TEMPO VISUAL

Rafael Ruiz-Conejo[1], Jorge Buenabad-Chávez[2]
[1]Escuela de Informática Mazatlán, Universidad Autónoma de Sinaloa, México
[2]Departamento de Computación, CINVESTAV Zacatenco, México
E-mail: rafaelrc21@hotmail.com

(*Enviado Febrero 10, 2013; Aceptado Marzo 20, 2013*)

Resumen
La programación concurrente facilita el desarrollo de aplicaciones, dividiéndolas en módulos que interactúan entre sí. Para llevar a cabo la interacción entre los módulos se utilizan mecanismos de sincronización como: semáforos, barreras, mensajes, entre otros. Estos mecanismos no son fáciles de usar en aplicaciones grandes o complejas. Para facilitar la programación concurrente se han propuesto otros mecanismos o lenguajes. Por ejemplo Tempo, que es un lenguaje declarativo que sirve para facilitar la interacción entre módulos. Pero Tempo no es tan fácil de usar, pues en ocasiones es un poco difícil entender cómo funciona. Tempo Visual (TV) es un modelo de Tempo en ambiente gráfico, que ayuda a desarrollar aplicaciones concurrentes de manera sencilla. Con TV el programador no tendrá que preocuparse por entender cómo funcionan las reglas de Tempo. Solo desarrollará el código secuencial de su aplicación con los diferentes procesos (o *threads*) que intervendrán durante la ejecución de la misma. Entonces, con TV, el programador marcará con el mouse las partes de código que se ejecutarán concurrentemente y así obtener una aplicación concurrente.

Palabras Clave: *Programación Concurrente, Tempo Visual, Lenguajes Declarativos.*

Abstract
The concurrent programming facilitates the development of applications, dividing them into modules that interact with each other. To carry out the interaction between the modules, synchronization mechanisms are used, such as: traffic lights, barriers, messages, among others. These mechanisms are not easy to use in large or complex applications. To facilitate concurrent programming other mechanisms or languages have been proposed. For example Tempo, which is a declarative language that serves to facilitate the interaction between modules. But Tempo is not so easy to use, because sometimes it's a bit difficult to understand how it works. Tempo Visual (TV) is a Tempo model in graphic environment, which helps to develop concurrent applications in a simple way. With TV the programmer will not have to worry about understanding how the Tempo rules work. Only develop the sequential code of your application with the different processes (or threads) that will intervene during the execution of the same. Then, with TV, the programmer will mark with the mouse the parts of code that will be executed concurrently and thus obtain a concurrent application.

Keywords: *Concurrent Programming, Tempo Visual, Declarative Languages.*

1 INTRODUCCIÓN

La programación concurrente no es tan sencilla, aunque se ha venido incrementando su uso, pues ayuda a mejorar el funcionamiento de aplicaciones grandes y complejas [1].

Para hacer un poco más sencilla la programación concurrente hemos desarrollado un sistema en el cual la especificación de la concurrencia en las aplicaciones se lleva a cabo de manera gráfica. Además, la especificación de la sincronización de procesos está separada del resto del código y la sincronización de procesos se basa en Tempo [2, 3], un lenguaje declarativo basado en lógica de primer orden. Su sintaxis es muy simple (su único predicado predefinido es "<", que representa precedencia en ejecución).

Desarrollar un sistema concurrente no es sencillo, ya que dentro del sistema se manejan varios threads o procesos que interactúan entre sí. Por lo tanto, es necesario especificar la comunicación y sincronización entre los mismos. Actualmente, la mayoría de los mecanismos existentes de sincronización de procesos son de bajo nivel, y el código de sincronización está mezclado con el resto del código de la aplicación. Estos factores dificultan seriamente el diseño y la implementación de aplicaciones concurrentes.

En la programación secuencial el problema se reduce básicamente a obtener un resultado final que sea correcto y que el programa termine, mientras que en la programación concurrente se producen resultados durante su ejecución y no siempre se requiere que el programa termine.

