

Manual de introducción a Maude

Adrián Riesco

Auditoría, Calidad y Fiabilidad Informáticas

Curso 2019/20

Índice

1. Introducción	1
2. Lógica Lineal Temporal	2
2.1. Operadores lógicos	2
2.2. Operadores temporales	2
2.3. Ejemplos	3
3. Maude	3
4. Cómo programar en Maude	4
4.1. Módulos funcionales	4
4.1.1. Números naturales	4
4.1.2. Pilas	7
4.1.3. Listas	8
4.2. Módulos de sistema	12
4.3. Más ejemplos	15
4.4. Verificación de propiedades en Maude	17
4.4.1. Comprobando invariantes mediante búsqueda	17
4.4.2. Comprobación de modelos	18
4.4.3. Abstracción ecuacional	19
4.5. Usando Maude	20
4.6. Errores frecuentes	21
4.7. Módulos predefinidos	22

1. Introducción

Los métodos formales en la Ingeniería Informática se definen sobre cuatro pilares básicos: (1) la existencia de un modelo semántico claro y conciso sobre el comportamiento de los sistemas software; (2) la existencia de una representación clara, detallada y sin ambigüedades que permita expresar sistemas software y asociarles una semántica concreta; (3) la existencia de un formalismo claro y detallado en el que se puedan definir propiedades de un sistema y en el que se pueda averiguar la validez o falsedad de dichas propiedades en función del modelo semántico; y (4) la existencia de técnicas eficientes y eficaces para verificar dichas propiedades. Hasta hoy, los métodos formales se han aplicado, por un lado, como parte del proceso de desarrollo de software, generando software fiable y seguro simplemente por construcción, gracias a las buenas cualidades de una representación y una semántica claras y concisas, pero también como técnicas, en este caso aplicadas a posteriori, de especificación y verificación de propiedades de los sistemas software ya creados

(con o sin garantías de corrección), gracias a las buenas cualidades de un formalismo claro para especificar propiedades y de técnicas efectivas para comprobarlas. En el primer caso, los lenguajes de programación (o especificación) denominados declarativos (entre los que se encuentran los lenguajes algebraicos Maude y OBJ, los lenguajes funcionales Haskell y ML, los lenguajes lógicos Prolog y Gödel, o los lenguajes lógico-funcionales Curry y TOY) son muy adecuados para ser utilizados durante el proceso de producción de software, especialmente en la definición de especificaciones prototípicas ejecutables, donde compiten con ventaja con otro tipo de especificaciones más farragosas como UML presentadas en lenguaje natural o semi-formal que no pueden ser completamente automatizadas. En el segundo caso, técnicas formales de especificación y verificación de propiedades basadas en diferentes formalismos, como la alcanzabilidad por reescritura o la comprobación de modelos o *model checking*, han obtenido un gran éxito y una notable aceptación en el ámbito académico pero también en el comercial o industrial.

Este segundo punto es justamente lo que ocurre con Maude, un lenguaje que ha sido utilizado, no como alternativa a lenguajes y técnicas de especificación como UML, sino para especificar y razonar “a posteriori” sobre especificaciones de sistemas realizadas usando otros lenguajes y técnicas de especificación. Por ejemplo, Maude se ha utilizado para razonar sobre protocolos de comunicaciones como el FireWire (conocido como IEEE 1394), las plataformas CORBA y SOAP, el metamodelo de UML, nuevos lenguajes de programación y lenguajes conocidos, como Java, e incluso se utiliza por la NASA para el desarrollo de sistemas de reconocimiento de objetos en el espacio.

2. Lógica Lineal Temporal

La *Lógica Lineal Temporal* (LTL por sus siglas en inglés) es una lógica temporal utilizada para expresar propiedades temporales de un sistema reactivo o concurrente dentro del contexto del model checking. Utiliza proposiciones elementales, operadores lógicos y operadores temporales (de estado o de caminos).

2.1. Operadores lógicos

Los operadores lógicos son los usuales en lógica: ! (negación), | (or), & (and), \rightarrow (implicación), y \leftrightarrow (equivalencia). También se pueden usar los valores lógicos **true** and **false** y los operadores lógicos de comparación: = (igualdad), \sim (distinto), <, >, >=, <=, así como los operadores aritméticos básicos: +, -, *, /, *mod*.

2.2. Operadores temporales

Toda fórmula LTL se entiende como una fórmula $A\ p$ donde p es una fórmula de camino que debe satisfacerse en todos los caminos que surjan del estado actual y A como un modificador temporal. Fijando un estado actual y un camino a partir de él, determinamos la satisfactibilidad de una fórmula de camino p con respecto a los estados siguientes al actual dentro del camino fijado.

- Fórmula $\bigcirc p$ (Next): la proposición p debe satisfacerse en el estado siguiente al actual.
- Fórmula $\Box p$ (Globally): la proposición p debe satisfacerse en todos los estados posteriores al estado actual.

- Fórmula $\Diamond p$ (Eventually): la proposición p debe satisfacerse en al menos un estado posterior de todos los caminos que comienzan en el estado actual.
- Fórmula $p U p'$ (Until): la proposición p debe satisfacerse en todos aquellos estados contiguos y posteriores al estado actual mientras la proposición p' no se satisfaga.

2.3. Ejemplos

Si suponemos que la proposición p describe “llueve” y que la proposición q describe “vamos a jugar al tenis”, tenemos las siguientes ejemplos de fórmulas LTL:

1. $\Box p$ indica que siempre llueve.
2. $\Diamond p$ indica que siempre llegará un momento en que llueva.
3. $\Diamond \Box p$ indica que siempre llegará un momento en que llueva y no pare de llover.
4. $\Box \Diamond (\neg p \wedge q)$ indica que siempre llegará un momento en que no llueva y juegue al tenis y esto ocurrirá un número infinito de veces.
5. $\bigcirc p$ indica que va a llover en el siguiente instante de tiempo, pase lo que pase.
6. $q U p$ indica que juego al tenis hasta que empieza a llover.

3. Maude

El lenguaje de programación Maude [2, 1] utiliza reglas de reescritura como los lenguajes denominados funcionales tales como Haskell [7], ML [6], Scheme [3] o Lisp [8]. En concreto, Maude está basado en la lógica de reescritura que permite definir multitud de modelos computacionales complejos tales como programación concurrente o programación orientada a objetos. Por ejemplo, Maude permite especificar objetos directamente en el lenguaje, siguiendo una aproximación declarativa a la programación orientada a objetos que no está disponible ni en lenguajes imperativos como C++ o Java ni en lenguajes declarativos como Haskell.

El desarrollo del lenguaje Maude parte de una iniciativa internacional cuyo objetivo consiste en diseñar una plataforma común para la investigación, docencia y aplicación de los lenguajes declarativos. Se puede encontrar más información en:

<http://maude.cs.uiuc.edu>

A continuación resumimos las principales características de Maude. Sin embargo, hay un extenso manual [1] y un “primer” [5] (libro de introducción muy sencillo y basado en ejemplos) en la dirección web indicada antes. Existe también un libro [2] sobre Maude, con ejemplares adquiridos por la Biblioteca y accesible online (solo desde dentro de la Complutense) en:

<http://www.springerlink.com/content/p6h32301712p>

4. Cómo programar en Maude

En esta sección introducimos la sintaxis de Maude mediante ejemplos sencillos, empezando por los módulos más básicos, los módulos funcionales, y pasando después a los módulos de sistema y los módulos orientados a objetos. Para cada uno de estos módulos se muestran brevemente los posibles comandos disponibles.

