Ingeniería del Software II
Proyecto Nº 1

Integrantes del Grupo:

Luna, Andrés
Oliva, Pablo
Vargas Vieyra, Mariana

Facultad de Matemática, Astronomía y Física
Domingo 8 de Mayo de 2011

Ejercicio 1: Problema de los Fumadores (FSP)

Para este ejercicio, debíamos modelar un sistema que consta de tres fumadores y un proveedor. Todo fumador necesita papel, tabaco y fósforos para fumar; cada uno de ellos dispone de una reserva infinita de alguno de los tres recursos (cada recurso es poseído en forma infinita por exactamente uno de los fumadores), y debe obtener los otros dos recursos del proveedor. El proveedor dispone de una reserva infinita de los tres recursos, e interactúa con los fumadores a través de una mesa, donde coloca dos de los tres recursos, para que el fumador que necesita exactamente esos dos recursos acceda a ellos.

Inicialmente, se decidió modelar el sistema como la interacción de dos procesos: un proceso PROVEEDOR, instanciado una vez, y un proceso FUMADOR, instanciado tres veces y con una variable que indicara de qué recurso dispone en forma infinita. Con el fin de garantizar exclusión mutua y ausencia de deadlocks, se agregó el proceso MESA_MUTEX, con lo que el sistema completo quedó modelado en el proceso MESA, que es la composición paralela de 3 fumadores, un proveedor, y una mesa.

El PROVEEDOR es el componente más sencillo; realiza una elección no determinística entre cualquiera de los tres recursos posibles y luego coloca el segundo recurso necesitado por aquel fumador que adquirió la mesa; tras ello espera a recibir un mensaje de “listo” por parte del fumador, volviendo entonces a su estado inicial. Mediante un uso cuidadoso del orden de colocado de los recursos, pudimos evitar tener que pasarle información al proveedor sobre qué recurso necesita el fumador que adquirió el lock.

El FUMADOR posee una variable que indica qué tipo de fumador es (esto es, de qué recurso dispone infinitamente). Cualquiera sea su tipo, lo primero que hace es adquirir el lock para acceder a los recursos; adquirido el lock (lo que se modela por la acción ir_mesa), obtiene los recursos que necesita, fuma, le avisa al proveedor que terminó mediante un mensaje de “listo”, y libera el lock (lo que se modela por la acción dejar_mesa).

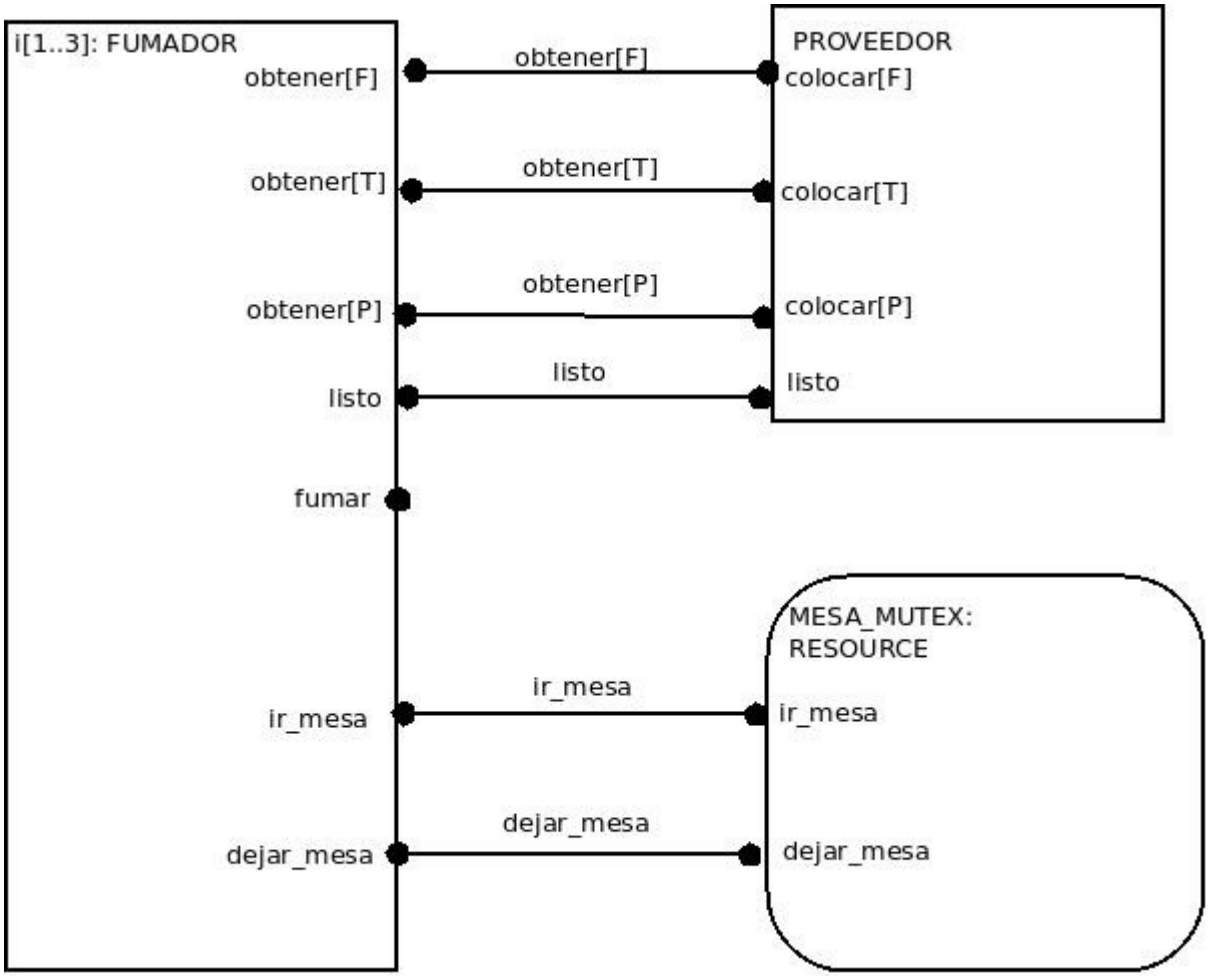
MESA_MUTEX es sencillamente un proceso utilizado para garantizar la exclusión mutua de los fumadores en su acceso a los recursos; se trata sencillamente de un lock, con las acciones de adquirir (llamado ir_mesa) y liberar (llamado dejar_mesa).

