Introducción al modelo CSP de cálculo paralelo

Ricardo Peña

Report LSI–90–4

Resum: es presenta el model algebraic CSP (Communicating Sequential Processes) els autors del qual son C.A.R. Hoare i el seu grup. Es mostra com es poden moldejar sistemes paral.lels i expresar formalment propietats. Es resumeixen algunes de les tecniques de verificaci'o i de transformaci'o de procesos desenvolupats per aquest model.

Abstract: The algebraic model CSP (Communicating Sequential Processes) due to C.A.R. Hoare and his group is presented. It is shown how parallel systems can be modeled and how formal properties can be expressed and proved in the model. Some verification and process transformation rules are also summarized.

Introducción al modelo CSP de cálculo paralelo

Ricardo Peña

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Barcelona

Resumen: Se presenta el modelo algebraico CSP (Communicating Sequential Processes) debido a C.A.R. Hoare y sus colaboradores. Se ilustra cómo se pueden modelar sistemas paralelos en el mismo y expresar formalmente propiedades. Se resumen algunas de las técnicas de verificación y transformación de procesos desarrolladas para dicho modelo.

INDICE:

1. Introducción

En este trabajo se presenta el modelo CSP (Communicating Sequential Processes) [BHR 84, Hoa 85]. Pertenece a la categoría de modelos abstractos de cálculo paralelo. En ellos, el énfasis se pone en modelar programas paralelos con un reducido número de operadores elementales cuya semántica está formalmente definida y, una vez conseguido, o bien ser capaces de demostrar propiedades sobre los mismos, o bien transformarlos en programas sintácticamente distintos pero semánticamente equivalentes.

Un lenguaje de programación convencional tiene excesivos detalles que dificultan el razonamiento. Aunque es posible dar una semántica formal para los mismos (por ejemplo, existe una semántica denotacional para Occam dada en términos del modelo CSP [Ro 85, RoHo 88]), dichas semánticas, al igual que sucede en programación secuencial, son de poca ayuda a la hora de expresar y verificar propiedades de un programa paralelo.

Por ello, se han propuesto notaciones y modelos de mayor nivel de abstracción con la esperanza de reducir el esfuerzo de verificación. Se parte del supuesto de que, una vez comprobadas las propiedades deseadas en "programas" escritos en este tipo de lenguajes, siempre será posible, con mayor o menor esfuerzo, *transformarlos* a una notación que pueda ser ejecutada en una máquina real.

Precursor de CSP es el modelo CCS (A Calculus of Communicating Systems) de Milner [Mil 80], el primero de esta categoría de modelos. Allí no se da una semántica denotacional explícita para los procesos, como ocurre en CSP, y el problema central consiste en decidir cuándo dos procesos sintácticamente distintos se comportan igual. La congruencia resultante resulta ser más fina que la de CSP, es decir, procesos indistinguibles en CSP, pueden ser distinguidos en CCS. En correspondencia con ello, CCS satisface un menor número de leyes algebraicas de transformación de procesos que CSP.

En ambos modelos, el posible paralelismo de eventos se modela mediante entrelazado. Dos eventos *a* y *b* han de considerarse paralelos si, siempre que sea posible una traza...*ab*... del sistema , es posible la correspondiente traza ...*ba*... De este modo, un proceso convencional (en el sentido de programa secuencial) y una red de procesos en paralelo, son indistinguibles en el modelo. Ambos son la misma entidad matemática *proceso*.

Otra característica común es dar por supuesta la existencia de un *entorno* externo, que puede decidir cual es el siguiente evento a ejecutar por el sistema entre todos los que el mismo está dispuesto a realizar. Así, el sistema *ofrece* eventos y el entorno *decide* cual de ellos se ejecutará.

Sin embargo, el sistema también puede tomar decisiones autónomas y, tras ejecutar una misma traza, no siempre ofrecerá el mismo conjunto de eventos. El *menú* ofrecido se selecciona de modo no determinista entre varios menús posibles en ese estado. En CSP existe un operador explícito de composición no determinista de procesos. En CCS se puede lograr el mismo efecto combinando el único operador de selección con la llamada *transición silenciosa*.

El no determinismo se introduce ante la necesidad de ignorar detalles de los procesos, irrelevantes en un cierto nivel de abstracción. Por ejemplo, si el proceso está implementado como una red de subprocesos que se sincronizan entre si, las distintas velocidades de los subprocesos pueden dar lugar a comportamientos externos de la red distintos.

2. Sintaxis de CSP

Cada proceso P se supone dotado de un alfabeto αP, finito o infinito, que comprende todos los eventos en los que el proceso *puede* participar. Emplearemos los símbolos a, b, c,... para denotar constantes de tipo evento, x, y, z,... para variables de tipo evento, A, B, C,... para conjuntos de eventos y alfabetos, P, Q, R,... para procesos concretos y X, Y, Z,... para variables de tipo proceso.

La sintaxis de un proceso CSP viene dada por la siguiente gramática:

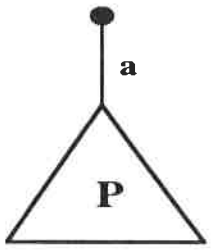
$$P ::= \textbf{stop} \mid \textbf{div} \mid a \rightarrow P \mid P \sqcap Q \mid P \,[]\, Q \mid P \parallel Q \mid P \,|||\, Q \mid P\backslash a \mid P[b/a] \mid \mu X.P(X)$$

donde P(X) es un término CSP sintácticamente correcto que contiene la variable X de tipo proceso. Se dice que la variable X está ligada al cuantificador μ. P y Q son procesos que no tienen variables libres. Una variable es libre cuando no está ligada a un cuantificador μ. Salvo que se indique lo contrario, todos los procesos tienen el mismo alfabeto A.

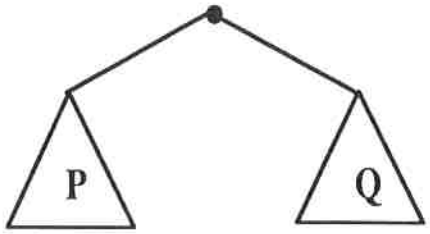
El proceso **stop**, o $\textbf{stop}_A$, representa el proceso bloqueado, es decir un proceso que no está dispuesto, permanentemente, a participar en evento alguno de A. Tanto este proceso como el que se describe a continuación, representan situaciones indeseadas. Se incorporan al modelo para poder demostrar que un sistema *no* es equivalente a ninguno de ellos.

El proceso **div** es el proceso divergente. Intuitivamente, un proceso diverge cuando realiza una secuencia infinita de eventos internos y ello le impide comunicarse con el entorno. Aunque el comportamiento visible no parezca muy diferente al del proceso bloqueado (la diferencia práctica sería que éste no consume recursos de máquina, y aquél si), el modelo ha de distinguir entre ambos ya que se presentan en contextos muy diferentes.

La expresión $a \rightarrow P$ denota un proceso que inicialmente sólo está dispuesto a participar en el evento *a* y a continuación se comporta como el proceso P. El operador $\rightarrow$ se denomina *prefijo*. Emplearemos frecuentemente la siguiente notación gráfica:

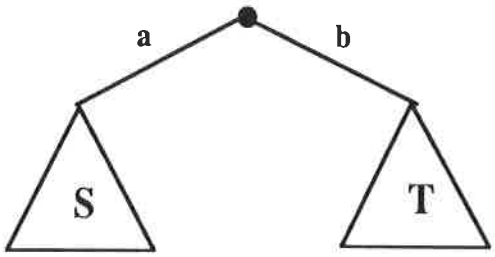


El operador binario ⌈⌉ se denomina *selección no determinista*. P ⌈⌉ Q representa un proceso que inicialmente realiza una selección interna, es decir, no controlable por el entorno, según la cual a continuación se comporta como P o como Q.



