa un código similar a `next`, salvo que invoca la función `allReachableNTerms`. La función `allReachableNTerms` en el listado 3.9 es la versión narrowing de la función `allReachableTerms` y hace uso de la función `metaReduce` y la función de búsqueda con narrowing (`metaNarrowingSearch`).

Listing 3.9: Función `allReachableNTerms`

```
op allReachableNTerms : Module Term -> TermList .
ceq allReachableNTerms(M, T) = allReachableNTerms(M, T, V, 0, empty)
 if Ty := getType(metaReduce(M, T)) /\
    V := qid("V:" + string(Ty)) .
```

[2]Las funciones `rewStrat`, `rewNStrat` y sus funciones auxiliares se definen en el módulo `SOFT-SET-FUN`.

```
op allReachableNTerms : Module Term Variable Nat TermList -> TermList .
ceq allReachableNTerms(M, T, V, N, TL) = TL
 if metaNarrowingSearch(putNarrowingAtt(M), T, V, '+, unbounded, 'none, N) ==
     failure .
ceq allReachableNTerms(M, T, V, N, TL) =
    allReachableNTerms(M, T, V, s(N), (TL, T'))
 if {T', Ty, SB, VQid:Qid, VSubs2:Substitution, VQid2:Qid} :=
     metaNarrowingSearch(putNarrowingAtt(M), T, V, '+, unbounded, 'none, N) .
```

En la búsqueda mediante narrowing con el comando `metaNarrowingSearch`, el módulo `M` se ha modificado con la función `putNarrowingAtt`.

Para que las reglas de reescritura definidas en el fichero `Tgfb1RulesSS.maude` puedan ser localizadas por `metaNarrowingSearch`, será necesario que al final de cada una de ellas aparezca el atributo `narrowing`. La función `putNarrowingAtt` permite añadir el atributo `narrowing` *on the fly* y de forma automática en todas las reglas del módulo `M`.

Con la función `putNarrowingAtt` implementada en el listado 3.10 se extraen todas las reglas del módulo dado y se sustituyen por las misma reglas con el atributo `narrowing`. Se hace una comprobación de la existencia de este atributo en la regla con la función booleana `hasNarrowing`. Para completar la tarea, la función auxiliar `getRules` toma el conjunto de reglas del módulo y la función `setRules` cambia las reglas nuevas por las anteriores.

Listing 3.10: Función `putNarrowingAtt`

```
op putNarrowingAtt : Module -> Module .
ceq putNarrowingAtt(M) = setRules(M, RS')
 if RS' := putNarrowingAtt(getRules(M)) .

op putNarrowingAtt : RuleSet -> RuleSet .
eq putNarrowingAtt(none) = none .
ceq putNarrowingAtt(rl L => R [AtS] . RS) = rl L => R [AtS] . putNarrowingAtt(
    RS)
 if hasNarrowing(AtS) .
ceq putNarrowingAtt(crl L => R if C [AtS] . RS) = crl L => R if C [AtS] .
    putNarrowingAtt(RS)
 if hasNarrowing(AtS) .
ceq putNarrowingAtt(rl L => R [AtS] . RS) = rl L => R [narrowing AtS] .
    putNarrowingAtt(RS)
 if not hasNarrowing(AtS) .
ceq putNarrowingAtt(crl L => R if C [AtS] . RS) = crl L => R if C [narrowing
    AtS] . putNarrowingAtt(RS)
 if not hasNarrowing(AtS) .

op hasNarrowing : AttrSet -> Bool .
eq hasNarrowing(narrowing AtS) = true .
eq hasNarrowing(AtS) = false [owise] .

op setRules : SModule RuleSet -> SModule .
eq setRules(mod H is IL sorts SS . SSDS ODS MAS EqS RS endm, RS') =  mod H is
    IL sorts SS . SSDS ODS MAS EqS RS endm .

op getRules : SModule -> RuleSet .
eq getRules(mod H is IL sorts SS . SSDS ODS MAS EqS RS endm)  =  RS .
```

En la siguiente regla de reescritura se puede observar como el atributo `narrowing` de la regla ya aparece:

```
rl[1719.Abl1.irt.Tgfb1]:            *** Abl1--
 {Tgfb1RC | tgfb1rc ([TgfbR1 - act] : [TgfbR2 - act] : Tgfb1) }
 {CLc   | clc   [Fak1 - fak1mods] Abl1  }
```

```
=>
{Tgfb1RC | tgfb1rc ([TgfbR1 - act] : [TgfbR2 - act] : Tgfb1) }
{CLc   | clc   [Fak1 - fak1mods] [Abl1 - act]  } [narrowing] .
```

3.4. Entorno de ejecución con IO y metaintérprete en PLSS

En esta sección se describe la programación del entorno de ejecución para PLSS utilizando flujos de entradas/salidas estándar y metaintérpretes. Con PLSS se ha desarrollado un lenguaje propio simple con su propia gramática y una transformación de análisis de los términos en Maude. El lenguaje de PLSS incluye las siguientes características:

- Se pueden declarar clases y operaciones, y especificar ecuaciones en los términos que podemos construir con ellas.
- Se pueden incluir módulos previamente definidos y almacenar módulos en la base de datos del metaintérprete para que puedan ser utilizados después.
- Se pueden reducir los términos a su forma canónica utilizando las ecuaciones de un módulo. La sintaxis definida para PLSS permite los siguientes comandos: `red` para la simplificación ecuacional, `softrew` para la simplificación con soft reglas, `softnarrowsearch` para el análisis mediante narrowing (o estrechamiento) y, por último, `softnarrowrew` para la simplificación de la búsqueda mediante narrowing.
- Se puede incluir la estrategia de decisión que se desee utilizar.
- Se puede solicitar información sobre los comandos con el comando `help`.
- Se puede ejecutar una serie de instrucciones almacenadas en un fichero de texto externo.
- Se puede salir del programa con los comandos `exit` y `q`.

La principal diferencia entre metanivel y metaintérprete es que el metanivel es funcional pero el metaintérprete no lo es y permite interactuar con él mediante mensajes. El metaintérprete permite almacenar módulos y vistas en su base de datos, y luego operar con ellos. Los metaintérpretes proporcionan la funcionalidad deseada tanto a nivel de objeto como a metanivel.

Con PLSS, se almacena un mínimo de información en un objeto que solicitará entradas al usuario utilizando el flujo de entrada estándar e intentará analizarlo en la gramática de PLSS. Para poder analizar las entradas usando el metaintérprete, se empieza introduciendo el módulo con la sintaxis de PLSS en él. Una vez insertado, se puede intentar analizar las entradas. Cuando el flujo estándar recibe un mensaje `getLine`, responde con la cadena tecleada por el usuario hasta que se pulsa una tecla de retorno. Para poder analizar las entradas de varias líneas, se necesita solicitar nuevas líneas hasta que la entrada se complete. Por supuesto, en cualquier momento podemos obtener un error de parseo o una ambigüedad, en cuyo caso se necesita informar del error.

Una vez que se introduce la entrada, se pueden utilizar distintos comandos. Si la entrada corresponde a un comando de simplificación, el término debe ser analizado y luego reducido por el metaintérprete.

Para simplificar el proceso, se utiliza un atributo `state` que lleva la cuenta de las distintas alternativas. Este objeto también almacena más información (e.g., el identificador del metaintérprete, el nombre del último módulo introducido y la entrada parcial introducida). Algunos de los estados más significativos son:

```
sort State .

*** Initial state, metainterpreter is created
op init : -> State [ctor] .
*** Loading from standard database
op load-std-db : -> State [ctor] .
*** Loading grammar
op load-grammar : -> State [ctor] .
*** Create user metainterpreter
op load-user-mi : -> State [ctor] .
*** Waiting input from the user
op idle : -> State [ctor] .
*** Parsing command
op parseComm : -> State [ctor] .
*** Execute command
op executeComm : -> State [ctor] .
*** Waits for a wrote msg and returns to idle with getLine
op print&idle : -> State [ctor] .
*** Waits for a wrote msg and keeps executing commands.
op print&executeComm : -> State [ctor] .
```

En la figura 3.1 se representa el diagrama de estados del entorno de ejecución, que muestra la secuencia de estados por los que pasa un objeto a lo largo de su vida.

Para guiar los pasos intermedios se utilizan varios mensajes, donde `plss` será el objeto y `PLSS` su clase. Estos son algunos ejemplos:

```
insertModule(MI, plss, upModule(Q, false)) .
write(stdout, plss, STR)
parseTerm(MI, plss, 'GRAMMAR, none, tokenize(Text), '@Input@) .
createInterpreter(interpreterManager, plss, none) .
```

El entorno de PLSS se inicia utilizando la configuración `run` que está formado por las constantes de configuración, con ello se creará un intérprete y se enviará el prompt del programa al flujo de salida:

```
< plss : PLSS | file: null, out: "", state: init, load: modList,
         uload: umodList, input: empty, module: userModule, strat: 'undefZero >
```

Los atributos que se asocian al objeto pertenecen a la clase `Attribute` y se definen junto con sus valores:

```
*** Attributes
op uload:_ : List{Preload} -> Attribute [ctor gather (&)] .
op load:_ : List{Preload} -> Attribute [ctor gather (&)] .
op module:_ : Maybe{Module} -> Attribute [ctor] .
op file:_ : Maybe{Oid} -> Attribute [ctor] .
op input:_ : TermList -> Attribute [ctor] .
op state:_ : State -> Attribute [ctor] .
op out:_ : String -> Attribute [ctor] .
op umi:_ : Oid -> Attribute [ctor] .
op mi:_ : Oid -> Attribute [ctor] .
op strat:_ : Maybe{Qid} -> Attribute [ctor] .
```

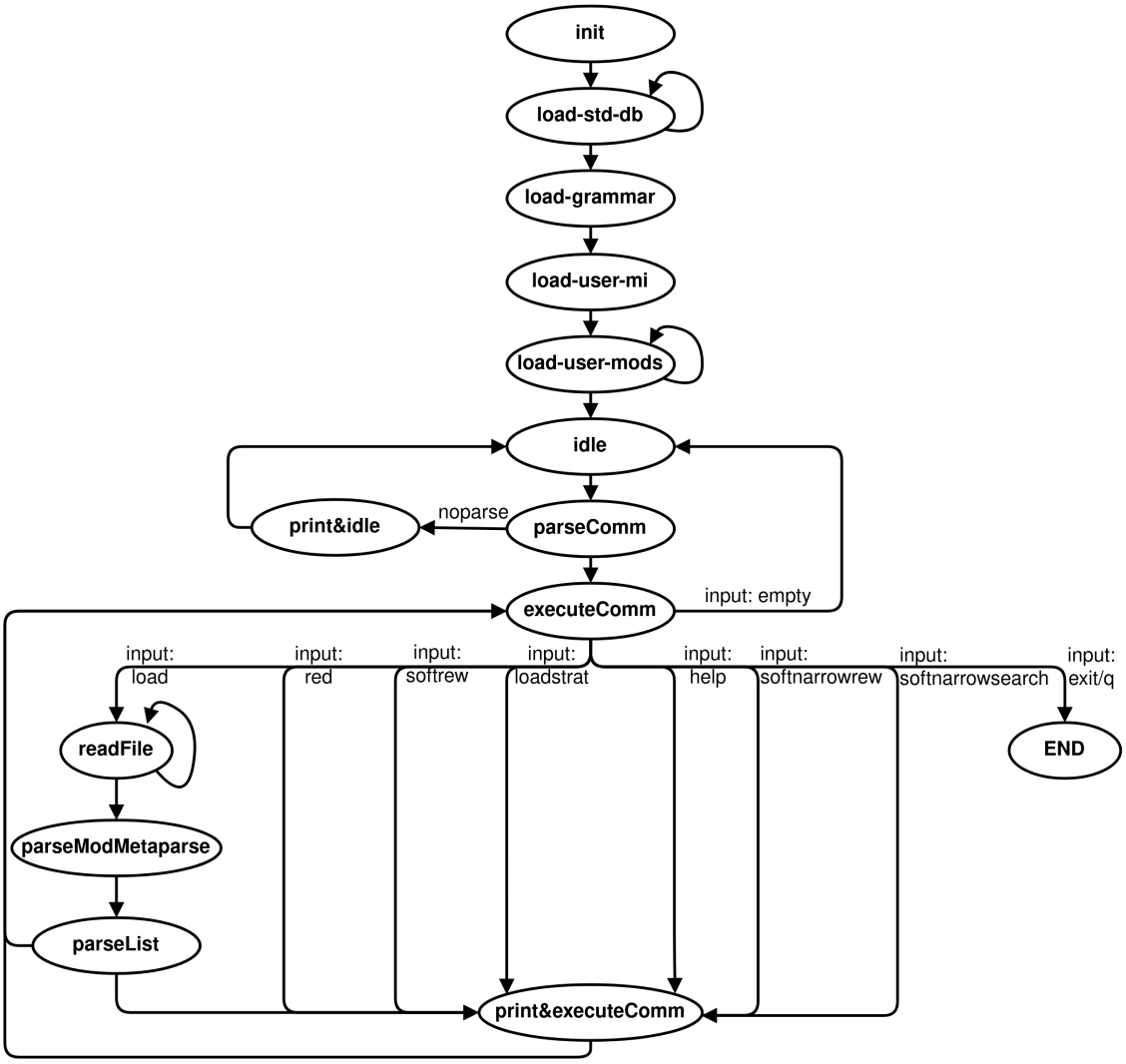

Figura 3.1: Diagrama de estados de PLSS.