Una vez descritos los distintos módulos pasaremos a explicar cómo verificar propiedades en Maude. Empezaremos describiendo cómo comprobar invariantes en Maude para después describir cómo usar el comprobador de modelos.

Por último, explicaremos brevemente el intérprete de consola para Maude, los errores más frecuentes y alguno de los módulos predefinidos más usados.

4.1. Módulos funcionales

En esta sección presentamos los módulos funcionales usando varios ejemplos: primero veremos cómo especificar los números naturales usando la notación de Peano, después mostraremos cómo especificar pilas y, por último, mostraremos cómo trabajar con listas.

Es importante tener en cuenta que en esta sección hablaremos de ecuaciones, y dichas ecuaciones tienen 2 importantes restricciones:¹ deben ser terminantes, es decir, no pueden entrar en cálculos infinitos; y deben ser confluentes, es decir, deben devolver el mismo resultado independientemente del orden en el que se apliquen.

4.1.1. Números naturales

Nuestro primer ejemplo en Maude, cuyo código está disponible en el Campus Virtual con el nombre `peano_nat.maude`, son los números naturales usando la notación de Peano.² Para ello, vamos a crear un módulo funcional llamado `PEANO` usando la palabra reservada `fmod`, que indica que empieza el módulo (más tarde veremos que usamos `endfm` para cerrarlo), seguida del nombre del módulo, en este caso `PEANO`,³ y por último la palabra reservada `is`:

```
fmod PEANO is
```

Lo primero que necesitamos en este módulo es definir los tipos de datos. En este ejemplo tan sencillo solo vamos a definir el tipo `PeanoNat`,⁴ así que usamos la palabra reservada `sort` para indicar que vamos a definir un tipo, seguido del nombre del tipo, `PeanoNat`, y seguido por un punto.

☹☹ Aunque este punto, como veremos en las próximas secciones, es muy utilizado en Maude, es uno de los errores de compilación más frecuentes, así que préstale especial atención cuando programes.

```
sort PeanoNat .
```

A continuación tenemos que definir las constructoras de los tipos definidos anteriormente. En el caso de los números naturales definidos con la notación de Peano los constructores

¹De hecho hay más restricciones, pero no las estudiaremos en el curso.

²Fíjate que, como se explica en la sección 4.7, en Maude existe un módulo predefinido para los números naturales, por lo que este módulo simplemente sirve de introducción a Maude, pero no será necesario en el futuro.

³Por convenio se usan mayúsculas para identificar los módulos, aunque no es necesario.

⁴Por convenio los tipos se suelen escribir con la primera letra mayúscula y el resto en minúscula, aunque no es necesario.

son, como todos sabemos, el 0 y el sucesor. El 0 podemos definirlo con la siguiente notación, que explicamos a continuación:

```
op 0 : -> PeanoNat [ctor] .
```

Veamos qué estamos indicando en esta línea: para empezar, tenemos la palabra reservada **op**, que indica que estamos definiendo un *operador*. Después de esta palabra reservada tenemos el símbolo que queremos usar para nuestro constructor; en este caso usamos el propio número 0, aunque podríamos haber decidido escribir **cero**, **miNumeroCero** o cualquier otra variante. Tras el identificador encontramos dos puntos (:), que indican que he terminado con el nombre del identificador y voy a describir qué argumentos recibe y qué tipo tiene. Como el constructor no espera ningún argumento no necesitamos escribir nada y llegamos directamente a la flecha (->). A la derecha de esta flecha tenemos que escribir el tipo de nuestro constructor, que en este caso es **PeanoNat**. Tras la definición del tipo podemos incluir una serie de atributos entre corchetes ([]); en este caso hemos usado **ctor**, que indica que el operador es un constructor. Por último, el operador termina con un punto. El operador sucesor es ligeramente más complejo:

```
op s : PeanoNat -> PeanoNat [ctor] .
```

En este caso vemos que tenemos el tipo **PeanoNat** después de :. Esto significa que este operador recibe un argumento de tipo **PeanoNat**, es decir, que podemos crear el término **s(0)** (que identifica, como sabemos, al número 1), **s(s(0))** (el 2 en notación usual), etc. Por lo demás, el constructor es idéntico al mostrado arriba.

Una vez tenemos definidos los constructores de los tipos de datos podemos definir funciones entre ellos, aunque es interesante tener antes definidas las variables que usaremos después. En este caso basta con definir 2 variables de tipo **PeanoNat**:

```
vars N M : PeanoNat .
```

Como se ve en este ejemplo, definimos variables usando la palabra reservada **vars** (**var** es también correcto, incluso para varias variables) seguido de una lista de nombres de variables separadas sin comas, seguido de :, el nombre del tipo y finalizado por un punto. La función más sencilla que podemos definir entre números naturales es la suma:

```
op _+_ : PeanoNat PeanoNat -> PeanoNat [assoc comm] .
```

En este caso observamos que el identificador del constructor tiene dos subrayados (_). Este elemento no es realmente parte del nombre del identificador, sino que indica que en esa posición se pueden introducir los argumentos. Como esta función recibe dos argumentos de tipo **PeanoNat** (por ello aparece 2 veces el tipo entre : y ->) podemos escribir la suma como, por ejemplo, **s(0) + s(s(0))**. Es decir, podemos colocar los argumentos en el lugar de los subrayados. Finalmente, vemos que esta función no es una constructora, por lo que no tiene el atributo **ctor**, pero sí tiene otros 2 atributos:

- El atributo **assoc** indica que la función es asociativa, es decir, que $(N + M) + P = N + (M + P)$, y por tanto no es necesario escribir paréntesis para indicar el orden de evaluación.
- El atributo **comm** indica que la función es conmutativa, es decir, que $N + M = M + N$. Este atributo simplifica la definición del comportamiento de las funciones, pues solo es necesario describir uno de los casos. Más adelante veremos ejemplos más complejos.

El comportamiento de las funciones se define mediante *ecuaciones*. Es habitual definir las ecuaciones mediante inducción estructural en uno de los argumentos, definiendo una ecuación por cada constructor. En este caso podemos elegir hacer inducción en el primer parámetro de la suma (dado que la suma es conmutativa, elegir el segundo parámetro no cambiaría la mecánica) y por tanto necesitamos 2 ecuaciones, la primera se definirá para el constructor 0 y la otra para el constructor sucesor aplicado a una variable ($s(N)$):

```
*** Caso 1: constructor 0
eq [s1] : 0 + N = N .

*** Caso 2: constructor sucesor
eq [s2] : s(N) + M = s(N + M) .
```

Hemos usado comentarios para indicar a qué caso se refiere cada ecuación, aunque es sencillo identificarlas sin ellos.⁵ Además, hemos usado etiquetas para identificar cada ecuación. Dichas etiquetas se colocan entre corchetes y seguidas de `:` y son opcionales; el módulo funcionaría de la misma manera sin ellas. Por lo demás, observamos que las ecuaciones se definen usando la palabra reservada `eq` y usando un igual (`=`) entre los 2 términos que queremos identificar, para finalmente acabar con un punto. En la ecuación `s1` hemos indicado que sumar 0 a cualquier número natural, identificado por la variable `N`, devuelve el mismo natural. En la ecuación `s2` indicamos que, si el primer argumento está definido con el constructor sucesor, entonces podemos sumar el número al que aplicamos este constructor (que será, obviamente, un número menor) con el otro argumento y aplicar sucesor al resultado. De esta manera el argumento irá haciéndose más pequeño hasta llegar al caso base en el que se aplica la ecuación `s1`.