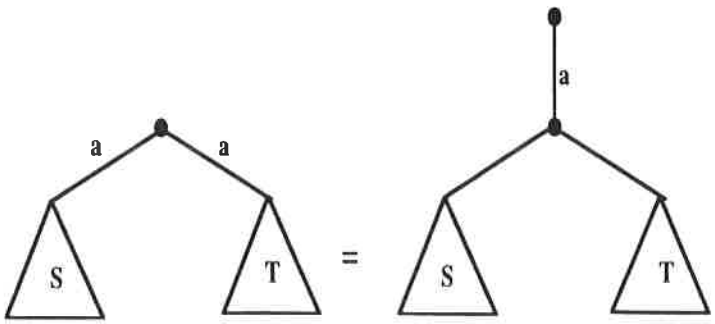
El operador binario [] se denomina *selección determinista*. P [] Q representa un proceso que inicialmente ofrece la unión de los menús posibles para P y para Q. El entorno puede seleccionar si el comportamiento futuro va a ser el de P o el de Q, ejecutando un evento que sea posible para el uno pero no para el otro.

Ejemplo: R = (a → S) [] (b → T)



Si todos los eventos inicialmente posibles para P, también lo son para Q y viceversa, la elección del comportamiento futuro es no determinista. Es decir, el operador [] es, en ciertos casos, expresable en términos del operador ⌈⌉.

Ejemplo: (a → S) [] (a → T) = a → (S ⌈⌉ T)



El operador || realiza la *composición paralela* de dos procesos. El proceso P || Q se comporta del

siguiente modo:

1) Si P puede ejecutar un evento *a* no perteneciente al alfabeto de Q, y luego comportarse como P', una posibillidad para P || Q es a → (P' || Q).

2) Si Q puede ejecutar un evento *b* no perteneciente al alfabeto de P y luego comportarse como Q', otra posibilidad para P || Q es b → (P || Q').

3) Si P y Q están ambos dispuestos a ejecutar un evento común *c* y, a continuación y respectivamente, comportarse como P'' y Q'', otra posibilidad es c → (P'' || Q'').

P || Q es la composición mediante selección determinista de las posibilidades (1) a (3):

P || Q = (a → (P' || Q)) [] (b → (P || Q')) [] (c → (P'' || Q''))

El operador |||, denominado de *entrelazado*, realiza otra forma de composición paralela en la cual no hay sincronización entre los procesos. Mientras que en P || Q se necesita la "cooperación" de ambos procesos para ejecutar un evento común, en P ||| Q ambos pueden ejecutar eventos comunes o no comunes independientemente. Las trazas de P ||| Q se consiguen entrelazando arbitrariamente cualquier traza de P con cualquier traza de Q.

Ejemplo:

P = a → b → **stop**
Q = a → c → **stop**
P ||| Q = (a → b → a → c → **stop**) [] (a → a → b → c → **stop**) []
(a → a → c → b → **stop**) [] (a → c → a → b → **stop**)

El operador \, denominado de *ocultamiento*, sirve para ignorar parte de la actividad de un proceso. P \ b denota el proceso P en el que no son visibles todas sus participaciones en el evento b. Es un operador muy importante porque permite *abstraer* detalles internos de un sistema, irrelevantes a un cierto nivel. Su tratamiento matemático es, sin embargo, bastante complejo.

Ejemplo:

P = (a → **stop**) [] (c → b → **stop**)
P \ b = (a → **stop**) [] (c → **stop**)

El operador [/], llamado de *renombramiento*, realiza el cambio de nombre los eventos de un proceso. P [b/a] denota al proceso P, en el que todas las apariciones del evento *a* se sustituyen

por el evento *b*. Se utiliza para crear ejemplares distintos a partir de un proceso genérico, situación frecuente en redes en que aparecen conjuntos de procesos con similar estructura.

Ejemplo:

$P = (a \rightarrow \textbf{stop}) [] (a \rightarrow (c \rightarrow b \rightarrow \textbf{stop} [] c \rightarrow \textbf{stop}))$
$P [b/a] = (b \rightarrow \textbf{stop}) [] (b \rightarrow (c \rightarrow b \rightarrow \textbf{stop} [] c \rightarrow \textbf{stop}))$

Finalmente, la notación $\mu X.P(X)$, donde X es una variable y P(X) es un término CSP sintácticamente correcto que puede contener la variable X y no contiene variables libres, designa al proceso solución de la ecuación recursiva $X = P(X)$. La expresión $\mu X.P(X)$ se lee: *menor punto fijo X de P(X).*

Ejemplo 1:

Sea $P = \mu X.(a \rightarrow X)$, es decir, P es la solución de $X = a \rightarrow X$.

En el modelo CSP, la solución a esta ecuación es el proceso infinito

$P = a \rightarrow a \rightarrow a \rightarrow \ldots$

siempre dispuesto a participar en el evento *a*.

Se denominan procesos *infinitos* a aquellos que no terminan. Excluyendo a **div**, todos los procesos que se pueden construir con los restantes operadores, son procesos *finitos* cuya actividad termina necesariamente en el proceso **stop**. La construcción recursiva $\mu X.P(X)$ permite definir procesos más realistas que se ejecutan ininterrumpidamente. No tienen por que ser necesariamente cíclicos, es decir, con un número finito de estados internos (ver ejemplo 3). La notación puede generalizarse a un conjunto de ecuaciones mútuamente recursivas, sustituyendo la variable simple X por una variable vectorial $\mathbf{X} = < X_i >_{i \in I}$, donde I es un conjunto cualquiera, finito o infinito, de índices y $\mathbf{P(X)}$ es un vector $< P_i(\mathbf{X}) >_{i \in I}$ de expresiones CSP en las que aparecen las variables ligadas $< X_i >_{i \in I}$.

Ejemplo 2: *Contador de valor máximo 2*

$P = X_0$
$X_0 = \text{inc} \rightarrow X_1$
$X_1 = (\text{dec} \rightarrow X_0) [] (\text{inc} \rightarrow X_2)$
$X_2 = \text{dec} \rightarrow X_1$

En este caso, $\mathbf{X} = < X_0, X_1, X_2 >$ y $\mathbf{X} = \mu\mathbf{X}.\mathbf{P}(\mathbf{X})$ es el sistema de ecuaciones mútuamente recursivas escrito más arriba. La solución a este sistema en el modelo CSP es un único vector $< X_0, X_1, X_2 >$. El proceso P es la componente X_0 de este vector. Representa el contador inicializado a cero. Las componentes X_1 y X_2 representan respectivamente contadores inicializados a uno y a dos.

Ejemplo 3: *Contador no acotado*

$$P = X_0$$
$$X_0 = inc \rightarrow X_1$$
$$X_i = (dec \rightarrow X_{i-1}) [] (inc \rightarrow X_{i+1}) \quad , \text{para } i \geq 1$$

En este caso $\mathbf{X} = < X_i >_{i \in N}$ es una variable vectorial de infinitos componentes y $\mathbf{X} = \mathbf{P}(\mathbf{X})$ es un sistema de infinitas ecuaciones mútuamente recursivas. El proceso P representa un contador inicializado a cero que puede pasar por un número infinito de estados, uno para cada número natural i.