En la composición de los procesos para modelar el sistema completo, las acciones de colocar un recurso (realizadas por el proveedor) fueron renombradas para que se sincronizaran con las acciones de los fumadores de obtener un recurso. Además, se prefijó al proveedor y al lock con los números de los fumadores, también a fin de sincronizar las acciones.

Como parte del ejercicio, también debía comprobarse la imposibilidad de que el sistema caiga en deadlock, y que cada uno de los fumadores puede fumar con infinita frecuencia. Para comprobar lo primero, bastó con correr el safety check de LTSA; para comprobar lo segundo, se agregaron los tres progress al final del código. Ambas comprobaciones fueron satisfactorias.

El modelo final puede observarse en el código LTS adjunto.

Con el fin de aclarar nuestro diseño, creamos también el siguiente diagrama de estructura:



Consideramos que es más claro representar cada una de las acciones obtener/colocar por separado, que indicarlas como acciones indexadas sobre un rango. Esto se debe a que cada una de esas acciones fue vista desde el primer momento en que comenzamos a diseñar el sistema como algo intrínsecamente diferente a las otras dos.

Ejercicio 2: Algoritmo Fast-Max (Promela)

Para el ejercicio 2 se nos pidió implementar un modelo que permita encontrar el mayor entero de un arreglo en O(1), para el caso de contar con n^2 procesadores (siendo n el tamaño del arreglo).
A los fines de implementar el paralelismo (a pesar de no contar con n^2 procesadores), creamos un *active proctype* FAST_MAX que produce tantos procesos como sea necesario para verificar el algoritmo. Esto se logra incluyendo un *run (<proceso>)* dentro de un ciclo *for*.

Declaramos dos arreglos, uno de booleanos *max*, y otro de enteros, *a*. El primero tiene inicialmente el valor *true* en todas las posiciones, luego se lo va actualizando con el valor *false* en la posición *i* según sea *a[i]* menor a otro elemento de dicho arreglo (para cada *i*). El segundo es un arreglo sobre el cual se correrá el algoritmo que elige el valor más grande.

Los principales *proctypes* creados son los siguientes:

INITBOOL(): inicializa el arreglo de booleanos *max* de manera tal que para cada i menor que n y mayor que 0, *max* en el lugar i es *true*. Este algoritmo se corre de manera paralela, es decir, en el cuerpo de FAST_MAX hay un *for(i:0..n-1) { run (*INITBOOL*(i)) }*.

LOAD(): recorre el arreglo de enteros *a* y lo llena de manera no determinística con enteros menores a 25. Se ejecuta de manera secuencial.

IN PARALLEL(): se encarga de llenar el arreglo *max* con *false* para los enteros que son menores que algún otro elemento de *a*, de este modo, en el arreglo de booleanos sólo queda un sólo *true* en la posición correspondiente al mayor número de *a*. Este también se ejecuta en paralelo.

PARALLEL_CHECK(): recorre el arreglo de booleanos, identifica el lugar que tiene el *true*, y coloca en *result* el correspondiente valor de *a*.

MOSTRAR (): imprime el arreglo de enteros y el resultado.

VERIFY(): verifica que el resultado sea correcto.

Como se puede ver, nos hemos limitado a correr de manera paralela los procesos que se nos indican en la consigna para tal fin.

Para lograr el correcto orden en las posibles trazas de ejecución, declaramos variables booleanas y enteras que actuaron como “banderas”, esto es, un determinado proceso esperaba un determinado evento para poder comenzar, y la ocurrencia de dicho evento se indicaba dándole a la bandera el valor correspondiente.

En cuanto a las propiedades a verificar, éstas eran dos: “el programa eventualmente termina”, y “el valor que arroja es correcto”.

“El programa eventualmente termina”: esta propiedad se representó mediante la variable *finished* inicializada en *false* y posteriormente valuada en *true* al momento de finalizar el algoritmo. Se verificó entonces la fórmula L.T.L. “*<> (finished == true)”* en iSpin y xSpin. En ambos casos se le dio al proceso verificador 3072mb de memoria ram y se usó una profundidad máxima de cuatromil nodos. La propiedad fue válida.

“El valor que arroja es correcto”: el que sea “correcto” significa que para todo elemento del arreglo, *result* es mayor o igual a dicho elemento. Esto se representó mediante el booleano *correct*, el cual se inicializó en *true* y cuyo valor permaneció invariable en la ejecución de VERIFY(). Este proceso ejecuta la acción “*correct && result >= <elemento_de_a>”* para todo elemento de *a*, y del que la variable se mantenga en *true* se deduce que el resultado es correcto. Se verificó la fórmula L.T.L. “[] *(correct == true)”*, y resultó válida en iSpin y xSpin, con 3072mb de memoria ram y una profundidad máxima de 10000 nodos. Sin embargo, la verificación resultó incompleta dado que la memoria otorgada le fue insuficiente.