Diseño y construcción de programas mediante CSP.

Dr. Hector Ruiz Barradas[1] & M. en C. Manuel Aguilar Cornejo[2]

RESUMEN

En este artículo presentamos, a través de un caso de estudio, el desarrollo de programas paralelos mediante métodos formales. El caso de estudio es el núcleo básico de un sistema operativo multitarea, y el método formal utilizado es la teoría de procesos secuenciales comunicantes, CSP.

CSP es una teoría de programación que permite modelar sistemas mediante procesos comunicantes. Los procesos se modelan mediante un conjunto de eventos observables del sistema a implantar. CSP dispone de un lenguaje de especificación que permite describir el comportamiento de los procesos a través de sus eventos observables. La especificación inicial de un sistema debe ser lo suficientemente abstracta para indicar lo que el sistema *debe* hacer, sin dar detalles de *cómo* lo hace. Para dar tales detalles, una especificación debe *refinarse* de manera paulatina; esto se hace a través de *pasos de refinamiento.* En cada paso de refinamiento se debe probar que el refinamiento satisface los criterios de la especificación original; para dar tal garantía, CSP dispone de un conjunto de leyes algebraicas.

El diseño y construcción del núcleo básico de un sistema operativo inicia con la especificación del entrelazamiento de acciones que modela nuestra intuición del paralelismo de un monitor multitareas. Esta especificación es refinada para itroducir los mecanismos necesarios para implantar la conmutación entre tareas del monitor. Se da la prueba formal de ciertos refinamientos y en otros, donde la complejidad aumenta, utilizamos FDR, un sistema que de manera automática, verifica la corrección de los refinamientos.

I. Introducción.

Los programas utilizados en sistemas críticos de control y comunicación están continuamente creciendo en tamaño y complejidad. Esto ha provocado que grupos de académicos e industriales se preocupen más por la *corrección* de este tipo de programas, esto es, que los programas cumplan con sus especificaciones. La necesidad de esta tendencia resulta evidente cuando errores en la programación provocan enormes pérdidas humanas o materiales. En los últimos años, se han explorado muchos métodos que garanticen la corrección de los programas. Los métodos que actualmente gozan de mayor popularidad son los denominados m*étodos formales.* Estos son métodos que utilizan matemáticas para el diseño, construcción y verificación de programas[1]

Un método formal consta de un lenguaje de especificación y de un sistema de prueba. El lenguaje de especificación permite describir las propiedades o comportamientos de un programa a diferentes niveles de abstracción, y mediante el sistema de prueba se garantiza, de manera formal, que las propiedades o comportamientos descritos a cada nivel, satisfacen las especificaciones del nivel más abstracto.

---

[1] Profesor investigador del Departamento de Electrónica de la Universidad Autónoma Metropolitana Azcapotzalco, México, D. F.
[2] Profesor investigador del Departamento de Ingeniería Eléctrica de la Universidad Autónoma Metropolitana Iztapalapa, México, D. F.

Existen diferentes métodos formales; en el presente artículo utilizamos un método basado en un *álgebra de procesos* denominado CSP ---*Communication Sequential Processes*--- . La razón fundamental de ésta decisión fue buscar un método que tuviera amplia aceptación y que además, proporcionara ambientes computarizados para el desarrollo de aplicaciones. CSP es una teoría de comunicación de procesos secuenciales desarrollada por C.A.R. Hoare en 1985 [2]

En este artículo nos interesamos en la especificación de programas típicos en aplicaciones de control y comunicaciones. Sin embargo, cualquier especificación real de esta índole está fuera de nuestros alcances en tiempo y espacio. Por lo tanto, decidimos tomar como caso de estudio el núcleo de un sistema operativo multitarea. El núcleo es un programa que continuamente interacciona con su medio ambiente, y reacciona a eventos generados por éste, de manera muy similar a los programas en aplicaciones de control y comunicaciones.

El presente artículo está dividido en seis secciones. La siguiente sección presenta de manera breve la teoría CSP. La tercera sección presenta la especificación inicial del núcleo y la cuarta los refinamientos necesarios para tener una especificación con los suficientes detalles de implantación. En la quinta sección discutimos la implantación de la especificación refinada. Por último, en la última sección damos nuestros resultados y conclusiones.

II. La teoría CSP.

CSP son las siglas de *Communicating Sequential Processes*, una teoría para la especificación, diseño e implantación de sistemas computacionales, que contínuamente actúan e interactúan con su medio ambiente. La idea básica es que esos sistemas puedan ser descompuestos en subsistemas, los cuales operen entre si concurrentemente, así como con su medio ambiente.

Dentro de la teoría CSP podemos distinguir al lenguaje CSP, las leyes algebraicas que gobiernan las combinaciones de dichos operadores y la noción de refinamiento. En los siguientes párrafos, después de presentar una visión general de la teoría, presentamos los principales operadores de CSP.

CSP es una teoría que permite modelar el patrón de comportamiento de cualquier objeto. A dicho patrón de comportamiento se le conoce con el nombre de proceso. El comportamiento se modela a través de eventos observables. Los procesos evolucionan de un estado a otro comprometiéndose en dichos eventos, que se consideran instantáneos. Los procesos pueden componerse por operadores que requieren sincronización en algunos eventos; cada componente de una composición de procesos debe poder participar en un evento dado antes de que todo el sistema pueda hacer la transición. Este tipo de sincronización, en lugar de variables de estado compartido, como es lo tradicional en los lenguajes imperativos de programación, es el mecanismo fundamental de interacción entre agentes.

La composición de procesos es en si misma un proceso, lo que permite una descripción jerárquica del sistema. Por otro lado, los componentes de un sistema pueden abstraerse en una sola entidad, gracias al operador de ocultamiento el cual hace que un conjunto dado de eventos se vean como eventos internos, es decir invisibles y ajenos al control del ambiente.

La teoría de CSP está basada en modelos matemáticos, alejados del lenguaje que permite describir los comportamientos. Esos modelos se basan en comportamientos observables

de los procesos como son las trazas, las fallas y las divergencias. Las trazas son secuencias de eventos observables. Dichos eventos corresponden a aquellos en que los procesos se comprometen en el curso de su ejecución . Un proceso determinista queda completamente modelado por el conjunto de las posibles trazas. Las fallas de un proceso corresponden a las trazas que un proceso inicialmente puede comprometerse y después de las cuales, rechaza los eventos propuestos por el ambiente, llevándolo así a una situación de candado mortal. Por último, las divergencias de un proceso corresponden a las trazas que un proceso inicialmente puede comprometerse y después de las cuales, diverge; un proceso diverge cuando éste puede realizar una secuencia infinita de acciones internas consecutivas.