La posibilidad de definir y operar con procesos infinitos da lugar a procesos de significado poco claro:

$$P = (\mu X.(a \rightarrow X)) \setminus a$$
$$Q = \mu X.X$$
$$R = (\mu X.((a \rightarrow X) [] (b \rightarrow X))) \setminus b$$

Todos ellos representan procesos que realizan una secuencia infinita de eventos ocultos. La solución de estas tres ecuaciones es, como se verá, el proceso divergente **div** explicado más arriba.

3. Modelización de sistemas paralelos

Con los operadores básicos CSP y con otros definidos en términos suyos, se puede modelizar cualquier situación que se presente en programación paralela. Se expone a continuación una serie de ejemplos de complejidad creciente modelados en CSP.

Ejemplo1: Sistema de usuarios y semáforo para exclusión mútua

$$usuario = \mu X.(o \rightarrow p \rightarrow cr \rightarrow fr \rightarrow v \rightarrow X)$$
$$semaforo = \mu X.(p \rightarrow v \rightarrow X)$$

$$usuario_i = usuario\ [o_i/o][cr_i/cr][fr_i/fr]\ \ , i \in \{1,\ldots,n\}$$

Los eventos o_i, cr_i, y fr_i simbolizan, respectivamente, las acciones privadas "otras cosas", "comienzo de región crítica" y "fin de región crítica". Los eventos p y v simbolizan la petición del permiso de acceso y la notificación de abandono.

$$sistema = (\ |||_{i=1}^{n} usuario_i \ ) \ || \ semaforo$$

Nótese que, aunque los eventos {p, v} están en el alfabeto de todos los procesos usuarios, éstos no se sincronizan entre sí debido a su composición mediante el operador de entrelazado. Si se desea un semáforo con canales separados para cada usuario, se modificarán las definiciones en el siguiente sentido:

$$usuario_i = usuario\ [o_i/o][p_i/p][cr_i/cr][fr_i/fr][v_i/v]\ \ , i \in \{1,\ldots,n\}$$

$$semaforo = \mu X.(\ []_{i=1}^{n} p_i \rightarrow v_i \rightarrow X \ )$$

$$sistema = (\ ||_{i=1}^{n} usuario_i \ ) \ || \ semaforo$$

Ejemplo 2: Canales

Un canal entre dos procesos, en el sentido de la notación definida en el capítulo 2, se modeliza introduciendo el concepto de *evento compuesto c.v*, donde *c* es el nombre del canal y *v* el valor transmitido a su través.

$c\ !\ v \rightarrow P$	es equivalente a	$c.v \rightarrow P$
$c\ ?\ x \rightarrow Q(x)$	es equivalente a	$[]_v\ (c.v \rightarrow Q(v))$

por tanto, $(c\ !\ v \rightarrow P) \ || \ (c\ ?\ x \rightarrow Q(x)) = c.v \rightarrow (P \ || \ Q(v))$

es decir, el efecto neto es que el valor *v* ha sido transmitido desde el primer proceso al segundo, y el sistema continúa comportándose como la composición paralela de P y de Q(x) donde la variable x se ha sustituido por el valor *v*.

Los procesos Q(x) que dependen de un parámetro de tipo valor, se pueden programar con instrucciones condicionales que realizan consultas booleanas sobre el valor de x.

Ejemplo 3: Proceso que copia caracteres sustituyendo apariciones de '$**$' por '↑'


```
FILTRO = μY.(   izq ? x →
                (si x ≠'*' ent ( der ! x → Y)
                sino    (izq ? z →  (si z='*' ent ( der ! '↑' → Y)
                                    sino ( der ! x → der ! z → Y)
                                    )
                        )
                ))
```

Ejemplo 4: Proceso almacén no acotado de mensajes, con estrategia FIFO

$ALMACEN = Y_{<>}$

$Y_{<>} = izq\ ?\ x \rightarrow Y_{<x>}$,para todo x

$Y_{<x>^s} = (izq\ ?\ z \rightarrow Y_{<x>^s{}^{<z>}})\ []\ (der\ !\ x \rightarrow Y_s)$,para todo x, z y s

En este caso, el conjunto de índices lo forman todas las secuencias posibles de mensajes. Y_s representa un proceso que tiene almacenada la secuencia s, <> representa la secuencia vacía, <x> la secuencia unitaria del mensaje x, y ^ la operación de concatenar.

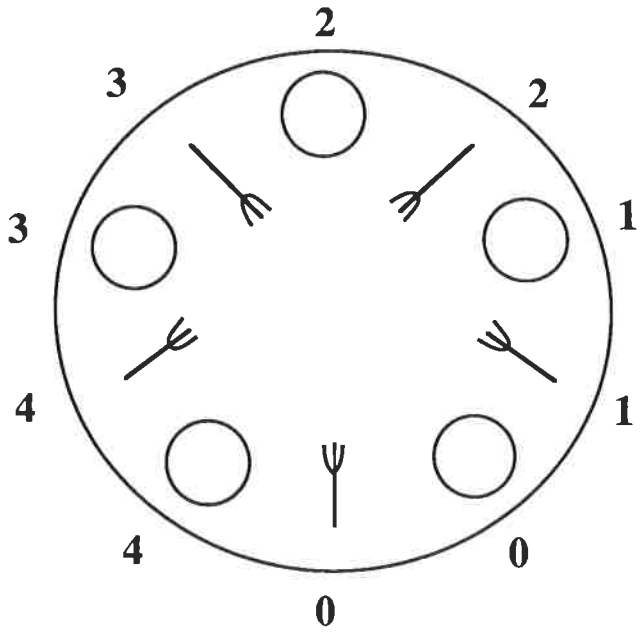
Nótese que el proceso ALMACEN no sólo es infinito (no termina) sino que además tiene un número infinito de estados internos.

Ejemplo 5: Proceso que simula una variable

$VAR = izq\ ?\ x \rightarrow VAR_x$
$VAR_x = (\ izq\ ?\ y \rightarrow VAR_y\ )\ []\ (\ der\ !\ x \rightarrow VAR_x\ )$

Una vez inicializada, la variable puede ser consultada por el canal *der* sin cambiar su valor, o actualizada por el canal *izq*. VAR_x es un proceso que representa la variable con valor en curso x.

Ejemplo 6: Filósofos que comen espaguetis



Se presenta la solución con cinco gestores tipo semáforo, uno por cada tenedor, para garantizar exclusión mútua en su uso, más un lacayo centralizado que asegura un número máximo de cuatro filósofos simultáneamente a la mesa [Hoa 85].

