Programación Concurrente en Java
Laboratorio 3: Puente de una sola vía (con autos pesados)

J. Blanco, N. Wolovick

Descripción del Problema

Para cruzar un río, existe un puente que es lo bastante angosto como para impedir el paso simultáneo de autos rojos que van de oeste a este, y autos azules que van en sentido contrario. Los autos azules son tremendamente pesados porque transportan mercurio en su baúl para la empresa Dioxitek S.A. de la CNEA, con lo cual solo puede circular uno solo por encima del puente, a fin de evitar que la estructura sufra peligro[1].

Se necesita implementar un **Controlador de Tránsito** que mantenga las siguientes propiedades de *seguridad*:

1. No puede haber simultáneamente en el puente autos rojos y autos azules (colisión de autos).
2. No puede haber en el puente más de un auto azul (caída del puente).

También se requiere la siguiente propiedad de *progreso*:

3. Un auto azul cargado de mercurio no puede esperar, de manera indefinida, cruzar el puente (el mercurio eventualmente tiene que llegar a Alta Córdoba).

Descripción del entorno

El tarball `SingleLaneBridge.tar.gz` contiene una implementación gráfica del problema escrito en Java. Éste programa se obtuvo del curso Concurrent Programming de la Universidad de Chalmers, y es una adaptación del applet propuesto en el Capítulo 7 de [1].

El código incluye una clase `TrafficController` que tiene una implementación vacía, luego los autos azules y rojos se pueden simultáneamente cruzar en el puente (colisión) y dos o más autos azules pueden estar sobre el puente (caída). Aclaramos que el entorno ofrecido es solo una simulación gráfica del los hilos y nada nos importa acerca de su (mala o buena) implementación.

Tareas

Derivación de las regiones atómicas condicionales

La clase *TrafficController* está literalmente vacía:

TrafficController.java

```
public class TrafficController {
        public void enterLeft() {}
        public void enterRight() {}
        public void leaveLeft() {}
        public void leaveRight() {}
}
```

[1] En [2], Dijkstra critica duramente el uso una perspectiva antropomorfa en las Ciencias de la Computación: "*... never refer to parts of programs or pieces of equipment in an anthropomorphic terminology ...*" .

Estos métodos son llamados desde los hilos que se corresponden con cada auto rojo o azul. Usando la notación de [5] los llamados estos métodos se dan siempre de la siguiente manera.

$Rojo_i$: *enterLeft*() ; *leaveRight*()	$Azul_j$: *enterRight*() ; *leaveLeft*()

Podemos abstraernos de los llamados a métodos y de la creación y destrucción de hilos y pensar que tenemos n componentes *Rojo* y m componentes *Azul* ejecutando en paralelo y agregamos la variables r, a que cuentan la cantidad de autos rojos y azules sobre el puente. Bajo esta abstracción el multiprograma resultante es:

Pre: $r = 0 \wedge a = 0$	
$Rojo_i$: $*[$ $r := r + 1$; *cruzaelpuente* ; $r := r - 1$ $]$	$Azul_j$: $*[$ $a := a + 1$; *cruzaelpuente* ; $a := a - 1$ $]$
Inv: ?	

Se deberá:

1. **[10 pts]** Escribir el invariante que especifica la *propiedad de seguridad* (no colisiones, no caídas) respecto a las variables r, a. Ayuda: probar con los casos de test:

a	r	Inv
0	0	*true*
0	1	*true*
0	6	*true*
1	0	*true*
1	1	*false*
1	6	*false*
2	0	*false*
2	2	*false*

2. **[35 pts]** Derivar las guardas B_r^1, B_r^2 y B_a^1, B_a^2 de forma tal que si cambiamos las asignaciones por *regiones atómicas condicionales* el invariante se mantiene.

Pre: $r = 0 \wedge a = 0$	
$Rojo_i$: $*[$ **if** $B_r^1 \rightarrow r := r + 1$ **fi** ; *cruzaelpuente* ; **if** $B_r^2 \rightarrow r := r - 1$ **fi** $]$	$Azul_j$: $*[$ **if** $B_a^1 \rightarrow a := a + 1$ **fi** ; *cruzaelpuente* ; **if** $B_a^2 \rightarrow a := a - 1$ **fi** $]$
Inv: ?	