Una vez escrito el mensaje y creado el metaintérprete, se inserta la lista de módulos que configuran la sintaxis de PLSS:

```
eq modList = module('TRUTH-VALUE) module('BOOL-OPS) module('TRUTH)
             module('BOOL) module('EXT-BOOL) module('NAT) module('RANDOM)
             module('COUNTER) module('INT) module('RAT) module('FLOAT)
             module('STRING) module('CONVERSION) module('BOUND)
             module('QID) view('Qid) module('TRIV) view('TRIV)
             module('LIST) module('QID-LIST)
             module('COMMON-SIGN) module('COMMAND-SIGN) module('PLSS-SIGN) .
```

Los módulos y vistas que se utilizan se van cargando con reglas similares a la regla `init-module`:

```
rl [init-module] :
   createdInterpreter(plss, interpreterManager, MI)
   < plss : PLSS | state: init, load: module(Q) LP, AtS >
=> < plss : PLSS | state: load-std-db, mi: MI, load: LP, AtS >
   insertModule(MI, plss, upModule(Q, false)) .
```

En esta regla se carga el primer módulo de la lista de módulos (disponible en el atributo `load:`. El segundo argumento del mensaje creado `createdInterpreter` es el remitente `interpreterManager`. Después PLSS carga el módulo con la gramática a través de la regla `load-predef-finished`:

```
rl [load-predef-finished] :
   insertedModule(plss, MI)
   < plss : PLSS | state: load-std-db,  load: nil, AtS >
=> < plss : PLSS | state: load-grammar, load: nil, AtS >
   insertModule(MI, plss, GRAMMAR) .
```

De la misma forma, a continuación se crea el metaintérprete de usuario y los módulos de su sintaxis:

```
rl [user-metainterpreter-created] :
   createdInterpreter(plss, interpreterManager, MI)
   < plss : PLSS | state: load-user-mi, uload: module(Q) LP, AtS >
=> < plss : PLSS | state: load-user-mods, umi: MI, uload: LP, AtS >
   insertModule(MI, plss, upModule(Q, false)) .

rl [load-user-predef-module-metainterpreter] :
   insertedModule(plss, MI)
   < plss : PLSS | state: load-user-mods, uload: module(Q) LP, AtS >
=> < plss : PLSS | state: load-user-mods, uload: LP, AtS >
   insertModule(MI, plss, upModule(Q, false)) .
```

Una vez que se han insertado todos los módulos del metaintérprete, se envía un mensaje `getLine` al objeto `stdin`, que contiene el prompt del programa (`PLSS>`).

```
rl [user-predef-module-metainterpreter-finished] :
   insertedModule(plss, MI)
   < plss : PLSS | state: load-user-mods, uload: nil, AtS >
=> < plss : PLSS | state: idle, AtS >
   getLine(stdin, plss, "PLSS>") .
```

En ese momento el programa se situa en el estado `idle`, preparado para recibir comandos. Cuando el usuario introduce algunas entradas, el objeto `stdin` responde con un mensaje `gotLine` con la cadena introducida. Las entradas se analizan por el metaintérprete y toda la lista `TermList` se almacena en el atributo `input`. Por ejemplo, el comando `exit` responde con un mensaje de despedida:

```
rl [exit] :
   < plss : PLSS | state: executeComm, input: ('exit.@Command@), AtS >
=> write(stdout, plss, "Thanks for using PLSS!\n") [print AtS] .
```

Si el análisis tiene éxito, el metaintérprete responde con un mensaje `parsedTerm` que incluye un término de tipo `ResultPair`. Este término se envía a la siguiente función y se sitúa en el estado `parseComm` para que continúe la ejecución.

Cuando falla el análisis, el mensaje `parsedTerm` del metaintérprete incluye un término `noParse` con la posición en la que falló el análisis. Si la posición es el final de la entrada, significa que la entrada estaba incompleta, y en ese caso esa entrada parcial se añade a la entrada actual y se solicita texto adicional al usuario. Si el error estaba en otra posición, se envía un mensaje de error.

```
rl [parseCommError] :
```

```
    parsedTerm(plss, MI, noParse(N))
    < plss : PLSS | mi: MI, state: parseComm, AtS >
 => < plss : PLSS | mi: MI, state: print&idle, AtS >
    write(stdout, plss, "The introduced command does not exit.\n") .
```

Como puede verse en la figura 3.1, cada vez que concluye un comando el programa vuelve al estado `idle` tras notificarlo al usuario, independientemente del resultado del análisis.

En la siguiente sección se comenta con detalle la codificación de los comandos de simplificación y de la manipulación de estrategias.

3.5. Comandos de simplificación con ecuaciones y reglas soft

Por una parte, el comando `red` realiza la simplificación ecuacional del término introducido. Su código se muestra en el listado 3.11. Para la codificación de este comando, después del análisis, se pasa al estado `executeComm`.

El comando recibe una burbuja como argumento y la regla `redComm` realiza las siguientes tareas:

1. Análisis de la burbuja con el operador `metaParse`:

   ```
   {T', Ty} := metaParse(M, downQidList(T), anyType)
   ```

2. Reducción del término analizado con el operador `metaReduce`:

   ```
   {T'', Ty'} := metaReduce(M, T')
   ```

3. Presentación mejorada del término reducido con el operador `metaPrettyPrint`:

   ```
   QIL := metaPrettyPrint(M, T'')
   ```

4. Con el operador `printTokens`, preparación del mensaje a mostrar con la lista de tokens devueltos:

   ```
   STR := "\nResult: " + printTokens(QIL) + "\n"
   ```

Al final envía el mensaje `STR` al metaintérprete y pasa PLSS al estado `print&executeComm` para que muestre el resultado y que pase de nuevo al estado `idle` a la espera de un nuevo comando o bien al estado `executeComm` para ejecutar el resto de comandos de la entrada.

Listing 3.11: Comando `red` en PLSS.

```
crl [redComm] :
    < plss : PLSS | state: executeComm, input: ('red_['bubble[T]], INPLST),
        module: M, AtS >
 => < plss : PLSS | state: print&executeComm, input: INPLST, module: M, AtS >
    write(stdout, plss, STR)
```

```
  if {T', Ty} := metaParse(M, downQidList(T), anyType) /\
     {T'', Ty'} := metaReduce(M, T') /\
     QIL := metaPrettyPrint(M, T'') /\
     STR := "\nResult: " + printTokens(QIL) + "\n" .
```

Por otra parte, el comando `softrew` realiza la simplificación del término introducido con las soft reglas, es decir, con las reglas escogidas de acuerdo con la función de elección definida en los soft sets. Su código se muestra en el listado 3.12. Para la codificación de este comando, después del análisis, se pasa al estado `executeComm`.

El comando recibe una burbuja como argumento y la regla `softrewstratComm` realiza las siguientes tareas:

1. Análisis de la burbuja con el operador `metaParse` y asignación mediante matching a las variables `DISH` y `ATTS` de los valores correspondientes:

   ```
   {'__[DISH, ATTS], 'SoftDish} := metaParse(M, downQidList(T), 'SoftDish)
   ```

2. Reescritura del término analizado con el operador `rewStrat`, que se comentó en la sección 3.3:

   ```
   T' := rewStrat(M, DISH, ATTS, MQ)
   ```

3. Presentación mejorada del término reducido con el operador `metaPrettyPrint`:

   ```
   QIL := metaPrettyPrint(M, T'')
   ```

4. Con el operador `printTokens`, preparación del mensaje a mostrar con la lista de tokens devueltos:

   ```
   STR := "\nResult: " + printTokens(QIL) + "\n"
   ```

Al final envía el mensaje `STR` al metaintérprete y pasa PLSS al estado `print&executeComm` para que muestre el resultado y que pase de nuevo al estado `idle` a la espera de un nuevo comando o bien al estado `executeComm` para ejecutar el resto de comandos de la entrada.

Listing 3.12: Comando `softrew` en PLSS.

```
crl [softrewstratComm] :
    < plss : PLSS | state: executeComm, input: ('softrew_['bubble[T]], INPLST)
        , module: M, strat: MQ, AtS >
 => < plss : PLSS | state: print&executeComm, input: INPLST, module: M, strat:
      MQ, AtS >
    write(stdout, plss, STR)
 if {'__[DISH, ATTS], 'SoftDish} := metaParse(M, downQidList(T), 'SoftDish) /\
    T' := rewStrat(M, DISH, ATTS, MQ) /\
    QIL := metaPrettyPrint(M, T') /\
    STR := "\nResult: " + printTokens(QIL) + "\n" .
```

Por último se ha implementado un comando con la función de realizar el análisis mediante el estrechamiento. El comando `softnarrowrew` reescribe el término introducido para

la simplificación de la búsqueda mediante narrowing. En la codificación de este comando, después del análisis, se pasa al estado `executeComm`.

El comando recibe una burbuja como argumento y la regla `softnarrowrewstratComm` realiza las siguientes tareas:

1. Análisis de la burbuja con el operador `metaParse` y asignación mediante matching a las variables `DISH` y `ATTS` de los valores correspondientes:

```
{'__[DISH, ATTS], 'SoftDish} := metaParse(M, downQidList(T), 'SoftDish)
```

2. Reescritura del término analizado con el operador `rewNStrat`, que se comentó en la sección 3.3:

```
T' := rewNStrat(M, DISH, ATTS, MQ)
```

3. Presentación mejorada del término reducido con el operador `metaPrettyPrint`:

```
QIL := metaPrettyPrint(M, T')
```

4. Con el operador `printTokens`, se prepara el mensaje a mostrar con la lista de tokens devueltos:

```
STR := "\nResult: " + printTokens(QIL) + "\n"
```

Al final envía el mensaje `STR` al metaintérprete y PLSS pasa al estado `print&executeComm` para que muestre el resultado y que pase de nuevo al estado `idle` a la espera de un nuevo comando o bien al estado `executeComm` para ejecutar el resto de comandos de entrada.

El listado 3.13 muestra el código de esta regla.

