Introducción a CSP

Maximiliano Cristiá

Ingeniería de Software 1
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

2022

Índice

1. Características generales de CSP

El formalismo *Communicating Sequential Processes* (CSP) fue propuesto por Tony Hoare entre 1975 y 1985 como un marco teórico-práctico dentro del cual poder estudiar formalmente el problema de la concurrencia y dominar su complejidad [1]. El trabajo seminal de Hoare fue continuado por otros hasta entrados los noventa cuando se propusieron variantes y extensiones al lenguaje original para incluir conceptos tales como tiempo real, *broadcasting*, comunicación asíncrona, etc. Es interesante señalar que la idea original de CSP nació durante un período en el cual Hoare trabajó con Edsger Dijkstra.

Al contrario de las notaciones Z y Statecharts, en CSP no hay una noción explícita de estado. En CSP, procesos y eventos son las nociones centrales. Los procesos se construyen combinando eventos y otros procesos por medio de operadores, formándose así un *álgebra de procesos*. Aunque suene extraño, todos los procesos en CSP son secuenciales a pesar de que el lenguaje fue diseñado para estudiar el problema de la concurrencia.

Si bien CSP es un formalismo sólido, completo, intuitivo y práctico tiene una desventaja frente a, por ejemplo, TLA pues especificaciones y propiedades se deben escribir en lenguajes diferentes. Las especificaciones se escriben en CSP en tanto que sus propiedades se escriben en lógica. Esto implica que no hay un lenguaje unificado para la verificación de propiedades de especificaciones.

Es difícil decir si CSP es un lenguaje tipado o no. La noción de tipo se utiliza superficialmente en contadas oportunidades.

A lo largo del apunte se desarrollará un ejemplo, que se enuncia en la sección 2, a través del cual se irán introduciendo diversos conceptos de CSP a medida que vayan siendo necesarios.

2. Un típico problema industrial

En esta sección enunciaremos los requisitos para un programa que debe resolver un típico problema industrial. En cierta planta industrial existe una celda de producción formada por un robot industrial con dos brazos, dos cintas transportadoras y una prensa. Cada brazo del robot debe tomar una pieza de una de las cintas y depositarla en la prensa (cada brazo toma piezas de una única cinta). Cada cinta transportadora emite una señal cada vez que una pieza pasa cerca de un sensor, con suficiente antelación como para que el brazo correspondiente pueda tomarla. Cada brazo sabe dónde está la pieza que debe tomar (no modelaremos este conocimiento). El robot mueve los brazos a partir de señales que recibe desde el exterior. Los brazos se detienen automáticamente cuando alcanzan tanto su cinta como la prensa. Las señales del robot y la cinta transportadora son las siguientes:

El sensor de la cinta transportadora detecta una pieza ≈ *item*

El brazo de robot toma una pieza de la cinta transportadora ≈ *take*

El brazo de robot comienza a moverse hacia la prensa ≈ *topress*

El brazo del robot suelta la pieza que tiene tomada ≈ *release*

El brazo del robot comienza a moverse hacia la cinta ≈ *tobelt*

Si la prensa está libre el brazo que haya llegado a ella debe soltar la pieza y debe retornar a su cinta; si la prensa está ocupada, hay que esperar a que se libere.

La prensa puede recibir señales para prensar y remover la pieza prensada, y emite señales cuando el prensado finaliza y cuando está libre como para volver a prensar. En resumen las señales son las siguientes:

La prensa está libre y lista para volver a prensar ≈ *free*

La prensa comienza a prensar ≈ *press*

El prensado ha terminado, es decir la prensa no ejerce presión sobre la pieza ≈ *stoppress*

Se retira la pieza de la prensa ≈ *remove*

Toda la celda se enciende (*start*) y apaga (*abort*) por medio de un interruptor electrónico.

El software de control debe llevar los dispositivos a un estado conocido cuando el sistema se enciende.

3. Eventos, procesos y recursividad

Comenzaremos por especificar el conocimiento de dominio de la celda de producción; en esta sección daremos una primera especificación (errónea) del comportamiento de la prensa.

Como mencionamos en la introducción, las especificaciones CSP se estructuran en *procesos* los cuales se definen por medio de eventos, operadores y otros procesos. La forma básica de un proceso es $NOMBRE\_PROCESO = definicion \ \ldots$, donde el nombre usualmente se escribe en mayúsculas y en la definición pueden participar eventos (que se escriben en minúsculas), operadores del álgebra de procesos de CSP y los nombres de otros procesos que pueden estar definidos o por definirse. Por ejemplo, veamos un proceso que describe, aunque erróneamente, el comportamiento natural de la prensa.

$$
\begin{aligned}
&DK\_PRESS = start \rightarrow PRESS1 \\
&PRESS1 = \overline{free} \rightarrow PRESS2 \\
&PRESS2 = press \rightarrow \overline{stoppress} \rightarrow remove \rightarrow PRESS1
\end{aligned}
\tag{1}
$$

En tanto el significado formal del proceso *DK_PRESS* lo veremos en la sección 15, por el momento nos limitaremos a decir que *DK_PRESS* esperará indefinidamente a que su entorno provea el evento *start* y cuando esto suceda se comportará según la especificación del proceso *PRESS*1. A su vez *PRESS*1 en algún momento emitirá el evento *free* luego de lo cual se comportará como el proceso *PRESS*2. Este último esperará indefinidamente a que su entorno provea el vento *press* luego de lo cual en algún momento emitirá el evento *stoppress* para a continuación esperar

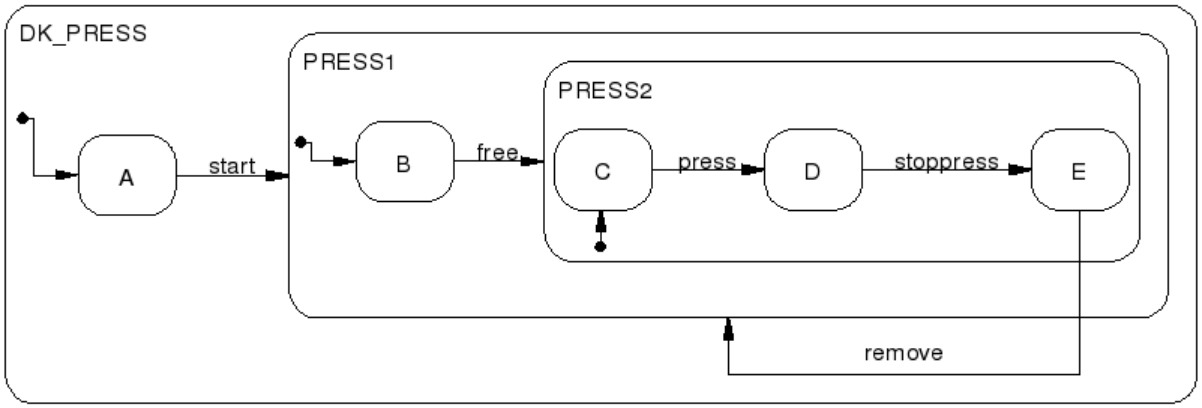

Figura 1: Un Statechart equivalente al proceso CSP *DK_PRESS*.

que el entorno emita *remove*, de forma tal que finalmente se comportará como el proceso *PRESS*1. En otras palabras *DK_PRESS* se comporta como el Statechart mostrado en la Figura 1.

En otro orden, la definición de *DK_PRESS* es recursiva en tanto que alguno de sus componentes (*PRESS*1 en este caso) está definido circularmente. Esto es muy común en CSP. Incluso es perfectamente legal y usual escribir procesos que son mutuamente recursivos.

La barra horizontal sobre eventos como *free* (es decir, $\overline{free}$ en *PRESS*1) no es parte de la formulación original de Hoare. Nosotros la incluimos a partir del trabajo de Allen [2] donde el autor utiliza CSP para dar la semántica de un lenguaje para descripción de arquitecturas de software. En ese trabajo Allen debe preservar la noción de cuál entidad emite cada evento y cuáles lo esperan. Según su notación, si el proceso P es el que emite (inicia o controla) el evento e entonces ese evento se escribe con una barra horizontal ($\overline{e}$) sólo en la definición del proceso P; en todos los otros procesos que forman parte de la especificación, e se escribe sin barra horizontal, indicando que esos procesos esperan a e. En otras palabras e y $\overline{e}$ son el mismo evento solo que se conviene que el proceso donde aparece $\overline{e}$ es el que lo emite. Esta noción de la entidad que controla cada evento está en consonancia con el trabajo de Jackson y Zave [3, 4] por lo que nosotros decidimos adoptarla.

Cuando escribimos un proceso usando $\rightarrow$ decimos que usamos *prefijos* o *prefijación* (*prefixing*). La $\rightarrow$ siempre se escribe entre dos eventos, excepto la última que se escribe entre un evento y un proceso. La flecha tiene la máxima prioridad entre los operadores CSP.

DK_PRESS también se podría haber escrito así:

$$DK\_PRESS = start \rightarrow PRESS3$$
$$PRESS3 = \overline{free} \rightarrow press \rightarrow \overline{stoppress} \rightarrow remove \rightarrow PRESS3$$

y también así:

$$DK\_PRESS = start \rightarrow PRESS4$$
$$PRESS4 = \overline{free} \rightarrow PRESS5$$
$$PRESS5 = press \rightarrow PRESS6$$
$$PRESS6 = \overline{stoppress} \rightarrow remove \rightarrow PRESS4$$

En general la forma de factorizar la definición de un proceso depende del gusto del especificador, de la legilibiliad y del reuso de algunos procesos en otras partes de la especificación.

4. Alternativa etiquetada, selección externa e interrupción

La especificación del comportamiento de la prensa no es correcta porque desde el entorno no solo puede encenderse (*start*) sino que también puede apagarse (*abort*), lo que no está representado en la especificación de *DK_PRESS*. Para poder representar este comportamiento alternativo necesitamos una construcción del lenguaje que nos permita expresar que un proceso puede comportarse de varias formas diferentes según se seleccione desde su entorno.

Existen tres construcciones que nos permiten expresar lo antedicho: $|$, llamada alternativa etiquetada; $\Box$, llamada *selección externa* (*external choice*) o *box*; y ∇, llamado interrupción. Los más utilizados son $|$ y $\Box$ aunque en nuestro caso deberemos recurrir a ∇. Veamos cada uno de ellos:

- El proceso $P = e \rightarrow Q \mid f \rightarrow R$ ofrece las alternativas e y f a su entorno (recordar que $\rightarrow$ tiene la máxima prioridad). Es decir que si el entorno quiere interactuar con P por medio de e o f puede hacerlo. La interacción entre P y su entorno determina por cuál de los dos caminos deberá transitar P.

  $P = Q \mid R$ no es legal en CSP. A ambos lados de $|$ deben estar explícitas las dos alternativas que P ofrece a su entorno y estas deben ser diferentes (en otras palabras también es ilegal $e \rightarrow P \mid e \rightarrow Q$). En la definición de un proceso se pueden utilizar todas las alternativas etiquetadas que se necesiten pero cada una debe estar explícitamente precedida por su etiqueta (primer evento); es decir $P = e_1 \rightarrow P_1 \mid e_2 \rightarrow P_2 \mid \cdots \mid e_n \rightarrow P_n$.

  Tampoco es legal escribir $P = \mid_{i=1}^{n} e_i \rightarrow P_i$.

- El proceso $P = Q \,\Box\, R$ se comporta como Q o como R según la interacción entre P y su entorno. No es obligatorio que sea explícito el primer evento de cada comportamiento posible.

  Es legal escribir cosas como $P = e \rightarrow Q \,\Box\, R$, $e \rightarrow P \,\Box\, e \rightarrow Q$, $P = \Box_{i=1}^{n} e_i \rightarrow P_i$, o $P = \Box i \in [1, n] \bullet e_i \rightarrow P_i$ (esta notación también es introducida en [2] y recuerda a la notación utilizada por Gries para cálculo de programas).

  En otras palabras, el significado de $|$ y $\Box$ es idéntico para los términos legales donde se usa únicamente $|$, pero el último admite términos que el primero no.

- Finalmente, $P = Q \,\nabla\, R$ se comporta como Q hasta tanto el entorno de P interactúe con el primer evento de R, a partir de lo cual se comportará como R. Es decir, el primer evento de R actúa como una *interrupción* para Q: si aparece el primer evento de R la ejecución de Q debe ser inmediatamente abortada y debe continuarse con R.

Por lo tanto, describir el comportamiento de la prensa con $DK\_PRESS = start \rightarrow PRESS1 \mid abort \rightarrow DK\_PRESS$ o con $\Box$ en lugar de $|$, sería incorrecto puesto que *abort* representa el apagado eléctrico de toda la celda lo que implica que la prensa debe detenerse sea lo que fuere que esté haciendo en ese momento. En consecuencia el verdadero comportamiento de la prensa se expresa por medio de:

$$DK\_PREES = start \rightarrow PRESS1 \,\nabla\, abort \rightarrow DK\_PRESS \tag{2}$$

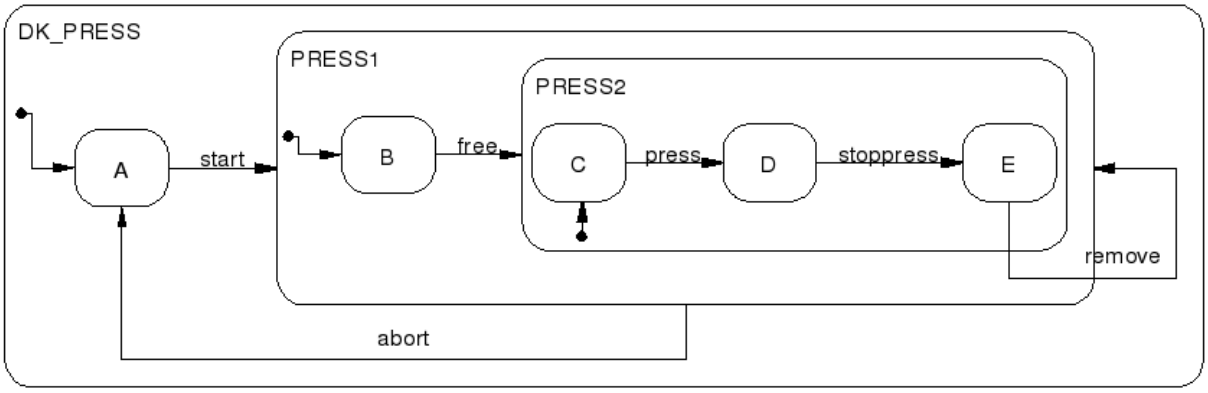

Figura 2: Statechart equivalente a *DK_PRESS* donde se usa ∇.

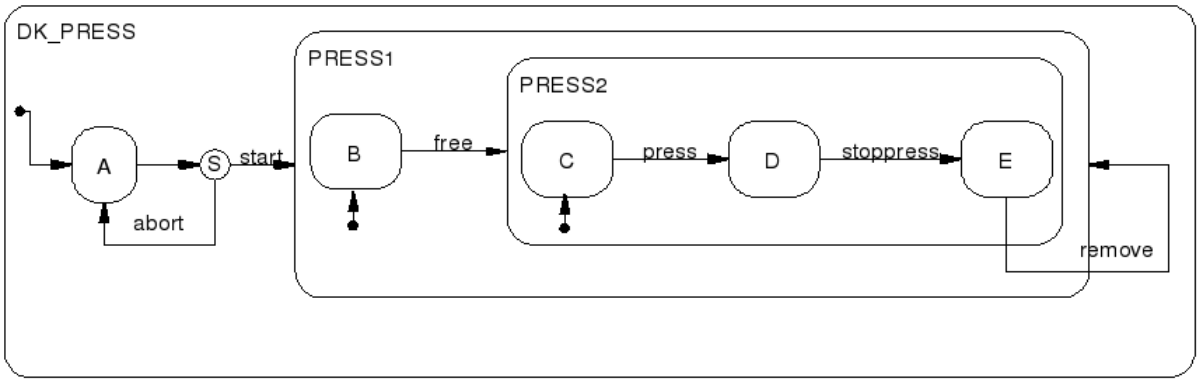

Figura 3: Statechart equivalente a *DK_PRESS* si se hubiera usado □ en lugar de ∇.

