Ingeniería del Software 2

Primer Parcial

Ejercicio 1

Un sistema multi-core implementa un mecanismo de reserva de recursos para tareas que quieren realizar cómputos.

Se implementa un módulo `Asignador` de recursos al que las tareas le solicitan la cantidad de recursos que requieren (ej., `get[3]`) y luego, una vez utilizados, informan que los devuelven (ej., `put[3]`). Si un proceso solicita una cantidad de recursos mayor a la disponible, este deberá esperar a que estén disponibles.

Las tareas que solicitan recursos pueden ser de tipo A o tipo B. Los de tipo A piden entre 2 y `N` cores. Los de tipo B sólo pueden pedir entre 1 y 2 cores. Cada tarea anuncia primero cuantos cores va a usar (`start[n]`), pide los recursos que necesita, trabaja (pero no hay comportamiento observable), devuelve los recursos, anuncia que terminó (`end`) y recomienza. El número de cores que solicita cada vez puede diferir.

a) Modelar usando FSP el proceso `Asignador`,

b) Modelar usando FSP el proceso genérico `Tarea(TIPO=1)` donde TIPO puede ser 1 o 0 dependiendo si es de tipo A o B.

c) Crear un sistema multi-core con un `Asignador` y cuatro `Tareas`, mitad de tipo A y la otra mitad de tipo B. Utilice `::` y `:` apropiadamente.

d) Modelado correctamente, el sistema puede exhibir problemas de liveness? Explique cual es y ejemplifique con una traza de la composición. Cómo podría detectarlo automáticamente con la herramienta MTSA?

e) Corrija el problema de liveness modificando `Tarea` y `Asignador` para que usen una ticketeadora (ver abajo).

   Cada `Tarea` debe pedir un ticket (`ticket[t:T]`) antes de pedir los recursos. Luego la `Tarea` pide el recurso usando el ticket recibido. Por ejemplo, si recibió el ticket 2 y quiere 3 recursos entonces hace `get[3][2]`.

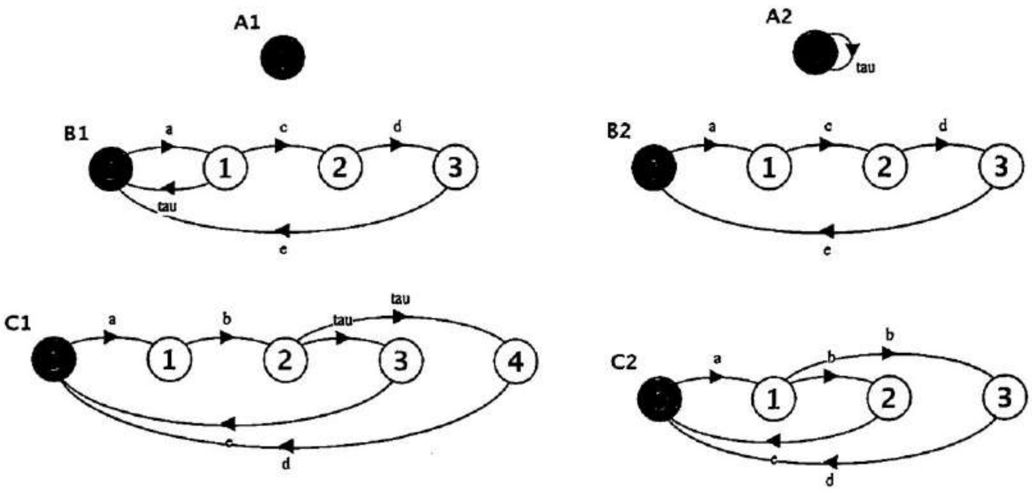
   El `Asignador` asigna recursos en por orden de tickets.

```
const TM = 5 // TM es más grande que el número de tareas del sistema
range T = 1..TM
TICKETEADORA = NEXT[1],
NEXT[t:T] = (ticket[t] ->NEXT[t%TM+1]).
```

Ejercicio 2

a) Para los siguientes pares de LTS, indique cuáles son débilmente bisimilares. Justifique mostrando una relación o un contraejemplo.

b) Considerando que la denotación de una acción *tau* es la ejecución de cómputo interno y que un estado del que no hay transiciones salientes suele interpretarse como un deadlock o el final de un cómputo, qué opinión le merece el resultado de comprarar por bisimulación A1 contra A2?