   Utilizar *program topology* y el invariante para simplificar algunas guardas que ya son válidas, es decir la condición del **if**...**fi** siempre evalua a verdadero, e **if** *true* $\rightarrow S$ **fi** es lo mismo que $\langle S \rangle$, o sea la sentencia ejecutada de manera atómica.

3. **[5 pts]** Escribir el multiprograma final con el invariante y las guardas resueltas y simplificadas.

Notar que no pedimos atacar con la teoría la *propiedad de progreso*.

Ayuda

- Escribir un invariante compacto simplifica mucho la tarea de derivación de guardas.
- La aserción central de Program Topology es fundamental para la simplificación.

Implementación de las regiones atómicas condicionales con SBS

Se deberán implementar las cuatro regiones atómicas condicionales derivadas en la sección anterior utilizando la técnica del Semáforo Binario Dividido [3, 4]. La biblioteca `java.util.concurrent` provee la clase `Sempahore` que implementa semáforos generales.

Para un par de regiones atómicas condicionales

$$\textbf{if } B_0 \rightarrow\ S_0 \textbf{ fi} \qquad\qquad \textbf{if } B_1 \rightarrow\ S_1 \textbf{ fi}$$

La implementación con SBS utiliza 3 semáforos m, s_0, s_1 y dos naturales b_0, b_1. Eliminando todas las aserciones intermedias de [4, Fig.2], se obtiene el siguiente código para la primera región atómica condicional, es decir el **if** $B_0 \rightarrow\ S_0$ **fi**, el segundo resulta simétrico:

$$\begin{array}{l} P.m \\ ;\textbf{if } B_0 \rightarrow \textbf{skip} \\ \ \square \neg B_0 \rightarrow\ \ b_0 := b_0 + 1 \\ \qquad\qquad\ ;V.m \\ \qquad\qquad\ ;P.s_0 \\ \qquad\qquad\ ;b_0 := b_0 - 1 \\ \textbf{fi} \\ ;\mathbf{S_0} \\ ;\textbf{if } B_0 \wedge b_0 > 0 \rightarrow V.s_0 \\ \ \square\ B_1 \wedge b_1 > 0 \rightarrow V.s_1 \\ \ \square\ (\neg B_0 \vee b_0 = 0) \wedge (\neg B_1 \vee b_1 = 0) \rightarrow V.m \\ \textbf{fi} \end{array}$$

Se deberá:

4. [**40 pts**] Implementar la clase `TrafficControler` utilizando la técnica del SBS poniendo antes de los incrementos y decrementos de las variables r, a `assert`s con las precondiciones que deben ser válidas y que fueron calculadas en el punto 2.

5. [**10 pts**] Modificar el código para darle prioridad para los autos azules, es decir la propiedad de progreso enunciada anteriormente.

Extras

Si disponen de tiempo y ganas:

6. [**20 pts**] Las guardas del protocolo de salida (lo que sigue al S_0) usualmente pueden ser simplificadas debido a la forma que tienen los programas. Por ejemplo suena razonable que en la entrada de un auto rojo no se despierte un auto azul esperando por entrar. Encontrar al menos dos simplificaciones, mostrar y demostrar las aserciones que son válidas antes del condicional de salida que permiten la simplificación de las guardas.

7. [**20 pts**] El programa con su interfaz de entrada y de salida asumen una serialización implícita en la entrada del puente, es decir nunca ocurre que dos autos rojos piden simultáneamente entrar al puente, con lo cual se eliminan ciertas condiciones de concurrencia que hacen funcionar implementaciones erróneas de `TrafficController`. Corregir este problema de la interfaz gráfica.

Referencias

[1] Jeff Magee, Jeff Kramer, *Concurrency: State Models and Java Programs*, 2nd edition, Wiley, 2006.

[2] Edsger W. Dijkstra, *On the cruelty of really teaching computing science*, EWD1036, 1988.

[3] Edsger W. Dijkstra, *A Tutorial on the Split Binary Semaphore*, EWD703, 1979.

[4] Damián Barsotti, Javier Blanco, *Automatic Refinement of Split Binary Semaphore*, Extended Abstract, 2007.

[5] W.H.J. Feijen and A.J.M. van Gasteren, *On a Method of MultiProgramming*, Monographs in Computer Science, Springer, 1999.