Creemos que los Statecharts de las Figuras 2 y 3 clarifican el significado de ∇ frente a □. De hecho, como veremos más adelante, ∇ se define en términos de □ por lo que *DK_PRESS* podría escribirse así (o reemplazando | por □):

$$DK\_PRESS = start \rightarrow PRESS1 \mid abort \rightarrow DK\_PRESS$$
$$PRESS1 = \overline{free} \rightarrow PRESS2 \mid abort \rightarrow DK\_PRESS$$
$$PRESS2 = press \rightarrow (\overline{stoppress} \rightarrow (remove \rightarrow PRESS1 \mid abort \rightarrow DK\_PRESS) \mid abort \rightarrow DK\_PRESS) \mid abort \rightarrow DK\_PRESS$$

Es decir, siempre se le da al entorno la posibilidad de abortar el normal funcionamiento de la prensa. Obviamente es mucho mejor la primer forma de *DK_PRESS* puesto que es mucho más concisa y legible.

4.1. Primera aproximación a las leyes algebraicas de CSP

Como ya hemos mencionado los operadores de CSP definen un álgebra de procesos. Esto significa que los operadores de CSP verifican ciertas leyes algebraicas tales como idempotencia, distributividad, asociatividad, etc. Cada ley se expresa por medio de una igualdad a los lados de la cual se escriben dos expresiones que representan sendos procesos. La igualdad de ambas expresiones implica que los procesos que ellas denotan son, en realidad, el mismo escrito de dos formas diferentes. Cuando decimos que ambas expresiones son el mismo proceso queremos decir que ambos tienen el mismo patrón de interacción con el entorno (veremos con más formalidad esto en la sección 15).

Que CSP constituya un álgebra de procesos no es un detalle menor. El hecho de poder expresar las propiedades de los operadores del lenguaje mediante leyes algebraicas tiene importantes consecuencias:

1. Nos permiten mejorar la comprensión e intuición del significado deseado para los operadores.
2. Son muy útiles a la hora de hacer pruebas de propiedades de procesos CSP (como veremos más adelante).
3. Si son presentadas y analizadas sistemáticamente, se puede probar que definen completamente la semántica de CSP (aunque en la sección 15 nosotros seguiremos un camino diferente).

   Este punto es interesante porque es posible dar la semántica de CSP y de allí deducir las leyes de los operadores o axiomatizar el álgebra y a partir de ella obtener el modelo semántico que veremos en la sección 15.

En [5, capítulo 5] hay un catálogo muy completo y ordenado de las leyes de CSP, aunque Roscoe en [6, página 271] dice que una gran proporción de esas leyes son falsas y que incluso algunas de ellas ni si quiera son leyes, por ejemplo $P \nabla Q \wedge Q \nabla R \Rightarrow P \nabla R$ pues los procesos no son proposiciones lógicas; por esta razón nosotros transcribimos en general las leyes de Roscoe. A continuación enunciaremos solo algunas de las leyes que verifican los operadores vistos hasta el momento (recordar que $\rightarrow$ tiene máxima prioridad excepto por los paréntesis).

$$e \rightarrow P \,\Box\, f \rightarrow Q = e \rightarrow P \mid f \rightarrow Q \tag{3}$$

$$P \,\Box\, P = P \tag{4}$$

$$P \,\Box\, Q = Q \,\Box\, P \tag{5}$$

$$P \,\Box\, (Q \,\Box\, R) = (P \,\Box\, Q) \,\Box\, R \tag{6}$$

$$e \rightarrow P \nabla Q = Q \,\Box\, e \rightarrow (P \nabla Q) \tag{7}$$

$$P \nabla (Q \nabla R) = (P \nabla Q) \nabla R \tag{8}$$

$$P \nabla (Q \,\Box\, R) = (P \nabla Q) \,\Box\, (P \nabla R) \tag{9}$$

Las leyes enunciadas expresan claramente que:

- $\mid$ y $\Box$ son idénticos si se dan explícitamente los primeros eventos de cada proceso.
- $\Box$ es idempotente, conmutativo y asociativo.
- La ley 7 actúa como definición de ∇ y muestra claramente que, aplicado a *DK_PRESS*, es equivalente al Statechart mostrado en la Figura 2 y al proceso 4.
- ∇ es asociativo y es distributivo respecto de $\Box$.

5. Concurrencia e intercalación: procesos secuenciales

Con los elementos introducidos hasta el momento podemos dar una especificación del conocimiento de dominio de una celda de producción que cuenta con una única cinta transportadora y un robot con un único brazo.

$$DK\_SWITCH = \overline{start} \rightarrow \overline{abort} \rightarrow DK\_SWITCH$$
$$DK\_BELT = start \rightarrow BELT1 \,\nabla\, abort \rightarrow DK\_BELT$$
$$BELT1 = \overline{item} \rightarrow BELT1$$
$$DK\_ARM = start \rightarrow ARM1 \,\nabla\, abort \rightarrow DK\_ARM$$
$$ARM1 = take \rightarrow ARM1 \,\Box\, topress \rightarrow ARM1$$
$$\Box\, release \rightarrow ARM1 \,\Box\, tobelt \rightarrow ARM1$$

Aquí vemos que los eventos *start* y *abort* son iniciados por el interruptor electrónico general. El primero de ellos inicia todos los componentes de la celda, en tanto que pueden ser interrumpidos por *abort*. Además, la cinta transportadora emite una señal cada vez que una pieza pasa cerca de su sensor. El comportamiento que describimos para el brazo es uno de los tantos posibles dependiendo del hardware con que se cuente (más abajo analizaremos otra posibilidad).

Si hubiésemos descripto este conocimiento con Statecharts, el paso final hubiese sido componer paralelamente todos ellos. Lo mismo puede hacerse en CSP utilizando el operador ||, que simboliza composición paralela:

$$DK\_CELL = DK\_ARM \,||\, DK\_BELT \,||\, DK\_SWITCH \,||\, DK\_PRESS \tag{10}$$

El operador paralelo (||) es un operador más del álgebra, no ocupa ningún lugar en particular y verifica algunas leyes como cualquier otro operador. Intuitivamente el significado de $P \,||\, Q$ es que P y Q ejecutarán en paralelo o concurrentemente, es decir simultáneamente. Existen tres leyes fundamentales que gobiernan el significado de ||, por el momento veremos solo la siguiente (en la sección 8 veremos las dos restantes y al final de esta sección veremos otras menos importantes):

$$e \notin \alpha P, f \notin \alpha Q \Rightarrow f \rightarrow P \,||\, e \rightarrow Q = f \rightarrow (P \,||\, e \rightarrow Q) \,\Box\, e \rightarrow (f \rightarrow P \,||\, Q) \qquad \text{(Interleaving)}$$

donde si P es un proceso αP es su *alfabeto*, es decir el conjunto de eventos que de una forma u otra participan en su definición. En las leyes de CSP en las que se mencionan explícitamente eventos se conviene en que (a) un nombre de evento representa cualquier evento y (b) si dos eventos tienen nombres diferentes nunca representan el mismo evento. Por lo tanto en Interleaving, e y f son dos eventos cualesquiera pero diferentes.

La ley Interleaving dice que si un proceso es el resultado de la composición paralela de dos procesos cuyos primeros eventos no son comunes entre ellos, entonces, desde el entorno, el proceso se comportará como si ofreciera ambas posibilidades (pero no simultáneamente, de allí el □) para luego "consumir" la posibilidad elegida. De esta forma el resultado es que se dan las intercalaciones posibles entre los dos eventos, es decir:

- El entorno puede elegir por interactuar a través de e o a través f.

- Si primero se consume e, luego podría consumirse f.
- Si primero se consume f, luego podría consumirse e.

Esta ley comienza a mostrar que en CSP la composición paralela entre dos procesos se entiende como un proceso secuencial que contempla todas las intercalaciones posibles. En otras palabras, la semántica de CSP responde a lo que se conoce como el *modelo de intercalaciones para la concurrencia* (*interleaving model of concurrency*). Este modelo ve a la concurrencia como la intercalación de todos los caminos de ejecución posibles entre los dos procesos que corren en paralelo, lo que en realidad no es verdadera concurrencia.

Por ejemplo, la ley Interleaving permite probar el siguiente teorema.

Teorema 1.

$$A_1 = a_1 \rightarrow A_1 \wedge A_2 = a_2 \rightarrow A_2 \wedge A = A_1 \,||\, A_2 \Rightarrow A = a_1 \rightarrow A \,\Box\, a_2 \rightarrow A$$

Demostración. La prueba es muy simple. Comenzamos por aplicar las definiciones de A, A_1 y A_2:

$$A = A_1 \,||\, A_2 = a_1 \rightarrow A_1 \,||\, a_2 \rightarrow A_2 \tag{T1.a}$$

Aplicando Interleaving obtenemos:

$$A = a_1 \rightarrow (A_1 \,||\, a_2 \rightarrow A_2) \,\Box\, a_2 \rightarrow (a_1 \rightarrow A_1 \,||\, A_2) \tag{T1.b}$$

Pero, por definición, $a_2 \rightarrow A_2 = A_2$ y $a_1 \rightarrow A_1 = A_1$. Por lo tanto (T1.b) queda así:

$$A = a_1 \rightarrow (A_1 \,||\, A_2) \,\Box\, a_2 \rightarrow (A_1 \,||\, A_2) \tag{T1.c}$$

Y reemplazando $A_1 \,||\, A_2$ por A demostramos el teorema:

$$A = a_1 \rightarrow A \,\Box\, a_2 \rightarrow A \tag{T1.d}$$

□

En resumen, la composición paralela de dos procesos que no comparten eventos equivale a un proceso secuencial. Esta es la regla fundamental de CSP: todos los procesos son secuenciales aunque parezcan concurrentes. De aquí el nombre de CSP: procesos *secuenciales* que se comunican. Más adelante veremos otras leyes que refuerzan este concepto esencial en CSP.

Al comienzo de la sección mencionamos que el comportamiento descripto para el brazo del robot podía ser otro si el hardware se comportaba de manera diferente. Consideremos la siguiente posibilidad:

$$DK\_ARM = start \rightarrow ARM1 \,\nabla\, abort \rightarrow DK\_ARM$$
$$ARM1 = ARM11 \,||\, ARM12$$
$$ARM11 = take \rightarrow release \rightarrow ARM11$$
$$ARM12 = topress \rightarrow tobelt \rightarrow ARM12$$

Probar que $ARM1$ es un proceso secuencial es algo más complicado pues es necesario recurrir a procesos mutuamente recursivos.

Teorema 2. *ARM*1 *es un proceso secuencial.*

Demostración. Comenzamos expandiendo la definición de *ARM*1:

$$\begin{aligned} ARM1 &= ARM11 \,||\, ARM12 \\ &= take \longrightarrow release \longrightarrow ARM11 \,||\, topress \longrightarrow tobelt \longrightarrow ARM12 \end{aligned} \tag{T2.a}$$

Ahora aplicamos Interleaving:

$$\begin{aligned} ARM1 =& take \longrightarrow (release \longrightarrow ARM11 \,||\, topress \longrightarrow tobelt \longrightarrow ARM12) \\ & \mid topress \longrightarrow (take \longrightarrow release \longrightarrow ARM11 \,||\, tobelt \longrightarrow ARM12) \end{aligned} \tag{T2.b}$$

Damos nombres apropiados a los procesos que siguen a *take* y *topress*:

$$ARM1A = release \longrightarrow ARM11 \,||\, topress \longrightarrow tobelt \longrightarrow ARM12 \tag{T2.c}$$

$$ARM1B = take \longrightarrow release \longrightarrow ARM11 \,||\, tobelt \longrightarrow ARM12 \tag{T2.d}$$

Volvemos a aplicar Interleaving a (T2.c) y (T2.d):

$$\begin{aligned} ARM1A =& release \longrightarrow (ARM11 \,||\, topres \longrightarrow tobelt \longrightarrow ARM12) \\ & \mid topress \longrightarrow (release \longrightarrow ARM11 \,||\, tobelt \longrightarrow ARM12) \end{aligned} \tag{T2.e}$$

$$\begin{aligned} ARM1B =& take \longrightarrow (release \longrightarrow ARM11 \,||\, tobelt \longrightarrow ARM12) \\ & \mid tobelt \longrightarrow (take \longrightarrow release \longrightarrow ARM11 \,||\, ARM12) \end{aligned} \tag{T2.f}$$

Notar que el proceso $release \longrightarrow ARM11 \,||\, tobelt \longrightarrow ARM12$ se repite en las dos ecuaciones anteriores. Entonces lo nombramos $ARM1C$:

$$ARM1C = release \longrightarrow ARM11 \,||\, tobelt \longrightarrow ARM12 \tag{T2.g}$$

Por lo tanto, reemplazando (T2.g) (más las definiciones de $ARM11$, $ARM12$ y ARM) en (T2.e) y (T2.f), obtenemos:

$$ARM1A = release \longrightarrow ARM1 \mid topress \longrightarrow ARM1C \tag{T2.h}$$

$$ARM1B = take \longrightarrow ARM1C \mid tobelt \longrightarrow ARM1 \tag{T2.i}$$

De esta forma $ARM1A$ y $ARM1B$ son procesos secuenciales, excepto por $ARM1C$ que sigue siendo una composición paralela. Veamos si $ARM1C$ es un proceso secuencial aplicando Interleaving:

$$\begin{aligned} ARM1C =& release \longrightarrow (ARM11 \,||\, tobelt \longrightarrow ARM12) \\ & \mid tobelt \longrightarrow (release \longrightarrow ARM11 \,||\, ARM12) \end{aligned} \tag{T2.j}$$

pero si miramos con atención, por ejemplo, $ARM11 \,||\, tobelt \longrightarrow ARM12$ es $ARM1B$ (solo es necesario expandir $ARM11$). Por lo tanto, (T2.j) queda así:

$$ARM1C = release \longrightarrow ARM1B \mid tobelt \longrightarrow ARM1A \tag{T2.k}$$

que es lo que buscábamos. Entonces, partiendo de (T2.b) y reemplazando convenientemente tenemos:

$$ARM1 = take \longrightarrow ARM1A \mid topress \longrightarrow ARM1B$$
$$ARM1A = release \longrightarrow ARM1 \mid topress \longrightarrow ARM1C$$
$$ARM1B = tobelt \longrightarrow ARM1 \mid take \longrightarrow ARM1C$$
$$ARM1C = release \longrightarrow ARM1B \mid tobelt \longrightarrow ARM1A$$

Es decir, desapareció el paralelismo y por lo tanto $ARM1$ es un proceso secuencial (aunque mutuamente recursivo). □

La posibilidad de probar estos teoremas no significa que deban probarse siempre y mucho menos que siempre se deban escribir especificaciones como procesos secuenciales. La idea es, precisamente, aprovechar la simplicidad de la composición paralela definiendo especificaciones que hagan uso de $\|$ siempre que sea posible. De esta forma, las implementaciones podrán paralelizarse tanto como indica la especificación, o podrá buscarse el proceso secuencial equivalente e implementarse como un único programa secuencial. Por otro lado, podríamos estar tentados de pensar que en CSP los procesos paralelos son secuenciales porque hasta el momento hemos tratado únicamente con procesos que no comparten eventos, sin embargo en la sección 8 veremos que lo mismo sigue siendo válido para procesos que comparten uno o más eventos.