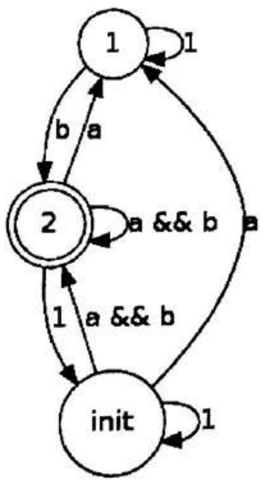


Ejercicio 3

Pruebe la siguiente ley de absorción de LTL: $\Box(\Diamond(\Box(p))) = \Diamond(\Box(p))$

Ejercicio 4

Indique una fórmula LTL que crea que caracteriza las mismas trazas que el siguiente autómata de Buchi. Justifique.



Ejercicio 5

Cómo usaría los algoritmos/herramientas vistas en clase para, dadas dos fórmulas LTL F1 y F2, verificar automáticamente que toda traza que satisface F1 satisface necesariamente F2. Justifique

[redacted]

19/09/19

① | 2 | 2 | 2 | 2 | 2 | (10)

a)

```
Asignador(R=4) = ASIG[R],
ASIG[i:0..R] = (get[j:1..i] -> ASIG[i-j] |
    put[j:1..R-i] -> ASIG[i+j]).
```

b)

```
const A = 1
const B = 0
Tarea(TIPO=A) = (
    when (TIPO==A) start[i:2..N] -> get[i] -> trabajo ->
        put[i] -> end -> Tarea
   | when (TIPO==B) start[i:1..2] -> get[i] -> trabajo ->
        put[i] -> end -> Tarea
   )\{trabajo}.
```

c)

```
||SYSTEM = (a[i:1..2]:Tarea(A) ||
            b[i:1..2]:Tarea(B) ||
            {a[1],a[2],b[1],b[2]}::Asignador).
```

d) El modelo de arriba no incluye una "cola de pedidos", por lo que si un proceso necesita más recursos de los disponibles, simplemente no los obtiene y no hay garantía de que se le den ni bien estén disponibles.

No hay progreso para, por ejemplo, la acción "a.1.end".

Una traza que evidencia esto es:

```
a1.start.4 → b1.start.2 → b1.start.2 →
    b1.get.2 → b1.put.2 → b1.end
    → b1.start.2 → b1.get.2 → b1.put.2 → b1.end
    → b1.start.2 → b1.get.2 → b1.put.2 → b.end.
    →            ''
    →            ''
    ⋮
```

Esto ocurre porque a1 se queda esperando a que se liberen 4 recursos, pero b1 vuelve a pedir 2 apenas libera 2, por lo que nunca llegan a liberarse 4 y a1 no puede avanzar.

Esto tiene que ver con el "scheduling", y para detectarlo deberíamos establecer prioridad de etiquetas (por ejemplo, darle prioridad a los "get", para que se asignen recursos apenas estén, a quien menos demande). Luego usar "check progress" de MTSA.

e)
```
Asignador (R=4) = ASIG[R][1],
ASIG[i:0..R][t:T] = (get[j:1..i][t] → ASIG[i-j][(t%TM+1)]
    | put[j:1..N-i] → ASIG[i+j][t]).

Tarea (TIPO=A) = ( ticket[t:T] → if (TIPO==A) then
        start[i:2..N] → get[i][t] → trabajo → put[i] → end → Tarea
    else
        start[i:1..2] → get[i][t] → trabajo → put[i] → end → Tarea).

||SYSTEM = ( a[i:1..2]:Tarea(A) || b[i:1..2]:Tarea(B) ||
        {a[1],a[2],b[1],b[2]}::TICKETEADORA ||
        {        ''         }::Asignador ).
```

(2) a) Decidir si son débilmente bisimilares:

A1 vs A2

Sí. La relación $R = \{(A1, A2)\}$ es una bisimulación débil. Para ver que lo es basta con verificar la única transición que hay: $A2 \xrightarrow{tau} A2$

- $A2 \xRightarrow{tau} A2 \Rightarrow \exists Q / A1 \xRightarrow{tau} Q \wedge (Q, 2)$

  Con $Q = A1$ vale.

- "Idem para $A1 \xRightarrow{tau} A1$".

B1 vs B2

No. La de abajo es una estrategia ganadora para el atacante (A), siendo D el defensor:

A: $B1.0 \xrightarrow{a} B1.1$

D: $B2.0 \xrightarrow{a} B2.1$ (única opción)

A: $B1.1 \xrightarrow{tau} B1.0$

D: $B2.1 \xRightarrow{tau} B2.1$ (única opción)

A: $B1.0 \xrightarrow{a} B1.1$

D: ¡No puede imitar!

