Practica3-Maude

Ejercicio 1.1: La panadería de Lamport

1 - Analiza la confluencia, terminación y coherencia del sistema definido.

Es confluente, puesto que en todo momento solo hay una ecuación en la que un input puede hacer matching. De forma que solo hay un posible camino de reducción y por tanto no puede ser no-confluente.

No es terminante, ya que el protocolo logra la exclusión mutua entre procesos, y por lo tanto no hay estados de bloqueo, considerados finales. La siguiente ejecución muestra un contraejemplo:

```
Maude> rew initial(5) .
rewrite in Bakery : initial(5) .
Debug(1)> q
Bye.
```

Como podemos ver el estado generado no termina y por tanto no es terminante.

Es coherente, dado que una vez creado el estado inicial, todos los estados se encuentran en forma irreducible. De forma que no se intercalan pasos de rescriturar con reducción, y por tanto no puede ser no-coherente.

2 - ¿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.

El espacio de búqueda es infinito. Dado que cada vez que un cliente (*BProcess*) hace un ciclo—Entendiendo hacer un ciclo como que pasas de sleep a wait, de wait a crit y de crit de nuevo a sleep—el número del tícket expendido por el dispensador (*last*) ha incrementado en al menos 1, y dado que no es terminante; se pueden generar infinitos estados en lo que único que cambia es el número del ticket expendido, que puede tomar como valor todo número naturarl (*Nat*).

```
Maude> search [1] initial(5) =>* [[< O:Nat : BProcess | mode:
crit, number: P:Nat > < O':Nat : BProcess | mode: crit, number:
P':Nat > C:Configuration ]] s.t. O:Nat =/= O':Nat .  search [1] in
Bakery : initial(5) =>* [[C < O : BProcess | mode: crit, number:
P > < O':Nat : BProcess | mode: crit,number: P':Nat >]] such that
O =/= O':Nat = true . Debug(1)> q Bye.
```

Como podemos ver no podemos demostrarlo con el comando search, ya que al ser el espacio de búqueda infinito, es imposible explorar todo el especio de búqueda. Lo único que podemos decir es que no lo ha encontrado en lo que ha explorado.

3 - Utiliza el comando search para comprobar si hay estados de bloqueo.

```
Maude> search [1] initial(5) =>! S:GBState .
search [1] in Bakery : initial(5) =>! S:GBState .
Debug(1)> q
Bye.
```

omo podemos ver no podemos demostrarlo con el comando search, ya que al ser el espacio de búqueda infinito, es imposible explorar todo el especio de búqueda. Lo único que podemos decir es que no lo ha encontrado en lo que ha explorado.

Ejercicio 1.2: ABSTRACT-BAKERY

4 - Justifica la validez de la abstracción (protección de los booleanos, confluencia y terminación de la parte ecuacional, y coherencia de ecuaciones y reglas).

Por cómo están definidas las reglas de este problema, los estados

```
[[< 0 : Dispenser | next: s(N), last: s(M) > < 0 : BProcess |
mode: M, number: s(N0) > < 1 : BProcess | mode: M, number: s(N1)
> ...]]
```

son equivalentes a

```
[[< 0 : Dispenser | next: N, last: s(M) > < 0 : BProcess | mode:
M, number: N0 > < 1 : BProcess | mode: M, number: N1 > ...]]
```

- **Protección de los booleanos**: No se identifican *true* y *false*, luego hay protección de los booleanos.
- **Confluencia**: Las única ecuacion que podría dar lugar a que la especificación fuese no-confluente es *simplify*, ya que al estar definida para un tipo commutativo (*Configuration*) la misma expresión puede simplificarse escogiendo para simplificar cualquiera de sus elementos y luego el resto de ellos recursivamente.

```
op simplify : Configuration -> Configuration .
eq simplify(none) = none .
eq simplify(< N : BProcess | mode: M, number: 0 > C)
    = < N : BProcess | mode: M, number: 0 > simplify(C) .
eq simplify(< N : BProcess | mode: M, number: s(N') > C)
    = < N : BProcess | mode: M, number: N' > simplify(C) .
```

Pero que se simplifique primero uno u otro elemento de la configuración, no influye, ya que al terminar la recursión ambos elementos estarán simplificados, y el resto de ellos también. Es decir una misma configuración de entrada confluye en una variación de esta en la que todos los clientes (*BProcess*) han decrementado el número del ticket en 1 si este era mayor que 1.

