Capítulo 2

Programación de Simuladores

2.1. Introducción

En el tema anterior hemos visto cómo crear modelos de Simulación de sucesos discretos. En este tema veremos cómo implementar un simulador a partir de este tipo de modelos. Afortunadamente, en la actualidad existen numerosos paquetes de software y bibliotecas de funciones que contienen los elementos necesarios para programar un simulador de sucesos discretos. Por lo tanto no será necesario que implementemos desde cero nuestros simuladores.

En este tema vamos a ver dos paradigmas de programación para simulación: la orientada a sucesos y la orientada a procesos. Básicamente, el primero de ellos se basa en la ejecución secuencial del diagrama de sucesos, mientras que el segundo se basa en la identificación de procesos recurrentes en el diagrama de sucesos y su ejecución en paralelo. Como ejemplo del primer paradigma introduciremos la biblioteca SMPL para C, y para el segundo la biblioteca JavaSim para Java. En las siguientes secciones vamos a ver con detalle en qué consiste cada uno de estos paradigmas.

2.2. Programación orientada a sucesos

Para ilustrar cómo funciona un simulador orientado a sucesos vamos a estudiar un posible algoritmo para ejecutar un diagrama de sucesos. Tomaremos como ejemplo el diagrama de una estación FCFS (ver figura 2.1), cuyo funcionamiento estudiamos en el tema anterior.

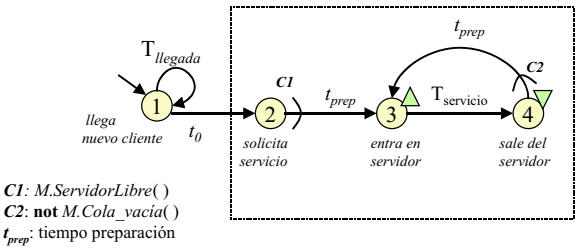

Figura 2.1: Diagrama de sucesos para una estación FCFS.

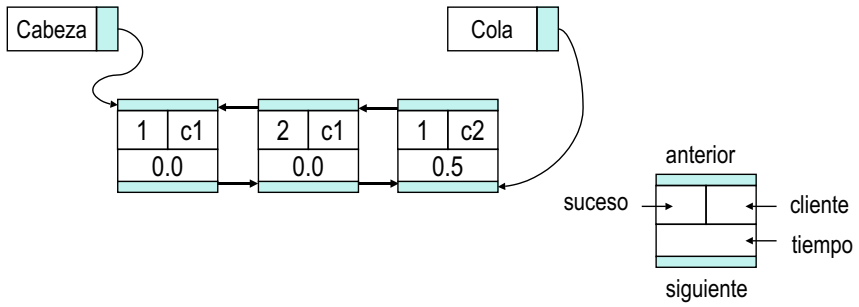

Figura 2.2: Planificador de sucesos basado en una lista doblemente enlazada.

Supondremos que cada suceso se describe con un fragmento de código que actúa sobre las variables de estado afectadas por el suceso. Por ejemplo, en el diagrama de sucesos de una estación FCFS, los sucesos tendrían asociado los siguientes fragmentos de código:

Suceso	Código
①	cliente++;
②	if (! servidorLibre) encola(Q, cliente);
③	servidorLibre=0; desencola(Q, cliente);
④	procesados++; if (colaVacia(Q)) servidorLibre=1;

Para ir ejecutando el diagrama de sucesos, es necesario mantener una estructura de datos con aquellos sucesos que deben ejecutarse en un determinado tiempo (sucesos planificados). Naturalmente, los primeros sucesos que contendrá esta estructura serán los sucesos iniciales de la simulación.

La estructura más utilizada para guardar los sucesos planificados es una lista ordenada por el tiempo de ocurrencia. Así, en la cabeza de la lista siempre estará el siguiente suceso a ser ejecutado por el simulador (ver figura 2.2). Fíjate que junto a cada suceso se indica además el cliente involucrado.

Cuando la cantidad de sucesos planificados es muy grande, entonces debe utilizarse una estructura de datos que acelere la búsqueda e inserción de los sucesos planificados, generalmente una estructura en árbol. La más aceptada es el *montículo*, ya que en la raíz siempre se ubica el suceso con el tiempo de ejecución menor. Recuerda que un montículo es un árbol binario donde el contenido de cada nodo siempre debe ser menor que el de sus hijos.