En la programación concurrente el programador debe asegurar que ciertas propiedades existan durante la ejecución de la aplicación. Por ejemplo, en el problema de los lectores y escritores, un escritor no puede escribir una variable mientras un lector la este leyendo u otro escritor

la este escribiendo. Para asegurar estas propiedades se usan los mecanismos de sincronización, pero su uso no es del todo intuitivo pues se deben definir y actualizar variables de dichos mecanismos en diferentes puntos de la aplicación. Consecuentemente, una aplicación concurrente requiere la programación de sus procesos junto con la programación de su sincronización, lo que dificulta la tarea del programador.

Para facilitar la programación concurrente se han desarrollado alternativas que permiten al programador separar la tarea de sincronizar los procesos, de la programación de los mismos (parte secuencial). Tal es el caso de Tempo, un lenguaje de programación declarativo que utiliza lógica de restricciones para especificar la sincronización entre procesos. Las restricciones son restricciones de precedencia de la forma A < B, y se leen como "A precede B", o bien a se ejecuta primero que B, donde A y B son eventos (o procesos) y < especifica el orden parcial.

En Tempo para cualquier evento A, A+ es implícitamente precedido por A, es decir A < A+. También, A+ < A++, A++ < A+++,... La ocurrencia N de un evento A se presenta por A(+N). Los predicados de restricciones son de la forma p(A1;...;An) donde p es el nombre del predicado y Ai es un evento. La definición de las reglas de precedencia de los predicados son de la forma p(A1;...;An) ← X, donde X es una regla de precedencia entre los eventos Ai. Un ejemplo ayudará a entender mejor la especificación de sincronización con Tempo. La Figura 1 muestra un ejemplo de la definición y manejo de una pila en Java (izquierda) y los predicados de restricciones entre eventos Tempo y la definición de la reglas de precedencia (derecha).

```
class Stack {
    static final int MAX = 10;
    int pos = -1 ;
    Object [] contents = new Object [ MAX ];

    public Object pop() { < a1 >
        return contents[ pos-- ] ; < a2 > }

    public Object push( Object e) { < a3 >
        return contents[ ++pos ] = e ; < a4 > }
}
```

```
pop(a4, a1)
pop(A, B) ← A < B, p(A+, B+).

push(a2, a3(+MAX))
push(A, B) ← A < B, p(A+, B+).
```

Figura 1 Sincronización de procesos con Tempo.

Los eventos de interés en el programa son marcados con < > y pueden considerarse como comentarios por ahora. El predicado *pop* y su definición garantizan que el sacar datos de la pila ocurra si y solo si existen datos en ella, es decir, hayan ocurrido llamados al procedimiento *push* y su definición garantiza que solo se metan datos a la pila hasta la capacidad de la pila. Para correr el programa es necesario preprocesar el programa java para sustituir los marcadores de la forma < x > por un llamado al procedimiento *tempo.check("x")*, por ejemplo. A tiempo de corrida el llamado a *tempo.check()* pasa al interprete Tempo el evento a evaluar por su precedencia con otros eventos.

Como se puede ver el uso actual de Tempo no es sencillo, no es del todo intuitivo. Particularmente en la definición de los predicados y sus restricciones. Pues para aplicaciones más complejas tales predicados serán aún más complejos.

TV es una interfaz gráfica que facilita la programación concurrente basándose en Tempo. TV guía al programador para que después de definir los diferentes procesos secuenciales que componen su aplicación, gráficamente seleccione con el mouse secciones de código y les especifique un orden de ejecución.

TV generará automáticamente las definiciones de los predicados y los llamados a *tempo.check()* al inicio y al final de cada sección de código marcado. Además, ofrece los óordenes de ejecución correspondientes a los problemas de concurrencia clásicos como: productor-consumidor, lectores y escritores, entre otros.