El desarrollo de un sistema inicia con el enunciado de su especificación. La especificación es en si un proceso que debe refinarse hasta su implantación. La refinación se hace de manera gradual en pasos de refinamiento, donde se precisan paulatinamente los detalles de implantación. Dado que cada uno de los modelos representa a un proceso por el conjunto de sus posibles comportamientos, un refinamiento se presenta como la reducción de dicho conjunto. Si P refina a Q se escribe $P] Q$, y algunas veces subindizando $]$ para indicar qué modelo respeta el refinamiento ---i.e. el modelo de trazas, de fallas o divergencias---. De esta manera, es posible demostrar que la implantación de un sistema P_n respeta la especificación P_0 , si $\forall\ i : 0 = i < n : P]_i\ P_i$. Los diferentes procesos P_i, para i mayores que cero, corresponden a los refinamientos de la especificación inicial P_0. El refinamiento P_{i+1} se obtiene dando un mayor detalle al comportamiento observado por Pi; para satisfacer la relación de refinamiento, generalmente se abstraen, mediante la operación de ocultamiento, los nuevos eventos observados en eventos internos.

La verificación de un refinamiento depende del modelo utilizado. Sin embargo, para cualquier modelo, la prueba de un refinamiento se hace a través del sistema de prueba consistente de un conjunto de leyes algebraicas. En el presente trabajo presentamos, en los límites de lo prudente, una serie de pruebas utilizando tal sistema de prueba. Esto lo hacemo únicamente con fines didácticos para mostrar la aplicación de las leyes algebraicas. No obstante, tal tipo de pruebas sólo es pertinente cuando el grado de complejidad del refinamiento no es muy grande. En casos complejos, actualmente disponemos de una herramienta denominada FDR, la cual de manera automática verifica la corrección del refinamiento.

FDR nos ayuda en la verificación de la corrección de refinamientos, así como en la prueba de propiedades elementales como la ausencia de ciclos infinitos o de candados mortales. FDR presenta una interfaz gráfica que permite cargar archivos con las definiciones de las especificaciones y los refinamientos. Una vez verificada la sintaxis de dichas especificaciones, el usuario puede probar los refinamientos. Si al momento de desarrollar una prueba, FDR encuentra la imposibilidad de una demostración, el sistema genera la información necesaria para que el usuario se de cuenta del por qué de tal problema. FDR es una herramienta importante en el diseño de sistemas a través de CSP, que ha sido exitosamente utlizada en aplicaciones industriales en sistemas hasta con 1 000 000 estados.

El lenguaje CSP

En estos párrafos nos concentraremos en la descripción de los operadores utilizados por la teoría CSP para describir los procesos.

Un proceso CSP se define como cualquier secuencia de eventos; de esta forma cada proceso *P*, tendrá asociado un alfabeto, $\propto P$, que denota el conjunto de eventos que *P* puede ejecutar. Los eventos los representamos con letras minúsculas y los procesos con letras mayúsculas. De acuerdo a lo anterior, veamos cuales son los operadores CSP más comunes[2,5]

$(e \rightarrow P)$ Prefijamiento: Proceso que primero ejecuta el evento *e* y después se comporta como el proceso *P*. El evento *e* funciona como guardia en el proceso.

$(e_1 \rightarrow P \ / e_2 \rightarrow Q)$ Elección: si el evento *e1* ocurre, entonces el proceso se comporta como el proceso *P*. En otro caso, si el evento *e2* ocurre, entonces el comportamiento es el de *Q*.

$(P \ // \ Q)$ Concurrencia: *P* se ejecuta de manera concurrente con Q. Los eventos que se encuentren en $\propto P$ y en el $\propto Q$ se ejecutan de manera síncrona, los demás pueden ejecutarse en cualquier orden.

in ? x Canal de entrada: el proceso espera recibir por el canal de nombre *in* un mensaje el cual será almacenado en la variable x.

out ! x Canal de salida: el proceso emite por el canal de nombre *out* el valor denotado por *x*.

P1 >> *P2* Pipes: Existe un canal que comunica *P1* con *P2*. El canal es unidireccional de *P1* a *P2*.

$(P \sqcap Q)$ No determinismo interno (*P* ó *Q*): se ejecuta cualquier de los dos procesos. La elección se hará de manera interna y aleatoria .

$(P \square Q)$ No determinismo externo: Se ejecutará *P* ó *Q*, dependiendo del primer evento *e* que ofrezca el ambiente. Si *e,* es el primer evento de *P*, se ejecuta *P*; pero si es el primer evento de *Q,* se ejecuta *Q*. Si *e,* es el primer evento de ambos se ejecuta $(P \sqcap Q)$.

$(P \ /// \ Q)$ Entrelazamiento: La ejecución paralela de *P* y *Q* se ejecuta de manera entrelazada y asíncrona.

(P;Q) Composición secuencial: inicia a ejecutarse el proceso *P* y una vez que halla terminado de ejecutarse de manera satisfactoria se ejecuta *Q*.

(P//Q) Subordinación: se ejecuta *P | | Q* teniendo que el $\alpha P \subseteq \alpha Q$. De tal forma que *P* espera sincronizarse con *Q* en los eventos $(\alpha P \cap \alpha Q)$. De esta forma *P* esta subordinado a *Q* y éste puede ejecutar libremente los eventos $\alpha Q - (\alpha P \cap \alpha Q)$.

$(P \nabla Q)$ Interrupción: se comporta como el proceso *P* hasta que el primer evento del proceso *Q* se presente y entonces se comporta como *Q*.

III. Especificación inicial.

La idea básica en nuestra especificación de un núcleo multitareas es la de *entrelazamiento de acciones*. En efecto, para implantar la ejecución concurrente de varios programas, el procesador entrelaza las acciones de los diferentes programas para dar la impresión que todos ellos se ejecutan en forma concurrente. Por lo tanto, nuestra especificación inicial

consiste en observar una serie de eventos que corresponden al entrelazamiento de los eventos generados por dos programas denominados Corutina1 y Corutina2. El hecho de sólo interesarnos a dos programas en la especificación no es restrictivo, ya que los mecanismos necesarios para realizar el entrelazamiento de las acciones de dos programas, son los mismos que se necesitan para entrelazar las acciones de *n* programas. Los programas Corutina1 y Corutina2 corresponden a lo que puede llamarse como "programas de usuario" en un sistema de cómputo.

