UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escuela Técnica Superior de Ingeniería Informática

Universitat Politècnica de València

Implementación del algoritmo de Tableau para la lógica temporal

Proyecto Fin de Carrera

Ingeniería Informática

Autor: Ghada El Khamlichi

Director: Alicia Villanueva

Valencia 2012

Resumen

El objetivo de este proyecto es crear un método que comprueba la satisfacibilidad automática de las propiedades de la lógica temporal lineal proposicional, mediante un algoritmo Tableau. Esto se implementará en el lenguaje Maude. El algoritmo se realizará en una pasada, y utilizará reglas Tableau que garantizan un tratamiento correcto de las eventualidades así como la completitud y la terminación del algoritmo de satisfacibilidad.

Índice de contenidos

Índice de figuras

Índice de tablas

Introducción

La lógica temporal es un tipo de lógica modal que permite estudiar el valor de verdad de las fórmulas en función del tiempo. Hoy en día se considera una herramienta imprescindible para muchos campos, en particular para el análisis de los sistemas donde el tiempo, entre otros factores, determina su comportamiento: protocolos de seguridad, sistemas en tiempo real, etc. Este tipo de lógica presenta aplicaciones en muchos otros campos, que abarcan desde el mundo industrial (control de procesos, robótica), hasta el mundo de la medicina (diagnóstico de enfermedades).

Existen varios métodos de decisión para la satisfacibilidad de las fórmulas lógicas. Uno de ellos es el método Tableau, que consiste en automatizar el proceso de validación de un conjunto de fórmulas. Como resultado de la aplicación de las reglas Tableau se obtiene un árbol cuyos nodos finales determinan el valor de verdad del conjunto de fórmulas de entrada.

A diferencia de la lógica clásica, la concepción de un algoritmo de Tableau para la lógica modal no resulta trivial: el tratamiento de las eventualidades se hace mediante ecuaciones que podrían no terminar.

Los algoritmos Tableau para la lógica temporal clásicos funcionan en dos pasadas: una primera donde se genera un grafo auxiliar y una posible segunda pasada para encontrar componentes fuertemente conectados en dicho grafo y verificar que sus premisas se cumplen.

El objetivo del presente proyecto es la implementación del algoritmo definido en (1) que evita tener que realizar dos pasadas, ya que incluye todas las verificaciones en el propio algoritmo. Además, se utilizan reglas que garantizan una gestión correcta de las eventualidades, y por lo tanto la terminación del mismo. Para la implementación se ha usado el lenguaje Maude, un lenguaje especialmente adecuado para este tipo de tareas ya que gracias a su simplicidad, intuitividad y buen rendimiento, permite definir fácilmente las pasos del algoritmo de Tableau obteniendo resultados satisfactorios.

La entrada del algoritmo será una fórmula de la lógica temporal proposicional lineal. Se irán aplicando las reglas del método de Tableau sobre ésta hasta que se cumplan las condiciones de finalización del algoritmo. Si se obtiene al menos un nodo abierto (de valor true), la fórmula será cierta, de lo contrario será falsa. Cada nodo final abierto representa un modelo para dicha fórmula.

La estructura de la memoria se divide en 4 partes: La sección de teoría constituirá una introducción a las bases requeridas para entender este trabajo; la sección de implementación y análisis contendrá dos versiones de la solución y un análisis del código de éstas, donde se destacarán las principales dificultades abordadas; en el tercer apartado se presentarán las pruebas ejecutadas para verificar la corrección del programa, así como una comparación entre las diferentes versiones. Seguidamente se expondrán las conclusiones sobre los resultados obtenidos, junto con las dificultades presentadas a lo largo del proyecto, y las posibles ampliaciones del trabajo realizado. Finalmente, se incluye un anexo con un pequeño tutorial sobre la sintaxis del lenguaje Maude, el código de la solución y las pruebas unitarias realizadas para comprobar la corrección de dicha solución.

1. Teoría

En el siguiente apartado se exponen los conceptos teóricos necesarios para entender el algoritmo por desarrollar. Se explicarán las bases de la lógica temporal, el método de Tableau y sus reglas, y finalmente se formalizará el algoritmo elegido para la solución.

1.1. Lógica Temporal

La lógica temporal (LT) es una clase particular de la lógica modal. Se utiliza para analizar los sistemas que no tienen un final determinado o cuyo comportamiento varía con el tiempo (p. ej. sistemas concurrentes, protocolos de seguridad, hardware…). Para este tipo de sistemas la lógica proposicional resulta insuficiente ya que se necesita poder expresar el sistema en términos del tiempo. La LT permite expresar hechos como: "el cielo estará despejado hasta el momento en que llueva".

Dentro de la Lógica Temporal existen varias clasificaciones: lógicas proposicionales frente a las de primer orden, ramificadas frente a lineales, discretas frente a continuas, etc. Este proyecto se centra en la Lógica Temporal Proposicional Lineal, conocida como PLTL. La naturaleza del tiempo para este caso es discreta y lineal, es decir que se trata de una colección de secuencias de transición infinitas y que el estado del sistema se puede expresar en cada instante de tiempo.

Las fórmulas PLTL se expresan mediante los siguientes elementos:

- las proposiciones atómicas (p, q…)
- las proposiciones constantes `false` y `true`
- el operador Not (¬)
- los operadores And (∧) y Or (∨)
- los operadores temporales:
  - Until (U): p U q se lee como p es cierta hasta el instante en que q sea cierta
  - Next (X): X p indica que la proposición p es cierta en el instante de tiempo siguiente
  - El operador de necesidad Globally (G): G p expresa que la proposición p será cierta en todos los instantes futuros
  - El operador de posibilidad Eventually (F): F p expresa que la proposición p será cierta en algún instante futuro.
  - Release (R): p R q se lee como q es cierto hasta el momento en que p lo sea (incluido este momento). Si p nunca es cierta, q seguirá siéndolo.

Las fórmulas de tipo p y ¬ p, donde p es una proposición, se denominan literales, y las fórmulas de tipo p U q, F p y ¬ G p se llaman eventualidades.

Para definir la semántica formal de la lógica temporal se usa el modelo de Kriple. Una estructura PLTL se representa mediante una tupla [S, R, E], donde S es un conjunto finito y no vacío de estados, R representa la relación de la transición entre dichos estados y E es una función de etiquetado que define para cada estado qué proposiciones son ciertas.

La estructura de Kriple *M* de una fórmula *f* permite definir si ésta es cierta en un estado s_j, donde j >= 0.

Dada una secuencia de estados S = s_0, … s_j … con j >= 0, y $\forall i$ con i >= 0, se cumple $(s_i, s_{i+1}) \in R$, *[M, s_j] |= f* se interpreta como *f* es cierta en el estado s_j, *y [M, s_j] |/= f* se interpreta como *f* es falsa en el estado s_j.

Para una proposición *p*, y dos fórmulas lógicas *f1* y *f2*, la satisfacibilidad de una fórmula PLTL en un estado s_j se define de la siguiente forma:

[M,s_j] |= p si y solo si *p ϵ E(s).*

[M,s_j] |/= false.

[M,s_j] |= ¬ f1 si y solo si *[M,s_j] |/= f1.*

[M,s_j] |= f1 ∧ f2 si y solo si *[M, s_j] |=f1* y *[M, s_j] |= f2.*

[M,s_j] |= X(f1) si y solo si *[M,s_{j+1}] |= f1.*

[M,s_j] |= f1 U f2 si y solo si existe un estado k >= j >= 0 donde *[M,s_k] |= f2* y que para todo estado i donde j =< *i < k [M,s_i] |= f1.*

1.2. Método Tableau para PLTL

Los métodos Tableau se introdujeron para comprobar la validez de las fórmulas lógicas. Se trata de deducir automáticamente el valor de verdad de una fórmula aplicando reglas lógicas intuitivas. Este método resulta muy útil en entornos didácticos, ya que facilita el entendimiento de la deducción lógica.

El método Tableau recibe como entrada la fórmula que se quiere demostrar. Se va desarrollando un árbol de nodos etiquetados con fórmulas y donde la raíz contiene la fórmula de entrada. Se van aplicando sucesivamente las reglas definidas por el algoritmo hasta que todos los nodos sean finales. Un nodo final puede ser abierto o cerrado: un nodo es abierto si contiene únicamente literales, únicamente el booleano `true`, o un bucle donde las premisas de las eventualidades se cumplen. Un nodo es cerrado cuando contiene la constante false, una contradicción o bien un bucle donde las premisas de las eventualidades no se cumplen. Si se obtiene algún nodo abierto significa que la fórmula es válida. De lo contrario, si todos los nodos finales son cerrados la fórmula es falsa.

El árbol resultante puede verse en forma disyuntiva: los nodos son conjunciones de las sub-fórmulas por satisfacer, y el árbol es una disyunción de estas conjunciones. Las ramas se pueden ver como una serie de implicaciones que van en sentido ascendente (desde las hojas hasta la raíz del árbol).

Las implementaciones para el método de Tableau generalmente son en dos pasadas. La primera crea un grafo de reescritura de las fórmulas, y la segunda revisa el primer grafo para verificar si las premisas de las eventualidades se han cumplido o no.

En este caso se presenta un algoritmo que evita hacer las dos pasadas, a cambio de modificar las reglas de Tableau y de añadir información al nodo. Por una parte se aplica una regla `Until` que recuerda el contexto actual y fuerza a que cambie para asegurar que el algoritmo finaliza. Por otra, se guarda el historial de cada nodo, para así detectar la presencia de bucles.

1.2.1. Reglas

Las reglas de Tableau se clasifican en conjuntivas (α), disyuntivas (β) y de tipo *Next*. Las reglas conjuntivas derivan en un nodo mientras que las disyuntivas derivan en dos. La regla de tipo `Next` realiza un paso al instante siguiente de tiempo: se borra el presente y el contenido del operador `Next` se vuelve cierto.

En las siguientes tablas se presentan las reglas conjuntivas y disyuntivas de Tableau: para las reglas conjuntivas, α representa la fórmula actual, y α1 el resultado de la aplicación de la regla. Para el caso de las reglas disyuntivas, β es la fórmula actual, y la disyunción de β1 y β2 forma el resultado de la aplicación de dichas reglas.

La Tabla 1 muestra la aplicación de las reglas de la doble negación y del operador And.

Tabla 1: Reglas Tableau conjuntivas

Regla	α	α1
Doble negación	¬¬p	p
And	p ∧ q	p, q

La tabla 2 muestra la aplicación de las reglas de la negación del operador And y del operador Until, en particular las reglas original y modificada de este último.

Tabla 2: Reglas Tableau disyuntivas

Regla	β	β1	β2
Negación And	¬(p ∧ q)	¬p	¬q
Negación Until	¬(p U q)	¬q	¬p, q, ¬X(p U q)
Until	p U q	q	p, ¬q, X(p U q)
Until(s)	s, p U q	s, q	s, p, ¬q, X(¬s, p U q)

Regla Until

Si la fórmula `p U q` es cierta, en el siguiente estado se obtiene `q ∨ (p, ¬q, X(p U q))`. Esto significa que si la premisa q no se cumple (¬q), p seguirá siendo cierta, y `p U q` será cierta en el instante siguiente (`X(p U q)`). Sin embargo, existe un problema con esta regla: al contenerse a sí misma, su evaluación puede producir un bucle infinito. Para resolverlo se introduce la regla `Until(s)`.

Regla Until(s)

La regla `Until` modificada utiliza el contexto `s` como información adicional para determinar el valor de verdad de la fórmula. `S` se compone de las fórmulas que en conjunción con la fórmula `Until` forman el nodo actual.

Esta regla recibe como entrada la fórmula `Until` y su contexto. En el estado siguiente hay dos posibilidades: que q sea cierta, o de lo contrario que lo sea `¬q, p y X(¬s, p U q)`. En ambos casos guardamos el contexto anterior.

La idea intuitiva se basa en que dada una fórmula `p U q`, debe existir un contexto para el que q sea cierta. Asimismo, si no se cumple q, forzaremos a que el contexto cambie negándolo (`X(¬s, p U q)`), de manera que se asegura que el cumplimiento de la premisa q no se va a posponer para siempre.

Regla Next

Si en un instante dado la fórmula `X(p)` es cierta, en el instante siguiente p lo será.

En la tabla 3, X representa el instante de tiempo actual, mientras que X1 es el instante de tiempo futuro.

Tabla 3: Regla Next

X	X1
X(p)	p

Para definir las reglas adicionales (los operadores ∨, G, F) existen dos opciones: la primera es representarlos en función de los operadores básicos; la segunda es definir reglas Tableau para reducirlos directamente.

En la siguiente tabla se muestra la representación de los operadores adicionales en función de los básicos:

Tabla 4: Representación de los operadores adicionales en función de los básicos

Operador	Expresión	Conversión
∨	p ∨ q	¬(¬p ∧ ¬q)
G	G p	¬(true U ¬p)
F	F p	true U p

En las dos tablas siguientes se exponen las ecuaciones Tableau conjuntivas y disyuntivas para los operadores adicionales:

Tabla 5: Reglas conjuntivas para los operadores adicionales

Regla	α	α1
Negación Or	¬ (p ∨ q)	¬p, ¬q
G	G p	p, X(G p)
¬F	¬ F p	¬p, ¬ X(F p)

Tabla 6: Reglas disyuntivas para los operadores adicionales

Regla	β	β1	β2
Or	p ∨ q	p	q
F	F p	p	¬p, X(F p)
F(s)	s, F p	s, p	s, ¬p, X(¬s U p)
¬G	¬(G p)	¬p	p, ¬X(G p)
¬G(s)	s, ¬(G p)	s, ¬p	s, p, ¬X(¬s U p)

1.2.2. Formalización del algoritmo

En esta sección se formalizará el algoritmo de Tableau. Para un conjunto de fórmulas *f0*, se construye un Tableau *T* para verificar la satisfacibilidad de *f0*. Los nodos del tableau estarán etiquetados por las fórmulas *F(n)* donde n es el nodo actual y *F* la fórmula que este contiene, el nodo inicial contendrá el conjunto de fórmulas *f0*.

La expansión de *T* se realiza mediante la aplicación de las reglas expuestas en la sección anterior. Si la regla utilizada es conjuntiva (α), se crea un nodo en la rama actual, en cambio, si es disyuntiva (β), se crean dos. Finalmente, si no quedan reglas α o β por aplicar, se aplica la regla `Next` y se pasa al instante siguiente. La aplicación del operador `Next` implica la creación de una nueva fase de tiempo en el árbol.

Un nodo final puede ser cerrado o abierto. Un nodo cerrado es aquél que contiene la constante `false`, una fórmula y su negación (una contradicción) o un bucle cuyas premisas no se han cumplido. Un nodo abierto es aquél que contiene únicamente la constante `true`, únicamente literales o un bucle cuyas premisas se han cumplido.

El algoritmo distingue las eventualidades para saber si aplicar la regla `Until` o `Until(s)`: si la eventualidad es distinguida se le aplica `Until(s)`. Como esta regla es costosa e implica negar el contexto, se fuerza a que solo haya una eventualidad distinguida por nodo.

La formalización del algoritmo presentada a continuación fue extraída del documento (1):

Entrada Conjunto de fórmulas PLTL *f0.*
Salida Tableau para la fórmula de entrada.
Algoritmo

Para un conjunto de fórmulas *F* contenido en un nodo *n*, se aplican los siguientes pasos en el orden en el que aparecen:

i. Si *F* contiene `false`, o bien dos proposiciones complementarias de la forma `{P,^P}`, se marca el nodo como cerrado
ii. Si *F* solamente contiene literales, el nodo final se considera abierto.
iii. Si *F(n)* es igual a *F(n')*, donde *n'* es un nodo antecesor de *n*, se trata de un bucle. Se averigua si las premisas de las eventualidades se han cumplido. Si es el caso, el nodo se marca como abierto, en otro caso se marca como cerrado

iv. Si el conjunto de fórmulas no cumple con ninguno de los puntos anteriores, se elige una fórmula *f* de *F(n)* que no sea una fórmula de tipo *Next*:

- Si se trata de una fórmula conjuntiva *α* (donde la regla de aplicación del operador es conjuntiva), se crea un nodo hijo *n'* cuya etiqueta es:
  - *F(n') = (F(n) - α) U α1* , siendo *α = f y α1* el resultado de la aplicación de la regla Tableau a *α*.