$FIL = \mu X.(s \rightarrow cizq \rightarrow cder \rightarrow cc \rightarrow fc \rightarrow dizq \rightarrow dder \rightarrow l \rightarrow X)$

El significado de estos eventos es, en el mismo orden: "sentarse", "coger tenedor izquierdo", "coger tenedor derecho", "comienzo de comer", "fin de comer", "dejar tenedor izquierdo", "dejar tenedor derecho" y "levantarse". Un gestor-tenedor tendrá la forma:

$TEN = \mu X.((cizq \rightarrow dizq \rightarrow X)\ []\ (\ cder \rightarrow dder \rightarrow X))$

es decir, o bien se deja capturar y liberar como tenedor izquierdo del filósofo que tiene su mismo número, o bien se deja capturar y liberar como tenedor derecho del filósofo con un número menos. Suponiendo que '-' representa la resta módulo 4, los cinco filósofos y tenedores se obtienen mediante el siguiente renombramiento:

FIL_i =
FIL [i.s/s][i.cizq/cizq][i.cder/cder][i.cc/cc][i.fc/fc][i.dizq/dizq][i.dder/dder][i.l/l]
TEN_i = TEN [i-1.cder/c.der][i-1.dder/dder][i.cizq/c.izq][i.dizq/dizq]

Un primer sistema con riesgo de bloqueo se obtiene componiendo los diez procesos en paralelo:

$$MESA = \left(\|_{i=0}^{4} FIL_i \right) \| \left(\|_{i=0}^{4} TEN_i \right)$$

El proceso lacayo se define mediante el sistema recursivo:

$$LAC = LAC_0 = \ \Box_{i=0}^{4} \ (i.s \rightarrow LAC_1)$$

$$LAC_j = \left(\Box_{i=0}^{4} \ (i.s \rightarrow LAC_{j+1}) \right) \Box \left(\Box_{i=0}^{4} \ (i.l \rightarrow LAC_{j-1}) \right) \qquad \text{para } 1 \leq j \leq 3$$

$$LAC_4 = \ \Box_{i=0}^{4} \ (i.l \rightarrow LAC_3)$$

Obviamente, LAC_j representa el proceso lacayo en un estado en que la diferencia entre permisos de sentarse y de levantarse es de j, $0 \leq j \leq 4$. El sistema que garantiza que no hay más de cuatro filósofos sentados es:

OTRA_MESA = MESA || LAC

que está libre de bloqueo.

4. Modelo denotacional de CSP

La pregunta fundamental a contestar una vez definida la sintaxis es:

¿Cuándo dos procesos CSP son iguales ?.

Para ello, CSP prefiere contestar previamente otra:

¿Qué es un proceso ?

Se construye un modelo explícito en el que, a toda construcción sintácticamente correcta, se le asigna una única entidad matemática en el modelo. Dos procesos son iguales si y solo si la entidad matemática que les corresponde en el modelo es la misma.

Una primera propuesta de modelo es identificar un proceso con el conjunto de sus *trazas* posibles. Se denomina *modelo de trazas.*

Una **traza** *s* es una secuencia finita de eventos. Si A es el alfabeto del proceso, $s \in A^*$. Emplearemos la siguiente notación:

<>	traza vacía
<a b c>	secuencia de eventos a, b, c ∈ A
#s	número de eventos de la traza s
s^t, a^s, s^a	concatenación de trazas o de evento y traza
s ↑ B	traza s restringida a los eventos de B
s \ b	traza s, eliminando las apariciones del evento b
s ≤ t	la traza s es un prefijo de la traza t
s [b/a]	traza s, cambiando las apariciones de a por b
b^n	traza formada por n b's consecutivas
s || t	conjunto de trazas formado entrelazando la traza s con la traza t, respetando la sincronización de eventos comunes. Ejemplo: <a b c> || <c b d> = {<a c b c d>, <a c b d c>, <c a b c d>, <c a b d c> }
s ||| t	conjunto de trazas formado entrelazando líbremente la traza s con la traza t.

El modelo de trazas **T** asigna a cada término CSP sintácticamente correcto, un conjunto de trazas:

$$\mathbf{T}: \text{Términos\_CSP} \rightarrow \mathbf{P}(A^*)$$

donde $\mathbf{P}(A^*)$ denota las partes finitas de A^*.

Dicho conjunto de trazas se construye inductívamente a partir de la estructura sintáctica del término CSP:

$\mathbf{T}$ [**stop**] = {<>}

$\mathbf{T}$ [**div**] = {<>}

$\mathbf{T}$ [a → P] = {<>} ∪ {a^s | s ∈ $\mathbf{T}$ [P] }

$\mathbf{T}$ [P ⌈] Q] = { $\mathbf{T}$ [P] } ∪ { $\mathbf{T}$ [Q] }

$\mathbf{T}$ [P [] Q] = { $\mathbf{T}$ [P] } ∪ { $\mathbf{T}$ [Q] }

$\mathbf{T}$ [P || Q] = { s | ∃ t_1 ∈ $\mathbf{T}$ [P], ∃ t_2 ∈ $\mathbf{T}$ [Q]: s ∈ t_1 || t_2 }

$\mathbf{T}$ [P ||| Q] = { s | ∃ t_1 ∈ $\mathbf{T}$ [P], ∃ t_2 ∈ $\mathbf{T}$ [Q]: s ∈ t_1 ||| t_2 }

$\mathbf{T}$ [P [b/a]] = { s [b/a] | s ∈ $\mathbf{T}$ [P] }

$\mathbf{T}$ [P \b] = { s\b | s ∈ $\mathbf{T}$ [P] }

$\mathbf{T}$ [μX.P(X)] = $\cup_{n\geq 0}$ $\mathbf{T}$ [P^n (**stop**)] donde $P^0(X) = X$ y $P^{i+1}(X) = P(P^i(X))$

,es decir, el menor punto fijo de la función Φ_P: $A^* \rightarrow A^*$
inducida por el término P(X)

El modelo de trazas realiza *demasiada* identificación entre los procesos: dos procesos son iguales si el conjunto de sus historias posibles coinciden. Una historia o traza de un proceso nos informa sobre su *pasado* pero no sobre los eventos futuros que está dispuesto a realizar. En particular, el modelo $\mathbf{T}$ no distingue entre trazas posibles y obligatorias. El hecho de que *s* y *s^a* estén ambas en $\mathbf{T}$ [P] no informa sobre si *a* se realizará obligatoriamente despues de *s*, si el entorno insiste.

Los siguientes procesos:

P1 = (a → **stop**) [] (b → **stop**)
P2 = (a → **stop**) ⌈] (b → **stop**)
P3 = (a → **stop**) [] (b → **stop**) [] **div**
P4 = ((a → **stop**) [] (c → b → **stop**)) \ c
P5 = ((a → **stop**) [] (b → **stop**)) ⌈] (b → **stop**)

son indistinguibles en el modelo de trazas.

$\mathbf{T}$ [P1] = ... = $\mathbf{T}$ [P5] = {<>, <a>, <b> }

El modelo denotacional CSP se denomina *modelo de fallos* (failures model, [BHR 84], [BrRo 85]). En él, un proceso es una entidad compleja que incluye, además del conjunto de sus trazas posibles, información sobre los *menús* ofrecidos por el proceso después de cada traza e información sobre en qué condiciones el proceso diverge.

Se denomina *conjunto rechazable* de un proceso P de alfabeto A, a un subconjunto $X \subseteq A$ tal que, si el entorno ofrece X, P *puede* bloquearse (aunque no necesariamente lo haga).

Ejemplo: Sean $A = \{a, b, c\}$ y $P = (a \rightarrow \textbf{stop}) \sqcap (b \rightarrow \textbf{stop})$

Todos los conjuntos rechazables de P son:

rechazos (P) = $\{\varnothing, \{a\}, \{b\}, \{c\}, \{a, c\}, \{b, c\}\}$

En cambio, no lo son $\{a, b\}$ y $\{a, b, c\}$

Algunas leyes interesantes sobre conjuntos rechazables:

- *rechazos* (**stop**) = **P** (A)
- *rechazos* (**div**) = **P** (A)
- $\forall$ P, $\varnothing \in$ *rechazos*(P)
- $X \in$ *rechazos* (P) $\Rightarrow \forall\ Y \subseteq X$: $Y \in$ *rechazos* (P)
- *rechazos* (P $\sqcap$ Q) = *rechazos* (P) $\cup$ *rechazos* (Q)
- *rechazos* (P [] Q) = *rechazos* (P) $\cap$ *rechazos* (Q)

Como puede observarse, los conjuntos rechazables permiten distinguir entre la composición determinista y no determinista de procesos. En cambio, no hacen distinción entre un proceso bloqueado y uno divergente.

