Sistema de búsquedas basado en N-soft sets para sistemas biológicos simbólicos modelados con Pathway Logic en Maude

N-soft set based search system for symbolic biological systems modeled with Pathway Logic in Maude

Trabajo de Fin de Máster
Curso 2019–2020

Autor
Rocío María Santos Buitrago

Director
Adrián Riesco Rodríguez

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Sistema de búsquedas basado en N-soft sets para sistemas biológicos simbólicos modelados con Pathway Logic en Maude
N-soft set based search 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 María Santos Buitrago

Director
Adrián Riesco Rodríguez

Convocatoria: *Febrero 2020*
Calificación:

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

12 de enero 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 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. Por supuesto, también 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.

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 búsquedas basado en N-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 el lenguaje de reescritura Maude. Se han desarrollado numerosos modelos de rutas de señalización celular con esta herramienta, por ejemplo, SKMEL133 (melanoma cells). SKMEL133 es una red de interacciones y modificaciones de proteínas que se utilizan por la célula para transmitir señales de su entorno al núcleo. En el sistema de Pathway Logic, en este trabajo se pretende implementar una variante de búsqueda basada en N-soft sets. El comando search estándar permite realizar búsquedas a través del árbol de reescrituras partiendo de un estado inicial. La implementación propuesta se realizará en Full Maude y proporcionará las ventajas sobre la toma de decisiones bajo información incompleta de los soft sets.

Palabras clave

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

Abstract

N-soft set based search 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 dishes and the Maude rewriting language. Numerous cell signaling pathway models have been developed with this tool, e.g. SKMEL133 (melanoma cells). SKMEL133 is a network of protein interactions and modifications used by the cell to transmit signals from its environment to the nucleus. In the Pathway Logic system, this work aims to implement a search variant based on N-soft sets. The standard search command allows searching through the rewriting tree starting from an initial state. The proposed implementation will be carried out in Full Maude and will provide the advantages over decision making under incomplete information of the soft sets.

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

Capítulo 1

Introducción

"Frase célebre dicha por alguien inteligente"
— Autor

El estudiante elaborará una memoria descriptiva del trabajo realizado, con una **extensión mínima recomendada de 50 páginas** incluyendo al menos una introducción, objetivos y plan de trabajo, resultados con una discusión crítica y razonada de los mismos, conclusiones y bibliografía empleada en la elaboración de la memoria.

Además del cuerpo principal describiendo el trabajo realizado, la memoria contendrá los siguientes elementos, que no computarán para el cálculo de la extensión mínima del trabajo:

- un resumen en inglés de media página, incluyendo el título en inglés,
- ese mismo resumen en castellano, incluyendo el título en castellano,
- una lista de no más de 10 palabras clave en inglés, y esa misma lista en castellano,
- un índice de contenidos, y una bibliografía.

Todo el material no original, ya sea texto o figuras, deberá ser convenientemente citado y referenciado. En el caso de material complementario se deben respetar las licencias y copyrights asociados al software y hardware que se emplee.

1.1. Motivación

Introducción al tema del TFM.

1.2. Objetivos

Descripción de los objetivos del trabajo.

1.3. Plan de trabajo

Aquí se describe el plan de trabajo a seguir para la consecución de los objetivos descritos en el apartado anterior.

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 2.3 introduce a Pathway Logic, el entorno de trabajo que se ha utilizado en la modelización de los sistemas biológicos. En cada una de las secciones se estudia y presenta el conocimiento actual sobre estas áreas acompañado por referencias de publicaciones científicas relevantes.

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.

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 muy 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 de Maude

El módulo `META-LEVEL` 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. Por ejemplo, la función descendente de `metaSearch` con un límite de, por ejemplo 3, es totalmente determinista, ya que dadas las metarepresentaciones $\bar{\mathcal{R}}$ del módulo del sistema deseado y $\bar{t}$ del término inicial más el límite 3, el resultado obtenido por `search` para $\mathcal{R}$, t y 3 a nivel de objeto, y por lo tanto por `metaSearch` en metanivel, se determina de manera única (Clavel et al., 2020).

Aunque `META-LEVEL` es muy potente, su naturaleza puramente funcional significa que no tiene noción de estado. Por lo tanto, las aplicaciones reflexivas en las que la interacción del usuario en un modo de cambio de estado es esencial requieren el uso de `META-LEVEL` en el contexto de las características adicionales que apoyan dicha interacción.

La característica de los metaintérpretes de Maude hace posible tipos muy flexibles de interacciones reflexivas 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.

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 comandos y mensajes de respuesta que cubren casi la totalidad del intérprete Maude. Por ejemplo, puede ser construido 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 ser usado 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 E/S 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, siendo 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.

2.2.1. 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.

Listado 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 (2016), 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 formación 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.

Definición 3 *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

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.

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}.

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.

	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.

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.