- **Terminación**: La parte ecuacional es terminante, al igual que lo era en *BAKERY*. Las únicas ecuaciones que podrían llevar a una no-terminación son las recursivas—en el caso de que hubiese ciclos. Pero en este caso, tanto *init* como *simplify* son terminantes. Vamos a demostrarlo.
  - ***init***: Está definida de forma recursiva y siempre que se llama decrementa el valor de entrada en 1. De forma que cuando llegue a 0, tiene un caso base `init(0) = none`, por lo que es terminante.
  - ***simplify***: Está definida de forma recursiva sobre un conjunto de elementos (*Configuration*), de forma que en cada llamada procesa uno de estos y llama recursivamente para procesar el resto del conjunto. De forma que cuando el conjunto se queda sin elementos esta recursión termina con el caso base `simplify(none) = none`.
- **Coherencia**: En este caso las únicas ecuaciones que se aplican entre ejecuciones de reglas son las de abstracción. Para demostrar que son coherentes basta con demostrar la igualdad `eq(rl(x)) = eq(rl(eq(x)))`.

Vamos a suponer que la función *rl* es la equivalente `to_sleep(to_crit(to_wait(x)))` es decir la equivalente a que que un cliente haga un ciclo completo—entendiendo ciclo comom que entre a la tienda y pida el ticket `to_wait`, el panadero le atienda `to_crit` y una vez comprado el pan se vaya de la tienda `to_slepp`. Y que *eq* es nuestra regla de simplificación. Dado que *rl* es la comppsición de nuestras reglas, si *rl* es coherente con *eq* es porque todas ellas (las reglas) lo son—ya que si alguna no fuese coherente con *eq rl* tampoco lo sería.

Sea x (irreducible) el estado en el que había un cliente esperando al que le tocab. Y al aplicar *rl(x)* que el panadero atienda al que estaba esperando, llegue un cliente y haga el ciclo completo y el cliente que estaba esperando vuelva a coger ticket:

Sea `M' = M - N`

```
x = [[< 0 : Dispenser | next: s(N), last: s(M) > < cliente :
BProcess | mode: wait, number: s(N) >]]
```

```
rl(x) = [[< 0 : Dispenser | next: sss(N), last: sss(M) > < cliente
: BProcess | mode: wait, number: sss(N) >]]
```

```
eq(rl(x)) = [[< 0 : Dispenser | next: s(0), last: s(M') > <
cliente : BProcess | mode: wait, number: s(0) >]]
```

```
eq(x) = [[< 0 : Dispenser | next: s(0), last: s(M') > < cliente :
BProcess | mode: wait, number: s(0) >]]
```

```
rl(eq(x)) = [[< 0 : Dispenser | next: sss(0), last: sss(M') > <
cliente : BProcess | mode: wait, number: sss(0) >]]
```

```
eq(rl(eq(x))) = [[< 0 : Dispenser | next: s(0), last: s(M') > <
cliente : BProcess | mode: wait, number: s(0) >]]
```

Como podemos ver `eq(rl(x)) = eq(rl(eq(x)))` por lo que *rl*, que es la composición de todas nuestra reglas (un ciclo), es coherente. Por consecuente todas lo son, es decir, nuestro módulo es coherente.

5 - En el módulo ABSTRACT-BAKERY, ¿es finito el espacio de búsqueda alcanzable a partir de estados definidos con el operador initial?

Sí, el espacio de búsqueda alcanzable **es finito**. Podemos demostrarlo dando una cóta máxima finita.

Sea N el número de clientes:

El dispensador de tickets (*Dispenser*) puede tener una difernecia entre el número del último ticket asignado y el número de ticket al que le toca el turno de máximo el número de clientes que hay, es decir N posibles estados.

Cada cliente pude tener tres estados: fuera da tienda (*sleep*), esperando con un ticket en la mano (*wait*) ó siendo atendido por el panadero (*crit*) y para *wait* el número del ticket que tiene en la mano puede tomar tantos valores como clientes hay, es decir N. Por lo que un cliente pude tomar N+2 estados diferentes.

Luego `O(estados(N)) = N * (N+2)^N` que es un número finito.

6 - Utiliza el comprobador de modelos de Maude para comprobar la exclusión mutua del sistema con 5 procesos.

Para ello creamos una propiedad *dissaster* que se satisfaga si hay dos procesos en estado crítico a la vez, de forma que comprobar con el model-checker la exclusión mútua sea tan simple como negar siempre *dissaster*.

```
Maude> red modelCheck(initial(5), [] ~ dissaster) .
reduce in BAKERY-CHECK : modelCheck(initial(5), []~ dissaster) .
rewrites: 8447 in 9ms cpu (10ms real) (848347 rewrites/second)
result Bool: true
```

7 - Utiliza el comprobador de modelos de Maude para comprobar si hay estados de bloqueo.