Listing 3.13: Regla `softnarrowrewstratComm`.

```
crl [softnarrowrewstratComm] :
    < plss : PLSS | state: executeComm, input: ('softnarrowrew_['bubble[T]],
        INPLST), module: M, strat: MQ, AtS >
 => < plss : PLSS | state: print&executeComm, input: INPLST, module: M, strat:
      MQ, AtS >
    write(stdout, plss, STR)
 if {'__[DISH, ATTS], 'SoftDish} := metaParse(M, downQidList(T), 'SoftDish) /\
    T' := rewNStrat(M, DISH, ATTS, MQ) /\
    QIL := metaPrettyPrint(M, T') /\
    STR := "\nResult: " + printTokens(QIL) + "\n" .
```

3.6. Comando de búsqueda con narrowing

En esta sección se describe el comando de búsqueda `softnarrowsearch` mediante estrechamiento. De esta forma, se analizarán todos los cómputos de un conjunto de estados gracias a la descripción de los mismos mediante técnicas simbólicas.

El comando `softnarrowsearch` se encarga de realizar la búsqueda narrowing de un término inicial a un término final. En lo referente a los estados, después del análisis del comando, se pasa al estado `executeComm`. La sintaxis de este comando se define en el fichero `grammar.maude`, como se muestra a continuación:

```
op softnarrowsearch_ _ : @Bubble@ @Bubble@ -> @Command@ [ctor] .
```

Como se puede observar, este comando recibe dos burbujas como argumentos de entrada. Estos argumentos se corresponden con un término `SoftDish` inicial y otro término `SoftDish` que será el patrón a alcanzar. Ambos términos pueden contener variables, que serán asociadas cuando se complete la ejecución de `metaNarrowingSearch`.

La regla `softNarrowSearchComm` realizará las tareas necesarias para que el comando `softnarrowsearch` se pueda ejecutar. En la implementación de esta búsqueda, se han codificado tres reglas para cada uno de los siguientes casos:

1. Error en el parseo de los términos;
2. No existe resultado en la búsqueda;
3. Se encuentra al menos una solución.

En el primer caso, cuando existe un error de parseo en alguno de los argumentos de entrada del `metaNarrowingSearch`, se resuelve mediante el uso de disyunción de dos condiciones negativas. Cada condición consiste en comprobar que el `metaParse` de cada uno de los dos términos (`T` y `T1`) no pertenece al tipo `ResultPair`:

```
( not (metaParse(M, downQidList(T),  'SoftDish) :: ResultPair)  or
  not (metaParse(M, downQidList(T1), 'SoftDish) :: ResultPair))
```

Cuando se ejecuta esta regla, se muestra el siguiente mensaje de error:

```
STR := "\nError en los términos parseados para MetaNarrowingSearch.\n\n"
```

El listado 3.14 muestra el código de esta regla.

Listing 3.14: Regla `softNarrowSearchComm` en el caso 1.

```
crl [softNarrowSearchComm] :
< plss : PLSS | state: executeComm, input: ('softnarrowsearch__['bubble[T],
         'bubble[T1]], INPLST), module: M, AtS >
=> < plss : PLSS | state: print&executeComm, input: INPLST, module: M, AtS >
   write(stdout, plss, STR)
if (not (metaParse(M, downQidList(T), 'SoftDish) :: ResultPair) or
  not (metaParse(M, downQidList(T1), 'SoftDish) :: ResultPair)) /\
  STR := "\nError en los términos parseados para MetaNarrowingSearch.\n\n" .
```

El segundo caso se encarga de comprobar que la búsqueda con `metaNarrowingSearch` no encuentra solución. Para ello, se comprueba que el resultado no sea del tipo `NarrowingSearchResult`, es decir, que sea la constante `failure`:

```
not (metaNarrowingSearch(M, T', T1', '*, unbounded, 'none, 0) ::
    NarrowingSearchResult)
```

Cuando se ejecuta esta regla, se muestra el siguiente mensaje:

```
STR := "\nNo solución en la búsqueda con MetaNarrowingSearch.\n\n"
```

El listado 3.15 muestra el código de esta regla.

Listing 3.15: Regla `softNarrowSearchComm` en el caso 2.

```
crl [softNarrowSearchComm] :
< plss : PLSS | state: executeComm, input: ('softnarrowsearch__['bubble[T],
        'bubble[T1]], INPLST), module: M, AtS >
=> < plss : PLSS | state: print&executeComm, input: INPLST, module: M, AtS >
   write(stdout, plss, STR)
if  {T', Ty} := metaParse(M, downQidList(T), 'SoftDish) /\
  {T1', T1y} := metaParse(M, downQidList(T1), 'SoftDish) /\
  not (metaNarrowingSearch(putNarrowingAtt(M), T', T1', '*, unbounded, 'none,
      0) :: NarrowingSearchResult) /\
  STR := "\nNo solución en la búsqueda con MetaNarrowingSearch.\n\n" .
```

En el tercer caso, el comando `metaNarrowingSearch` encuentra al menos una solución, que será del tipo `NarrowingSearchResult`. La asignación de las salidas de la solución de este comando se realiza con la siguiente instrucción:

```
{T'', 'SoftDish, VSubs:Substitution, VQid:Qid, VSubs2:Substitution, VQid2:Qid }
     := metaNarrowingSearch(M, T', T1', '*, unbounded, 'none, 0)
```

Los argumentos de entrada del comando `metaNarrowingSearch` son:

- El primer argumento de entrada (`M`) corresponde a la metarrepresentación del módulo en que se realice la búsqueda.
- Los siguientes argumentos (`T'` y `T1'`) son, respectivamente, las metarrepresentaciones del término inicial y la del patrón a alcanzar.
- Se continúa con la metarrepresentación de la flecha de búsqueda, en nuestro caso `'*`, correspondiente a `=>*` en la ejecución del comando `search`. Existen otros tipos de búsqueda:
  - `'*` para una búsqueda que implique cero o más reescrituras (`=>*`)
  - `'+` para una búsqueda que consta de una o más reescrituras (`=>+`)
  - `'!` para una búsqueda que solo coincide con formas canónicas (`=>!`)
- El siguiente argumento es la longitud máxima de narrowing. En nuestro caso es `unbounded` del tipo `Bound`.
- Después, podrán aparecer las constantes `'match` o `'none`. La diferencia entre ellas, consiste en que la constante `'none` indica que se aplica una reducción estándar sin ningún plegado, mientras que con la constante `'match` se aplica la reducción de plegado.
- El último argumento será un numero natural, que representa la solución elegida. De esta forma, como en muchos otros comandos de metanivel, se pueden proporcionar todas las soluciones de forma secuencial.

Los elementos de la salida de `NarrowingSearchResult` se corresponden con:

- El primer elemento `Term` (`T''`) es el término patrón, es decir, el término al que se desea llegar, junto con la última sustitución.
- El segundo elemento se corresponde con `Type` ('SoftDish), que es la metarrepresentación del tipo del término.
- `Substitution` (`VSubs:Substitution`) es equivalente a la sustitución para conseguirlo, junto con la última sustitución incorporada.
- El siguiente elemento es un `Qid` (`VQid:Qid`) que equivale a la metarrepresentación del identificador utilizado para crear nuevas variables.
- `Substitution` (`VSubs2:Substitution`) es la última sustitución y por último, otro `Qid` (`VQid2:Qid`) que metarrepresenta el identificador utilizado para el unificador de variante.

El listado 3.16 muestra el código de esta regla.

Listing 3.16: Regla `softNarrowSearchComm` en el caso 3.

```
crl [softNarrowSearchComm] :
< plss : PLSS | state: executeComm, input: ('softnarrowsearch__['bubble[T],
    'bubble[T1]], INPLST), module: M, AtS >
=> < plss : PLSS | state: print&executeComm, input: INPLST, module: M, AtS >
   write(stdout, plss, STR)
if  {T', Ty} := metaParse(M, downQidList(T), 'SoftDish) /\
   {T1', T1y} := metaParse(M, downQidList(T1), 'SoftDish) /\
   metaNarrowingSearch(putNarrowingAtt(M), T', T1', '*, unbounded, 'none, 0) ::
        NarrowingSearchResult /\
   {T'', 'SoftDish, VSubs:Substitution, VQid:Qid, VSubs2:Substitution,
       VQid2:Qid} :=
   metaNarrowingSearch(putNarrowingAtt(M), T', T1', '*, unbounded, 'none, 0) /\
   QIL := metaPrettyPrint(M, T'') /\
   QIL1 := printSB(M, VSubs:Substitution) /\
   QIL2 := printSB(M, VSubs2:Substitution) /\
   QIL3 := metaPrettyPrint(M, T') /\ QIL4 := metaPrettyPrint(M, T1') /\
   STR := "\nResult (SoftDish-term): " + printTokens(QIL) +
     "\nSubstitution1: " + printTokens(QIL1) +
     "\nSubstitution2: " + printTokens(QIL2) +
     "\nTerm-ini (T): "  + printTokens(QIL3) +
     "\nTerm-patron (T1): " + printTokens(QIL4) + "\n\n" .
```

Para mostrar adecuadamente las sustituciones del resultado se ha creado la función `printSB`. Con esta función, dada una sustitución, se devuelve una lista de `Qid`s.

```
op printSB : Module Substitution -> QidList .
eq printSB (M, none) = 'none .
eq printSB (M, V:Variable <- T:Term) =
   getName(V:Variable)  '<-  metaPrettyPrint(M, T:Term) .
eq printSB (M, V:Variable <- T:Term ; S:Substitution) =
   printSB (M, V:Variable <- T:Term)  printSB(M, S:Substitution) .
```

Al final de cada uno de estos casos, el mensaje `STR` se envia al metaintérprete y PLSS pasa al estado `print&executeComm` para que muestre la cadena correspondiente y pase de nuevo al estado `idle` a la espera de un nuevo comando o bien al estado `executeComm` para ejecutar el resto de comandos de la entrada.

3.7. Otros comandos de PLSS

En esta sección se describen los comandos de carga de ficheros con instrucciones (`load`), asignación de la función de elección (`loadstrat`) y sistema de ayuda (`help`).

3.7.1. Comando de carga de ficheros con instrucciones

El comando `load` realiza la carga de ficheros con instrucciones. Su código se muestra en el listado 3.17. Para la codificación de este comando, después del análisis, se pasa al estado `executeComm`.

El comando recibe como argumento un token con el nombre del fichero a cargar y se realizan las siguientes tareas:

1. La regla `loadComm` recibe el nombre del fichero como argumento y lo abre. También pasa a PLSS al estado `readFile`.
2. La regla `openedFile` coloca el nombre del fichero en el atributo `file:`.
3. La regla `readingModule` va leyendo cada una de las líneas del fichero. El texto leído lo almacena en el atributo `out:`. Cuando se encuentra con una línea vacía, envía el mensaje de cerrar el fichero.
4. La regla `closedFile` cierra el fichero con el nombre dado y pasa al estado `parseModMetaparse`. Además vacía el atributo `file:`, que antes contenía el nombre del fichero.
5. Las reglas `parseModuleMetaparse`, `parseModuleNoBubblesOK` y `parseModuleNoBubblesError` llevan a cabo el análisis del contenido del módulo y en caso de error muestra el mensaje correspondiente.

Al final se colocan todas las instrucciones en el atributo `input:` y pasa PLSS al estado `print&executeComm`. De esta forma, se irán ejecutando uno por uno todos los comandos que se han colocado en la entrada.