Volviendo a la ejecución del diagrama de sucesos, un posible algoritmo podría ser el que se muestra en la figura 2.3.

El planificador de sucesos se utilizaría en las líneas 3, 5 y 8. Es decir, buena parte del algoritmo del simulador está formado por la gestión del planificador de sucesos, de ahí la importancia de seleccionar una buena estructura de datos para esta tarea.

Es importante comentar que la condición de fin de simulación depende de los objetivos de la simulación, y puede venir dada bien por el tiempo de simulación o bien por alguna variable de estado, como por ejemplo el número de clientes creados.

Al algoritmo básico anterior habría que añadirle las rutinas de recolección de

```
1   Inicializa tiempo de simulación
2   Inicializa variables de estado
3   Inicializa planificador de sucesos
4   Mientras no FIN-SIMULACIÓN hacer
5       Selecciona el suceso S con menor tiempo de ejecución T
6       Avanza el tiempo de simulación a T
7       Ejecuta el código del suceso seleccionado S
8       Planifica/cancela todos los sucesos que son apuntados en el
9           diagrama de sucesos por S, y que cumplan las condiciones necesarias.
10  Imprime resultados
```

Figura 2.3: Algoritmo para la ejecución de un diagrama de sucesos.

resultados, así como la presentación de los mismos al finalizar la simulación. Pero esto ya lo analizaremos con más detalle en el tema de tratamiento de resultados.

De todo lo anterior, podemos concluir que un lenguaje de simulación orientado a sucesos debe proveer los siguientes elementos básicos:

1. Un *planificador* de sucesos, que proporcione y gestione la estructura de datos de los sucesos planificados.
2. Un *reloj* para controlar el tiempo de simulación.
3. Rutinas para la definición e inicialización de variables de estado.
4. Rutinas para la implementación de los sucesos.
5. Generadores para variables aleatorias.
6. Facilidades para el tratamiento de resultados.

Los puntos 2 y 3 se realizan generalmente con el lenguaje de programación que sirve de base al paquete o biblioteca de simulación que estemos utilizando. Por ejemplo, en SMPL las variables de estado son variables de C y los sucesos se implementan con código C.

2.3. SMPL

SMPL es una biblioteca de C que proporciona todas las funciones necesarias para la crear pequeños simuladores de sucesos discretos. Esta biblioteca proporciona principalmente los siguientes elementos:

- La gestión de los sucesos (*events*).
- La gestión de los recursos del sistema (*facilities*).
- La generación de informes.

- La generación de variables aleatorias.

Los tres primeros aspectos están recogidos en la biblioteca `smpl.c` y el último en `rand.c`. Cuando se compila un programa SMPL deben enlazarse ambos ficheros como sigue:

```
gcc -o ejecutable smpl.c rand.c programaSMPL.c -lm
```

Para mostrar las distintas características de este lenguaje vamos a introducir un pequeño ejemplo de SMPL que implementa una estación FCFS de tipo M/M/1, es decir, con un servidor, y con tiempos de llegada y de servicio que siguen una distribución exponencial.

```
#include "smpl.h"

main()
{
     smpl(0,"Simple");
     FCFS=facility("Estacion",1);
     Ta=1.0;
     Ts=0.8;
     schedule(1,0.0,1);

     while (time()<1000.0)
     {
        cause(&event,&cli);
        switch(event)
        {
           case 1:
              schedule(2,0.0,cli);
              cli++;
              schedule(1,expntl(Ta),cli);
              break;
           case 2:
              if (request(FCFS,cli,0)==0)
              schedule(4,expntl(Ta),cli);
              break;
           /* ojo! el suceso 3 no debe especificarse */
           case 4:
              release(FCFS,cli);
              break;
        }
     }
     report();
}
```

Todo programa en SMPL debe incluir la biblioteca "smpl.h". La estructura de un programa SMPL es siempre la misma:

1. Primero se inicializan las estructuras de datos internas de SMPL invocando al procedimiento:

`void smpl(int` *Modo* `, char *`*Nombre*`)`

El *Modo* puede ser interactivo (1) o no (0), y el *Nombre* es una cadena que indetifica el simulador, y que se utiliza en la generación de informes.