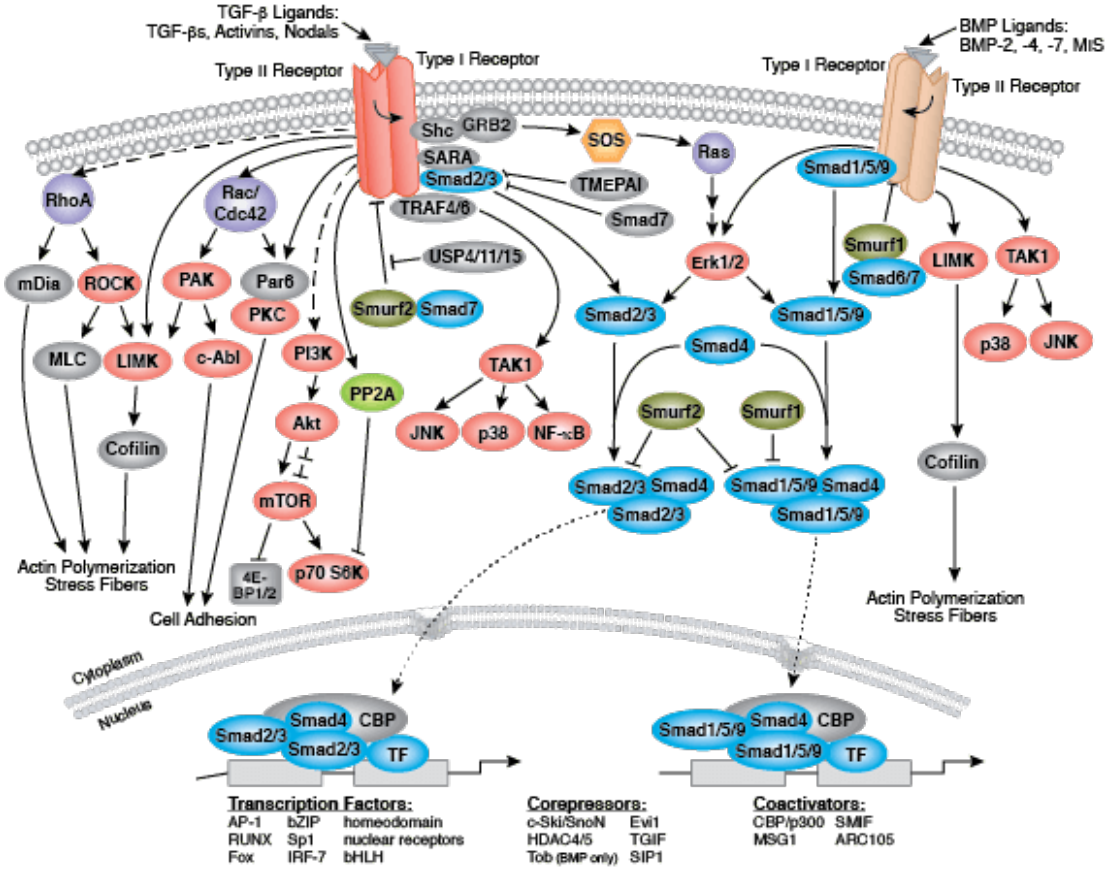

Figura 2.1: Ruta de señalización TGF-β. Fuente: Cell Signaling Technology.

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 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:

Listado 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 los 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:

Listado 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:

Listado 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`:

Listado 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]}) .
```

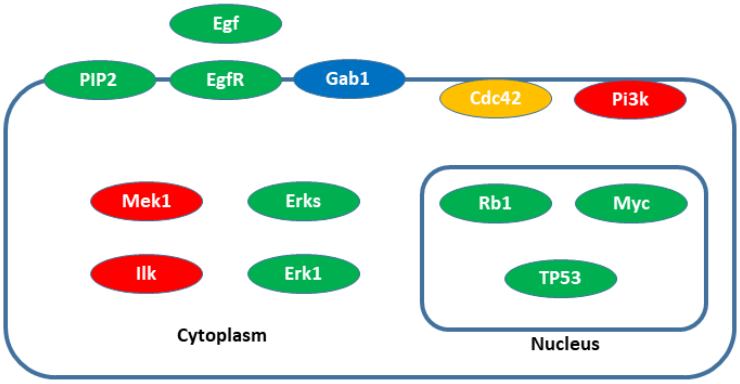

Figura 2.2: Representación esquemática de una célula. Las proteínas activadas están marcadas en rojo, las fosforiladas en azul y las unidas a GDP en amarillo. Las proteínas que no tienen modificaciones se muestran en verde.

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`.

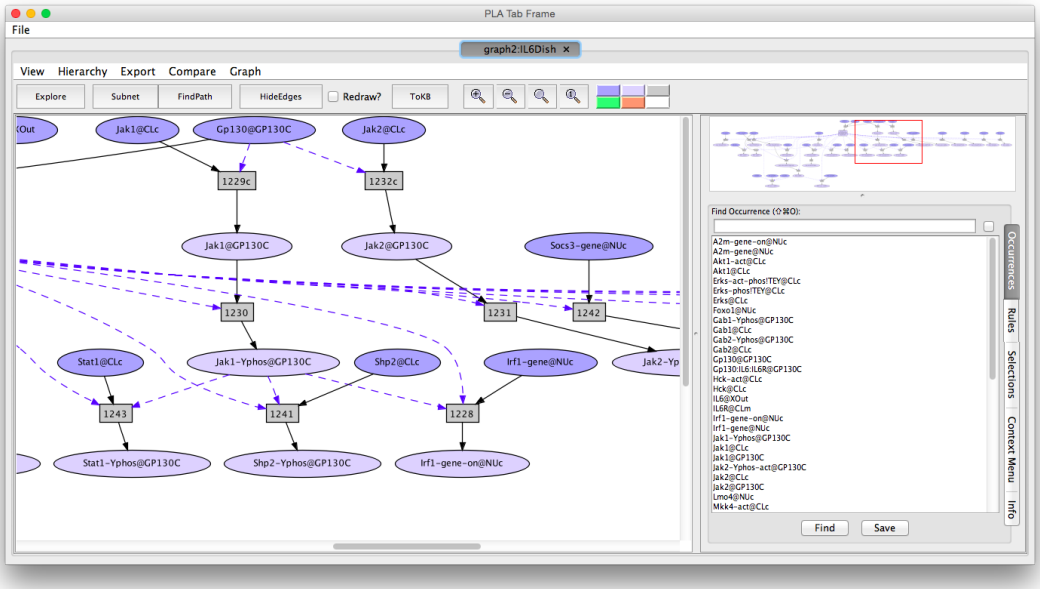

Figura 2.3: Vista general de una ruta de señalización con Pathway Logic Assistant.

Por último, detallamos la codificación completa de un ejemplo específico de la placa de

preparación `Tgfb1Dish`:

Listado 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`);
- 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 construyen 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.

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:

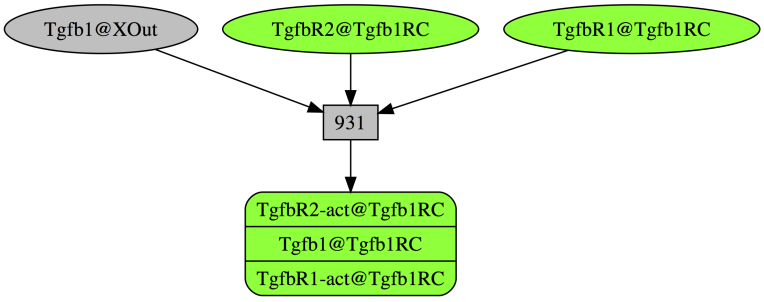

Figura 2.4: Representación esquemática de la regla de reescritura `931.TgfbR1.TgfbR2.by.Tgfb1` en Pathway Logic Assistant.

Listado 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**.

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 redes de Petri y el lenguaje de reescritura Maude. 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 toma el mapa que corresponde al factor de crecimiento transformante beta 1 (Tgfb1).

Por otra parte, en este proyecto se pretende utilizar un modelo que permita trabajar con datos imprecisos o inciertos. 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 trabajo 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 el sistema de Pathway Logic, en este trabajo se pretende implementar una variante de búsqueda basada en soft sets. El comando `search` estándar 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 sobre la toma de decisiones bajo información incompleta de los soft sets. En las secciones 2.1 y 2.2, se comentó brevemente los aspectos fundamentales del lenguaje de reescritura Maude y de 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 en el modelo STM7 de Pathway Logic

2. Implementación en Maude de los soft sets y desarrollo de los operadores y funciones asociados

3. Definición de los *softdish*, que son los platos o dishes 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 entradas/salidas y metaintérpretes
5. Implementación en el metaintérprete de los comandos de simplificación con ecuaciones y reglas soft
6. Implementación en el metaintérprete de otros comandos: carga de ficheros de instrucciones y salida del programa
7. Definición e implementación de especificaciones para los softdish
8. Implementación en el metaintérprete de los comandos de selección y reescritura a través de la especificación
9. Integración de todas las partes en el proyecto PLSS

En las secciones que vienen a continuación se desarrollan cada uno de los bloques enumerados. A continuación, en el resto del capítulo, se describen los módulos y ficheros desarrollados para este proyecto. Después se muestran algunos ejemplos de ejecución del programa. Por último, se incluyen algunos comentarios sobre el rendimiento de la aplicación y se realiza un análisis comparativo de los resultados obtenidos con los resultados del uso estándar de la reescritura.

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:

Listado 3.1: Módulo `QQ`

