Ingeniería del Software 2

Taller 4 – (y Guía de Ejercicios) – Modelado de Procesos Concurrentes

Fecha límite de entrega de taller: jueves 19 de mayo a las 23:59hs

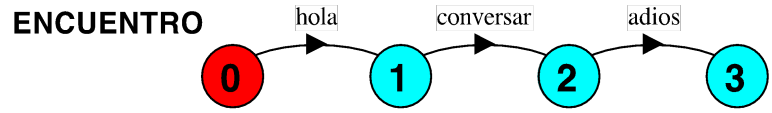
Aclaraciones

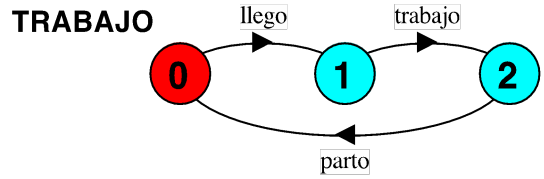
- Este documento unifica un taller entregable con ejercicios de práctica para el parcial.
- Los ejercicios de taller entregables están marcados.
- Sugerimos hacer los ejercicios en orden, es decir no saltear los no-entregables.
- Entregaremos soluciones a los ejercicios de este documento luego de la fecha de entrega del taller.
- La herramienta MTSA se encuentra en https://mtsa.dc.uba.ar. La herramienta se levanta corriendo
  $ java -jar https://bitbucket.org/lnahabedian/mtsa/downloads/mtsa-1.0-SNAPSHOT.jar
- La sintaxis de FSP puede encontrarse en https://www.doc.ic.ac.uk/~jnm/LTSdocumention/FSP-Syntax.html

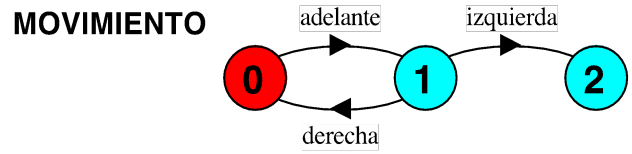
Ejercicios

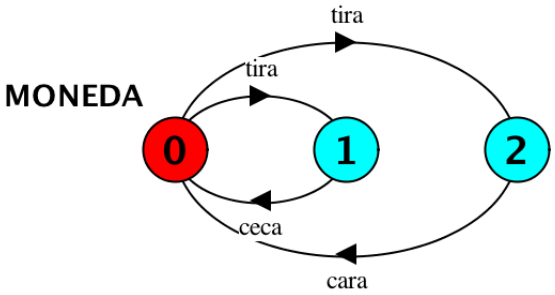
Ejercicio 1

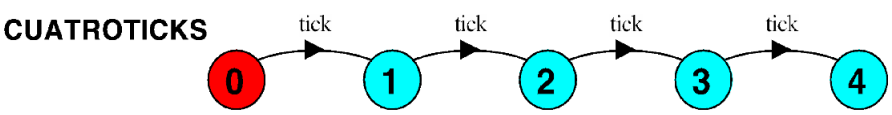
Para cada uno de los siguientes LTS, dar una expresión en FSP (lo más compacta posible) cuya semántica representa al LTS dibujado.

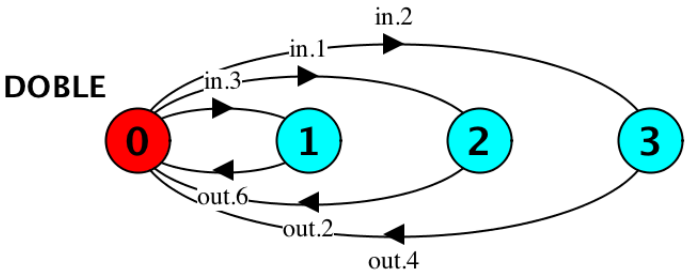











DOBLE
0
1
2
3
in.1
in.2
in.3
out.6
out.2
out.4

Ejercicio 2

Considere los procesos **P** y **Q** descritos en FSP y su composición paralela **S**:

```
P = (a -> b -> P).
Q = (c -> b -> Q).
||S = (P || Q).
```