2. Después se inicializan todas las variables de estado que hayamos declarado, y se crean todos los recursos del modelo. Para ello utilizaremos la función:

   `int facility(char *`*Nombre* `, int` *NumeroServidores*`)`

   La cadena *Nombre* solo se utiliza para la generación de informes. Lo que a nosotros nos interesa es el identificador numérico que nos devuelva la función, ya que éste será el que utilicemos a lo largo de todo el programa. El argumento *NumeroServidores* permite indicar cuántos servidores tiene la estación de servicio.

3. A continuación se planifican los sucesos iniciales de la simulación. En nuestro ejemplo, el suceso inicial es el suceso ① que implementa la creación de los identificadores de nuevos clientes, (incrementando la variable `cli`). Para planificar un suceso, se utiliza el procedimiento siguiente:

   `void schedule(int` *suceso*`, double` *Tiempo*`, int` *cliente*`)`

   Fíjate que en SMPL los sucesos siempre van ligados a algún cliente, de modo que cuando se planifica un suceso hay que indicar tanto el identificador del suceso como el del cliente. El argumento *Tiempo* indica el tiempo que debe transcurrir a partir del tiempo de simulación actual para ejecutar el suceso planificado. En otras palabras, corresponde exactamente con el tiempo que ponemos en los arcos de los diagramas de suceso. Ojo, no se trata de un tiempo absoluto, sino relativo.

4. El ciclo de simulación se realiza con un bucle, cuya condición de parada es la condición de fin de simulación. En el ejemplo, la condición de parada es que el tiempo de simulación actual sea igual o superior a 1000.0 unidades de tiempo. Para saber el tiempo actual de simulación utilizamos la función `double time()`.

5. La primera acción del bucle debe ser la selección del siguiente suceso a ejecutar (recuerda el algoritmo de la sección anterior). Esto se realiza con el procedimiento siguiente:

   `void cause(int *`*suceso*`, int *`*cliente*`)`

   En este caso ambos argumentos son de salida. Es decir, el procedimiento nos devuelve el identificador del suceso y el del cliente a ejecutar.

6. Una vez recogido el suceso a ejecutar, debemos seleccionar y ejecutar el código del mismo. Para ello, se utiliza la sentencia `switch`, donde cada caso contiene el código de cada suceso. Recuerda que dentro del código de cada suceso también debe incluirse la planificación/cancelación de los sucesos que están conectados con él en el diagrama de sucesos.

7. Por último, y ya fuera del bucle de simulación, se presentan los resultados. En este caso se ha utilizado el procedimiento `report()`, el cual muestra por pantalla las estadísticas de cada estacion de servicio.

Como habrás podido notar, en SMPL, los sucesos, los clientes y los recursos se identifican siempre con números enteros positivos.

Recursos en SMPL. Uno de los aspectos más destacables de SMPL es que se encarga completamente de la gestión de los recursos del modelo. Es por ello que el suceso ③, asi como todas las planificaciones por él implicadas, no deben ser incluidas en el programa SMPL. Así, el programador sólo debe indicar dónde se realiza la petición de servicio a un recurso (suceso ②) y dónde se libera (suceso ④).

Para la peticion de servicio a un recurso se utiliza la función:

`int request(int` *facility*`, int` *cliente*`, int` *prioridad*`)`

Si la función devuelve 0, significa que la estación está libre, y si devuelve 1, la estación está ocupada. Fíjate que en el caso de que esté libre, en el diagrama de sucesos planificábamos el suceso ③. Pero como este suceso modifica las variables de la estación de servicio, éste debe ser ejecutado internamente por SMPL, y por consiguiente lo omitiremos del programa. De este modo, cuando el servidor esté libre, desde el suceso ② planificamos directamente el suceso ④.

Fíjate también que en el suceso ②, el programa no hace nada si se encuentra la estación ocupada. En realidad, la función `request` además de devolver el estado de la estación, también encola al cliente si encuentra la estación ocupada. Por esta razón no debe programarse nada si la función devuelve un valor distinto de cero.