```
mod QQ is
 inc ALLRULES .
 inc TGFB1DISH .
 inc ALLOPS .
 inc UNDEF-TO-SEMI .
endm
```

Así pues, este módulo importa los 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) `UNDEF-TO-SEMI` que define la estrategia del soft set utilizado.

El módulo `ALLOPS` a su vez incluye los módulos en los que se definen las constantes para los químicos, genes y proteinas que pueden existir en la célula y también sus modificaciones (`MODIFICATIONOPS`). En otros módulos se definen las signaturas, sitios y tensiones. El listado 3.6 muestra el código de este módulo.

El módulo `ALLOPS` también importa el módulo donde se definen las localizaciones de la célula `LOCATIONOPS`, 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.

Listado 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, según muestra el listado 3.3 de forma reducida. Se comprueba la construcción 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, proteinas modificadas, compuestos, etc.

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 proteina) 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.

Listado 3.3: Módulo `THEOPS` y asociados

```
*****************************************************************
fmod THEOPS is inc DISH .

endfm
*****************************************************************
fmod DISH is inc LOCATION .

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

endfm
*****************************************************************
fmod LOCATION is inc MODIFICATION .

sort LocName .

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

sort CompName .
subsort CompName < LocName .
```

```
endfm
*******************************************************************
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
*******************************************************************
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
*******************************************************************
fmod THING is pr PROTEIN .

sorts SimpleThing Complex Thing .
sorts Gene Chemical .
subsort Protein Gene Chemical < SimpleThing .
subsort Complex SimpleThing < Thing .

op (_:_) : Thing Thing -> Complex [ctor assoc comm] .

endfm
*******************************************************************
fmod PROTEIN is pr NAT .

sort AminoAcid Protein BProtein .
subsort AminoAcid < Protein .
subsort BProtein < Protein .

ops A C T Y S K P N L M V I F D E R H Q W G : -> AminoAcid [ctor] .

endfm
```

A continuación, el módulo `TGFB1DISH` define el dish o estado inicial en el modelo Tgfb1 de STM7. En el listado 3.4 vemos que se define 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 contiene, entre otras, las proteinas `Abl1`, `Akt1` y `Atf2`; y que en el interior de la membrana celular está

presente la proteina `Hras` con la modificaicón `GDP` (`[Hras - GDP]`)

Listado 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. En el listado 3.5 se muestran dos reglas. El significado de estas reglas se explicó con detalle en la seccion 2.4.

Listado 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 `UNDEF-TO-SEMI` que define la estrategia del soft set utilizado se comentará en la siguiente sección 3.2.

3.2. Soft sets y funciones asociadas

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 a las reglas que se pueden ejecutar y las columnas a cada uno de los atributos que se consideran. Por tanto, la implementación consiste en definir matrices. En el módulo `MATRIX` se han definido una fila de la 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 (número) a cada regla y la regla *mejor* será la 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`:

```
*** 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) .
```

3.3. Definición de *softdish*

El objeto fundamental en nuestro proyecto es el *softdish*. Se define un *softdish* como 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. Para realizar la reescritura de un softdish, se necesita construir la matriz buscando todas las reglas posibles a aplicar al término inicial con el valor de sus atributos y, por último, se elige aplicar la regla cuyo valor de elección sea mayor (ver módulo `SS-STRAT`).

La implementación de la reescritura de un softdish se realiza con el operador `rewStrat` que se apoya en el operador auxiliar `next`, que es el que calcula el término reescrito:

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

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

Como se observa en el código, 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 haciendo uso del `metaReduce` y `metaSearch`:

```
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) .
```

3.4. Metaintérprete: entorno de ejecución para PLSS

Elaboración de un entorno de ejecución para PLSS utilizando entradas/salidas y metaintérpretes

Listado 3.6: Módulo `ALLOPS`

```
\begin{lstlisting}[language=Maude,caption={Modelo \texttt{QQ}}]
mod QQ is
```

Listado 3.7: Fichero `softsetfile.txt`

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


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


(q)
```

3.5. Comandos de simplificación con ecuaciones y reglas soft

Implementación en el metaintérprete de los comandos de simplificación con ecuaciones y reglas soft

3.6. Comando de carga de ficheros

Implementación en el metaintérprete de otros comandos: carga de ficheros de instrucciones y salida del programa

3.7. Especificaciones para *softdish*

Definición e implementación de especificaciones para los softdish

3.8. Comandos a través de especificaciones

Implementación en el metaintérprete de los comandos de selección y reescritura a través de la especificación

3.9. Integración de los módulos

Integración de todas las partes en el proyecto PLSS

3.10. Ficheros utilizados

incluir la dependencia entre los ficheros utilizados.