Listing 3.17: Comando `load` en PLSS.

```
crl [loadComm] :
   < plss : PLSS | state: executeComm, input: ('load_['token[T]], INPLST), AtS
        >
=> < plss : PLSS | state: readFile, input: INPLST, AtS >
   openFile(fileManager, plss, STR, "r")
 if STR := string(downQid(T)) .

rl [openedFile] : openedFile(plss, fileManager, FHIn)
   < plss : PLSS | file: null, state: readFile, AtS >
=> < plss : PLSS | file: FHIn, state: readFile, AtS >
   getLine(FHIn, plss) .

rl [readingModule] : gotLine(plss, FHIn, Text)
   < plss : PLSS | file: FHIn, out: Read, state: readFile, AtS >
=> if Text == ""
   then < plss : PLSS | file: FHIn, out: Read, state: readFile, AtS >
        closeFile(FHIn, plss)
   else < plss : PLSS | file: FHIn, out: (Read + Text), state: readFile, AtS >
        getLine(FHIn, plss)
   fi .
```

```
rl [closedFile] : closedFile(plss, FHIn)
   < plss : PLSS | file: FHIn, state: readFile, AtS >
=> < plss : PLSS | file: null, state: parseModMetaparse, AtS > .

rl [parseModuleMetaparse] :
   < plss : PLSS | out: Read, state: parseModMetaparse, mi: MI, AtS >
=> < plss : PLSS | out: Read, state: parseList, mi: MI, AtS >
   parseTerm(MI, plss, 'GRAMMAR, none, tokenize(Read), '@Input@) .

crl [parseModuleNoBubblesOK] :
    parsedTerm(plss, MI, {T, Ty})
    < plss : PLSS | out: Read, state: parseList, mi: MI, input: INPLST, AtS >
 => < plss : PLSS | out: "", state: executeComm, mi: MI, input: (TL, INPLST),
     AtS >
 if TL := extractTerms(T) .

crl [parseModuleNoBubblesError] :
    parsedTerm(plss, MI, noParse(N))
    < plss : PLSS | out: Read, state: parseList, mi: MI, AtS >
 => < plss : PLSS | out: "", state: print&executeComm, mi: MI, AtS >
    write(stdout, plss, printTokens(QIL))
 if QIL := showMsg(tokenize(Read), N) .
```

3.7.2. Comando de asignación de la función de elección

El comando `loadstrat` realiza la asignación de la función de elección de los soft sets. Para la codificación de este comando, después del análisis, se pasa al estado `executeComm`.

```
crl [loadstratComm] :
   < plss : PLSS | state: executeComm, input: ('loadstrat_['token[T]], INPLST)
      , strat: MQ, AtS >
=> < plss : PLSS | state: print&executeComm, input: INPLST, strat: downQid(T),
     AtS >
    write(stdout, plss, STR)
 if STR := "\nLoaded strategy function ( " +  string(downQid(T)) + " )\n" .
```

El comando recibe un token con el nombre de la función de estrategia como argumento y entonces la regla `loadstratComm` coloca ese término en el atributo `strat: downQid(T)`.

Al final envía el mensaje `STR` al metaintérprete y pasa PLSS al estado `print&executeComm` para que muestre el resultado y que pase de nuevo al estado `idle` a la espera de un nuevo comando o bien al estado `executeComm` para ejecutar el resto de comandos de la entrada.

3.7.3. Comando del sistema de ayuda

El comando `help` muestra la ayuda para PLSS:

```
rl [help] :
   < plss : PLSS | state: executeComm, input: ('help.@Command@, INPLST), AtS >
=> < plss : PLSS | state: print&executeComm, input: INPLST, AtS >
   write(stdout, plss, help + "\n") .
```

El comando no recibe argumentos, simplemente muestra el contenido de la ayuda que está almacenado en la constante `help`:

```
op help : -> String .
eq help =
```

```
"\n   ======================== \n" +
"   AYUDA DEL PROGRAMA: PLSS\n" +
"   ========================\n\n" +

"   1. load + ( Fichero de texto ): \n" +
"      Carga el fichero y ejecuta los comandos incluidos en el fichero. \n" +
"      Cuando exista más de una instrucción, cada una deberá estar \n" +
"      entre paréntesis. \n" +
"      Ejemplo: \n" +
"      PLSS> load softsetfile.txt \n" +
"       \n" +
"   2. loadstrat + ( Estrategia ): \n" +
"      Permite cargar una función de estrategia para el comando softrew.\n" +
"      Ejemplos: \n" +
"      PLSS> loadstrat undefSemi \n" +
"      PLSS> loadstrat undefWRow \n" +
"      PLSS> loadstrat undefOne \n" +
"       \n" +
"   3. red + ( Término ): \n" +
"      Reduce el término para la simplificación ecuacional. \n" +
"      Ejemplos: \n" +
"      PLSS> red Tgfb1Dish \n" +
"      PLSS> red (Tgfb1Dish  ([cdkn2b = *])) \n" +
"      PLSS> red (PD({CLc | empty})  ([cdc6 = 0])) \n" +
"      PLSS> red (atts(Tgfb1Dish  ([cdc6 = 0]))) \n" +
"       \n" +
"   4. softrew + ( Término SoftDish ): \n" +
"      Reescribe el término con soft reglas. \n" +
"      Ejemplos: \n" +
"      PLSS> softrew (Tgfb1Dish  ([cdc6 = 0], [cdkn2b = *])) \n" +
"      PLSS> softrew (PD({CLc | empty})  ([cdc6 = 0])) \n" +
"       \n" +
"   5. softnarrowrew + ( Término SoftDish ): \n" +
"      Reescribe el término para la simplificación de la búsqueda mediante
    narrowing. \n" +
"      Ejemplos: \n" +
"      PLSS> softnarrowrew (Tgfb1Dish  ([cdc6 = 0], [cdkn2b = *])) \n" +
"      PLSS> softnarrowrew (PD({CLc | empty})  ([cdc6 = 0])) \n" +
"       \n" +
"   6. softnarrowsearch + ( Término SoftDish inicial ) ( Término SoftDish
    patrón ): \n" +
"      Se encarga de realizar el análisis con narrowing (estrechamiento).\n"
    +
"      Ejemplos: \n" +
"      PLSS> softnarrowsearch ( manolito ) ( pepito ) \n" +
"      PLSS> softnarrowsearch (DD:Dish([cdc6 = 0]))(PD({XOut|Tgfb1})([cdc6
    =1])) \n" +
"      PLSS> softnarrowsearch (SD:SoftDish) (PD({XOut | Tgfb1}) ([cdc6 = 1]))
     \n" +
"       \n" +
"   7. exit / q: \n" +
"      Abandona la ejecución del programa. \n" +
"      Ejemplos: \n" +
"      PLSS> exit \n" +
"      PLSS> q \n" +
"       \n" +
"   8. help: \n" +
"      Ayuda del programa. \n" +
"      Ejemplo: \n" +
"      PLSS> help \n" .
```

Al final envía el mensaje `STR` al metaintérprete y pasa PLSS al estado `print&executeComm` para que muestre el resultado y que pase de nuevo al estado `idle` a la espera de un nuevo comando o bien al estado `executeComm` para ejecutar el resto de comandos de la entrada.

3.8. Integración de los módulos en PLSS

En esta sección se describen los módulos y ficheros desarrollados para este proyecto. En la figura 3.2 se muestran de forma gráfica las dependencias (importación) entre los distintos ficheros del proyecto.

Los módulos que se han desarrollado están incluidos en los siguientes ficheros:

- `plss.maude`: módulo principal del proyecto PLSS.
- `grammar.maude`: definición de la gramática de PLSS. En el módulo `COMMAND-SIGN` se definen los comandos (`load`, `red`, etc.).
- `plallSS.maude`: fichero con los operadores y reglas adaptadas de Pathway Logic.
- Las carpetas `ops`, `rules` y `SDishes` contienen los ficheros auxiliares para trabajar con los modelos de Pathway Logic.
- `parsing-command.maude` realiza el análisis de los comandos. Además incluye el módulo `COMMAND-PROCESSING`.
- `parsing.maude` realiza el resto de tareas de análisis y también contiene módulos auxiliares de la base de datos, mensajes, errores y otras operaciones con módulos.
- `softset.maude`: fichero de definición de softsets con los comandos de reescritura `rewStrat` y `rewNStrat` junto con sus funciones de estrategias.
- `softsetfile.txt`, `softsetfile1.txt`, `softsetfile2.txt`: ejemplos de ficheros con instrucciones para importar con la instrucción `load`.
- `test_plss.txt`: ejemplos de ejecución de comandos para utilizar dentro de PLSS con sus salidas.

Además de los ficheros descritos, se han utilizado los ficheros `metainterpreter.maude` y `file.maude` incluidos en la distribución de Maude 3.1.

En lo referente a los módulos, las figuras 3.3-3.5 muestran la relación de dependencia entre ellos. La figura 3.3 muestra la dependencia entre otros módulos, por ejemplo los para su correcta gramática, la figura 3.4 se centra en los módulos específicos de Pathway Logic y la figura 3.5 agrupa los módulos de uso general de PLSS.

3.9. Instalación y ejecución de PLSS

En esta sección se explica el procedimiento de instalación y el funcionamiento del programa con algún ejemplo.

El único requisito previo para el uso de PLSS es la instalación de Maude (Clavel et al., 2020). Es necesaria la versión 3.1 o posterior. Después solo hay que descargar los ficheros del repositorio en un único directorio.

Respecto del funcionamiento del programa, una vez descargado el código del repositorio, en primer lugar se debe ejecutar el módulo principal `plss.maude` en Maude con la opción `allow-files` para poder abrir ficheros externos:

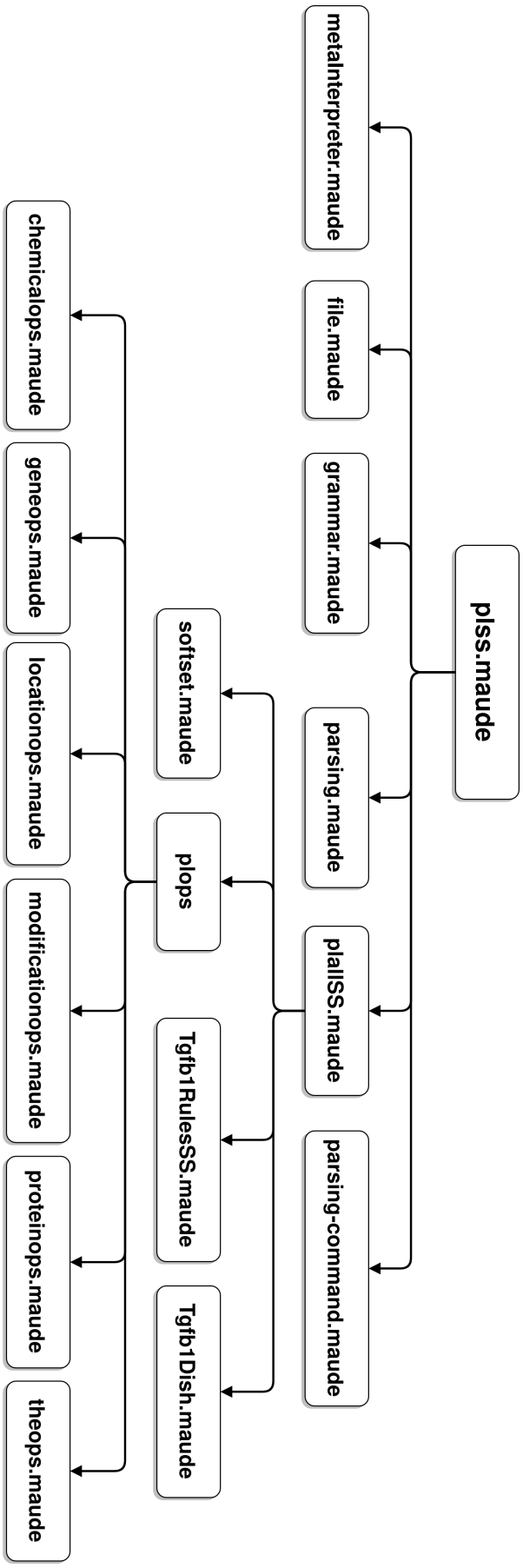

Figura 3.2: Dependencia de ficheros Maude.