- Si es una fórmula disyuntiva β (donde la regla de aplicación del operador es disyuntiva), se crean dos nodos hijos *n'* y *n'',* cuyas etiquetas son :
  - *F(n') = (F(n) - β) U β1*
  - *F(n'') = (F(n) - β) U β2*
    Siendo *β = f* y la disyunción entre *β1* y *β2* el resultado de la aplicación de la regla Tableau a *β.*
  - Si *f* es una eventualidad:
    - Si f está distinguida, se aplica la regla Until(s), y se distingue la fórmula *Next* dentro de *β*.
    - Si f no está distinguida, pero hay otra fórmula distinguida, se aplica la regla Until. Se mantiene la fórmula distinguida en los nodos hijo *β1* y *β2*.
    - Si no hay ninguna fórmula distinguida, se distingue *f*, y seguido se aplica la regla Until(s) y se distingue la fórmula *Next* dentro de *β2*.

v. Si ninguno de los anteriores casos se cumple, y si *F(n)* contiene fórmulas *Next*, se aplica el operador X de la siguiente manera:

- Dado *F(n)* tal que:
  - *F (n) = {X (P1), X (P2),…., X (Pi), s }*
- El nuevo nodo será:
  - *F'(n) = {P1, P2,…., Pi }*

Cualquier conjunto de fórmulas *s* fuera del operador *Next* no forma parte del nuevo nodo.

La ejecución terminará cuando todas las hojas estén etiquetadas como abiertas o cerradas. Si todas las hojas son cerradas significa que *f0* es falsa. De lo contrario, si alguna hoja es abierta, cada una representará un modelo para la fórmula *f0.*

Véase un ejemplo del Tableau resultante del algoritmo anterior presentado en forma de árbol. El conjunto de fórmulas de entrada es $(p \ U \ q) \ \wedge \ \neg q$. Los nodos que contienen una fórmula por línea representan la conjunción de esas fórmulas. En la figura 1 se indica la regla aplicada para cada paso. Las eventualidades distinguidas se representan en negrita.

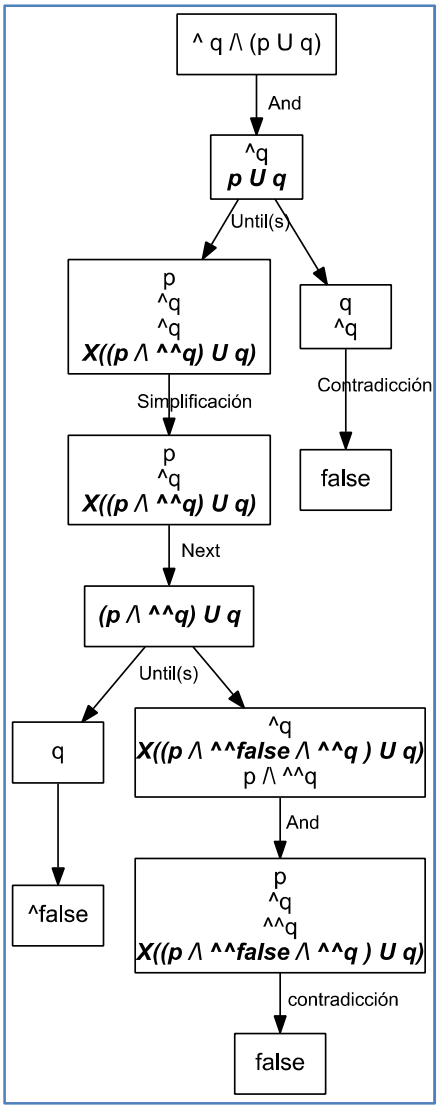

Figura 1: Árbol para la fórmula (p U q) ∧ ¬q

El proceso de validación es el siguiente: primero se aplica la regla And. A continuación se distingue la eventualidad `p Until q` y seguidamente se aplica la regla `Until(s)`. Desde este punto se expanden dos ramas: la rama derecha contiene una contradicción y acaba en un nodo cerrado, mientras que en la rama izquierda se distingue la eventualidad dentro del operador X y se aplica la fórmula `Next`, ya que no hay ningún otro operador por aplicar. Se distingue la eventualidad `(p /\ ^^ q) U q` y seguidamente se aplica la regla `Until(s)`. De la aplicación de ésta surgen dos nodos: uno contiene un literal q, lo que da lugar a un nodo abierto; en el segundo se niega el contexto, que al ser vacío en este caso es equivalente a `true`, y por lo tanto su negación es `false`. En esta rama se encuentra una contradicción y por lo tanto resulta en un nodo cerrado.

La obtención de un nodo abierto significa que existe un modelo para la fórmula y que ésta es cierta.

El modelo de la fórmula sería:

$E(S_1)$ = {p = true, q = false} , $E(S_2)$ = {q = true}

1.3. Maude

El algoritmo de Tableau se ha implementado en Maude. Éste es un lenguaje de programación declarativo funcional. Fue introducido y desarrollado en la universidad de Illinois en Estados Unidos. Es un lenguaje que sirve para implementar modelos de sistemas y realizar verificaciones formales sobre éstos.

Maude se ha utilizado para especificar y comprobar la fiabilidad de varios sistemas conocidos, como por ejemplo el metamodelo de UML, las plataformas de CORBA y SOAP, protocolos de comunicación como el FireWire y el lenguaje Java.

Se trata de un lenguaje potente en cuanto a expresividad, pues con él se puede modelar casi cualquier sistema. La otra ventaja es que su sintaxis resulta muy intuitiva, y se puede entender fácilmente incluso por usuarios principiantes. Esto facilita bastante la tarea de la codificación, y deja más tiempo para analizar el problema.

Otro aspecto importante de Maude es que soporta la lógica de reescritura, y gracias a sus reglas permite modelar sistemas concurrentes. Esto resulta muy conveniente para el algoritmo visto ya que permite definir de forma natural las reglas de deducción del Tableau.

A continuación se presentarán los elementos más importantes de la sintaxis de Maude y en el anexo del documento se puede encontrar un tutorial completo del lenguaje y los comandos más útiles para manejarlo en nuestro contexto.

1.3.1. Módulos

Un programa Maude está compuesto por módulos. En Maude existen tres tipos de módulos: funcional, de sistema y orientado a objetos. De éstos solamente se verán los dos primeros.

El módulo funcional es donde se declaran los tipos de los datos del sistema (sorts), los operadores (ops) que son las operaciones aplicables sobre éstos y las ecuaciones (eqs) que sirven para reducir o simplificar los términos formados por los dos anteriores. Una expresión de entrada será reducida por las ecuaciones del módulo funcional hasta que quede en forma canónica, es decir, cuando ya no se pueda reducir más.

Se declara de la siguiente manera:

```
fmod NombreMóduloFuncional
{Declaraciones del módulo}
endfmod
```

En el módulo de sistema se definen las reglas del sistema, aunque en él también se pueden definir tipos, operadores y ecuaciones. Las reglas definen las transiciones que puede haber entre diferentes estados del sistema y su ejecución puede ser concurrente. Éstas se aplican cuando los términos ya no son reducibles por las ecuaciones definidas.

Se declara con la siguiente sintaxis:

```
fmod NombreMóduloSistema
{Declaraciones del módulo}
endfmod
```

1.3.2. *Tipos (Sorts)*

Lo primero que se necesita definir en un sistema son los tipos. La definición de tipos se lleva a cabo de la siguiente manera:

```
sort  sortId  .
```

Si se quiere definir varios tipos de golpe, se puede utilizar la palabra `sorts`:

```
sorts  sortId1,  sortId2,  … ,  sortIdX  .
```

Nótese la importancia de que todas las instrucciones de un programa escrito en Maude acaben en un punto precedido por un espacio, pues de lo contrario la compilación fallaría.

Para establecer una jerarquía entre los tipos se definen los subtipos. Hay que prestar atención a no declarar ciclos en la jerarquía.

```
subsort  sortId1 <  sortId2 .
subsort  sortId2 <  sortId3 .
```

O bien:

```
subsorts  sortId1 <  sortId2 <  sortId3 .
```

1.3.3. *Operadores*

La sintaxis para declarar operadores es la siguiente:

```
op opId :  sortId1  sortId2  …  sortIdX  ->  sortOperator
[operator attributes] .
```

Si varios operadores comparten los mismos tipos, se declaran de la siguiente forma:

```
ops  opId1  opId2:  sortId1  sortId2  …  sortIdX  ->
sortOperator .
```

1.3.4. *Variables*

Se pueden declarar instancias de los tipos definidos para un sistema, para así poder definir sus ecuaciones y reglas.

Las variables se declaran de la siguiente forma:

```
var N : tipoN .
```

Si son varias variables del mismo tipo:

```
vars N M : tipoN .
```

1.3.5. Ecuaciones no condicionales

Las ecuaciones se declaran mediante la palabra reservada eq de la siguiente manera:

```
eq termino1 = termino2  [atributos]
```

También existen las ecuaciones condicionales, pero no se usarán en este programa.

1.3.6. Reglas

Las reglas son un instrumento para la reescritura en Maude. La sintaxis para éstas es la siguiente:

```
rl [etiqueta] : término1 => término2  [atributos]
```

Esto se utiliza para realizar una transición desde un estado a otro. Básicamente consiste en que si se encuentra una instancia de la parte izquierda en el estado actual, en el estado siguiente será reemplazado por la parte derecha de la regla. Lo más potente de este concepto es la posibilidad de que la ejecución de varias reglas sea concurrente.

Las reglas también pueden ser condicionales.

2. Implementación y análisis

En esta sección se presentan dos versiones de la implementación del algoritmo Tableau: Una primera versión donde los operadores adicionales de la lógica temporal se declaran en función de los operadores básicos y una versión extendida donde se declaran reglas de Tableau para los operadores adicionales sin tener que pasar por la conversión a los operadores básicos; para la primera se explican todos los elementos declarados, mientras que para la segunda gran parte del código es parecida, así que sólo se comentan las diferencias con la primera versión.

2.1. Versión simple

La implementación está estructurada en dos módulos: El módulo funcional `TEMPLOGIC` y el módulo de sistema `TEMP-LOGIC-RULES`.

2.1.1. Módulo funcional

En el módulo `TEMPLOGIC`, se van a definir los tipos del programa, su jerarquía, sus operadores y ecuaciones.

```
fmod TEMPLOGIC is
      protecting BOOL .
      protecting QID .
```

`TEMPLOGIC` importa el módulo `BOOL`, ya que el tipo de datos booleano es necesario para expresar el valor de verdad de las fórmulas. También se importa el módulo `QID` para poder utilizar el tipo `Qid` y así utilizar cualquier cadena de caracteres precedida por el carácter " ' " como una variable a la hora de hacer pruebas.

2.1.2. Tipos

Los tipos básicos de `TEMPLOGIC` son:

- **Prop**: el tipo de las proposiciones lógicas.
- **Fórmula:** el tipo para representar fórmulas lógicas
- **Literal**: representa una proposición lógica o su negación (p o ¬p).
- **LiNext**: fórmula compuesta por un literal y/o una expresión `Next`
- **SetLiteral**: conjunto de literales
- **SetLiNext**: conjunto de literales y/o expresiones `Next`
- **SetFormula**: conjunto de fórmulas, que estarán separadas por comas.
- **NESetFormula:** conjunto de fórmulas no nulas. Éste tipo se crea para poder declarar el id, y para considerar casos en los que se necesita que la `SetFormula` no esté vacía.
- **History:** historial de un nodo. Es necesario para la detección de bucles.
- **Estado**: indica la fase de comprobación en la que se encuentra el algoritmo.
- **Nodo:** el elemento nodo engloba el estado actual, seguido de la fórmula y su historial entre corchetes, y finalmente el separador `D( )` que contendrá la fórmula distinguida.
- **SetNodos:** expresa la disyunción entre los nodos del árbol. Éstos están separados por un "punto y coma".

Los tipos anteriores pueden clasificarse en dos clases: los tipos que se refieren a la estructura del árbol: `Nodo`, `SetFormula`, `SetLiNext`, `SetLiteral`, `History` y `SetNodos`, y los tipos que construyen la fórmula PLTL: `Formula`, `Prop`, `Literal`, `LiNext` *y* `Next`.

La jerarquía de tipos está pensada de forma que los tipos referentes a la estructura del árbol serán manipulados de forma separada a los del interior de la fórmula lógica. A la hora de codificar las ecuaciones, no se permite que dentro de una fórmula haya un `SetFormula`. Además, el flujo de ejecución está orientado a convertir los operadores de las fórmulas en `SetFormula` y `SetNodos`, y nunca al revés, excepto en el caso de la función `toFormula` que se verá más adelante.

El tipo `History` se ha declarado como supertipo de `SetFormula` y de todos los subtipos de éste, ya que se construye combinando varios elementos `SetFormula` separados por el operador "<".

La declaración del tipo eventualidad puede parecer necesaria, sin embargo las expresiones de este tipo serán reconocidas mediante el operador `Until` y sus operandos.

A continuación, se presenta el código Maude donde se declaran los tipos.

```
sorts Prop Literal Next LiNext Formula .
sorts SetLiteral SetLiNext NESetFormula SetFormula .
subsort Qid < Prop .


subsort Bool < Prop.
subsort Prop < Literal < LiNext < Formula .
subsort Next < LiNext .
subsort Literal < SetLiteral .
subsort LiNext < SetLiNext .
subsort Formula < NESetFormula < SetFormula .
subsort SetLiteral < SetLiNext < NESetFormula .

sort History .
subsort SetFormula < History .

sorts Nodo SetNodos .
subsort Nodo < SetNodos .

sort Estado .
```

A continuación se muestra un gráfico con la jerarquía de los tipos recién comentados. Las dos clases de tipos mencionadas anteriormente están separadas en diferentes columnas.

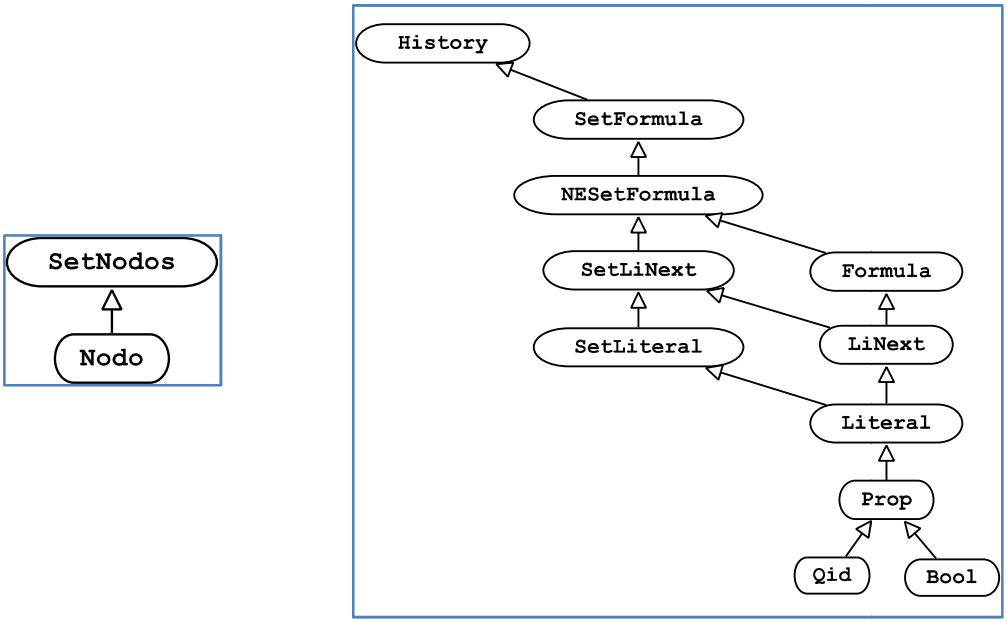

Figura 2: Jerarquía de los tipos

2.1.3. *Operadores*

Operadores básicos

Existen cuatro operadores de base. Por razones técnicas y prácticas, se les representa de otra forma en Maude, como figura en la siguiente tabla.