Para comprobar que no hay estados de bloqueo, podemos comprobar que siempre hay un estado siguiente. Para ello podemos usar la expresion `[] (True -> O True)`.

```
Maude> red modelCheck(initial(5), [] (True -> O True)) .
reduce in BAKERY-CHECK : modelCheck(initial(5), [](True -> O True)) .
rewrites: 19 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

Ejercicio 2: La panadería modificada

8 - Analiza la confluencia, terminación y coherencia del sistema definido.

Es confluente, pese a que nuestro modulo bakery+ cuenta con una nueva operacion, esta solo es llamada por el conjunto de reglas. Respeco al input, seguimos teniendo solo una posibilidad de hacer matching, lo que implica que este sistema tambien es confluente, al no haber ningun camino de reduccion posible que lo haga no confluente.

No es terminante, una vez mas, debido a que el programa es incapaz de alcanzar un estado de bloqueo gracias a la exclusion mutua entre los procesos. Tal como se muestra a continuacion:

```
Maude> load bakery+.maude
Maude> rew initial(5) .
rewrite in BAKERY+ : initial(5) .
Debug(1)> q
Bye.
```

Como la ejecucoion no termina, entonces es no terminante.

Es coherente, ya que una vez creado el estado inicial, todos los estados se encuentran en forma irreducible. Por lo que no se intercalan pasos de rescriturar con reducción, y por tanto no puede ser no-coherente.

9 - ¿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.

No, **no es finito** puesto que si un cliente coge un ticket y lo suelta seguidamente, lo que generamos es el mismo estado con *last* incrementado en 1. Este proceso pude repetirse infinitamente, dando como resultado un infinito número de estados equivalentes con *last* diferentes.

Si el espacio de búqueda fuese finito, el comando *search* siempre terminaría. En cambio el siguiente comando no termina:

```
Maude> load bakery+.maude
Maude> search initial(3) =>! S:GBState .
search in BAKERY+ : initial(3) =>! S:GBState .
Debug(1)> q
Bye.
```

Comprobamos la exclusión mútua para 5 procesos:

```
Maude> rew initial(5) .     Maude> search [1] initial(5) =>* [[C <
O : BProcess | mode: crit, number: P:Nat > < O':Nat : BProcess
| mode: crit,number: P':Nat >]] such that O =/= O':Nat = true .
Debug(1)> q Bye.
```

Como el espacio de busqueda es infinito, a pesar de que no encuentre ningún contraejemplo no podemos asegurar que no lo haya. Necesitamos una abstracción.

10 - 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+.

Por que a pesar de que el intervalo `[N + next, N + last]` pase a `[1, last - next + 1]`, el `last - next + 1` **no** tiene como cota máxima el número de clientes en la panadería. Ya que un cliente al coger y soltar el ticket incrementa en 1 la diferencia `last - next`, y esto puede hacerse infinitamente.

11 - Utiliza la abstracción anterior para comprobar la no existencia de bloqueos y la exclusión mutua utilizando el comando search.

Primero que nada, comprobamos si existe algun estado de bloqueo mediante el siguiente comando:

```
Maude> search initial(5) =>! S:GBState . search in ABSTRACT-BAKERY+
: initial(5) =>! S:GBState .

No solution.       states: 651  rewrites: 531636 in 89ms cpu (89ms
real) (5916664 rewrites/second)
```

Como podemos observar, se ha hecho uso del comando *=>!* el cual nos debería mostrar una configuracion de bloqueo, pero no hay soluciones, por lo que no hay estados de bloqueo.

Comprobamos la exclusión mútua para 5 procesos:

```
Maude> search [1] initial(5) =>* [[C < N:Nat : BProcess | mode:
crit, number: M:Nat > < N':Nat : BProcess | mode: crit,number:
M':Nat >]] such that N:Nat =/= N':Nat . search [1] in ABSTRACT-BAKERY+
: initial(5) =>* [[C < N : BProcess | mode: crit,number: M:Nat >
< N' : BProcess | mode: crit,number: M':Nat >]] such that N =/=
N' = true .

No solution.       states: 651  rewrites: 531636 in 89ms cpu (91ms
real) (5923586 rewrites/second)
```

Como se puede observar, no se encuentra ningún contraejemplo.

Ejercicio 3: El lenguaje PARALLEL++

12 - Comprueba utilizando el comando search la ausencia de bloqueo y la exclusión mutua de esa versión del algoritmo

Comprobamos ausencia de bloqueos:

```
Maude> search initial =>! MS:MachineState .
search in DEKKER++ : initial =>! MS:MachineState .

No solution.
states: 206  rewrites: 1971 in 6ms cpu (5ms real) (296747 rewrites/second)
```

Como podemos ver no hay estados de bloqueo.

Comprobamos exclusión mútua:

```
Maude> search initial =>* {S | [1,crit ; R] | [2, crit ; P], M} .
search in DEKKER++ : initial =>* {S | [1,crit ; R] | [2,crit ; P],M} .

No solution.
states: 206  rewrites: 1971 in 3ms cpu (2ms real) (592603 rewrites/second)
```

Como podemos ver hay exclusión mútua, ya que no hay estados en los que ambos procesos estén el la sección crítica.