Cerramos la sección mostrando algunas de las leyes más o menos secundarias de $\|$ y algunas de las que lo relacionan con los otros operadores que ya hemos definido.

$$P \| P = P \tag{11}$$
$$P \| Q = Q \| P \tag{12}$$
$$P \| (Q \| R) = (P \| Q) \| R \tag{13}$$
$$P \| (Q \,\square\, R) = (P \| Q) \,\square\, (P \| R) \tag{14}$$

5.1. Alfabeto de un proceso

Más arriba mencionamos el concepto de alfabeto de un proceso. Si bien no nos interesa adentrarnos demasiado en este concepto resulta necesario mencionar las siguientes leyes:

$$\alpha(a \longrightarrow P) = \alpha P \tag{15}$$
$$\alpha(a \longrightarrow P \mid b \longrightarrow Q) = \alpha P = \alpha Q \tag{16}$$
$$\alpha(P \| Q) = \alpha P \cup \alpha Q \tag{17}$$

Respecto a la primera ley, claramente a forma parte del alfabeto de $a \longrightarrow P$. Entonces la ley estipula que P debe "acarrear" ese evento. Dicho de otro modo se fuerza a que a pertenezca al alfabeto del proceso P. Por ejemplo, si $P = b \longrightarrow P$ entonces $\alpha(a \longrightarrow P) = \{a, b\}$, aunque a no está explícitamente en la definición de P. Algo similar vale para la segunda ley

Ejercicios

Ejercicio 1. *Probar que el proceso $ARM1 \| BELT1$ es en realidad un proceso secuencial.*

6. Renombramiento funcional e indexado

La especificación del entorno (10) es incompleta porque la celda de producción consta de dos cintas transportadoras y dos brazos robóticos. Por otro lado, ambas cintas y ambos brazos tienen exactamente el mismo comportamiento excepto que los eventos emitidos o esperados por ellos no pueden tener los mismos nombres (de otra forma, por ejemplo, el evento *take* podría ser recibido por cualquiera de los dos brazos). Entonces, sería deseable que pudiéramos aprovechar los procesos que ya hemos definidos modificándolos mínimamente para adaptarlos a nuestros caso.

CSP provee un mecanismo para renombrar eventos, llamado *renombramiento funcional*, que es muy útil en casos como este. Primero se define una función que transforma cada nombre de evento que debe se renombrado en otro nombre y luego se aplica la función al proceso. En el caso de la celda de producción tenemos lo siguiente:

$$\mathcal{R} = \{(tobelt, tbr), (take, tr), (topress, tpr), (release, rr), (item, ir)\} \tag{18}$$

$$\mathcal{L} = \{(tobelt, tbl), (take, tl), (topress, tpl), (release, rl), (item, il)\} \tag{19}$$

$$ARMR = \mathcal{R}[DK\_ARM]$$

$$ARML = \mathcal{L}[DK\_ARM]$$

$$BELTR = \mathcal{R}[DK\_BELT]$$

$$BELTL = \mathcal{L}[DK\_BELT]$$

Por lo tanto la especificación completa y final del entorno es la siguiente:

$$\begin{aligned} &DK\_CELL = \\ &\quad DK\_SWITCH \,||\, DK\_PRESS \,||\, ARMR \,||\, ARML \,||\, BELTR \,||\, BELTL \end{aligned} \tag{20}$$

Algunas de las leyes del renombramiento funcional son las siguientes:

$$e \in \mathrm{dom}\,\mathcal{F} \Rightarrow \mathcal{F}[e \to P] = \mathcal{F}(e) \to \mathcal{F}[P] \tag{21}$$

$$e \notin \mathrm{dom}\,\mathcal{F} \Rightarrow \mathcal{F}[e \to P] = e \to \mathcal{F}[P] \tag{22}$$

$$\mathcal{F}[P \,\square\, Q] = \mathcal{F}[P] \,\square\, \mathcal{F}[Q] \tag{23}$$

$$\mathcal{F}[P \,||\, Q] = \mathcal{F}[P] \,||\, \mathcal{F}[Q] \tag{24}$$

En general el renombramiento funcional es distributivo respecto de los restantes operadores del álgebra.

Si bien el renombramiento funcional es adecuado para muchas situaciones, en algunas se torna muy engorroso o inaplicable. Por tal motivo, CSP ofrece el *renombramiento indexado*. Supongamos que en lugar de dos brazos el robot tiene diez. En tal caso deberíamos definir diez funciones de renombramiento lo cual no es muy práctico. Entonces usamos el renombramiento indexado así:

$$ROBOT = \|_{i=1}^{10} i : DK\_ARM \tag{25}$$

donde $i : DK\_ARM$ es un proceso para cada i de 1 a 10. Los eventos de cada $i : DK\_ARM$ son los mismos que los de DK_ARM excepto que se anteceden con el número natural correspondiente;

por ejemplo, $1.take, 2.take, \ldots, 10.take$. Notar que si $i \neq j$, $i : DK\_ARM$ y $j : DK\_ARM$ no tienen eventos en común.

Puede ocurrir que ni el renombramiento funcional ni el renombramiento indexado resuelvan el problema, pero ciertamente usándolos inteligentemente podremos resolver todas las situaciones. Por ejemplo, en (25) el renombramiento indexado renombra también los eventos *start* y *abort* presentes en *DK_ARM* cosa que no es deseable puesto que se supone que todos los brazos del robot se apagan y encienden con los mismos eventos. En este caso combinamos renombramiento funcional con indexado para obtener la especificación correcta.

$$\mathcal{H} = \{x \in [1,10] \bullet x.start \mapsto start\} \cup \{x \in [1,10] \bullet x.abort \mapsto abort\}$$
$$ROBOT = \mathcal{H}\left[\left\|_{i=1}^{10} i : DK\_ARM\right.\right]$$

7. Especificaciones no deterministas

El software de control debe llevar los dispositivos a un estado conocido cuando el sistema se enciende. Más concretamente esto significa llevar ambos brazos a sus respectivas cintas transportadoras y soltar cualquier cosa que ellos tengan. A nivel de la especificación es indiferente en qué orden se inicializan los brazos (primero el izquierdo y luego el derecho o viceversa). Dejamos que el programador determine cuál de los dos comportamientos es el que se implementará.

En CSP el no determinismo se introduce por medio del operador $\sqcap$ llamado *selección interna* (*internal choice*) o *selección no determinista* (más sobre no determinismo en la sección 9). Entonces, la inicialización del programa de control para la celda de producción queda especificada de esta forma:

$$SCELL = start \rightarrow INIT \,\nabla\, abort \rightarrow SCELL \tag{26}$$
$$INIT = INITL \sqcap INITR$$
$$INITL = \overline{tbl} \rightarrow \overline{rl} \rightarrow \overline{tbr} \rightarrow \overline{rr} \rightarrow WORK$$
$$INITR = \overline{tbr} \rightarrow \overline{rr} \rightarrow \overline{tbl} \rightarrow \overline{rl} \rightarrow WORK$$

donde *WORK* lo especificaremos en la siguiente sección. De esta forma, la decisión sobre ejecutar *INITL* o *INITR* es arbitraria. Precisamente ese es el significado de $P \sqcap Q$: el entorno no puede saber ni decidir si se ejecutará P o Q.

Como ya hemos analizado en otros formalismos, la introducción de no determinismo en una especificación es un recurso de abstracción conveniente pero que en última instancia debe ser eliminado para poder arribar a una implementación razonable. El no determinismo puede aparecer tanto en la descripción de DK como en S pero tiene significados muy diferentes:

- En DK se utiliza para capturar cierto desconocimiento que se tiene sobre el entorno o para indicar que este es autónomo respecto de la máquina; no tiene por qué desaparecer de DK.
- En S se utiliza como mecanismo de abstracción o para postergar alguna decisión de implementación; en algún momento debería desaparecer de S.

  Por otro lado, el no determinismo es inevitable en sistemas concurrentes donde hay alguna forma de arbitraje. Veremos una introducción a este tema en la sección 9.

Algunas de las leyes más importantes de $\sqcap$ son las siguientes.

$$e \to P \sqcap e \to Q = e \to (P \sqcap Q) \tag{27}$$
$$e \to P \,\Box\, e \to Q = e \to P \sqcap e \to Q \tag{28}$$
$$P \sqcap P = P \tag{29}$$
$$P \sqcap Q = Q \sqcap P \tag{30}$$
$$P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R \tag{31}$$
$$P \parallel (Q \sqcap R) = (P \parallel Q) \sqcap (P \parallel R) \tag{32}$$
$$P \sqcap (Q \,\Box\, R) = (P \sqcap Q) \,\Box\, (P \sqcap R) \tag{33}$$
$$P \,\Box\, (Q \sqcap R) = (P \,\Box\, Q) \sqcap (P \,\Box\, R) \tag{34}$$
$$\mathcal{F}[P \sqcap Q] = \mathcal{F}[P] \sqcap \mathcal{F}[Q] \tag{35}$$
$$P \,\triangledown\, (Q \sqcap R) = (P \,\triangledown\, Q) \sqcap (P \,\triangledown\, R) \tag{36}$$

8. Concurrencia: modelo sincrónico

Ya hemos reunido todos los elementos como para terminar la especificación del sistema de control de la celda de producción. A continuación definimos el proceso *WORK* que nos había quedado pendiente en (26).

$$WORK = WL \parallel WR \parallel PRESS \tag{37}$$
$$WL = il \to \overline{tl} \to \overline{tpl} \to irl \to \overline{rl} \to lar \to \overline{tbl} \to WL \tag{38}$$
$$WR = ir \to \overline{tr} \to \overline{tpr} \to irr \to \overline{rr} \to rar \to \overline{tbr} \to WR \tag{39}$$
$$PRESS = free \to (irr \to rar \to PRESSING \mid irl \to lar \to PRESSING) \tag{40}$$
$$PRESSING = \overline{press} \to stoppress \to \overline{remove} \to PRESS \tag{41}$$
$$SYSTEM = DK\_CELL \parallel SCELL \tag{42}$$

Esta especificación muestra que el software de control debe coordinar acciones con las cintas transportadoras, los brazos del robot y la prensa. ¿Qué significado tiene el hecho de que, por ejemplo, *il* sea un evento tanto de *BELTL* como de *WL*? En Statecharts esta relación entre *BELTL* y *WL* estaba regida por el modelo *broadcast*: *BELTL* anunciaría *il* y *WL* transicionaría si el evento llegaba en el momento preciso, caso contrario se perdería. CSP, por el contrario, se rige por un modelo *sincrónico*: si dos procesos comparten un evento, ambos deben estar dispuestos a consumirlo al mismo tiempo, caso contrario uno de ellos deberá esperar a que el otro proceso esté en condiciones de hacerlo; es decir, los eventos no se pierden, sino que los procesos se bloquean. Las leyes algebraicas que describen esta semántica son las siguientes.

$$e \in \alpha P \wedge f \in \alpha Q \Rightarrow f \to P \parallel e \to Q = STOP \qquad \text{(Deadlock)}$$
$$e \to P \parallel e \to Q = e \to (P \parallel Q) \qquad \text{(Sincronization)}$$
$$e \in \alpha P \wedge f \notin \alpha Q \Rightarrow f \to P \parallel e \to Q = f \to (P \parallel e \to Q) \qquad \text{(IEOF)}$$

STOP es un proceso estándar de CSP que indica terminación errónea o anormal (ver más detalles en la sección 13). En este caso *STOP* está señalando que hay abrazo mortal (*deadlock*), es

decir ninguno de los dos procesos puede avanzar. Precisamente, siendo e y f eventos compartidos, el modelo sincrónico indica que cualquiera de los procesos debe esperar a que el otro esté dispuesto a comunicar el evento que el primero está esperando, pero como ambos están en la misma situación el progreso es imposible.

La segunda ley, Sincronization, muestra otro aspecto del modelo sincrónico adoptado por CSP: si dos procesos deben interactuar a través de un evento en común, entonces ambos consumen el evento al mismo tiempo. La tercera ley, llamada "los eventos independientes se consumen primero" (abreviada IEOF, por *independent events occur first*), representa la tercera posibilidad: los eventos que no son compartidos tienen prioridad sobre los eventos comunes pues de lo contrario la composición paralela no podría progresar, ya que para consumir el evento común ambos procesos deben estar en condiciones de hacerlo y uno de ellos no lo está.

Estas tres leyes más Interleaving constituyen el núcleo de la semántica de la composición paralela en CSP. Si bien puede parecer un detalle menor queremos remarcar que estas cuatro leyes pueden aplicarse únicamente si los procesos a ambos lados de $\|$ tienen la forma $evento \rightarrow PROCESO$. En otras palabras, no podemos aplicar ninguna de las leyes a la composición paralela $a \rightarrow P \,\|\, (a \rightarrow Q \,\square\, b \rightarrow R)$ sin antes transformarla en $(a \rightarrow P \,\|\, a \rightarrow Q) \,\square\, (a \rightarrow P \,\|\, b \rightarrow R)$ aplicando (14).

Analicemos ahora, brevemente, la especificación *WORK*. Intuitivamente el software de control se implementará como un ciclo (posiblemente infinito) que escucha señales provenientes de los dispositivos de entrada y envía señales a los dispositivos de salida. *WORK* nos indica que podemos dividir este ciclo en tres que en algún momento se comunican entre sí. Uno de los ciclos esperará señales provenientes de la prensa (*PRESS*) y le comunicará a ésta cuando prensar y retirar la pieza. Los otros dos harán lo propio con cada una de las cintas transportadoras y los brazos del robot (*WL* y *WR*). Si, por ejemplo, se recibe una señal proveniente de la cinta izquierda, el ciclo de control correspondiente (*WL*) ordenará al brazo izquierdo que se mueva según lo especificado. Sin embargo, ambos brazos deberán sincronizar con la prensa para soltar la pieza en exclusión mutua. Esta sincronización se da si el ciclo que interactúa con la prensa está en el estado que representa que la prensa está libre, en cuyo caso espera a que uno de los brazos esté dispuesto a soltar su pieza. *PRESS* especifica que sea cual fuere el brazo que suelta la pieza, el otro deberá esperar pues tanto *irl* como *irr* son eventos compartidos con otros procesos. Se supone que si ambos brazos estaban en condiciones de soltar sus piezas y uno de ellos gana, el otro tendrá más oportunidades de soltarla una vez que la prensa se haya liberado nuevamente puesto que el primero estará en camino de tomar otra pieza. Los eventos *lar* y *rar* son necesarios para que el brazo que gana la prensa suelte su pieza antes de que *PRESSING* le ordene a la prensa prensar.