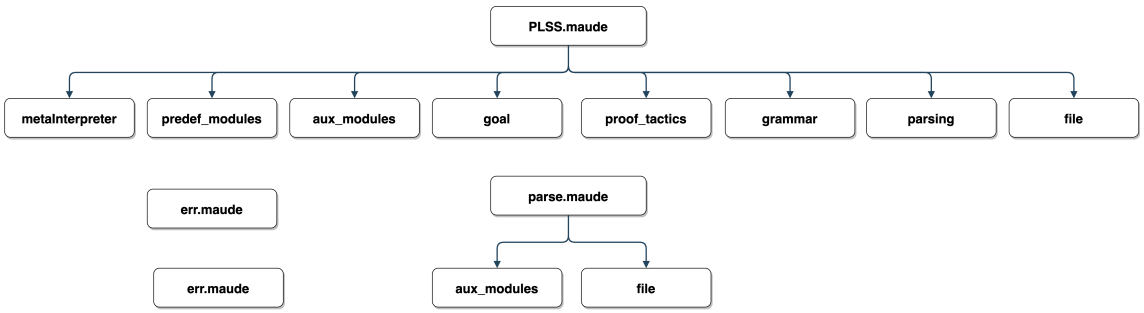

Figura 3.1: Dependencia de carga de ficheros (ficheros.png)

(2) He cambiado algo los nombres de ficheros. Los ficheros que intervienen en el proyecto son:

`plss.maude`: modulo principal del proyecto PLSS

`aux_modules.maude` y `predef_modules.maude`: módulos auxiliares de database, mensajes, ... y otras operaciones con módulos

`grammar.maude`: definición de la gramática (todavía quedan restos del mtp y cuantificación) Los comandos son definidos (en `COMMAND-SIGN`): load, red, softrew, softdishselect, specsoftrew, q y exit, En la signatura de la especificación, he definido los operadores dish y attributes (y kkk para probar).

`softdishspec.plss`: Definición de la especificación MySoftdish De esta forma, pretendemos que se seleccione el softdish con la instrucción:

```
softdishselect MySoftdish >> dish PD({NUc | empty}) . attributes ([cdc6 =
    0], [cdkn1a = 0]) .
```

`parsing-softdish.maude`: hace el parsing del softdish (todavía sin cambiar)

`proof_tactics.maude` (no se utiliza)

`qqallSS.maude` y `qqSS.maude`: ficheros de operadores y reglas de Pathway Logic (antiguo allSS.maude)

`softset.maude`: fichero de definición de softsets con la función de reescritura rewStrat.

`softdish.maude`: (antiguo goal.maude) define la clase `Softdish`, la constante `noSoftdish` y el operador `>>` para la especificación

`softsetfile.txt`, `softsetfile1.txt`, `softsetfile2.txt`: ejemplos de ficheros para importar con la instrucción load

`test_plss.maude`: ejemplos de comandos para utilizar dentro de PLSS

Las carpetas `ops`, `rules` y `SDishes`: igual que antes (salvo que he creado un par de rules sencillas para comprobar el comportamiento del `softrew`)

3.11. Módulos utilizados

incluir la dependencia entre los módulos utilizados.

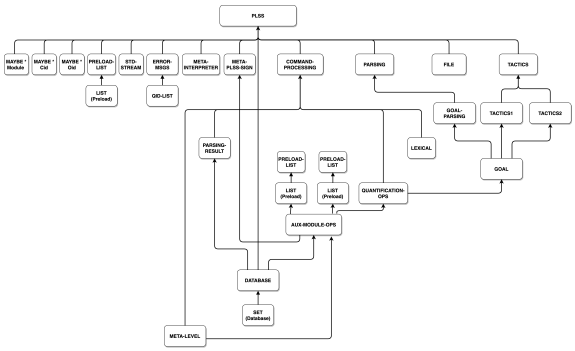

Figura 3.2: Dependencia de módulos 1/2

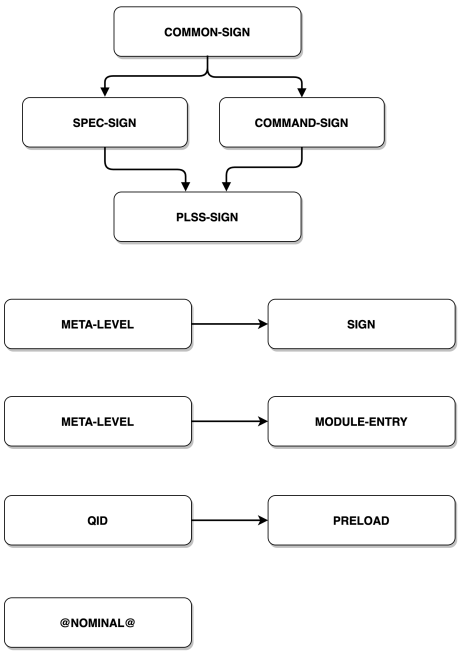

Figura 3.3: Dependencia de módulos 2/2

3.12. Ejemplo de ejecución

Una vez que se han descrito los bloques y el código del proyecto, se incluye ahora un ejemplo de ejecución (con todos los comandos implementados y también con la carga de un fichero de instrucciones)

1. Lanzamos el comando `run` que nos devuelve una configuración con un portal que el interprete puede comunicarse y podemos tratrarlo. Esta configuración comienza en un estado `init` que luego se irá modificando. La base de datos se encuentra vacía al comienzo.