Tabla 7: Representaciones de los operadores básicos

Nombre	Símbolo	Operador en Maude
Not	¬	^
And	∧	/\
Next	X	X
Until	U	U

Se declaran tres operadores `not`: uno para el tipo `Formula`, otro para el tipo `Next` y otro para el tipo `Prop`. De esta forma se construye el tipo `literal` y el tipo `LiNext`.

```
op ^_    : Formula          -> Formula .
op ^_    : Next             -> LiNext  .
op ^_    : Prop             -> Literal .
```

El operador X se aplica sobre una fórmula, y produce una fórmula de tipo `Next`.

```
op X_    : Formula          -> Next    .
```

Los operadores And y U se aplican sobre el tipo formula y dan como resultado el mismo tipo.

```
op _/\_  : Formula Formula -> Formula [comm assoc] .
op _U_   : Formula Formula -> Formula .
```

En Maude es posible declarar los atributos de asociatividad y conmutatividad sin tener que definir ecuaciones específicas para ello (ver ejemplo anterior). Técnicamente, el operador /\ (*And)* no necesita las propiedades de asociatividad y conmutatividad ya que en el futuro se transformará en el operador coma, y éste sí las tiene. Sin embargo, añadimos estos atributos para evitar que aparezcan avisos de ambigüedad, como por ejemplo:

```
Warning: <standard input>, line 177: ambiguous term, two parses are:
[(('p U 'q) /\ (^ X F 'q /\ ^ 'q)) < noH]D(noD)
-versus-
[((('p U 'q) /\ ^ X F 'q) /\ ^ 'q) < noH]D(noD)

Arbitrarily taking the first as correct.
```

Operadores adicionales

A continuación se declaran los operadores adicionales.

El operador \/ (*Or*) posee los atributos de asociatividad y conmutatividad por la misma razón que el operador /\. Este se convertirá posteriormente en el operador "punto coma".

```
op _\/_  : Formula Formula -> Formula [comm assoc] .
```

Los operadores R, F y G reciben y devuelven el tipo Formula.

```
op _R_   : Formula Formula -> Formula .
op G_    : Formula         -> Formula .
op F_    : Formula         -> Formula .
```

El operador T (then) representa la implicación y se declara por razones técnicas. Será útil para la fase de testeo ya que las pruebas utilizadas contienen este operador.

```
op _T_   : Formula Formula -> Formula .
```

Operador "coma"

El operador coma puede concatenar fórmulas, literales o fórmulas de tipo LiNext. Será asociativo y conmutativo, y tendrá como id el operador nil.

```
op nil :                        -> SetFormula .
op _,_ : SetFormula SetFormula -> SetFormula [comm assoc id: nil] .
op _,_ : SetLiNext SetLiNext   -> SetLiNext  [comm assoc id: nil] .
op _,_ : SetLiteral SetLiteral -> SetLiteral [comm assoc id: nil] .
```

La declaración del id para el operador coma es importante. Implica que un SetFormula S utilizado en una ecuación puede ser nulo, y esto evita tener que escribir una ecuación diferente para cada uno de los casos.

La siguiente declaración del operador coma es necesaria para que el atributo id funcione correctamente. Sin esta declaración, se haría matching de forma indefinida con el elemento nil, y la ejecución no terminaría. La solución consiste en forzar que nil sea de tipo mayor que el de cualquier fórmula. Se usa el tipo NESetFormula ya que es un subtipo del tipo SetFormula.

```
op _,_ : SetFormula NESetFormula -> NESetFormula[comm assoc id: nil] .
```

Operadores constructores del nodo

El operador < permite ordenar el historial por antigüedad (las fórmulas se guardan de menos a más recientes), lo cual permite localizar fácilmente la fórmula actual del nodo. Posee la propiedad asociativa, pero no la conmutativa ya que el orden es significativo.

```
op _<_ : History History -> History [assoc]  .
```

El operador noH indica que no hay historial, esto ocurre en el caso del nodo inicial.

```
op noH :                  -> History .
```

La ejecución de las ecuaciones en Maude es no determinista, lo cual significa que no se puede asegurar que se ejecuten en un orden dado (esto se explica de forma más detallada en el punto 3.3). Para asegurar que el algoritmo sigue el orden esperado se usa el concepto de estado: cada estado representa una fase de comprobación del algoritmo visto en la sección anterior.

Técnicamente, se declara el tipo estado, y a continuación un operador por cada instancia de éste. Se dispondrá de 7 estados: S0 representa la fase inicial y se corresponde con el primer paso del algoritmo de Tableau, S1 se corresponde con el paso 2 del mismo, S2 con el tercer paso, S4 con el quinto, S5 con el cuarto, y SF es la fase final. El cambio de orden en la correspondencia entre los estados S4 y S5 con las fases del algoritmo es un detalle técnico de la implementación y no cambia el resultado de la ejecución.

Las ecuaciones Tableau usarán estos operadores para saber en qué fase del algoritmo se está, y si se hace un *matching* se pasará al estado siguiente del algoritmo.

```
ops S0 S1 S2 S3 S4 S5 SF : -> Estado .
```

Los operadores T0 y T1 representan transiciones entre los estados. Estas transiciones se declaran en el módulo de sistema del programa. Su uso no es imprescindible, sin embargo, técnicamente resulta útil para saber la regla que se está aplicando y por lo tanto la fase actual del algoritmo.

```
ops T0 T1                 : -> Estado .
```

Nótese que las transiciones se realizan para las comprobaciones que resultan ciertas y reescriben la fórmula. Esto solo ocurre para los estados S0 y S1. Para el resto de estados, no se declaran transiciones, ni tampoco para el nodo final.

La representación del nodo requiere simplicidad, a pesar de contener mucha información. Primero se guarda el estado actual del algoritmo, seguido del historial que incluye la fórmula actual, envueltos entre corchetes, y por último la fórmula distinguida.

A continuación, se muestra la declaración del operador nodo:

```
op _[_] D(_)  : Estado History Formula -> Nodo .
```

El hecho de envolver el historial del nodo entre corchetes permite manejarlo fácilmente a la hora de escribir las ecuaciones. Las ecuaciones aplican la reescritura al nodo entero, así se asegura que no haya reescrituras en el interior del historial.

La fórmula distinguida está contenida dentro del separador `D()`. En la explicación del algoritmo se ha visto que esta fórmula solo puede ser una eventualidad (fórmula de forma p U q). Sin embargo, también se pueden distinguir eventualidades a las que se aplica el operador X y para facilitar la distinción entre ambos casos se guardará la fórmula de tipo `Next`. El tipo de la fórmula distinguida será `Formula`. Nótese que solo puede haber una fórmula distinguida en cada estado del nodo.

El operador noD se utiliza para indicar que no hay expresión distinguida en el nodo.

```
op noD : -> Formula .
```

Operador "punto coma"

```
op _;_ : SetNodos SetNodos -> SetNodos [comm] .
```

El operador punto coma representa la disyunción entre nodos o grupos de nodos en el árbol de Tableau. Es conmutativo porque el orden no es importante, en cambio no es asociativo porque se utilizan paréntesis y de esta manera se conserva la profundidad de cada subárbol.

2.1.4. Ecuaciones

Primero se declaran las variables para su posterior uso en las ecuaciones.

```
vars h h1 h2 : History      .
vars s t     : SetFormula   .
vars nes     : NESetFormula .
vars f g d   : Formula      .
vars sl      : SetLiteral   .
vars sln     : SetLiNext    .
```

Las primeras ecuaciones por definir serán aquéllas que expresan los operadores adicionales en función de los básicos.

```
eq true   = ^ false .
eq e \/ f = ^(^ e /\ ^ f) .
eq G(e)   = ^ F(^ e) .
eq F(e)   = true U e .
eq e R f  = ^ (^ e U ^ f) .
eq e T f = ^ (e /\ ^ f) .
```

A continuación, se exponen las ecuaciones que implementan las comprobaciones de Tableau. Se presentan por separado las ecuaciones de cada fase del algoritmo.

Fase 0

El primer paso del algoritmo Tableau consiste en ver si la fórmula contiene `false` o una contradicción. Si el caso se da, se pasa a la fase final, dando como resultado la proposición `false`, y borrando la fórmula distinguida si la hay.

De lo contrario se pasa a la fase 1 (S1).

```
eq S0[(s, false) < h] D(d) = SF[false < (s, false) < h] D(noD) .
eq S0[(s, f,^ f) < h] D(d) = SF[false < (s, f, ^ f) < h] D(noD)
[owise].
eq S0[h] D(d) = S1[h] D(d) [owise] .
```

Fase 1

Si el estado actual es S1, se comprueba si la fórmula actual es igual a `true`, si es así se obtiene un nodo final abierto. Si no, se simplifica la fórmula, por ejemplo eliminando las constantes

`true` o las proposiciones repetidas (nótese que se usa la variable `nes` de tipo `NESetFormula`). Se repite el proceso hasta que la fórmula actual ya no sea simplificable, momento en que se pasará a la fase 2 (S2).

```
eq S1[(^ false) < h] D(d) = SF[(^ false) < h] D(noD) .
eq S1[(nes, ^ false) < h] D(d) = T1[nes  < (nes, ^ false) < h] D(d) .
eq S1[(nes, f, f) < h]D(d) = T1[(nes, f) < (nes, f, f)< h]D(d) [owise] .
eq S1[h] D(d) = S2[h] D(d) [owise] .
```

Fase 2

Si el estado actual es S2, se verifica si la fórmula hace *matching* con una colección de literales (`sl` es una variable de tipo `SetLiteral`). Si es el caso, se transita a un estado final donde el nodo es abierto. En caso contrario se va al estado S3.

```
eq S2[sl < h] D(d) = SF[^ false < sl < h] D(d) .
eq S2[h] D(d) = S3[h] D(d) [owise] .
```

Fase 3

En el estado S3 se verifica si existen bucles en el árbol. Si la fórmula actual `s` se encuentra en algún punto del historial que no sea el actual, se invoca la función `loopIsTrue` para comprobar que en caso de que el historial contenga eventualidades, las premisas de estas se cumplan. En caso afirmativo el nodo final resultante será abierto, de lo contrario será cerrado.

En otro caso se pasa a la fase 4.

```
eq S3[s < h2 < s < h1]D(d) =
SF[loopIsTrue(s < h2) < s < h2 < s < h1] D(noD) .

eq S3[h] D(d) = S4[h]D(d) [owise] .
```

Fase 4

Se comprueba si la fórmula es una colección de fórmulas de tipo `LiNext`. Si es el caso, se invocan las funciones `next` y `nextD` (que serán explicadas en el apartado 3.1.5). Estas dos funciones se encargan de realizar el paso en el tiempo para la fórmula actual y la fórmula distinguida respectivamente. Seguido, se retorna a la fase inicial para repetir todas las comprobaciones.

Si la fórmula actual no es de tipo `LiNext se pasa a la fase 5.`

```
eq S4[sln < h]D(d) = T0[next(sln) < sln < h] D(nextD(d)) .
eq S4[h] D(d) = S5[h] D(d) [owise] .
```

Fase 5

En la fase 5 se ejecutan las ecuaciones Tableau de los operadores.

En el siguiente bloque de código se realiza la operación de la doble negación. La doble negación se convierte en afirmación, y se retorna a la fase inicial para reanudar las comprobaciones.

```
eq S5[(s, ^ ^ f) < h] D(d) = T0[(s, f)  < (s, ^ ^ f) < h] D(d) .
```

En el siguiente bloque de código se presentan las ecuaciones para la conjunción y su negación. Para la conjunción, el operador And se convierte en el operador "coma", y la fórmula distinguida se conserva sea cual sea.

Para la negación de la conjunción, por ser una regla disyuntiva, el resultado es la creación de dos nodos en disyunción (separados por ";").

```
eq S5[(s,(f /\ g)) < h] D(d) =
T0[(s, f, g) < (s,(f /\ g)) < h] D(d) .

eq S5[(s, ^(f /\ g)) < h] D(d) =
T0[(s, ^ f) < (s, ^(f /\ g)) < h]D(d)  ;
T0[(s, ^ g) < (s, ^(f /\ g)) < h] D(d) .
```

A continuación se presentan las ecuaciones para la negación del operador `Until`. El resultado es la creación de dos nodos en disyunción.

```
eq S5[(s, ^(f U g))  < h] D(d) =
T0[(s, ^ f, ^ g) < (s, ^(f U g))  < h] D(d) ;
T0[(s, f, ^ g, ^ X(f U g)) < (s, ^(f U g)) < h] D(d) .
```

Para las cuatro operaciones anteriores, se conserva la fórmula distinguida para el/los nuevo(s) nodo(s). Después de realizar estas operaciones se vuelve a la fase inicial para reanudar las comprobaciones del algoritmo.

El siguiente bloque de ecuaciones contiene el tratamiento de las eventualidades.

La primera ecuación detecta si el nodo está en la fase 5 (S5). Si la eventualidad actual es la que está distinguida, se aplica la regla `Until(s)`. Para negar el contexto actual que es de tipo `SetFormula` se llama a la función `toFormula` para convertir este en el tipo `Formula` y así poder incluirlo dentro de la fórmula `Next`; esta función será explicada en el punto 3.1.5. Después de realizar esta operación, se retorna a la fase inicial mediante la transición T0.

```
eq S5[(s, (f U g)) < h] D(f U g) =
T0[(s, g) < (s,(f U g)) < h] D(noD);
T0[ (s, f, ^ g, X((^ toFormula(s) /\ f) U g))
  < (s, (f U g))
  < h
  ] D(X((^ toFormula(s) /\ f) U g)) .
```

Si el caso anterior no se da, entonces se ejecutan las dos ecuaciones siguientes, que al contener el atributo `owise` sólo se ejecutan en caso de no ejecutarse la primera ecuación.

```
eq S5[(s, d, (f U g)) < h] D(d) =
T0[(s, d, g) < (s, d, (f U g)) < h] D(d);
T0[(s, d, f, ^ g, X(f U g)) < (s, d, (f U g)) < h] D(d) [owise] .
```

La ecuación anterior comprueba si existe alguna fórmula `until` que no sea la actual y que esté distinguida. En este caso se aplica la regla `Until`. Se crean dos nodos en disyunción donde la fórmula distinguida se mantiene.

```
eq S5[(s,(f U g)) < h] D(d) = S5[(s, (f U g)) < h] D(f U g) [owise] .
```

La ecuación anterior distingue la eventualidad actual y vuelve a la misma fase (S5), de forma que se aplicará la regla `Until(s)`.

