Trabajo práctico N° 3

Abril 2025

Estudiante Emanuel Nicolás Herrador

Ejercicio 2

El modelo a considerar para el problema de los filósofos es el siguiente:

```
FORK = (get -> put -> FORK).

PHIL(I=0) = (when(I%2 == 0) sitdown -> left.get -> right.get ->
                            eat -> left.put -> right.put -> arise -> PHIL
            |when(I%2 == 1) sitdown -> right.get -> left.get ->
                            eat -> left.put -> right.put -> arise -> PHIL
            ).

||DINERS(N=5) = forall[i:0..N-1]
                  (phil[i]:PHIL(i) || {phil[i].left, phil[((i-1)+N)%N].right}::FORK).
```

Ahora, si queremos agregar un proceso que al componerlo solo permita que en cada momento haya como máximo 4 filósofos en la mesa, podemos plantear lo siguiente:

```
BUTLER(N=4) = BUTLER[N],
BUTLER[i:0..N] = (when(i < N) arise -> BUTLER[i+1]
                 |when(i > 0) sitdown -> BUTLER[i-1]
                 ).

||NEW_DINERS(N=5) = (DINERS(N) || BUTLER(4))
                    /{phil[0..N-1].sitdown/sitdown,
                      phil[0..N-1].arise/arise
                     }.
```

Y el hecho de que no tiene deadlock se puede corroborar en LTSA. Además, esto resulta trivial de notar dado que la anterior implementación no tenía deadlock y lo único que se agrega ahora es una limitante de la cantidad de filósofos que puede haber en simultáneo. Es decir, se restringen las acciones de los filósofos, pero no se cambian ni se agregan nuevas.

Ejercicio 3

Ahora queremos ver que la intersección de propiedades de safety es una propiedad de safety. Para ello, si recurrimos a la definición, sabemos que $P \subseteq \Sigma^\omega$ es una propiedad de safety si:

$$\forall \sigma : \sigma \notin P \Rightarrow \exists i \geq 0 : \forall \beta : \sigma[..i]\beta \notin P \tag{3.1}$$

Por ello, sean $P, Q \subseteq \Sigma^\omega$ propiedades de safety, veamos que $\forall \sigma \notin P \cap Q$ se cumple que $\sigma \notin P \wedge \sigma \notin Q$. Con ello, por (3.1) tenemos que:

$$\exists i \geq 0 : \forall \beta : \sigma[..i]\beta \notin P$$
$$\exists j \geq 0 : \forall \beta : \sigma[..j]\beta \notin Q$$

Si consideramos la primera, es sencillo notar que $P \cap Q \subseteq P$ y que $\forall \gamma : \gamma \notin P \Rightarrow \gamma \notin P \cap Q$. Por ello, entonces, se llega a que $\exists i \geq 0 : \forall \beta : \sigma[..i]\beta \notin P \cap Q$. Como está hecho para un σ en general que no se encuentra en la intersección, entonces tenemos que:

$$\forall \sigma : \sigma \notin P \cap Q \Rightarrow \exists i \geq 0 : \forall \beta : \sigma[..i]\beta \notin P \cap Q$$

Y, como esta es una definición de propiedad de safety por (3.1), llegamos a que $P \cap Q$ es de safety por lo que se demuestra. ■

Notar que esto significa que safety es cerrado por intersección.

Ejercicio 4

Ahora queremos ver si liveness es cerrado por intersección. Para ello, vamos a ver por contraejemplo que esto no es así.

Supongamos $\Sigma = \{a, b\}$, $P = (a+b)^* a^\omega$ y $Q = (a+b)^* b^\omega$. Dado que nuestra definición de liveness es la siguiente:

$$\forall a \in \Sigma^* : \exists \beta \in \Sigma^\omega : \alpha\beta \in P \tag{4.1}$$

resulta trivial ver que P y Q son propiedades de liveness.

De igual modo, resulta trivial ver que $\emptyset$ **no** es una propiedad de liveness. Por ello, como $P \cap Q = \emptyset$ vemos que la intersección de dos propiedades de liveness no necesariamente es otra propiedad de liveness. Con ello, se da el contraejemplo de la afirmación y se demuestra que no se cumple. ■

Notar que esto significa que, al contrario que safety, liveness no es cerrado por intersección.

