Métodos Formales Industriales (MFI)

—prácticas—

Grado de Ingeniería en Informática

Práctica 2: Modelado y verificación en Maude

Santiago Escobar

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.illinois.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 problema de dos procesos accediendo a una zona compartida podría codificarse como sigue:

```
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> search 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)
```

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.

Podemos hacer otra consulta para ver cómo funciona el comando search. Por ejemplo, qué configuraciones internas pueden tener los dos procesos partiendo del estado inicial del ejemplo anterior y sin que el semáforo esté a cierto.

```
Maude> search false || p 1{idle} p 2{idle} =>* false || p 1{X:PState} p 2{Y:PState} .

Solution 1 (state 0)
states: 1  rewrites: 0 in 0ms cpu (0ms real) (0 rewrites/second)
X:PState --> idle
Y:PState --> idle

Solution 2 (state 1)
states: 2  rewrites: 3 in 0ms cpu (0ms real) (63829 rewrites/second)
X:PState --> entering
Y:PState --> idle

Solution 3 (state 2)
states: 3  rewrites: 4 in 0ms cpu (0ms real) (57971 rewrites/second)
X:PState --> idle
Y:PState --> entering

Solution 4 (state 3)
states: 4  rewrites: 6 in 0ms cpu (0ms real) (60606 rewrites/second)
X:PState --> entering
Y:PState --> entering

No more solutions.
states: 12  rewrites: 28 in 0ms cpu (0ms real) (145833 rewrites/second)
```

Podemos además preguntar qué camino lleva a la cuarta solución, que corresponde al estado número 3 del grafo de alcanzabilidad.

```
Maude> show path 3 .
state 0, GlobalState: false || p 1{idle} p 2{idle}
===[ rl S || PS p Id{idle} => S || PS p Id{entering} . ]===>
state 1, GlobalState: false || p 1{entering} p 2{idle}
===[ rl S || PS p Id{idle} => S || PS p Id{entering} . ]===>
state 3, GlobalState: false || p 1{entering} p 2{entering}
```

También es posible pedirle a Maude que nos muestre todo el grafo de alcanzabilidad con el comando "`show search graph .`".

4. Objetivo de la práctica

El objetivo de esta práctica consiste en responder a la serie de preguntas cortas que se muestran a continuación, modificando los programas de la forma requerida. Vamos a utilizar el proyecto QUIZIFY de PSW como inspiración para los ejercicios de toda la asignatura.

Evaluación: La evaluación de esta práctica se realizará con un examen sobre las diez preguntas incluidas a continuación.

4.1. Modelado de quiz con dos nuevas constantes

El módulo funcional de la práctica anterior modela una lista de preguntas de un quiz o encuesta de QUIZIFY. Esa lista tendrá tantas posiciones como preguntas tenga el quiz. El símbolo `?` se añade a la lista para indicar que es una pregunta aún sin respuesta. Hemos asumido solamente cuatro respuestas en los quiz o encuestas: `A`, `B`, `C` y `D`.