Otra forma de realizar una petición a una estación consiste en expulsar al cliente que se esté siriviendo en ese momento (petición con derecho de expulsión). Para ello utilizaremos la función `preempt`, en lugar de `request`.

Finalmente, para liberar una estación de servicio utilizaremos el procedimiento:

`void release(int` *facility*`, int` *cliente*`)`

Fíjate que para liberar una estación es necesario conocer el cliente que la está ocupando. Un error corriente en SMPL se produce cuando una estación se libera con el cliente no apropiado. Es por ello que debemos ser especialmente cuidadosos con la planificación de las peticiones y las liberaciones de las estaciones.

Otra función básica de SMPL es la cancelación de sucesos, que se realiza con la siguiente función:

`int cancel(int` *suceso*`)`

Esta devuelve el cliente al cual se le ha cancelado el suceso.

Como has podido comprobar, todo lo concerniente a las estaciones de servicio (variables de estado y sucesos) esta totalmente controlado por SMPL, entonces ¿cómo se puede consultar el estado de las estaciones de servicio?

SMPL también proporciona las funciones necesarias para consultar las variables de estado de una estación de servicio, éstas se resumen en la tabla siguiente.

`int inq(int f)`	Número de trabajos en cola
`int status(int f)`	Estado de la estación (1=ocupada)
`double U(int f)`	Nivel de utilización de la estación
`double B(int f)`	Duración media de ocupación
`double Lq(int f)`	Longitud media de la cola

A modo de ejercicio, puedes extender el programa en SMPL anterior para incluir los sucesos de roturas que vimos en el ejemplo 1.3.1 el tema anterior.

Variables aleatorias en SMPL. En el ejemplo de SMPL anterior hemos utilizado la función `expntl` para representar los tiempos de llegada entre clientes y sus tiempos de servicio. Esta función devuelve una secuencia de números aleatorios que sigue una distribución exponencial (es decir representa una variable aleatoria con distribución exponencial). Ojo, no debes confundir esta función con la función exponencial `exp(x)` la cual representa la función e^x.

En SMPL tenemos las siguientes funciones para generar valores para variables aleatorias:

`int stream()`	obtiene/selecciona un flujo de números aleatorios
`double ranf()`	genera una uniforme $U(0,1)$
`double uniform(a, b)`	genera una uniforme $U(a,b)$
`double expntl(m)`	genera una exponencial Exp con media m
`double erlang(m, s)`	genera una $Erlang$ con media m y desviación s
`double hyperx(m, s)`	genera una hiperexponencial
`double normal(x, s)`	genera una normal con media m y desviación s
`int random(i, j)`	genera un entero aleatorio en $[i,j]$ con $(i < j)$

Otras opciones avanzadas. Existen dos procedimientos más para tareas avanzadas, a saber:

Traza de la ejecución Mediante el procedimiento `void trace(n)` podemos seleccionar los distintos modos de traza de SMPL:

- con n=0, se desactiva la traza de la simulación
- con n=1, se da una salida de forma continua
- con n=2, se detiene la ejecución cuando se llena la pantalla
- con n=3, la ejecución se detiene en cada mensaje
- con n=4, se combinan mensajes de usuario con los pre-definidos

Generador de informes: Con el procedimiento `report()` se imprime pantalla los resultados generados internamente por SMPL. Un ejemplo de informe sería el siguiente:

Las estadísticas que recoge SMPL para visualizar con el procedimiento `report()` son las siguientes:

```
                    smpl SIMULATION REPORT

MODEL: Tres colas y tres servidores                    TIME:    10006.070
                                                    INTERVAL:   10006.070

                    MEAN BUSY      MEAN QUEUE          OPERATION COUNTS
FACILITY       UTIL.    PERIOD       LENGTH    RELEASE  PREEMPT  QUEUE
servidor      0.0306     1.001        0.001        306        0      8
servidor      0.0476     0.934        0.002        510        0     20
servidor      0.0182     1.032        0.000        177        0      8
```

[UTIL.]	Porcentaje total de utilización de los servidores (tanto por número de servidores).
[MEAN BUSY PERIOD]	Tiempo medio de ocupación de la estación (debe tender al tiempo medio de servicio).
[MEAN QUEUE LENGTH]	Tamaño medio de la cola de la estación de servicio.
[RELEASE]	Número de operaciones de liberación efectuadas (clientes procesados).
[PREEMPT]	Número de clientes expulsados.
[QUEUE]	Número de clientes que han tenido que encolarse.