2. Una vez lanzado, comenzará a entrar en las reglas con las que haga matching. En nuestro caso puede encontrarse con dos (`init-module` o `init-view`), ya que poseen el `msg` de `createdInterpreter`, es decir, que el intérprete fue creado. Además, pasará al estado `load-std-db`.

3. Cuando se inserta el primer módulo o vista, existen varias formas de continuar. Entre ellas se encuentra `insertaModulo` o `insertaVista` si se ha insertado un módulo y `insertaModulo` o `insertaVista` si se ha insertado una vista.

Continúa en el mismo estado hasta que termina con los módulos y vistas, entonces pasará al estado `load-grammar` que carga la gramática.

Cuando se complete, cambia de estado a `load-user-mi` y crea un nuevo interprete.

Si ha sido creado con éxito, pasa al estado `load-user-mods` que sube el módulo aplanado hasta que se acaba la lista y cambia el estado a `idle` mientras espera recibir una entrada del usuario a través del prompt `PLSS>`.

Si la recibe entonces pasa al estado `parseComm`. Este se encargará de parsear el texto introducido por el usuario utilizando la gramática elevada al metanivel.

Si el texto fue parseado con éxito, entonces vuelve al primer paso, en caso contario cambiaría de estado a `print&idle` y escribiría un mensaje de que el comando que se intenta introducir no existe.

Del estado en que se encuentra volvería a `idle` (tal como se muestra en el diagrama de estados) hasta que el usuario volviera a escribir algo en el prompt del programa.

Existen varios comandos que mi programa es capaz de gestioanr, entre ellos están: `load`, `red`, `softrew`, `softdishselect`, `specsoftrew`, `exit` y `q`. Con los dos últimos se saldría del programa.

3.13. Rendimiento

Por último, se comentan algunos aspectos sobre el rendimiento del programa. Se muestra también una comparativa entre resultados de reescritura guidados para los distintos choice values implementados y con el sistema de reescritura nativo de Maude.

3.14. Un entorno de ejecución para Mini-Maude utilizando IO y metaintérpretes

En esta sección se pueden encontrar un ejemplo adicional del uso de flujos estándar y metaintérpretes, donde se utilizan para desarrollar un entorno de ejecución simple. Se presenta un entorno para el lenguaje MiniMaude.

Estas mismas técnicas se pueden utilizar para desarrollar un lenguaje propio. Una vez definida una gramática (`MINI-MAUDE-SYNTAX` en el caso de MiniMaude) y una transformación de análisis de los términos en Maude (como proporciona la operación `processModule` en el módulo `MINI-MAUDE`) se puede construir un entorno de ejecución utilizando las técnicas que se muestran a continuación.

El lenguaje de MiniMaude se ha diseñado para ser lo más simple posible, pero queríamos que incluyera varias características:

- Podemos declarar clases y operaciones, además de especificar ecuaciones en los términos que podemos construir.
- Podemos incluir módulos previamente definidos. También podemos almacenar módulos en la base de datos del metaintérprete para que puedan ser utilizados después.
- Podemos reducir los términos a sus formas normales utilizando las ecuaciones de un módulo. La sintaxis definida para MiniMaude sólo permite el comando `reduce`.

La principal diferencia entre metanivel y metaintérprete es que el metanivel es funcional pero el metaintérprete no lo es y se puede interactuar con él a través de mensajes. Al no ser funcional, el metaintérprete proporciona funcionalidad, por ejemplo, para almacenar módulos y vistas en su base de datos, y luego operar con ellos.

Los metaintérpretes proporcionan la misma funcionalidad que el metanivel, además de algunas características adicionales para insertar elementos en su base de datos y recuperarlos. En otras palabras, podemos decir que proporciona la funcionalidad deseada tanto a nivel de objeto como a metanivel.

En nuestro caso, almacenaremos 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 MiniMaude. Para poder analizar las entradas usando el metaintérprete, empezaremos introduciendo el módulo `MINI-MAUDE-SYNTAX` en él. Una vez insertado, podemos 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, necesitaremos 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 necesitamos informar del error.

Una vez que se complete la entrada, podemos tener un módulo o un comando de reducción. Si es un módulo, primero necesitamos extraer la signatura y luego resolver las burbujas en sus ecuaciones. El proceso debe llevarse a cabo en dos pasos, ya que la signatura puede referirse a submódulos de la base de datos del metaintérprete, pero las ecuaciones con las burbujas procesadas deben insertarse en el módulo superior. Si la entrada corresponde a un comando de reducción, el término debe ser analizado y luego reducido por el metaintérprete.

Aunque el proceso es bastante sistemático, se deben tener en cuenta diferentes casos. Para simplificar el proceso, utilizamos un estado de atributo que lleva la cuenta de las

diferentes alternativas. En la figura 3.4 se muestra el diagrama de estado del entorno de ejecución. Este objeto también guarda el identificador del metaintérprete, el nombre del último módulo insertado y la entrada parcial introducida.

Los objetos de MiniMaude se representan con las siguientes declaraciones:

```
sort MiniMaude .
```

Se utilizan varios mensajes para los pasos intermedios:

```
op processInput : Oid Term -> Msg .
```

El entorno de MiniMaude puede iniciarse utilizando las constantes de minimaude, con ello se crea un intérprete y se envía un banner al flujo de salida.

```
op o : -> Oid .
```

Una vez escrito el mensaje y creado el metaintérprete, se inserta el módulo `MINI-MAUDE-SYNTAX` en él. El segundo argumento del mensaje creado `createdInterpreter` es el remitente `interpreterManager`.

```
rl < O : MiniMaude | mi: null, st: 0, Atts >
```

Una vez que se inserta el módulo, se envía un mensaje `getLine` al objeto `stdin`.

```
rl < O : MiniMaude | mi: MI, st: 1, Atts >
```