Por otro lado, el modelo sincrónico implica otra propiedad de la especificación. *il* representa la señal enviada por el sensor de la cinta transportadora izquierda cada vez que se detecta una pieza. Este evento lo comparten *BELTL* (que describe al entorno) y *WL* (que describe al software), por lo que ambos están obligados a sincronizar en él. Si bien *BELTL* parece poder recibir las señales con una velocidad arbitraria, no parece ser el caso de *WL* el cual debe transicionar varias veces e incluso, tal vez, esperar su turno en la prensa para volver a estar en condiciones de recibir una señal *il*. Entonces, ¿qué ocurre si las piezas viajan muy rápido sobre la cinta? El sensor detectará todas y cada una de las piezas pero llegará un momento en el cual cuando quiera emitir una de esas señales deberá bloquearse hasta que *WL* regrese a su punto inicial, puesto que la señal es un evento compartido. En otras palabras, el software de control no puede controlar qué tan rápido o qué tantas piezas viajan sobre las cintas, lo único que puede hacer

es capturar tantas señales como le sea posible. Claramente, el modelo está diciendo que si las piezas viajan demasiado rápido el robot no podrá con todas y algunas se perderán. Incluso podríamos pensar que el sensor o el mismo software de control guardan todas las señales en memoria y las van consumiendo a medida que pueden. Pero, ¿no está relacionado el momento en que se emite la señal con la capacidad del brazo para encontrar y tomar la pieza? ¿De qué sirve que el software de control le ordene al brazo tomar una pieza cuya señal se produjo varias unidades de tiempo antes? Lo más probable es que no sirva de mucho y que la solución sea que el software de control pueda influir en la velocidad de la cinta o que por cuestiones del dominio de aplicación se sepa que la relación entre velocidad de la cinta y ubicación de las piezas es adecuada para la velocidad del robot y de la prensa. Si se da el primer caso, en efecto, deberíamos modificar el modelo.

Nos interesa ahora analizar si *SYSTEM* es en realidad un proceso secuencial. Dado que es un proceso muy complejo y largo, que nos llevaría mucho tiempo analizar, optamos por analizar una parte más manejable pero representativa: $BELT1l \,||\, WL$, donde $BELT1l = \mathcal{L}[BELT1]$ siendo $\mathcal{L}$ la función de renombramiento definida en (19). En primer término observemos que $\alpha BELT1l = \{il\}$ y $\alpha WL = \{il, tl, tpl, irl, rl, lar, tbl\}$, es decir que tienen eventos en común, por lo que deberemos valernos de las leyes Deadlock, Sincronization y IEOF para analizar esta composición paralela.

Teorema 3. *El proceso $P = BELT1l \,||\, WL$ es un proceso secuencial.*

Demostración. Comenzamos por expandir P:

$$P = il \to BELT1l \,||\, il \to tl \to tpl \to irl \to rl \to lar \to tbl \to WL \tag{T3.a}$$

Luego aplicamos Sincronization:

$$P = il \to (BELT1l \,||\, tl \to tpl \to irl \to rl \to lar \to tbl \to WL) \tag{T3.b}$$

Ahora la única ley que podemos aplicar es IEOF pues el primer evento de $BELT1l$ es il, que es compartido, y el primero del otro término es tl que no lo es.

$$P = il \to tl \to (BELT1l \,||\, tpl \to irl \to rl \to lar \to tbl \to WL) \tag{T3.c}$$

Con el mismo razonamiento aplicamos IEOF varias veces hasta obtener:

$$P = il \to tl \to tpl \to irl \to rl \to lar \to tbl \to (BELT1l \,||\, WL) \tag{T3.d}$$

Pero $BELT1l \,||\, WL$ es la definición de P, por lo que demostramos el teorema con una simple sustitución:

$$P = il \to tl \to tpl \to irl \to rl \to lar \to tbl \to P \tag{T3.e}$$

□

Ejercicios

Ejercicio 2. *Elimine los eventos lar y rar de las ecuaciones 38, 39 y 40 y determine qué situación anómala puede darse. Justifique formalmente su respuesta.*

Ejercicio 3. *Modele las piezas sobre la cintas y su relación con las señales del sensor. ¿Necesita más designaciones? ¿Qué dice su modelo en relación con la separación entre las piezas? ¿Cómo calificaría esta parte del modelo: hipótesis, comportamiento natural o de otra forma?*

9. Concurrencia, no determinismo y ocultación de eventos

En la sección anterior mencionamos que el no determinismo es inevitable en cualquier teoría de la concurrencia en la cual haya alguna forma de arbitraje. En esta sección estudiaremos con algún detalle este tema.

El arbitraje aparece en una composición concurrente de la siguiente forma:

$$A = P \parallel (a \to Q \mid b \to R) \tag{43}$$

donde $\{a, b\} \cap \alpha P \neq \emptyset$. En este caso, el proceso a la derecha arbitra entre a y b. Es decir la ejecución de A depende del camino elegido por el proceso derecho.

El no determinismo aparece cuando en una composición concurrente como (43) se *ocultan* eventos. En CSP se pueden ocultar eventos de un proceso por medio del operador de ocultación: $P \setminus X$, donde $X \subseteq \alpha P$. Por ejemplo, si $P = a \to b \to STOP$, $P \setminus \{b\} = a \to STOP$. El operador $\setminus$ tiene más prioridad que $\Box$, $\sqcap$, ∇ y $\parallel$, o sea que $a \to P \setminus X \Box b \to Q \setminus Y$ es igual a $((a \to P) \setminus X) \Box ((b \to Q) \setminus Y)$.

Pero, ¿por qué habría que ocultar eventos en un proceso? Si en (43) a es un evento *interno*, es decir, no es visible desde el entorno entonces habría que ocultarlo para obtener el proceso que realmente percibe el entorno. En otras palabras, para el entorno, A se comporta en realidad como $A \setminus \{a\}$ simplemente porque el entorno no sabe (o no debería saber) que A es una composición concurrente que sincroniza en a el cual, a su vez, es un evento que el entorno ni siquiera sabe que existe.

CSP no ofrece una ley para calcular $A \setminus \{a\}$ cuando $a \in \alpha P$. En ese caso la única posibilidad que nos queda para calcular $A \setminus \{a\}$ es encontrar el proceso secuencial equivalente a A, digamos A', y luego calcular $A' \setminus \{a\}$. En general, cuando seguimos ese camino llegamos a expresiones como la siguiente (más adelante veremos un ejemplo concreto):

$$(a \to Q \mid b \to R) \setminus \{a\}$$

Según las leyes de CSP tenemos lo siguiente:

$$(a \to Q \mid b \to R) \setminus \{a\} = Q \setminus \{a\} \sqcap (Q \setminus \{a\} \Box b \to (R \setminus \{a\})) \tag{44}$$

donde vemos que aparece el no determinismo. La explicación o justificación es la siguiente. La selección interna aparece porque el entorno pierde el control del proceso al no poder controlarlo a través de a. De aquí tenemos los dos comportamientos básicos: $Q \setminus \{a\}$, por un lado; y $Q \setminus \{a\} \Box b \to (R \setminus \{a\})$, por el otro. $Q \setminus \{a\}$ claramente corresponde a "distribuir" la ocultación a ambos lados de $|$. El lado derecho de $\sqcap$ se explica de la siguiente forma: el entorno aun puede controlar Q a través de su primer evento en cuyo caso el entorno tiene la posibilidad de elegir el primer evento de $Q \setminus \{a\}$ o b. Notar que si el primer evento de Q es un evento interno entonces se *puede* ejecutar el proceso a la izquierda de $\sqcap$. Es decir que $Q \setminus \{a\}$ a la izquierda de $\sqcap$ cubre el caso en el cual el primer evento es interno; mientras que $Q \setminus \{a\}$ a la derecha de $\sqcap$ cubre el caso en el cual el primer evento tiene que sincronizar con el entorno. Roscoe lo explica del siguiente modo [6, página 80]: al elegir entre eventos visibles y ocultos, *podemos* tener la posibilidad de elegir uno de los visibles, pero si esperamos lo suficiente uno de los ocultos *debe* darse.

La igualdad (44) es un caso particular de una ley general de CSP que permite "distribuir" la ocultación en un proceso dominado por $\Box$. Esta ley es bastante complicada pues hay que decidir qué ocurre cuando se oculta uno o varios de los primeros eventos. Mantener consistente esta ley

con el resto de CSP no es tan simple como podría parecer a primera vista (ver más en [6, páginas 77 – 81]). Para establecer la ley necesitamos definir algunos conjuntos. Sea A un subconjunto finito de $\mathbb{N}$, sea B un subconjunto de A, sea E un conjunto de eventos de un cierto alfabeto tal que $E \cap \{a_i\}_{i \in A} = \emptyset$ y sea $X = \{a_i\}_{i \in B} \cup E$, entonces la ley que necesitamos es la siguiente:

$$(\underset{i \in A}{\Box} a_i \rightarrow P_i) \setminus X = \begin{cases} \Box_{i \in A} a_i \rightarrow (P_i \setminus X) & B = \emptyset \\ Q \sqcap (Q \Box (\underset{i \in A \setminus B}{\Box} a_i \rightarrow (P_i \setminus X))) & B \neq \emptyset \end{cases} \tag{45}$$

donde $Q = \sqcap_{i \in B} P_i \setminus X$. El caso $B = \emptyset$ es bastante obvio y simple pues no hay que ocultar el primer evento de ninguna de las alternativas de la selección externa. El caso $B \neq \emptyset$ es la generalización de (44).

Otras leyes de la ocultación son las siguientes:

$$(P \sqcap Q) \setminus X = P \setminus X \sqcap Q \setminus X \tag{46}$$

$$(P \parallel Q) \setminus X = P \setminus X \parallel Q \setminus X, \text{ si } (\alpha P \cap \alpha Q) \cap X = \emptyset \tag{47}$$

$$a \rightarrow P \setminus X = \begin{cases} P \setminus X & a \in X \\ a \rightarrow (P \setminus X) & a \notin X \end{cases} \tag{48}$$

Ahora vamos a aplicar todo lo que hemos visto en esta sección a *SCELL*. Los eventos *irl*, *irr*, *lar* y *rar* usados en (38), (39) y (40) son eventos *internos*, es decir, no son visibles desde el entorno de *SCELL* (no tienen designación y no tienen por qué tenerla). El entorno no sabe o no debería saber que *SCELL* es la composición concurrente de tres procesos. En consecuencia, para que *SCELL* represente lo que el entorno percibe del software de control los eventos *irl*, *irr*, *lar* y *rar* no deberían aparecer en la especificación. Por lo tanto, *DK_CELL* (el entorno de *SCELL*) percibe al software de control en realidad como:

$$SCELL \setminus \{irl, irr, lar, rar\} \tag{49}$$

Entonces, si llamamos I al conjunto $\{irl, irr, lar, rar\}$ y aplicamos las leyes anteriores sobre (49) tenemos:

$$SCELL \setminus I = start \rightarrow (INIT \setminus I) \triangledown abort \rightarrow (SCELL \setminus I)$$

$$INIT \setminus I = INITL \setminus I \sqcap INITR \setminus I$$

$$INITL \setminus I = \overline{tbl} \rightarrow \overline{rl} \rightarrow \overline{tbr} \rightarrow \overline{rr} \rightarrow (WORK \setminus I)$$

$$INITR \setminus I = \overline{tbr} \rightarrow \overline{rr} \rightarrow \overline{tbl} \rightarrow \overline{rl} \rightarrow (WORK \setminus I)$$

Como *WORK* es una composición concurrente e $I \cap \alpha WORK \neq \emptyset$, no podemos aplicar (47). Entonces, para calcular $WORK \setminus I$ tenemos que calcular el proceso secuencial equivalente a *WORK* y luego aplicar la ocultación. Como *WORK* es un proceso muy largo, analizaremos la siguiente versión simplificada que es estructuralmente similar:

$$W = W1 \parallel W2 \parallel P \tag{50}$$

$$W1 = a \rightarrow e_1 \rightarrow W1$$

$$W2 = b \rightarrow e_2 \rightarrow W2$$

$$P = a \rightarrow P \Box b \rightarrow P \tag{51}$$

donde a y b son eventos internos y e_1 y e_2 son eventos visibles desde el entorno. El proceso secuencial equivalente a W es (ver ejercicio 4):

$$W = a \rightarrow A \,\square\, b \rightarrow B \tag{52}$$
$$A = e_1 \rightarrow W \mid b \rightarrow C \tag{53}$$
$$B = e_2 \rightarrow W \mid a \rightarrow C \tag{54}$$
$$C = e_1 \rightarrow B \mid e_2 \rightarrow A \tag{55}$$

Ahora podemos aplicar (45) a (52):

$$W \setminus \{a,b\} = A \setminus \{a,b\} \sqcap B \setminus \{a,b\}$$

pues debemos aplicar el caso $A \cap X \neq \emptyset$. Claramente, la ocultación de eventos introduce no determinismo en el contexto de la composición secuencial. Esto es razonable pues desde el entorno no es posible saber cuándo W sincroniza con e_1 y cuando con e_2 porque eso depende de que haya optado por a o b, eventos que ni siquiera son visibles desde el entorno; para el entorno esa decisión es no determinista.

Por lo tanto, dado que siempre es conveniente ocultar las comunicaciones internas de una composición secuencial, es muy probable que se introduzca no determinismo donde no lo había, lo que torna al no determinismo en un elemento inherente a la concurrencia.

Ejercicios

Ejercicio 4. *Probar que el proceso W definido en* (50) *es equivalente al proceso definido en* (52).

Ayuda:

1. Usar la táctica de nombrar apropiadamente procesos como se hizo en la demostración del Teorema 2, siguiendo la estructura del resultado que se busca.
2. Expandir y distribuir P sobre $W1$ y $W2$.
3. Aplicar leyes de parlelismo apropiadamente a cada lado de $\square$.
4. Nombrar apropiadamente los procesos que se obtienen.
5. Como lo que se obtiene es simétrico, mencionaremos cómo seguir con uno de ellos.
6. Ahora primero aplicar leyes de paralelismo y luego distribuir P.
7. Observar que una de las "ramas" obtenidas vuelve a W.
8. En la otra expandir y distribuir P; una de las ramas queda en abrazo mortal, la otra es la que sirve.
9. Eliminar el abrazo mortal aplicando (81).
10. En la rama sobreviviente aplicar leyes de paralelismo y renombrar procesos apropiadamente.
11. El resto consideramos que es relativamente simple.

Ejercicio 5. *Determine si* (45) *es más general que* (48). *Justifique su respuesta.*

Ejercicio 6. *Considere* (45). *Suponga que $A = X$. Pruebe que el proceso se torna completamente no determinista.*

Ejercicio 7. *Pruebe que $(P \sqcap STOP) \,\square\, Q = (P \,\square\, Q) \sqcap Q$. Reescriba* (45) *aplicando esa igualdad.*

Ejercicio 8. *Considere la definición* (52) *del proceso W. Reescriba $W \setminus \{a,b\}$ de manera tal que no aparezca la ocultación de eventos.*

10. Especificación de requisitos temporales

En esta sección mostraremos un método para especificar requisitos temporales en CSP. El método requiere introducir varios conceptos más o menos avanzados del formalismo. Procederemos de la siguiente forma: en esta sección introduciremos el método sin explicar en detalle los nuevos conceptos y en las secciones siguientes explicaremos dichos conceptos más profundamente.

Existe una variante de CSP llamada Timed CSP que, obviamente, agrega el concepto de tiempo a CSP [5, capítulo 4]. En Timed CSP el tiempo es continuo y denso. La semántica de Timed CSP es muy elegante, consistente y reduce a CSP cuando no se utiliza el tiempo pero, justamente, es una semántica diferente a la de CSP que en muchos casos da lugar a modelos notablemente diferentes a los que se dan en CSP. Por este motivo no abordaremos Timed CSP en este apunte sino que optamos por un método alternativo para especificar requisitos temporales.