2.1.5. Funciones

En este apartado se explicarán las funciones auxiliares usadas en el programa.

toFormula

Convierte un `SetFormula` en el tipo `Formula` reemplazando las comas por operadores `/\`. Esto se necesita a la hora de definir la ecuación `Until(s)`, ya que el contexto de tipo `SetFormula` se introduce dentro de la fórmula `Next` que debe ser de tipo `Formula`.

```
op toFormula_ : SetFormula -> Formula .
eq toFormula(nil)    = true .
eq toFormula(nes, e) = toFormula(nes) /\ e .
eq toFormula(e)      = e .
```

El operador `nil` se convierte en `true` ya que semánticamente `true` es el elemento neutro del operador `/\`.

next

Recibe un `SetFormula` y elimina del conjunto todo lo que no esté englobado por el operador `X` o por la negación de éste.

```
op next_ : SetFormula -> SetFormula .
eq next(s,   X(e)) = next(s),  e .
eq next(s, ^ X(e)) = next(s), ^ e [owise] .
eq next(s,   e)    = next(s)      [owise] .
eq next nil        = nil                  .
```

nextD

Recibe una fórmula y devuelve el contenido del operador `X`. La razón por la cual no se usa la función `next` para este propósito es porque el tipo del operando es `Formula`, y porque el elemento nulo para la fórmula distinguida es `noD`. Si se usara la función `Next` el elemento nulo sería `nil` lo cual sería incorrecto.

```
op nextD_ : Formula -> Formula .
eq nextD(X(e)) = e .
eq nextD(e)    = noD [owise] .
```

Las siguientes funciones se usan para evaluar el bucle detectado en el historial en su caso. Se buscan las eventualidades de tipo `p U q` en el bucle y se comprueba si las premisas (q) de éstas son válidas en algún punto. Si el caso se da, el bucle tiene un valor `true`, en caso contrario tiene un valor `false`.

loopIsTrue

Comprueba si en un historial dado las premisas de las eventualidades se han cumplido, esto decidirá si este bucle dará lugar a una hoja cerrada o abierta, y para ello devolverá un booleano.

```
op loopIsTrue(_) : History -> Bool .
eq loopIsTrue(h) = promisesAreFulfilled(historyToSetFormula(h)) .
```

historyToSetFormula

Convierte el historial en un `SetFormula`, para así extraer las premisas de las fórmulas. Nótese que cuando la función encuentra un `noH` devuelve `nil` ya que se espera un tipo `SetFormula` cuyo elemento neutro es `nil`.

```
op historyToSetFormula(_) : History -> SetFormula .
eq historyToSetFormula(s < h) = (s, historyToSetFormula(h)) .
eq historyToSetFormula(s)     = s .
eq historyToSetFormula(noH)   = nil .
```

promisesAreFulfilled

Recibe como entrada el historial convertido en `SetFormula`, y comprueba que las premisas de las eventualidades del historial estén contenidas en el mismo. En caso de contener la proposición `false`, la respuesta es false.

```
op promisesAreFulfilled(_) : SetFormula -> Bool .
eq promisesAreFulfilled(s, false) = false .
eq promisesAreFulfilled(s)        = (s contains promises(s)) [owise] .
```

promises

Extrae las premisas cuando la fórmula es una eventualidad. Todas las eventualidades están expresadas en términos p `U` q, y la premisa por lo tanto será q.

```
op promises(_) : SetFormula -> SetFormula .
eq promises(s, (e U f)) = (promises(s), f) .
eq promises(s, e)       = promises(s) [owise] .
eq promises(nil)        = nil                 .
```

simplify

Elimina las expresiones repetidas en un `SetFormula` de forma que la función `contains` reciba una fórmula simplificada.

```
op simplify(_) : SetFormula -> SetFormula.
eq simplify(s, e, e) = simplify(s, e) .
eq simplify(s) = s [owise] .
```

containsInternal

Comprueba que un `SetFormula` está incluido en otro. Se devuelve `true` si se encuentra, y `false` en caso contrario.

```
op _containsInternal_ : SetFormula SetFormula -> Bool .
eq (s, t) containsInternal t = true .
eq s containsInternal t = false [owise] .
```

contains

Comprueba que un `SetFormula` está contenido en otro. Esta función llama a la función `containsInternal` así se le pasará un `SetFormula` simplificado.

Se fuerza a que la constante `true` esté contenida en todas las expresiones `SetFormula` ya que implícitamente todos los `SetFormula` contienen `true` (elemento neutro).

```
op _contains_ : SetFormula SetFormula -> Bool .
eq s contains t = ((s, ^ false) containsInternal simplify(t)) .
```

2.1.6. *Simplificaciones*

Las siguientes ecuaciones simplifican el árbol obtenido durante la fase final. Las dos primeras borran el historial en caso de encontrarse con la proposición `true` o `false`. El historial se sustituye por noH, y la fórmula distinguida por noD.

```
*** limpieza historico
eq SF[   false  < s < h] D(d) = SF[   false  < noH] D(noD) .
eq SF[(^ false) < s < h] D(d) = SF[(^ false) < noH] D(noD) .
```

Las dos últimas reducen el árbol en caso de tener un nodo que contenga `false` o `true`. Al ser una disyunción, en caso de `false` se guarda el resto de nodos, y en el caso de `true` se borran todos los nodos ya que el árbol tendrá un valor `true`.

```
*** reduccion arbol
vars sn : SetNodos .
eq SF[   false  < h] D(d) ; sn = sn .
eq SF[(^ false) < h] D(d) ; sn = SF[(^ false) < h] D(d) .
```

2.1.7. *Módulo de sistema*

En este punto se analiza el módulo de sistema. `TEMP-LOGIC-RULES` importará el módulo funcional `TEMPLOGIC` para reutilizar los tipos, operadores y ecuaciones declaradas en éste.

En este módulo se declaran las reglas de las transiciones entre los estados del sistema: Si se encuentra el patrón T0 o T1 se pasa al estado correspondiente (S0 o S1). Esto se hizo en primer lugar para facilitar la depuración de los errores. Sin embargo, sabiendo que las reglas se ejecutan después de las ecuaciones, esto garantiza que todas las posibles ecuaciones del módulo funcional han sido ejecutadas antes de pasar a un nuevo estado, y que los términos se han reducido totalmente.

```
mod TEMP-LOGIC-RULES is
protecting TEMPLOGIC .

vars h : History .
vars d : Formula .

rl [transition0] : T0[h] D(d) => S0[h] D(d) .
rl [transition1] : T1[h] D(d) => S1[h] D(d) .
endm
```

Esta primera versión se puede mejorar en cuanto a eficiencia declarando ecuaciones de Tableau para los operadores G, F y \/. Este cambio permitirá recortar el camino a la solución ya que en vez de reescribir el operador en función de los de base y posteriormente aplicar las reglas Tableau para éste, se aplicará directamente la regla correspondiente. Con esto habrá menos reescrituras y por lo tanto menos tiempo de ejecución. Esta mejora se presenta en el siguiente apartado.

2.2. Versión extendida

La segunda versión del programa consiste básicamente en que los operadores adicionales no se expresan a partir de los operadores básicos, sino que se definen ecuaciones Tableaux para ellos. Esto permite mejorar el rendimiento ya que se requieren menos operaciones de reescritura para llegar al resultado.

A continuación se muestra la parte del código que cambia respecto a la primera versión. Se prescinde de las instrucciones que expresan los tres operadores G, F y \/ en función de los

operadores básicos. En lugar de esto, se añaden las líneas que implementan las ecuaciones Tableaux para ellos.

Las dos primeras representan las reglas de los operadores G y ^F. Nótese que estos operadores no generan eventualidades.

```
eq S5[(s,  (G f)) < h] D(d) =
T0[(s, f, X(G f)) < (s, (G f)) < h] D(d) .

eq S5[(s, ^(F f)) < h] D(d) =
T0[(s, ^ f, ^ X(F f)) < (s, ^(F f)) < h] D(d) .
```

Las dos siguientes reescriben el operador \/ y su negación.

```
eq S5[(s, ^(f \/ g)) < h] D(d) =
T0[(s, ^ f, ^ g) < (s, ^(f \/ g)) < h] D(d) .

eq S5[(s,  (f \/ g)) < h] D(d) =
T0[(s, f) < (s, (f \/ g)) < h] D(d) ;
T0[(s, g) < (s, (f \/ g)) < h] D(d) .
```

A continuación se muestran las ecuaciones para las eventualidades F y ^G. Las ecuaciones para éstas son muy parecidas a las del operador U: Si la eventualidad está distinguida, se aplica la regla F(s) o ^ G(s). Si existe una regla distinguida que no sea la actual, se aplica la regla F o G. Finalmente, si no hay ninguna eventualidad distinguida se distingue la actual.

```
eq S5[(s,(F f)) < h] D(F f) =
T0[(s, f) < (s,(F f)) < h] D(noD) ;
T0[(s, (^ f), X(^ toFormula(s) U f)) <(s, (F f)) < h]
      D(X(^ toFormula(s) U f)) .

eq S5[(s, d, (F f)) < h] D(d) =
T0[(s, d, f) < (s, d, (F f)) < h] D(d) ;
T0[(s, d, (^ f), X(F f)) < (s, d, (F f)) < h] D(d) [owise] .

eq S5[(s, (F f)) < h] D(d) =
S5[(s, (F f)) < h] D(F f) [owise] .


eq S5[(s,  ^(G f)) < h] D(^(G f)) =
T0[(s, ^ f) < (s,(G f)) < h] D(noD) ;
T0[(s, X((^ toFormula(s)) U (^ f))) < (s, (G f)) < h]
      D(X((^ toFormula(s)) U (^ f))) .

eq S5[(s, d, ^(G f)) < h] D(d) =
T0[(s, d, ^ f) < (s, d, (G f)) < h] D(d) ;
T0[(s, d, ^ X(G f)) < (s, d, (G f)) < h] D(d) [owise].

eq S5[(s, ^(G f)) < h] D(d) =
S5[(s, ^(G f)) < h] D(^(G f)) [owise] .
```

Finalmente, para que esto funcione correctamente, se necesita especificar cómo se detectan las premisas de las eventualidades ^ G y F por la función promises. Para el caso de F(f) la premisa es f, en cuanto a ^ G(f) la premisa es ^ f.

```
op promises(_) : SetFormula -> SetFormula .
eq promises(s, (e U f)) = (promises(s), f) .
eq promises(s, (F f))   = (promises(s), f)    [owise] .
eq promises(s, ^ (G f)) = (promises(s), ^ f)  [owise] .
eq promises(s, e)       = promises(s)         [owise] .
eq promises(nil)        = nil .
```

2.3. La importancia de los estados

Antes de pasar a la fase de pruebas, es importante resaltar por qué es necesario el uso de los estados.

Las primeras versiones del programa no tenían en cuenta este detalle y por lo tanto fallaban, debido a que no se puede asegurar que todas las comprobaciones Tableau se hagan para cada fórmula. Un ejemplo de esto podría ser:

```
fmod EJEMPLO is
       protecting BOOL .
       protecting QID  .

       sorts History .
       subsort Qid < History .
       subsort Bool < History .

       op noH : -> History .
       op _<_ : History History -> History [assoc]  .

       vars a  b  c : Qid .
       vars h h1 h2 : History .

       eq a < h1 < a < h2 = true .
       eq a < b < h = b < a < b < h .
endfm
```

El ejemplo muestra un programa con el tipo `History` y una ecuación para detectar bucles en el historial, seguida de otra ecuación que crea un bucle en éste. Si se ejecuta un ejemplo simple con una fórmula que contiene un bucle, se crea un bucle infinito, ya que la ejecución no pasa por la detección del bucle.

```
Maude> rewrite in EJEMPLO : 'x < 'y < 'z < 'x < noH .

Fatal error: stack overflow.
This can happen because you have an infinite computation, say a runaway
Etc…
```

3. Pruebas y resultados

3.1. Ejemplo ilustrativo

En esta sección se presenta un ejemplo simple de la ejecución del programa usando la regla extendida de la implementación. Para ello se usará la fórmula `(p U q) /\ ¬ q`, y se explicarán los elementos más importantes para interpretar el resultado.

Para hacer pruebas, se puede usar cualquiera de los tres comandos de Maude `rewrite`, `frewrite` o `search`. En el anexo de este documento se puede encontrar información sobre el funcionamiento de cada uno de ellos.

Para elegir uno de los tres comandos anteriores, hay que tener en cuenta que el programa implementado impone un orden secuencial en la ejecución de las ecuaciones, esto hace que sea cual sea el comando elegido, el resultado será el mismo.

El comando `rewrite` será el que se usará para realizar las pruebas. Éste utiliza una estrategia arriba-abajo (*top down*) y ejecuta las reglas desde el exterior (*outermost*). En casos generales, no es el mejor comando posible, ya que existen posibilidades de que la ejecución no termine, pero en el caso actual resulta ser el más simple y rápido en cuanto a la búsqueda de la solución, y no existe el riesgo de que la ejecución no termine.

Generalmente, el comando `search` suele ser el más recomendado, ya que es el único que asegura que se va a llegar a una solución. Sin embargo, esta ventaja en el caso actual no se va a apreciar, y el resultado será igual que utilizando el `rewrite`.

Las ventajas del comando `frewrite` frente al `rewrite` también resultan invisibles en el caso actual. Éste utiliza una estrategia justa en cuanto a la aplicación de las reglas, de forma que no habrá reglas que nunca se consideren. Como ya se mencionó, esto no tendrá ninguna utilidad ya que el algoritmo actual impone un orden para la ejecución de las reglas.

Se dispone de una opción para visualizar la traza de la ejecución. Para ello se debe invocar el comando siguiente.

```
set trace on .
```

Se emplea el comando `rewrite` para validar la fórmula `(p U q) /\ ^ q`. El estado actual es S0 y por lo tanto el historial es vacio (noH), y tampoco se dispone de una fórmula distinguida (noD).

```
rewrite in TEMP-LOGIC-RULES : S0[(^ 'q /\ ('p U 'q)) < noH]D(noD) .
```

A continuación se muestra una parte de la traza de la ejecución:

```
eq S5[(s:SetFormula,f:Formula /\ g:Formula) < h]D(d) = T0[(s:SetFormula,
f:Formula,g:Formula) < (s:SetFormula,f:Formula /\ g:Formula) < h]D(d) .
s:SetFormula --> nil
f:Formula --> 'p U 'q
g:Formula --> ^ 'q
h --> noH
d --> noD
S5[(^ 'q /\ ('p U 'q)) < noH]D(noD)
```

```
--->
T0[(nil,('p U 'q),^ 'q) < (^ 'q /\ ('p U 'q)) < noH]D(noD)
```

En las dos primeras líneas aparece la ecuación que se va a aplicar (se han ignorado las comprobaciones de las fases anteriores a la fase 5 ya que no se cumplen los casos considerados por éstas). A continuación se presentan todas las instancias para las variables encontradas en la fórmula inicial que permiten la aplicación de la ecuación, y finalmente las dos últimas líneas representan la fórmula antes y después de aplicar la ecuación.

En este paso se está aplicando la ecuación and para convertir el operador `/\` en el operador coma.

El resultado final de la evaluación de la fórmula es el siguiente:

```
rewrite in TEMP-LOGIC-RULES : S0[(^ 'q /\ ('p U 'q)) < noH]D(noD) .

SF[^ false < noH]D(noD)
rewrites: 48 in 1628036047000ms cpu (202ms real) (0 rewrites/second)
result Nodo: SF[^ false < noH]D(noD)
```

El resultado indica que la fórmula `(p U q) /\ ^ q` es cierta. Se han realizado 48 reescrituras (incluyendo ecuaciones y reglas) y el tiempo de ejecución es de 202 milisegundos reales. Debe tenerse en cuenta que cuando la visualización de la traza está activa el tiempo de ejecución aumenta.

Finalmente se ve el tipo del resultado que es `Nodo` seguido del árbol resultante.

Si se quisiera visualizar el historial y el árbol entero habría que comentar las líneas de código que simplifican el árbol y eliminan el historial, y se obtendría lo siguiente:

```
rewrites: 53 in 1628036047000ms cpu (245ms real) (0 rewrites/second)
result SetNodos:

(SF[^ false < 'q < (('p /\ ^ ^ 'q) U 'q)
            < ('p,^ 'q,X (('p /\ ^ ^ 'q) U 'q))
             < ('p,^ 'q,^ 'q,X (('p /\ ^ ^ 'q) U 'q)) < (^ 'q,('p U 'q))
            < (^ 'q /\ ('p U 'q)) < noH]D(noD) ;

SF[false < ('p,^ 'q,^ ^ 'q,X (false U 'q))
         < (^ 'q,X (false U 'q),'p /\ ^ ^ 'q) < (('p /\ ^ ^ 'q) U 'q)
        < ('p,^ 'q,X (('p /\ ^ ^ 'q) U 'q))
         < ('p,^ 'q,^ 'q,X (('p /\ ^ ^ 'q) U 'q)) < (^ 'q,('p U 'q))
         < (^ 'q /\ ('p U 'q)) < noH]D(noD)) ;
SF[false < ('q,^ 'q) < (^ 'q,('p U 'q))
         < (^ 'q /\ ('p U 'q)) < noH]D(noD)
```

Con este resultado se observa que existe un nodo abierto y por lo tanto un modelo para la fórmula, y dos nodos cerrados.