Sea $<m_{1,1}, m_{1,2}, m_{1,3}, ... >$ y $< m_{2,1}, m_{2,2}, m_{2,3}, ...>$ dos secuencias de eventos generadas por los programas Corutina1 y Corutina2 respectivamente. Cada evento $m_{i,j}$ puede verse a cualquier nivel de abstracción, desde ser la ejecución de una micro instrucción hasta la ejecución de un subprograma o función. Por lo tanto, la especificación inicial en CSP es:

$$Núcleo = Corrutinas_{1,1}$$

$$Corrutinas_{i,j} = m_{1,i} \rightarrow m_{2,j} \rightarrow Corrutinas_{i+1,j+1}$$

IV. Refinamientos de la especificación inicial.

La especificación inicial del proceso *Núcleo* sólo nos indica cuales son los eventos observables de la ejecución concurrente de los programas Corutina1 y Corutina2. Esto es intencional, pues la especificación inicial sólo debe concentrarse a lo que *se debe* hacer y no al *cómo* se hace. Para pasar a la implantación de una especificación en algún lenguaje de programación, debemos de *refinar* la especificación inicial, de modo que, además de decir lo que se debe hacer, se describa el cómo se hace. Sin embargo, el paso de una especificación inicial a una que contenga todos los detalles de implantación, debe hacerse en forma gradual a través de *pasos de refinamiento*. Cada paso de refinamiento se dedica a un detalle particular de la especificación. El refinamiento de una especificación debe hacerse en forma cuidadosa, de modo que no se alteren las propiedades fundamentales de la especificación inicial. Es aquí donde los métodos formales muestran su utilidad. Con la ayuda del sistema de prueba, se debe probar la corrección de cada refinamiento, de modo que la especificación que se refina no se vea alterada en cada paso de refinamiento. Los siguientes párrafos muestran los refinamientos de la especificación original.

IV.1. Primer refinamiento.

IV.1.1 Especificación del primer refinamiento

El primer refinamiento consiste en modularizar la especificación inicial mediante el enunciado explícito de dos nuevos procesos: *Corrutina1i* y *Corrutina2i*. Estos modelan el comportamiento de los programas de usuario Corutina1 y Corutina2 respectivamente. La definición de cada uno de ellos es una definición mútuamente recursiva. Los subíndices *i* en la definición de los nuevos procesos, señalan el *siguiente* evento en que éstos se comprometen ---i.e. "se ejecutan"---. El primer refinamiento de la especificación inicial es:

$$Núcleo1 = Corrutina1_1_1$$

$$Corrutina1_1_i = m_{1,i} \rightarrow Corrutina1_2_i$$

$$Corrutina1_2_i = m_{2,i} \rightarrow Corrutina1_1_{i+1}$$

En donde el alfabeto de *Núcleo1* es el mismo que el de *Núcleo* y está dado por la siguiente expresión:

$\propto Núcleo1 = \{m_{i,j} \mid i,j \geq 1\}$

IV.1.2. Prueba del primer refinamiento

A continuación probaremos la correctitud del presente refinamiento, esto lo llevaremos a cabo mediante las leyes del álgebra de procesos CSP.

Dado que ambos procesos son deterministas, para probar que Núcleo1 es un refinamiento de Núcleo, bastará probar que:

i) $\propto Núcleo = \propto Núcleo1$
ii) *trazas(Núcleo) = trazas(Núcleo1)*

Pero de sus definiciones, observamos que la primer condición se cumple de manera obvia, pues ambos procesos tienen definidos sus alfabetos, y estos son los mismos. Por lo cual, sólo nos resta probar que las trazas de ambos procesos son las mismas.

Prueba.

ii) Por demostrar que:

$Trazas(Núcleo) = Trazas(Núcleo1)$

O bien que:

$Trazas(Corrutina_{1,1}) = Trazas(Corrutina1_1_{1})$ < sustituyendo *Núcleo* y *Núcleo1* por sus definiciones >

Esto equivale a probar que:

$Trazas(\mu X_{i,j}{:}A.\ F(X_{i,j})) = Trazas(\mu Y_{i}{:}B.\ G(Y_{i}))$ < de las definiciones de procesos recursivos con guardias >

Dado que:

$Trazas(\mu X_{i,j}{:}A.\ F(X_{i,j})) = \bigcup_{n \geq 0} Trazas(F^{n}(STOP_{\propto Núcleo}))$ <ver la ley L5 de la sección 2.1.6.1>

y que:

$Trazas(\mu Y_{i}{:}B.\ G(Y_{i})) = \bigcup_{n \geq 0} Trazas(G^{n}(STOP_{\propto Núcleo1}))$ <ver la ley L5 de la sección 2.1.6.1>

entonces, esto equivale a probar que:

$$\bigcup_{n \geq 0} Trazas(F^{n}(STOP_{\propto Núcleo})) = \bigcup_{n \geq 0} Trazas(G^{n}(STOP_{\propto Núcleo1}))$$

Demostración:

La demostración la realizaremos por inducción sobre *n*, el número de iteraciones de los
procesos recursivos con guardias, (Ver 2.1.6.1 L5)

i) Para $n=0$ tenemos:

$$Trazas(F_0(STOP_{\alpha Núcleo})) = Trazas(G_0(STOP_{\alpha Núcleo1}))$$

$$Trazas(STOP_{\alpha Núcleo}) = Trazas(STOP_{\alpha Núcleo1}) \qquad \langle \text{def. de } G_0(X) \text{ y } F_0(X) \rangle$$

$$\{\langle\rangle\} = \{\langle\rangle\} \qquad \langle \text{ def. de trazas para } STOP \rangle$$

ii) Suponemos cierta la igualdad, para *n-1,* esto es:

$$Trazas(F_{n-1}(STOP_{\alpha Núcleo})) = Trazas(G_{n-1}(STOP_{\alpha Núcleo1}))$$

iii) Ahora, en base a tal suposición demostremos la igualdad para *n* iteraciones.

Por demostrar que:

$$Trazas(F_n(STOP_{\alpha Núcleo})) = Trazas(G_n(STOP_{\alpha Núcleo1})).$$

Partamos del lado izquierdo de la igualdad:

$$Trazas(F^n(STOP_{\alpha Núcleo}))$$

< Desarrollando $F^n(STOP_{\alpha Núcleo})$ tenemos: >

$$= Trazas(F(F^{n-1}(STOP_{\alpha Núcleo})))$$

< sustituyendo la definición de $F(F^{n-1}(STOP_{\alpha Núcleo}))$, tenemos: >

$$= Trazas(\, m_{1,i-1} \rightarrow m_{2,j-1} \rightarrow F^{n-1}(STOP_{\propto Núcleo})\,)$$

< de la definición de trazas de procesos 2.1.6 L4, tenemos: >

$$= \{\langle\rangle, \langle m_{1,i-1} \rangle, \langle m_{1,i-1}, m_{2,j-1} \rangle\} \cup Trazas(\, F^{n-1}(STOP_{\propto Núcleo})\,)$$

< Por hipótesis de inducción, tenemos: >

$$= \{\langle\rangle, \langle m_{1,i-1} \rangle, \langle m_{1,i-1}, m_{2,j-1} \rangle\} \cup Trazas(G^{n-1}(STOP_{\propto Núcleo1})\,)$$

< formando un proceso con las trazas anteriores, tenemos: >

$$= Trazas(\, m_{1,1} \rightarrow m_{2,1} \rightarrow G^{n-1}(STOP_{\propto Núcleo1})\,)$$

< de acuerdo a la definición de $G(G^{n-1}(STOP_{\propto Núcleo1})\,)$, tenemos: >

$$= Trazas(G(G^{n-1}(STOP_{\alpha Núcleo1})))$$

$$= Trazas(G^n(STOP_{\alpha Núcleo1})).$$

< Por lo cual: >

$$Trazas(F^n(STOP_{\alpha Núcleo})). = Trazas(G^n(STOP_{\alpha Núcleo1})).$$

Con esto terminamos la demostración por inducción, con lo cual podemos concluir que efectivamente *Núcleo1* es un refinamiento de *Núcleo* □

IV.1.3. Motivación al segundo refinamiento

La modularización de la especificación inicial en el primer refinamiento, nos permite especificar el comportamiento del núcleo multitareas en función del comportamiento de los programas de usuario. Sin embargo, en este refinamiento no aparecen los "programas del sistema" que le permiten a los programas de usuario realizar el entrelazamiento de sus acciones ---i.e. compartir el uso de la CPU entre diferentes procesos---. Por lo tanto, en el segundo refinamiento, tenemos como objetivo la especificación de un mecanismo que, de manera *explícita*, le permita a un programa la transferencia de control del procesador; a este mecanismo le denominamos Transfer.