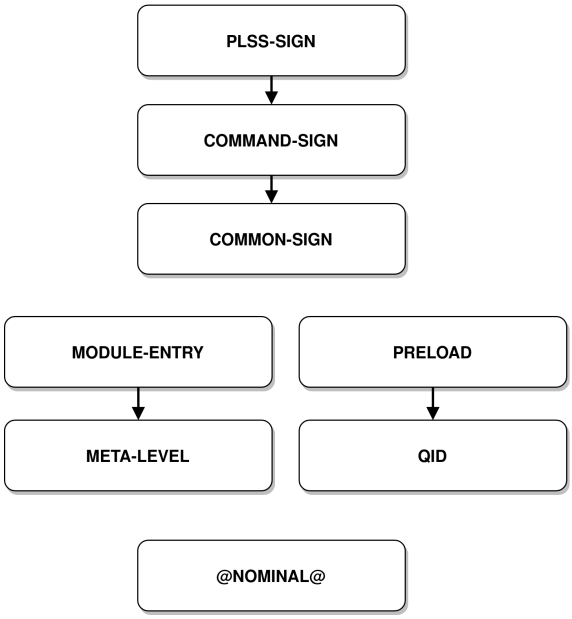

Figura 3.3: Dependencia de otros módulos.

```
$ maude -allow-files plss
                \|||||||||||||||||||/
              --- Welcome to Maude ---
                /|||||||||||||||||||\
      Maude 3.1 built: Oct 12 2020 20:13:41
      Copyright 1997-2020 SRI International
             Wed Jan  6 19:04:10 2021
==========================================
erewrite in PLSS : run .
PLSS>
```

Se ejecutó el comando `erewrite` para la configuración `run`, que está formado por las constantes de configuración, con ello se creará un intérprete y se enviará el prompt del programa al flujo de salida:

```
< plss : PLSS | file: null, out: "", state: init, load: modList, uload:
    umodList, input: empty, module: userModule, strat: 'undefZero >
```

Ejemplos con los comandos `red`, `softrew`, `softnarrowrew` y `softnarrowsearch`:

```
PLSS> red (PD({CLm | empty}) ([cdc6 = 1]))

Result: PD(
{CLm | empty})[cdc6 = 1]

PLSS> red (Tgfb1Dish)

Result: PD(
{CLc | Abl1 Akt1 Atf2 Erks Fak1 Jnks Mekk1 Mlk3 P38s Pak2 Pml Smad2 Smad3 Smad4
     Smurf1 Smurf2 Tab1 Tab2 Tab3 Tak1 Traf6 Zfyve16}
{CLi |[Cdc42 - GDP][Hras - GDP][Rac1 - GDP]}
{CLm | empty}
{CLo | empty}
{NUc | Cdc6-gene Cdkn1a-gene Cdkn2b-gene Col1a1-gene Col3a1-gene Csrp2-gene
    Cst6-gene Ctdsp1 Ctgf-gene Dst-gene Ets1 Fn1-gene Gfi1-gene Mmp2-gene
    Mmp9-gene Mylk-gene Pai1-gene Pthlh-gene RoRc-gene Smad6-gene Smad7
```

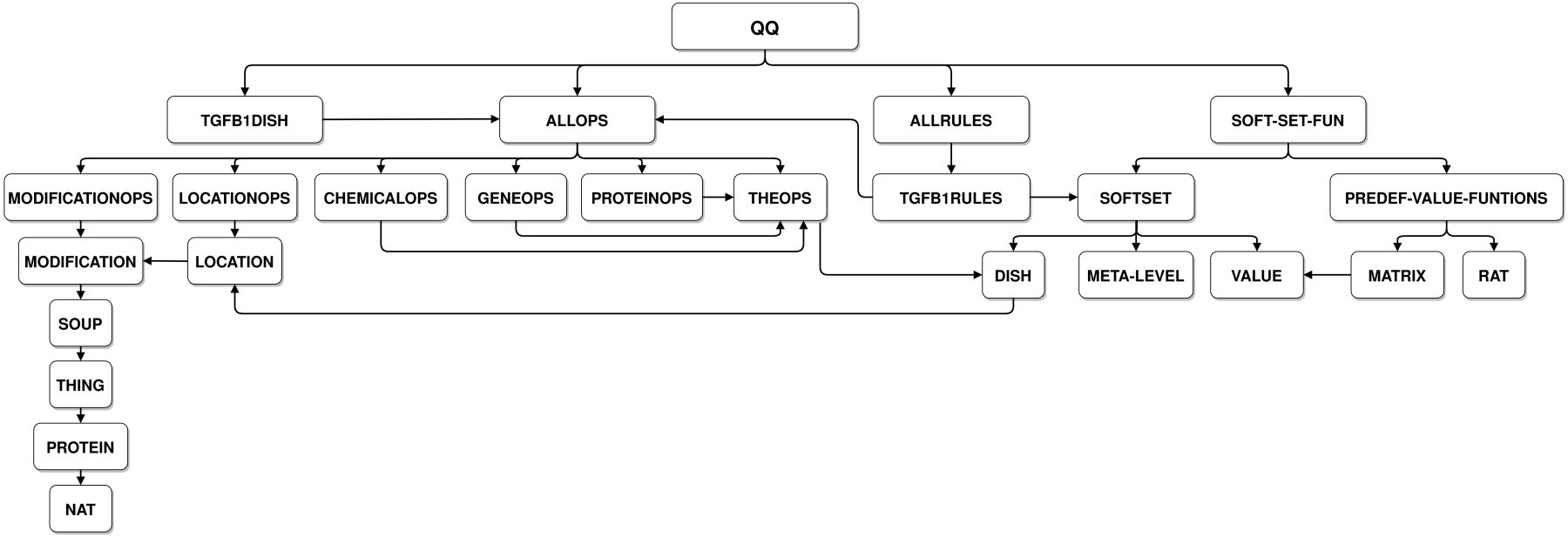

Figura 3.4: Dependencia de módulos específicos de Pathway Logic.

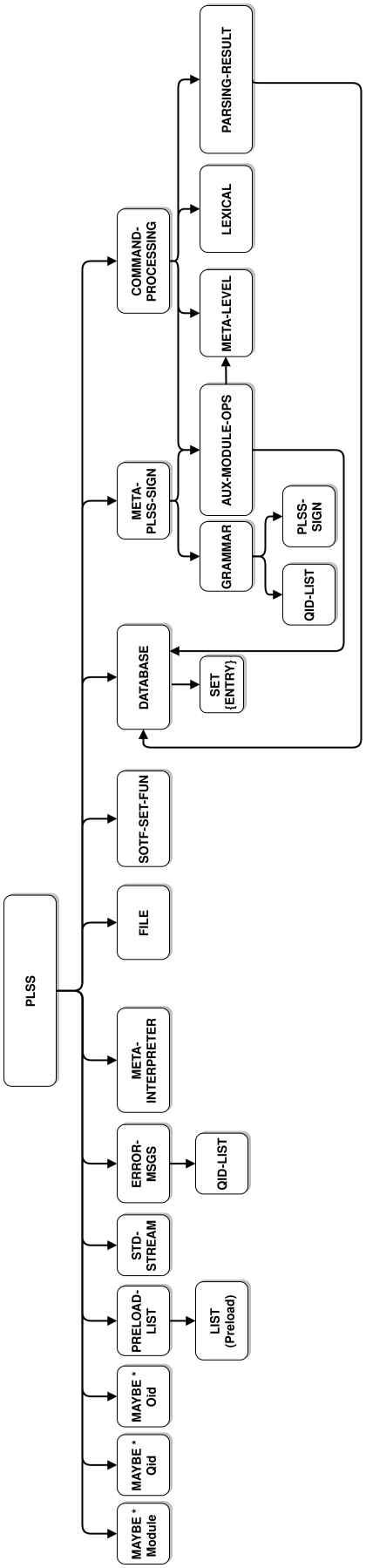

Figura 3.5: Dependencia de módulos de uso general de PLSS.

```
    Smad7-gene Tgfb1-gene Timp1-gene}
{Tgfb1RC | TgfbR1 TgfbR2}
{XOut | Tgfb1})

PLSS> red (Tgfb1Dish  ([cdc6 = 0], [cdkn1a = 0], [cdkn2b = 0], [col1a1 = 0],
    [col3a1 = 0], [ctgf = *], [fn1 = *], [mmp2 = 0], [pai1 = *], [smad6 = 0],
    [smad7 = 0], [tgfb1 = 0], [timp1 = 0], [cst6 = 0], [dst = 0], [mmp9 = *],
    [mylk = 0], [pthlh = *], [gfi1 = 0], [csrp2 = 0], [rorc = *]))

Result: PD(
{CLc | Abl1 Akt1 Atf2 Erks Fak1 Jnks Mekk1 Mlk3 P38s Pak2 Pml Smad2 Smad3 Smad4
     Smurf1 Smurf2 Tab1 Tab2 Tab3 Tak1 Traf6 Zfyve16}
{CLi |[Cdc42 - GDP][Hras - GDP][Rac1 - GDP]}
{CLm | empty}
{CLo | empty}
{NUc | Cdc6-gene Cdkn1a-gene Cdkn2b-gene Col1a1-gene Col3a1-gene Csrp2-gene
    Cst6-gene Ctdsp1 Ctgf-gene Dst-gene Ets1 Fn1-gene Gfi1-gene Mmp2-gene
    Mmp9-gene Mylk-gene Pai1-gene Pthlh-gene RoRc-gene Smad6-gene Smad7
    Smad7-gene Tgfb1-gene Timp1-gene}
{Tgfb1RC | TgfbR1 TgfbR2}
{XOut | Tgfb1})[cdc6 = 0],[cdkn1a = 0],[cdkn2b = 0],[col1a1 = 0],[col3a1 = 0],
    [cst6 = 0],[ctgf = *],[dst = 0],[fn1 = *],[gfi1 = 0],[mmp2 = 0],[mmp9 = *],
    [mylk = 0],[pai1 = *],[pthlh = *],[rorc = *],[smad6 = 0],[smad7 = 0],
    [csrp2 = 0],[tgfb1 = 0],[timp1 = 0]
```

```
PLSS> softrew (Tgfb1Dish ([cdc6 = 0]))

Result: PD(
{CLc | Akt1 Atf2 Erks Fak1 Jnks Mekk1 Mlk3 P38s Pml Smad2 Smad3 Tab1 Tab2 Tab3
    Tak1 Traf6 Zfyve16[Abl1 - act]}
{CLi |([Pak2 - act] :[Rac1 - GTP])[Cdc42 - GTP][Hras - GTP]}
{CLm | empty}
{CLo | empty}
{NUc | Cdc6-gene Cdkn1a-gene Cdkn2b-gene Col1a1-gene Col3a1-gene Csrp2-gene
    Cst6-gene Ctdsp1 Ctgf-gene Dst-gene Ets1 Fn1-gene Gfi1-gene Mmp2-gene
    Mmp9-gene Mylk-gene Pai1-gene Pthlh-gene RoRc-gene Smad4 Smad6-gene
    Smad7-gene Tgfb1-gene Timp1-gene}
{Tgfb1RC | Smad7 Smurf1 Smurf2 Tgfb1 :[TgfbR1 - ubiq] :[TgfbR2 - act]}
{XOut | empty})
```

```
PLSS> softnarrowrew (PD({CLc | empty})  ([cdc6 = 0]))

Result: PD(
{CLc | empty})
```

```
PLSS> softnarrowsearch (SD:SoftDish) (PD({XOut | Tgfb1}) ([cdc6 = 1]))

Result (SoftDish-term): #1:SoftDish
Substitution1: SD <- #1:SoftDish
Substitution2: #1 <- PD({XOut | Tgfb1})[cdc6 = 1]
Term-ini (T): SD:SoftDish
Term-patron (T1): PD({XOut | Tgfb1})[cdc6 = 1]
```

Por último, existen dos comandos `exit` y `q` que terminarán con la ejecución y muestran un mensaje de despedida:

```
PLSS> exit

Thanks for using PLSS!
rewrites: 4234 in 580ms cpu (89892ms real) (7299 rewrites/second)
result Portal: <>
```

Capítulo 4

Analisis de resultados

En este capítulo se realiza un análisis comparativo de los resultados obtenidos frente a los del uso estándar de la reescritura. También se realiza entre los distintos comandos de simplificación que se han implementado. Por último, se analiza el rendimiento de la aplicación.