Se denomina *fallo* de un proceso P, a un par (s, X) donde s es una traza posible de P y X es un conjunto rechazable de P/s (P después de ejecutar s). Es decir, $X \in$ *rechazos* (P/s).

Se denomina *divergencia* o traza divergente de un proceso P a aquellas trazas de P tras ejecutar las cuales el proceso puede diverger, es decir, puede realizar una secuencia infinita de acciones internas.

Ejemplo:

$P1 = \mu X.X$
$P2 = (\mu X.((a \rightarrow X)\ []\ (b \rightarrow X))) \setminus b$

$\langle\rangle \in$ *divergencias* (P1) , $\langle\rangle \in$ *divergencias* (P2) , $\langle a\rangle \in$ *divergencias* (P2)

El fenómeno de divergencia aparece exclusivamente en procesos infinitos, bien como consecuencia de definiciones recursivas *no guardadas*, o debido al ocultamiento de secuencias infinitas de eventos.

El modelo de fallos **F** identifica un proceso P con un par formado por el conjunto de sus fallos y el conjunto de sus divergencias:

$\mathbf{F}\,[P] = (F, D)$
$F = \{ (s, X) \mid s \in \mathbf{T}\,[P] , X \in \textit{rechazos}\ (P/s) \}$
$D = \textit{divergencias}\ (P) \subseteq \mathbf{T}\,[P]$

Las trazas del proceso se pueden obtener a partir de **F** [P] ya que, si s es una traza de P, el fallo $(s, \varnothing)$ siempre estará en **F** [P].

Se requieren algunas condiciones técnicas para que (F, D) sea un proceso:

1) **T** [P] es cerrado por prefijo, es decir, $t \in \mathbf{T}\,[P]$ y $s \leq t \Rightarrow s \in \mathbf{T}\,[P]$
2) $(s, X) \in F \Rightarrow \forall\, Y \subseteq X: (s, Y) \in F$
3) $(s, X) \in F \wedge a \in A \Rightarrow (s, X \cup \{a\}) \in F \vee (s^\frown a, \varnothing) \in F$
4) $D \subseteq \mathbf{T}\,[P]$
5) D es cerrado por extensión: $s \in D \Rightarrow \forall\, t \in A^*: s^\frown t \in D \wedge \forall\, X \subseteq A: (s, X) \in F$

Al igual que se hizo con el modelo de trazas, la semántica de fallos de un proceso se dará inductivamente a partir de la estructura sintáctica del término CSP. Para simplificar, (F, D) se dará como un solo conjunto en el que los fallos se indicarán como sX, las divergencias como $s\uparrow$, y $s\Delta$ denota tanto sX como $s\uparrow$. Como siempre, A denota el alfabeto del proceso.

$\mathbf{F}\ [\mathbf{stop}] = \{<>X \mid X \subseteq A\}$

$\mathbf{F}\ [\mathbf{div}] = \{sX, s\uparrow \mid s \in A^*, X \subseteq A\}$

$\mathbf{F}\ [a \rightarrow P] = \{<>X \mid a \notin X\} \cup \{a^\frown s\Delta \mid s\Delta \in \mathbf{F}\ [P]\ \}$

$\mathbf{F}\ [P \sqcap Q] = \{\ \mathbf{F}\ [P]\ \} \cup \{\ \mathbf{F}\ [Q]\ \}$

$\mathbf{F}\ [P\ []\ Q] = \{\ <>X \mid <>X \in \mathbf{F}\ [P] \cap \mathbf{F}\ [Q]\ \} \cup \{\ <>\Delta \mid <>\uparrow \in \mathbf{F}\ [P] \cup \mathbf{F}\ [Q]\ \} \cup$
$\{\ s\Delta \mid s \neq <> \wedge s\Delta \in \mathbf{F}\ [P] \cup \mathbf{F}\ [Q]\ \}$

$$\mathbf{F}\,[P \,\|\, Q] = \{\, s(X_1 \cup X_2) \mid \exists\, t_1 X_1 \in \mathbf{F}\,[P], \exists\, t_2 X_2 \in \mathbf{F}\,[Q]: s \in t_1 \,\|\, t_2 \,\} \cup$$
$$\{\, s^\frown t\Delta \mid t \in A^*, \exists\, t_1 \varnothing \in \mathbf{F}\,[P], \exists\, t_2 \varnothing \in \mathbf{F}\,[Q]: s \in t_1 \,\|\, t_2 \wedge$$
$$(t_1\!\uparrow \in \mathbf{F}\,[P] \vee t_2\!\uparrow \in \mathbf{F}\,[Q]) \,\}$$

$$\mathbf{F}\,[P \,|||\, Q] = \{\, sX \mid \exists\, t_1 X \in \mathbf{F}\,[P], \exists\, t_2 X \in \mathbf{F}\,[Q]: s \in t_1 \,|||\, t_2 \,\} \cup$$
$$\{\, s^\frown t\Delta \mid t \in A^*, \exists\, t_1 \varnothing \in \mathbf{F}\,[P], \exists\, t_2 \varnothing \in \mathbf{F}\,[Q]: s \in t_1 \,|||\, t_2 \wedge$$
$$(t_1\!\uparrow \in \mathbf{F}\,[P] \vee t_2\!\uparrow \in \mathbf{F}\,[Q]) \,\}$$

$$\mathbf{F}\,[P\,[b/a]\,] = \{\, s\,[b/a]\Delta \mid s\Delta \in \mathbf{F}\,[P] \,\}$$

$$\mathbf{F}\,[P \setminus b] = \{\, s\setminus bX \mid s(X \cup \{b\}) \in \mathbf{F}\,[P] \,\} \cup \{\, s\setminus b^\frown t\Delta \mid t \in A^* \wedge (\forall\, n \geq 0: s^\frown b^n \varnothing \in \mathbf{F}[P]) \,\}$$

$$\mathbf{F}\,[\mu X.P(X)] = (\cap_{n \geq 0} F_n \,,\, \cap_{n \geq 0} D_n)$$

donde $(F_n, D_n) = \mathbf{F}\,[P^n\,(\mathbf{div})]$,

$P^0(X) = X$ y $P^{i+1}(X) = P(P^i(X))$,

es decir, el menor punto fijo de la función Φ_P: Dom $\rightarrow$ Dom

inducida por el término P(X)

Los cinco procesos anteriores:

P1 = (a $\rightarrow$ **stop**) [] (b $\rightarrow$ **stop**)
P2 = (a $\rightarrow$ **stop**) ⌈⌉ (b $\rightarrow$ **stop**)
P3 = (a $\rightarrow$ **stop**) [] (b $\rightarrow$ **stop**) [] **div**
P4 = ((a $\rightarrow$ **stop**) [] (c $\rightarrow$ b $\rightarrow$ **stop**)) \ c
P5 = ((a $\rightarrow$ **stop**) [] (b $\rightarrow$ **stop**)) ⌈⌉ (b $\rightarrow$ **stop**)

indistinguibles en el modelo de trazas, pueden ser ahora distinguidos en el modelo de fallos. P1, P2 y P4 son los tres diferentes: si el entorno ofrece {a} P2 y P4 pueden bloquearse pero no así P1; si el entorno ofrece {b}, P2 puede bloquearse pero no así P1 y P4. P3 es equivalente a **div**, es decir, es un proceso divergente. P5 es el mismo proceso que P4 (ver ley algebraica L19).

