Práctica 3: La panadería de Lamport y el lenguaje PARALLEL++

Ejercicio 1: La panadería de Lamport

El protocolo de la panadería de Lamport es un protocolo que logra la exclusión mutua entre procesos utilizando el sistema habitual en panaderías y otros comercios. El sistema consiste en tener un dispensador de números, de forma que los clientes toman un número al llegar al comercio y son atendidos secuencialmente de acuerdo con el número que tienen.

Especifica el protocolo de la panadería de Lamport como una teoría de reescritura en Maude siguiendo las siguientes indicaciones en un módulo `BAKERY` (en un fichero `bakery.maude`).

- Modelaremos el sistema como una colección de objetos, que tendremos dentro de un operador

  ```
  op [[_]] : Configuration -> GBState .
  ```

- Un proceso puede estar en modo `sleep`, `wait` o `crit`, y cuando entra en la panadería obtiene un número de orden. Los procesos se representarán como objetos de una clase `BProcess`, con atributos `mode` y `number`:

  ```
  class BProcess | mode: Mode, number: Nat .
  ```

- Los números de la panadería estarán gestionados por un dispensador, que tendrá el número que debe ser atendido en un momento determinado (`next`) y el último número de orden dispensado (`last`). Para ello definiremos una clase `Dispenser`.

  ```
  class Dispenser | next: Nat, last: Nat .
  ```

- El comportamiento del sistema viene dado por tres posibles acciones:
  - cuando un proceso pasa de estado `sleep` a estado `wait` toma el número de orden disponible en el dispensador (`last`), el cual es incrementado;
  - cuando llega el turno de un proceso (coincide su número con el `next` del dispensador), este pasa de modo `wait` a modo `crit`; y
  - cuando un proceso termina su sección crítica, este pasa a modo `sleep`, se queda con número de orden 0, y se pasa el turno al siguiente, es decir, se incrementa el `next` del dispensador.

- Define una operación

  ```
  op initial : Nat -> GBState .
  ```

  que permita crear sistemas con un número cualquiera de procesos. Inicialmente, el dispensador tendrá `1` como valores de `next` y `last`.

[Q1] Analiza la confluencia, terminación y coherencia del sistema definido.

[Q2] ¿Es el espacio de búsqueda alcanzable a partir de estados definidos con el operador `initial` finito? Utiliza el comando `search` para comprobar la exclusión mutua del sistema con 5 procesos.

[Q3] Utiliza el comando `search` para comprobar si hay estados de bloqueo.

En el mismo fichero `bakery.maude`, crea un módulo `ABSTRACT-BAKERY` en el que se defina una abstracción de la teoría de reescritura definida en el módulo `BAKERY` utilizando la siguiente idea. Conforme el sistema avanza, los números de los procesos en espera están en el rango definido por los valores `next` y `last - 1` del dispensador. En realidad, nos da igual que los valores estén en el rango [`next`,`last`) que en [`next`-1,`last`-1) siempre que no se cambie el orden entre los procesos. Podemos por tando abstraer el sistema simplemente decrementando en uno los números de orden de todos los procesos (los que no estén en modo `sleep`, que tendrán número de orden 0), siempre que el valor de `next` sea mayor que 1. Observa que de esta forma identificamos todos aquellos estados que tienen sus procesos en los mismos modos y con el mismo orden entre ellos.

[**Q4**] Justifica la validez de la abstracción (protección de los booleanos, confluencia y terminación de la parte ecuacional, y coherencia de ecuaciones y reglas).

[**Q5**] En el módulo `ABSTRACT-BAKERY`, ¿es finito el espacio de búsqueda alcanzable a partir de estados definidos con el operador `initial`?

[**Q6**] Utiliza el comprobador de modelos de Maude para comprobar la exclusión mutua del sistema con 5 procesos.

[**Q7**] Utiliza el comprobador de modelos de Maude para comprobar si hay estados de bloqueo.

Ejercicio 2: La panadería modificada

Contruye un módulo `BAKERY+` que extienda el módulo `BAKERY` y permita que un proceso abandone la espera.

- Un proceso en modo `wait` puede pasar a modo `sleep` en cualquier momento. Al hacerlo perderá su número de orden, pasando a ser este `0`.
- Cuando llegue el turno de un proceso que ha abandonado la panadería, no será posible darle paso al siguiente proceso. Para adaptarnos a la nueva situación, el dispensador podrá pasar el turno (incrementar su `next`) si no hay ningún proceso con dicho número de orden.

[**Q8**] Analiza la confluencia, terminación y coherencia del sistema definido.

[**Q9**] ¿Es finito el espacio de búsqueda alcanzable a partir de estados definidos con el operador `initial`? Utiliza el comando `search` para comprobar la exclusión mutua del sistema con 5 procesos.

[**Q10**] La abstracción proporcionada por el módulo `ABSTRACT-BAKERY` no es suficiente para este sistema modificado, ¿por qué? Especifica una abstracción válida para este nuevo sistema en una módulo `ABSTRACT-BAKERY+`.

[**Q11**] Utiliza la abstracción anterior para comprobar la no existencia de bloqueos y la exclusión mutua utilizando el comando `search`.

Ejercicio 3: El lenguaje PARALLEL++

En este ejercicio vamos a extender el lenguaje PARALLEL visto en clase de la siguiente forma:

- Además de variables de tipo entero, daremos soporte para variables de tipo booleano, con constantes `true` y `false`. En lugar de utilizar el tipo `Qid` para representar variables utilizaremos tipos `IntVar` para variables de tipo `Int` y `BoolVar` para variables de tipo `Bool`. Las variables serán declaradas entonces como constantes de estos tipos. Por ejemplo, podemos tener una variable `turn` de tipo `IntVar` añadiendo una declaración

  ```
  op turn : -> IntVar .
  ```

- Modificaremos los operadores de comparación `_=_` y `_>_` de forma que, no sólo permitan comparar una variable con un valor literal, sino que nos permitan comparar cuales quiera dos expresiones (de tipo `Int`). Podremos utilizar también `_=_` para comparar expresiones de tipo booleano.
- Añadiremos un tercer operador de comparación `_!=_` que permita comprobar si dos expresiones, de tipos entero o booleano, son distintas (tras ser evaluadas).
- Añadiremos soporte para arrays de enteros y de booleanos, con operaciones `_[_]`, para acceder al valor en una posición determinada de una variable de tipo array, y `_[_] :=_`, para modificar dicho valor. Así, por ejemplo, podremos escribir expresiones como `wants-to-enter[1]` o `wants-to-enter[0] := true`. En este caso, `wants-to-enter` será declarada como una constante de tipo `BoolVar` (por lo que no podemos utilizar el carácter `_`). Para guardar los valores de las variables de tipo array en memoria tendremos declaraciones

```
op [_,_] : IntVar List{Int} -> Memory .
op [_,_] : BoolVar List{Bool} -> Memory .
```

de forma que a una variable de tipo array se le asocie una lista de elementos del tipo correspondiente.

- Añadiremos operadores `if_then_else_fi` y `repeat_until_li` con la semántica habitual.

Se creará un fichero `parallel++.maude` con tantos módulos como se considere necesario.

[**Q12**] Comprueba utilizando el comando `search` la ausencia de bloqueo y la exclusión mutua de esa versión del algoritmo.