4.1. Trabajos relacionados

Este trabajo es multidisciplinar y ha necesitado de áreas de conocimiento muy distintas entre sí: lógica de reescritura, biología computacional y soft computing. Existen varios trabajos relacionados para las distintas partes del TFM. En lo referente a la programación, también existen distintas facetas: el entorno de ejecución, la utilización de la reflexión, el narrowing, etc.

Para la parte del **entorno de ejecución** del PLSS, la referencia clave es el manual de Maude 3.1 (Clavel et al., 2020, Capítulo 19). En la sección "An execution environment for Mini-Maude using IO and meta-interpreters" se implementa un pequeño entorno de ejecución que gestiona la entrada/salida y los metaintérpretes.

La referencia fundamental del trabajo de fin de máster es el artículo de Santos-Buitrago et al. (2019). Este artículo ha sido la base de la parte de **toma de decisiones y soft sets**. Los autores han desarrollado sobre un modelo de Pathway Logic y se utilizan soft sets para guiar las reescrituras.

En este TFM se ha necesitado utilizar la reflexión de Maude. Con respecto al **metanivel** y sus operaciones, las fuentes bibliográficas fundamentales para la implementación del PLSS han sido: la tesis doctoral de Pita (2003) y la tesis de máster de Cuenca Ortega (2013), además del manual de Maude.

Para las búsquedas con **narrowing** se han utilizado los trabajos de Santiago Escobar y Adrián Riesco (Escobar et al., 2005; Viciano Negre, 2012; Riesco, 2012).

Por último, en lo referente a **Pathway Logic** y la manipulación de sus modelos biológicos con Maude, los trabajos fundamentales se deben a Carolyn Talcott. Los trabajos relacionados que he seguido para la realización de búsquedas y reescrituras con sobre Pathway Logic son de Adrián Riesco con Carolyn Talcott y otros (Riesco et al., 2020; Santos-Buitrago et al., 2017; Santos-García et al., 2016).

4.2. Análisis y comparación con otros sistemas de búsqueda

En esta sección se distinguen las diferencias entre los comandos de búsqueda han sido creados en este programa. En concreto hacemos referencia a "search"(red) y a "softnarrow-search".

```
+ IMPLEMENTACIÓN

( Ejecución estrategias/rew y el paper de IEEE Access.

Habría que tomar varios casos de softdish y para cada uno:
1. Ejecutar los comandos red/search y softnarrowsearch
2. Comprobar que los resultados son distintos
Explicar el significado de cada función (se da más peso a los
valores desconocido, o menos, etc.)
3. ¿¿Construir la matriz del soft set asociado??
4. Explicar y analizar las diferencias encontradas )
```

ANTIGUO: Ejecución estrategias/rew y el paper de IEEE Access.

Habría que tomar varios casos de softdish y para cada uno:

(1) ejecutarlos en cada sistema con todas las estrategias

(2) comprobar que los resultados son distintos para cada función de estrategia. Explicar el significado de cada función (se da más peso a los valores desconocido, o menos, etc.)

(3) construir la matriz del soft set asociado

(4) calcular matemáticamente y analizar el cv según cada método

4.3. Análisis y comparación entre comandos de reescritura

Para comenzar comentaré brevemente de que trata esta subsección. En PLSS se han implementado distintos comandos de simplificación con soft reglas. Entre ellas se encuentran "softrewz "softnarrowrew".

```
+ IMPLEMENTACIÓN

( Habría que tomar varios casos de softdish y para cada uno:
1. Ejecutar cada uno (softrew y softnarrowrew) con todas las estrategias
2. Comprobar que los resultados son distintos para cada función de estrategia
Explicar el significado de cada función (se da más peso a los
valores desconocido, o menos, etc.)
3. Comparar tiempos )
```

4.4. Análisis del rendimiento

Por último, se comentan algunos aspectos sobre el rendimiento del programa. Se muestra una comparativa entre resultados de reescritura con los distintos choice values implementados.

(Se muestra también una comparativa entre resultados de reescritura guiados para los distintos choice values implementados y con el sistema de reescritura nativo de Maude.

Tiempo ejecución: para cada función PLSS y con rew/search Analizar que pasa si... - Si crece el número de atributos - Si crece el número de valores desconocidos - Si crece el número de términos alcanzables)

Capítulo 5

Conclusiones y trabajo futuro

"Comience por el principio", dijo el rey con gran gravedad,
"y continúe hasta que llegue al final: entonces deténgase"
— Lewis Carroll, Alicia en el País de las Maravillas

Por último, en este capítulo se presentan las conclusiones de este proyecto. Se incluyen también las dificultades encontradas en el desarrollo del trabajo. En la segunda sección se enumeran las posibles mejoras del proyecto para un trabajo futuro.

5.1. Conclusiones del trabajo

Una vez cerrado el proyecto, es el momento de presentar las conclusiones.

En primer lugar, en base a las pruebas realizadas sobre el programa final, se ha conseguido alcanzar de forma satisfactoria el objetivo esencial que se fijó en las fases de planificación y especificación del proyecto. El objetivo fundamental consiste en desarrollar e implementar nuevas variantes de reescrituras basadas en soft sets en el contexto de Pathway Logic.

Los objetivos intermedios del proyecto se pueden concretar en: (1) implementación de una aplicación en Maude con su sistema de comandos propios; (2) integración con Pathway Logic; y (3) definición de estrategias de reescritura para guiar la reescritura.

Cada uno de ellos ha planteado los retos de investigar en ese área concreta e integrar esos conocimientos dentro del lenguaje Maude. Durante la ejecución del proyecto, son de destacar el esfuerzo en el estudio del metaintérprete y de la modelización de sistemas biológicos basados en reglas de Pathway Logic.

Por otra parte, concretando los objetivos parciales, la aplicación PLSS permite ejecutar dentro de Maude sus propios comandos (por ejemplo, simplificación, reescritura, etc.). Esta es una funcionalidad novedosa de las últimas versiones de Maude mediante su metaintérprete.

Además se ha adaptado el modelo de la ruta de señalización celular TGFB1 (factor de crecimiento transformante beta 1) desarrollado en Pathway Logic. La integración de su estructura y su base de conocimiento ha sido posible gracias a que el diseño de Pathway Logic está basado en el lenguaje Maude.

La definición de las nuevas estrategias de reescritura para guiar la reescritura se han

basado en las propuestas en el reciente artículo de Santos-Buitrago et al. (2019). De hecho, las descritas en ese artículo también se han implementado en PLSS (`undefZero`, `undefOne` y `undefSemi`).

Por último, se han validado los resultados haciendo un análisis comparativo con otros sistemas de reescritura. Se muestran las diferencias entre las distintas funciones de estrategia dependiendo del valor de los atributos.

También se ha analizado el rendimiento del programa en lo referente al tiempo de ejecución. Se observa que todas las funciones, excepto `undefWCol`, se ejecutan en un tiempo similar a la reescritura estándar de Maude.

Las principales dificultades encontradas en el desarrollo del proyecto han sido: (1) el aprendizaje de las nuevas tecnologías implicadas en el proyecto, especialmente el manejo de los soft sets y Pathway Logic; y (2) la dificultad conceptual de la reflexión en Maude y del manejo del metaintérprete.

El código elaborado para este proyecto está disponible en el repositorio de GitHub `https://github.com/rsantosb/TFM-PLSS`, bajo la licencia MIT (`https://opensource.org/licenses/MIT`).

5.2. Líneas de Trabajo Futuro

Durante la elaboración del trabajo han aparecido numerosos puntos de mejora e ideas que sería interesante realizar en un futuro. Algunas de esas líneas de trabajo futuro son:

- Implementar nueva funciones de estrategia para la toma de decisión, que se pueden ajustar mejor a otros tipos de problemas.
- Incorporar un lenguaje de especificaciones que permita a los usuarios del programa el desarrollo de funciones propias.
- Trabajar con otras extensiones de fuzzy sets o soft sets.
- Desarrollar nuevos comandos y funcionalidades en PLSS.
- Permitir seleccionar el modelo de Pathway Logic con el que trabajar en PLSS.
- Investigar sobre los atributos que puedan ser significativos en la dinámica de la modelización celular.

Chapter 6

Introduction

"Why, sometimes I've believed as many as six impossible things before breakfast"
— Lewis Carroll, Alice in Wonderland

This chapter begins with a brief introduction to the topic of this project. It is followed by a section on the objectives and work plan. Finally, the organization of the dissertation is described.

6.1. Motivation

Many real-life problems require the use of inaccurate or unknown data. Their analysis must involve the application of mathematical principles capable of capturing these characteristics. In the field of Artificial Intelligence, the theory of soft sets provides an appropriate framework for decision making in situations of lack of information.

Pathway Logic is a tool designed to deal with symbolic biological systems. Numerous cellular signaling pathway models have been developed with this platform. This tool facilitates the understanding of complex biological systems and the verification of hypotheses in the design of experiments.

One of the characteristics of the Maude language is its ability to naturally express a wide range of applications, for example, Pathway Logic. The new features in the meta-interpreter and input/output management of the new versions allow for efficient management of a runtime environment within Maude.

Maude's standard rewrite system provides all search tree terms that are attainable. In some applications, it is desirable to choose only one option from all the possible ones. Based on this situation, this project aims to implement an execution environment with different strategies in the rewriting system that allows choosing the best rewritten term. Taking into account that the available information may be incomplete, the strategies have been defined through the theory of soft sets.

With the intention of providing an application with real utility, the implementation of this rewriting system has been developed with Pathway Logic's biological models.

6.2. Objectives

The main objective of this project is to develop and implement new variants of rewrites based on soft sets in the context of Pathway Logic.

Intermediate goals for achieving this objective can be specified at:

1. Implement an application in Maude with its own command system.
2. Integrate the application with Pathway Logic.
3. Define strategies to guide rewriting.

The work done is classified into the following areas:

- Rewriting logic.
- Soft computing, especially decision making under uncertainty with soft sets.
- Modeling of biological systems based on rules.

6.3. Work plan

The realization of the project has been supported by the work plan with the director. The first meetings were dedicated to the concretion of the theme and objectives of the project. Afterwards, every two weeks we had meetings in which the tasks defined in the previous meeting were reviewed. On the part of the director, corrections were made and improvements proposed. In each meeting the work to be done during the following two weeks was specified.

Apart from the regular meetings, occasional doubts about any aspect of the work have been resolved by the director by e-mail. At all times, the answers have been accurate and the guidance valuable.

The milestones that were established to achieve the objectives are:

1. Search and initial definition of the subject of the work.
2. Going deeper into the Maude language.
3. Research on Pathway Logic and soft sets.
4. Design and implementation of a basic prototype of the application.
5. Progressive extension of the functionalities (both in the commands and in the strategies).
6. Testing.
7. Drafting of the project report.

6.4. Organization of the dissertation

The dissertation is divided into the following chapters:

- **Chapter 1**: Introduction. The theme of the TFM is introduced, the objectives of the work are described and the work plan followed to achieve the objectives is detailed.
- **Chapter 2**: State of the art. This chapter introduces the areas on which the work is based: the Maude programming language, the Maude meta-interpreter, soft set theory and Pathway Logic.
- **Chapter 3**: Description of the work. This chapter describes the work done in the project **PLSS**: **P**athway **L**ogic with **S**oft **S**ets.
- **Chapter 4**: Results Analysis. This chapter includes some comments on the performance of the application and a comparative analysis of the results obtained with those of the standard use of rewriting.
- **Chapter 5**: Conclusions and Future Work. Finally, the conclusions and lines for future work have been set out.

The code developed for this project is available in the GitHub repository `https://github.com/rsantosb/TFM-PLSS`, under the MIT license (`https://opensource.org/licenses/MIT`).