5. Leyes algebraicas CSP

El modelo de fallos es demasiado complejo para ser utilizado directamente en demostraciones de propiedades de sistemas reales.

Sin embargo, a partir de él se demuestran un amplio conjunto de leyes algebraicas que pueden emplearse para simplificar y transformar procesos. La identificación entre procesos en el modelo de fallos es menor, como se ha dicho, que la inducida por el modelo de trazas. Permite distinguir

procesos deterministas de no deterministas y procesos divergentes de los que no lo son.

Las leyes permiten reducir todo proceso finito CSP a otro equivalente que sólo contiene los operadores primitivos **stop**, →, [] y ⌈⌉ . Es decir, el resto de los operadores (||, |||, \b y [b/a]) son operadores derivados que pueden expresarse en función de los anteriores. Nótese que, aparte de **stop**, sólo || es capaz de provocar bloqueos.

Los procesos infinitos, al ser el límite de una sucesión de procesos finitos, también pueden expresarse en función exclusivamente de los operadores primitivos, a los que hay que añadir **div**. Llamaremos *forma normal* de un proceso a un proceso equivalente que sólo contiene operadores primitivos. Si ésta contiene a **div**, el proceso será divergente.

Leyes de la selección no determinista

L1) $P \sqcap P = P$

L2) $P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R$

L3) $P \sqcap Q = Q \sqcap P$

Leyes de la selección determinista

L4) $P \,[]\, P = P$

L5) $P \,[]\, (Q \,[]\, R) = (P \,[]\, Q) \,[]\, R$

L6) $P \,[]\, Q = Q \,[]\, P$

L7) $P \,[]\, \mathbf{stop} = P$

Leyes distributivas

L8) $(a \rightarrow P) \sqcap (a \rightarrow Q) = a \rightarrow (P \sqcap Q)$

L9) $(a \rightarrow P) \,[]\, (a \rightarrow Q) = a \rightarrow (P \sqcap Q)$

L10) $P \,[]\, (Q \sqcap R) = (P \,[]\, Q) \sqcap (P \,[]\, R)$

L11) $P \sqcap (Q \,[]\, R) = (P \sqcap Q) \,[]\, (P \sqcap R)$

Leyes de la divergencia

L12) $P \,[]\, \mathbf{div} = \mathbf{div}$

L13) $P \sqcap \mathbf{div} = \mathbf{div}$

L14) $\mathbf{div} \setminus b = \mathbf{div}$

L15) $P \parallel \mathbf{div} = \mathbf{div}$

L16) $P \mid\mid\mid \mathbf{div} = \mathbf{div}$

Leyes del ocultamiento

L17) $\mathbf{stop} \setminus b = \mathbf{stop}$

L18) $(P \sqcap Q) \setminus b = (P \setminus b) \sqcap (Q \setminus b)$

L19) $((a \rightarrow P) \,[]\, Q) \setminus a = (P \setminus a) \sqcap ((P \,[]\, Q) \setminus a)$

L20) $([]_{y \in X} (y \rightarrow P_y)) \setminus a = []_{y \in X} (y \rightarrow (P_y \setminus a))$, si $a \notin X$

Leyes de la composición paralela

Sean

$$P = []_{b \in B} (b \rightarrow P_b) \text{ y } Q = []_{c \in C} (c \rightarrow Q_c)$$

En el caso en que B o C sean el conjunto vacío, estos procesos denotan **stop**.

L21) $P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$

L22) $P \parallel Q = Q \parallel P$

L23) $P \mid\mid\mid (Q \mid\mid\mid R) = (P \mid\mid\mid Q) \mid\mid\mid R$

L24) $P \mid\mid\mid Q = Q \mid\mid\mid P$

L25) $P \parallel Q = ([]_{b \in B - C} b \rightarrow (P_b \parallel Q)) \,[]$
$([]_{c \in C - B} c \rightarrow (P \parallel Q_c)) \,[]$
$([]_{d \in B \cap C} d \rightarrow (P_d \parallel Q_d))$

L26) $P \mid\mid\mid Q = ([]_{b \in B} b \rightarrow (P_b \mid\mid\mid Q)) \,[]$
$([]_{c \in C} c \rightarrow (P \mid\mid\mid Q_c))$

L27) $P \parallel (Q \sqcap R) = (P \parallel Q) \sqcap (P \parallel R)$

L28) $P \mid\mid\mid (Q \sqcap R) = (P \mid\mid\mid Q) \sqcap (P \mid\mid\mid R)$

6. Propiedades de los programas paralelos

Una vez construido un programa paralelo, resulta bastante natural preguntarse si satisface una serie de propiedades deseables. Por ejemplo, si se trata de un esquema de competencia, probablemente se querrá demostrar que cada recurso compartido es utilizado en exclusión mútua por los distintos usuarios.

Ello conduce a diversos problemas encadenados: en primer lugar, cómo expresar formalmente las propiedades que se desean para el sistema; en segundo lugar, definir con precisión en qué condiciones diremos que un programa cumple la propiedad así expresada y, en tercer lugar, cómo demostrar que efectivamente la cumple.

El primero es un problema de *especificación*. Se trata de expresar en un lenguaje formal, y preferiblemente antes de construir el programa, los requisitos que deseamos para nuestro sistema.

El segundo exige una noción de *satisfacción*, es decir, definir que quiere decir la expresión *"el programa P satisface la especificación S"*.

El tercero se refiere al problema de *verificación* o de demostración de corrección. Se trata de demostrar que, con la noción de satisfacción elegida, el programa P satisface la especificación S. Para resolverlo, hay que establecer una relación precisa entre el lenguaje de especificación y el lenguaje de implementación.

Empleando una terminología similar a la utilizada en programación secuencial, se podría hablar de *corrección parcial* y de *corrección* total de un programa paralelo. Por la primera se entiende que, salvo esperas indefinidas, el programa satisface los requisitos de la especificación. La segunda incluye, además, ausencia de esperas indefinidas, es decir, ausencia de bloqueos e inaniciones.

En lugar de esta terminología, es más usual en programación paralela agrupar las propiedades que un programa ha de satisfacer en dos grandes grupos:

- *propiedades de seguridad*
- *propiedades de viveza*

Una propiedad de *seguridad* delimita lo que está permitido para un sistema de lo que no lo está. Establece lo que el sistema *puede* hacer y lo que *no debe* hacer. Un ejemplo sería la propiedad de exclusión mútua: no está permitido que dos o más usuarios utilicen al tiempo el mismo recurso.

Una propiedad de *viveza* expresa la obligación del sistema de hacer algo de lo que es posible para él. Establece lo que el sistema *debe* necesariamente hacer. Un sistema bloqueado satisface cualquier propiedad de seguridad ya que, al no hacer nada, no puede hacer nada incorrecto. La ausencia de bloqueo es interpretada por algunos autores como una propiedad de viveza: en todo estado, el sistema está obligado a realizar algún evento. Otros la incluyen entre las propiedades de seguridad, al interpretar éstas como las que se refieren a todo comportamiento finito de un sistema.

La equitatividad (*fairness*) sería en principio una propiedad de viveza: el sistema está obligado a realizar un evento que es infinitas veces posible. Sin embargo es difícil de expresar formalmente. En algunos modelos (entre ellos CSP) se renuncia de entrada a contemplar el problema y la propiedad no es ni siquiera expresable. Se relega la satisfacción de la misma a la fase de implementación del programa. Se encomienda la responsabilidad de asegurar la equitatividad a los mecanismos subyacentes al lenguaje de programación y al sistema operativo.