Nuestro método se basa en [6, capítulo 14] y en la forma en que Statecharts maneja el tiempo. A diferencia de Timed CSP nosotros utilizaremos únicamente tiempo discreto (lo que obviamente es una desventaja frente a Timed CSP, aunque aceptable), pero no tendremos que modificar la semántica puesto que no agregamos nada al lenguaje. Frente a Timed CSP nuestro método tiene las siguientes ventajas:

- Brinda una flexibilidad importante sobre cómo modelar y razonar sobre sistemas con requisitos temporales.

  Nuestra experiencia nos indica que con este método se pueden modelar de manera más o menos sencilla una gran variedad de problemas .
- El ingeniero puede entender las especificaciones temporizadas con solo comprender CSP más unas pocas designaciones y dos procesos.
- Si no hay requisitos temporales los modelos no tienen nada diferente a aquellos escritos en CSP "puro"; el método se aplica cuando es necesario.

Comenzamos por definir un reloj discreto universal que representa el avance del tiempo real.

$$DKRT = \overline{tick} \rightarrow DKRT \tag{56}$$

donde *tick* tiene el siguiente significado:

El tiempo del universo avanza una unidad $\approx$ *tick*

Luego, definimos un *timer* que será la forma en que mediremos el tiempo en nuestros modelos; es decir, no accederemos al tiempo real directamente (a menos que sea muy necesario o simplifique considerablemente el modelo).

$$\begin{aligned} TIMER = &(start?n \rightarrow (TIMER(n) \lceil n > 0 \rceil TIMER) \nabla stop \rightarrow TIMER) \\ &\mid tick \rightarrow TIMER \end{aligned} \tag{57}$$

$$TIMER(0) = \overline{timeout} \rightarrow TIMER \tag{58}$$

$$TIMER(n+1) = tick \rightarrow TIMER(n) \tag{59}$$

donde $P \lceil b \rceil Q =$ **if** b **then** P **else** Q. Los eventos *start*?*n*, *stop* y *timeout* tienen el siguiente significado:

Se configura e inicia el *timer* para que cuente n unidades de tiempo $\approx start?n$

Se detiene el *timer* $\approx stop$

El *timer* alcanza la cota de tiempo estipulada $\approx timeout$

Intuitiva e informalmente la especificación del *timer* dice lo siguiente: el *timer* percibe el paso del tiempo pasivamente hasta tanto sea configurado e iniciado para que cuente n unidades de tiempo y, una vez iniciado, espera a que transcurra una unidad de tiempo para decrementar un contador interno hasta que alcance cero, momento en el cual comunica al entorno que el tiempo estipulado ha transcurrido. En virtud de la interrupción (∇) la cuenta regresiva puede detenerse en cualquier momento. La alternativa $tick \rightarrow TIMER$ en (57) es necesaria por dos motivos: (a) *tick* es compartido entre *TIMER* y *DKRT* por lo que *DKRT* no podría avanzar si el *timer* no es iniciado[1]; (b) al no resolver (a) se complican las especificaciones en las cuales se requieren varios *timers* funcionando solapadamente.

Finalmente, definimos un par de procesos paralelos pues es la forma más usual de utilizar el método. El primero se usa cuando solo se necesita un *timer* y el segundo cuando se necesita más de uno.

$$RTR = DKRT \parallel TIMER \tag{60}$$

$$RTR(n) = DKRT \parallel \mathcal{T} \left[\parallel_{i=1}^{n} i : TIMER \right] \tag{61}$$

$\mathcal{T} = \{i \in [1,n] \bullet i.tick \mapsto tick\}$, permite que los *timers* sincronicen con el tiempo real. Tener en cuenta que en $RTR(n)$ los eventos para iniciar y detener cada *timer* y el evento producido al llegar al límite de tiempo son distintos entre sí, aunque el paso del tiempo es el mismo para todos. (De no haber incluido la rama $tick \rightarrow TIMER$ en (57), se hubiera complicado mucho escribir $RTR(n)$.)

En las secciones que siguen explicaremos las nuevas construcciones del lenguaje que usamos para definir *TIMER* con mayor detalle.

Ejercicios

Ejercicio 9. *¿Podría perderse un tick en el momento en que el timer es configurado? ¿Cuál es la hipótesis básica sobre la duración de un evento?*

Ejercicio 10. *Sin usar renombramiento funcional en (61), ¿cómo podría lograr que los timers sincronicen con el tiempo real?*

10.1. Algunos requisitos temporales para la celda de producción

Veremos algunas aplicaciones del método para especificar requisitos temporales proponiendo ciertos requisitos temporales sobre el problema de la celda de producción.

[1] Aunque esto no es realmente un problema porque si el *timer* no fue iniciado es porque no es necesario contar el paso del tiempo.

Tiempo de prensado. Digamos que la prensa demora 5 unidades de tiempo desde que remueve la pieza que acaba de prensar y está lista para volver a prensar. Para incluir esta propiedad en la especificación del conocimiento de dominio de la prensa modificamos el proceso *PRESS*2 definido en (1), aplicamos renombramiento funcional sobre *RTR* (pues de lo contrario habría conflicto de nombres entre el evento *start* usado en *DK_PRESS* y el definido en *TIMER*), y componemos en paralelo.

$$\mathcal{F} = \{m \in \mathbb{N} \bullet start.m \mapsto st.m\} \tag{62}$$

$$PRESS2 = press \rightarrow \overline{stoppress} \rightarrow remove \rightarrow st!5 \rightarrow timeout \rightarrow PRESS1 \tag{63}$$

$$DK\_PREES = (start \rightarrow PRESS1 \,\triangledown\, abort \rightarrow DK\_PRESS) \,||\, \mathcal{F}[RTR] \tag{64}$$

De esta forma, el evento *st*!5 en *PRESS*2 inicia el *timer* (pues, como veremos más adelante, *st*!5 y *st*?*n* deben sincronizar) lo que implica que el proceso de la prensa debe esperar 5 unidades de tiempo hasta que *TIMER* lance *timeout* para que *PRESS*2 pueda continuar y volver a *PRESS*1.

Detener las cintas. Supongamos que el software de control puede detener las cintas transportadoras izquierda y derecha mediante los eventos *sbl* y *sbr*, respectivamente. Digamos que el software debe detener la cinta si no se detecta una nueva pieza a los 3 segundos de haber detectado la anterior, y que en tal caso debe volver a encenderla, con los eventos *rbl* y *rbr*, cuando el sensor detecte una nueva pieza (tener en cuenta que el sensor puede detectar objetos aunque la cinta esté apagada). Como son dos cintas necesitamos dos *timers* por lo que recurrimos a *RTR*(2). Entonces modificamos *WL* definido en (38), como se muestra a continuación (la modificación de *WR* queda para el lector).

$$WL = 1.start!3 \rightarrow (il \rightarrow 1.stop \rightarrow WLON \mid 1.timeout \rightarrow WLOFF) \tag{65}$$

$$WLON = \overline{tl} \rightarrow \overline{tpl} \rightarrow irl \rightarrow \overline{rl} \rightarrow lar \rightarrow \overline{tbl} \rightarrow WL \tag{66}$$

$$WLOFF = \overline{sbl} \rightarrow il \rightarrow \overline{rbl} \rightarrow WLON \tag{67}$$

Es decir, el software de control inicia un *timer* para que "suene" a los tres segundos, luego de lo cual permite que el entorno (el sensor de la cinta o el *timer*) comunique la llegada de una nueva pieza o el fin del conteo del tiempo, el evento que llegue primero decide el camino a seguir por el software. Es importante remarcar la necesidad de detener el *timer* cuando no se da *timeout* pues de lo contrario el siguiente 1.*start*!3 no necesariamente sincronizará con 1.*start*?*n* en el momento apropiado.

Finalmente, incluimos los *timers* en la especificación:

$$SCELL = (start \rightarrow INIT \,\triangledown\, abort \rightarrow SCELL) \,||\, RTR(2) \tag{68}$$

Ejercicios

Ejercicio 11. *Modificar el proceso WR definido en* (39) *para modelar el requisito temporal para detener y reiniciar la cinta derecha.*

Ejercicio 12. *Analizar formalmente qué puede implicar no incluir* 1.*stop en* (65) *cuando il aparece antes de las 3 unidades de tiempo.*

10.2. Abreviaturas comunes para especificar requisitos temporales

En esta sección definiremos algunas abreviaturas que son de uso común cuando se especifican requisitos temporales. Todas están definidas en base a *TIMER*, *DKRT*, y *RTR* por lo que no agregan nada nuevo. Los operadores así definidos fueron tomados de Timed CSP pero redefinidos según nuestro método. Pueden usar estos operadores a la hora de resolver los ejercicios.

Comenzamos con el operador denominado *WAIT* que representa un espacio de tiempo en el que no se producen eventos. Para definirlo hacemos uso de otro proceso estándar de CSP llamado *SKIP* que es como *STOP* pero indica terminación exitosa (lo veremos con más detalle en la sección 13).

$$WAIT(t) = start!t \rightarrow timeout \rightarrow SKIP \qquad \text{(Wait)}$$

Como *WAIT* es un proceso que termina, usualmente se lo utiliza junto a la composición secuencial (ver también sección 13). Por ejemplo podríamos reescribir (63) de la siguiente forma:

$$PRESS2 = press \rightarrow \overline{stoppress} \rightarrow remove \rightarrow WAIT(5); PRESS1 \qquad (69)$$

Cabe remarcar que para que *WAIT* realmente funcione, el proceso en el cual participa (por ejemplo *PRESS*2) en algún momento tiene que ponerse en paralelo con *RTR* o algo similar (pues de lo contrario será imposible iniciar un *timer* y esperar el transcurso del tiempo). En el caso de *PRESS*2 esto se cumple pues forma parte de *DK_PRESS* el cual contiene a *RTR* (ver (64))

También podemos definir una versión temporizada de $\rightarrow$, llamada *prefijación temporizada* (*delayed prefix*). Notar que es diferente a *WAIT* pues luego del tiempo que demora la transición se continúa con cualquier proceso, y no sólo con *SKIP*.

$$e \xrightarrow{t} P = e \rightarrow start!t \rightarrow timeout \rightarrow P \qquad \text{(Delayed prefix)}$$

Podríamos volver a reescribir (63) usando Delayed prefix:

$$PRESS2 = press \rightarrow \overline{stoppress} \rightarrow remove \xrightarrow{5} PRESS1 \qquad (70)$$

Finalmente es posible definir el operador llamado Timeout, aunque la versión más general plantea un problema. Timeout espera el primer evento del proceso a su izquierda hasta t unidades de tiempo, y si tal evento no aparece se comportará como el proceso de la derecha.

$$P \overset{t}{\triangleright} Q = start!t \rightarrow (P \,\Box\, timeout \rightarrow Q) \qquad \text{(Timeout)}$$

El problema que tiene esta definición es que si el entorno ofrece el primer evento de P antes de que transcurra el tiempo t, el *timer* no se detendrá inmediatamente sino que seguirá corriendo lo que puede causar problemas según sea la definición de P o Q. No hay forma de solucionar este problema con nuestro método a menos que debilitemos un poco la definición:

$$e \rightarrow P \overset{t}{\triangleright} Q = start!t \rightarrow (e \rightarrow \overline{stop} \rightarrow P \,\Box\, timeout \rightarrow Q) \qquad \text{(WTimeout)}$$

Tengan en cuenta que si varios de estos operadores se usan en la misma especificación, técnicamente, debería ser necesario renombran sus eventos (por ejemplo *start*, *timeout*, etc.) para

evitar comportamientos no deseados. Por ejemplo, si usamos varias veces el operador $\triangleright$ el evento *timeout* será el mismo en todos los procesos y por consiguiente deberían sincronizar lo que implica que los procesos se bloquearán en *timeout* hasta que se dé el último de ellos. En general este no será el comportamiento que queremos especificar. De todas formas, en lo que respecta a los ejercicios, asumiremos que este renombramiento se hace mágicamente de alguna forma.

Ejercicios

Ejercicio 13. *Utilice el operador* WTimeout *para especificar WL en* (65).

11. Comunicación de datos y eventos compuestos

En la definición de *TIMER* (57) utilizamos expresiones de la forma $c?x$ y en varias de las aplicaciones de *TIMER* (por ejemplo (63)) usamos expresiones de la forma $c!y$. En esta sección explicaremos el significado de estos conceptos.

Los eventos de la forma $c?x$ representan la recepción del dato x a través del *canal* c. En tanto que los eventos de la forma $c!x$ representan el envío del dato x a través del canal c. Las expresiones escritas a la derecha de ? y ! se llaman *parámetros* de entrada y salida, respectivamente. A las expresiones de la forma $c?x$ se las llama *eventos de entrada*, en tanto que las otras son *eventos de salida*. Los parámetros de entrada pueden ser usados para definir el comportamiento del proceso de allí en más, por ejemplo:

$$c?x \rightarrow P(x)$$

donde $P(x)$ es entonces un *proceso parametrizado* (ver sección 12). No tiene sentido hacer lo mismo con los parámetros de salida.

La ley fundamental que rige la comunicación de datos a través de canales es la siguiente:

$$c?x \rightarrow P(x) \parallel c!y \rightarrow Q = c!y \rightarrow (P(y) \parallel Q) \qquad \text{(Communication)}$$

Según Communication cuando un evento de entrada y uno de salida sincronizan, se convierten en un único evento de salida y el parámetro de entrada es sustituido por el de salida[2].

Las siguientes leyes de la comunicación a través de canales muestran que las expresiones a la derecha de ? y ! no forman parte del nombre del canal.

$$\mathcal{F}[c?x \rightarrow P(x)] = \mathcal{F}(c)?x \rightarrow \mathcal{F}[P(x)] \tag{71}$$

$$\mathcal{F}[c!x \rightarrow P] = \mathcal{F}(c)!x \rightarrow \mathcal{F}[P] \tag{72}$$

[2]En [5, página 15] los autores indican que los canales son unidireccionales y que un canal vincula exactamente dos procesos (o uno solo pero en ese caso se asume que el proceso faltante es el entorno), por lo que se requieren procesos multiplexadores para comunicaciones múltiples. Estas fueron las propiedades sobre canales que hemos enseñado en este curso en los años anteriores. Sin embargo, un análisis bastante cuidadoso de [6] nos ha convencido que no es necesario que se verifiquen esas propiedades para mantener CSP consistente. De hecho Roscoe en la página 27 de su libro, dice explícitamente que la comunicación de datos sobre un canal puede darse en múltiples direcciones a la vez. Al mismo tiempo no hemos podido encontrar una sola mención en el sentido de que un canal debe comunicar exactamente dos procesos, tanto que en la página 334 dice que en general varios procesos pueden compartir eventos del mismo canal. Consideramos que en general el libro de Roscoe es más riguroso que el de Hinchey y Jarvis. Por todas estas razones a partir de ahora seguiremos las ideas de Roscoe sobre canales. No obstante, nuestra experiencia nos indica que en la práctica es conveniente que se verifique las propiedades enunciadas por Hinchey y Jarvis pues de lo contrario se corre el riesgo de que las especificaciones se tornen confusas.