a) Describa en FSP un proceso secuencial equivalente a **S**.
b) Muestre un proceso **R**, tal que compuesto con **S** *restrinja* el conjunto de trazas a las secuencias donde cada evento *a* sucede siempre antes que un evento *c* (i.e., trazas de la forma a,c,b,a,c,b,...).
c) Soponga un proceso **T** construido a partir de **P** pero con la acción `b` ocultada. Qué trazas exhibiría la composición de **T** con **Q**, es decir

```
||TQ = (T || Q).
```

Compruebelo usando MTSA.

Ejercicio 3

Una variable guarda valores entre 0..N y permite ser leída y escrita a través de eventos *read* y *write*. Aunque sea obvio, recordamos que cuando uno lee de una variable espera que el valor retornado sea el último que fue escrito. Modelar la variable como un proceso, **VARIABLE**, usando FSP. Considere que la variable empieza inicializada en 0.

Utilizar etiquetas *read.x* y *write.x* para denotar respectivamente el evento de haber leido x y el evento de haber escrito x en la variable.

Para N>2, el modelo debería exhibir, entre otras, la siguiente traza:

$$\text{write.2} \rightarrow \text{read.2} \rightarrow \text{read.2} \rightarrow \text{write.1} \rightarrow \text{write.0} \rightarrow \text{read.0} \rightarrow \ldots$$

Para N>2, el modelo **NO** debería exhibir la siguiente traza:

$$\text{write.2} \rightarrow \text{read.2} \rightarrow \text{read.1} \rightarrow \ldots$$

Para validar que el modelo que hizo se corrresponde con un modelo mental de una variable **utilice la funcionalidad de animación de MTSA** (el botón "A" sobre la derecha arriba).

Ejercicio 4

(Entregable)
Un sensor recibe señales o pulsos según los centímetros de agua acumulados en un depósito (*agua.0, ..., agua.10*). El sensor es capaz de responder a consultas sobre el nivel de agua (*nivel*) respondiendo *bajo*, *medio*, o *alto* según el último pulso recibido. El nivel es bajo cuando hay hasta 2cm, alto cuando hay desde 8cm, y medio en los demás casos.

El sensor no responde a consultas hasta que no hay recibido al menos un pulso.

- Modelar el sensor como un solo proceso FSP suponiendo que es suficientemente veloz en responder a una consulta *nivel* como para siempre responder antes de que llegue un nuevo pulso *agua.x*.
- Valide su modelo usando el animador de MTSA.
- Volver a modelar el sensor pero ahora sin ninguna suposición sobre la velocidad de respuesta del sensor. Es decir, entre un evento *nivel* y una respuesta, podría ocurrir otro evento *agua.x*.
- Valide su modelo usando el animador de MTSA.
- Para pensar pero no entregar: En el último modelo, puede ocurrir que nunca haya respuesta por parte del sensor? Cómo podría cambiarse el modelo para este tipo de trazas? .

Ejercicio 5

(Entregable)
En el ejercicio anterior modelamos un sensor que recibe señales de nivel de agua que no controla. Podría primero recibir *agua.0* y luego *agua.10* para finalmente recibir *agua.0* nuevamente.

En este ejercicio vamos a escribir un modelo que describe el comportamiento del entorno del sensor. Es decir describe el comportamiento del agua restringiendo como pueden suceder los eventos *agua.x*.

- Escribir en FSP un modelo para describir el comportamiento del agua con un proceso cuyo alphabeto es *agua.0,. . . , agua.10*. El comportamiento debe estar restringido de tal manera que el agua nunca sube tan rápido como para saltear un pulso de nivel (ej, no puede hacer *agua.0, agua.7, agua.3*). Modelar que el agua comienza en nivel 5.
- Valide el modelo del agua usando el animador de MTSA.
- Componer el modelo del agua con alguno de los modelos de sensor del ejercicio anterior. Validar el modelo mediante una animación que ya el sensor no recibe señales de agua due dan "saltos".

