Métodos Formales Industriales (MFI)

—prácticas—

Grado de Ingeniería en Informática

Práctica 2: Modelado y verificación en Maude

Santiago Escobar

Despacho 237
Edificio DSIC 2 piso

1. Introducción

En esta práctica vamos a aprender a modelar sistemas en el lenguaje de programación de alto rendimiento Maude y verificar algunas propiedades sobre dichos sistemas.

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.

Los lenguajes de especificación ejecutables como Maude proporcionan un vehículo expresivo claro y conciso enemigo de la ambigüedad propia de otro tipo de especificaciones como UML o SDL presentadas en lenguaje natural o semi-formal. Permiten considerar la propia especificación como un programa y, en consecuencia, ejecutar de forma inmediata la especificación definida. Esto proporciona una forma simple de prototipado, que permite al ingeniero obtener una primera aproximación de los resultados que su especificación produciría. Pero además, proporciona una forma sencilla y práctica de desarrollo de programas si se disponen de las herramientas de compilación y generación de código apropiadas para el lenguaje de especificación en cuestión.

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. El lenguaje de programación Maude

El lenguaje de programación Maude utiliza reglas de reescritura, como los lenguajes denominados funcionales tales como Haskell, ML, Scheme, o Lisp. En concreto, el lenguaje 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 del lenguaje Maude. Sin embargo, hay un extenso manual y un "*primer*" (libro de introducción muy sencillo y basado en ejemplos) en la dirección web indicada antes. Existe también un libro sobre Maude, con ejemplares adquiridos por la Biblioteca General y de la Escuela, y accesible online en:

`http://www.springerlink.com/content/p6h32301712p`

3. Verificación en Maude

Una forma de verificación menos potente que el model checking se consigue a través del comando de búsqueda `search`, que nos permitirá especificar propiedades de alcanzabilidad, e incluso de

seguridad (negación de alcanzabilidad) si el espacio de búsqueda es finito. A continuación mostramos un ejemplo sobre dos procesos y un semáforo para entrar a la zona crítica compartida y cómo podemos verificar algunas propiedades de alcanzabilidad.

El código en Maude que describe el problema de dos procesos accediendo a una zona compartida podría ser el siguiente:

```
mod SEMAPHORE is
   protecting NAT .
   protecting BOOL .

   --- Declaraciones de tipos para los procesos
   sorts Process PState .

   --- Un proceso tiene un identificador y su estado
   op p_{_} : Nat PState -> Process .

   --- Hay cuatro posibles estados
   ops idle entering critical exiting : -> PState .

   --- Ponemos todos los procesos juntos en un conjunto
   sort PSet .
   subsort Process < PSet .
   op empty : -> PSet .
   op __ : PSet PSet -> PSet [assoc comm id: empty] .
   eq X:Process X:Process = X:Process . --- Propiedad idempotencia

   --- Definimos el tipo semaforo
   sort Semaphore .
   subsort Bool < Semaphore .

   --- Juntamos todos los procesos con un semaforo a compartir
   --- que define la situacion global de todo nuestro sistema
   sort GlobalState .
   op _||_ : Semaphore PSet -> GlobalState .

   --- Variables
   var S : Semaphore . var PS : PSet . var Id : Nat .

   --- Y ahora damos la reglas que definen
   --- el comportamiento de cada proceso
   rl S || p Id {idle} PS --- El proceso sigue en idle
   => S || p Id {idle} PS .

   rl S || p Id {idle}     PS --- El proceso pasa a entering
   => S || p Id {entering} PS .

   rl false || p Id {entering} PS  --- El proceso entra en la zona critica
   => true  || p Id {critical} PS .

   rl S || p Id {critical} PS   --- El proceso sale de la zona critica
   => S || p Id {exiting}  PS . --- Solo un proceso estaba en su zona critica

   rl S     || p Id {exiting} PS  --- El proceso sale de la zona critica
   => false || p Id {idle}    PS .
```

```
endm
```

Este programa tiene un espacio de búsqueda finito, es decir, tiene un número finito de estados si se proporciona un conjunto finito de procesos. Por lo tanto, podemos especificar una propiedad de seguridad (negación de alcanzabilidad).

```
$ maude
           \||||||||||||||||||/
         --- Welcome to Maude ---
           /||||||||||||||||||\
   Maude 2.7.1 built: Jun 27 2016 16:43:23
    Copyright 1997-2016 SRI International
         Mon Feb 13 10:17:23 2017
Maude> load semaphore.maude
Maude> search false || p 1 {idle} p 2 {idle} =>* S || p 1 {critical} p 2 {critical} .
search in SEMAPHORE : false || p 1{idle} p 2{idle} =>* S || p 1{critical} p 2{critical} .

No solution.
states: 12  rewrites: 28 in 0ms cpu (5ms real) (~ rewrites/second)
Maude> quit
Bye.
```

En este caso Maude devuelve que no es posible ir del estado "`false || p 1 {idle} p 2 {idle}`" con dos procesos que comienzan en "idle" a un estado donde los dos procesos están en su zona crítica. Por lo tanto esta propiedad es cierta en todas las posibles ejecuciones.