IV.2. Segundo Refinamiento

IV.2.1 Especificación del segundo refinamiento

El comportamiento del mecanismo Transfer lo modelamos a través del proceso *Transfer*$_{i,j}$. Este es utilizado en el comportamiento de *Corrutina1*$_i$ y *Corrutina2*$_i$ para modelar la transferencia de control del procesador entre los procesos. El subíndice *j* del proceso Transfer indica qué corrutina gana el control del procesador, mientras que el subíndice *i* señala el siguiente evento a desarrollarse en el programa al que se le transfiere el control. De esta manera *Transfer*$_{i,j}$ modela el mecanismo que le permite a un programa ceder el control del procesador en forma explícita.

El segundo refinamiento de la especificación inicial es:

$$Núcleo2 = Corrutina2_1_1$$

$$Corrutina2_1_i = m_{1,i} \rightarrow Transfer2_{i,2}$$
$$Corrutina2_2_i = m_{2,i} \rightarrow Transfer2_{i,1}$$

$$Transfer2_{i,j} = (\ if\ j=1\ then\ Corrutina2_1_{i+1}\ else\ Corrutina2_2_i\ )$$

En donde:

$$\alpha Núcleo2 = \{ m_{i,j} \mid i,j \geq 1 \}$$

IV.2.2. Prueba del segundo refinamiento

Antes de probar que *Núcleo2* es un refinamiento correcto de *Núcleo1*, reescribamos la definición de los procesos, para ello partamos de la definicón de *Núcleo2*.

$$Núcleo2 = Corrutina2_1_i \qquad i=1$$

<de la definición de $Corrutina2_1_i$, tenemos >

$= m_{1,i} \rightarrow Transfer2_{i,2} \qquad i=1$

<sustituyendo $Transfer2_{1,2}$ por su definición>

$= \mathrm{m}_{1,i} \rightarrow Corrutina2_2_i \qquad i=1$

<de la definición de $Corrutina2_2_i$ tenemos: >

$= \mathrm{m}_{1,i} \rightarrow m_{2,i} \rightarrow Transfer2_{i,1} \qquad i=1$

<sustituyendo $Transfer2_{1,1}$ por su definición >

$= \mathrm{m}_{1,i} \rightarrow \mathrm{m}_{2,i} \rightarrow Corrutina2_1_{i+1} \qquad i=1$

<Por lo tanto: >

$Núcleo2 = Corrutina2_1_i = \mathrm{m}_{1,i} \rightarrow \mathrm{m}_{2,i} \rightarrow Corrutina2_1_{i+1} \qquad i=1$

En donde claramente se ve que *Nucleo2* está definido en términos de un proceso recursivo con guardias.

Por otro lado tenemos de la definición de *Núcleo1* que:

$Núcleo1 = Corrutina1_1_i \qquad i=1$

<de la definición de $Corrutina1_1_i$, tenemos: >

$= \mathrm{m}_{1,i} \rightarrow Corrutina1_2_i \qquad i=1$

<de la definición de $Corrutina1_2_i$ tenemos: >

$= \mathrm{m}_{1,i} \rightarrow \mathrm{m}_{2,i} \rightarrow Corrutina1_1_{i+1} \qquad i=1$

<Por lo tanto: >

$Núcleo1 = Corrutina1_1_i = \mathrm{m}_{1,i} \rightarrow \mathrm{m}_{2,i} \rightarrow Corrutina1_1_{i+1} \qquad i=1$

En donde claramente se puede observar que *Núcleo1* también está definido en términos de un proceso recursivo con guardias.

Prueba

Para probar que *Núcleo2* refina a *Núcleo1* debemos probar que:

i) $\alpha Núcleo1 = \alpha Núcleo2$
ii) $trazas(Núcleo1) = trazas(Núcleo2)$

Pero i) es evidente que se cumple, pues ambos alfabetos fueron definidos explícitamente y son los mismos, por lo cual, sólo debemos probar que las trazas de ambos procesos son las mismas, esto es:

$trazas(Núcleo1) = trazas(Núcleo2)$

sustituyendo *Núcleo1* y *Núcleo2* por sus definiciones, tenemos:

$Trazas(Corrutina1_1_1) = Trazas(Corrutina2_1_1)$

Dado que ambos procesos fueron definidos de manera recursiva con guardias, entonces, lo que tenemos que probar es que:

$Trazas(\mu X_i{:}A.\ F(X_i)) = Trazas(\mu Y_i{:}B.\ G(Y_i))$

bien que:

$$Trazas(\mu X_i{:}\{m_{i,j} \mid i,j \geq 1\}.(m_{1,i-1} \rightarrow m_{2,i-1} \rightarrow X_i) =$$
$$Trazas(\mu Y_i{:}\ \{m_{i,j} \mid i,j \geq 1\}.(m_{1,i-1} \rightarrow m_{2,i-1} \rightarrow Y_i)$$

Dado que dos procesos recursivos con guardias son iguales si: tienen el mismo alfabeto, y sus ecuaciones recursivas con guardias son similares (ver. sección 2.1.3), por lo tanto la igualdad de arriba es correcta, con lo que nos permite concluir que *Núcleo2* es un refinamiento correcto de *Núcleo1*. □

IV.2.3. Motivación al tercer refinamiento

La utilización de los subíndices *i* y *j* en la definición del mecanismo Transfer, provocan que el programador de una aplicación concurrente tenga en cuenta a cada momento que programa debe tomar el control de la CPU y a partir de donde éste debe reasumirse. La administración de esos subíndices aumenta la complejidad de una aplicación, a medida que ésta es compuesta de un número considerable de programas cooperantes. Esta restricción es resuelta en el siguiente refinamiento.

IV.3. Tercer Refinamiento

IV.3.1 Especificación del tercer refinamiento.

