# La panadería de Lamport y el lenguaje PARALLEL++

Jean Edouard Roy Caro & Tom van Greevenbroek

## Ejercicio 1: La panadería de Lamport

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

Nuestro sistema es confluente ya que no tenemos ecuaciones, aparte de la *initial* y su auxiliar *initialAux*, por lo que solamente puede seguir un camino de reducción. Es no terminante ya que podemos seguir aplicando reglas y se observa que al hacer *rew initial(5)* nunca termina.

```
Maude> rew initial(5) .
rewrite in BAKERY-CHECK : initial(5) .
```

Es coherente, porque no tenemos ecuaciones que se aplican a nuestro estado inicial, que está en forma irreductible, por lo que solo se aplican reglas ya que ninguna operación hace matching, por lo que al no intercalarse no pueden ser no-coherentes.

### [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.

El espacio de búsqueda no es el finito, porque al ser no terminante, nunca llegamos al final del espacio de búsqueda. Esto se debe a que el dispenser no tiene un número límite de tickets, por lo que los clientes pueden ir entrando, ser atendidos e irse indefinidamente.

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

En este caso ejecutamos el comando *search 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 .* , el cual busca si dos procesos están en modo crítico al mismo tiempo. En este caso, no parece encontrar ningún momento en el que esa condición se cumpla, pero al tener un espacio de búsqueda **infinito**, y por lo tanto, un programa no terminante, tampoco podemos afirmar al 100% que exista exclusión mutua.

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

```
Maude> load bakery.maude
Maude> search initial(5) =>! S:GBState .
search in BAKERY : initial(5) =>! S:GBState .
Debug(1)>
```

Utilizando el comando *search initial(5) =>! G:GBState .* vemos que de nuevo el programa se queda en un bucle infinito. Y volvemos a lo mismo que antes, aunque parezca que no habrá

nunca un estado de bloqueo, como es no terminante, no podemos garantizar que en algún momento pueda haber un estado de bloqueo.

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

Hay protección de booleanos ya que no redefinimos true y false. En cuanto a la confluencia, tenemos una operación *decrement* que es conmutativa de tipo *Configuration*, por lo tanto, se puede simplificar de maneras distintas, pero realmente, ambas van a terminar de simplificarse de la misma manera.

```
op decrement : Configuration -> Configuration .
eq decrement (none) = none .
ceq decrement (< P : BProcess | mode: M, number: s N > C) = < P : BProcess | mode: wait, number: N > decrement (C) if M =/= sleep .
eq decrement (< P : BProcess | Atts > C) = < P : BProcess | Atts > decrement (C) [owise] .
```

Ahora, esta nueva abstracción se convierte en terminante, ya que no llegamos a un espacio infinito como antes. Ya que de la forma en la que hemos hecho la ecuación *decrement*, que se va llamando de forma recursiva, pero que cuando se queda sin elementos, esta se termina en *decremente (none) = none*.
El nuevo módulo es coherente, ya que, las ecuaciones que hemos añadido no alteran el ciclo de un cliente a la hora de ser atendido por el panadero.

### [Q5] 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 es finito, de hecho, si ejecutamos el comando *search initial(5) =>* G:GBState .* obtenemos 651 estados finales.

```
Solution 651 (state 8333)
states: 8334  rewrites: 669543 in 6406ms cpu (18076ms real) (104514 rewrites/second)
G:GBState --> [[decrement(decrement(decrement(decrement(decrement(decrement(< 4 : BProcess | mode: sleep,number: 0 >) <
    3 : BProcess | mode: sleep,number: 0 >) < 2 : BProcess | mode: sleep,number: 0 >) < 1 : BProcess | mode: sleep,
    number: 0 >) < 0 : BProcess | mode: sleep,number: 0 >) < 0 : BProcess | mode: sleep,number: 0 >) < 0 : Dispenser |
    next: 1,last: 1 >]]

No more solutions.
states: 8334  rewrites: 669543 in 6406ms cpu (18097ms real) (104514 rewrites/second)
```

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

```
*** Operations
op processIn : Nat Mode -> Prop .
op notInMutex : -> Prop .

*** Equations
eq [[ < P : BProcess | mode: M, number: N > C ]] |= processIn(P, M) = true .
eq [[ < P : BProcess | mode: crit, number: N >
      < P' : BProcess | mode: crit, number: N' > C ]] |= notInMutex = true .
```

Hemos definido unas propiedades para comprobar que no existen dos procesos en estado crítico al mismo tiempo, para ello ejecutamos el comando *red modelCheck(initial(5), [] ~ notInMutex) .*

```
Maude> red modelCheck(initial(5), [] ~ notInMutex) .
reduce in BAKERY-CHECK : modelCheck(initial(5), []~ notInMutex) .
rewrites: 669550 in 2000ms cpu (2000ms real) (334775 rewrites/second)
result Bool: true
```

Como podemos comprobar, la propiedad nos devuelve un true, eso significa que la condición de que dos procesos no estén en estado crítico a la vez, se cumple.