Cuando el usuario introduce algunas entradas, el objeto `stdin` responde con un mensaje `gotLine` con la cadena introducida. Se espera que el usuario escriba `quit` o `q` para abandonar el entorno. Si la entrada es una de estas, se envía un mensaje de despedida al objeto `stdout` y se mata al metaintérprete. De lo contrario, se intenta analizar la entrada. Hay que tener en cuenta que algunas entradas pueden estar almacenadas en el atributo `in`, por lo que el mensaje al metaintérprete para que analice la entrada incluye toda la lista `Qid`.

```
rl < O : MiniMaude | mi: MI, in: QIL, st: 2, Atts >
```

Si se enviara un mensaje de renuncia al metaintérprete, éste respondería con un mensaje de despedida. Esta es la regla final, que termina la ejecución.

```
rl < O : MiniMaude | mi: MI, st: 3, Atts >
```

Si el análisis tiene éxito, el metaintérprete responde con un mensaje `parsedTerm` que incluye un término de tipo `ResultPair`. Este término se envía en un mensaje de `processInput`.

```
rl < O : MiniMaude | mi: MI, in: QIL, st: 4, Atts >
```

Si el análisis falló, 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 < O : MiniMaude | mi: MI, in: QIL, st: 4, Atts >
```

La respuesta también puede informar sobre una ambigüedad. Aunque se podría dar información más precisa al usuario, hemos simplificado el mensaje de error.

```
rl < O : MiniMaude | mi: MI, in: QIL, st: 4, Atts >
```

El mensaje `processInput` puede corresponder a un módulo funcional o a un comando `reduce`. En el primer caso, si la entrada contiene una firma válida, tal módulo se inserta en el metaintérprete. La entrada se guarda en un mensaje `pendingBubbles` para su posterior procesamiento.

```
rl < O : MiniMaude | mi: MI, mn: QI?, st: 6, Atts >
```

Una vez que se inserta la firma, el objeto MiniMaude solicita el módulo aplanado. Si el módulo contiene importaciones de módulos previamente introducidos, se debe utilizar el módulo completo para el procesamiento de las burbujas del módulo superior.

```
rl < O : MiniMaude | mi: MI, mn: QI, st: 7, Atts >
```

El módulo aplanado recuperado se utiliza entonces para procesar las burbujas en las ecuaciones del módulo. Si el procesamiento falla, se muestra un mensaje de error al usuario. Si tiene éxito, se solicita el módulo superior para que las ecuaciones se añadan a él.

```
rl < O : MiniMaude | mi: MI, mn: QI, st: 8, Atts >
```

El módulo superior se inserta en el metaintérprete una vez que se le añaden las ecuaciones procesadas.

```
rl < O : MiniMaude | mi: MI, mn: QI, st: 9, Atts >
```

Cuando la inserción se completa con éxito, se informa al usuario.

```
rl < O : MiniMaude | mi: MI, st: 10, Atts >
```

Como puede verse en la figura 3.4, el estado `5` es aquel al que vuelve el objeto cada vez que concluye una operación, ya sea con éxito o sin éxito, tras notificarlo al usuario y solicitar más entradas.

```
rl < O : MiniMaude | mi: MI, st: 5, Atts >
```

La entrada también puede corresponder a un comando de reducción. La siguiente regla trata el caso en que se introduce un comando pero no hay ningún módulo previo insertado en el que se pueda evaluar el comando.

```
rl < O : MiniMaude | mn: null, st: 6, Atts >
```

Si hay un módulo anterior, primero hay que analizar el término a reducir.

```
rl < O : MiniMaude | mi: MI, mn: MN, st: 6, Atts >
```

El análisis sintáctico del término puede dar lugar a éxito o fracaso. En el primer caso, se envía un mensaje `ReduceTerm` al metaintérprete y, en el segundo caso, se da un mensaje de error.

```
rl < O : MiniMaude | mi: MI, mn: MN, st: 11, Atts >
```

Una vez completado el comando de simplificación, el metaintérprete devuelve un mensaje `ReduceTerm` con el resultado de la ejecución. Antes de mostrar el resultado al usuario, se debe solicitar al metaintérprete la impresión del término.

```
rl < O : MiniMaude | mi: MI, mn: MN, st: 12, Atts >
```

Podemos hacer funcionar el entorno con la configuración con:

```
Maude> erew minimaude .
```

En primer lugar, introducimos un ejemplo sencillo, el módulo `NAT3`.