El objetivo de este refinamiento es refinar el mecanismo Transfer, de modo que el programador no deba cuidar cual es el programa que se ejecuta, ni a partir de cual instrucción. Para ello se introducen tres nuevos procesos en la especificación: *Cola1* y *Cola2* que modelan dos colas e *Init*, que permite dar valores iniciales a las colas modeladas. El proceso *Cola1* lleva el control de que programa debe ejecutarse en un momento dado; *Cola2* guarda el número de instrucción que debe ejecutarse por el programa que gana el control del procesador. Con este refinamiento, podemos decir que *Cola1* modela el comportamiento de un "planificador" ---*scheduler*---, mientras que *Cola2,* modela el comportamiento de una "pila" que guarda el apuntador a la siguiente instrucción a ejecutarse. En este refinamiento, el proceso *Transfer* que modela el mecanismo Transfer, ya no utiliza subíndices para la administración del procesador y se comunica con los

procesos *Cola1* y *Cola2* para obtener dicha información. A continuación se presenta la especificación completa de este refinamiento.

La especificación de *Núcleo3* contiene la composición secuencial de *Init* y T*ransfer*, los cuales subordinan a los procesos *Cola1* y *Cola2*. *Cola1* y *Cola2* se ejecutan de manera concurrente e independiente entre ellos; además éstas se ejecutan de manera concurrente y síncrona con (*Init ; Transfer)*:

$$Núcleo3 = (\ (Cola1{:}Queue\,|||\,Cola2{:}\ Queue)\ //\ (Init_{<1,2>}\ ;\ Transfer3)\ )\ \backslash\ \surd$$

El proceso *Init* inicializa las colas (de acuerdo a la sublista que tiene asociada) y termina satisfactoriamente. Todos los eventos ejecutados por *Init* (salvo la terminación satisfactoria, Ö) son también eventos de las colas, por lo cual no pertenecen al alfabeto de *Núcleo3* (def. de procesos subordinados):

$$Init_{<x>\hat{}s} = Cola1.left!x \rightarrow Cola2.left!1 \rightarrow Init_s$$
$$Init_{<>} = SKIP$$

$$
\begin{aligned}
Queue &= S_{<>} \\
S_{<>} &= left?x \rightarrow S_{<x>} \\
S_{<x>\hat{}s} &= (\ \ (\ left?y \rightarrow if\ \ \#<x>\hat{}s\hat{}<y> < n \quad then \quad S_{<x>\hat{}s\hat{}<y>} \\
&\qquad\qquad\qquad else\ (right!x \rightarrow S_{s\hat{}<y>}) \\
&\qquad\ ) \\
&\qquad\ \square \\
&\qquad\ (\ right!x \ \rightarrow S_{<s>}\ ) \\
&\quad)
\end{aligned}
$$

$$Corrutina3_1_i = m_{1,i} \rightarrow Transfer3$$
$$Corrutina3_2_i = m_{2,i} \rightarrow Transfer3$$

El proceso *Transfer* siempre invoca al proceso cuyo identificador se encuentra al frente de *Cola1,* y transmite el número de evento a ejecutarse en ese proceso; éste lo obtiene del frente de *Cola2*. El valor recibido de *Cola1* se vuelve a encolar, al igual que el de *Cola2*, pero este último incrementado en 1, para que se ejecute el siguiente evento.

$$
\begin{aligned}
Transfer3 = {} & Cola1.right?j \rightarrow Cola2.right?i \rightarrow Cola1.left!j \rightarrow Cola2.left!i+1 \rightarrow \\
& \rightarrow (\ if\ \ j=1\ \ then\ Corrutina3_1_i \\
& \qquad else\ \ Corrutina3_2_i \\
& \quad)
\end{aligned}
$$

Cabe señalar que en la definición de *Núcleo3* ocultamos el evento $\surd$, ya que no está dentro del alfabeto de la especificación anterior

IV.3.2. Prueba del tercer refinamiento.

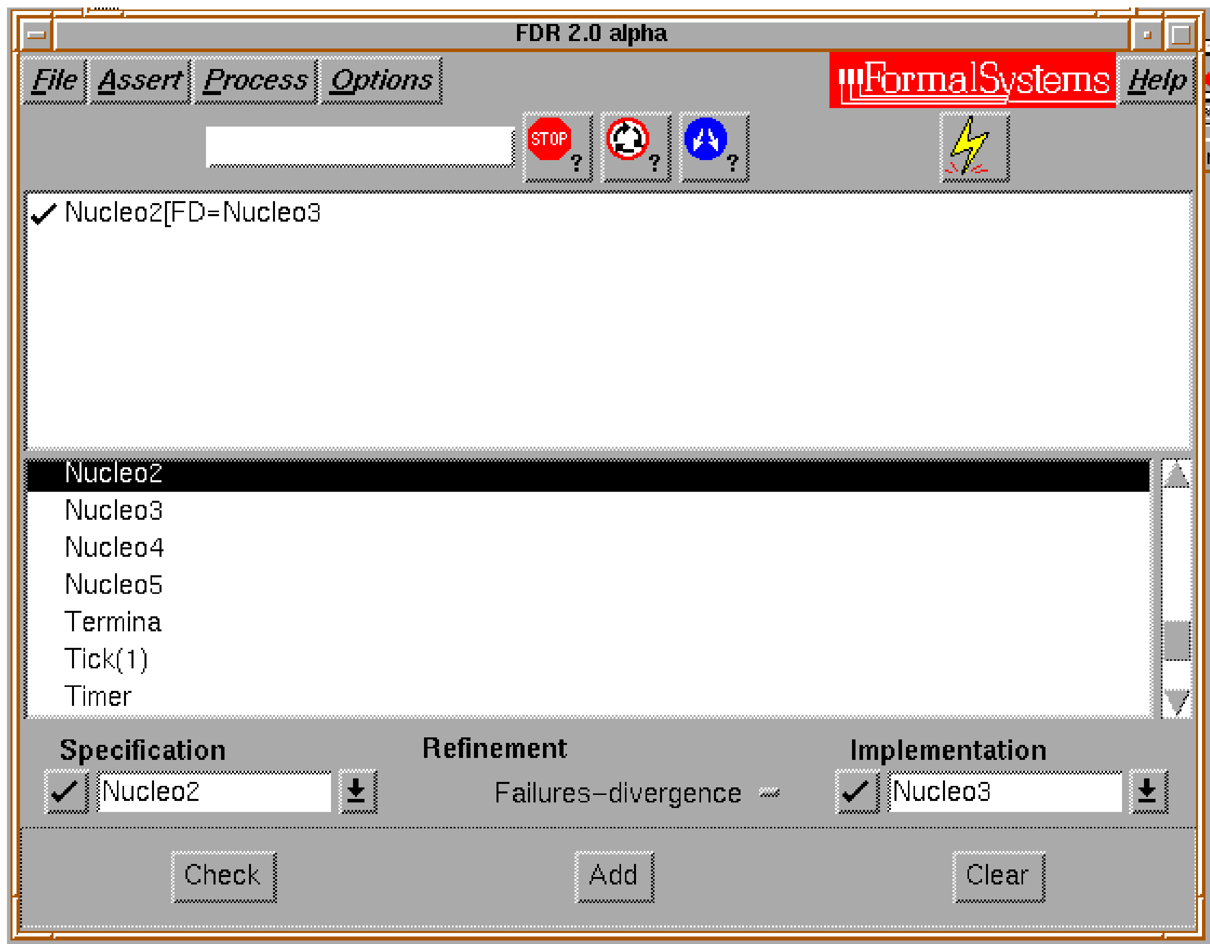
La prueba del presente refinamiento es similar al de los anteriores, pero por falta de espacio ésta es omitida. Más sin embargo, damos una breve explicación de la manera en que la prueba es hecha usando FDR.