C1 vs C2

No. Nuevamente doy una estrategia ganadora para el atacante:

A: $C1.0 \xrightarrow{a} C1.1$

D: $C2.0 \xrightarrow{a} C2.1$ (única opción)

A: $C1.1 \xrightarrow{b} C1.2$

Dos posibles jugadas de D.

D: $C2.1 \xrightarrow{b} C2.2$

A: $C1.2 \xrightarrow{tau} C1.4$

D: $C2.2 \xRightarrow{tau} C2.2$ (única op.)

A: $C1.4 \xrightarrow{d} C1.0$

D: ¡No puede imitar!

D: $C2.1 \xrightarrow{b} C2.3$

A: $C1.2 \xrightarrow{tau} C1.3$

D: $C2.3 \xRightarrow{tau} C2.3$ (única op.)

A: $C1.3 \xrightarrow{e} C1.0$

D: ¡No puede imitar!

b) Podría ser un resultado inesperado dependiendo la aplicación, ya que no es lo mismo un proceso en deadlock que uno que está computando internamente, y son débilmente bisimilares.

Notar que:

- La bisimilitud débil sirve bajo la perspectiva de un análisis puramente agnóstico al comportamiento interno del proceso, pero los procesos no son iguales y se comportan diferente frente a composiciones paralelas con otros procesos.

- A1 y A2 no son fuertemente bisimilares. Dicha relación logra capturar la diferencia que aquí se discute.

(3) $\square\Diamond\square p = \Diamond\square p$

$\Rightarrow$) Sé que si para σ vale $\sigma \models \square\Diamond\square p$, luego $\forall j \geq 0$ $\sigma[j] \models \Diamond\square p$. Particularmente para $j=0$, $\sigma[0] \models \Diamond\square p$. Esto es, $\sigma \models \Diamond\square p$. ✓ ✓

$\Leftarrow$) Sé que si para σ vale $\sigma \models \Diamond\square p$, luego $\exists j \geq 0$ / $\forall i \geq j$ $\sigma[i] \models p$. Quiero ver que $\sigma \models \square\Diamond\square p$, es decir que $\forall K \geq 0$ $\exists j' \geq K$ / $\forall i' \geq j'$ $\sigma[i] \models p$. Veamos que esto es cierto para cualquier $K \geq 0$:

- Si $K \leq j$ (al j que existe según $\sigma \models \Diamond\square p$), luego vale trivialmente con $j' = j$. ✓
- Si $K > j$, como sabemos que $\forall i \geq j$ $\sigma[i] \models p$, particularmente vale para aquellos $i \geq K > j$. Entonces, tomando $j' = K$ vale que $\forall i' \geq j'$ $\sigma[i] \models p$. ✓

Quod erat demonstrandum.

(4)

▸ $\varphi = (\Box\Diamond a) \wedge \Box(a \Rightarrow \Diamond b) = \Box\Diamond a \wedge \Box\Diamond b$

Para justificar la fórmula le doy semántica a los estados:

$\underline{\text{init}}$ ≡ "esperando a que ocurra 'a'"
$\underline{1}$ ≡ "ya ocurrió 'a', entonces espero a que ocurra 'b'"
$\underline{2}$ ≡ "ya ocurrió 'a', y luego (o junto con) ocurrió 'b'".

Para que el Büchi acepte una cadena hay que pasar infinitas veces por $\underline{2}$, el único de aceptación. Esto se logra cuando "ya ocurrió 'a', y luego (o al mismo tiempo) ocurrió 'b'"; esto es $\Box\Diamond a$ (ocurre 'a' infinitas veces) y $\Box(a \Rightarrow \Diamond b)$ (cada vez que ocurrió viene luego o junto una 'b').

(5) Dadas F1 y F2, ver que toda traza que cumpla F1 cumpla F2 es equivalente a ver que F1 $\Rightarrow$ F2 es tautología, que es equivalente a ver que $\neg$(F1 $\Rightarrow$ F2) es contradicción.
Si una fórmula LTL no es satisfacible, su Büchi asociado reconoce el lenguaje vacío.
Luego, podría verificarse la vacuidad del Büchi asociado a $\neg$(F1 $\Rightarrow$ F2) para verificar automáticamente lo pedido.