2 PROGRAMACIÓN CONCURRENTE

Una aplicación concurrente consiste de varios procesos secuenciales que interactúan entre si. En ella se manejan hilos o threads (procesos), son aplicaciones que resuelven un problema en menos tiempo, pueden ejecutarse en paralelo (en una maquina con varios procesadores) o en forma distribuida (en varias maquinas), pero son todas aplicaciones concurrentes.

En un programa concurrente los procesos interactúan entre si para competir por recursos compartidos como: bloques particulares de memoria o algún dispositivo periférico; o bien, comunicarse para compartir información.

La programación concurrente hace posible aprovechar al máximo los recursos de un sistema de computo, ya que se pueden programar varios procesos para que corran intercalados entre si en un solo procesador, o al mismo tiempo, en un sistema con varios procesadores. Las aplicaciones que requieren gran capacidad de cómputo duran días para llevar a cabo su tarea en un solo procesador, pero pueden hacerlo mucho mas rápido utilizando varios procesadores, a la vez.

Pero al mismo tiempo la programación concurrente es una tarea difícil para los programadores. Pues hay que programar los procesos secuenciales y también hay que especificar la sincronización entre los mismos mediante primitivas de sincronización que no son fáciles de usar. Esta sincronización tiene el propósito de controlar situaciones típicas de competencia por recursos del sistema.

En la programación concurrente, necesitamos las siguientes condiciones: los procesos no deben encontrarse al mismo tiempo dentro de sus secciones criticas (aquellas partes del programa donde se tiene acceso a los recursos compartidos), no deben hacerse hipótesis sobre cuanto tiempo usara el CPU un proceso, ninguno de los procesos que estén en ejecución fuera de su sección critica puede

bloquear a otros procesos, ningún proceso debe esperar eternamente para entrar a su sección crítica.

A continuación daremos a conocer varias formas de cómo se han modelado las diferentes situaciones en las que dos o más procesos compiten entre sí por entrar a su sección crítica.

3 PROBLEMAS CLÁSICOS DE CONCURRENCIA

Las situaciones típicas de competencia en aplicaciones concurrentes se han abstraído en lo que hoy se conoce como problemas clásicos de concurrencia. Estas abstracciones facilitan la enseñanza y el diseño de las primitivas de sincronización.

Por ejemplo, el problema del productor-consumidor (también conocido como el problema del buffer limitado). En este problema se involucran dos procesos que comparten un almacén (*buffer*) de tamaño fijo. El productor coloca información en el almacén, mientras que el consumidor la obtiene del mismo.

El problema surge cuando el productor desea colocar un nuevo elemento en el almacén, pero éste está totalmente lleno. La solución para el productor es irse a dormir para ser despertado cuando el consumidor ha eliminado uno o más elementos. De la misma manera, si el consumidor desea sacar un elemento del almacén y éste está vacío, se va a dormir hasta que el productor coloca elementos en el almacén y lo despierta.

Y así con el problema anterior existen mas tales como: Filósofos Comensales, Lectores y Escritores, entre otros. Problemas en los cuales los diferentes procesos deben sincronizarse para resolver adecuadamente las situaciones de competencia por recursos compartidos.

4 PRIMITIVAS DE SINCRONIZACIÓN

Para coordinar los procesos secuenciales de una aplicación concurrente es necesario especificar comunicación entre los mismos. Esta comunicación permite sincronizar sus tareas bajo un orden requerido. Existen varios métodos para llevar a cabo la comunicación entre procesos. Su conveniencia depende de la arquitectura particular donde se utiliza. Por ejemplo, la comunicación a través de variables compartidas puede ser la mas conveniente para maquinas de memoria compartida, mientras que la comunicación a través de paso de mensajes es más conveniente para maquinas de memoria distribuida.

Para llevar a cabo la sincronización correctamente se tienen las siguientes primitivas de sincronización: semáforos, monitores, paso de mensajes, entre otras. El uso de cada una de ellas depende de los requerimientos de cada aplicación en particular.