Es importante resaltar el hecho de que estas estadísticas son insuficientes para sacar conclusiones del simulador. Como veremos en el tema de tratamiento de resultados, será necesario analizar la evolución de la varianza de las variables de estado y calcular los intervalos de confianza de los resultados obtenidos con el simulador. Aún así, la información que nos muestra `report()` puede ser útil para depurar el programa. Por ejemplo, podremos comprobar que los tiempos de servicio se aproximan a los que hemos modelado, y que las colas funcionan más o menos de acuerdo a la carga del sistema.

2.4. Programación orientada a procesos

Una de las principales limitaciones de un simulador orientado a sucesos es que los sucesos se ejecutan de forma secuencial, aunque ocurran en el mismo tiempo de simulación. Esto implica que el tiempo de ejecución sea directamente proporcional al número de sucesos ejecutados por el simulador, que suele ser bastante elevado. Además, el planificador de sucesos puede llegar a desbordar la memoria cuando los sucesos ocurren en tiempos muy próximos.

Como alternativa, se propone el concepto de *proceso*, que trata de paliar estas limitaciones. Con los procesos vamos a tratar de identificar las secuencias de sucesos *recurrentes* que pueden ejecutarse en paralelo. Para ello vamos a utilizar el diagrama de sucesos. Por ejemplo, en el diagrama de la figura 2.1 podemos identificar dos secuencias recurrentes: el suceso periódico de llegada de clientes (suceso ①), y la secuencia de sucesos ③ y ④, que representa el procesamiento de los clientes en la estación de servicio. Fíjate que se trata de dos secuencias repetitivas que pueden ejecutarse en paralelo.

De forma esquemática, podríamos expresar estos procesos como sigue:

Proceso 1	**Proceso 2**
Repite	Mientras c2
Ejecuta ①	Ejecuta ③
Espera Ta	Espera Ts
	Ejecuta ④

Según el diagrama de sucesos, Ta es el tiempo entre llegadas, Ts es el tiempo de servicio, y c2 es la condición de que la cola no esté vacía. El código de los sucesos de este ejemplo se describieron al principio del tema.

En los procesos hemos introducido la instrucción Espera para indicar que el proceso se detiene y planifica su reanudación según el tiempo especificado. En realidad, este será tiempo de simulación (según el reloj de la simulación), y no tiene relación directa con el de ejecución (tiempo de CPU). Más tarde veremos como se gestiona la ejecución de estos procesos.

De lo anterior, podemos deducir que un proceso puede estar en uno de los siguientes estados:

- *Activo*, cuando el proceso está ejecutándose.
- *Planificado*, cuando el proceso está detenido esperando su reanudación para un determinado tiempo.
- *Pasivo*, cuando no está ni planificado ni activo, aunque puede ser activado o planificado por otro proceso.
- *Terminado*, cuando el proceso ha finalizado totalmente su ejecución. Un proceso terminado ya no podrá ser nunca más activado ni planificado.

Volviendo al ejemplo anterior, todavía tenemos que ubicar el suceso ② en los procesos anteriores. Dado que la ejecución de este suceso planifica la ejecución del suceso ③, ubicado en el Proceso 2, éste sólo puede colocarse en el Proceso 1. La conexión

entre ambos procesos se realiza con la instrucción Activa, la cual permite activar un proceso determinado desde otro proceso distinto. Con esta instrucción estamos implementando la planificación del Proceso 2 desde el Proceso 1. La implementación definitiva de estos dos procesos sería la siguiente:

Proceso 1	**Proceso 2**
1 Repite	1 Repite
2 Ejecuta ①	2 Mientras c2
3 Ejecuta ②	3 Ejecuta ③
4 Si c1	4 Espera Ts
5 Activa Proceso 2	5 Ejecuta ④
6 Espera Ta	6 Cancela

Normalmente, los procesos recurrentes se ejecutan de forma indefinida utilizando un bucle infinito. Fíjate que en el Proceso 2 hemos introducido además la instrucción Cancela, para indicar que este proceso pasa a un estado *pasivo* cuando no tenga clientes que procesar (la cola se ha quedado vacía). En este caso, cuando llegue un nuevo cliente en el Proceso 1 se activará de nuevo el Proceso 2, y pasará a procesar los nuevos clientes de la cola.