Ejercicio 5

Si queremos ver una propiedad que sea simultáneamente de safety y de liveness, podemos notar que es $P = \Sigma^\omega$. Esto resulta directo de ver las definiciones (3.1) y (4.1).

Ejercicio 6

Ahora, queremos ver que la propiedad anterior es la única que cumple ser tanto de safety como de liveness. Vamos a demostrarlo por el absurdo.

Digamos que $P \subsetneq \Sigma^\omega$ es una propiedad tanto de safety y liveness, y supongamos $Q = P - \Sigma^\omega$. Como P es de safety y de liveness, por (3.1) y (4.1) tenemos que:

$$\begin{gathered} \forall \sigma : \sigma \notin P \Rightarrow \exists i \geq 0 : \forall \beta : \sigma[..i]\beta \notin P \\ \forall a \in \Sigma^* : \exists \beta \in \Sigma^\omega : \alpha\beta \in P \end{gathered} \tag{1}$$

Como $P \neq \Sigma^\omega$, entonces $Q \neq \emptyset$. Digamos $\sigma \in Q$. Como $\sigma \notin P$, por safety tenemos que $\exists i \geq 0 : \forall \beta : \sigma[..i]\beta \notin P$. Digamos que tenemos ese i, entonces $\forall \beta : \sigma[..i]\beta \notin P$.

Sin embargo, por liveness, tenemos que como $\sigma[..i] \in \Sigma^*$, entonces $\exists \beta : \sigma[..i]\beta \in P$. Con ello, llegamos a una contradicción que vino de suponer que P es tanto de safety como de liveness.

Como se hizo en general para cualquier propiedad P que no sea el universo, se demuestra que Σ^ω es la única propiedad que es tanto de safety como de liveness. ■.

Ejercicio 7

Es trivial ver que el complemento de una propiedad de safety **no necesariamente** es una propiedad de liveness dado que Σ^ω es de safety pero $\emptyset$ no es de liveness.

Ejercicio 8

Considerando $\Sigma = \{a, b\}$, veamos cada una de las propiedades para catalogarlas (no consideramos que puedan ser ambas dado que solo el universo puede serlo):

- $a^* b^\omega$: No es de safety porque a^ω no cumple la propiedad pero para todo prefijo si lo concatenamos con b^ω sí cumple la propiedad. No es de liveness porque el prefijo aba no se puede arreglar.
- $(b+a)^+ b^\omega$: Es de liveness porque para todo prefijo, si se concatena con b^ω entonces cumple la propiedad.
- $a^* b^+ a^\omega$: No es de safety porque b^ω no cumple la propiedad pero para todo prefijo si lo concatenamos con a^ω sí la cumple. No es de liveness porque el prefijo bab no se puede arreglar.
- $(a+b)^*(a^\omega + b^\omega)$: Es de liveness porque para todo prefijo posible, si lo concatenamos con a^ω o con b^ω, cumple la propiedad.
- $(ab)^\omega$: Es de safety porque las trazas se pueden caracterizar del siguiente modo: $\{\sigma \in \Sigma^\omega : \forall i \geq 0 : \sigma(2i) = a \wedge \sigma(2i+1) = b\}$. Luego, el complemento de la propiedad es $\{\sigma \in \Sigma^\omega : \exists i \geq 0 : \sigma(2i) = b \vee \sigma(2i+1) = a\}$, lo cual puede expresarse en ω-regular como $(ab)^*(b + (a+b)a)(a+b)^\omega$. Por ello, es sencillo notar que si una traza pertenece al complemento de la propiedad, entonces para ese prefijo malo no existe ninguna traza que pueda "arreglarlo", lo que significa que se cumple la implicación de safety.

- $(ab)^*a^\omega$: No es de safety porque $(ab)^\omega$ no cumple la propiedad pero para todo prefijo si lo concatenamos con a^ω sí cumple la propiedad. No es de liveness porque el prefijo b no se puede arreglar.

Ejercicio 9

Para a^*b^ω solo hace falta agregar la traza a^ω. Es decir, la reformulamos como $a^*(a^\omega + b^\omega)$ de modo que el prefijo malo es $(a+b)^*ba$.

Para $(b+a)^+b^\omega$ se puede notar que todo prefijo se puede extender concatenando b^ω para que cumpla la propiedad. Por ello, ninguna extensión podrá hacer que esta propiedad sea de safety salvo que sea el universo mismo. Es decir, lo extendemos a Σ^ω.