Sin embargo, nos plantean que ahora las preguntas pueden haber sido leídas sin haber elegido una opción, por lo que decidimos extender el módulo funcional de la práctica anterior con una nueva constante `*` que representa que un usuario está leyendo la pregunta o la ha leído, sin responder.

```
fmod QUIZIFY-MAUDE-EXT1 is
  protecting NAT .
  sorts Opt OptV Opt? .
  op ? : -> Opt? [ctor] .
  op * : -> OptV [ctor] .
  ops A B C D : -> Opt [ctor] .
  subsort Opt < OptV < Opt? .

  sorts OptNat OptVNat Opt?Nat .
  subsort OptNat < OptVNat < Opt?Nat .
  op _<_> : Opt Nat -> OptNat [prec 10] .
  op _<_> : OptV Nat -> OptVNat [prec 10] .
  op _<_> : Opt? Nat -> Opt?Nat [prec 10] .

  sorts OptNatList OptVNatList Opt?NatList .
  subsort OptNatList < OptVNatList < Opt?NatList .
  op nil : -> OptNatList [ctor] .
  subsort OptNat < OptNatList .
  subsort OptVNat < OptVNatList .
  subsort Opt?Nat < Opt?NatList .
  op _@_ : Opt?NatList Opt?NatList -> Opt?NatList [ctor assoc id: nil] .
  op _@_ : OptVNatList OptVNatList -> OptNatList [ditto] .
  op _@_ : OptNatList OptNatList -> OptNatList [ditto] .

  var L : OptNatList . vars N M : Nat . var L? : Opt?NatList . var LV : OptVNatList .

  op add : Opt?NatList Nat -> Opt?NatList .
  eq add(L?,N) = L? @ ? < N > .

  op first* : Opt?NatList ~> Nat .
  eq first*(L @ * < N > @ L?) = N .

  op first? : Opt?NatList ~> Nat .
  eq first?(LV @ ? < N > @ L?) = N .
endfm
```

Se pueden lanzar las siguientes expresiones a ejecución:

```
reduce add(A < 1 > @ * < 2 > @ nil, 3) .
result Opt?NatList: A < 1 > @ * < 2 > @ ? < 3 >

reduce first*(* < 1 > @ ? < 2 > @ nil) .
result NzNat: 1

reduce first?(* < 1 > @ ? < 2 > @ nil) .
result NzNat: 2
```

(Pregunta 1) Nos dicen que el modelo anterior es incompleto y debemos tener también otra constante para cuando el usuario está listo para responder. Decidimos escoger la constante `#`. ¿Dónde es más conveniente añadir esta nueva constante? Razona tu respuesta pensando que el orden esperado para cada pregunta sería `?` $\rightarrow$ `*` $\rightarrow$ `#` $\rightarrow$ `A`, `B`, `C`, `D` . Modifica el código anterior de forma apropiada.

4.2. Modelado de quiz con semáforo

Ahora nos comentan que debemos asegurarnos que solo una pregunta está siendo respondida a la vez porque el usuario puede realizar el quiz desde varios dispositivos, como un teléfono móvil o una página web. Se nos ha ocurrido escribir el siguiente trozo de código usando un semáforo donde solo una pregunta de las que el usuario esté visualizando puede pasar a modo respuesta.

```
mod QUIZIFY-MAUDE-SEMAPHORE is
  protecting QUIZIFY-MAUDE-EXT2 .

  sort State .
  op _|_ : Bool OptNat?List -> State [ctor] .

  ...
  rl false | L1? @ * < N > @ L2? => true | L1? @ # < N > @ L2? .
  ...
endm
```

(Pregunta 2) Completa el código usando el ejemplo del semáforo de la sección anterior.

(Pregunta 3) ¿Qué resultado da el siguiente comando de búsqueda? Razona tu respuesta.

```
Maude> search false | ? < 1 > @ ? < 2 > @ ? < 3 > =>* B | L .
```

(Pregunta 4) ¿Qué resultado da el siguiente comando de búsqueda? Razona tu respuesta.

```
Maude> search false | ? < 1 > @ ? < 2 > @ ? < 3 > =>* B | L1? @ * < N > @ L2? .
```

(Pregunta 5) ¿Qué resultado da el siguiente comando de búsqueda? Razona tu respuesta.

```
Maude> search false | ? < 1 > @ ? < 2 > @ ? < 3 >
          =>* true  | L1? @ # < N > @ L2? @ # < M > @ L3? .
```

4.3. Modelado de quiz con un protocolo

Se complica la situación y nos dicen que en la realidad no es tan sencillo, porque hay un mensaje que cada dispositivo envía a un servidor central y éste debe responder dando permiso.