3.2. Pruebas

3.2.1. Pruebas unitarias

Se han realizado pruebas unitarias para todos los operadores y funciones, intentando cubrir la totalidad de los casos. Son 157 las pruebas que se han escrito.

Como ejemplo, se muestran las pruebas que se hicieron para la función `loopIsTrue`. Al lado de cada prueba aparecen los resultados esperados.

```
rew loopIsTrue(noH)                                  . *** ^ false
rew loopIsTrue('p < noH)                             . *** ^ false
rew loopIsTrue(('p  U 'q) < 'p < noH)                . ***   false
rew loopIsTrue(('p  U 'q) < 'q < noH)                . *** ^ false
rew loopIsTrue((('p  U 'q), 'q) < 'p < noH)          . *** ^ false
rew loopIsTrue((('p  U 'q), 'q) < 'q < noH)          . *** ^ false
rew loopIsTrue(('p  U 'q) < ('p U 'q) < 'q < noH)    . *** ^ false
rew loopIsTrue(('p  U 'q) < ('p U 'q) < 'p < noH)    . ***   false
```

El resto de pruebas unitarias se adjunta en el anexo de este documento.

3.2.2. Pruebas globales

En este punto se muestran los principales ejemplos utilizados para verificar el buen funcionamiento del algoritmo. Los cinco ejemplos fueron extraídos de los documentos (1) y (3).

Se utilizará la versión extendida para las pruebas, y se expondrán los resultados en código de Maude así como el árbol que lo representa, para mayor claridad.

Ejemplo 1: Evaluación de la fórmula F ^ 'q /\ ('p U 'q)

```
rewrite in TEMP-LOGIC-RULES : S0[(F ^ 'q /\ ('p U 'q)) < noH]D(noD) .
rewrites: 98 in 1628036047000ms cpu (4ms real) (0 rewrites/second)

result Nodo: SF[^ false < noH]D(noD)
```

A continuación se detallan los pasos más importantes de la aplicación del algoritmo a la fórmula `F ^ 'q /\ ('p U 'q)`:

Primero se aplica la regla And para convertir el operador /\ en el operador “coma”. Seguidamente se distingue la fórmula `p U q` y se aplica la regla `Until(s)` de forma que se crean dos nodos, y se distingue la eventualidad (`p /\ ^ F ^ q) U q` dentro del operador X. En el paso siguiente, al ser la eventualidad distinguida diferente de la eventualidad actual F ^ q, se aplica la regla `until` dando lugar a dos ramas. El nodo de la rama derecha contiene las fórmulas `^ q` y `^^ q` y por lo tanto resulta en un nodo cerrado, mientras que el nodo de la rama izquierda se simplifica (se elimina la fórmula repetida `^ q`) y seguidamente se le aplica el operador X para pasar a un nuevo instante de tiempo.

El resultado obtenido son cuatro hojas cerradas y dos abiertas y por lo tanto la fórmula `F ^ q /\ (p U q)` es válida. Las dos hojas abiertas representan modelos para la fórmula: El primer modelo sería:

$E(S_0)$ = {p = true, q = false} , $E(S_1)$ = {q = true}

El segundo modelo sería:

$E(S_0)$ = {q = true} , $E(S_1)$ = {q = false}

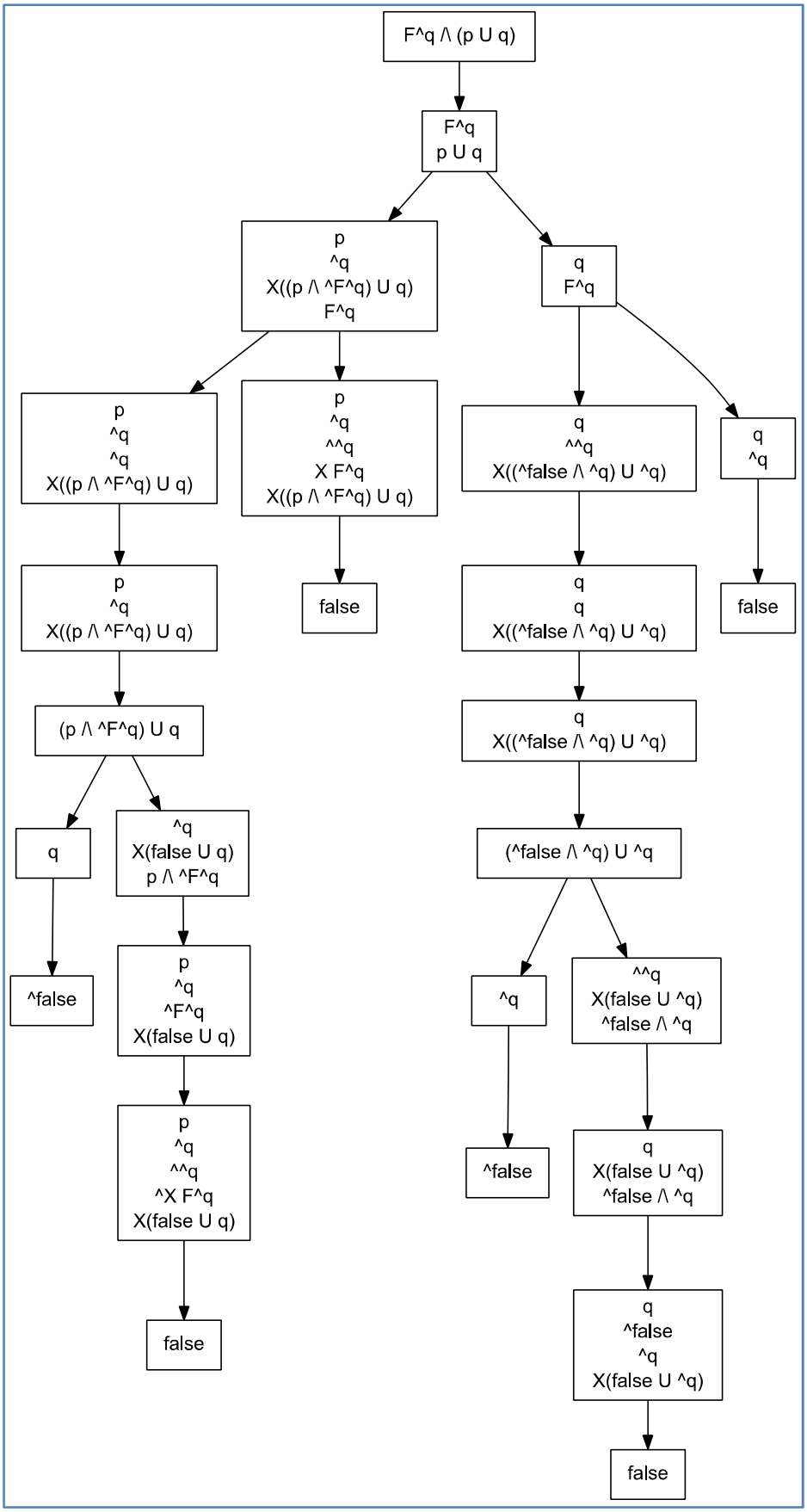

Figura 3: Árbol para la fórmula F ^ q /\ (p U q)

Ejemplo 2: Evaluación de la formula p U false

```
Maude> rewrite in TEMP-LOGIC-RULES : S0[('p U false) < noH]D(noD) .
rewrites: 43 in 1628036047000ms cpu (0ms real) (0 rewrites/second)

result Nodo: SF[false < noH]D(noD)
```

La evaluación de la fórmula `p U false` ocurre de la siguiente forma: la eventualidad `p U false` se distingue y se aplica la regla `until(s)`. De ello se crean dos nodos: El primero es: `p, ^ false, X(false U false)`, donde el primer `false` de la fórmula `until` (dentro del operador X) resulta de la negación del contexto, que al ser vacío, es igual a la constante `true`. A continuación se aplica la regla del paso en el tiempo `(Next)`. Al final de la evaluación se obtienen tres nodos con la constante `false`, es decir, cerrados, y por lo tanto la fórmula `p U false` es falsa.

El árbol resultado se muestra a continuación:

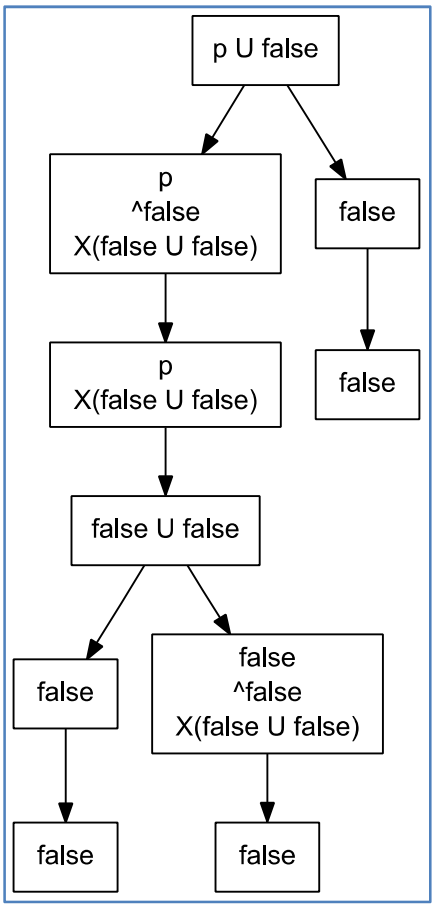

Figura 4: Árbol para la fórmula p U false

Ejemplo 3: Evaluación de la formula ^ 'q /\ ^ X F 'q /\ ('p U 'q)

```
Maude> rewrite in TEMP-LOGIC-RULES : S0[((^ 'q /\ ^ X F 'q) /\ ('p U
'q)) < noH]D(noD).
```

```
rewrites: 95 in 1628036047000ms cpu (45ms real) (0 rewrites/second)

result Nodo: SF[false < noH]D(noD)
```

El resultado indica que la fórmula `(^ q) /\ (^ X F q) /\ (p U q)` no es válida, debido a que existen tres nodos finales cerrados en el árbol.

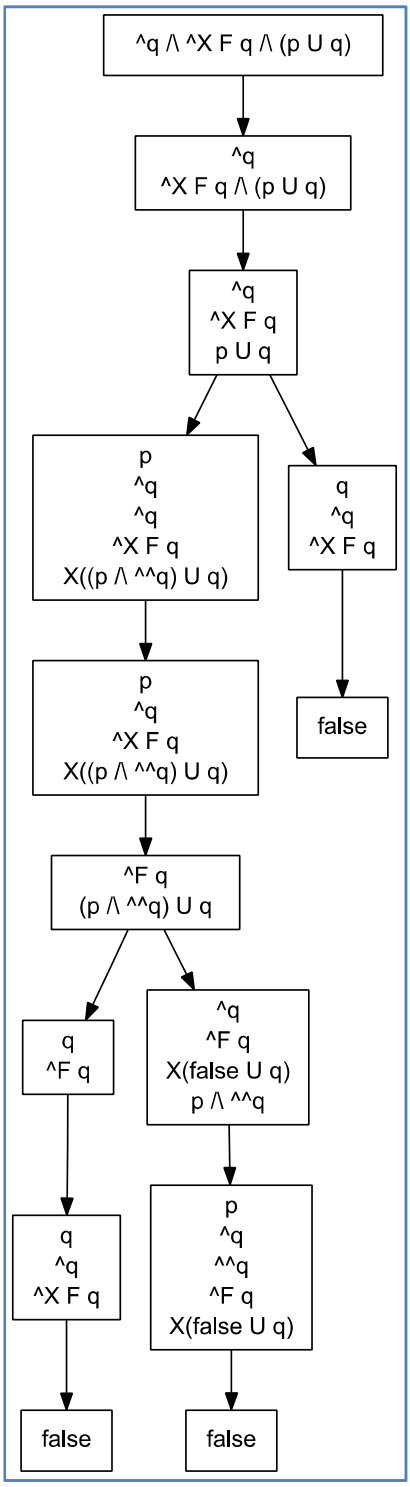

Figura 5: Árbol para la fórmula (^ q) /\ (^ X F q) /\ (p U q)

Ejemplo 4: Evaluación de la fórmula 'p /\ X ^ 'p /\ (^ false U ^ 'p)

```
Maude> rewrite in TEMP-LOGIC-RULES : S0[('p /\ X ^ 'p /\ (^ false U ^
'p)) < noH]D(noD) .
rewrites: 63 in 1628036047000ms cpu (1ms real) (0 rewrites/second)
result Nodo: SF[^ false < noH]D(noD)
```

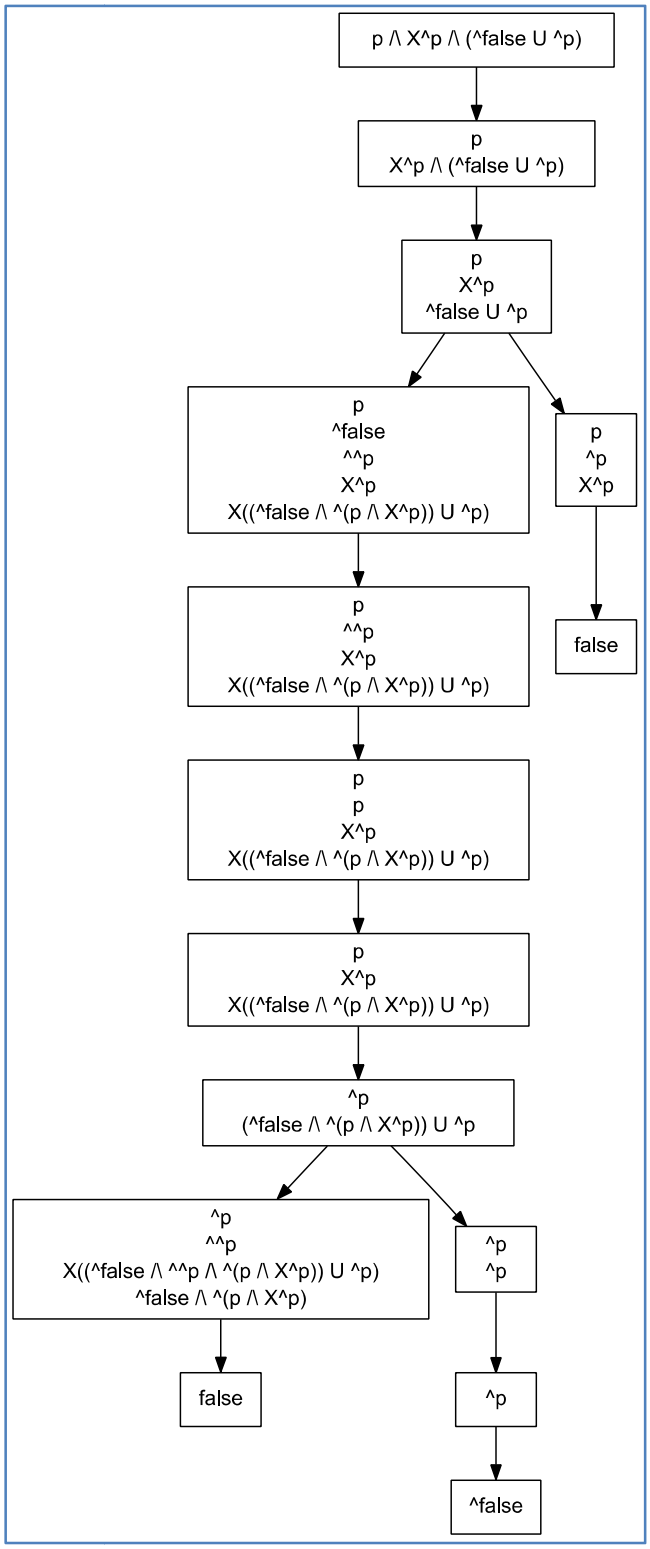

Figura 6: Árbol para la fórmula p /\ X ^ p /\ (^ false U ^ p)

El árbol y la solución obtenidos indican que la fórmula `p /\ X ^ p /\ (^ false U ^ p)` es cierta, ya que se obtienen dos nodos cerrados y uno abierto.