Para $a^*b^+a^\omega$ solo hace falta agregar las trazas a^ω y b^ω, de modo que el prefijo malo sea $(a+b)^*ba^+b$.

Para $(a+b)^*(a^\omega + b^\omega)$, del mismo modo que antes, se debe extender al universo Σ^ω al ser una propiedad de liveness.

Para $(ab)^*a^\omega$, debemos extenderlo con la traza $(ab)^\omega$ de modo que los prefijos malos sean $(ab)^*b$ y $(ab)^*a^+ab$.

Ejercicio 10

Para a^*b^ω, tenemos que $P = (a^*b^\omega) + a^\omega$ es de safety por el ejercicio anterior y que $Q = (a^*b^\omega) + (a+b)^*b^\omega$ es de liveness porque para todo prefijo, si lo concatenamos con b^ω cumple Q. Luego, es claro que $P \cap Q = a^*b^\omega$.

Para $a^*b^+a^\omega$, tenemos que $P = a^*b^+a^\omega + a^\omega + b^\omega$ es de safety por el ejercicio anterior y que $Q = a^*b^+a^\omega + (a+b)^*(ab)^\omega$ es de liveness porque para todo prefijo si lo concatenamos con $(ab)^\omega$ entonces cumple Q. Además, es claro que $P \cap Q = a^*b^+a^\omega$.

Para $(ab)^*a^\omega$, tenemos que $P = (ab)^*a^\omega + (ab)^\omega$ es de safety por el ejercicio anterior y que $Q = (ab)^*a^\omega + (a+b)^*b^\omega$ es de liveness porque para todo prefijo si lo concatenamos con b^ω entonces cumple Q. Además, es claro que $P \cap Q = (ab)^*a^\omega$.

Ejercicio 11

Notar que la propiedad de estas trazas puede expresarse en el lenguaje ω-regular como $(aa + ab + ba)^\omega$. Por ello, es trivial ver que toda traza de la forma $(aa + ab + ba)^*bb(a+b)^\omega$ no cumple la propiedad.

Ejercicio 13

El modelo queda bastante similar a la propiedad:

```
const MAX = 10

ELEVATOR = ELEVATOR[0],
ELEVATOR[i:0..MAX] = (when(i > 0) exit -> ELEVATOR[i-1]
                     |when(i < MAX) enter -> ELEVATOR[i+1]
                     ).

property CONTROL = CONTROL[0],
CONTROL[i:0..MAX] = (when(i > 0) exit -> CONTROL[i-1]
                    |when(i < MAX) enter -> CONTROL[i+1]
                    ).

||SYSTEM = (ELEVATOR || CONTROL).
```

Ejercicio 14

El modelo según cómo se planteó el protocolo es el siguiente:

```
/* Model neighbor's protocol */
FLAG = FLAG[0],
FLAG[b:0..1] = (when(!b) raise -> FLAG[1]
               |when(b) lower -> FLAG[0]
               | [b] -> FLAG[b]
               ).
```

```
NEIGHBOR = (raise_flag -> (see_other_flag -> lower_flag -> NEIGHBOR
                          |doesnt_see_other_flag -> enter_field ->
                              get_blackberries -> exit_field -> lower_flag -> NEIGHBOR
                          )
           ).

||PROTOCOL = (flag[1..2]:FLAG || neighbor[1..2]:NEIGHBOR)
             /{neighbor[i:1..2].raise_flag/flag[i].raise,
               neighbor[i:1..2].lower_flag/flag[i].lower,
               neighbor[i:1..2].see_other_flag/flag[!(i-1)+1].[1],
               neighbor[i:1..2].doesnt_see_other_flag/flag[!(i-1)+1].[0]
               }.

/* Check safety and liveness */
property FIELD = FIELD[0],
FIELD[i:0..1] = (when(!i) enter -> FIELD[i+1]
                |when(i) exit -> FIELD[i-1]
                ).

progress PN1 = {neighbor[1].enter_field}
progress PN2 = {neighbor[2].enter_field}

||SYSTEM_CHECKS = (PROTOCOL || field:FIELD)
                  /{neighbor[i:1..2].enter_field/field.enter,
                    neighbor[i:1..2].exit_field/field.exit
                   }.
```