Ejercicio 6

Los modelos de sensores anteriores modelan un proceso que es interrumpido por eventos *agua.x* o tal vez, en un lenguaje de más alto nivel, algún mecanismo de *action listener*. A su vez el sensor anterior implementaba un patrón de *request-response* para informar niveles de agua a un usuario del sensor: El request es el evento *nivel* que un usuario utiliza para consultar el nivel del agua, el response al usuario es *bajo*, *medio* o *alto*.

Ahora modelaremos un proceso que funciona de manera opuesta. El usuario no le consulta el nivel sino que es interrumpido por un evento *alto* cuando el sensor advierte que el nivel del agua superó un umbral. Por otro lado el el sensor en vez de ser interrumpido por eventos *agua.x*, realiza *polling* o un *wait activo*: es decir el periodicamente, mediante un comando *sense*, "le consulta" a una báscula qué nivel tiene. La báscula responde con un evento *agua.x*

Valide el comportamiento del modelos animándolo.

Ejercicio 7

(Entregable)
Dado el siguiente proceso que describe el comportamiento de una variable:

```
range R = 0..7

VARIABLE = VARIABLE[3],
VARIABLE[i:R] = (
       read[i] -> VARIABLE[i] |
       write[j:R] -> VARIABLE[j] |
       write[8]-> overflow -> STOP |
       write[-1] -> underflow -> STOP).
```

Modelar los siguientes procesos:

a) Un proceso **SUMA1** que incrementa de a uno la variable.
b) Un proceso **RESTA1** que resta de a uno la variable.

Compónga los tres procesos y valide manualmente animándo la composición. Considere que la variable es un recurso compartido entre los procesos SUMA1 y RESTA1. Según como modele el problema, tal vez necesite hacer extensión de alphabetos.

Finalmente, en caso de que exista, ejemplifique con trazas casos en que:

- la variable da overflow.

- la variable da underflow.
- nunca sucede under u overflow.
- los procesos SUMA1 y RESTA1 se interfieren resultando en una actualización incorrecta de la variable.

Ejercicio 8

Un museo que puede albergar hasta N turistas, permite entrar a los turistas por la entrada oriental y salir por la salida occidental. Arribos y partidas se señalan al controlador del museo mediante señales *entry* y *exit* emitidas por molinetes. Cuando el museo debe abrir, el director del mismo le da la señal *open* al controlador y el controlador permite el ingreso y egreso de visitantes. A la hora de cerrar, el director da la señal *close* al controlador que a partir de ese momento solo permite egresos. Cuando el museo está vacío, el director recibe la señal *empty* del controlador. El director no reabre el museo hasta que el museo se encuentra sin turistas dentro.

a) Modele un proceso **ENTRADA** que represente la entrada al museo aceptando eventos *entry*.
b) Modele un proceso **SALIDA** que represente la salida del museo aceptando eventos *exit*.
c) Modele un proceso **DIRECTOR** que abra, cierre y espere a que se vacie el museo.
d) Modele un proceso **CONTROL** que se encargará de que todo el sincronizado funcione correctamente: primero espera a que el director abra el museo, luego deja entrar y salir gente siempre que sea permitido hasta que el museo cierra. En este momento solo deja salir a los turistas y eventualmente confirma estar vacío.
e) La composición de todos los anteriores.

Ejercicio 9

En este ejercicio vamos a trabajar con el problema clásico de los filósofos. Este problema ejemplifica el problema de deadlocks en procesos concurrentes que utilizan recursos compartidos.

