Ejercicio 3: El lenguaje PARALLEL++ y la verificación de progrmaas escritos en él

En este ejercicio vamos a extender el lenguaje PARALLEL visto en clase y vamos a utilizar el comando `search` y el model checker de Maude para verificar programas escritos en él.

[Q1] Sigue todos los pasos necesarios para utilizar el model checker de Maude para verificar la exclusión mútua del algoritmo de Dekker escrito en el lenguaje PARALLEL. (Todos: comprobación de que el espacio de búsqueda es finito, comprobaciones de confluencia, terminación, protección de booleanos, definición de proposiciones atómicas, ...)

[Q2] Define la semántica del lenguaje PARALLEL++. Se creará un fichero `parallel++.maude` con tantos módulos como se considere necesario, que extenderán los módulos en el `parallel.maude`. Este lenguaje extiende el lenguaje PARALLEL de la siquiente forma:

- Añadiremos un operador `_%_` para calcular el resto de una división entera. (El operador correspondiente en Maude es `_rem_`.)
- Además de variables de tipo entero, daremos soporte para variables de tipo booleano, con constantes `true` y `false`. Definiremos un tipo `BoolVar` para variables de tipo `Bool`. Igual que para variables enteras, las variables de tipo booleano serán declaradas entonces como constantes de ese tipo. Por ejemplo, podemos tener una variable `turn` de tipo `IntVar` y una variable `wte0` de tipo booleano añadiendo declaraciones

  ```
  op turn : -> IntVar .
  op wte0 : -> BoolVar .
  ```

- Modificaremos los operadores de comparación `_=_` y `_>_` de forma que no sólo permitan comparar una variable con un valor literal como en PARALLEL, 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).
- Además de la conjunción de expresiones booleanas, añadiremos operadores para su disjunción (`_|_`) y su negación (`!_`).
- 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. Para ello definiremos tipos `IntArray` y `BoolArray` cuyos términos vendrán dados por constructores

  ```
  op _[_] : IntVar IntExpr -> IntArray .
  op _[_] : BoolVar IntExpr -> BoolArray .
  ```

  Obsérvese que las variables de tipo array se declaran como constantes del tipo `???Var` correspondiente. Extenderemos el operador `_:=_` para permitir la modificación de valores en posiciones de arrays. Por ejemplo, dada una declaración

  ```
  op wants-to-enter : -> BoolVar .
  ```

  podremos escribir una expresión como `wants-to-enter[1]` o una instrucción como `wants-to-enter[0] := true`. (Obsérvese que como `wants-to-enter` ha de ser declarada como una constante de tipo `BoolVar` no podemos utilizar el carácter `_` en su identificador.)

  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. El acceso para consulta o modificación de una posición del array resultará en la operación correspondiente sobre la posición de la lista sobre la que se guarda el array. Supondremos que los

arrays están inicializados con el tamaño adecuado con valores por defecto (`0` para arrays de tipo `Int` y `false` para arrays de tipo `Bool`).

- Añadiremos operadores `if_{_}else{_}` y `repeat{_}until_` con la semántica habitual.
- Tendremos una nueva instrucción `new(_,_)` que cree una nueva hebra de ejecución, un nuevo proceso, con el identificador especificado como primer argumento y el código especificado como segundo argumento.

[Q3] Utiliza el comando `search` para verificar la exclusión mutua y la ausencia de bloqueos para el algoritmo de Dekker en su implementación original usando las definiciones de PARALLEL++. (El código está en el fichero `dekker++.maude`.)

[Q4] Modifica la implementación del algoritmo de Dekker en el fichero `dekker++.maude` (crea una copia en un fichero `dekker++-modificado.maude`) de forma que tengamos un programa con el que podemos crear las hebras correspondientes. Tendremos un programa `main`:

```
op main : -> Program .
eq main = new(0, p(0)) ; new(1, p(1)) .
```

De forma que el estado inicial de nuestro sistema pueda ser simplemente:

```
eq initial = { [2, main], [wants-to-enter, false false] [turn, 0] } .
```

[Q5] Comprueba utilizando el comando `search` la ausencia de bloqueo y la exclusión mutua de esa versión del algoritmo.

[Q6] Escribe el algoritmo de la panadería de Lamport (el original) utilizando el lenguaje PARALLEL++ y utiliza su semántica de reescritura para comprobar la ausencia de bloqueo y la exclusión mutua.

[Q7] Utiliza el model checker para probar la exclusión mutua y la viveza débil. Al tener un espacio de búsqueda infinito necesitaremos realizar una abstracción.