Chapter 7

Conclusions and future work

"Begin at the beginning," the King said, very gravely,
"and go on till you come to the end: then stop"
— Lewis Carroll, Alice in Wonderland

This chapter presents the conclusions of this project. It also includes the difficulties encountered in the development of the work. The second section lists possible improvements to the project for future work.

7.1. Conclusions of the work

Once the project is closed, it is time to present the conclusions.

Firstly, on the basis of the tests carried out on the final programme, the essential objective set in the planning and specification phases of the project has been successfully achieved. The fundamental objective is to develop and implement new variants of rewrites based on soft sets in the context of Pathway Logic.

The intermediate objectives of the project can be specified as: (1) implementation of an application in Maude with its own command system; (2) integration with Pathway Logic; and (3) definition of rewriting strategies to guide the rewrite.

Each of them has posed the challenges of researching that particular area and integrating that knowledge into the Maude language. During the execution of the project, the effort in the study of the meta-interpreter and the modelling of biological systems based on Pathway Logic rules is noteworthy.

Moreover, by specifying the partial objectives, the PLSS application allows you to execute your own commands within Maude (e.g. simplification, rewriting, etc.). This is a new feature of the latest versions of Maude through its meta-interpreter.

In addition, the TGFB1 (transforming growth factor beta 1) cell signalling pathway model developed in Pathway Logic has been adapted. The integration of its structure and knowledge base has been possible thanks to the fact that the design of Pathway Logic is based on the Maude language.

The definition of new rewriting strategies to guide rewriting has been based on the proposals in the recent Santos-Buitrago et al. (2019). In fact, those described in that

article have also been implemented in PLSS (`undefZero`, `undefOne` and `undefSemi`).

Finally, the results have been validated by making a comparative analysis with other rewriting systems. The differences between the various strategy functions are shown depending on the value of the attributes.

The performance of the programme in terms of execution time has also been analysed. It was found that all functions, except `undefWCol`, run at a similar time to Maude's standard rewrite.

The main difficulties encountered in the development of the project have been: (1) the learning of the new technologies involved in the project, especially the handling of the soft sets and Pathway Logic; and (2) the conceptual difficulty of reflecting on Maude and handling the meta-interpreter.

The code developed for this project is available in the GitHub repository `https://github.com/rsantosb/TFM-PLSS`, under the MIT license (`https://opensource.org/licenses/MIT`).

7.2. Lines of future work

During the elaboration of the work, numerous points of improvement and ideas have appeared that would be interesting to carry out in the future. Some of these lines of future work are:

- Implement new strategy functions for decision making, which can be better adjusted to other types of problems.
- Incorporate a specification language that allows program users to develop their own functions.
- Work with other fuzzy set extensions or soft sets.
- Develop new commands and functionalities in PLSS.
- Allow you to select the Pathway Logic model to work with in PLSS.
- Research on the attributes that can be significant in the dynamics of cell modelling.

Bibliografía

Cuanto menos se lee,
más daño hace lo que se lee.

Miguel de Unamuno

Aktaş, H. y Çağman, N. Soft sets and soft groups. *Information Sciences*, vol. 177, páginas 2726–2735, 2007.

Alcantud, J. C. R. A novel algorithm for fuzzy soft set based decision making from multiobserver input parameter data set. *Information Fusion*, vol. 29, páginas 142–148, 2016a.

Alcantud, J. C. R. Some formal relationships among soft sets, fuzzy sets, and their extensions. *International Journal of Approximate Reasoning*, vol. 68, páginas 45–53, 2016b.

Alcantud, J. C. R. y Santos-García, G. A new criterion for soft set based decision making problems under incomplete information. *International Journal of Computational Intelligence Systems*, vol. 10, páginas 394–404, 2017.

Ali, M., Feng, F., Liu, X., Min, W. K. y Shabir, M. On some new operations in soft set theory. *Computers & Mathematics with Applications*, vol. 57(9), páginas 1547–1553, 2009.

Ali, M., Mahmood, T., Rehman, M. M. U. y Aslam, M. F. On lattice ordered soft sets. *Applied Soft Computing*, vol. 36, páginas 499–505, 2015.

Bernardo, M., Degano, P. y Zavattaro, G., editores. *Formal Methods for Computational Systems Biology, 8th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2008, Bertinoro, Italy, June 2-7, 2008, Advanced Lectures*, vol. 5016 de *Lecture Notes in Computer Science*. Springer, 2008. ISBN 978-3-540-68892-1.

Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Rubio, R. y Talcott, C. L. *Maude manual (version 3.1)*, 2020. `http://maude.cs.illinois.edu/w/images/6/62/Maude-3.1-manual.pdf`.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J. y Talcott, C. L. *All about Maude - A high-performance logical framework, how to*

specify, program and verify systems in Rewriting Logic, vol. 4350 de *Lecture Notes in Computer Science*. Springer, 2007. ISBN 978-3-540-71940-3.

Cuenca Ortega, Á. *Técnicas de Refactorización para Maude*. Proyecto Fin de Carrera, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 2013.

Eker, S., Knapp, M., Laderoute, K., Lincoln, P., Meseguer, J. y Sönmez, M. K. Pathway Logic: Symbolic analysis of biological signaling. En *Proceedings of the 7th Pacific Symposium on Biocomputing, PSB 2002, Lihue, Hawaii, USA, January 3-7, 2002* (editado por R. B. Altman, A. K. Dunker, L. Hunter y T. E. Klein), páginas 400–412. 2002a.

Eker, S., Knapp, M., Laderoute, K., Lincoln, P. y Talcott, C. L. Pathway Logic: Executable models of biological networks. *Electronic Notes in Theoretical Computer Science*, vol. 71, páginas 144–161, 2002b.

Eker, S., Laderoute, K., Lincoln, P., Sriram, M. G. y Talcott, C. L. Representing and simulating protein functional domains in signal transduction using Maude. En *Computational Methods in Systems Biology, First International Workshop, CMSB 2003, Roverto, Italy, February 24-26, 2003, Proceedings* (editado por C. Priami), vol. 2602 de *Lecture Notes in Computer Science*, páginas 164–165. Springer, 2003. ISBN 3-540-00605-2.

Escobar, S., Meseguer, J. y Thati, P. Natural narrowing for general term rewriting systems. En *International Conference on Rewriting Techniques and Applications. RTA 2005* (editado por J. Giesl), vol. 3467 de *Lecture Notes in Computer Science*, páginas 279–293. Springer, 2005. ISBN 978-3-540-25596-3.

Feng, F. y Li, Y. Soft subsets and soft product operations. *Information Sciences*, vol. 232, páginas 44–57, 2013.

Han, B.-H., Li, Y., Liu, J., Geng, S. y Li, H. Elicitation criterions for restricted intersection of two incomplete soft sets. *Knowledge-Based Systems*, vol. 59, páginas 121–131, 2014.

Knapp, M., Briesemeister, L., Eker, S., Lincoln, P., Poggio, A., Talcott, C. L. y Laderoute, K. Pathway Logic helping biologists understand and organize pathway information. En *Fourth International IEEE Computer Society Computational Systems Bioinformatics Conference Workshops & Poster Abstracts (CSB 2005 Workshops), Stanford, California, USA, 8-11 August, 2005*, páginas 155–156. IEEE Computer Society, 2005. ISBN 0-7695-2442-7.

Maji, P., Biswas, R. y Roy, A. Fuzzy soft sets. *Journal of Fuzzy Mathematics*, vol. 9, páginas 589–602, 2001.

Maji, P., Biswas, R. y Roy, A. An application of soft sets in a decision making problem. *Computers and Mathematics with Applications*, vol. 44, páginas 1077–1083, 2002.

Maji, P., Biswas, R. y Roy, A. Soft set theory. *Computers and Mathematics with Applications*, vol. 45, páginas 555–562, 2003.

Molodtsov, D. Soft set theory - first results. *Computers and Mathematics with Applications*, vol. 37, páginas 19–31, 1999.

Nakao, A., Afrakhte, M., Morn, A., Nakayama, T., Christian, J. L., Heuchel, R., Itoh, S., Kawabata, M., Heldin, N.-E., Heldin, C.-H. et al. Identification of Smad7, a TGFβ-inducible antagonist of TGF-β signalling. *Nature*, vol. 389(6651), páginas 631–635, 1997.

Pita, I. *Técnicas de especificación formal de sistemas orientados a objetos basadas en lógica de reescritura*. Tesis Doctoral, Facultad de Matemáticas, Universidad Complutense de Madrid, 2003.

Qin, H., Ma, X., Herawan, T. y Zain, J. Data filling approach of soft sets under incomplete information. En *Intelligent Information and Database Systems* (editado por N. Nguyen, C.-G. Kim y A. Janiak), vol. 6592 de *Lecture Notes in Computer Science*, páginas 302–311. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-20041-0.

Qin, K., Meng, D., Pei, Z. y Xu, Y. Combination of interval set and soft set. *International Journal of Computational Intelligence Systems*, vol. 6(2), páginas 370–380, 2013.

Riesco, A. Using narrowing to test maude specifications. En *Rewriting Logic and Its Applications - 9th International Workshop, WRLA 2012, Held as a Satellite Event of ETAPS, Tallinn, Estonia, March 24-25, 2012, Revised Selected Papers* (editado por F. Durán), vol. 7571 de *Lecture Notes in Computer Science*, páginas 201–220. Springer, 2012.

Riesco, A., Santos-Buitrago, B., Knapp, M., Santos-García, G., Galilea, E. H. y Talcott, C. L. Fuzzy matching for cellular signaling networks in a choroidal melanoma model. En *Practical Applications of Computational Biology & Bioinformatics, 14th International Conference (PACBB 2020), LÁquila, Italy, 17-19 June, 2020* (editado por G. Panuccio, M. Rocha, F. Fdez-Riverola, M. S. Mohamad y R. Casado-Vara), vol. 1240 de *Advances in Intelligent Systems and Computing*, páginas 80–90. Springer, 2020.

Santos-Buitrago, B., Riesco, A., Knapp, M., Alcantud, J. C. R., Santos-García, G. y Talcott, C. L. Soft set theory for decision making in computational biology under incomplete information. *IEEE Access*, vol. 7, páginas 18183–18193, 2019.

Santos-Buitrago, B., Riesco, A., Knapp, M., Santos-García, G. y Talcott, C. L. Reverse inference in symbolic systems biology. En *11th International Conference on Practical Applications of Computational Biology & Bioinformatics, PACBB 2017, Porto, Portugal, 21-23 June, 2017* (editado por F. Fdez-Riverola, M. S. Mohamad, M. P. Rocha, J. F. D. Paz y T. Pinto), vol. 616 de *Advances in Intelligent Systems and Computing*, páginas 101–109. Springer, 2017. ISBN 978-3-319-60815-0.

Santos-García, G., Talcott, C. L., Riesco, A., Santos-Buitrago, B. y Rivas, J. D. L. Role of nerve growth factor signaling in cancer cell proliferation and survival using a reachability analysis approach. En *10th International Conference on Practical Applications of Computational Biology & Bioinformatics, PACBB 2016, Sevilla, Spain, 1–3 June 2016* (editado por M. S. Mohamad, M. P. Rocha, F. Fdez-Riverola, F. J. D. Mayo y J. F. D. Paz), vol. 477 de *Advances in Intelligent Systems and Computing*, páginas 173–181. Springer, 2016.

Talcott, C. L. Symbolic modeling of signal transduction in Pathway Logic. En *Proceedings of the Winter Simulation Conference WSC 2006, Monterey, California, USA,*

December 3-6, 2006 (editado por L. F. Perrone, B. Lawson, J. Liu y F. P. Wieland), páginas 1656–1665. WSC, 2006. ISBN 1-4244-0501-7.

Talcott, C. L. Pathway Logic. En *Formal Methods for Computational Systems Biology, 8th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2008, Bertinoro, Italy, June 2-7, 2008, Advanced Lectures* (editado por M. Bernardo, P. Degano y G. Zavattaro), vol. 5016 de *Lecture Notes in Computer Science*, páginas 21–53. Springer, 2008. ISBN 978-3-540-68892-1.