Es importante destacar que si el Proceso 2 no incluyese el bucle infinito, al volverse a despertar su hilo terminaría (pasaría al estado *terminado*), y no podría volverse a activar nunca más.

Como puedes apreciar, el diagrama de sucesos se ha simplificado a dos procesos: el proceso de llegadas y el proceso de servicios. Con esta nueva visión, conseguimos reducir el número de planificaciones, y además permitimos su ejecución en paralelo.

Para realizar la ejecución de un conjunto de procesos, tendremos que mantener al menos dos estructuras de datos:

- *Procesos Planificados*, que es una lista con los procesos que están esperando su reanudación en un tiempo determinado. Esta estructura es muy parecida a la de los sucesos planificados que vimos en la sección anterior.

- *Procesos Activos*, que contiene los procesos que están siendo ejecutados actualmente.

El algoritmo de ejecución de un simulador con procesos es muy parecido al que hemos visto para los simuladores orientados a sucesos. En la figura 2.4 se resume este algoritmo. El simulador tiene que gestionar ahora los procesos planificados, de tal forma que si en un momento dado no hay ningún proceso activo, se saca del planificador el proceso "dormido" que tenga menor tiempo de simulación, se avanza el reloj de simulación y se activa dicho proceso.

Veamos ahora parte de la ejecución de los procesos del ejemplo mediante la traza de la figura 2.5. Junto a cada proceso hemos indicado la línea de código que se está ejecutando (o planificando) en cada momento. Para esta traza se ha tomado un tiempo entre llegadas de $Ta = 0{,}5$ y un tiempo de servicio de $Ts = 0{,}6$. Se deja como ejercicio completar la traza hasta procesar cuatro clientes. En la traza hemos sombreado los procesos planificados por el proceso activo en esos momentos.

```
1   Inicializa tiempo de simulación
2   Inicializa variables de estado
3   Inicializa planificador de procesos
4   Mientras hayan procesos activos y/o planificados hacer
5       Si no hay proceso activo
6           Selecciona el siguiente proceso planificado con tiempo mínimo T
7           Avanza el tiempo de simulación a T
8           Ejecuta el proceso seleccionado (pasa a activo)
9   Muestra los resultados de la simulación
```

Figura 2.4: Algoritmo para la ejecución de procesos.

Planificador			Estación		
T_{sim}	*Activo*	*Planificados*	♯*Cliente*	*Servidor*	*Cola*
0	P1(1)	-	-	libre	[ ]
0	P1(2)	-	1	”	”
0	P1(3)	-	”		[1]
0	P1(4,5)	P2(1),0.0	”	”	”
0	P1(6)	P2(1),0.0	”	”	”
0		P1(1),0.5			
0	-	P2(1),0.0 √	”	”	”
0		P1(1),0.5			
0	P2(1,2)	P1(1),0.5	”	”	”
0	P2(3)	P1(1),0.5	”	ocupado	[ ]
0	P2(4)	P1(1),0.5	”	”	”
0		P2(5),0.6			
0	-	P1(1),0.5 √	”	”	”
0		P2(5),0.6			
0.5	P1(1)	P2(5),0.6	-	”	”
0.5	P1(2)	P2(5),0.6	2	ocupado	”
0.5	P1(3)	P2(5),0.6	”	”	[2]
0.5	P1(4)	P2(5),0.6	”	”	”
0.5	P1(6)	P2(5),0.6	”	”	”
		P1(1),1.0			
0.5	-	P2(5),0.6 √	-	”	”
		P1(1),1.0			
0.6	P2(5)	P1(1),1.0	1	”	”
0.6	P2(2,3)	P1(1),1.0	2	”	[ ]
0.6	P2(4)	P1(1),1.0	”	”	”
		P2(5),1.2			
...	...	...	...	...	...

Figura 2.5: Traza de ejecución basada en procesos.