4. Objetivo de la práctica

El objetivo de esta práctica consiste en realizar la especificación de los siguientes problemas en Maude y comprobar algunas propiedades de alcanzabilidad o seguridad. En todos los ejemplos hay que especificar módulos de sistema con reglas para especificar acciones indeterministas y, posiblemente, con ecuaciones para acciones deterministas.

Evaluación: La evaluación de esta práctica se realizará subiendo las respuestas a la tarea de PoliformaT.

4.1. Problema del semáforo

Modificar el ejemplo de los dos procesos con un semáforo para que haya un dispensador de tickets y cada proceso interesado en entrar en la zona crítica obtiene un número de ticket y sólo podrá entrar cuando le toque. ¿Se sigue satisfaciendo la misma propiedad de un único proceso en la zona crítica? Se adjunta el siguiente código como ejemplo.

```
mod SEMAPHORE is
   protecting NAT .
   protecting BOOL .

   --- Declaraciones de tipos para los procesos
   sorts Process PState .

   --- Un proceso tiene un identificador, tu ticket y su estado
   op p_[_]{_} : Nat Nat PState -> Process .
```

```
   --- Hay cuatro posibles estados
   ops idle entering critical exiting : -> PState .

   --- Ponemos todos los procesos juntos en un conjunto
   sort PSet .
   subsort Process < PSet .
   op empty : -> PSet .
   op __ : PSet PSet -> PSet [assoc comm id: empty] .
   eq X:Process X:Process = X:Process .

   --- Definimos el tipo semaforo
   sort Semaphore .
   subsort Bool < Semaphore .

   --- Juntamos todos los procesos con un semaforo a compartir
   --- que define la situacion global de todo nuestro sistema
   sort GlobalState .
   op _||_||_ : Semaphore Nat PSet -> GlobalState .

   var N : Nat . var PS : PSet .

   --- incrementar ticket servido (en base 10)
   rl false || N || PS  => false || (N + 1) rem 10 || PS .

   ... añadir reglas

endm
```

4.2. Un ascensor (típico de modelado de Sistemas Concurrentes)

El problema consiste en modelar el comportamiento de un ascensor de 4 plantas. Cada una de las plantas tiene un botón de llamada. El recorrido del ascensor de una planta a otra durará una única unidad de tiempo, es decir, el cambio de la planta 2 a la planta 3 debe realizarse en un único cambio de estado. Cuando llega a una de las plantas, el ascensor abre las puertas y las mantiene abiertas durante una unidad de tiempo, luego las cierra y procede a servir otra petición. El ascensor deberá mantener una dirección hasta satisfacer todas las peticiones existentes en esa dirección.

Deberéis especificar el problema como un programa Maude que satisfaga las siguientes propiedades:

1. si se pulsa un botón de llamada de una planta, éste es eventualmente satisfecho en alguna ejecución (propiedad de alcanzabilidad), y
2. el ascensor nunca se moverá con las puertas abiertas (propiedad de seguridad).

Se adjunta el siguiente código como ejemplo.

```
mod Ejercicio is
         protecting NAT .
         sorts Boton Pulsado Puertas Sentido Botones Ascensor .
         ops x o : -> Pulsado .
         ops <> >< : -> Puertas .
         ops up down : -> Sentido .
         op b_[_] : Nat Pulsado -> Boton .
         subsort Boton < Botones .
```

```
        op vacio : -> Botones .
        op __ : Botones Botones -> Botones [assoc comm id: vacio] .
        op _`(_,_,_`) : Botones Nat Sentido Puertas -> Ascensor .
        var Bts : Botones . var Piso : Nat .
        var UD : Sentido . var P : Puertas .
        rl [cerrar] : Bts (Piso,UD,<>) => Bts (Piso,UD,><) .
        crl [subir] : Bts (Piso,up,P) => Bts (Piso + 1,up,><) if Piso < 3 .
        crl [bajar] : Bts (Piso,down,P) => Bts (sd(Piso,1),down,><) if 1 < Piso .
        ...
endm
```

4.3. Alternating-bit-protocol (típico de modelado de Redes)

El protocolo de bit de alternancia es un protocolo muy simple pero efectivo que evita la aparición de mensajes duplicados o perdidos. Considerese un emisor A, un receptor B y un canal de A a B, que suponemos que inicialmente no transmite ningún mensaje. Cada mensaje contiene un bit de protocolo (valores 0 o 1).

Cuando A tiene un mensaje para enviar a B, lo repite indefinidamente por el canal de A a B utilizando el mismo bit de protocolo hasta que recibe una confirmación (acknowledgement) de B con el mismo bit de protocolo. Al recibir la confirmación, A empieza a transmitir el siguiente mensaje disponible con el bit de protocolo cambiado, es decir, mensaje anterior: 0, siguiente mensaje: 1 o al revés. Cuando B recibe un mensaje, lo acepta para procesar y manda a A una confirmación incluyendo el mismo bit de protocolo del mensaje recibido. Ante sucesivos mensajes con el mismo bit de protocolo, lo único que hace es mandar la misma confirmación con el mismo bit de protocolo.