Ejemplo 5: Evaluación de la formula G F 'p /\ F ^ 'q

```
Maude> rewrite in TEMP-LOGIC-RULES : S0[(G F 'p /\ F ^ 'q) < noH]D(noD)
.
rewrites: 251 in 1628036047000ms cpu (4ms real) (0 rewrites/second)
result Nodo: SF[^ false < noH]D(noD)
```

La fórmula `G F p /\ F ^ q` resulta válida. Durante la ejecución se detecta la existencia de varios bucles. Este ejemplo resulta muy largo para exponer, así que se mostrarán los puntos más importantes de la detección de bucle:

La primera línea contiene la ecuación que detecta el bucle. El nodo repetido es el de la fórmula `G F 'p`.

```
eq S3[s:SetFormula < h2:History < s:SetFormula < h1:History]D(d) =

SF[ loopIsTrue(s:SetFormula < h2:History) < s:SetFormula <
h2:History < s:SetFormula < h1:History]D(noD) .

s:SetFormula --> G F 'p

h2:History --> ('p,X G F 'p) < X G F 'p,F 'p

h1:History --> ('p,^ 'q,X G F 'p) < ('p,X G F 'p,F ^ 'q)
               < (X G F 'p,F'p,F ^ 'q) < (G F 'p,F ^ 'q)
               < (G F 'p /\ F ^ 'q) < noH
d --> noD

--->

SF[loopIsTrue(G F 'p < ('p,X G F 'p) < X G F 'p,F 'p) < G F 'p
                     < (('p,X G F  'p) < X G F 'p,F 'p) < G F 'p
                     < ('p,^ 'q,X G F 'p) < ('p,X G F 'p,F ^ 'q)
                     < (X G F 'p,F 'p,F ^ 'q) < (G F 'p,F ^ 'q)
                     < (G F 'p /\ F ^ 'q) < noH]D(noD)
```

A continuación se aplica la función `loopIsTrue` al bucle para averiguar si se han cumplido las premisas de las eventualidades.

```
*********** equation
eq loopIsTrue(h) = promisesAreFulfilled(historyToSetFormula(h)) .

h --> G F 'p < ('p,X G F 'p) < X G F 'p,F 'p

loopIsTrue(G F 'p < ('p,X G F 'p) < X G F 'p,F 'p)

--->
promisesAreFulfilled(historyToSetFormula(G F 'p < ('p,X G F 'p) < X G F
'p,F 'p))
```

En esta parte se comprueba que las premisas, en este caso p, se encuentran en el histórico convertido en un `SetFormula` y se obtiene una respuesta positiva. De esta forma el bucle se sustituye por un nodo abierto.

```
*********** equation
eq (s:SetFormula,t:SetFormula) containsInternal t:SetFormula = true .
s:SetFormula --> ^ false,X G F 'p,X G F 'p,G F 'p,F 'p
t:SetFormula --> 'p
```

```
('p,^ false,X G F 'p,X G F 'p,G F 'p,F 'p) containsInternal 'p
--->
true
********** equation
eq true = ^ false .
empty substitution
true
--->
^ false
```

3.3. Comparación de resultados

Para testear las implementaciones, primero se han utilizado las cinco pruebas globales expuestas en el apartado anterior. Al ver que éstas dan el resultado esperado, se ha querido testear el programa de forma más exhaustiva.

Se han ejecutado 33 ejemplos escogidos de las páginas de las herramientas T*RS tool* (4) y *TTM* (5). La herramienta TTM será nuestra referencia para comparar el coste temporal ya que se basa en el mismo algoritmo utilizado para este proyecto. Sin embargo, ésta utiliza una optimización que aquí no se implementa, y por esta razón los ejemplos se ejecutarán con la optimización desactivada, de manera que la comparación tendrá más sentido. Otro detalle importante es que la herramienta, además de ejecutar el algoritmo, dibuja el árbol de salida, cosa que nuestro programa no hace, así que es de esperar que la herramienta TTM tarde más.

A continuación se presentan los resultados obtenidos en dos tablas: la primera tabla compara las dos versiones de la implementación en cuanto a la satisfacibilidad, mientras que la segunda lo hace en cuanto al tiempo de ejecución en segundos. La primera columna representa los resultados de la herramienta TTM, la segunda los de la primera versión de la implementación y la tercera los de la versión extendida de la misma.

Tabla 8: Comparación de la satisfacibilidad

Test	Satisfacible	TlOrden	TLOrdenExt
paqui1	Sí	Sí	Sí
paqui2	Sí	Sí	Sí
unsat1	No	No	No
unsat2	No	No	No
unsat3	No	??	No
unsat4	No	??	??
unsat5	No	No	No
unsat6	No	No	No
unsat7	No	No	No
unsat8	No	No	No
unsat9	No	No	No
unsat10	No	No	No
unsat11	No	No	No
unsat12	No	No	No
unsat13	No	No	No
unsat14	No	No	No

unsat15	No	No	No
unsat16	No	No	No
sat1	Sí	Sí	Si
sat2	Sí	Sí	Si
sat3	Sí	Sí	Si
sat4	Sí	Sí	Si
sat5	Sí	Sí	Si
sat6	Sí	Sí	Si
sat7	Sí	Sí	Si
sat8	Sí	Sí	Si
sat9	Sí	Sí	Si
sat10	Sí	Sí	Si
sat11	Sí	Sí	Si
sat12	Sí	Sí	Si
sat13	Sí	Sí	Si
sat14	Sí	Sí	Si

En la tabla anterior presentamos la respuesta del algoritmo respecto a la satisfacibilidad de la fórmula: cuando la fórmula es válida la celda de la respuesta es azul, de lo contrario es de color rojo, las celdas de con dos puntos de interrogación indican que la ejecución no termina por falta de tiempo.

Se observa que para los casos que terminan, la respuesta de ambas versiones de la implementación es la misma que la de la herramienta TTM. En cambio, la primera versión no termina para dos casos, y la versión extendida no termina para el ejemplo `unsat4`.

Tabla 9: Comparación del coste temporal en segundos

Test	TTM	TLOrden	TLOrdenExt
paqui1	0	0	0
paqui2	84,07	5,011	0,448
unsat1	0	0	0,001
unsat2	0,01	0,002	0,001
unsat3	0,66	??	5,26
unsat4	35,8	??	??
unsat5	0,01	0,001	0,001
unsat6	0	0,003	0
unsat7	0,03	0,126	0,016
unsat8	0,02	0,002	0,001
unsat9	179,87	2427,935	12,53
unsat10	1,72	0,369	0,112

unsat11	0,05	0,003	0,002
unsat12	0,06	0,143	0,145
unsat13	0,01	0,001	0,001
unsat14	0,05	0,009	0,012
unsat15	0,05	0,014	0,006
unsat16	0,05	0,004	0,002
sat1	0,45	0,01	0,001
sat2	0,02	0,003	0
sat3	0,03	0,005	0,004
sat4	2,93	0,005	0,002
sat5	0,1	0,002	0,001
sat6	0,42	0,02	0,009
sat7	0,4	0,067	0,027
sat8	0,18	0,007	0,003
sat9	0,12	0,004	0,001
sat10	14,47	0,161	0,114
sat11	0,01	0,002	0
sat12	0,13	0,007	0,005
sat13	26,56	1,669	0,645
sat14	215,66	41,38	0,2

La tabla 9 representa los tiempos de ejecución de las tres herramientas para cada ejemplo. En azul se representan los resultados en los que la implementación del proyecto tarda menos que la herramienta TTM, y en rojo lo contrario.

En general se puede observar que las dos versiones propuestas en este proyecto son más rápidas que la herramienta TTM. Aunque hay que tener en cuenta que la TTM además de aplicar el algoritmo dibuja el árbol de salida. Si se compara la primera versión con la versión extendida, se constata que la segunda acorta mucho el tiempo de ejecución para los casos más largos. Sin embargo, para los casos normales la diferencia no es significativa.

Para el ejemplo `unsat4`, la herramienta TTM tarda unos 35.8 segundos lo cual es bastante largo. La TTM además implementa una versión para optimizar el tratamiento de eventualidades, propuesta en el documento (3 p. 716) y que en este trabajo no se considera. Dicha optimización consiste en detectar un tipo de fórmulas llamadas persistentes, al ser de valor `true`, su negación como contexto daría una respuesta `false`. De esta forma se cierran ramas que de antemano se puede saber que darán una respuesta negativa y se ahorra mucho tiempo. Dicha optimización disminuiría el tiempo de ejecución y conseguiría que el ejemplo termine.

4. Conclusiones

4.1. Objetivos Conseguidos

Este proyecto presenta la implementación de un algoritmo para la determinación de la satisfacibilidad de una fórmula lógica PLTL. El algoritmo utilizado (presentado en (1)) es óptimo en cuanto a que evita el cómputo en las fases de los algoritmos tradicionales. Así pues, la implementación es competitiva en tiempo de ejecución como así lo demuestran las pruebas realizadas.

La elección de Maude como lenguaje proporciona flexibilidad para la implementación, fácil extensibilidad, simplicidad y un alto rendimiento y resulta muy conveniente para este proyecto.

Para realizar este proyecto se pasó por varias fases. El primer paso fue adquirir conocimientos sobre la lógica temporal y sus operadores, así como lograr el entendimiento del algoritmo Tableau en profundidad para dicha lógica. También se hizo un estudio de los diferentes algoritmos propuestos para finalmente elegir una versión que garantizara la corrección y la optimización de los resultados.

Por otro lado fue necesario aprender a manejar el lenguaje de programación Maude. Se exploraron muchos de sus elementos para así descubrir cuáles serían útiles para la implementación.

Al principio se encontraron dificultades para la adaptación del algoritmo al lenguaje elegido, lo que sin embargo permitió adquirir conocimientos sólidos de Maude. La modelación de los tipos resultó crucial para el buen funcionamiento del algoritmo, y la elección de una buena representación del nodo facilitó la escritura de las ecuaciones y reglas.

Cabe mencionar que se intentó implementar una versión simplificada del algoritmo, sin tener en cuenta la distinción de las fórmulas, la detección de bucles ni la imposición de un orden para las comprobaciones. Sin embargo, el algoritmo resultó ser incompleto y no finalizaba en algunos casos.

Llegados a la fase de testeo, se hicieron pruebas unitarias para cada operador, función y regla, lo cual ayudó a detectar varios fallos. Después se realizaron unas 33 pruebas globales, extraídas de las páginas web de las herramientas TRS (4) y TTM (5) , para compararlas con éstas.

En cuanto a corrección, el algoritmo ha conseguido dar las respuestas esperadas para todos los casos exceptuando uno, para el cual la ejecución no termina. El coste temporal es similar al de la herramienta TTM, lo cual es satisfactorio teniendo en cuenta que esta implementa optimizaciones que no se consideraron aquí.

4.2. Posibles ampliaciones

Desde el punto de vista del algoritmo, una posible ampliación sería implementar la mejora propuesta en el documento (3 p. 716) para el tratamiento de las eventualidades. En teoría esto conseguiría mejores resultados que los obtenidos y con menor coste temporal.

Desde el punto de vista de la presentación, se podría añadir una interfaz para el usuario. De esta forma no se exigirían conocimientos de Maude al usuario. Si además se desea que la herramienta esté disponible en la red, sería interesante crear una página web para ésta, quedando accesible para fines académicos o de investigación por cualquier persona. Para una representación más clara de los resultados, se podría añadir una herramienta para visualizar el árbol resultante.

Anexo

Sintaxis básica de Maude

En esta sección se expondrán los elementos más importantes de la sintaxis de Maude, especialmente los utilizados para la implementación.

Identificadores

Es el elemento más básico de la sintaxis de Maude. Se utiliza para dar nombre a los módulos declarados, tipos, operadores, etc.

El identificador puede estar formado por cualquier cadena de caracteres, a excepción de los espacio en blanco, llaves, corchetes, paréntesis, comas o comillas.

Módulos

Un programa Maude está compuesto por módulos. En Maude existen tres tipos de módulos: Funcional, de sistema y orientado a objetos. De éstos solamente se verán los dos primeros.

El módulo funcional es donde se declaran los tipos de los datos del sistema (sorts), los operadores (ops) que son las operaciones aplicables sobre éstos y las ecuaciones (eqs) que sirven para reducir o simplificar los términos formados por los dos anteriores. Una expresión de entrada será reducida por las ecuaciones del módulo funcional hasta que quede en forma canónica, es decir, cuando ya no se puede reducir más. Se declara de esta manera:

```
fmod NombreMóduloFuncional
{Declaraciones del módulo}
endfmod
```

En el módulo de sistema se definen las reglas del sistema, aunque en él también se pueden definir tipos, operadores o ecuaciones. Las reglas definen las transiciones que puede haber entre diferentes estados del sistema y su ejecución puede ser concurrente. Éstas se aplican cuando los términos ya no son reducibles por las ecuaciones definidas.

Se declara con la siguiente sintaxis:

```
fmod NombreMóduloSistema
{Declaraciones del módulo}
endfmod
```

En un módulo se puede importar otro módulo para hacer uso de sus declaraciones. Por ejemplo, se pueden importar módulos ya definidos en Maude como por ejemplo: BOOL, NAT, FLOAT, STRING**,** o bien un módulo de desarrollo propio**.** Esto se haría mediante las palabras reservadas `including`**,** `protecting` o bien `extending`**.**

`Protecting` se usa cuando se quiere importar módulos para utilizarlos estrictamente tal y como están definidos. `Including` permite redefinir las operaciones del módulo origen, y también definir nuevas operaciones si se desea. Finalmente, `extending` es un término medio entre `protecting` e `including`. Este último permite definir nuevas operaciones en el módulo extendido, pero no modificar las operaciones declaradas en este. En el programa

desarrollado se usará la operación `protecting`, ya que no se necesita modificar los módulos reutilizados.

Tipos (Sorts)

Lo primero que se necesita definir en un sistema son los tipos. La definición de tipos se lleva a cabo de la siguiente manera:

```
sort  sortId  .
```

Si se quiere definir varios tipos de golpe, se puede utilizar la palabra `sorts`:

```
sorts  sortId1,  sortId2,  … ,  sortIdX  .
```

Nótese la importancia de que todas las instrucciones de un programa escrito en Maude acaben en un punto precedido por un espacio, pues de lo contrario la compilación fallaría.

Para establecer una jerarquía entre los tipos se definen los subtipos. Hay que prestar atención a no declarar ciclos en la jerarquía.

```
subsort  sortId1 <  sortId2 .
subsort  sortId2 <  sortId3 .
```

O bien:

```
subsorts  sortId1 <  sortId2 <  sortId3 .
```

Operadores

La sintaxis para declarar operadores es la siguiente:

```
op opId :  sortId1  sortId2  …  sortIdX  ->  sortOperator
[operator attributes] .
```

Si varios operadores comparten los mismos tipos, se declaran de la siguiente forma:

```
ops  opId1  opId2:  sortId1  sortId2  …  sortIdX  ->
sortOperator .
```

Variables

Se pueden declarar instancias de los tipos definidos para un sistema, para así poder definir sus ecuaciones y reglas.

Las variables se declaran de la siguiente forma:

```
var N : tipoN .
```

Si son varias variables del mismo tipo:

```
vars N M : tipoN .
```

Términos

Un término es la aplicación de un operador a una variable o bien constante. Por ejemplo:

```
opNameid1(N: tipoN)
```

O bien

```
_+_(N:Nat, M:Nat)
```

Ecuaciones no condicionales

Las ecuaciones se declaran mediante la palabra reservada eq de la siguiente manera:

```
eq termino1 = termino2  [atributos]
```

También existen las ecuaciones condicionales, pero no se usarán en este programa.

Atributos

Los atributos pueden ser definidos para los operadores o las ecuaciones. Los atributos que serán utilizados para la implementación son los siguientes:

`assoc`: Propiedad de asociatividad. Significa que no hace falta el uso de los paréntesis para separar los términos que contengan el operador.