Para utilizar FDR debemos traducir el segundo y tercer refinamiento a un *script* en el formato aceptado por éste, el cual tiene una sintaxis muy similar a la de CSP. La sintaxis del lenguaje de entrada a FDR puede consultarse en [3]. El *script* FDR obtenido de acuerdo a lo explicado anteriormente es:

```
-- ==========================================================
--                      Segundo Refinamiento
-- ==========================================================

Nucleo2 = Corrutina2_1(1)

Corrutina2_1(i) = if  i<4  then   ( m.1.i -> Transfer2(2,i) )
                  else (SKIP)

Corrutina2_2(i) = m.2.i -> Transfer2(1,i)

Transfer2(j,i) = if  j==1  then Corrutina2_1(i+1)
                 else Corrutina2_2(i)


-- ==========================================================
--                      Tercer Refinamiento
-- ==========================================================

channel c1left, c1right, c2left, c2right:{0,1,2,3,4}


-- Dado que el operador de subordinación no existe en FDR2, la subordinación fue
-- implantada como el paralelismo entre procesos con sus eventos síncronos y ocultos.

Nucleo3 = ( ( Cola1(<>)||| Cola2(<>) )
             [|{| c1left,c2left,c1right, c2right |}|]
            (Init(<1,2>) ; Transfer3 )
          ) \ {| c1left, c1right, c2left, c2right |}


Init(s) = if ( null(s) ) then (SKIP)
      else  ( c1left!head(s) -> c2left!1 -> Init(tail(s)) )

-- El tamaño de las colas lo limitamos a 4 para que la prueba entre procesos fuera rápida.

Cola1(s) =   if (s == <>) then   ( c1left?x -> Cola1(<x>) )
        else
        (
         (
          (c1left?y -> if ((#s)+1 < 4)  then Cola1(s^<y>)
                       else (c1right!(head(s)) -> Cola1(tail(s)^<y>))
          )
          []
```

```
          (c1right!(head(s)) -> Cola1(tail(s)) )
         )
         []
         SKIP
        )

-- Definimos Cola2 en función de Cola1 renombrando sus eventos.

Cola2(s) = Cola1(s)[[ c1left.n <- c2left.n, c1right.n <- c2right.n | n<-{0..4}]]

-- Limitamos el número de eventos en los cuales se comprometen los programas de usuario

Corrutina3_1(i) = if   i<4  then    ( m.1.i -> Transfer3  )
                  else  (SKIP)
Corrutina3_2(i) = m.2.i -> Transfer3

Suma1(x) = (x + 1) % 5

Transfer3 = c1right?j -> c2right?i -> c1left!j -> c2left!Suma1(i)->
            ( if j==1 then Corrutina3_1(i)
               else Corrutina3_2(i)
            )


-- ============================================================
--                    Fin del 3er. refinamiento.
-- ============================================================
```

Una vez que el *script* es cargado en FDR los procesos son listados en la lista de procesos. Seleccionamos el proceso *Núcleo2* mediante el selector de procesos para la especificación y seleccionamos *Núcleo3* mediante el selector de procesos para la implantación, por último seleccionamos el modelo de trazas en el cual la prueba se va a efectuar y seleccionamos el botón de verificación. El resultado de tal manipulación lo podemos observar en la siguiente figura que muestra la interfaz principal de FDR.



IV.3.3. Motivación al cuarto refinamiento

En este momento, tenemos la especificación de cuatro procesos que modelan los programas del sistema que permiten a un programador conmutar el procesador entre dos programas de usuario. El cuarto refinamiento tiene como objetivo permitir que un número mayor de programas pueda transferirse el control del procesador entre si.

IV.4. Cuarto Refinamiento

IV.4.1 Especificación del cuarto refinamiento.

En este refinamiento se generaliza el mecanismo Transfer, de modo que no se restrinja la transferencia de control del procesador a solo dos programas. Para ello se introduce la familia de procesos *Ci,j* que modelan los diferentes programas de usuario en un sistema. El proceso *Transfer* decide los valores de *i* y *j* después de comunicarse con los procesos *Cola2* y *Cola1* respectivamente. En este contexto, podemos pensar que *j* denota el "indentificador" ---*pid*--- del programa que gana el control del procesador e *i*, el valor de contador de programa a partir de donde el programa seleccionado se ejecuta. La especificación de este refinamiento es:

$$Núcleo4=((Cola1:Queue|||Cola2:Queue) \mathbin{//} (Init_{<1,2,\ldots,m>} ;Transfer4)) \setminus \surd$$

$$Init_{\langle x\rangle \frown s} = Cola1.left!x \rightarrow Cola2.left!1 \rightarrow Init_s$$
$$Init_{\langle\rangle} \;=\; SKIP$$

$$Queue \;=\; S_{\langle\rangle}$$
$$S_{\langle\rangle} \;=\; left?x \rightarrow S_{\langle x\rangle}$$
$$\begin{aligned} S_{\langle x\rangle \frown s} \;=\; (\; & (\; left?y \rightarrow if \;\; \#\langle x\rangle \frown s \frown \langle y\rangle < n \;\; then \;\; S_{\langle x\rangle \frown s \frown \langle y\rangle} \\ & \qquad\qquad else\; (right!x \rightarrow S_{s \frown \langle y\rangle}) \\ &) \\ & \square \\ & (\; right!x \;\rightarrow S_{\langle s\rangle} \;) \\ ) & \end{aligned}$$

$$\begin{aligned} Transfer4 = {} & Cola1.right?j \;\rightarrow Cola2.right?i \rightarrow Cola1.left!j \rightarrow Cola2.left!i{+}1 \\ & \rightarrow C_{j,i} \setminus \{m_{i,j} \mid i>2 \wedge j\geq 1\} \end{aligned}$$

$$C_{j,i} :\; Corrutina4_{j,i}$$
$$Corrutina4_{j,i} \;=\; m_{j,i} \;\rightarrow Transfer4$$

En donde el alfabeto de *Núcleo4* es exactamente el mismo que el de *Núcleo3.*

Nótese que el único proceso que cambió en este refinamiento es el proceso *Transfer4*, ahora en vez de invocar sólo a dos procesos de usuario invoca a *n.* Por otro lado podemos ver que *Transfer4* oculta los eventos de los procesos de usuario que fueron introducidos en este refinamiento, esto tiene como finalidad facilitar la prueba de la corrección de este refinamiento, pero de ninguna manera implica que los programas de usuario no se ejecuten. El ocultamiento de estos eventos debe considerarse únicamente para fines de la prueba de corrección del refinamiento.