Es decir, básicamente, el bit de protocolo identifica cada uno de los mensajes y el emisor repite el mensaje hasta que recibe una confirmación del receptor, a su vez el receptor repite siempre la confirmación del mensaje que le llega.

Deberéis especificar el problema como un programa Maude. Para modelar la pérdida de mensajes, el canal de comunicación deberá escoger de forma indeterminista entre copiar el mensaje en la salida o desecharlo. La propiedad a satisfacer es

> si dado un estado en que A envia un dato a B y B le devuelve algo, ese dato devuelto es justamente el mismo que A envió (propiedad de alcanzabilidad).

Hay que darse cuenta que la propiedad de que cuando A quiere enviar un 0, en algún momento le llegará una confirmación de vuelta no se puede probar completamente, ya que hay caminos donde los mensajes se pierden indefinidamente. Se adjunta el siguiente código como ejemplo.

```
mod Ejercicio is
        sorts Bit Mensaje Canal Estado .
        ops 0 1 : -> Bit .
        ops _>> <<_ : Bit -> Mensaje .
        subsort Mensaje < Canal .
        op perdido : -> Canal .
        op A{_}_B{_} : Bit Canal Bit -> Estado .
        var M : Mensaje .
        vars X Y Z : Bit .

        rl [perdida] : A{X}   M   B{Y} => A{X} perdido B{Y}  .
        rl [B-recibe] : A{X} Z >> B{Y} => A{X} Z >> B{Z}  .
        rl [B-repite] : A{X}  M   B{Y} => A{X} << Y B{Y}  .
        ...
endm
```

4.4. Problema de planificación de trenes (típico de Inteligencia Artificial)

Disponemos de cuatro trenes que salen de estaciones distintas y deben llegar todos a la misma estación. Los trenes todos viajan a 1 km/h pero tienen distintas características:

- Tren 1 tiene cuatro vagones y la distancia a recorrer es de 10 kilómetros.
- Tren 2 tiene diez vagones y la distancia a recorrer es de 10 kilómetros.
- Tren 3 tiene seis vagones y la distancia a recorrer es de 10 kilómetros.
- Tren 4 tiene dos vagones y la distancia a recorrer es de 20 kilómetros.

Sin embargo hay un túnel por el que todos los trenes deben pasar antes de llegar a la estación de destino y en el cual sólo cabe un tren cada vez y, debido al peso, sólo puede llevar dos vagones a la vez. El túnel se encuentra a 5 kilómetros antes de la estación de llegada.

Se deberá modelar este problema como un programa Maude. Se puede considerar que cada unidad de tiempo del sistema implica un kilómetro. La propiedad a verificar es que existe una solución al problema donde todos los trenes llegan al destino. Se adjunta el siguiente código como ejemplo.

```
mod Ejercicio is
    protecting INT .
    sorts Configuracion Vagon Vagones Tren Trenes .
    op x : -> Vagon .
    subsort Vagon < Vagones .
    op vacio : -> Vagones .
    op __ : Vagones Vagones -> Vagones [assoc comm id: vacio] .
    op tren_[_](_) : Int Int Vagones -> Tren .
    op tren*_[_](_) : Int Int Vagones -> Tren . --- pasando túnel
    op vacio : -> Trenes .
    subsort Tren < Trenes .
    op __ : Trenes Trenes -> Trenes [assoc comm id: vacio] .

    op _|#_#|_ : Trenes Trenes Trenes -> Configuracion .

    vars Id N : Int . vars VG VG1 VG2 : Vagones . vars TR1 TR2 TR3 : Trenes .

   crl [tren-avanza-izq] : tren Id [N](VG) TR1 |# TR2 #| TR3
                        => tren Id [N - 1](VG) TR1 |# TR2 #| TR3
    if N > 5 .
   crl [tren-avanza-der] : TR1 |# TR2 #| TR3 tren Id [N](VG)
                        => ...
    if ... .

    rl [empieza-tunel] :
       tren Id [5](x x VG) TR1 |# vacio #| TR3
    => tren* Id [5](VG) TR1 |# tren* Id [5](x x) #| tren* Id [5](vacio) TR3 .
    rl [entra-tunel] :
       tren* Id [5](x x VG1) TR1 |# vacio #| tren* Id [5](VG2) TR3
    => ... .
    rl [sale-tunel] :
       tren* Id [5](x VG1) TR1 |# tren* Id [5](x x) #| tren* Id [5](VG2) TR3
    => ... .
    rl [termina-tunel] :
```

```
       tren* Id [5](vacio) TR1 |# tren* Id [5](x x) #| tren* Id [5](VG2) TR3
    =>                     TR1 |# vacio #| tren Id [5](VG2 x x) TR3 .
endm
```