`Comm`: Propiedad de conmutatividad, que significa que se puede intercambiar el orden de los operandos.

`(Id: término)`: Define el elemento identidad del operador.

`Owise`: El atributo de ecuación `otherwise` permite expresar lo que pasa en los casos que la primera ecuación no cubre. Además permite imponer un orden para la ejecución de las ecuaciones de un mismo bloque `owise`.

Dominio de tipo (Kind)

Es un concepto que no está definido por el propio usuario, a diferencia de los tipos. Los tipos están agrupados implícitamente en componentes conexos. En algunos casos se puede cometer el error de que el resultado de una operación esté fuera del dominio de los tipos que lo definen. En este caso, una solución sería definir el tipo de éste resultado mediante su dominio, esto es, `[tipoOperador]`. Los dominios de tipo son supertipos de error, de esta manera si se define un término mediante éstos, es considerado un término de error.

Reglas

Las reglas son un instrumento para la reescritura en Maude. La sintaxis para éstas es la siguiente:

```
rl [etiqueta] : término1 => término2  [atributos]
```

Esto se utiliza para realizar una transición desde un estado a otro. Básicamente consiste en que si se encuentra una instancia de la parte izquierda en el estado actual, en el estado siguiente será reemplazado por la parte derecha de la regla. Lo más potente de este concepto es la posibilidad de que la ejecución de varias reglas sea concurrente.

Las reglas también pueden ser condicionales.

Comandos de ejecución

En esta sección se explicarán los comandos de Maude que serán de más utilidad.

```
reduce [in module]  term1
```

Reduce el término `term1`, haciendo uso de los axiomas definidos sobre los tipos y las ecuaciones del módulo funcional. Puede reemplazarse por la palabra `red`. Si no se especifica la parte `[in module]` se tendrá en cuenta el módulo actual.

```
Set trace on
```

Este comando permitirá mostrar la traza completa de la ejecución de los comandos en Maude.

```
Set trace off
```

Esto hace lo contrario que el comando anterior. Se desactiva la visualización de la traza.

```
rewrite [n]  [in module] term1
```

n: número máximo de pasos por aplicar.

Reescribe el término `term1` utilizando las ecuaciones, axiomas y reglas definidas en el módulo.

Este comando hará tantas iteraciones como número de reglas aplicadas que se especifica por el usuario. Si no se especifica este número, se asume que es infinito. El comando puede sustituirse por `rew`.

Rewrite utiliza la estrategia de búsqueda en profundidad (*top down*) y ejecuta las reglas desde el exterior (*outermost*). La estrategia en profundidad puede causar que haya reglas que nunca lleguen a ejecutarse, y puede que así la ejecución nunca termine.

```
frewrite [n,m]  [in module] term1
```

n: número máximo de reglas por aplicar.

m: número máximo de reescrituras para un término.

Este comando hace exactamente lo mismo que el `rewrite`, excepto que `frewrite` no explora las soluciones de izquierda a derecha, con el objetivo de que todas las reglas tengan la misma prioridad para decidir el curso de la ejecución.

Esto podría resolver el inconveniente del comando `rewrite`, y asegurar que la ejecución termina. Sin embargo, el inconveniente es que puede que la solución obtenida no esté totalmente reducida.

```
search [n, m] [in module] term1  tipoBusqueda term2  {such that condición}
```

n: número máximo de soluciones que se quiere encontrar. Este límite es importante porque existen sistemas en los que la ejecución no acabaría, de esta forma se garantiza el fin de la ejecución.

`m`: La máxima profundidad de la búsqueda

`term1`: El término del que parte la búsqueda.

`term2`: El término al que se quiere llegar.

`TipoBusqueda`: La forma en que se efectuará la búsqueda del término `term2`. Las opciones de las que se dispone para el tipo de búsqueda son los siguientes:

`=> 1`: El proceso de reescritura se realizará en un paso.

`=> +`: El proceso de reescritura se realizará en al menos un paso.

`=> *`: El proceso de reescritura se realizará en 0, 1, o muchos pasos.

`=> !`: El proceso de reescritura finalizará cuando se llegue a nodos finales.

El comando `search` sigue la estrategia de búsqueda en anchura. Tiene la ventaja frente a los comandos `rewrite` y `frewrite` de que realiza la búsqueda de diferentes formas para alcanzar el término deseado, lo cual garantiza el hecho de encontrar una solución si ésta existe.

```
Continue number .
```

El comando `continue` permite seguir con la ejecución viendo los siguientes niveles de profundidad, y se puede utilizar en el caso del `frewrite` y `rewrite` también. Se puede definir nuevamente el número máximo de profundidad (`number`).

Código versión extendida

A continuación se expone el código que resuelve los objetivos de este proyecto:

```
fmod TEMPLOGIC is
protecting BOOL .
protecting QID .

*** TIPOS
sorts Prop Literal Next LiNext Formula .
sorts SetLiteral SetLiNext NESetFormula SetFormula .
subsort Qid < Prop .
subsort Bool < Prop .
subsort Prop < Literal < LiNext < Formula .
subsort Next < LiNext .
subsort Literal < SetLiteral .
subsort LiNext < SetLiNext .
subsort Formula < NESetFormula < SetFormula .
subsort SetLiteral < SetLiNext < NESetFormula .

sort History .
subsort SetFormula < History .
```

```
sorts Nodo SetNodos .
subsort Nodo < SetNodos .

sort Estado .
ops S0 S1 S2 S3 S4 S5 SF : -> Estado .
ops T0 T1                : -> Estado .

*** VARIABLES
vars h h1 h2 : History      .
vars s t     : SetFormula   .
vars nes     : NESetFormula .
vars f g d   : Formula      .
vars sl      : SetLiteral   .
vars sln     : SetLiNext    .


*** OPERADORES LOGICA TEMPORAL
*** operadores de base
op ^_    : Formula         -> Formula .
op ^_    : Next            -> LiNext .
op ^_    : Prop            -> Literal .
op _/\_  : Formula Formula -> Formula [comm assoc] .
op X_    : Formula         -> Next    .
op _U_   : Formula Formula -> Formula .

*** operadores adicionales
op _\/_  : Formula Formula -> Formula [comm assoc] .
op _R_   : Formula Formula -> Formula .
op G_    : Formula         -> Formula .
op F_    : Formula         -> Formula .
op _T_   : Formula Formula -> Formula .

eq true   = ^ false .
eq f R g  = ^ (^ f U ^ g) .
eq f T g = ^ (f /\ ^ g) .

*** NIVEL SETPROP
op nil :                          -> SetFormula .
op _,_ : SetFormula SetFormula    -> SetFormula   [comm assoc id: nil ] .
op _,_ : SetFormula NESetFormula  -> NESetFormula [comm assoc id: nil ] .
op _,_ : SetLiNext  SetLiNext     -> SetLiNext    [comm assoc id: nil ] .
op _,_ : SetLiteral SetLiteral    -> SetLiteral   [comm assoc id: nil ] .

*** NIVEL NODO
op _[_] D(_)  : Estado History Formula    -> Nodo .
op _;_ : SetNodos SetNodos -> SetNodos [comm] .
op noD : -> Formula .

*** HISTORY
op noH : -> History .
op _<_ : History History -> History [assoc]  .


*** Estado 0 : contradicciones
eq S0[(s, false)  < h] D(d) = SF[false < (s, false)  < h] D(noD) .
eq S0[(s, f, ^ f) < h] D(d) = SF[false < (s, f, ^ f) < h] D(noD)[owise] .
eq S0[h] D(d) = S1[h] D(d) [owise] .

*** Estado 0 : simplificaciones
eq S1[(^ false) < h] D(d)      = SF[(^ false) < h] D(noD) .
eq S1[(nes, ^ false) < h] D(d) = T1[nes  < (nes, ^ false) < h] D(d) .
eq S1[(nes, f, f) < h]D(d)     = T1[(nes, f) < (nes, f, f)< h]D(d) [owise] .
eq S1[h] D(d) = S2[h] D(d) [owise] .

*** Estado 2 : literales
eq S2[sl < h] D(d) = SF[^ false < sl < h] D(d) .
eq S2[h] D(d) = S3[h] D(d) [owise] .
```

```
*** Estado 3 : Histórico
eq S3[s < h2 < s < h1] D(d) = SF[loopIsTrue(s < h2) < s < h2 < s < h1]D(noD) .
eq S3[h] D(d) = S4[h] D(d) [owise] .

*** Estado 4
eq S4[sln < h]D(d) = T0[next(sln) < sln < h] D(nextD(d)) .
eq S4[h] D(d) = S5[h] D(d) [owise] .

*** Estado 5
*** ecuaciones tableaux
eq S5[(s,  (G f)) < h] D(d) =
T0[(s,   f,   X(G f)) < (s,  (G f)) < h] D(d) .

eq S5[(s, ^(F f)) < h] D(d) =
T0[(s, ^ f, ^ X(F f)) < (s, ^(F f)) < h] D(d) .

eq S5[(s, ^(f \/ g)) < h] D(d) =
T0[(s, ^ f, ^ g) < (s, ^(f \/ g)) < h] D(d) .

eq S5[(s,  (f \/ g)) < h] D(d) =
T0[(s, f) < (s, (f \/ g)) < h] D(d) ;
T0[(s, g) < (s, (f \/ g)) < h] D(d) .

eq S5[(s, (f U g)) < h] D(f U g) =
T0[(s, g) < (s,(f U g)) < h] D(noD);
T0[ (s, f, ^ g, X((^ toFormula(s) /\ f) U g))
  < (s, (f U g))
  < h
  ] D(X((^ toFormula(s) /\ f) U g)) .

eq S5[(s, d, (f U g)) < h] D(d) =
T0[(s, d, g) < (s, d, (f U g)) < h] D(d);
T0[(s, d, f, ^ g, X(f U g)) < (s, d, (f U g)) < h] D(d) [owise] .

eq S5[(s,(f U g)) < h] D(d) =
S5[(s, (f U g)) < h] D(f U g) [owise] .


eq S5[(s,(F f)) < h] D(F f) =
T0[(s, f) < (s,(F f)) < h] D(noD) ;
T0[(s, (^ f), X(^ toFormula(s) U f)) <(s, (F f)) < h]
             D(X(^ toFormula(s) U f)) .

eq S5[(s, d, (F f)) < h] D(d) =
T0[(s, d, f) < (s, d, (F f)) < h] D(d) ;
T0[(s, d, (^ f), X(F f)) < (s, d, (F f)) < h] D(d) [owise] .

eq S5[(s, (F f)) < h] D(d) =
S5[(s, (F f)) < h] D(F f) [owise] .


eq S5[(s,  ^(G f)) < h] D(^(G f)) =
T0[(s, ^ f) < (s,(G f)) < h] D(noD) ;
T0[(s, X((^ toFormula(s)) U (^ f))) < (s, (G f)) < h]
             D(X((^ toFormula(s)) U (^ f))) .

eq S5[(s, d, ^(G f)) < h] D(d) =
T0[(s, d, ^ f) < (s, d, (G f)) < h] D(d) ;
T0[(s, d, ^ X(G f)) < (s, d, (G f)) < h] D(d) [owise].

eq S5[(s, ^(G f)) < h] D(d) =
S5[(s, ^(G f)) < h] D(^(G f)) [owise] .


*** FUNCIONES
*** funciones para el Next
op next_  : SetFormula -> SetFormula .
eq next(s,   X(f)) = next(s),  f .
```

```
eq next(s, ^ X(f)) = next(s), ^ f [owise] .
eq next(s,   f)    = next(s)      [owise] .
eq next nil        = nil                  .

op nextD_ : Formula -> Formula .
eq nextD(X(f)) = f .
eq nextD(f)    = noD [owise] .

*** funciones para la detección del bucle
op loopIsTrue(_) : History -> Bool .
eq loopIsTrue(h) = promisesAreFulfilled(historyToSetFormula(h)) .

op historyToSetFormula(_) : History -> SetFormula .
eq historyToSetFormula(s < h) = (s, historyToSetFormula(h)) .
eq historyToSetFormula(s)     = s .
eq historyToSetFormula(noH)   = nil .

op promisesAreFulfilled(_) : SetFormula -> Bool .
eq promisesAreFulfilled(s, false) = false .
eq promisesAreFulfilled(s)        = (s contains promises(s)) [owise] .

op promises(_) : SetFormula -> SetFormula .
eq promises(s, (f U g)) = (promises(s), g) .
eq promises(s, (F g))   = (promises(s), g)    [owise] .
eq promises(s, ^ (G g)) = (promises(s), ^ g)  [owise] .
eq promises(s, f)       = promises(s)         [owise] .
eq promises(nil)        = nil                         .

op _contains_ : SetFormula SetFormula -> Bool .
eq s contains t = ((s, ^ false) containsInternal simplify(t)) .

op _containsInternal_ : SetFormula SetFormula -> Bool .
eq (s, t) containsInternal t = true .
eq s containsInternal t = false [owise] .

op simplify(_) : SetFormula -> SetFormula .
eq simplify(s, f, f) = simplify(s, f) .
eq simplify(s) = s [owise] .

*** Convierte un SetFormula en su expr equivalente (, => /\)
op toFormula_ : SetFormula -> Formula .
eq toFormula(nil)    = true .
eq toFormula(nes, f) = toFormula(nes) /\ f .
eq toFormula(f)      = f .


*** SIMPLIFICACION
***
*** limpieza historico
eq SF[   false  < s < h] D(d) = SF[   false  < noH] D(noD) .
eq SF[(^ false) < s < h] D(d) = SF[(^ false) < noH] D(noD) .

*** reduccion arbol
vars sn : SetNodos .
eq SF[   false  < h] D(d) ; sn = sn .
eq SF[(^ false) < h] D(d) ; sn = SF[(^ false) < h] D(d) .

endfm


mod TEMP-LOGIC-RULES is
protecting TEMPLOGIC .

vars h : History .
vars d : Formula .

rl [transition0] : T0[h] D(d) => S0[h] D(d) .
rl [transition1] : T1[h] D(d) => S1[h] D(d) .
```

```
endm
```

Pruebas unitarias

En la sección siguiente se exponen las pruebas unitarias que se han realizado para la comprobación del código.

Para ello se han comentado las líneas que simplifican el historial, de forma que se pueda ver el proceso de reescritura, aunque en el caso de bifurcación de ramas solo se visualiza una rama, debido a la simplificación del árbol.