La prueba de éste y los siguientes refinamientos se hicieron mediante FDR, de manera similar a la prueba del tercer refinamiento. A partir de ahora por falta de espacio las pruabas serán omitidas. El lector interesado puede consultar [6] para ver los detalles de tales pruebas.

IV.3.3. Motivación al quinto refinamiento

Tenemos hasta el momento la especificación de un mecanismo general que permite a los programadores transferir de manera *explícita* el control del procesador a cualquier programa. Este mecanismo sirve para implantar una facilidad de programación conocida como *corrutina*. Esta es una facilidad de programación similar al de subrutinas. La diferencia fundamental es que el llamado a una corrutina, transfiere el control de un programa a un punto de la corutina, que no es necesariamente el inicio. El llamado a una subrutina siempre transfiere el control del programa al inicio de la subrutina.

La programación paralela mediante corutinas se vuelve difícil a medida que el número de corutinas aumenta, dado que el programador, ademas de tratar con la lógica de su problema, debe cuidar la manera en que el procesador se conmuta de corrutina a corrutina. Es en este sentido donde se propone el último refinamiento. El programador de una aplicación paralela, solo se concentra a su problema y deja que los programas del sistema decidan a que programa se le transfiera el control del procesador y el momento de hacerlo.

IV.5. Quinto Refinamiento.

Para quitar al programador, la tarea de decidir que corrutina toma el control del procesador al momento de una transferencia de control, introducimos un mecanismo de temporización denominado Timer que se especifica por el proceso *timer*. Este tiene como tarea "interrumpir" cualquier secuencia de eventos ---i.e. las corrutinas--- e invocar el mecanismo de transferencia entre corrutinas Transfer.
Este último mecanismo también es refinado, de modo que en cada transferencia de control, se reinicie el proceso *timer*. El proceso *timer* inicialmente está bloqueado, en espera de una comunicación con el proceso *Transfer*, para que le proporcione el valor del intervalo de interrupción. Una vez lograda la comunicación, el proceso *timer* genera una secuencia de eventos --- *tick*--- de longitud igual al valor comunicado por *Transfer*. Al término de dicha secuencia, la ejecución del proceso *timer* es suspendida, en espera de una nueva comunicación con el proceso *Transfer*, repitiéndose indefinidamente su comportamiento.

$$Núcleo5 = (\ (Cola1{:}Queue\,|||\,Cola2{:}Queue)\ //\ ((Init_{<1,2,\ldots,m>}\ ;Transfer5)\ //\ (Timer))\ )\ \setminus \{\surd, tick, Timer.left, Timer.right\}$$

$$Init_{<x>\hat{}s} = Cola1.left!x \rightarrow Cola2.left!1 \rightarrow Init_s$$
$$Init_{<>} = SKIP$$

$$Queue = S_{<>}$$
$$S_{<>} = left?x \rightarrow S_{<x>}$$
$$S_{<x>\hat{}s} = (\ (\ left?y \rightarrow if\ \ \#<x>\hat{}s\hat{}<y> < n\ \ then\ \ S_{<x>\hat{}s\hat{}<y>}\ \ else\ (right!x \rightarrow S_{s\hat{}<y>})\ )\ \square\ (\ right!x \rightarrow S_{<s>}\ )\ )$$

$$Transfer5 = (\ Cola1.right?j \rightarrow Cola2.right?i \rightarrow Cola1.left!j \rightarrow Cola2.left!i{+}1 \rightarrow Timer.left!1 \rightarrow C_{j,i}\ \nabla Termina\ ) \setminus \{mi,j \mid i \geq 3, j \geq 1\}$$

$$C_{j,i} : Corrutina5_{j,i}$$
$$Corrutina5_{j,i} = (\ m_{j,i} \rightarrow tick \rightarrow tick \rightarrow Corrutina5_{j,i+1}\ )$$

$$Termina = Timer.right?x \rightarrow Transfer5$$

$$Timer = left?i \rightarrow tick_i$$

$$tick_i = (\ if\ \ i>0\ \ then\ \ (tick \rightarrow tick_{i-1})\ \ else\ \ right!0 \rightarrow Timer\ )$$

V. Hacia la implantación.

Para llevar el quinto refinamiento a una implantación en una máquina con un solo procesador, es necesario eliminar los operadores de entrelazamiento, subordinación, paralelismo, etc. Al eliminar los operadores de alto nivel de este refinamiento, tendremos sólo una secuencia de eventos. Esta secuencia de eventos formará justamente nuestro programa secuencial del núcleo básico del sistema operativo multitareas. Eliminamos los operadores de alto nivel mediante la aplicación sucesiva de las leyes del álgebra de procesos CSP, dadas en [6].

La aplicación sucesiva de las leyes A.2.2.| |.4 sobre el operador | |, la ley A.3.|||.1 sobre el operador ||| y la ley A.10.comm.1 sobre los canales de comunicación entre otras, da como resultado el siguiente refinamiento:

$$
\begin{aligned}
&\mathit{Núcleo5} = \mathit{Implantación} = \\
&\quad (\textbf{\textit{Inicializa}} \;;\; \textbf{\textit{Sched(1,1)}} \;;\; \textbf{\textit{Corrut}}_{1,1} \;;\; \textbf{\textit{Interrup}} \;; \\
&\qquad \textbf{\textit{Sched(2,1)}} \;;\; \textbf{\textit{Corrut}}_{2,1} \;;\; \textbf{\textit{Interrup}} \;; \\
&\qquad (\quad (\mathit{Cola1.S}_{<3,\dots,m,1,2>} \;|||\; \mathit{Cola2.S}_{<1,\dots,1,2,2>}) \\
&\qquad\quad // \\
&\qquad\quad (\; (\mathit{Transfer5} \;||\; \mathit{Timer}\;)\;) \\
&\qquad) \\
&\quad) \;\backslash\; \{\surd, \mathit{tick}, \mathit{Timer.left}, \mathit{Timer.right}\}
\end{aligned}
$$

en donde :

$$
\begin{aligned}
\textbf{\textit{Inicializa}} = \;& \mathit{Cola1.left.1} \to \mathit{Cola2.left.1} \to \mathit{Cola1.left.2} \to \mathit{Cola2.left.1} \to \dots \to \\
& \mathit{Cola1.left.m} \to \mathit{Cola2.left.1} \to \mathit{SKIP}
\end{aligned}
$$

$$
\begin{aligned}
\textbf{\textit{Sched(i,j)}} = \;& \mathit{Cola1.right.i} \to \mathit{Cola2.right.j} \to \mathit{Cola1.left.i} \to \mathit{Cola2.left.j{+}1} \to \\
& \mathit{Timer.left.1} \to \mathit{SKIP}
\end{aligned}
$$

$$\textbf{\textit{Corrut}}_{i,j} = m_{i,j} \to \mathit{SKIP}$$

$$\textbf{\textit{Interrup}} = \mathit{tick} \to \mathit{Timer.right!0} \to \mathit{SKIP}$$

El lector interesado en la aplicación de las leyes para probar la correción del refinamiento puede consultar [6].