7. Predicados sobre trazas

Dentro del marco del modelo CSP, la observación más relevante que se puede hacer de un proceso es su traza. Dado P, **T[P]** es el conjunto de las trazas posibles para P (ver apartado 4).

Si A es un alfabeto, llamaremos *especificación* S a un conjunto de trazas sobre A que incluye la traza vacía:

$$S \subseteq A^*, <> \in S$$

Normalmente definiremos S de forma intensiva, como un predicado que tiene la variable libre *tr* que denota cualquier traza. Así, S(tr) denota el conjunto $\{ tr \in A^* \mid S(tr) \}$. La ventaja de hacer explícita *tr* es que esta variable puede ser sustituida, si conviene, por expresiones más elaboradas. Ello será útil al establecer las reglas de verificación.

Ejemplo: Sea A = {up, down}

$$S(tr) = (\forall\ tr' : tr' \leq tr : 0 \leq tr'\#up - tr'\#down \leq 1\ )$$

donde tr#c es una abreviatura de #(tr↑{c}). Especifica el conjunto de trazas *tr* tales que, en todas sus trazas prefijo, el número de eventos *up* excede como mucho en uno al de eventos *down*.

Diremos que el proceso P *satisface* la especificación S, denotado P **sat** S, si toda traza posible para P está incluida en S:

$$P \textbf{ sat } S \quad \text{sii} \quad (\forall tr: tr \in \mathbf{T}[P]: S(tr))$$

Esta noción de satisfacción es útil para verificar propiedades de seguridad. La especificación S establece el límite entre trazas permitidas y no permitidas. Si P satisface S, a P no le está permitido realizar traza alguna fuera de S.

Ahora bien, P no está obligado a realizar las trazas de S. De hecho, **stop** satisface cualquier especificación S ya que la traza vacía está siempre permitida. Esta técnica es pues inadecuada para expresar propiedades de viveza. Más adelante indicaremos cómo, cambiando la noción de satisfacción, se pueden expresar algunas de estas propiedades a costa de restringir el tipo de procesos que se pueden especificar. En el siguiente apartado se cambiará incluso la definición de especificación.

Las especificaciones basadas en trazas son convenientes porque realizan una abstracción de la actividad interna de los procesos. Dos procesos que se diferencien sólo en su actividad interna, satisfarán las mismas especificaciones. Pero, como ya se indicó en el capítulo 4, sólo las trazas no son suficientes para distinguir entre procesos deterministas y no deterministas ni entre divergentes y no divergentes.

Dada una especificación S y un proceso P, *verificar* P **sat** S consiste en mostrar que toda traza de P está en S. Para ello hay que conocer el efecto que sobre la satisfacción de especificaciones tiene cada construcción sintáctica de P. Las siguientes reglas de inferencia proporcionan esta información:

L1) P **sat** *true*

L2) si P **sat** S y P **sat** T, entonces P **sat** $S \wedge T$

L3) si P **sat** S y $S \Rightarrow T$, entonces P **sat** T

L4) **stop sat** $(tr = <>)$

L5) si P **sat** S(tr), entonces $(a \rightarrow P)$ **sat** $(tr = <> \vee (tr_0 = a \wedge S(tr')))$

L6) si P **sat** S(tr) y Q **sat** T(tr), entonces

$(a \rightarrow P)\ [] \ (b \rightarrow Q)$ **sat** $tr = <> \vee (tr_0 = a \wedge S(tr')) \vee (tr_0 = b \wedge T(tr'))$

L7) si P **sat** S(tr) y Q **sat** T(tr), entonces

$(a \rightarrow P) \lceil\rceil (b \rightarrow Q)$ **sat** $tr = <> \vee (tr_0 = a \wedge S(tr')) \vee (tr_0 = b \wedge T(tr'))$

L8) si P **sat** S(tr) y Q **sat** T(tr), entonces

$P \parallel Q$ **sat** $S(tr \uparrow \alpha P) \wedge T(tr \uparrow \alpha Q)$

L9) **stop sat** S y X **sat** S $\Rightarrow$ P(X) **sat** S, entonces $\mu X.P(X)$ **sat** S

donde tr_0 denota el primer evento de la traza tr y tr' denota el resto de tr.

Estas leyes permiten (con cierto esfuerzo) demostrar, en base a su estructura sintáctica, que un proceso satisface una especificación.

Ejemplo: El siguiente sistema representa la ejecución de dos regiones críticas en exclusión mútua. Los eventos cr_i y fr_i simbolizan respectivamente el comienzo y el fin de la región crítica para el proceso i.

$$MUTEX = \mu X.((cr_1 \rightarrow fr_1 \rightarrow X) \lceil\rceil (cr_2 \rightarrow fr_2 \rightarrow X))$$

$$S(tr) = \neg((tr\#cr_1 - tr\#fr_1 = 1) \wedge (tr\#cr_2 - tr\#fr_2 = 1))$$

La especificación S establece la propiedad de seguridad de que como máximo uno de los dos procesos está en mitad de la región crítica.

verificación: MUTEX **sat** S(tr)

a) **stop sat** S(tr). Sustituyendo <> por tr se obtiene:

$$\neg((<>\#cr_1 - <>\#fr_1 = 1) \wedge (<>\#cr_2 - <>\#fr_2 = 1)) =$$
$$\neg((0 = 1) \wedge (0 = 1)) = \textit{true}$$

b) X **sat** S(tr) $\Rightarrow$ P(X) **sat** S(tr)

P(X) **sat** $S(tr) = ((cr_1 \rightarrow fr_1 \rightarrow X) \lceil\rceil (cr_2 \rightarrow fr_2 \rightarrow X))$ **sat** S(tr) =

$(tr \leq <cr_1\ fr_1> \wedge S(tr)) \vee (tr \leq <cr_2\ fr_2> \wedge S(tr)) \vee$
$(tr \geq <cr_1\ fr_1> \wedge X$ **sat** $S(tr'')) \vee (tr \geq <cr_2\ fr_2> \wedge X$ **sat** $S(tr''))=$

true $\vee$ *true* $\vee$ (tr $\geq$ <cr_1 fr_1> $\wedge$ X **sat** S(tr")) $\vee$
(tr $\geq$ <cr_2 fr_2> $\wedge$ X **sat** S(tr"))

lo cual viene implicado por X **sat** S(tr)

Como se ha dicho, las especificaciones basadas en trazas hacen difícil expresar propiedades de viveza, precisamente porque las trazas representan la historia pasada del proceso pero no la obligación de involucrarse en eventos futuros.

Una propuesta para paliar este problema consiste en modificar la noción de satisfacción manteniendo como únicas observaciones posibles de un proceso sus trazas. Así, una especificación S continúa siendo un conjunto de trazas, pero la notación P **sat** S significa ahora:

1) **T**[P] $\subseteq$ S (propiedad de seguridad)
2) P no diverge
3) si s $\in$ S, s $\in$ **T**[P] y s.b $\in$ S entonces P, tras ejecutar s, no puede rechazar b si el entorno insiste (propiedad de viveza).

El problema con esta noción de satisfacción es que los únicos procesos que pueden satisfacer una especificación dada son los *deterministas*, es decir, aquellos que en su forma normal no figura el operador $\lceil\rceil$. Esta parece una restricción poco razonable ya que dicho operador aparece casi inevitablemente cuando en P intervienen los operadores |||, \ y [].

8. Predicados sobre trazas y conjuntos rechazables

Para no restringir el dominio de procesos especificables, se puede enriquecer la noción de observaciónes de un proceso haciendo que, además de las trazas, se puedan observar informaciones sobre su comportamiento futuro. En [Hoa 85] se propone que la especificación de un proceso incluya las trazas permitidas y los conjuntos rechazables permitidos tras cada traza.