Talcott, C. L. The Pathway Logic formal modeling system: Diverse views of a formal representation of signal transduction. En *IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2016, Shenzhen, China, December 15-18, 2016* (editado por T. Tian, Q. Jiang, Y. Liu, K. Burrage, J. Song, Y. Wang, X. Hu, S. Morishita, Q. Zhu y G. Wang), páginas 1468–1476. IEEE Computer Society, 2016. ISBN 978-1-5090-1611-2.

Talcott, C. L. y Dill, D. L. The Pathway Logic Assistant. En *Proceedings of the Third International Workshop on Computational Methods in Systems Biology, CMSB'05, Edinburgh, Scotland, April 2005* (editado por G. Plotkin), vol. 3, páginas 228–239. 2005.

Talcott, C. L. y Dill, D. L. Multiple representations of biological processes. En *Transactions on Computational Systems Biology* (editado por C. Priami y G. D. Plotkin), vol. 4220 de *Lecture Notes in Computer Science*, páginas 221–245. Springer, 2006. ISBN 3-540-45779-8.

Talcott, C. L., Eker, S., Knapp, M., Lincoln, P. y Laderoute, K. Pathway Logic modeling of protein functional domains in signal transduction. En *Proceedings of the 2nd IEEE Computer Society Bioinformatics Conference, CSB 2003, Stanford, California, August 11-14, 2003* (editado por P. Markstein y Y. Xu), páginas 618–619. IEEE Computer Society, 2003. ISBN 0-7695-2000-6.

Talcott, C. L. y Knapp, M. Explaining response to drugs using Pathway Logic. En *Computational Methods in Systems Biology - 15th International Conference, CMSB 2017, Darmstadt, Germany, September 27-29, 2017, Proceedings* (editado por J. Feret y H. Koeppl), vol. 10545 de *Lecture Notes in Computer Science*, páginas 249–264. Springer, 2017. ISBN 978-3-319-67471-1.

Viciano Negre, P. *Extensiones a la comprobación de satisfacibilidad de restricciones*. Proyecto Fin de Carrera, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 2012.

Wang, F., Li, X. y Chen, X. Hesitant fuzzy soft set and its applications in multicriteria decision making. *Journal of Applied Mathematics*, página Article ID 643785, 2014.

Zadeh, L. Fuzzy sets. *Information and Control*, vol. 8, páginas 338–353, 1965.

Zhang, X. On interval soft sets with applications. *International Journal of Computational Intelligence Systems*, vol. 7(1), páginas 186–196, 2014.

Zou, Y. y Xiao, Z. Data analysis approaches of soft sets under incomplete information. *Knowledge-Based Systems*, vol. 21(8), páginas 941–945, 2008.

Apéndice A

Ejemplos de ejecución

Ejecución del programa desde Maude:

```
$ maude -allow-files plss
$ maude -allow-files plss
               \||||||||||||||||||/
             --- Welcome to Maude ---
               /||||||||||||||||||\
      Maude 3.1 built: Oct 12 2020 20:13:41
      Copyright 1997-2020 SRI International
             Wed Jan  6 19:04:10 2021
==========================================
erewrite in PLSS : run .
PLSS>
```

Ejemplos con el comando `red`:

```
PLSS> red (Tgfb1Dish)

Result: PD(
{CLc | Abl1 Akt1 Atf2 Erks Fak1 Jnks Mekk1 Mlk3 P38s Pak2 Pml Smad2 Smad3 Smad4
    Smurf1 Smurf2 Tab1 Tab2 Tab3 Tak1 Traf6 Zfyve16}
{CLi |[Cdc42 - GDP][Hras - GDP][Rac1 - GDP]}
{CLm | empty}
{CLo | empty}
{NUc | Cdc6-gene Cdkn1a-gene Cdkn2b-gene Col1a1-gene Col3a1-gene Csrp2-gene
    Cst6-gene
    Ctdsp1 Ctgf-gene Dst-gene Ets1 Fn1-gene Gfi1-gene Mmp2-gene Mmp9-gene Mylk-
        gene
    Pai1-gene Pthlh-gene RoRc-gene Smad6-gene Smad7 Smad7-gene Tgfb1-gene Timp1
        -gene}
{Tgfb1RC | TgfbR1 TgfbR2}
{XOut | Tgfb1})

PLSS> softrew (Tgfb1Dish ([cdc6 = 0]))

Result: PD(
{CLc | Akt1 Atf2 Erks Fak1 Jnks Mekk1 Mlk3 P38s Pml Smad2 Smad3 Tab1 Tab2 Tab3
    Tak1
    Traf6 Zfyve16[Abl1 - act]}
{CLi |([Pak2 - act] :[Rac1 - GTP])[Cdc42 - GTP][Hras - GTP]}
{CLm | empty}
{CLo | empty}
{NUc | Cdc6-gene Cdkn1a-gene Cdkn2b-gene Col1a1-gene Col3a1-gene Csrp2-gene
    Cst6-gene
    Ctdsp1 Ctgf-gene Dst-gene Ets1 Fn1-gene Gfi1-gene Mmp2-gene Mmp9-gene Mylk-
        gene
```

```
    Pai1-gene Pthlh-gene RoRc-gene Smad4 Smad6-gene Smad7-gene Tgfb1-gene Timp1
        -gene}
{Tgfb1RC | Smad7 Smurf1 Smurf2 Tgfb1 :[TgfbR1 - ubiq] :[TgfbR2 - act]}
{XOut | empty})
```

Ejemplos con el comando `softrew`:

```
PLSS> softrew (PD({CLm | empty}) ([cdkn1a = 0], [cdc6 = 0]))

Result: PD(
{CLm | empty})

PLSS> softrew (PD({CLm | empty}) ([cdkn1a = 0]))

Result: PD(
{CLm | empty})
```

Ejemplos con el comando `load`:

Listing A.1: Fichero `softsetfile.txt`

```
(softrew (Tgfb1Dish ([cdc6 = 0])) )


(red (Tgfb1Dish  ([cdc6 = 0])) )


(q)
```

```
PLSS> load softsetfile.txt

Result: PD(
{CLc | Akt1 Atf2 Erks Fak1 Jnks Mekk1 Mlk3 P38s Pml Smad2 Smad3 Tab1 Tab2 Tab3
    Tak1
    Traf6 Zfyve16[Abl1 - act]}
{CLi |([Pak2 - act] :[Rac1 - GTP])[Cdc42 - GTP][Hras - GTP]}
{CLm | empty}
{CLo | empty}
{NUc | Cdc6-gene Cdkn1a-gene Cdkn2b-gene Col1a1-gene Col3a1-gene Csrp2-gene
    Cst6-gene
    Ctdsp1 Ctgf-gene Dst-gene Ets1 Fn1-gene Gfi1-gene Mmp2-gene Mmp9-gene Mylk-
        gene
    Pai1-gene Pthlh-gene RoRc-gene Smad4 Smad6-gene Smad7-gene Tgfb1-gene Timp1
        -gene}
{Tgfb1RC | Smad7 Smurf1 Smurf2 Tgfb1 :[TgfbR1 - ubiq] :[TgfbR2 - act]}
{XOut | empty})

Result: PD(
{CLc | Abl1 Akt1 Atf2 Erks Fak1 Jnks Mekk1 Mlk3 P38s Pak2 Pml Smad2 Smad3 Smad4
    Smurf1 Smurf2 Tab1 Tab2 Tab3 Tak1 Traf6 Zfyve16}
{CLi |[Cdc42 - GDP][Hras - GDP][Rac1 - GDP]}
{CLm | empty}
{CLo | empty}
{NUc | Cdc6-gene Cdkn1a-gene Cdkn2b-gene Col1a1-gene Col3a1-gene Csrp2-gene
    Cst6-gene
    Ctdsp1 Ctgf-gene Dst-gene Ets1 Fn1-gene Gfi1-gene Mmp2-gene Mmp9-gene Mylk-
        gene
    Pai1-gene Pthlh-gene RoRc-gene Smad6-gene Smad7 Smad7-gene Tgfb1-gene Timp1
        -gene}
{Tgfb1RC | TgfbR1 TgfbR2}
{XOut | Tgfb1})[cdc6 = 0]

Thanks for using PLSS!
rewrites: 4234 in 580ms cpu (89892ms real) (7299 rewrites/second)
```

```
result Portal: <>
```

Ejemplos con uso de estrategias:

```
PLSS> loadstrat undefSemi

Loaded strategy function ( undefSemi )
PLSS> softrew (Tgfb1Dish ([cdc6 = 0]))

Result: PD(
{CLc | Akt1 Atf2 Erks Fak1 Jnks Mekk1 Mlk3 P38s Pml Smad2 Smad3 Tab1 Tab2 Tab3
    Tak1
    Traf6 Zfyve16[Abl1 - act]}
{CLi |([Pak2 - act] :[Rac1 - GTP])[Cdc42 - GTP][Hras - GTP]}
{CLm | empty}
{CLo | empty}
{NUc | Cdc6-gene Cdkn1a-gene Cdkn2b-gene Col1a1-gene Col3a1-gene Csrp2-gene
    Cst6-gene
    Ctdsp1 Ctgf-gene Dst-gene Ets1 Fn1-gene Gfi1-gene Mmp2-gene Mmp9-gene Mylk-
        gene
    Pai1-gene Pthlh-gene RoRc-gene Smad4 Smad6-gene Smad7-gene Tgfb1-gene Timp1
        -gene}
{Tgfb1RC | Smad7 Smurf1 Smurf2 Tgfb1 :[TgfbR1 - ubiq] :[TgfbR2 - act]}
{XOut | empty})
PLSS> loadstrat undefOne

Loaded strategy function ( undefOne )
PLSS> loadstrat undefWRow

Loaded strategy function ( undefWRow )
PLSS> loadstrat undefWCol

Loaded strategy function ( undefWCol )
```

```
PLSS> help

   ========================
   AYUDA DEL PROGRAMA: PLSS
   ========================

   1.
   2.
   3.
```

Ejemplos de funciones auxiliares:

```
PLSS> red (dish(Tgfb1Dish  ([cdc6 = 0])))

Result: PD(
{CLc | Abl1 Akt1 Atf2 Erks Fak1 Jnks Mekk1 Mlk3 P38s Pak2 Pml Smad2 Smad3 Smad4
    Smurf1 Smurf2 Tab1 Tab2 Tab3 Tak1 Traf6 Zfyve16}
{CLi |[Cdc42 - GDP][Hras - GDP][Rac1 - GDP]}
{CLm | empty}
{CLo | empty}
{NUc | Cdc6-gene Cdkn1a-gene Cdkn2b-gene Col1a1-gene Col3a1-gene Csrp2-gene
    Cst6-gene
    Ctdsp1 Ctgf-gene Dst-gene Ets1 Fn1-gene Gfi1-gene Mmp2-gene Mmp9-gene Mylk-
        gene
    Pai1-gene Pthlh-gene RoRc-gene Smad6-gene Smad7 Smad7-gene Tgfb1-gene Timp1
        -gene}
{Tgfb1RC | TgfbR1 TgfbR2}
{XOut | Tgfb1})
PLSS> red (atts(Tgfb1Dish  ([cdc6 = 0])))

Result: [cdc6 = 0]
```

```
PLSS> red cdc6

Result: cdc6

PLSS> red over(2, 2)

Result: (1) .NzNat
PLSS> red sumUndefWRow(3)

Result: 12
PLSS> red addUndefWRow((0, 1, 0, 0, 0))

Result: (1) .NzNat

PLSS> red undefWRow((1, 0, 1) (1, 0, 1) (1, *, 1))

Result: 2
PLSS> red undefWCol((0, 1, 0, 0) (0, 1, *, 1) (0, 1, 0, 1) (1, *, 0, 1) )

Result: 3
```