En algunos casos la fórmula distinguida no es la esperada ya que la simplificación del historial es la que se ocupa de reemplazarla por noD en el caso del nodo final.

```
rew toFormula(nil)      . *** (^ false) .
rew toFormula('p)       . *** ('p) .
rew toFormula('p, 'q)   . *** ('p /\ 'q) .
rew toFormula('p U 'q) . *** ('p U 'q) .
rew toFormula(nil, 'p) . *** ('p) .
rew S0[X(toFormula(('a U 'b), 'p, 'q)) < noH] D(noD) .
*** SF[^ false < ('b,'p,'q) < ('p,'q,('a U 'b)) < ('p,'q /\ ('a U 'b))
***   < ('p /\ 'q /\ ('a U 'b)) < X ('p /\ 'q /\ ('a U 'b)) < noH]D(noD) .
```

```
rew simplify(     'p             ) . *** 'p .
rew simplify(     'p, 'p         ) . *** 'p .
rew simplify(     'p, 'q         ) . *** ('p, 'q) .
rew simplify(     'p, 'p, 'q     ) . *** ('p, 'q) .
rew simplify(     'p, 'p, 'q, 'q ) . *** ('p, 'q) .
rew simplify(nil, 'p             ) . *** 'p .
rew simplify(nil, 'p, 'p         ) . *** 'p .
rew simplify(nil, 'p, 'q         ) . *** ('p, 'q) .
rew simplify(nil, 'p, 'p, 'q     ) . *** ('p, 'q) .
rew simplify(nil, 'p, 'p, 'q, 'q ) . *** ('p, 'q) .
rew simplify(nil                 ) . *** nil .
```

```
rew ((nil        )  contains  (nil             )) . *** ^ false .
rew ((     'p    )  contains  (nil             )) . *** ^ false .
```

```
rew ((     'p, 'q)  contains  (nil             )) . *** ^ false .
rew ((nil, 'p, 'q)  contains  (nil             )) . *** ^ false .
rew ((nil        )  contains  (     'p         )) . *** false   .
rew ((     'p    )  contains  (     'p         )) . *** ^ false .
rew ((     'p, 'q)  contains  (     'p         )) . *** ^ false .
rew ((nil, 'p, 'q)  contains  (     'p         )) . *** ^ false .
rew ((nil        )  contains  (nil, 'p         )) . *** false   .
rew ((     'p    )  contains  (nil, 'p         )) . *** ^ false .
rew ((     'p, 'q)  contains  (nil, 'p         )) . *** ^ false .
rew ((nil, 'p, 'q)  contains  (nil, 'p         )) . *** ^ false .
rew ((nil        )  contains  (nil, 'p, 'q     )) . *** false   .
rew ((     'p    )  contains  (nil, 'p, 'q     )) . *** false   .
rew ((     'p, 'q)  contains  (nil, 'p, 'q     )) . *** ^ false .
rew ((nil, 'p, 'q)  contains  (nil, 'p, 'q     )) . *** ^ false .
rew ((nil        )  contains  (     'p, 'q     )) . *** false   .
rew ((     'p    )  contains  (     'p, 'q     )) . *** false   .
rew ((     'p, 'q)  contains  (     'p, 'q     )) . *** ^ false .
rew ((nil, 'p, 'q)  contains  (     'p, 'q     )) . *** ^ false .
rew ((nil        )  contains  (nil       , true)) . *** ^ false .
rew ((     'p    )  contains  (nil       , true)) . *** ^ false .
rew ((     'p, 'q)  contains  (nil       , true)) . *** ^ false .
rew ((nil, 'p, 'q)  contains  (nil       , true)) . *** ^ false .
rew ((nil        )  contains  (     'p   , true)) . *** false   .
rew ((     'p    )  contains  (     'p   , true)) . *** ^ false .
rew ((     'p, 'q)  contains  (     'p   , true)) . *** ^ false .
rew ((nil, 'p, 'q)  contains  (     'p   , true)) . *** ^ false .
rew ((nil        )  contains  (nil, 'p   , true)) . *** false   .
rew ((     'p    )  contains  (nil, 'p   , true)) . *** ^ false .
rew ((     'p, 'q)  contains  (nil, 'p   , true)) . *** ^ false .
rew ((nil, 'p, 'q)  contains  (nil, 'p   , true)) . *** ^ false .
rew ((nil        )  contains  (nil, 'p, 'q, true)) . *** false   .
rew ((     'p    )  contains  (nil, 'p, 'q, true)) . *** false   .
rew ((     'p, 'q)  contains  (nil, 'p, 'q, true)) . *** ^ false .
rew ((nil, 'p, 'q)  contains  (nil, 'p, 'q, true)) . *** ^ false .
rew ((nil        )  contains  (     'p, 'q, true)) . *** false   .
```

```
rew ((     'p    )  contains  (     'p, 'q, true)) . *** false   .
rew ((     'p, 'q)  contains  (     'p, 'q, true)) . *** ^ false .
rew ((nil, 'p, 'q)  contains  (     'p, 'q, true)) . *** ^ false .
rew ((nil        )  contains  (             true)) . *** ^ false .
rew ((     'p    )  contains  (             true)) . *** ^ false .
rew ((     'p, 'q)  contains  (             true)) . *** ^ false .
rew ((nil, 'p, 'q)  contains  (             true)) . *** ^ false .
```

```
rew promises(nil                              ) . *** (nil) .
rew promises(nil, 'p                          ) . *** (nil) .
rew promises(nil, 'p, 'q                      ) . *** (nil) .
rew promises(nil, ('p U 'q  ), 'r             ) . *** ('q) .
rew promises(nil, ('p U true), 'r             ) . *** (^ false) .
rew promises(nil, ('p U 'q  ), 'r             ) . *** ('q) .
rew promises(nil, ('p U true), 'r             ) . *** (^ false) .
rew promises(nil, ('p U 'q  ), ('r U 's)      ) . *** ('q, 's) .
rew promises(nil, ('p U true), ('r U 's)      ) . *** (^ false, 's) .
rew promises(nil, ('p U 'q  ), ('r U 's)      ) . *** ('q, 's) .
rew promises(nil, ('p U true), ('r U 's)      ) . *** (^ false, 's) .
rew promises(nil, ('p U 'q  ), ('r U 's), 't) . *** ('q, 's) .
rew promises(nil, ('p U true), ('r U 's), 't) . *** (^ false, 's) .
rew promises(nil, ('p U 'q  ), ('r U 's), 't) . *** ('q, 's) .
rew promises(nil, ('p U true), ('r U 's), 't) . *** (^ false, 's) .
rew promises(     'p                          ) . *** (nil) .
rew promises(     'p, 'q                      ) . *** (nil) .
rew promises(     ('p U 'q  ), 'r             ) . *** ('q) .
rew promises(     ('p U true), 'r             ) . *** (^ false) .
rew promises(     ('p U 'q  ), 'r             ) . *** ('q) .
rew promises(     ('p U true), 'r             ) . *** (^ false) .
rew promises(     ('p U 'q  ), ('r U 's)      ) . *** ('q, 's) .
rew promises(     ('p U true), ('r U 's)      ) . *** (^ false, 's) .
rew promises(     ('p U 'q  ), ('r U 's)      ) . *** ('q, 's) .
rew promises(     ('p U true), ('r U 's)      ) . *** (^ false, 's) .
rew promises(     ('p U 'q  ), ('r U 's), 't) . *** ('q, 's) .
```

```
rew promises(     ('p U true), ('r U 's), 't) . *** (^ false, 's) .
rew promises(     ('p U 'q  ), ('r U 's), 't) . *** ('q, 's) .
rew promises(     ('p U true), ('r U 's), 't) . *** (^ false, 's) .
rew promises(     (^ G 'p )) .                  *** (^ 'p) .
rew promises(     (^ G 'p), ('r U 's)) .        *** (^ 'p, 's) .
rew promises(     (F 'p )) .                    *** ('p) .
rew promises(nil, (^ G 'p )) .                  *** (^ 'p) .
rew promises(nil, (^ G 'p ), (F 'q ), ('r U 's)) . *** (^ 'p, 'q, 's) .
```

```
rew promisesAreFulfilled(nil)           . *** ^ false .
rew promisesAreFulfilled('p)            . *** ^ false .
rew promisesAreFulfilled(nil, 'p)       . *** ^ false .
rew promisesAreFulfilled('p U true)     . *** ^ false .
rew promisesAreFulfilled('p U 'q)       . *** false   .
rew promisesAreFulfilled(('p U 'q), 'q) . *** ^ false .
rew promisesAreFulfilled(^ G 'q, 'q)    . *** false .
rew promisesAreFulfilled(F 'q, 'q)      . *** false .
```

```
rew historyToSetFormula(noH)                  . *** (nil) .
rew historyToSetFormula('p < noH)             . *** ('p) .
rew historyToSetFormula('q < 'p < noH)        . *** ('p, 'q) .
rew historyToSetFormula('q < ('p U 'q) < noH)  . *** ('q, ('p U 'q)) .
```

```
rew loopIsTrue(noH)                             . *** ^ false .
rew loopIsTrue('p < noH)                        . *** ^ false .
rew loopIsTrue(('p  U 'q) < 'p < noH)           . *** false   .
rew loopIsTrue(('p  U 'q) < 'q < noH)           . *** ^ false .
rew loopIsTrue((('p  U 'q), 'q) < 'p < noH)     . *** ^ false .
rew loopIsTrue((('p  U 'q), 'q) < 'q < noH)     . *** ^ false .
rew loopIsTrue(('p  U 'q) < ('p U 'q) < 'q < noH) . *** ^ false .
rew loopIsTrue(('p  U 'q) < ('p U 'q) < 'p < noH) . *** false   .
```

```
*** TESTS ECUACIONES FALSE
```


```
rew S0[(^ false) < noH] D('e) .
*** SF[(^ false) < noH] D(noD)
rew S0[('a, ^ false) < noH] D('e) .
*** SF[^ false < 'a < ('a,^ false) < noH]D('e)
rew S0[(false) < noH] D('e) .
*** SF[false < false < noH]D(noD)
rew S0[('a, false) < noH] D('e) .
*** SF[false < (false,'a) < noH]D(noD)
```

```
*** TESTS contradicción
rew S0[('a, ^ 'a) < h] D('e) .
*** SF[false < ('a,^ 'a) < h]D(noD)
rew S0[('b, 'a, ^ 'a) < h] D('e) .
*** SF[false < ('a,'b,^ 'a) < h]D(noD)
rew S0[('b, ^ 'a, ^ ^ 'a) < h] D('e) .
*** SF[false < ('b,^ 'a,^ ^ 'a) < h]D(noD)
rew S0[('c, ('a /\ 'b), ^ ('a /\ 'b)) < h] D('e) .
*** SF[false < ('c,^ ('a /\ 'b),'a /\ 'b) < h]D(noD)
```

```
*** TESTS OPERADORES (Doble negación)
rew S0[(^ ^ 'a) < h] D('e) .
*** SF[^ false < 'a < ^ ^ 'a < h]D('e)
rew S0[('b, ^ ^ 'a) < h] D('e) .
*** SF[^ false < ('a,'b) < ('b,^ ^ 'a) < h]D('e)
rew S0[('c, ^ ^ ('a /\ 'b)) < h] D('e) .
*** SF[^ false < ('a,'b,'c) < ('c,'a /\ 'b) < ('c,^ ^ ('a /\ 'b)) < h]D('e)
```

```
*** TESTS OPERADORES (And)
rew S0[('a /\ 'b) < h] D('e) .
*** SF[^ false < ('a,'b) < ('a /\ 'b) < h]D('e)
rew S0[('b /\ (^ ^ 'a)) < h] D('e) .
*** SF[^ false < ('a,'b) < ('b,^ ^ 'a) < ('b /\ ^ ^ 'a) < h]D('e)
rew S0[('c, ('a /\ 'b)) < h] D('e) .
```

```
*** SF[^ false < ('a,'b,'c) < ('c,'a /\ 'b) < h]D('e)
rew S0[('a /\ (^ 'a)) < h] D('e) .
*** SF[false < ('a,^ 'a) < ('a /\ ^ 'a) < h]D(noD)
```

```
*** TESTS OPERADORES (not And)
rew S0[^('a /\ 'b) < h] D('e) .
*** SF[^ false < ^ 'a < ^ ('a /\ 'b) < h]D('e)
rew S0[^(^ 'a /\ ('b /\ 'a)) < h] D('e) .
*** SF[^ false < ^ 'a < ^ ('a /\ 'b /\ ^ 'a) < h]D('e)
```

```
*** TESTS negación de eventualidades
rew S0[^('a U 'b) < h] D('e) .
*** SF[^ false < (^ 'a,^ 'b) < ^ ('a U 'b) < h]D('e)
rew S0[(^('a U 'b), ('a U 'b)) < h] D('e) .
*** SF[false < (^ ('a U 'b),('a U 'b)) < h]D(noD)
rew S0[('c, ^('a U 'b)) < h] D('e) .
*** SF[^ false < ('c,^ 'a,^ 'b) < ('c,^ ('a U 'b)) < h]D('e)
rew S0[^ F 'a < h] D('e) .
*** SF[^ false < ^ F 'a < (^ 'a,^ X F 'a) < ^ F 'a < h]D(noD)
rew S0[ G 'a < h] D('e) .
*** SF[^ false < G 'a < ('a,X G 'a) < G 'a < h]D(noD)
```

```
*** TESTS eventualidades
rew S0[('a, 'b, ('e U 'f)) < h] D('e U 'f) .
*** SF[^ false < ('a,'b,'f) < ('a,'b,('e U 'f)) < h]D(noD)
rew S0[(('e U 'f), ('e U 'f)) < h] D('e U 'f) .
*** SF[^ false < ('f,'f) < ('f,('e U 'f)) < (('e U 'f),('e U 'f)) < h]D(noD)
rew S0[('e U 'f) < noH] D(noD) .
*** SF[^ false < 'f < ('e U 'f) < noH]D(noD)
rew S0[(^ G 'e) < noH] D(noD) .
*** SF[^ false < ^ 'e < G 'e < noH]D(noD)
rew S0[(F 'e) < noH] D(noD) .
*** SF[^ false < 'e < F 'e < noH]D(noD)
```

```
rew S0[(^ G 'e) < noH] D(^ G 'e) .
*** SF[^ false < ^ 'e < G 'e < noH]D(noD)
rew S0[(F 'e) < noH] D(F 'e) .
*** SF[^ false < 'e < F 'e < noH]D(noD)
rew S0[(F 'e, ^ G 'f) < noH] D(F 'e) .
*** SF[^ false < ('e,^ 'f) < ('e,G 'f) < (^ G 'f,F 'e) < noH]D(noD)
```

```
*** Test Next
rew S0[(X('e)) < 'h] D('g) .
*** SF[^ false < 'e < X 'e < 'h]D(noD)
rew S0[(X('e)) < 'h] D(noD) .
*** SF[^ false < 'e < X 'e < 'h]D(noD)
rew S0[('f, ^ X('e)) < 'h] D('g) .
*** SF[^ false < ^ 'e < ('f,^ X 'e) < 'h]D(noD)
rew S0[(X('e), X('f)) < 'h] D('g) .
*** SF[^ false < ('e,'f) < (X 'e,X 'f) < 'h]D(noD)
rew S0[(X('e), X('f)) < 'h] D(X('e)) .
*** SF[^ false < ('e,'f) < (X 'e,X 'f) < 'h]D('e)
rew S0[(^ X('e), X('f), 'a) < 'h] D(X('e)) .
*** SF[^ false < ('f,^ 'e) < ('a,^ X 'e,X 'f) < 'h]D('e)
rew S0[(X('e U 'f)) < 'h] D('g) .
*** SF[^ false < 'f < ('e U 'f) < X ('e U 'f) < 'h]D(noD)
rew S0[('a, X('e U 'f)) < 'h] D('g) .
*** SF[^ false < 'f < ('e U 'f) < ('a,X ('e U 'f)) < 'h]D(noD)
rew S0[('a, 'b, X('e U 'f)) < 'h] D('g) .
*** SF[^ false < 'f < ('e U 'f) < ('a,'b,X ('e U 'f)) < 'h]D(noD)
rew S0[('a, 'b, X('e U 'f), X(false)) < 'h] D('g) .
*** SF[false < (false,('e U 'f)) < ('a,'b,X false,X ('e U 'f)) < 'h]D(noD)
rew S0[('a, 'b, X('e U 'f), ^ X('a)) < 'h] D('g) .
*** SF[^ false < ('f,^ 'a) < (^ 'a,('e U 'f))
***             < ('a,'b,^ X 'a,X ('e U 'f)) < 'h]D(noD)
rew S0[(X('e U 'f)) < 'h] D(X('e U 'f)) .
*** SF[^ false < 'f < ('e U 'f) < X ('e U 'f) < 'h]D(noD)
```

Bibliografía

1. *Systematic Semantic Tableaux for PLTL.* **Gaintzarain, J., et al.** 2008, Electronic Notes in Theorical Computer Science, pp. 59-73.

2. **Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., et al.** Maude Manual. 2008.

3. *Dual Systems of Tableaux and Sequents for PLTL.* **Gaintzarain, J., et al.** 2009, The Journal of Logic and Algebraic Programming, pp. 701-722.

4. TRS: A Resolution-based Theorem Prover for Propositional Linear Temporal Logic. [En línea] http://www.sc.ehu.es/jiwnagom/paginaMarisa_archivos/TRS.html.

5. **Lucio, Paqui.** TTM: A Tableau-based Theorem Prover for Temporal Logic PLTL. [En línea] http://www.sc.ehu.es/jiwlucap/TTM.html.

6. **Villanueva, Alicia.** Lógica aplicada a la verificación del software y optimización. mayo de 2005.