5 EL LENGUAJE JAVA

Java es un lenguaje de alto nivel que ha tenido una gran aceptación en el desarrollo de aplicaciones distribuidas, ya que cuenta con grandes herramientas que son de gran ayuda al momento de desarrollar dichas aplicaciones. Java es un lenguaje que es compilado e interpretado de forma simultánea [4]. Pues cuando se compila una aplicación en Java se genera el llamado *ByteCode*, un código intermedio entre el lenguaje maquina del procesador y Java. Este código no es ejecutable por si mismo en ninguna plataforma hardware. Por lo que es necesario disponer de la Maquina Virtual de Java (JVM) que es capaz de ejecutar el *ByteCode*.

Por lo tanto, Java es portable pues tan solo se debe tener la JVM para la plataforma deseada y se podrá ejecutar el programa Java en cualquier plataforma sin modificación alguna.

Java esta basado en el lenguaje C++, siendo este uno de los lenguajes de programación de mayor aceptación en el mundo. Pero Java elimina muchas de las características de C++, características que no son muy útiles y añade nuevas características tales como el reciclador de memoria dinámica, para que de esta manera sea más fácil la tarea de programar. Java soporta programación concurrente, que se puede llevar a cabo de forma distribuida (en varias máquinas a la vez) a través de Sockets o Paso de Mensajes. O bien, en una sola máquina mediante la implementación de la clase Thread o la clase Runnable.

6 TEMPO

En Tempo la especificación de la sincronización se lleva a cabo mediante variaos operadores que se combinan bajo una lógica de primer orden. Los operadores Tempo son:

" < "	Precedencia	" , "	Conjunción
" ; "	Disyunción	"←"	Implicación Lógica
" + "	Ocurrencia		

Con estos operadores se definen restricciones temporales entre eventos. Un evento particular se puede ejecutar solo cuando se satisfacen las restricciones que lo incluyen. Así, con Tempo se describen explícitamente grupos de eventos ordenados. Por ejemplo, en el problema de los filósofos comensales, un filósofo pasa por los estados pensando, hambriento y comiendo, en este orden y así, vuelve a pasar por los mismos estados, en el mismo orden, indefinidamente.

El siguiente esquema representa mas claramente la los estados por los que pasa el filósofo.

Eventos: H C P
----------+----------+----------+---------
Estados: Pensando Hambriento Comiendo Pensando

La transición entre los estados se representa por los eventos H, C y P que denotan el inicio de un estado y el final de otro. Los eventos H, C y P representan la transición de estar pensando a estar hambriento, de estar

hambriento a estar comiendo y de estar comiendo a estar pensando, respectivamente. En Tempo los primeros tres eventos H, C y P, junto con su orden de ejecución se pueden representar de la siguiente manera:

$$\text{filosofo(H,C,P)} \leftarrow \text{H} < \text{C}, \text{C} < \text{P}$$

La anterior es la especificación de una restricción de precedencia en Tempo. Donde la parte izquierda al símbolo de implicación lógica (←) se conoce como la cabecera o predicado de la restricción. El predicado solo declara cuales eventos participan en la restricción. La parte derecha se conoce como cláusula y es el cuerpo o regla de la restricción, esta parte especifica la relación de precedencia entre los eventos.

Las ocurrencias de eventos más interesantes son las que comprenden un número grande o infinito de eventos. A continuación presentamos el comportamiento indefinido de un reloj. El reloj hace *tic* por siempre, produciendo una secuencia de ejecución infinita; es decir, que se ejecuta tic1, tic2, tic3, etc. Las restricciones que definen su comportamiento son:

$$\text{reloj(tic1)} \leftarrow \text{tic1} < \text{tic2}, \text{reloj(tic2)}$$

En el ejemplo anterior se ve que en el cuerpo o regla de una restricción se pueden especificar las ocurrencias de otros eventos usando las variables del mismo cuerpo. Pero esta manera de especificar eventos infinitos es un tanto complicada. Por lo que Tempo permite a un evento asociarse con uno o más eventos similares, llamados sucesores. Los sucesores de tic son tic+, tic++, etc. Y son implícitamente precedidos por tic. Las restricciones que denotan lo anterior son:

$$\text{reloj(tic)} \leftarrow \text{reloj(tic+)}$$

La ocurrencia N de un evento X se representa por X(+N). Entonces, si queremos que el reloj haga N veces tic, las restricciones quedan de la siguiente manera:

$$\text{reloj(tic)} \leftarrow \text{reloj(tic+N)}$$

```
Class Pila {
    static final int MAX = 100;
    int pos = -1
    Object [] contents = new Object [MAX];

    public Object pop(){
        < a1 >
        return contents[pos--];
        < a2 >
    }
    public void push( Object e){
        < a3 >
        return contets[++pos]= e;
        < a4 >
    }

}
```

restricciones Tempo

mutex(a4,a1)
$\text{mutex(A,B)} \leftarrow A < B, \text{mutex(A+,B+)}$

Figura 2 Sincronización de procesos Tempo/Java

Tempo solo permite especificar la sincronización entre procesos. Por lo tanto, debe usarse con otro lenguaje en el que se especificarán los procesos secuenciales de la aplicación concurrente y a estos procesos se les liga el intérprete de Tempo el cual debe estar codificado en el mismo lenguaje. Para que pueda existir la comunicación entre ambas partes. La Figura 2 muestra como sería una aplicación Tempo/Java.

En la Figura 2 se pueda ver claramente que las restricciones de Tempo aseguran que ejecute a4 antes de ejecutar a1, lo que nos indica que primero se colocara un objeto en la pila y posteriormente se sacara de la misma en un ciclo infinito.

7 TEMPO VISUAL

Como pudimos observar en la Figura 2, con Tempo podemos especificar la sincronización de los procesos secuenciales y tener una aplicación concurrente, de forma relativamente fácil. Pero esto es por que solo son dos procesos y los puntos de sincronización son pocos.

Sin embargo, en una aplicación con más procesos y con más puntos de control, las reglas serán mas complejas, lo que puede provocar que el usuario de un mal uso a Tempo y concluya que Tempo no vale la pena pues su uso no es tan sencillo [5].

TV es una interfaz gráfica que facilita el uso de Tempo, ya que el usuario solo deberá definir sus procesos secuenciales y escoger que tipo de restricción quiere aplicar. TV tiene un menú con los tipos de restricción más comunes. El programador escoge el tipo de restricción a aplicar y TV lo guía mediante instrucciones claras para que obtenga un programa concurrente fácilmente. Además, el programador podrá ver que efectivamente su programa concurrente corre correctamente pues desde TV puede mandar ejecutar el programa.

Con TV el programador ni si quiera tiene que saber como funciona Tempo. El programador solo debe saber que tipo de restricción quiere aplicar. Pues TV agrega al código de los procesos secuenciales los marcadores *tempo.check()* en el lugar correspondiente, así como las reglas y restricciones para que el interprete Tempo pueda llevar a cabo la sincronización correctamente.

El programador debe definir los procesos secuenciales (en Java), pues TV toma como modelo Tempo el intérprete desarrollado por Lee Hong [6], el cual está desarrollado en Java.

8 DISEÑO E IMPLEMENTACIÓN

Para llevar a cabo la implementación de TV se utilizo el lenguaje de programación Qt, un lenguaje orientado a objetos que esta basado en el C++ [7]. Qt ofrece una gran variedad de herramientas que facilitan el desarrollo de aplicaciones gráficas. Lo que facilitó el desarrollo de TV.