En realidad los símbolos ? y ! son *decoraciones* que por convención representan entrada o recepción y salida o envío, respectivamente. La forma básica de definir un canal es $c.x$, en cuyo caso hablamos de *eventos compuestos* más que de canales. Más aun, los eventos compuestos no tienen por qué tener un único parámetro y no tienen por qué ser del mismo tipo. CSP admite eventos de la forma $base.x_1 : T_1.x_2 : T_2. \ldots .x_n : T_n$ donde *base* es el nombre del canal y el resto son los parámetros cada uno de un tipo posiblemente diferente. Si el tipo de un parámetro se puede deducir del contexto entonces podemos obviar declararlo, como hicimos en (57) con el parámetro n que es de tipo $\mathbb{N}$. Los tipos pueden ser básicos o inductivos. Asumiremos la definición de los tipos más comunes (y sus combinaciones) tales como los naturales, enteros, listas, conjuntos, productos cartesianos, etc. Si el nombre base de un evento compuesto es c entonces todas las apariciones de c en una especificación deben tener el mismo número de parámetros y del mismo tipo, caso contrario hay un error sintáctico en la especificación.

En un evento compuesto cualquier punto puede reemplazarse por una decoración, por ejemplo $ch?w.x!y.z$, en cuyo caso todos los puntos a la derecha de una decoración se asume que están decorados con esa decoración. Es decir, el ejemplo es equivalente a $ch?w?x!y!z$. Si bien CSP admite eventos compuestos o canales muy complejos, raramente son necesarios aquellos con dos o más parámetros y aquellos en los que las decoraciones están mezcladas. Por lo general se usan eventos compuestos con no más de tres parámetros y canales que solo esperan o solo envían datos.

Pero si eliminamos las decoraciones, ¿por qué en un proceso se escribe $c.x$ y en el otro $c.y$? ¿Por qué no escribir lo mismo en ambos? ¿Por qué cambiar el nombre del parámetro? ¿Cuál es el sentido de la comunicación, entonces? ¿Cuál es la entrada y cuál la salida? Más importante aun, ¿cómo se generaliza la ley Communication? Las respuestas a todas estas preguntas yacen en la *concordancia de patrones* (*pattern-matching*). La generalización de la ley Communication, llamada "sincronización de eventos compuestos", abreviada CES, dice que dos procesos deben sincronizar en un evento compuesto común solo si hay concordancia de patrones entre los respectivos parámetros, y en tal caso los patrones son reemplazados por los valores, como en Communication. Los patrones son expresiones que cumplen ciertas reglas sintácticas que no detallaremos pues son semejantes a las de Haskell u otros lenguajes funcionales (ver la lista completa en [6, página 503]).

Formalmente CES se enuncia de la siguiente manera. Sean $\mathcal{E}^1 = \{e_1^1, \ldots, e_n^1\}$ y $\mathcal{E}^2 = \{e_1^2, \ldots, e_n^2\}$ dos conjuntos de expresiones tales que si e_i^j es un patrón, e_i^k, con $j \neq k$, es una expresión que concuerda con aquel, y sea $\mathcal{V} = \{v_1, \ldots, v_n\}$ un conjunto de expresiones tal que $\mathcal{V} \subseteq \mathcal{E}^1 \cup \mathcal{E}^2$ y si e_i^j es un patrón entonces $v_i = e_i^k$ con $j \neq k$. Entonces si existe un único $\mathcal{V}$ con esas propiedades:

$$c.e_1^1.e_2^1 \ldots e_n^1 \to P \parallel c.e_1^2.e_2^2 \ldots e_n^2 \to Q = c.v_1.v_2 \ldots v_n \to (P \parallel Q) \qquad \text{(CES)}$$

En general el conjunto $\mathcal{V}$ no es único para dos conjuntos de expresiones dados. Por ejemplo, asumiendo que todos los parámetros son de tipo $\mathbb{N}$, en $c.x.2 \to P \parallel c.y.z \to Q$, $\mathcal{V}$ podría ser $\{x, 2\}$ o $\{y, 2\}$ pues la expresión x concuerda con y y viceversa (en cambio 2 concuerda con z pero no al revés). Entonces, cuando existe ambigüedad en el valor de $\mathcal{V}$ se puede resolver reemplazando uno de los puntos a la izquierda del parámetro conflictivo por ? en uno de los procesos y por ! en el otro (tener en cuenta la convención sobre la sustitución de puntos por ? o ! enunciada más arriba). En el ejemplo anterior la ambigüedad se resuelve redefiniendo el proceso a $c!x.2 \to P \parallel c?y.z \to Q$ que resulta equivalente a $c.x.2 \to (P \parallel Q)$ por CES y la convención de sustitución. Por otro lado, si el proceso fuese $c.x.2 \to P \parallel c.y.3 \to Q$ no se puede aplicar CES

debido a que 2 no concuerda con 3 ni viceversa; en esos casos se considera que $c.x.2$ y $c.y.3$ no son el mismo evento.

En general en la práctica los eventos de la forma $c?x : T$ se utilizan para recibir cualquier valor del tipo T por lo que el parámetro x debería ser un patrón que concuerde con cualquier valor del tipo. Por ejemplo, podríamos haber definido el *timer* de la siguiente forma:

$$TIMER = start?n+1 \rightarrow TIMER(n+1) \,\nabla\, stop \rightarrow TIMER$$

pero quedaría sin especificar cómo debe comportarse *TIMER* si se lo intenta inicializar con un tiempo 0. La precondición de la definición (57) equivale a *true* en tanto que la de la anterior es $n > 0$. Como ya hemos visto en los capítulos anteriores es conveniente especificar operaciones totales, por lo que preferimos la primera definición de *TIMER*.

Si bien CSP admite eventos compuestos muy complejos y CES es muy general, normalmente en la práctica no se utiliza todo este poder expresivo que por otra parte es muy propenso a errores. En consecuencia sugerimos usar con mucho cuidado los eventos compuestos; en particular sugerimos usar las decoraciones cuando la intención de los parámetros sea comunicar datos provenientes del entorno o destinados a él.

12. Procesos parametrizados

Los *procesos parametrizados* son aquellos procesos cuya definición depende de uno o más parámetros. Los parámetros de un proceso parametrizado pueden escribirse como subíndices, superíndices o como "argumentos funcionales"; nosotros optamos por la última forma. Los eventos compuestos o canales y los procesos parametrizados están íntimamente relacionados pues los primeros sirven para definir a los segundos. Toda especificación CSP debe comenzar con un proceso no parametrizado.

Los procesos parametrizados se usan en dos partes tal y como los subprogramas de los lenguajes de programación imperativos. Hay *llamadas* a procesos parametrizados y cada proceso parametrizado tiene su *definición*. La diferencia está en que CSP existe transparencia referencial en tanto que en muchos lenguajes imperativos no existe. Un proceso parametrizado puede ser llamado desde cualquier otro proceso e incluso desde el cuerpo de la defínción de ese mismo proceso u otro. Para efectuar la llamada a un proceso parametrizado se debe dar el número y tipo de parámetros reales adecuados. El proceso *TIMER* se definió en (57) con una llamada a un proceso parametrizado con el mismo nombre (se puede aplicar polimorfismo para saber de cuál proceso se habla):

$$TIMER = (start?n \rightarrow TIMER(n) \lceil n > 0 \rceil TIMER) \,\nabla\, stop \rightarrow TIMER$$

Para definir un proceso parametrizado se procede como en los lenguajes funcionales: se deben dar definiciones suficientes como para cubrir todas las combinaciones *posibles* de los constructores de sus parámetros. Por ejemplo, si el proceso P depende de un único parámetro de tipo $\mathbb{N}$ entonces hay que dar dos definiciones de P pues los naturales tienen dos constructores:

$$P(0) = \ldots$$
$$P(n+1) = \ldots$$

o una que abarque todas las formas posibles de construir un número natural:

$$P(n) = \ldots$$

Si el proceso Q depende de un parámetro de tipo $\mathbb{N}$ y otro de tipo $List(\mathbb{N})$ entonces podrían ser necesarias:

$$Q(0, \langle\rangle) = \ldots$$
$$Q(0, x : xs) = \ldots$$
$$Q(n+1, \langle\rangle) = \ldots$$
$$Q(n+1, x : xs) = \ldots$$

Aunque no siempre serán necesarias todas las definiciones. Consideremos un *buffer* de números naturales cuya capacidad es definida desde el entorno y cuenta con un canal, *left*, por donde recibe los números y otro, *right* por donde los emite al entorno. La definición es la siguiente.

$$BUFFER = long?n + 1 \rightarrow B(n+1, \langle\rangle) \tag{73}$$
$$B(m+1, \langle\rangle) = left?n : \mathbb{N} \rightarrow B(m, \langle n\rangle) \tag{74}$$
$$B(m+1, s \frown \langle y\rangle) = \tag{75}$$
$$\quad left?n : \mathbb{N} \rightarrow B(m, \langle n\rangle \frown s \frown \langle y\rangle)$$
$$\quad \mid right!y \rightarrow B(m+2, s)$$
$$B(0, s \frown \langle y\rangle) = right!y \rightarrow B(1, s) \tag{76}$$

Aparentemente faltaría definir el caso $B(0, \langle\rangle)$. No obstante no es necesario hacerlo pues, por las propiedades del modelo, esa combinación de constructores para los parámetros de B no puede darse nunca ya que la capacidad máxima del *buffer* es siempre positiva.

La llamada a un proceso parametrizado tiene éxito si hay concordancia de patrones entre los parámetros reales y los parámetros formales de alguna de las definiciones del proceso. Si hay concordancia en más de una definición el resultado es no determinista.

Los procesos parametrizados son el mecanismo que provee CSP para modelar sistemas con estados complejos. En efecto, la interpretación de un proceso paramentrizado es que es una máquina de estados cuyos estados son todos los posibles valores de sus parámetros y la relación de transición está dada por la definición del proceso.

12.1. *Buffer* sin parámetros

En general los parámetros se pueden eliminar de la definición de un proceso. De hecho es "más CSP" especificar procesos no-parametrizados que procesos parametrizados. Por ejemplo, el siguiente proceso B es un *buffer* de dos o tres elementos (depende de cómo se lo mida) que recibe datos en su canal *left* y los emite en su canal *right*:

$$L = left?x : \mathbb{N} \rightarrow p_1!x \rightarrow L \tag{77}$$
$$P = p_1?x \rightarrow p_2!x \rightarrow P \tag{78}$$
$$R = p_2?x \rightarrow right!x \rightarrow R \tag{79}$$
$$B = L \,||\, P \,||\, R \tag{80}$$

donde p_i son todos eventos (canales) internos.

Suponiendo que recién iniciamos B, funciona de la siguiente forma. El productor sincroniza con $left?x$ y le pasa, digamos, 5. Entonces L transiciona a $p_1!x$ que sincroniza (inmediatamente) con $p_1?x$ de P puesto que este estaba en su estado inicial (dado que asumimos que el *buffer* recién se inicia). Lo mismo ocurre con p_2 lo que significa que x en $p_2?x$ vale 5. Cuando R transiciona a $right!x$ deberá esperar a que el consumidor sincronice con él; supongamos que esto no ocurre por un cierto tiempo. Al mismo tiempo, mientras se producen las transiciones internas, L ha vuelto a su estado inicial lo que le permite al productor enviar otro número, digamos 17. Este 17 recorre el mismo camino que el 5 solo que esta vez llega hasta $p_2!x$ en P dado que R no está en su estado inicial (porque está esperando que el consumidor sincronice en $right!x$). P queda bloqueado en $p_2!x$ hasta que R esté en condiciones de ejecutar $p_2?x$. En el ínterin, nuevamente, L volvió a su estado inicial lo que le permite al productor enviar otro número, 9. Ahora el 9 llega hasta $p_1!x$ puesto que P está tratando de enviar el 17 en $p_2!x$. Entonces el productor no puede enviar más datos porque L está esperando en $p_1!x$. O sea que ahora tenemos algo así como $\langle p_1!9, p_2!17, right!5\rangle$[3].

Cuando el consumidor finalmente sincroniza en *right*, en B se producen una serie de transiciones que "empujan" el 9 y el 17 hacia la "derecha" permitiendo que L vuelva a estar en condiciones de recibir otro número del productor.

Ejercicios

Ejercicio 14. *Redefina* (73) *de manera tal que esté especificado cómo debe comportarse si se lo inicializa con 0.*

Ejercicio 15. *Determine formalmente el comportamiento de la composición paralela de* (73) *con cada uno de los siguientes procesos:*

$$SETLENGTH1 = long!0 \rightarrow STOP$$
$$SETLENGTH2 = long!3 \rightarrow STOP$$

Ejercicio 16. *¿Es posible definir un* buffer *de capacidad n (variable) sin que uno de los parámetros del proceso mantenga la lista de elementos?*

13. *STOP*, *SKIP* y composición secuencial

En algunas ocasiones hemos mencionado dos de los procesos estándar de CSP, *STOP* y *SKIP*. En algún sentido estos procesos representan la unidad o el elemento neutro del álgebra de procesos (es decir son algunas de las constantes del álgebra), de aquí su importancia teórica y su relevancia práctica. *STOP* es el proceso más simple de CSP pues no hace nada. Se lo utiliza para indicar terminación anormal o errónea y en particular las situaciones de abrazo mortal (*deadlock*). *STOP* no tiene un alfabeto específico sino que cuando es necesario hay que indicarlo

[3]Decimos que el *buffer* tiene 2 o 3 posiciones porque en realidad el número almacenado en $right!5$ no está estrictamente en el *buffer* sino en su canal de salida.

explícitamente: $STOP_A$. Las leyes más importantes que verifica *STOP* son las siguientes:

$$STOP \,\square\, P = P \tag{81}$$
$$P \,||\, STOP_A = STOP \Leftrightarrow \alpha P \cap A \neq \emptyset \tag{82}$$
$$P \,||\, STOP_A = P \Leftrightarrow \alpha P \cap A = \emptyset \tag{83}$$

Notar que *STOP* "hereda" su alfabeto del proceso donde participa. Por ejemplo, en $a \rightarrow b \rightarrow STOP$ tenemos que $\alpha(STOP) = \{a, b\}$ (ver Sección 5.1), lo que se escribe como $STOP_{\{a,b\}}$.

SKIP es igual a $\surd \rightarrow STOP$ donde $\surd$ es un evento muy particular que indica terminación exitosa de un proceso. $\surd$ no puede pertenecer al alfabeto de ningún proceso, excepto *SKIP*; en otras palabras si se desea especificar que un proceso termina exitosamente hay que utilizar *SKIP*, por ejemplo: $P = e \rightarrow f \rightarrow SKIP \,|\, f \rightarrow e \rightarrow STOP$.

Cuando un proceso termina correctamente otro puede comenzar a ejecutarse lo que se logra mediante la *composición secuencial*: $P; Q$. En CSP la composición secuencial cumple un papel semejante a la concatenación de sentencias en los lenguajes de programación imperativos, tanto que se utiliza el mismo símbolo. Si bien los conceptos de terminación y composición secuencial parecen muy simples, no lo son tanto. Consideremos la siguiente proposición:

$$c?x \rightarrow P; Q = c?x \rightarrow (P; Q) \tag{♣}$$

¿Es efectivamente una igualdad? ¿Es ley? Pareciera serlo. Sin embargo, consideremos un caso particular muy simple:

$$c?x \rightarrow SKIP; out!x+1 \rightarrow STOP = c?x \rightarrow (SKIP; out!x+1 \rightarrow STOP) \tag{84}$$

¿Por qué es tan problemático? Es que llama la atención el parámetro x en dos procesos que están separados por la terminación del primero. ¿Puede subsistir el valor de x una vez que el proceso $c?x \rightarrow SKIP$ haya terminado? En el proceso a la derecha de (84) está bastante claro que ambas x refieren al mismo objeto, pero en el lado izquierdo requeriría que el valor de x sea recordado una vez finalizado $c?x \rightarrow SKIP$ y comunicado a $out!x+1 \rightarrow STOP$. ¿Es esto posible? ¿Es correcto?