Se nos ha ocurrido escribir el siguiente trozo de código, donde los dispositivos no se modelan pero sí sus interacciones con el servidor. Hemos sustituido el valor booleano por un canal de comunicación compartido donde se depositan las peticiones y un "servidor" responde depositando las respuestas. El servidor almacena o bien `void` ó la pregunta permitida para responder. Hemos representado tres tipos de mensajes que se intercambian entre los dispositivos y el servidor.

- "`#? N`" Un dispositivo está visualizando la pregunta `N` y solicita pasar a modo respuesta.
- "`#! N`" El servidor da permiso para la pregunta `N` solo si no había ninguna permitida y, entonces, se almacena ese índice de pregunta.
- "`x N`" La pregunta ha sido respondida y el servidor pasa a `void`.

```
mod QUIZIFY-MAUDE-CHOREOGRAPHY is
  protecting QUIZIFY-MAUDE-EXT2 .

  sort State .
  op _|_|_ : Nat? MsgSet Opt?NatList -> State [ctor] .

  sort Nat? .
  subsort Nat < Nat? .
  op void : -> Nat? [ctor] .

  sorts Msg MsgSet .
  ops #?_ #!_ x_ : Nat -> Msg [ctor] .
  subsort Msg < MsgSet .
  op empty : -> MsgSet .
  op __ : MsgSet MsgSet -> MsgSet [assoc comm id: empty] .
  eq X:Msg X:Msg = X:Msg . --- Propiedad idempotencia

  vars L1 L2 : OptNatList . vars L1V L2V : OptVNatList . vars L1# L2# : Opt#NatList .
  vars L? L1? L2? L3? : Opt?NatList .
  var N? : Nat? . var MS : MsgSet . vars N M : Nat .

  --- One device
  ...
  rl N? | MS | L1? @ ? < N > @ L2? => N? | MS #? N | L1? @ * < N > @ L2? .
  rl N? | MS | L1? @ # < N > @ L2? => N? | MS x N | L1? @ A < N > @ L2? .
  rl N? | MS | L1? @ # < N > @ L2? => N? | MS x N | L1? @ B < N > @ L2? .
  rl N? | MS | L1? @ # < N > @ L2? => N? | MS x N | L1? @ C < N > @ L2? .
  rl N? | MS | L1? @ # < N > @ L2? => N? | MS x N | L1? @ D < N > @ L2? .
  ...
  --- Server
  rl void | MS #? N | L? => N | MS #! N | L? .
  ...

endm
```

(Pregunta 6) Completa el código del módulo anterior.

(Pregunta 7) ¿Qué resultado da el siguiente comando de búsqueda? Razona tu respuesta.

```
Maude> search void | empty | ? < 1 > @ ? < 2 > @ ? < 3 >
          =>* void | empty | A < 1 > @ B < 2 > @ C < 3 > .
```

(Pregunta 8) ¿Qué resultado da el siguiente comando de búsqueda? Razona tu respuesta.

```
Maude> search void | empty | ? < 1 > @ ? < 2 > @ ? < 3 >
          =>* N | MS | L? .
```

(Pregunta 9) ¿Qué resultado da el siguiente comando de búsqueda? Razona tu respuesta.

```
Maude> search void | empty | ? < 1 > @ ? < 2 > @ ? < 3 >
          =>* N?   |   MS  | L1? @ # < N > @ L2? @ # < M > @ L3? .
```

4.4. Modelado de quiz con pérdida de mensajes

Se complica aún más la situación y nos dicen que el canal de comunicación es muy inestable y algunos mensajes se pierden y la persona de la empresa pide un modelo más tolerante a fallos. La idea de que los mensajes se pierdan la expresamos con la siguiente regla que elimina todos los mensajes del canal sin que el servidor o los dispositivos puedan recibirlos.

```
rl N? | MS | L? => N? | empty | L? .
```

(Pregunta 10) Arregla el módulo de sistema anterior para que funcione el siguiente comando (y otros similares).

```
Maude> search void | empty | ? < 1 > @ * < 2 > @ ? < 3 >
          =>* void | empty | ? < 1 > @ A < 2 > @ ? < 3 >  .
```