TV consta de varios módulos en los cuales se basa para convertir un programa secuencial en uno concurrente. Los módulos más importantes son los siguientes:

- **Abrir:** Abre el archivo secuencial.java para convertirlo en uno concurrente con Tempo.
- **Menú de Restricciones:** En el se encuentran las restricciones más comunes. Por ejemplo: productor-consumidor, exclusión mutua, entre otras. En este módulo se agregan las reglas de Tempo según se haya escogido.
- **Botón Siguiente:** este módulo sirve para ir guiando al programador, durante el proceso de conversión, por ejemplo, le indica que partes de código debe seleccionar con el mouse para agregarle los marcadores de Tempo y llevar a cabo el linkeo con el intérprete Tempo.
- **Instrucciones Tempo:** Este módulo muestra las instrucciones que el programador debe seguir, las instrucciones varían dependiendo del tipo de restricción seleccionado.
- **Corre Tempo:** Este módulo ejecuta el archivo que se haya abierto, abriendo una pantalla de comando para que pueda verse la ejecución del programa, cabe señalar que el archivo debe de contener ya los marcadores de *tempo.check()*, así como el likeo al interprete Tempo. Además, deberá estar en el mismo directorio que TV para pueda correr, pues en el se encuentra el intérprete Tempo.

La Figura 3 muestra la interfaz gráfica de TV, en la cual se lleva a cabo la sincronización de los programas secuenciales.

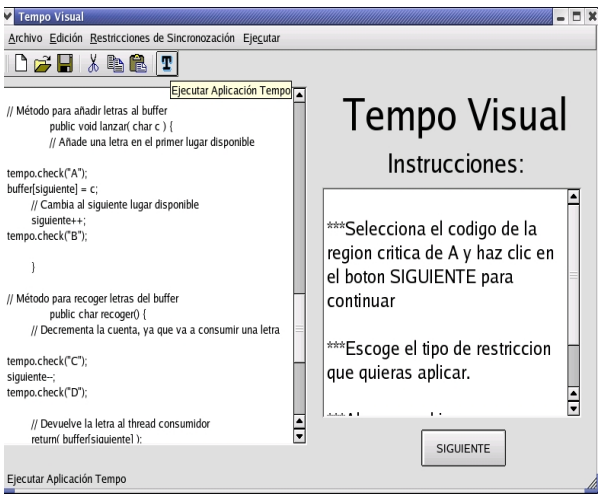

Figura 3 Interfaz gráfica de TV.

9 CONCLUSIONES

TV facilita la programación de programas concurrentes usando Tempo como lenguaje de sincronización. Pues Tempo facilita la programación concurrente, ya que con Tempo el programador toma por separado la programación de los procesos secuenciales y las sincronización de los mismos.

Sin embargo, el uso de Tempo no es del todo sencillo, particularmente la especificación de las reglas. Pues el usuario debe de conocer bien como funciona Tempo para poder desarrollar correctamente su aplicación concurrente.

Es por todo esto que desarrollamos TV para que el usuario no tenga que preocuparse por la parte de sincronización, ni siquiera tendrá que saber como funciona Tempo, pues con solo seleccionar el tipo de restricción que desee aplicar, TV lo guiará para que obtenga una aplicación concurrente ya corriendo.

10 REFERENCIAS

[1] Mordechar, B. A. Principles of Concurrent and Distributed Programming. Prentice Hall International, 2da Ed. 1990.
[2] Ramírez, R. A logic-based concurrent object-oriented programing language. Thesis submitted to the University of Bristol in accordance with the requirements for the degree of Doctor of Philosophy in the Faculty of Engineering, Department of Computer. 1997.
[3] Ramírez, R. Concurrent programming made easy. In Proceedings of the 6th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS). 2000.
[4] Froufe, A. JAVA 2 Manual de usuario y Tutorial, Alfaomega, 2da Ed. 2000.
[5] Ruiz Conejo, R. Tempo Visual. Tesis de Maestría, Facultad de Informática Mazatlán, Universidad Autónoma de Sinaloa, México. 2004.
[6] Wei Hong, L. Declarative Concurrent Programming in Java. National University of Singapore, 2000.
[7] Deitel, H. M.; Deitel, P. J. Como programar en C / C++, Prentice Hall Hispanoamericana, S. A., México. 1995.