De manera similar podemos definir la multiplicación:

```
op *_ : PeanoNat PeanoNat -> PeanoNat [assoc comm] .
eq 0 * N = 0 .
eq s(N) * M = M + (N * M) .
```

Maude incluye por defecto en todos los módulos el módulo predefinido `BOOL`, que permite usar el tipo `Bool` para valores Booleanos (más información está disponible en la sección 4.7). Así pues, podemos definir una función Booleana que, dado un número definido en notación de Peano, indique si el número es positivo o no. El operador se define como:


```
op esPositivo : PeanoNat -> Bool .
```

Las ecuaciones para esta función distinguen entre las 2 constructoras y usan los valores `true` y `false` como resultado:

```
eq esPositivo(0) = false .
eq esPositivo(s(N)) = true .
```

Por último, el módulo se cierra con la palabra reservada `endfm`:

```
endfm
```

Las funciones definidas en módulos funcionales se ejecutan (para ver cómo iniciar Maude y cargar los módulos visita la sección 4.5) con el comando `reduce`, abreviado `red`. Por tanto, podemos probar a sumar 1 y 2 (lo que en nuestra notación se escribe como $s(0) + s(s(0))$) como sigue ( El comando también finaliza con un punto):

⁵Los comentarios de bloque se escriben en Maude con sintaxis `***(...)`.

```
Maude> red s(0) + (s(s(0))) .
reduce in Peano : s(0) + s(s(0)) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result PeanoNat: s(s(s(0)))
```

En estos primeros ejemplos es interesante ver cómo funciona Maude. En este caso tenemos la siguiente derivación, en la que mostramos para cada paso qué ecuación se está usando y en qué parte del término:

$$\underline{s(0) + s(s(0))} \xrightarrow{s^2} \underline{s(0 + s(s(0)))} \xrightarrow{s^1} s(s(s(0)))$$

También podemos probar a ejecutar la función `esPositivo` como sigue:

```
Maude> red esPositivo(s(s(0))) .
reduce in Peano : esPositivo(s(s(0))) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

4.1.2. Pilas

Ahora que dominamos la notación básica de Maude podemos tratar de crear pilas de números naturales (el código de esta sección está disponible en el fichero `pila.maude`). Para ello, empezamos creando el módulo funcional `PILA`:

```
fmod PILA is
```

Lo siguiente que necesitamos son los números naturales. Aunque hemos definido unos números naturales básicos en la sección anterior es conveniente usar, a partir de ahora, los números naturales predefinidos en Maude. El módulo `NAT`, como se explica en la sección 4.7, define el tipo `Nat` para trabajar con números naturales, así que lo importamos con el comando `pr`:⁶

```
pr NAT .
```

Acto seguido, podemos definir el tipo `Pila`, que tiene como constructoras la `pila-vacia` y el operador `apila`, y definir variables para los naturales y para las pilas:

```
sort Pila .

op pila-vacia : -> Pila [ctor] .
op apila : Nat Pila -> Pila [ctor] .

var N : Nat .
var P : Pila .
```

Lo más interesante de estas definiciones lo encontramos en el operador `apila`, que recibe un número natural como primer argumento. Es decir, la pila que contiene solo un 7 se representa como `apila(7, pila-vacia)`, mientras que la pila que apila un 3 sobre un 7 se representa como `apila(3, apila(7, pila-vacia))`.

La función `desapila` elimina la cima de la pila. Si no queremos tratar con errores podemos decidir que la operación no modifique la `pila-vacia`:

⁶Maude permite importar módulos con distintas palabras reservadas, que imponen ciertas restricciones en los módulos importados. Sin embargo, en este curso solo usaremos el modo `protecting`, en general en su forma abreviada `pr`.

```

op desapila : Pila -> Pila .
eq desapila(pila-vacia) = pila-vacia .
eq desapila(apila(N, P)) = P .

```

Sin embargo, no es tan sencillo esquivar el problema de los errores al definir la función `cima`. Para solucionarlo podemos definir un operador parcial en lugar de uno total; Maude considerará que la función falla cuando no encuentre ecuaciones que pueda aplicar. Para definir un función parcial cambiamos la flecha (\rightarrow) en la definición del operador por una flecha de la forma \rightsquigarrow , como vemos en este ejemplo:

```

op cima : Pila  $\rightsquigarrow$  Nat .
eq cima(apila(N, P)) = N .
endfm

```

Una vez finalizado el módulo podemos hacer algunas pruebas:

```

Maude> red cima(desapila(apila(3, apila(7, pila-vacia)))) .
reduce in PILA-EXT : cima(desapila(apila(3, apila(7, pila-vacia)))) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: 7

```

En este caso la derivación es:

$$\text{cima}(\text{desapila}(\text{apila}(3, \text{apila}(7, \text{pila-vacia})))) \rightsquigarrow \text{cima}(\text{apila}(7, \text{pila-vacia})) \rightsquigarrow 7$$

4.1.3. Listas

El último módulo funcional que vamos a explicar, disponible en el Campus Virtual con el nombre `listas.maude`, muestra cómo especificar listas de personas. Para ello, empezamos definiendo las personas en el módulo `PERSONA`:

```

fmod PERSONA is
pr STRING .

sort Persona .

op <_,_> : String Nat -> Persona [ctor] .

vars S S' : String .
vars N N' : Nat .

```

Como podemos ver, usamos el módulo `STRING` para definir las personas como parejas con nombre, de tipo `String`, y el dinero que tienen, de tipo `Nat`. Definimos además las variables correspondientes, que usaremos a continuación. Vamos a definir ahora la funciones “máximo” y “mínimo” entre personas en función de su dinero. Fíjate que estas funciones son conmutativas y, por tanto, basta una ecuación condicional para definir cada una:

```

ops max min : Persona Persona -> Persona [comm] .

ceq [max] : max(< S, N >, < S', N' >) = < S, N >
if N >= N' .
ceq [min] : min(< S, N >, < S', N' >) = < S, N >
if N <= N' .

```


Como hemos adelantado, las ecuaciones `max` y `min` son condicionales. Para declarar ecuaciones condicionales debemos usar la palabra reservada `ceq` en lugar de `eq`. Tras definir la ecuación podemos escribir una lista de condiciones terminada en punto; cada una de estas condiciones se une mediante el operador `/\`.

☹☹ Maude tiene también un operador `if_then_else_fi` que no debe confundirse con una ecuación condicional. Si deseamos usar este operador para las funciones anteriores debemos definir una ecuación no condicional⁷ y usarlo a la derecha del igual. Es decir, tendríamos:

```
ops max min : Persona Persona -> Persona [comm] .

eq [max] : max(< S, N >, < S', N' >) = if N >= N'
                                     then < S, N >
                                     else < S', N' >
                                     fi .

eq [min] : min(< S, N >, < S', N' >) = N <= N'
                                     then < S, N >
                                     else < S', N' >
                                     fi .
```

Pretendemos definir listas ordenadas en el próximo módulo usando la cantidad de dinero como criterio de ordenación, así que es conveniente definir las funciones “menor o igual que” y “mayor que”:

```
ops _<=_ _>_ : Persona Persona -> Bool .
eq < S, N > <= < S', N' > = N <= N' .
eq < S, N > > < S', N' > = N > N' .
```

Por último, vamos a definir algunas constantes para facilitar los ejemplos:

```
ops a b c d : -> Persona .
eq a = < "a", 100 > .
eq b = < "b", 80 > .
eq c = < "c", 150 > .
eq d = < "d", 10 > .
endfm
```

El módulo `LISTA` importa el módulo `PERSONA` y define los tipos `Lista`, para listas, y `ListaOrd`, para listas ordenadas:

```
fmod LISTA is
pr PERSONA .

sorts Lista ListaOrd .
```

Vamos a introducir ahora la idea de *subtipo*. Los subtipos, definidos en Maude con la palabra `subsort` y el operador `<`, indican que todos los elementos definidos para el tipo menor tienen también el tipo mayor. En este caso estamos indicando que un elemento de tipo `Persona` es también una lista ordenada unitaria. A su vez, cualquier lista ordenada es, lógicamente, una lista (☹☹ los subtipos también terminan en punto):

```
subsort Persona < ListaOrd < Lista .
```

⁷Es posible, por supuesto, mezclar ecuaciones condicionales con la función `if`, pero para ello es mejor tener más práctica con el sistema.

Una vez definidos los tipos nos encontramos con un grave problema: sabemos que la lista vacía (que escribimos como `lv`) está ordenada pero, ¿qué constructor podemos usar para definir listas ordenadas no vacías? La única opción posible parece ser usar un constructor para las listas generales, y definir después las listas ordenadas describiendo sus propiedades:

```
op lv : -> ListaOrd [ctor] .
op __ : Lista Lista -> Lista [ctor assoc id: lv] .

vars P P' : Persona .
vars L L' : Lista .
var L0 : ListaOrd .
```

El operador para definir listas no vacías es interesante por una serie de razones:

- El operador recibe como argumento dos elementos de tipo `Lista`. Pero, ¿cómo son esos elementos de tipo lista? Como hemos visto arriba, al principio estas listas pueden ser simplemente personas, ya que una persona es un caso concreto de lista unitaria. Una vez tengamos varias personas iremos creando listas mayores. La manera concreta de crear estas listas se detalla a continuación.
- La notación `__` indica que puedes poner juntas 2 listas para crear una mayor sin usar ninguna notación. Es decir, si tenemos las personas `a` y `b` del módulo `PERSONA` podemos crear la lista `a b`. La idea es que colocamos `a` en la posición del primer subrayado, dejamos un espacio y situamos `b` en la posición del segundo subrayado. Si ahora quisiéramos añadir a la persona `c` tendríamos `a b c` (la lista `a b` corresponde ahora al primer subrayado y la lista `c` al segundo).
- Las listas tienen a la lista vacía, `lv`, como elemento identidad. Esto quiere decir que Maude interpreta que cualquier lista `L` es igual a la listas `L lv` y `lv L`. Esto es muy útil para definir las ecuaciones, como veremos a continuación.

Ahora debemos definir las propiedades que deben cumplirse para que una lista esté ordenada. Para ello usaremos *axiomas de pertenencia*, escritos en Maude como `mb`, los no condicionales, y como `cmb`, los condicionales. Después de la correspondiente palabra reservada debemos escribir un término seguido de `:` y el nombre del tipo que queremos indicar. En nuestro caso necesitamos un axioma condicional que nos indique, como primera condición, que el primer elemento de la lista es menor o igual que el segundo, mientras que la segunda condición pide que el resto de la lista esté también ordenada. Esta segunda condición es una condición de pertenencia, escrita con `:`, igual que el axioma:

```
cmb P P' L : ListaOrd
if P <= P' /\
  P' L : ListaOrd .
```

☹☹ ¿Por qué no hemos definido un axioma de pertenencia para indicar que la lista vacía está ordenada? Porque lo hemos indicado en la definición del operador. Una pregunta más complicada, ¿por qué no lo hemos usado para indicar que una lista con un solo elemento está ordenada? Porque lo hemos indicado con el subtipo.

Ahora podemos pasar a definir las funciones del módulo. La cabeza de la lista es una función parcial definida como sigue:

```
op head : Lista ~> Persona .
eq head(P L) = P .
```

En esta ecuación vemos la utilidad del elemento identidad. ¿Cómo funciona esta ecuación cuando solo tenemos una persona? Es sencillo: esa persona se liga a la variable P y es devuelta, mientras que la variable L se liga a la lista vacía. De esta manera la ecuación puede aplicarse siempre que la lista no esté vacía. La función para la cola de la lista se define de la misma manera:

```
op tail : Lista ~> Lista .
eq tail(P L) = L .
```

El tamaño de la lista se define así:

```
op tam : Lista -> Nat .
eq tam(lv) = 0 .
eq tam(P L) = s(tam(L)) .
```

Mientras que la función **esta?**, que comprueba si un elemento está en la lista, puede definirse de manera “astuta”. Basta con escribir variables a la derecha y a la izquierda del valor que estamos buscando para que “recojan” el resto de la lista, que no nos interesa. Si no podemos aplicar la primera ecuación es porque el elemento no está, así que podemos devolver **false** con seguridad. Para ello, usamos el valor especial **owise**, abreviatura de *otherwise*, es decir, *en otro caso*:

```
op esta? : Lista Persona -> Bool .
eq esta?(L P L', P) = true .
eq esta?(L, P) = false [owise] .
```

De manera similar podemos definir la ordenación por el método de la burbuja: si dos elementos adyacentes están desordenados los intercambiamos y seguimos ordenando. Cuando no podamos hacer esto es porque todos los elementos están ordenados, así que devolvemos la lista:

```
op ordena : Lista -> Lista .
ceq ordena(L P P' L') = ordena(L P' P L')
  if P > P' .
eq ordena(L) = L [owise] .
```

Por último definimos la inserción ordenada. Insertar en la lista vacía resulta en la lista unitaria formada por la persona que está siendo insertada. En otro caso, comprobamos la relación entre el primer elemento de la lista y el elemento a insertar y actuamos en consecuencia:

```
op inserta-ord : ListaOrd Persona -> ListaOrd .
eq inserta-ord(lv, P) = P .
eq inserta-ord(P L, P') = if P <= P'
  then P inserta-ord(L, P')
  else P' P L
  fi .
```

Podemos ahora definir la ordenación por inserción como sigue:

```
op ordena-por-insercion : ListaOrd -> ListaOrd .
eq ordena-por-insercion(lv) = lv .
ceq ordena-por-insercion(P L) = inserta-ord(L', P)
  if L' := ordena-por-insercion(L) .
endfm
```

Es interesante ver el uso de la condición de encaje de patrones, **:=**. Esta condición sirve básicamente como una asignación en un lenguaje imperativo, es decir, guardamos el valor de la evaluación en la variable L' y la usamos después para continuar con la inserción.

4.2. Módulos de sistema

Los módulos de sistema amplían los módulos funcionales con *reglas de reescritura*, que representan transiciones entre estados. Simplificando, para definir las estructuras de datos de nuestros sistemas usaremos ecuaciones, posiblemente en módulos funcionales, que luego serán importados por los módulos de sistema encargados de describir el comportamiento del sistema. La principal ventaja de las reglas es que, al contrario de las restricciones para ecuaciones que vimos en la sección 4.1, las reglas no necesitan ser terminantes ni confluentes.

Como ejemplo de módulo de sistema proponemos el clásico problema de las vasijas, que aparece habitualmente en problemas de ingenio o en situaciones más “críticas” como en la película La Jungla de Cristal 3[©]:

<http://www.wikihow.com/Solve-the-Water-Jug-Riddle-from-Die-Hard-3>

La idea es la siguiente: tenemos 3 vasijas no graduadas de capacidades 3, 5 y 8 litros, y además tenemos una fuente inagotable de agua. Queremos conseguir tener 4 litros de agua en alguna de las vasijas y lo único que podemos hacer con ellas es llenarlas, vaciarlas y pasar agua de una a otra. ¿Cómo conseguimos esa cantidad de agua?

Para resolver este problema (el código correspondiente está disponible en el Campus Virtual en el fichero `die-hard.maude`) definimos el módulo `DIE-HARD` como sigue:

```
mod DIE-HARD is
  protecting NAT .
  sorts Vasija ConjVasija .
  subsort Vasija < ConjVasija .

  op vasija : Nat Nat -> Vasija [ctor] .    *** Capacidad / Contenido actual
  op _ : ConjVasija ConjVasija -> ConjVasija [ctor assoc comm] .

  vars M1 N1 M2 N2 : Nat .
```

Observamos en primer lugar que los módulos de sistema empiezan con la palabra reservada `mod`, pero que por lo demás la notación es la misma que para módulos funcionales. En nuestro caso hemos definido las vasijas como un operador que recibe 2 números naturales, indicando el primero la capacidad total y el segundo la cantidad de agua actualmente. Además, tenemos el tipo `ConjVasija` para conjuntos de vasijas. Podemos definir una constante `initial` de tipo `ConjVasija` indicando que inicialmente todas las vasijas están vacías como sigue:

```
op initial : -> ConjVasija .
eq initial = vasija(3, 0) vasija(5, 0) vasija(8,0) .
```

Definiremos los cambios de estado de las vasijas mediante reglas de reescritura. Las reglas siguen una notación similar a las ecuaciones, cambiando la palabra reservada `eq` por `r1` (respectivamente `ceq` por `cr1` para las reglas condicionales) y el igual (=) por una flecha (\Rightarrow). Aunque no es necesario que estén etiquetadas, es muy habitual. En nuestro problema podemos empezar por definir la regla `vacía`, que elimina toda el agua de una vasija:

```
r1 [vacía] : vasija(M1, N1) => vasija(M1, 0) .
```

Usando la misma idea podemos definir una regla que rellene una vasija:

```
rl [llena] : vasija(M1, N1) => vasija(M1, M1) .
```

Tenemos además 2 reglas para transferir agua entre vasijas. La primera, **transfer1**, muestra el caso en el que una de las vasijas se queda vacía, mientras en la segunda se añade el contenido de la primera en su totalidad:

```
cr1 [transfer1] : vasija(M1, N1) vasija(M2, N2)
=> vasija(M1, 0) vasija(M2, N1 + N2)
if N1 + N2 <= M2 .
```

En la segunda regla, **transfer2**, llenamos la segunda vasija mientras en la primera todavía queda algo de agua:

```
cr1 [transfer2] : vasija(M1, N1) vasija(M2, N2)
=> vasija(M1, sd(N1 + N2, M2)) vasija(M2, M2)
if N1 + N2 > M2 .
endm
```


Vemos arriba que el módulo termina con la palabra reservada **endm**. Una vez cargado un módulo de sistema podemos ejecutarlo de diversas maneras. El comando básico es **rewrite**, abreviado **rew**, que aplica las ecuaciones y las reglas del módulo al término dado como argumento. Sin embargo, este ejemplo no termina (es posible vaciar todo el tiempo una vasija vacía, por ejemplo), así que tenemos que elaborar el comando un poco más. Podemos, por ejemplo, pedir a Maude que aplique como máximo 10 reglas poniendo dicho número entre corchetes después del comando **rew**:


```
Maude> rew [10] initial .
rewrite [10] in DIE-HARD : initial .
rewrites: 139 in 0ms cpu (0ms real) (1022058 rewrites/second)
result ConjVasija: vasija(3, 0) vasija(5, 0) vasija(8, 0)
```

Este resultado no es demasiado interesante, pues parece que el sistema no ha hecho nada. En efecto, lo más probable es que Maude haya simplemente aplicado la regla **vacía** 10 veces, lo que no tiene ningún efecto en el término inicial. Para forzar al sistema a aplicar distintas reglas podemos usar el comando **frew**, que indica que se debe realizar una *reescritura justa* (*fair rewrite* en inglés):

```
Maude> frew [10] initial .
frewrite in DIE-HARD : initial .
rewrites: 18 in 0ms cpu (0ms real) (~ rewrites/second)
result ConjVasija: vasija(3, 0) vasija(5, 5) vasija(8, 8)
```

Aunque ahora obtenemos un resultado en el que “ha ocurrido algo”, no nos sirve de mucho si lo que queremos saber es cómo alcanzar cierta configuración. Para ello podemos usar el comando **search**, que muestra si es posible alcanzar un cierto resultado a partir de otro dado como argumento. De ser posible, muestra el camino (esto es, la secuencia de reglas) que se debe seguir para alcanzarlo. En nuestro caso buscamos que cualquiera de las vasijas contenga 4 litros, así que indicamos que buscamos una solución mediante la opción **[1]** (si no ponemos esta opción Maude buscará todas las soluciones posibles); indicamos además que queremos que la búsqueda dé 0 o más pasos con la flecha **=>***; si el estado inicial cumpliera la condición nos lo devolvería como resultado (las otras opciones son **=>+**, que requiere 1 o más pasos, y **=>!**, que solo devuelve estados finales, es decir, expresiones a las que no se les pueden aplicar ecuaciones ni reglas). Por último indicamos la forma

del resultado con un patrón. Usamos variables ( que deben estar declaradas junto a su tipo, no es posible usar las variables del módulo) en los valores en los que no estamos interesados, como por ejemplo la capacidad de la vasija (queremos llegar a 4 litros, nos da igual la capacidad de la jarra que los contenga).

 La variable `B:ConjVasija` es importante pero suele olvidarse en muchos ejercicios: esta variable es la encargada de “cargar” con el resto de las vasijas que no nos interesan. Este tipo de variables, que podríamos llamar “recolectoras de basura”, suelen usarse para abstraer las partes de la expresión en las que no estamos interesados:

```
Maude> search [1] initial =>* vasija(N:Nat, 4) B:ConjVasija .
search in DIE-HARD : initial =>* B:ConjVasija vasija(N:Nat, 4) .
```

```
Solution 1 (state 75)
states: 76 rewrites: 2134 in 0ms cpu (8ms real) (~ rewrites/second)
B:ConjVasija --> vasija(3, 3) vasija(8, 3)
N:Nat --> 5
```

Vemos en el resultado que hemos conseguido 4 litros en la vasija de 5 litros, y que el estado en el que esto sucede es el 75. Ahora podemos usar el comando `show path` para ver el camino:

```
Maude> show path 75 .
state 0, ConjVasija: vasija(3, 0) vasija(5, 0) vasija(8, 0)
===[ rl vasija(M1, N1) => vasija(M1, M1) [label fill] . ]==>
state 2, ConjVasija: vasija(3, 0) vasija(5, 5) vasija(8, 0)
===[ crl vasija(M1, N1) vasija(M2, N2) => vasija(M1, sd(M2, N1 + N2))
      vasija(M2, M2) if N1 + N2 > M2 = true [label transfer2] . ]==>
state 9, ConjVasija: vasija(3, 3) vasija(5, 2) vasija(8, 0)
===[ crl vasija(M1, N1) vasija(M2, N2) => vasija(M1, 0) vasija(M2, N1 + N2)
      if N1 + N2 <= M2 = true [label transfer1] . ]==>
state 20, ConjVasija: vasija(3, 0) vasija(5, 2) vasija(8, 3)
===[ crl vasija(M1, N1) vasija(M2, N2) => vasija(M1, 0) vasija(M2, N1 + N2)
      if N1 + N2 <= M2 = true [label transfer1] . ]==>
state 37, ConjVasija: vasija(3, 2) vasija(5, 0) vasija(8, 3)
===[ rl vasija(M1, N1) => vasija(M1, M1) [label fill] . ]==>
state 55, ConjVasija: vasija(3, 2) vasija(5, 5) vasija(8, 3)
===[ crl vasija(M1, N1) vasija(M2, N2) => vasija(M1, sd(M2, N1 + N2))
      vasija(M2, M2) if N1 + N2 > M2 = true [label transfer2] . ]==>
state 75, ConjVasija: vasija(3, 3) vasija(5, 4) vasija(8, 3)
```

El comando `search` es muy potente, como veremos en la sección 4.4.1. Además de la forma que hemos visto arriba, es posible usar condiciones más complejas usando `such that` (abreviado como `s.t.`). El comando anterior podría reescribirse así:

```
Maude> search [1] initial =>* vasija(N:Nat, M:Nat) B:ConjVasija s.t. M:Nat == 4 .
search in DIE-HARD : initial =>* B:ConjVasija vasija(N:Nat, 4) .
```

```
Solution 1 (state 75)
states: 76 rewrites: 2134 in 0ms cpu (8ms real) (~ rewrites/second)
B:ConjVasija --> vasija(3, 3) vasija(8, 3)
N:Nat --> 5
```

4.3. Más ejemplos

Vamos a especificar una pequeña red en anillo en Maude, y en concreto vamos a ver cómo se realiza la elección del nodo líder. La idea es la siguiente: tenemos una serie de nodos, donde cada uno tiene una prioridad y solo conoce la dirección del siguiente nodo en la arquitectura. Todos los nodos tienen una prioridad, expresada como un número natural (diferente para cada uno); cuanto más alto sea dicho número mayor será la prioridad. Los nodos pueden estar en 3 estados diferentes: `inactivo` (`idle`), `activo` o `lider`; suponemos que el estado inicial de todos los nodos es `idle`. El código correspondiente a este ejemplo está disponible en el Campus Virtual en el fichero `lider.maude`.

El protocolo de elección funciona como sigue: (i) la primera acción que tomarán todos los nodos será pasar de `idle` a `activo`; cuando dicha activación tenga lugar se enviará un mensaje al nodo siguiente en la arquitectura indicando la prioridad del nodo; (ii) cuando un nodo activo recibe un mensaje con una cierta prioridad puede actuar de 3 maneras: (iia) si la prioridad que recibe en el mensaje es menor que la suya, entonces descarta el mensaje; (iib) si la prioridad que recibe en el mensaje es mayor que la suya, entonces reenvía el mensaje al siguiente nodo en la arquitectura; y (iic) si la prioridad que recibe en el mensaje es la suya, entonces cambia su estado a `lider`.

El módulo `LIDER` importa el módulo `QID`, que nos permite usar constantes que empiezan por `'`, como por ejemplo `'a` o `'hola`.

```
mod LIDER is
  pr QID .
```

Para empezar definimos los tipos y las constantes para el modo:

```
sorts Mode Node Configuration Attribute AttributeSet Msg .

ops idle activo lider : -> Mode [ctor] .
```

Vamos a definir un conjunto de atributos para los nodos usando usando las estructuras habituales:

```
subsort Attribute < AttributeSet .

op mt : -> AttributeSet [ctor] .
op _,_ : AttributeSet AttributeSet -> AttributeSet [ctor assoc comm id: mt] .

op modo':_ : Mode -> Attribute [ctor] .
op prioridad':_ : Nat -> Attribute [ctor] .
op siguiente':_ : Qid -> Attribute [ctor] .
```

Los nodos se identifican por un `Qid` y tienen un conjunto de atributos, mientras que los mensajes se definen con un destinatario y un natural que indica la prioridad:

```
op <_|_> : Qid AttributeSet -> Node [ctor] .

op to_:_ : Qid Nat -> Msg [ctor] .
```

Las configuraciones son a su vez conjuntos de nodos y mensajes:

```
subsort Msg Node < Configuration .

op none : -> Configuration [ctor] .
op __ : Configuration Configuration -> Configuration [ctor assoc comm id: none] .
```

Declaramos las variables como de costumbre:

```
var AtS : AttributeSet .
vars N N' : Nat .
vars O O' : Qid .
```

Y usamos reglas para definir el comportamiento. La regla que cambia el estado de *idle* a *activo* se define como sigue:

```
rl [activar] :
  < O | modo : idle, siguiente : O', prioridad : N >
=> < O | modo : activo, siguiente : O', prioridad : N >
  (to O' : N) .
```

Aquí hemos usado todos los atributos porque todos intervienen en la regla, pero el orden no importa. **Pista:** La regla para trabajar con mensajes es simple: si un mensaje aparece a la izquierda de la flecha (\Rightarrow) es que el mensaje se está *emphrecibiendo*; si aparece a la derecha es que se está *enviando*.

La regla condicional **borrar** modela el comportamiento explicado anteriormente para el escenario (iia). Cuando el objeto recibe un mensaje con una prioridad menor que la suya se elimina. Es importante ver que usamos la misma variable para el mensaje y el identificador del nodo, indicando así que el objeto que lo está recibiendo es el adecuado. También usamos una variable *AtS* para englobar los atributos que no intervienen en la regla:

```
crl [borrar] :
  (to O : N)
  < O | modo : activo, prioridad : N', AtS >
=> < O | modo : activo, prioridad : N', AtS >
  if N < N' .
```

La regla **reenviar** modela el comportamiento para el escenario (iib). Cuando el mensaje recibido tiene una prioridad mayor que la del nodo destino, se reenvía poniendo como nuevo destinatario el siguiente nodo:

```
crl [reenviar] :
  (to O : N)
  < O | modo : activo, prioridad : N', siguiente : O' >
=> < O | modo : activo, prioridad : N', siguiente : O' >
  (to O' : N)
  if N > N' .
```

Por último, al regla **lider** modela el escenario (iic). En este caso no es necesaria una regla condicional, pues podemos usar la misma variable tanto en el mensaje como en el atributo *prioridad*:

```
rl [lider] :
  (to O : N)
  < O | modo : activo, prioridad : N, AtS >
=> < O | modo : lider, prioridad : N, AtS > .
```

De esta forma podemos definir una configuración inicial con 5 nodos y cerrar el módulo con la palabra reservada **endom**. Es importante fijarse en que los identificadores de objetos tienen valores concretos, pues estamos definiendo una instancia concreta de red:


```

op init : -> Configuration .
eq init = < 'n1 | modo : idle, prioridad : 1, siguiente : 'n2 >
          < 'n2 | modo : idle, prioridad : 2, siguiente : 'n3 >
          < 'n3 | modo : idle, prioridad : 3, siguiente : 'n4 >
          < 'n4 | modo : idle, prioridad : 4, siguiente : 'n5 >
          < 'n5 | modo : idle, prioridad : 5, siguiente : 'n1 > .
endm

```

4.4. Verificación de propiedades en Maude

Una de las principales ventajas de especificar un sistema en Maude es que podemos verificar automáticamente muchas de sus propiedades. Aunque muchos tipos distintos de análisis están disponibles, como la verificación de sistemas en tiempo real o la demostración de teoremas, en este curso nos centraremos en la demostración de invariantes y en la comprobación de modelos.

4.4.1. Comprobando invariantes mediante búsqueda

Como sabemos, un *invariante* es una propiedad que se cumple durante toda la ejecución de un programa. Este tipo de propiedades es sencillo probarlas usando el comando **search**: basta con buscar un estado que no cumpla la propiedad. Si Maude es capaz de encontrarlo entonces el invariante es falso; en otro caso, es cierto.

En el ejemplo mostrado en la sección 4.3 podemos definir el invariante “El número de líderes es siempre menor o igual que 1”. Para comprobarla podemos definir un módulo INVARIANTE (disponible en el fichero `lider.maude`) que importe el módulo LIDER:

```

mod INVARIANTE is
  pr LIDER .

  var C : Configuration .
  var O : Qid .

```

Definiremos el invariante con la función `llider`, que comprueba que el número de líderes sea menor o igual que 1:

```

op llider : Configuration -> Bool .
eq llider(C) = numLideres(C) <= 1 .

```

Esta función usa una función auxiliar a cargo de contar los nodos en una configuración. La variable `C` se encarga, como hemos visto en otros ejemplos, de guardar información sobre los elementos que no nos interesan en un cierto momento:

```

op numLideres : Configuration -> Nat .
eq numLideres(< O | modo : lider, AtS > C) = 1 + numLideres(C) .
eq numLideres(C) = 0 [otherwise] .
endm

```

Ahora ya podemos comprobar el invariante. Como hemos dicho, para ello debemos buscar estados que *no* lo cumplan:

```

Maude> search init =>* C:Configuration s.t. not llider .

```

```

No solution.

```

En efecto, no hay ningún estado que cumpla la negación del invariante, es decir, el invariante se cumple en todos los estados y es, por tanto, correcto.

4.4.2. Comprobación de modelos

La comprobación de modelos o *model checking* es una técnica que permite comprobar si tu sistema (o modelo) satisface ciertas propiedades en lógica temporal. En el caso de Maude usaremos Lógica Lineal Temporal. En esta sección explicaremos cómo comprobar varias propiedades en LTL para el protocolo del líder que presentamos en la sección 4.3. Los pasos que se detallan a continuación se explican primero en general para después centrarse en el ejemplo concreto.

Paso 1. Definir un módulo, que en general llamaremos PROPS, que importe el módulo SATISFACTION y el módulo sobre el que queremos demostrar propiedades, en este caso LIDER:

```
mod PROPS is
  pr SATISFACTION .
  pr LIDER .
```

El módulo SATISFACTION es un módulo predefinido que define lo siguiente:

- El sort **State**, que indica el tipo sobre el que demostraremos las propiedades.
- El sort **Prop**, necesario para definir propiedades atómicas.
- El operador **op** `_|=_ : State Prop -> Bool` ., que permite definir cuándo un estado cumple una cierta propiedad.

Paso 2. Definir el estado sobre el que vamos a probar las propiedades. Para ello es necesario definir un subtipo del tipo **State**. En nuestro caso vamos a probar propiedades sobre configuraciones, así que tenemos:

```
subsort Configuration < State .
```

Paso 3. Definir, usando operadores, las propiedades que vamos a probar. Dichas propiedades deben tener tipo **Prop**. En nuestro caso, 4 propiedades diferentes: la primera comprueba si hay un líder en el sistema:

```
op hayLider : -> Prop [ctor] .
```

Las restantes propiedades comprueban si un nodo en concreto está en cierto estado. Para ello, es necesario que reciban como argumento un término de tipo **Qid** que identifique el nodo concreto que queremos estudiar:

```
ops idle activo esLider : Qid -> Prop [ctor] .
```

Paso 4. Definimos las propiedades con ecuaciones. En este caso basta con definir las ecuaciones para los casos en los que la propiedad se cumple; Maude deduce que el resultado es falso cuando la propiedad no está definida.⁸ Las ecuaciones se escriben usando un elemento de tipo **State** como primer argumento, seguido del operador `|=` y escribiendo la propiedad, con los argumentos que sean necesarios, después. El resultado será una expresión Booleana. Por ejemplo, nuestra primera propiedad se define así:

```
var C : Configuration .
var Q : Qid .

eq < Q | modo : lider, AtS > C |= hayLider = true .
```

⁸Fíjate que es lo mismo que definir un caso *otherwise* para cada propiedad.

Fíjate en la variable **C** que, como hemos visto en secciones anteriores, sirve para “recoger” la información del resto de la configuración, que no aparece en la ecuación. El resto de propiedades se definen de manera similar:

```
eq < 0 | modo : lider, AtS > C |= esLider(0) = true .
eq < 0 | modo : idle, AtS > C |= idle(0) = true .
eq < 0 | modo : activo, AtS > C |= activo(0) = true .
endom)
```

Paso 5. Crear un módulo que importe:

- El módulo **PROPS**, definido siguiendo los pasos anteriores.
- El módulo **MODEL-CHECKER**, que permite usar el comando `modelCheck`.
- El módulo **LTL-SIMPLIFIER**, que mejora la eficiencia del comprobador de modelos.
- Un término inicial sobre el que demostrar las propiedades. En nuestro caso esto no es necesario porque ya definimos una configuración `init` en la sección 4.3.

```
mod CHECK is
  pr PROPS .
  pr MODEL-CHECKER .
  pr LTL-SIMPLIFIER .
endom
```

Paso 6. Demostrar (o refutar) las propiedades. Una vez definidas las propiedades, solo nos queda usar el comando `modelCheck` para comprobar propiedades. Por ejemplo, podemos comprobar que, si alguna vez hay líder, ese líder es el nodo `'n5'`:

```
red modelCheck(init, <> hayLider -> <> esLider('n5')) .
```

Este comando puede devolver:

- `true`, indicando que la propiedad se cumple.
- Un contraejemplo, mostrando una posible ejecución que viola la propiedad.

4.4.3. Abstracción ecuacional

El mayor problema de los comprobadores de modelos es la *explosión del espacio de estados* [4]. Este problema, expresado de manera simple, se refiere a la existencia de un número tan grande de estados que es imposible guardarlos en memoria en un tiempo adecuado.

En Maude, el comprobador de modelos genera estados cuando se aplican reglas de reescritura pero no cuando se aplican ecuaciones, así que una posibilidad para reducir el número de estados es transformar las reglas en ecuaciones. Evidentemente, si aplicamos esta idea indiscriminadamente obtendremos resultados erróneos, por lo que debemos cumplir ciertas reglas:⁹

- Las nuevas ecuaciones deben ser *terminantes*. Es decir, las reglas

⁹De hecho deben cumplir otra condición que no tenemos en cuenta aquí: las reglas y las ecuaciones deben ser *coherentes*, es decir, no podemos perder posibilidades de reescritura por aplicar una ecuación.