Los conjuntos rechazables dan información negativa (el proceso se niega a participar en los eventos de X ofrecido por el entorno) útil para razonar sobre lo que *debe* ocurrir tras una traza s.

Una especificación S(tr, ref) es un predicado con dos variables libres: *tr* denota cualquier traza del proceso y *ref*, cualquier conjunto rechazable tras ejecutar la traza *tr*.

La noción de satisfacción es ahora:

$$P \textbf{ sat } S(tr, ref) \text{ sii } \forall\, tr, ref: tr \in \mathbf{T}[P] \wedge ref \in \text{rechazos } (P/s): S(tr, ref)$$

Ejemplos:

Una máquina expendedora de café tiene dos eventos {moneda, café}. Se quiere especificar que, siempre que el número de monedas introducidas exceda al de cafés suministrados, la máquina no puede negarse a suministrar un nuevo café (propiedad de viveza):

$$S(tr, ref) = (tr\#moneda > tr\#café \Rightarrow café \notin ref)$$

Una especificación más completa de la misma máquina establecería que la máquina no da cafés por adelantado (cuando el número de monedas y de cafés es el mismo, sólo acepta monedas), pero puede aceptar varias monedas antes de dar el primer café:

$$\begin{aligned} S(tr, ref) = \quad & (tr\#moneda > tr\#café \Rightarrow café \notin ref) \wedge \\ & (tr\#moneda = tr\#café \Rightarrow ref = \{café\}) \wedge \\ & (tr\#café \leq tr\#moneda) \end{aligned}$$

La ausencia de bloqueo, propiedad típica de viveza, puede ser especificada de modo muy simple. Si A es el alfabeto del proceso,

$$NOBLOQUEO\ (tr, ref) = (ref \neq A)$$

es decir, tras ejecutar cualquier traza tr, al menos un evento de A no puede ser rechazado.

En el modelo de fallos, el proceso divergente **div** es aquel que puede ejecutar cualquier traza y rechazar cualquier conjunto de eventos en todo momento (ver capítulo 4). Una condición suficiente para que un proceso no diverja es que, en particular, no se bloquee. Por tanto, podemos escribir el predicado

$$NODIV\ (tr, ref) = (ref \neq A)$$

como condición suficiente de no divergencia.

- - -

Las leyes de verificación, cuando intervienen conjuntos rechazables en las especificaciones, se complican ligeramente. Son las siguientes:

L4') $\mathbf{stop}_A$ **sat** $(tr = <> \wedge ref \subseteq A)$

L5') si P **sat** S(tr), entonces $(a \rightarrow P)$ **sat** $(tr = <> \wedge a \notin ref) \vee (tr_0 = a \wedge S(tr'))$

L6') si P **sat** S y Q **sat** T, entonces

$P \,[]\, Q$ **sat** $(tr = <> \wedge S \wedge T) \vee (tr \neq <> \wedge (S \vee T))$

L7') si P **sat** S y Q **sat** T, entonces

$P \sqcap Q$ **sat** $S \vee T$

L8') si P **sat** S(tr, ref) y Q **sat** T(tr, ref) y P y Q no divergen, entonces

$P \,||\, Q$ **sat** $\exists X, Y \subseteq A : ref = X \cup Y \wedge S(tr \uparrow \alpha P, X) \wedge T(tr \uparrow \alpha Q, Y)$

L9') Sea S(n) un predicado que contiene la variable natural n. Si S(0) y $(X$ **sat** $S(n)) \Rightarrow$ $(P(X)$ **sat** $S(n + 1))$, entonces $\mu X.P(X)$ **sat** $(\forall n: S(n))$

L10) si P **sat** S(tr, ref) y P no diverge, entonces

$P \setminus b$ **sat** $\exists s \in A^*: tr = s \uparrow (\alpha P - \{b\}) \wedge S(s, ref \cup \{b\})$

9. Transformación algebraica de procesos

La principal ventaja de trabajar dentro de un modelo formal como CSP es que existe una noción precisa de igualdad entre procesos. Las leyes algebraicas del apartado 5 pueden utilizarse para simplificar procesos. Si P es igual a Q, todas las propiedades válidas para Q, también lo son para P. Si Q es más simple que P en algún sentido, puede ser una ventaja manipular primero P para transformarlo en Q y demostrar después sobre Q las propiedades deseadas.

En particular, esta estrategia es útil cuando Q procede de ocultar eventos en P. P podría ser una red de procesos compuestos en paralelo mediante el operador ||, C el conjunto de eventos que representan las sincronizaciones entre los componentes de P y Q = P \ C. Sin duda Q es un proceso más simple que P por tener menos eventos. Si se logra expresar Q en términos de los operadores primitivos $\rightarrow$, [] y $\sqcap$, es decir, si se logran eliminar los operadores || y \, Q será un proceso más cómodo que P para demostrar propiedades.

Ejemplo: Sean

$$USUARIO = \mu X.(p \rightarrow cr \rightarrow fr \rightarrow v \rightarrow X)$$

$$U_i = USUARIO\ [cr_i / cr\ ][fr_i / fr]$$

$$SEMAFORO = \mu X.(p \rightarrow v \rightarrow X)$$

$$RED = \left(\left|\!\left|\!\left| \prod_{i=1}^{n} U_i \right) \right|\!\right|\!\right| SEMAFORO$$

$$SISTEMA = RED \setminus \{p, v\}$$

Utilizando las leyes algebraicas del apartado 5 se puede manipular SISTEMA y demostrar que es equivalente al proceso:

$$SISTEMA = \mu X.\left(\left[\!\right]_{i=1}^{n} (cr_i \rightarrow fr_i \rightarrow X) \right)$$

En esta forma reducida, es más fácil demostrar la propiedad de exclusión mútua que puede expresarse así:

$$SISTEMA\ \textbf{sat}\ (N\ j:\ 1 \leq j \leq n : (tr\#cr_j - tr\#fr_j) = 1) \leq 1$$

donde N es el cuantificador de conteo. El predicado expresa que el número de procesos j que se halla dentro de la región crítica es a lo sumo 1.

En [PeAl 89] se amplian las leyes de transformación para procesos CSP especificados según una *forma normal* en la que intervienen especificaciones algebraicas de tipos abstractos de datos.

REFERENCIAS

[BrRo 85] Brookes, S.D.; Roscoe, A.W.
An Improved Failures Model for Communicating Processes
LNCS No. 197, Springer_Verlag 1985, pp. 281-305

[BHR 84] Brookes, S.D.; Hoare, C.A.R.; Roscoe, A.W.
A Theory of Communicating Processes

[Hoa 85] Hoare, C.A.R.
Communicating Sequential Processes
Prentice-Hall, 1985

[Mil 80] Milner, R.
A Calculus of Communicating Systems
LNCS, No. 92, Springer-Verlag 1980

[PeAl 89] Pèña, R.; Alonso, L.M.
Specification and Verification of TCSP Systems by Means of Partial Abstract Data Types
TAPSOFT'89, LNCS no. 352, Springer-Verlag 1989, pp. 328-344

[Ro 85] Roscoe, A.W.
Denotational Semantics for Occam
LNCS n. 197, Seminar on concurrency, Springer-Verlag 1985, pp. 306-329

[RoHo 85] Roscoe, A.W.; Hoare, C.A.R.
The Laws of Occam Programming
TCS 60, 1988, pp.177-229