La respuesta remite a cómo deben interpretarse los valores y los ámbitos de los parámetros en CSP. La cuestión importante a resolver es si CSP es un lenguaje imperativo, donde el valor de un parámetro puede ser modificado, o es declarativo, donde eso no es posible. CSP es un lenguaje declarativo por lo que una expresión como $c?x \rightarrow P$ crea un nuevo identificador llamado x que almacenará el mismo valor sólo durante la vida de P, pues una vez que este haya terminado se habrá abandonado el ámbito de x. Un parámetro toma su valor en el momento en que es declarado y lo mantiene en todo su ámbito y solo en él.

En conclusión, (♣) es ley sí y sólo sí x no es una variable libre en Q. Por consiguiente (84) no es una igualdad a menos que se renombre x en $out!x+1 \rightarrow STOP$.

A continuación enumeramos las leyes más importantes de *SKIP* y la composición secuencial,

dejando para un poco más adelante la relación entre *SKIP* y ||:

$$c?x \rightarrow P;\ Q = c?x \rightarrow (P;\ Q) \text{ si } x \text{ no está libre en } Q \tag{85}$$
$$(P \sqcap Q);\ R = (P;\ R) \sqcap (Q;\ R) \tag{86}$$
$$P;\ (Q \sqcap R) = (P;\ Q) \sqcap (P;\ R) \tag{87}$$
$$P;\ (Q;\ R) = (P;\ Q);\ R \tag{88}$$
$$SKIP;\ P = P \tag{89}$$
$$P;\ SKIP = P \tag{90}$$
$$STOP;\ P = STOP \tag{91}$$
$$P \,\Box\, SKIP = (P \sqcap STOP) \,\Box\, SKIP \tag{92}$$

Si bien las leyes anteriores son bastante simples, a excepción de la primera, resulta más complejo entender y formalizar la relación entre terminación exitosa y paralelismo. ¿Qué debe ocurrir en $P \parallel Q$ si uno de los dos alcanza *SKIP*? El principio fundamental de CSP a este respecto se llama *terminación distribuida* y dice que una composición paralela termina cuando todos sus miembros terminan. La terminación distribuida considera que el evento especial $\surd$ es un evento compartido entre todos los procesos participantes de una composición paralela y por ende solo puede consumirse cuando todos están dispuestos a consumirlo. A su vez, si uno de los procesos alcanza $\surd$ no se comunicará más con el entorno en tanto que los restantes continuarán haciéndolo, lo que desde el entorno se percibe como que la composición paralela aun no ha terminado. Las leyes de CSP reflejan este principio:

$$SKIP \parallel SKIP = SKIP \tag{93}$$
$$SKIP \parallel e \rightarrow P = e \rightarrow (SKIP \parallel P) \tag{94}$$

14. Selección condicional

Existen algunos operadores más del álgebra pero en general no son de mucha utilidad en la práctica. El más útil que aun no hemos visto con detalle se llama *selección condicional* pues es semejante al **if** de los lenguajes de programación imperativos[4]. La selección condicional se puede escribir de varias formas pero Roscoe sugiere usar la forma más algebraica, es decir $P \lceil b \rceil Q$ en lugar de P **if** b **else** Q como lo hiciera Hoare en sus trabajos originales. Sea cual fuere la notación que se use las leyes fundamentales para la selección condicional son:

$$P \lceil true \rceil Q = P \tag{95}$$
$$P \lceil flase \rceil Q = Q \tag{96}$$

es decir $P \lceil b \rceil Q$ se comporta como P si b es verdadera y como Q en caso contrario.

En la condición de la selección condicional se pueden usar los parámetros del proceso (si los tiene) y de los canales y eventos compuestos que están dentro del ámbito léxico (tener en cuenta la discusión al respecto en la sección 13). La selección condicional puede utilizarse

[4]En años anteriores, siguiendo el libro de Hinchey y Jarvis [5], mostrábamos también el operador **while** semejante a la sentencia homónima de los lenguajes de programación imperativos. Roscoe muestra que no es posible definir tal operador debido a la semántica declarativa de CSP, como se mencionó en la sección 13.

como alternativa a las definiciones inductivas de los procesos parametrizados. Por ejemplo, la definición inductiva:

$$P(0) = a \rightarrow P(1)$$
$$P(n+1) = a \rightarrow P(n+2) \mid b \rightarrow P(n)$$

se puede reemplazar por:

$$P(n) = a \rightarrow P(1) \lceil n = 0 \rceil (a \rightarrow P(n+1) \mid b \rightarrow P(n-1))$$

Ejercicios

Ejercicio 17. *Suponga que desea agregar a CSP el operador* **while** *con la siguiente semántica:*

$$\textbf{while } b \textbf{ do } P = (P; \textbf{ while } b \textbf{ do } P) \lceil b \rceil SKIP$$

con la intención de que P se ejecute mientras b sea verdadera. Determine si la semántica formal es equivalente a la intención informal. Justifique su respuesta.

Ejercicio 18. *Redefina* (73) *de manera de que haya una única definición de B que contemple todos los casos posibles.*

15. La semántica de CSP: introducción al modelo de fallas y divergencias

Existen al menos tres formas de dar la semántica de CSP, a saber:

Operacional. Los procesos CSP se explican en términos de máquinas de estados infinitos. Algo así hicimos al comienzo del capítulo al explicar el significado de los primeros procesos que especificamos en términos de Statecharts. Es la forma más cercana a una implementación y la que menos conocimientos de matemática requiere, pero a la vez puede ser engorroso formalizar algunos conceptos.

Denotacional. CSP se proyecta en alguna teoría matemática ya explicada, por ejemplo teoría de conjuntos y matemática discreta. Cada elemento del lenguaje se explica en términos de esa teoría y luego las construcciones compuestas se componen en la teoría con los elementos propios de ella. La forma denotacional se separa considerablemente de una estrategia de implementación del lenguaje, pero requiere mayores conocimientos matemáticos. Es la forma que desarrollaremos en el resto de la sección y es la más común para dar semántica a CSP.

Algebraica. Esta es una forma especialmente adecuada para dar la semántica de un lenguaje como CSP. Se procede como en matemática (por ejemplo al definir grupos, espacios vectoriales, anillos, etc.): se axiomatizan ciertas propiedades y de esta axiomatización se deducen teoremas sobre la teoría. Las propiedades toman la forma de leyes algebraicas de ciertos operadores (en su mayoría simples igualdades). Estos axiomas son los teoremas que se deducen si la semántica se da denotacionalmente, de aquí la relación entre ambas formas. La forma algebraica puede considerarse como la manera más abstracta de formalizar un lenguaje.

Puede decirse que CSP está constituido por dos lenguajes: el núcleo del formalismo formado por eventos, operadores y procesos; y un lenguaje para especificar eventos compuestos, parámetros, canales y procesos parametrizados. Si se pretende dar la semántica de CSP se debería dar la semántica de ambos lenguajes. Sin embargo, formalizar la semántica del segundo lenguaje y su relación con el núcleo de CSP está fuera del alcance de este capítulo y no agregaría sustancialmente nada al conocimiento que se gana de CSP si solo se formaliza la semántica de su núcleo. Por consiguiente, a continuación prescindiremos del lenguaje relacionado con los procesos parametrizados y solo explicaremos la semántica del núcleo de CSP en términos denotacionales.

15.1. Trazas

La forma más simple de entender un proceso CSP es examinando las secuencias de eventos con los que se puede comunicar, en el orden en que estos se van dando. A estas secuencias se las denomina *trazas* (*traces*). Las trazas pueden ser finitas, tanto debido a que se observa al proceso por un tiempo finito como que no hay más comunicación entre el proceso y el entorno; o infinitas, porque se observa al proceso infinitamente y este comunica una infinita cantidad de eventos. Como veremos un poco más adelante, las trazas no son suficientes para explicar la semántica de CSP.

Las trazas pueden contener los eventos en el orden en que se fueron dando o los eventos y el tiempo que media entre cada uno de ellos. Como nosotros hemos usando CSP sin considerar el tiempo (y cuando lo hicimos, el paso del tiempo se representa con eventos) utilizaremos la primera forma. Más aun, nos limitaremos a registrar únicamente las trazas finitas pues estas son suficientes para describir la mayoría de los procesos (solo los procesos que son infinitamente no deterministas requieren trazas infinitas, pero esto a expensas de complicar el modelo semántico).

Si P es un proceso, $t(P)$ es el conjunto de todas las trazas finitas de P. $t(P)$ verifica siempre estas propiedades:

- Es no vacío, ya que la traza vacía, $\langle\rangle$, siempre pertenece.
- Es cerrado por prefijos, es decir si $s \frown t \in t(P)$ entonces $s \in t(P)$.

$t(P)$ no solo sirve para dar la semántica de CSP sino también para especificar el comportamiento requerido de un proceso, es decir las propiedades que este debe tener. Claramente para especificar este comportamiento debemos recurrir a lógica, matemática discreta y teoría de conjuntos lo que implica que tenemos un lenguaje para describir el proceso y otro para describir sus propiedades (cf. sección 1), lo que es una desventaja de CPS.

Para formalizar la semántica de CSP con el concepto de traza se debe indicar cuál es el conjunto de trazas de cada proceso. Esto, a su vez, se consigue *definiendo* el conjunto de trazas para cada construcción sintáctica posible, es decir inductivamente. Aquí solo daremos algunas de estas reglas:

$$t(STOP) = \{\langle\rangle\} \tag{97}$$

$$t(a \rightarrow P) = \{\langle\rangle\} \cup \{\langle a\rangle \frown t \parallel t \in t(P)\} \tag{98}$$

$$t(P \,\Box\, Q) = t(P) \cup t(Q) \tag{99}$$

$$t(P \sqcap Q) = t(P) \cup t(Q) \tag{100}$$

$$t(P \parallel Q) = \{t \in (\alpha P \cup \alpha Q)^{*\surd} \mid t \uparrow \alpha P^{\surd} \in t(P) \wedge t \uparrow \alpha Q^{\surd} \in t(Q)\} \tag{101}$$

$$t(\mathcal{F}[P]) = \{\mathcal{F}(t) \mid t \in t(P)\} \tag{102}$$
$$t(P \nabla Q) = \{t \frown s \mid t \in t(P) \wedge s \in t(Q)\} \tag{103}$$
$$t(SKIP) = \{\langle\rangle, \langle\surd\rangle\} \tag{104}$$
$$t(P;\ Q) = t(P) \cup \{s \frown t \mid s \frown \langle\surd\rangle \in t(P) \wedge t \in t(Q)\} \tag{105}$$

donde

- Si A es un conjunto A^* es el conjunto de todas las secuencias finitas formadas por elementos de A.
- Si A es un conjunto $A^{\surd} = A \cup \{\surd\}$.
- Si A es un conjunto $A^{*\surd} = A^* \cup \{s \frown \langle\surd\rangle \mid s \in A^*\}$.
- Si $t = \langle t_1, \ldots, t_n\rangle$ es una traza y A es un conjunto, $t \upharpoonright A = \langle t_{i_1}, \ldots, t_{i_k}\rangle$ donde $t_{i_j} \in A$ para todo j y $\langle t_{i_1}, \ldots, t_{i_k}\rangle$ es una subtraza de t.
- Si $\mathcal{F}$ es una función de renombramiento y $t = \langle t_1, \ldots, t_n\rangle$ es una traza, $\mathcal{F}(t) = \langle\mathcal{F}(t_1), \ldots, \mathcal{F}(t_n)\rangle$.

Entonces, suponiendo que se dan todas las reglas para obtener las trazas de todas las construcciones sintácticas posibles de CSP y asumiendo que el modelo de trazas es suficiente para formalizar la semántica de CSP, se deberían poder probar las leyes algebraicas como teoremas de esta teoría. La estrategia básica de prueba de una ley es demostrar que las trazas a ambos lados de la igualdad son las mismas. A continuación damos algunos ejemplos.

Teorema 4.

$$P \,\Box\, Q = Q \,\Box\, P$$

Demostración. Por (99):

$$t(P \,\Box\, Q) = t(P) \cup t(Q) = t(Q) \cup t(P) = t(Q \,\Box\, P)$$

□

Notar que en la prueba anterior hicimos uso de la conmutatividad de la unión de conjuntos, la cual es una propiedad de la teoría matemática sobre la cual proyectamos CSP. Esto es típico al formalizar la semántica de un lenguaje denotacionalmente.

Teorema 5.

$$STOP;\ P = STOP$$

Demostración. De (97) y (105):

$$t(STOP;\ P) = t(STOP) \cup \{s \frown t \mid s \frown \langle\surd\rangle \in t(STOP) \wedge t \in t(P)\} = t(STOP)$$

pues $s \frown \langle\surd\rangle \in t(STOP)$ es falso para toda traza s ya que la única traza de $STOP$ es la traza vacía, lo que implica que el término de la derecha de la unión, en la segunda igualdad, es vacío. □

Teorema 6.

$$e \in \alpha Q \wedge f \in \alpha P \Rightarrow e \rightarrow P \,\|\, f \rightarrow Q = STOP$$

Demostración. Sea $t \in t(e \rightarrow P \,\|\, f \rightarrow Q)$ entonces $t \upharpoonright \alpha(e \rightarrow P)^{\surd} \in t(e \rightarrow P)$ y $t \upharpoonright \alpha(f \rightarrow Q)^{\surd} \in t(f \rightarrow Q)$, por (101). Entonces t debería empezar con e y f al mismo tiempo o ser la traza vacía. Por lo tanto:

$$t(e \rightarrow P \,\|\, f \rightarrow Q) = \{\langle\rangle\} = t(STOP)$$

por (97). □

Ejercicios

Ejercicio 19. *Pruebe el siguiente teorema (suponiendo que el modelo de trazas es suficiente para formalizar la semántica de CSP).*

Teorema 7.

$$e \to P \,[\!]\, e \to Q = e \to (P \,[\!]\, Q)$$

15.1.1. ¿En qué se diferencian, entonces, $\Box$ y $\sqcap$?

Las ecuaciones (99) y (100) estipulan que las trazas de $P \Box Q$ y de $P \sqcap Q$ son las mismas lo que significa que $P \Box Q = P \sqcap Q$ debería ser un teorema. Pero, por otro lado, sabemos que esos procesos no pueden ser iguales puesto que uno implica no determinismo y el otro no. Evidentemente el modelo de trazas por sí solo no es suficiente para formalizar todo CSP.

Las trazas representan lo que un proceso hace, no lo que *debe* hacer. Por el contrario, el no determinismo se relaciona más con lo que un proceso *podría* hacer pero elige no hacer. Precisamente, $P \sqcap Q$ indica que el proceso *podría* comportarse como P o como Q pero no *debe* hacer, necesariamente, ni lo uno ni lo otro. Entonces, deberíamos buscar la forma de diferenciar $\Box$ de $\sqcap$ en lo que un proceso no hace o, más precisamente, lo que un proceso *se niega* a hacer. La idea es que el entorno de un proceso le ofrece a este cierto evento, pero el proceso lo *rechaza*. En este sentido si el entorno de $a \to STOP \sqcap b \to STOP$ le ofrece a el proceso puede rechazarlo, lo mismo ocurre si el entorno le ofrece b, pero si el entorno le ofrece ambos, el proceso *debe* aceptar uno de los dos aunque en otra oportunidad similar puede aceptar el otro o volver a aceptar el mismo.