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

```
Maude> red modelCheck(initial(5), [] (processIn(1, wait) -> <> processIn(1, crit)
/\ (processIn(1, crit) -> <> processIn(1, sleep))
/\ (processIn(1, sleep) -> <> processIn(1, wait))
/\ processIn(2, wait) -> <> processIn(2, crit)
/\ (processIn(2, crit) -> <> processIn(2, sleep))
/\ (processIn(2, sleep) -> <> processIn(2, wait))
/\ processIn(3, wait) -> <> processIn(3, crit)
/\ (processIn(3, crit) -> <> processIn(3, sleep))
/\ (processIn(3, sleep) -> <> processIn(3, wait))
/\ processIn(4, wait) -> <> processIn(4, crit)
/\ (processIn(4, crit) -> <> processIn(4, sleep))
/\ (processIn(4, sleep) -> <> processIn(4, wait))
/\ processIn(5, wait) -> <> processIn(5, crit)
/\ (processIn(5, crit) -> <> processIn(5, sleep))
/\ (processIn(5, sleep) -> <> processIn(5, wait))
)) .
reduce in BAKERY-CHECK : modelCheck(initial(5), [](processIn(1, wait) -> processIn(2, wait) /\ (<> processIn(1,
    crit) /\ (processIn(1, crit) -> <> processIn(1, sleep)) /\ (processIn(1, sleep) -> <> processIn(1, wait))) ->
    processIn(3, wait) /\ (<> processIn(2, crit) /\ (processIn(2, crit) -> <> processIn(2, sleep)) /\ (processIn(2,
    sleep) -> <> processIn(2, wait))) -> processIn(4, wait) /\ (<> processIn(3, crit) /\ (processIn(3, crit) -> <>
    processIn(3, sleep)) /\ (processIn(3, sleep) -> <> processIn(3, wait))) -> processIn(5, wait) /\ (<> processIn(4,
    crit) /\ (processIn(4, crit) -> <> processIn(4, sleep)) /\ (processIn(4, sleep) -> <> processIn(4, wait))) -> <>
    processIn(5, crit) /\ (processIn(5, crit) -> <> processIn(5, sleep)) /\ (processIn(5, sleep) -> <> processIn(5,
    wait)))) .
rewrites: 802514 in 13156ms cpu (13240ms real) (60998 rewrites/second)
result Bool: true
```

Utilizando el comando que se puede ver en la imagen, lo que hacemos es comprobar si existe siempre un estado siguiente. Como podemos comprobar, nos devuelve un true, por lo tanto como siempre hay un estado siguiente al que ir, esto significa que no hay estados de bloqueo.

## Ejercicio 2: La panadería modificada

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

El nuevo sistema sigue siendo confluente. Hemos añadido de nuevo nuevas operaciones y ecuaciones, pero solamente son llamadas por el conjunto de reglas. En cuanto a la terminación, a diferencia de ABSTRACT-BAKERY, esta modificación vuelve a convertir el sistema en no terminante. Si ejecutamos el comando *rew initial(5) .* podemos ver que el programa se queda en un bucle infinito.

```
Maude> rew initial(5) .
rewrite in BAKERY+ : initial(5) .
Debug(1)>
```

Por lo tanto, el sistema es no terminante. El sistema, al igual que antes, sigue siendo coherente porque cuando creamos un estado inicial, todos los estados se encuentran en una forma irreductible.

**[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.**

```
Maude> search initial(5) =>* [[ < O:Oid : BProcess | mode: crit, number: N:Nat > < O2:Oid : BProcess | mode: crit, numbe
r: M:Nat > C:Configuration ]] .
search in BAKERY+ : initial(5) =>* [[C < O:Oid : BProcess | mode: crit,number: N > < O2:Oid : BProcess |
    mode: crit,number: M:Nat >]] .
Debug(1)>
```

Podemos intentar comprobarlo con el comando *search initial(5) =>* [[ < O:Oid : BProcess | mode: crit, number: N:Nat > < O2:Oid : BProcess | mode: crit, number: M:Nat > C:Configuration ]] .* . Volvemos al mismo problema que en el ejercicio 1, al ser un sistema no terminante, no podemos comprobar la exclusión mutua del sistema. Eso no quiere decir que no tenga exclusión mutua, pero con el comando search, no podemos afirmarlo.

**[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+.**

El módulo ABSTRACT-BAKERY no es suficiente para este sistema porque ahora un proceso puede abandonar antes de que le toque su turno, por lo que el dispenser se quedaría esperando a un cliente que nunca llegará si no cambiamos los números.

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

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

No solution.
states: 486  rewrites: 1005578 in 1125ms cpu (1142ms real) (893847 rewrites/second)
```

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

No solution.
states: 486  rewrites: 1005578 in 1109ms cpu (1119ms real) (906436 rewrites/second)
```

Para comprobar la no existencia de bloqueos y de la exclusión mutua hemos usado dos comandos: *search initial(5) =>! G:GBState .* y *search initial(5) =>* [[< N:Nat : BProcess | mode: crit, number: M:Nat > < N':Nat : BProcess | mode: crit,number: M':Nat >]] such that N:Nat =/= N':Nat .* Como podemos comprobar, en ningún caso nos da alguna solución, por lo tanto podemos afirmar que la nueva abstracción es válida y no genera bloqueos, además de asegurar una exclusión mutua ya que en ningún momento hay dos BProcess en estado crítico.

## Ejercicio 3: El lenguaje PARALLEL++

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

Para comprobar la no existencia de bloqueos usamos el comando *search initial =>! MS:MachineState .* y para comprobar que se cumple la exclusión mutua usamos el comando *search initial =>* {S | [1,crit1 ; R] | [2,crit2 ; P],M} .*

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

No solution.
states: 170  rewrites: 1856 in 0ms cpu (12ms real) (~ rewrites/second)
Maude> search initial =>* {S | [1,crit1 ; R] | [2,crit2 ; P],M} .
search in DEKKER++ : initial =>* {S | [1,crit1 ; R] | [2,crit2 ; P],M} .

No solution.
states: 170  rewrites: 1856 in 0ms cpu (10ms real) (~ rewrites/second)
```

Como podemos ver, con el lenguaje modificado de parallel se cumple la exclusión mutua y la ausencia de bloqueos.