- Componga el proceso y utilice la opción de menú "Check, Deadlock".

```
PHIL = (sitdown->right.get->left.get
          ->eat->left.put->right.put
          ->arise->PHIL).

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

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

- Una forma clásica de romper deadlocks es la introducción de asimetrías. Complete el siguiente modelo para evitar deadlocks en el problema de los filósofos.

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

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

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

- Otra solución al problema de los filósofos es tener un controlador que no permite que haya mas de N-1 filósofos comiendo simultáneamente. Modelar esta solución y probar que es libre de deadlock.

Ejercicio 10

(Entregable)

Una arquitectura *pipe and filter* es una forma de estructurar sistemas para procesar streams de datos de manera concurrente. Es una arquitectura que aparece frecuentemente en sistemas concurrentes y/o distribuidos. Los filtros son procesos que reciben como input uno o mas streams de datos, realizan una transformación, y tienen uno o mas streams de datos de salida. Aquí consideraremos sólo filtros con un input y un output. Los filtros son conectados mediante pipes. Los pipes son esencialmente buffers que redirigen el output de un filtro al input de otro.

Una familia de ejemplos clásicos de pipe y filter son los de unix:

```
ps aux | grep ltsa | grep -v grep | awk '{print $2}' | xargs kill
```

La arquitectura pipe and filter es muy apropiada para una implementación del algoritmo clásico de Eratostenes para el calculo de primos menores a N:

```
for (i=2; i<=N; i++) filtro[i]=i;
for (i=2; i<=N; i++) {
    if (filtro[i] != 0) {
print(i);
            for (j=i; j<=N; j++)
                      if (filtro[j]\%i == 0) filtro[j]=0;
    }
}
```

Acontinuación se muestra un modelo de una implementación pipe and filter para el cómputo de números primos. Consta de un primer filtro GEN que es un generador de numeros de 2 a N y que señala el fin del stream con EOS. Luego hay un filtro para cada primo entre 2 y N que filtra a los múltiplos del primo y re-envía los demás.

```
const MAX=15
range NUM=2..MAX
set S={[NUM],eos}

// Pipe process buffers elements from set S:
PIPE=(put[x:S]->get[x]->PIPE).

// GEN process outputs numbers from 2 to MAX followed by the signal eos:
GEN=GEN[2],
GEN[x:NUM]=(out.put[x]->
              if x<MAX then
                GEN[x+1]
                               else
                (out.put.eos->end->GEN)
            ).

 /*initialize from the first input from prev stage */
FILTER=(in.get[p:NUM] -> prime[p] -> FILTER[p]
       |in.get.eos -> ENDFILTER),
/*filter all inputs that are multiples of p*/
FILTER[p:NUM]=(in.get[x:NUM]->
                 if x%p!=0 then
                   (out.put[x]->FILTER[p])
                 else
                   FILTER[p]
               |in.get.eos->ENDFILTER
               ),
/*terminate filter on eos*/
ENDFILTER=(out.put.eos -> end -> FILTER).


||PRIMES(N=4)=
    (gen:GEN
     || pipe[0..N-1]:PIPE
     || filter[0..N-1]:FILTER)
    /{pipe[0]/gen.out,
      pipe[i:0..N-1]/filter[i].in,
      pipe[i:1..N-1]/filter[i-1].out,
      end/{filter[0..N-1].end,gen.end}}.
```

- Anime el proceso `PRIMES` para entender cómo funciona el sistema.
- Observe el término `PRIMES`. Representa la composición de múltiples procesos `PIPE` y `FILTER`. A partir de los renombres, deduzca la estructura de la arquitectura que se describe. Dibujela en formato de cajas, ports y bindings como en la teoríca.
- Modificar la arquitectura para que los pipes puedan hacer de buffers de dos (o N si se anima) posiciones en vez de una.
- Cual de las dos arquitecturas (buffers de 1 o buffers de 2+) permite mayor concurrencia, ejemplifique con una traza.

Formato de Entrega

El taller debe ser entregado a través del campus en la fecha de entrega indicada en el documento. Entregar un archivo con las respuestas y explicaciones. Se debe poder copiar el FSP y pegarlo en MTSA con facilidad.