Por lo tanto, si a las trazas le sumamos los rechazos podremos distinguir $a \to STOP \sqcap b \to STOP$ de $a \to STOP \Box b \to STOP$ pues, aunque las trazas son las mismas, los rechazos son muy distintos ya que el último no puede rechazar ni a ni b.

Ejercicios

Ejercicio 20. *Busque una forma de representar en Statecharts los eventos que se pueden rechazar en un determinado estado. Piense en transiciones espontáneas (recuerde que en su momento sugerimos que conviene no utilizarlas). Explique su solución.*

Represente, según su solución, $P = a \to P \sqcap b \to P$. Muestre los eventos que se pueden rechazar en cada estado.

15.2. Rechazos

Un conjunto de *rechazos* (*refusals*) de P es un conjunto de eventos en αP que el proceso puede evitar aceptar sin importar por cuánto tiempo el entorno se los ofrezca. El conjunto $r(P)$ es el conjunto de conjuntos de rechazos iniciales de P, es decir todos los conjuntos de eventos que P rechaza cuando solo ha producido la traza vacía. $r(P)$ se define, tal y como se hizo con $t(P)$, para cada construcción sintáctica elemental de CSP. Además, de manera muy semejante a $t(P)$, $r(P)$ verifica las siguientes propiedades:

- Es no vacío, ya que $\emptyset$ siempre pertenece.
- Es cerrado por subconjunots, es decir si $X \in r(P)$ y $W \subseteq X$, entonces $W \in r(P)$.

Los rechazos están íntimamente relacionados con el no determinismo del proceso: a más rechazos, más no determinista es el proceso. En efecto, por su misma definición, un conjunto de rechazos es un conjunto de eventos de su alfabeto que el proceso puede negarse a aceptar aun si el entorno se los ofrece, por lo que cuantos más eventos pueda rechazar el proceso más errático se comportará desde el punto de vista del entorno. Los rechazos son eventos que el entorno entiende que el proceso debería aceptar (pues están en su alfabeto) pero a veces los acepta y otras no. Claramente, esto es una formalización del no determinismo.

Algunas de las reglas para construir $r(P)$ inductivamente siguiendo la sintaxis de CSP son las siguientes:

$$r(STOP_A) = \mathbb{P}A^{\surd} \tag{106}$$
$$r(a \to P) = \{X \mid X \subseteq \alpha P \setminus \{a\}\} \tag{107}$$
$$r(P \,\Box\, Q) = r(P) \cap r(Q) \tag{108}$$
$$r(P \sqcap Q) = r(P) \cup r(Q) \tag{109}$$
$$r(P \parallel Q) = \{X \cup Y \mid X \in r(P) \wedge Y \in r(Q)\} \tag{110}$$
$$r(\mathcal{F}[P]) = \{\mathcal{F}(X) \mid X \in r(P)\} \tag{111}$$
$$r(P \,\triangledown\, Q) = r(P) \cup r(Q) \tag{112}$$
$$r(SKIP_A) = \mathbb{P}A \tag{113}$$
$$r(P;\, Q) = r(P) \tag{114}$$

Notar que ahora, $r(P \,\Box\, Q)$ es diferente de $r(P \sqcap Q)$ por lo que combinando las trazas con los rechazos podremos distinguirlos. No obstante, si quisiésemos diferenciar $e \to (a \to STOP \,\Box\, b \to STOP)$ de $e \to (a \to STOP \sqcap b \to STOP)$ los rechazos iniciales no alcanzan pues, inicialmente, ambos procesos tienen el mismo conjunto de rechazos, $\{\emptyset, \{a\}, \{b\}, \{a, b\}\}$. Entonces, para poder distinguir correctamente todos los procesos no solo necesitamos los rechazos iniciales de cada proceso sino los rechazos a medida que el proceso interactúa con su entorno. Por lo tanto, definimos P/t como el proceso P luego de ejecutar la traza $t \in t(P)$, y de ahora en más consideraremos, en general, $r(P/t)$ (que cuando $t = \langle\rangle$ coincide con $r(P)$).

Al combinar $t(P)$ con $r(P/t)$ para cada $t \in t(P)$ se obtienen las *fallas* del proceso, las cuales son el tema de la siguiente sección.

Ejercicios

Ejercicio 21. *Analice las reglas para calcular los rechazos iniciales y responda las siguientes preguntas dando una justificación apropiada.*

1. *¿Por qué en* (110) *se exige que* $X \setminus (\alpha P \cap \alpha Q)^{\surd} = Y \setminus (\alpha P \cap \alpha Q)^{\surd}$*? ¿Puede un evento en* $\alpha P \cap \alpha Q$ *pertenecer a algún elemento de* $r(P \parallel Q)$*? Ejemplifique una situación que aclare este último interrogante.*
2. *¿Por qué no aparecen los rechazos de* Q *en* (114)*?*
3. *¿Por qué sí aparecen los rechazos de* Q *en* (112)*?*

Ejercicio 22. *Pruebe que* $t(P \parallel P) = t(P)$.

15.3. Fallas = Trazas × Rechazos

Al combinar los rechazos con las trazas se obtienen las *fallas* (*failures*) del proceso. Más precisamente, $f(P) = \{(t,X) \mid t \in t(P) \wedge X \in r(P/t)\}$ es el conjunto de todas las fallas de P. Como con $t(P)$ y $r(P)$ las fallas se pueden calcular inductivamente siguiendo la sintaxis de CSP. Algunas de esas reglas son las siguientes:

$$f(STOP_A) = \{(\langle\rangle, X) \mid X \subseteq A^{\surd}\} \tag{115}$$

$$f(a \rightarrow P) = \{(\langle\rangle, X) \mid a \notin X \wedge X \subseteq \alpha P\} \cup \{(\langle a\rangle \frown t, X) \mid (t,X) \in f(P)\} \tag{116}$$

$$\begin{aligned} f(P \,\Box\, Q) = &\{(\langle\rangle, X) \mid X \in r(P) \cap r(Q)\} \\ &\cup \{(t,X) \mid (t,X) \in f(P) \cup f(Q) \wedge t \neq \langle\rangle\} \\ &\cup \{(\langle\rangle, X) \mid X \subseteq \alpha(P \,\Box\, Q) \wedge \langle\surd\rangle \in t(P) \cup t(Q)\} \end{aligned} \tag{117}$$

$$f(P \sqcap Q) = f(P) \cup f(Q) \tag{118}$$

$$f(P \parallel Q) = \{(t, X \cup Y) \mid t \in (\alpha P \cup \alpha Q)^{*\surd} \wedge (t \upharpoonright \alpha P, X) \in f(P) \wedge (t \upharpoonright \alpha Q, Y) \in f(Q)\} \tag{119}$$

$$f(\mathcal{F}[P]) = \{(\mathcal{F}(t), \mathcal{F}(X)) \mid (t,X) \in f(P)\} \tag{120}$$

$$f(P \,\nabla\, Q) = f(P) \cup \{(s \frown t, X) \mid s \in t(P) \wedge (t,X) \in f(Q)\} \tag{121}$$

$$f(SKIP_A) = \{(\langle\rangle, X) \mid X \subseteq A\} \cup \{(\langle\surd\rangle, X) \mid X \subseteq A^{\surd}\} \tag{122}$$

$$\begin{aligned} f(P;\, Q) = &\{(t,X) \mid \langle\surd\rangle \text{ ¡in } t \wedge (t, X^{\surd}) \in f(P)\} \\ &\cup \{(s \frown t, X) \mid s \frown \langle\surd\rangle \in t(P) \wedge (t,X) \in f(Q)\} \end{aligned} \tag{123}$$

Las reglas muestran claramente la gran diferencia que hay entre $\sqcap$ y $\Box$. Tanto es así que la regla para $\Box$ requiere tener en cuenta procesos como $SKIP \,\Box\, P$ que deben ser tratados especialmente debido a la posibilidad de terminación y su relación con el control que tiene el entorno sobre ese tipo de procesos. No ocurre lo mismo con $\sqcap$ pues el entorno no tiene tanto control sobre el proceso.

Aunque parezca que el modelo de fallas es suficientemente poderoso como para formalizar la semántica de CSP, no lo es. Existe cierto tipo de procesos que usualmente no se dan en la práctica pero que pueden aparecer en especificaciones e incluso a raíz de implementaciones erróneas. Son los procesos *divergentes*. Al combinar las fallas con las *divergencias* se obtiene el modelo semántico completo para CSP[5].

15.4. Divergencia

La *divergencia* es la posibilidad de que un proceso entre en una secuencia infinita de acciones internas. Claramente un proceso que diverge es inútil, incluso más que *STOP*, puesto que no tiene comunicación con el entorno y ni siquiera rechaza nada (en el sentido de $r(P)$). Por todas estas razones, el estudio de la divergencia tiene escaso interés práctico y no realiza un aporte significativo para la comprensión de la teoría de CSP, por lo que no le dedicaremos mucho espacio en este apunte.

[5]En realidad no es tan así pues el modelo de fallas y divergencias no resuelve correctamente el problema de los procesos infinitamente no deterministas, pero esto está decididamente fuera del alcance de este curso; para más información ver [6, capítulo 10].

Una de las causas por las que aparece la divergencia en CSP se debe a la ocultación de eventos en composiciones paralelas, en las cuales los procesos se comunican infinitamente entre ellos sin comunicarse jamás con el entorno.

CSP considera que una vez que un proceso diverge no es posible saber con precisión qué es lo que hace después por lo que se considera que dos procesos que divergen son equivalentes (iguales). Más aun, asumiremos que una vez que un proceso diverge este puede ejecutar cualquier traza, rechazar cualquier evento y siempre diverge en cualquier traza ulterior. Como todos los procesos que divergen son equivalentes, es posible definir un proceso que lo único que hace es divergir, **div**. Además, definimos $d(P)$ como el conjunto de todas las trazas de P a partir de las cuales P diverge, más todas las extensiones de aquellas. En otras palabras si $t \in d(P)$ y $s \in \alpha P^*$ entonces $t \frown s \in d(P)$.

15.5. Equivalencia de procesos

El modelo semántico de CSP se conoce como *modelo de fallas y divergencias*. En CSP dos procesos, P y Q, son equivalentes sí y sólo sí

$$(\alpha P, f_\perp(P), d(P)) = (\alpha Q, f_\perp(Q), d(Q)) \tag{124}$$

donde:

$$f_\perp(P) = f(P) \cup \{(t, X) \mid t \in d(P)\} \tag{125}$$

aunque para la mayoría de las cuestiones prácticas es suficiente con considerar la equivalencia si $(\alpha P, f(P)) = (\alpha Q, f(Q))$.

A las fallas en $f(P)$ se las llama *fallas estables* en tanto que las de la forma (t, X) con $t \in d(P)$ se llaman *fallas inestables*. $f_\perp(P)$ y $d(P)$ se definen inductivamente siguiendo la sintaxis de CSP.

$$f_\perp(STOP_A) = f(STOP_A) \tag{126}$$

$$d(STOP_A) = \emptyset \tag{127}$$

$$f_\perp(a \rightarrow P) = \{(\langle\rangle, X) \mid a \notin X \wedge X \subseteq \alpha P\} \cup \{(\langle a\rangle \frown t, X) \mid (t, X) \in f_\perp(P)\} \tag{128}$$

$$d(a \rightarrow P) = \{\langle a\rangle \frown t \mid t \in d(P)\} \tag{129}$$

$$\begin{aligned} f_\perp(P \,\Box\, Q) = &\{(\langle\rangle, X) \mid (\langle\rangle, X) \in f_\perp(P) \cap f_\perp(Q)\} \\ &\cup \{(t, X) \mid (t, X) \in f_\perp(P) \cup f_\perp(Q) \wedge t \neq \langle\rangle\} \\ &\cup \{(\langle\rangle, X) \mid \langle\rangle \in d(P) \cup d(Q)\} \\ &\cup \{(\langle\rangle, X) \mid \langle\surd\rangle \in t_\perp(P) \cup t_\perp(Q)\} \end{aligned} \tag{130}$$

$$d(P \,\Box\, Q) = d(P) \cup d(Q) \tag{131}$$

$$f_\perp(P \sqcap Q) = f_\perp(P) \cup f_\perp(Q) \tag{132}$$

$$d(P \sqcap Q) = d(P) \cup d(Q) \tag{133}$$

$$\begin{aligned} f_\perp(P \parallel Q) = &\{(t, X \cup Y) \mid \\ &t \in (\alpha P \cup \alpha Q)^{*\surd} \wedge (t \uparrow \alpha P, X) \in f_\perp(P) \wedge (t \uparrow \alpha Q, Y) \in f_\perp(Q)\} \\ &\cup \{(t, X) \mid t \in d(P \parallel Q)\} \end{aligned} \tag{134}$$

$$d(P \,\|\, Q) = \{t \mid (t \uparrow \alpha P \in d(P) \wedge t \uparrow \alpha Q \in t_\perp(Q)) \\ \vee (t \uparrow \alpha Q \in d(Q) \wedge t \uparrow \alpha P \in t_\perp(P))\} \tag{135}$$

$$f_\perp(SKIP_A) = f(SKIP_A) \tag{136}$$

$$d(SKIP_A) = \emptyset \tag{137}$$

$$f_\perp(P;\, Q) = \{(t,X) \mid \langle\surd\rangle \text{ ni } t \wedge (t, X^{\surd}) \in f_\perp(P)\} \\ \cup \{(s^\frown t, X) \mid s^\frown \langle\surd\rangle \in t_\perp(P) \wedge (t,X) \in f_\perp(Q)\} \\ \cup \{(t,X) \mid t \in d(P;\, Q)\} \tag{138}$$

$$d(P;\, Q) = d(P) \cup \{s^\frown t \mid s^\frown \langle\surd\rangle \in t_\perp(P) \wedge t \in d(Q)\} \tag{139}$$

donde $t_\perp(P) = t(P) \cup d(P)$. Notar que muchas de las leyes de $f_\perp(P)$ son semejantes a las dadas en la sección 15.3 excepto que se reemplaza $f(P)$ por $f_\perp(P)$ y/o se agregan explícitamente las divergencias del proceso.

15.5.1. Determinismo y abrazo mortal

En el modelo de fallas y divergencias es muy simple formalizar los conceptos de proceso determinista y abrazo mortal.

El proceso P es determinista sí y sólo sí $d(P) = \emptyset$ y si para todo $a \in \alpha P$ y toda $t \in t(P)$ si $t^\frown \langle a\rangle \in t(P)$, entonces $(t, \{a\}) \notin f(P)$.

El proceso P está libre de abrazo mortal sí y sólo sí para toda $t \in t(P)$ vale que $(t, \alpha P) \notin f(P)$.

Referencias

[1] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1985.

[2] R. Allen, "A formal approach to software architecture," Ph.D. dissertation, Carnegie Mellon School of Computer Science, 1997.

[3] M. Jackson, *Software requirements & specifications: a lexicon of practice, principles and prejudices*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1995.

[4] P. Zave and M. Jackson, "Four dark corners of requirements engineering," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 1, Jan. 1997.

[5] M. G. Hinchey and S. A. Jarvis, *Concurrent systems: formal development in CSP*. McGraw-Hill International Series in Software Engineering, 1995.

[6] A. W. Roscoe, *The Theory and Practice of Concurrency*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.