```
> fmod NAT3
```

Podemos ejecutar un simple comando en ese módulo de la siguiente manera. Un módulo más interesante se refiere al anterior, ampliándolo con una operación más.

Si se insertan entradas erróneas, se proporciona un mensaje de error. Finalmente podemos salir del entorno con un comando `q`.

También se pueden recibir los comandos `exit` y `q` que existen para abandonar el programa.

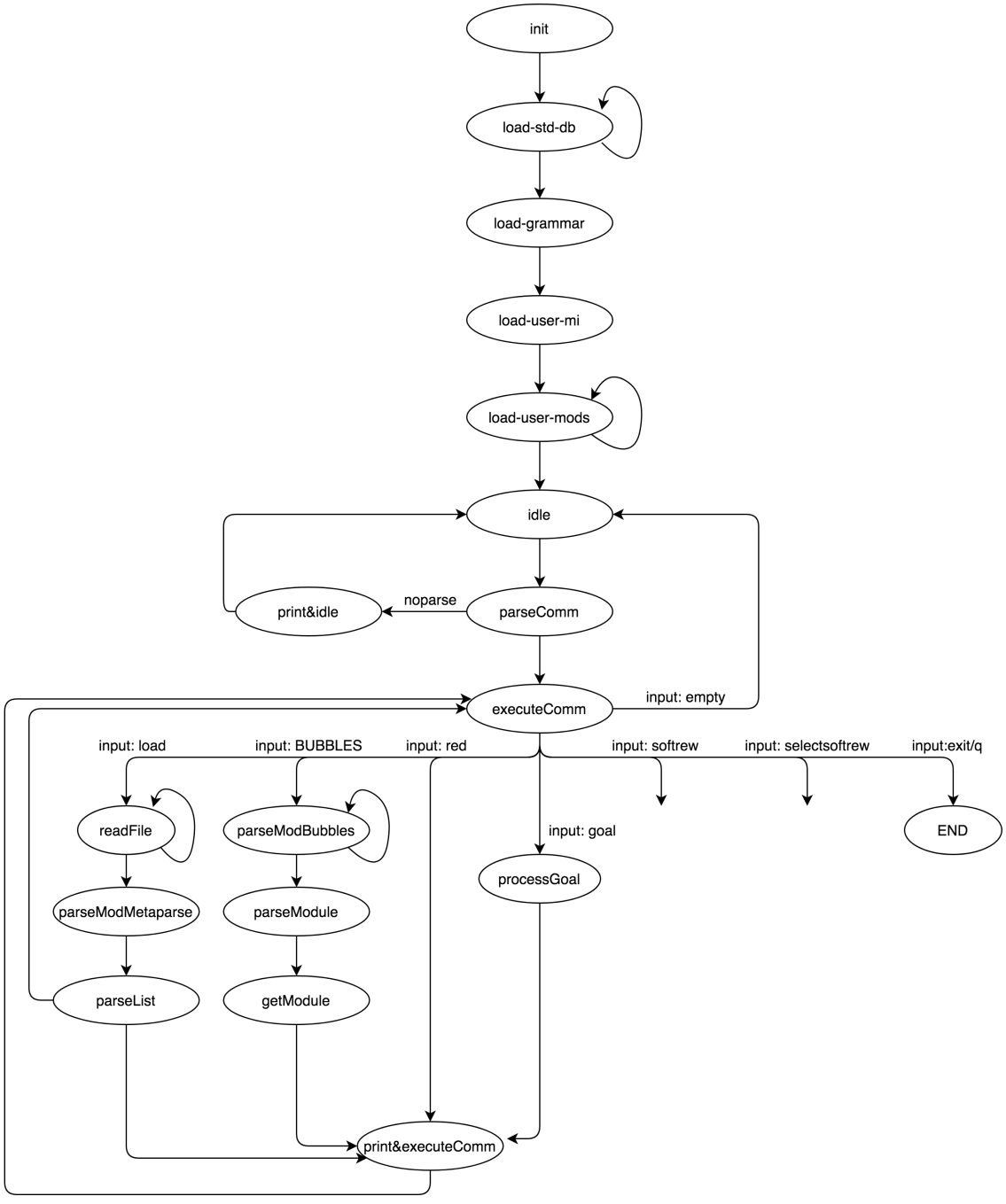

Figura 3.4: Diagrama de estados

Capítulo 4

Conclusiones y Trabajo Futuro

Conclusiones del trabajo y líneas de trabajo futuro.

Antes de la entrega de actas de cada convocatoria, en el plazo que se indica en el calendario de los trabajos de fin de máster, el estudiante entregará en el Campus Virtual la versión final de la memoria en PDF. En la portada de la misma deberán figurar, como se ha señalado anteriormente, la convocatoria y la calificación obtenida. Asimismo, el estudiante también entregará todo el material que tenga concedido en préstamo a lo largo del curso.

Chapter 5

Introduction

Introduction to the subject area. This chapter contains the translation of Chapter 1.

Chapter 6

Conclusions and Future Work

Conclusions and future lines of work. This chapter contains the translation of Chapter 4.

- LaTeX: Mittelbach et al. (2004); Lamport (1994); Fenn (2006) Koch (2015)
- Maude: Clavel et al. (2020, 2015, 2007, 1998)
  Meseguer (2012)
- IEEE Access: Santos-Buitrago et al. (2019)

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

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 3.0)*, 2020. `http://maude.cs.illinois.edu/w/images/e/ee/Maude-3.0-manual.pdf`.

Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N. y Talcott, C. L. Two decades of Maude. En *Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the occasion of his 65th birthday* (editado por N. Martí-Oliet, P. C. Ölveczky y C. L. Talcott), vol. 9200 de *Lecture Notes in Computer Science*, páginas 232–254. Springer, 2015.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N. y Meseguer, J. Metalevel computation in Maude. En *Proceedings of the Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1-4, 1998* (editado por C. Kirchner y H. Kirchner), vol. 15 de *Electronic Notes in Theoretical Computer Science*, páginas 331–352. Elsevier, 1998.

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.

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.

Feng, F. y Li, Y. Soft subsets and soft product operations. *Information Sciences*, vol. 232, páginas 44–57, 2013. ISSN 0020-0255.

Fenn, J. Managing citations and your bibliography with BibTeX. *The PracTEX Journal*, vol. 2006(4), 2006.

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.

Koch, R. TeXShop. `http://pages.uoregon.edu/koch/texshop/`, 2015. Accessed: 2020-09-06.

Lamport, L. *LaTeX: A Document Preparation System, 2nd Edition*. Addison-Wesley Professional, 1994.

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.

Meseguer, J. Twenty years of rewriting logic. *The Journal of Logic and Algebraic Programming*, vol. 81(7-8), páginas 721–781, 2012.

Mittelbach, F., Goossens, M., Braams, J., Carlisle, D. y Rowley, C. *The LaTeX Companion*. Addison-Wesley Professional, segunda edición, 2004.

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.

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.

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.

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.

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.

Wang, F., Li, X. y Chen, X. Hesitant fuzzy soft set and its applications in multicriteria decision making. *Journal of Applied Mathematics*, páginas Article ID 643785, 10 pages, 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. ISSN 0950-7051.

Apéndice A

Código del proyecto

Contenido del apéndice

Los archivos que forman parte de este proyecto están disponibles en el repositorio de Github `https://github.com/rsantosb/TFM-PLSS`.

Incluyo en este apéndice los ficheros fundamentales del proyecto.

Listado A.1: Fichero `softset.maude`

```
fmod VALUE is
 sort Value .

 ops 0 1 * : -> Value [ctor] .
endfm

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

fmod PREDEF-VALUE-FUNCTIONS is
 pr MATRIX .
 pr NAT .

 vars C BV S N : Nat .
 var  M : SoftSet .
 var  R : Row .