Si lo analizamos en LTSA, podemos notar que ambas propiedades (safety y liveness) se cumplen. Sin embargo, hay ciertos escenarios donde no se cumpliría liveness si no estamos bajo una estrategia de scheduling equitativa como cuando ninguno puede entrar si ambos son *solidarios*. Es decir, si cada vez que ven que la bandera del otro está alta, eligen bajar la suya, ninguno podrá entrar nunca dado que las trazas serán de la forma:

```
neighbor[1].raise_flag
neighbor[2].raise_flag
neighbor[1].see_other_flag
neighbor[2].see_other_flag
neighbor[1].lower_flag
neighbor[2].lower_flag
...
```

Y, si hipotéticamente uno fuera siempre egoísta, jamás bajaría su bandera. Es decir, es un proceso que mientras vea la bandera del otro vecino deja su bandera en alta esperando a que el otro la baje. En caso que los dos sean egoístas, no existe progreso por parte de ninguno. Y en caso de que solo uno sea egoísta, evita que el otro vecino pueda progresar.

Ejercicio 15

Ahora, para poder solucionar el problema de progreso y de deadlock potencial, se utilizará el truco clásico de asimetría otorgando turnos a los vecinos. Para ello, pondremos una flag que sea un turno y cada vez que uno alce su bandera, cede el turno al otro.

Con esto en mente, ahora el modelo nos queda así:

```
/* Model neighbor's protocol */
FLAG = FLAG[0],
FLAG[b:0..1] = (when(!b) raise -> FLAG[1]
               |when(b) lower -> FLAG[0]
               |[b] -> FLAG[b]
               ).

TURN = TURN[1],
```

```
TURN[i:1..2] = (change[1] -> TURN[1]
               |change[2] -> TURN[2]
               |[i] -> TURN[i]
               ).

NEIGHBOR = (raise_flag -> give_turn -> WAIT),
WAIT = (doesnt_see_other_flag -> ENTER
       |is_my_turn -> ENTER
       ),
ENTER = (enter_field -> get_blackberries -> exit_field -> lower_flag -> NEIGHBOR).

||PROTOCOL = (flag[1..2]:FLAG || neighbor[1..2]:NEIGHBOR || turn:TURN)
             /{neighbor[i:1..2].raise_flag/flag[i].raise,
               neighbor[i:1..2].lower_flag/flag[i].lower,
               neighbor[i:1..2].doesnt_see_other_flag/flag[!(i-1)+1].[0],
               neighbor[i:1..2].give_turn/turn.change[!(i-1)+1],
               neighbor[i:1..2].is_my_turn/turn.[i]
               }
             \ {flag[1..2].{[0], [1]}}.

/* Check safety and liveness */
property FIELD = FIELD[0],
FIELD[i:0..1] = (when(!i) enter -> FIELD[i+1]
                |when(i) exit -> FIELD[i-1]
                ).

progress PN1 = {neighbor[1].enter_field}
progress PN2 = {neighbor[2].enter_field}

||SYSTEM_CHECKS = (PROTOCOL || field:FIELD)
                  /{neighbor[i:1..2].enter_field/field.enter,
                    neighbor[i:1..2].exit_field/field.exit
                   }.
```

Aquí sí se cumplirían las propiedades de liveness dado que si un vecino quiere ingresar al campo de moras, se garantiza que en algún momento podrá hacerlo aunque el otro sea egoísta.

Ejercicio 16

El modelo es el siguiente:

```
MAZE(N=0) = MAZE[N],
MAZE[0] = (north -> STOP
          |east -> MAZE[1]
          ),
MAZE[1] = (east -> MAZE[2]
          |south -> MAZE[4]
          |west -> MAZE[0]
          ),
MAZE[2] = (south -> MAZE[5]
          |west -> MAZE[1]
          ),
MAZE[3] = (east -> MAZE[4]
          |south -> MAZE[6]
          ),
MAZE[4] = (north -> MAZE[1]
          |west -> MAZE[3]
          ),
MAZE[5] = (north -> MAZE[2]
          |south -> MAZE[8]
          ),
```

```
MAZE[6] = (north -> MAZE[3]),
MAZE[7] = (east -> MAZE[8]),
MAZE[8] = (north -> MAZE[5]
          |west -> MAZE[7]
          ).
```

Y, dependiendo cuál sea el parámetro, el análisis de deadlock nos dirá el camino mínimo para resolver el laberinto.