```
rl a => b .
rl b => a .
```

no pueden transformarse en ecuaciones, pues entraríamos en un bucle infinito.

- Las nuevas ecuaciones deben ser *confluentes* (es decir, *deterministas*). Por tanto, las reglas

```
rl a => b .
rl a => c .
```

no pueden transformarse en ecuaciones, pues al reducir el valor **a** no obtenemos siempre un mismo valor.

- Las reglas transformadas deben ser *invisibles* para la propiedad que estamos probando. La idea de la “invisibilidad” es que la regla no afecta a la propiedad, es decir, si la propiedad era cierta antes de aplicarse la regla entonces lo seguirá siendo después; lo mismo ocurre en el otro caso: si la propiedad era falsa, debe seguir siéndolo. Por tanto, la invisibilidad depende de la propiedad concreta.

Por ejemplo, en el protocolo de la sección 4.4.2 tenemos la propiedad **hayLider**. Estudiemos las reglas en la sección 4.3 para ver cuáles podemos transformar:

- La aplicación de la regla **activar** no puede modificar la propiedad, pues ni aparece ni desaparece ningún líder. Por tanto, se podría transformar en ecuación.
- Las reglas **borrar** y **reenviar**, que simplemente manipulan mensajes, tampoco pueden modificar la propiedad y, por tanto, se pueden transformar en ecuaciones.
- La regla **lider** sí puede modificar la propiedad: si fuese falsa antes de aplicarla pasará a ser cierta después. Por tanto, esta regla no se puede transformar.

Siguiendo con el mismo ejemplo, si trabajamos con la propiedad **idle(0)** podremos transformar las reglas **borrar**, **reenviar** y **lider**, pero no la regla **activar**.

☞ Si la fórmula con la que trabajamos usa varios predicados atómicos, solo podemos transformar aquellas reglas que se puedan transformar en todos los casos. Por ejemplo, si tenemos una fórmula que usa a la vez las propiedades **hayLider** y **idle(0)** entonces solo podemos transformar las reglas **borrar** y **reenviar**, porque **activar** y **lider** fallan en alguno de los casos.

4.5. Usando Maude

Como se dijo en la sección 3, el sistema Maude está disponible (Linux y Mac) en la siguiente dirección:

<http://maude.cs.uiuc.edu>

Existe una versión para Windows disponible en:

<http://moment.dsic.upv.es/>

En ambas páginas existe un manual de instrucciones para su instalación. Una vez instalado el sistema, podemos iniciarlo, obteniendo una pantalla parecida a esta:

```

\|||||/
--- Welcome to Maude ---
/|||||
Maude 2.7 built: Feb  7 2014 15:12:51
Copyright 1997-2014 SRI International
Mon Sep 1 12:02:38 2014

```

Desde este momento podemos cargar ficheros en Maude usando el comando `load`. Por ejemplo, `lider.maude` se carga como sigue:

```
Maude> load lider.maude
```

Si el fichero se carga correctamente se enumeran los módulos que se han cargado. Alternativamente, se puede cargar el fichero con el comando `in`. Además, se puede escribir esta línea en los ficheros para cargar ficheros dependientes. Una tercera forma es dar la lista de ficheros que se quieren cargar al inicial Maude (suponiendo que Maude se lanza con el comando `maude`):

```
$ maude lider.maude
```

Cualquier comando que se escriba se ejecutará en el último módulo cargado. Esto es importante, pues si se han cargado varios módulos independientes y ciertas funciones no están accesibles en el último módulo Maude mostrará un error. Es posible cambiar el módulo usando el modificador `in`. Por ejemplo, si queremos ejecutar un comando en el módulo `M` debemos escribir:

```
Maude> red in M : 3 + 4 .
```

Esto además hará que el módulo actual pase a ser `M`. Por último, podemos salir del sistema con el comando `q`:

```
Maude> q
Bye.
```

4.6. Errores frecuentes

En esta sección se listan algunos errores más frecuentes que se han observado al trabajar en Maude:

- Los nombres de variables y los atributos de los operadores se separan por espacios, no por comas.
- Si se declaran varios operadores en una sola línea es necesario usar la palabra reservada `ops`. Si se utiliza `op` Maude interpretará que se está declarando un operador que usa espacios.
- Los comandos acaban en punto.
- No dejar espacio entre operadores infijos genera errores. Por ejemplo, la expresión `0+0` la considera un solo operador, por lo que hay que escribir `0 + 0`.
- En ocasiones un paréntesis mal cerrado hace que Maude siga esperando datos de entrada. Si sospechas que esto puede estar ocurriendo se recomienda probar un comando sencillo, como reducir 1, para observar si se obtiene un resultado o el sistema está simplemente bloqueado.

4.7. Módulos predefinidos

En esta sección listamos algunos de los módulos predefinidos más usados. Recuerda que todos los módulos están disponibles en el fichero `prelude.maude`.

Los módulos más utilizados en Maude son:

BOOL. El módulo **BOOL** se incluye por defecto en todos los módulos en Maude. Define el tipo **Bool**, con las constructoras **true** y **false**, y proporciona funciones Booleanas como la igualdad (`_==_`), la función unaria **not** y las binarias `_and_` y `_or_`.

NAT. El módulo **NAT** proporciona al usuario los tipos **Zero**, que identifica al número 0; **NzNat**, que identifica los naturales positivos; y **Nat**, que es la unión de los tipos anteriores. Además proporciona las funciones más comunes entre números naturales, como la suma (`_+_`) o la multiplicación (`_*_`).

☉☉ Los números naturales no tienen resta, sino diferencia simétrica (**sd**), que sustrae al número mayor el número menor. Por ejemplo, tenemos $\text{sd}(3, 4) = \text{sd}(4, 3) = 1$.

STRING. El módulo **STRING** proporciona el tipo **String** para representar cadenas de caracteres encerradas entre comillas (`"`). Por ejemplo los términos `"hola"` y `"123"` tienen tipo **String**.

QID. El módulo **QID** proporciona el tipo **Qid**, que permite definir cadenas de caracteres que empiecen por apóstrofe (`'`). Por ejemplo los términos `'hola` y `'123` tienen tipo **Qid**.

SATISFACTION. El módulo **SATISFACTION** proporciona los tipos **Prop**, para definir propiedades en LTL, y **State**, para definir el tipo sobre el cual probaremos las propiedades. También proporciona el operador `_|= _`, que permite definir propiedades atómicas tomando como primer argumento un término de tipo **State** y como segundo argumento un término de tipo **Prop**. Su resultado es un valor Booleano.

MODEL-CHECKER. El módulo **MODEL-CHECKER** permite usar el comando `modelCheck` para comprobación de modelos.

LTL-SIMPLIFIER. El módulo **SATISFACTION** simplifica las fórmulas introducidas en el comando `modelCheck` para optimizar su rendimiento.

Referencias

- [1] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.7)*, March 2015. <http://maude.cs.illinois.edu/w/images/1/1a/Maude-manual.pdf>.
- [2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
- [3] R. K. Dybvig. *The Scheme Programming Language*. The MIT Press, 3rd edition, 2003.
- [4] A. Farzan and J. Meseguer. State space reduction of rewrite theories using invisible transitions. In M. Johnson and V. Vene, editors, *Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006, Kuressaare, Estonia, July 5-8, 2006, Proceedings*, volume 4019 of *Lecture Notes in Computer Science*, pages 142–157. Springer, 2006.
- [5] T. McCombs. *Maude 2.0 Primer*, 2003. <http://maude.cs.uiuc.edu/primer/maude-primer.pdf>.

- [6] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML - Revised*. The MIT Press, 1997.
- [7] S. Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003.
- [8] P. Seibel. *Practical Common Lisp*. Apress, 2005.