 *** 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) .

 *** Replaces * by 1 and adds up all values
 op undefOne : SoftSet -> Nat .
 eq undefOne(M) = compUndefOne(M, 0, 0, 0) .

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

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

 *** Replaces * by 1 and adds up all values
 op undefSemi : SoftSet -> Nat .
 eq undefSemi(M) = compSemi(M, 0, 0, 0) .

 *** Replaces * by 0.5 and adds up all values
 op compSemi : SoftSet Nat Nat Nat -> Nat .
 eq compSemi(mt, C, BV, S) = S .
 ceq compSemi(R M, C, BV, S) =
               if N >= BV
               then compSemi(M, s(C), N, C)
               else compSemi(M, s(C), BV, S)
               fi
  if N := (addUndefZero(R) + addUndefOne(R)) quo 2 .
endfm

--- load ops/theops.maude

fmod SOFTSET is
 pr META-LEVEL .
 pr VALUE .
 pr DISH .
 pr QID .

 sort AttId Att AttSet SoftDish .
 subsort Qid < AttId .
 subsort Att < AttSet .

 op [_=_] : AttId Value -> Att [ctor] .
 op mtSS : -> AttSet [ctor] .
 op _,_ : AttSet AttSet -> AttSet [ctor assoc comm id: mtSS] .

 op __ : Dish AttSet -> SoftDish [ctor] .


*** Seleccinar dish/atts de un SoftDish T: dish(T), atts(T)
var DSH : Dish .
var ATTS : AttSet .

op dish : SoftDish -> Dish .
op atts : SoftDish -> AttSet .

eq dish(DSH ATTS) =  DSH .
eq atts(DSH ATTS) =  ATTS .
```

```
endfm

fth SOFT-SET-FUN is
 pr MATRIX .
 pr NAT .

 op computeValue : SoftSet -> Nat .
endfth

fmod SS-STRAT{X :: SOFT-SET-FUN} is
 pr META-LEVEL .
 pr MATRIX .

 sort ApplyCell ApplyCellList .
 subsort ApplyCell < ApplyCellList .

 op [_,_] : Qid Nat -> ApplyCell [ctor] .

 vars TL TL' ATTS : TermList .
 var  SB : Substitution .
 var  V : Variable .
 var  MX : SoftSet .
 vars T T' : Term .
 var  M : Module .
 var  Ty : Type .
 var  N : Nat .

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

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

 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) .

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

 op rowFromTerm : Term TermList -> Row .
 *** Dish and Attributes
 eq rowFromTerm('__[T, T'], ATTS) = $rowFromTerm(T', ATTS) .
 eq rowFromTerm(T, ATTS) = mtRow [owise] .

 op $rowFromTerm : Term TermList -> Row .
 *** Attribute list
 eq $rowFromTerm('_`,_[TL], ATTS) = $rowFromTerm*(TL, ATTS) .
 eq $rowFromTerm(T, ATTS) = mtRow [owise] .

 op $rowFromTerm* : TermList TermList -> Row .
 eq $rowFromTerm*(empty, ATTS) = mtRow .
```

```
 eq $rowFromTerm*((''[_=_'][T,T'], TL), ATTS) =
                        if T in ATTS
                        then downTerm(T', 0)
                        else mtRow
                        fi, $rowFromTerm*(TL, ATTS) .
 eq $rowFromTerm*(TL, ATTS) = mtRow [owise] .

 op _in_ : Term TermList -> Bool .
 eq T in (TL, T, TL') = true .
 eq T in TL = false [owise] .

 op _[_] : TermList Nat ~> Term .
 eq (T, TL) [0] = T .
 eq (T, TL) [s(N)] = TL [N] .
endfm

view UndefToZero from SOFT-SET-FUN to PREDEF-VALUE-FUNCTIONS is
 op computeValue to undefZero .
endv

view UndefToOne from SOFT-SET-FUN to PREDEF-VALUE-FUNCTIONS is
 op computeValue to undefOne .
endv

view UndefSemi from SOFT-SET-FUN to PREDEF-VALUE-FUNCTIONS is
 op computeValue to undefSemi .
endv

fmod UNDEF-TO-ZERO is
 pr SS-STRAT{UndefToZero} .
endfm

fmod UNDEF-TO-ONE is
 pr SS-STRAT{UndefToOne} .
endfm

fmod UNDEF-TO-SEMI is
 pr SS-STRAT{UndefSemi} .
endfm
```

Listado A.2: Fichero `softsetfile.txt`

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


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


(q)
```

Listado A.3: Fichero `prueba.maude`

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

Algoritmo 1 - The algorithm of weighted choice values with N-soft sets.

Elements of the algorithm: aggregation methodology and weights w of attributes.

1: Input objects $O = \{o_1, o_2, \cdots, o_p\}$, and attributes $T = \{t_1, t_2, \cdots, t_q\}$.
2: Compute the EWCV σ_i^w of each $o_i \in O$, where $\sigma_i^w = \sum_{j=1}^{q} w_j \cdot r_{ij}$.

We can rank O by the EWCV of its members.

A.1. Explicaciones adicionales sobre el uso de esta plantilla

Si quieres cambiar el **estilo del título** de los capítulos, edita `TeXiS\TeXiS_pream.tex` y comenta la línea `\usepackage[Lenny]{fncychap}` para dejar el estilo básico de LaTeX.

Si no te gusta que no haya **espacios entre párrafos** y quieres dejar un pequeño espacio en blanco, no metas saltos de línea (\\) al final de los párrafos. En su lugar, busca el comando `\setlength{\parskip}{0.2ex}` en `TeXiS\TeXiS_pream.tex` y aumenta el valor de 0,2ex a, por ejemplo, 1ex.

TFMTeXiS se ha elaborado a partir de la plantilla de TeXiS[1], creada por Marco Antonio y Pedro Pablo Gómez Martín para escribir su tesis doctoral. Para explicaciones más extensas y detalladas sobre cómo usar esta plantilla, recomendamos la lectura del documento `TeXiS-Manual-1.0.pdf` que acompaña a esta plantilla.

[1] `http://gaia.fdi.ucm.es/research/texis/`