De este refinamiento podemos observar claramente que la derivación de código imperativo es casi inmediata. Para el mapeo de eventos CSP a instrucciones en pseudocódigo asumiremos de que disponemos de las operaciones del Tipo de Dato Abstracto Colas (TDA Colas), lo cual no resulta grave, pues estas operaciones son sencillas y fáciles de implantar. A continuación describimos las operaciones del TDA colas.

`HazNula(Cola)` : procedimiento que vacía la cola.

`ValorFrente(Cola)` : función que devuelve el valor del frente de la Cola sin alterar ésta.

`Encola(elem, Cola)` : procedimiento que inserta al final de la Cola el elemento elemento

`Desencola(Cola)` : función que retira el elemento del frente de la Cola, regresándolo a quien invocó la función.
`Vacía(Cola)` : función que devuelve cierto si la Cola está vacía, falso en caso contrario.

Utilizando las definiciones anteriores realizamos el mapeo de cada uno de los subprocesos del proceso *Implantación*.

Inicializa

este proceso es mapeado a las siguientes instrucciones en pseudocódigo

```
Encola(1,Cola1)
Encola(1,Cola2)
Encola(2,Cola1)
Encola(1,Cola2)
      .
      .
      .
Encola(m,Cola1)
Encola(1,Cola2)
```

en donde `Encola` es una función del TDA colas.

Sched(i,j)

el cual es mapeado a la siguiente serie de instrucciones en pseudocódigo.

```
i := Desencola(Cola1)
j := Desencola(Cola2)
Encola(Cola1,i)
Encola(Cola2,j+1)
InicializaReloj(1)
```

En donde la instrucción InicializaReloj se encargará de activar el reloj del sistema, el cual viene en cualquier tipo de computador, para que después de cierto tiempo interrumpa la ejecución del procesador y ejecute una rutina especial, la cual es la de suspender el programa de usuario que se ejecuta y realizar el cambio de contexto en las pilas.

$Corrut_{i,j}$

Este proceso modela, a cualquier nivel de abstracción, las instrucciones de algún programa de usuario, por lo que sus instrucciones dependerán del programa en particular a ejecutar. Con fines ilustrativos sustituiremos las instrucciones de usuario por mensajes escritos a pantalla. Dado que $Corrut_{i,j}$ fue definido por:

$Corrut_{i,j} = m_{i,j} \rightarrow SKIP$

implantaremos el proceso por

Escribe("Corrutina *i* mensaje *j* ")

en donde *i* y *j* dependerán de la corrutina que se ejecute, así como el número de instrucción que ejecute.

Interrup

Una vez activada la rutina de atención a la interrupción se ejecuta el proceso ***Sched(i,j)***

Con los mapeos anteriores constatamos que es posible transcribir la especificación de un proceso CSP a un lenguaje imperativo. Si bien el mapeo resulta un tanto laborioso, éste lo hicimos con fines didácticos. No obstante, podemos hacer un mapeo directo de una especificación CSP a *occam* [4], dado que éste es una concretización de la teoría CSP. Esta es una de las alternativas aconsejadas en la implantación de especificaciones CSP.

VI. Conclusiones.

Acabamos de presentar la derivación del núcleo de un sistema operativo multitarea, utilizando la teoría CSP. El núcleo es un ejemplo de programa que continuamente interacciona con su medio ambiente, de manera similar a los programas de control y comunicación en aplicaciones industriales. Estos programas han sido objeto de una gran investigación para asegurar la ausencia de errores, y así evitar los enormes costos que éstos ocasionan. El resultado expuesto en este artículo, es sólo una muestra de la ayuda que prestan los métodos formales en la *construcción razonada y metódica de programas*.

Es importante resaltar la utilidad de FDR en las demostraciones de correción de los refinamientos. El desarrollo de pruebas manuales en cualquier sistema de complejidad media haría imposible el uso práctico de la metodología. Sin embargo, la utilización de FDR nos permite contemplar aplicaciones complejas en ambientes industriales.

Si bien el programa derivado resulta modesto, es importante resaltar no el programa en sí, sino la manera en que éste fue obtenido. Los pasos presentados en este trabajo para la derivación del núcleo deberían ser los mismos para derivar cualquier otro programa de propósito industrial. Cabe señalar que los pasos de refinamiento presentados no son comunes en trabajos relacionados con CSP, y ésta es una originalidad del presente trabajo.

Los resultados obtenidos en este trabajo nos motivan a aplicar la teoría CSP en proyectos más ambiciosos como tolerancia a fallas en sistemas distribuidos y protocolos de comunicación entre otros. Por último, esperamos en el futuro contribuir con adecuaciones a la teoría CSP que permita un mejor manejo de datos.

Bibliografía.

[1] Wings M. Jeannette (1990), "A Specifier's Introduction to Formal Methods", Computer, IEEE, Sept. 1990, pp. 8-24.

[2] Hoare C.A.R. (1985), "Communicating Sequential Processes", Prentice Hall International Series in Computer Science, Hemel Hempstead & Englewood Cliffs.

[3] Formal Systems (Europe) Ltd. (1993), "Failures Divergence Refinement, User Manual and Tutorial ", August 1993.

[4] Inmos Limited, (1988), "occam2 Programming Manual", Prentice Hall International Series in Computer Science, Hemel Hempstead & Englewood Cliffs.

[5] Hinchey, M., Jarvis, S. (1995), "Concurrent Systems: Formal Development in CSP", McGraw Hill International Series in software engineering.

[6] Aguilar Cornejo Manuel (1997), "Especificación y diseño del núcleo básico de un sistema operativo mediante CSP", Tesis de maestría de CINVESTAV-IPN, México.

Héctor Ruiz Barradas e-mail: hrb@xochitl.uam.mx. Ingeniero en comunicaciones y electrónica por ESIME-IPN en 1987. Grado de Maestro en Ciencias, especialidad en computación por el CINVESTAV-IPN en 1988. Dr. en Computación por la universidad de Rennes I, Francia en 1993. Actualmente Profr. investigador de la UAM Azcapotzalco y del CINVESTAV-IPN. Las áreas de interés son: Sistemas Distribuidos, Protocolos y Métodos Formales.

Manuel Aguilar Cornejo e-mail: mcornejo@alpha.cs.cinvestav.mx. Lic. en Computación por la UAM Iztapalapa en 1993. Grado de Maestro en Ciencias con especialidad en computación en CINVESTAV-IPN en 1997. Exbecario de CONACYT. Actualmente Profr. Asociado de la UAM Iztapalapa. Las áreas de interés son: Sistemas Distribuidos, Ingeniería de Software y Métodos Formales.