Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Resúmen II

Ingeniería de Software I

Autor:
Arroyo, Joaquín

22 de junio de 2023

Índice

1. Designaciones

Una designación es un texto que vincula un elemento de R(Requerimiento) con uno de S(Especificación).

Regla de conocimiento(R) $\approx$ Término formal(S)

La Regla de conocimiento es un texto en lenguaje natural, una o dos líneas, que debe permitir reconocer un elemento de R, claramente, sin ambigüedad.

El Término formal es un texto formal en el lenguaje de especificación.

Algunos ejemplos:

d es un monto de dinero $\approx$ $d \in$ Dinero

La caja de ahorros n tiene saldo $s \approx (n, s) \in ca$

Las designaciones pueden tener distintas caracteristicas:

Pueden ser *Machine Controlled*(MC) o *Environment Controlled*(EC), y pueden ser, *Unshared*(U) o *Shared*(S). Siempre tenemos que terminar con designaciones que sean S y sin referencias a futuro, para esto, utilizamos sensores y temporizadores o contadores respectivamente.

2. DK, R y S

DK es conocimiento de dominio. Los esquemas de este tipo, deben hacerse lo más simple posible. Se hacen sobre 'dominio' del cual se sabe el comportamiento. Se podría decir que son 'especificaciones' que vienen por fuera de nuestro sistema. Por ejemplo, el funcionamiento de un semáforo, de una lámpara, etc.

R son los requerimientos. Se hacen a partir de las designaciones.

S es la especificacion. Se hacen a partir de las nuevas designaciones (NO compartidas se cambian por compartidas, y referencias a futuro por contadores/temporizadores).

La diferencia entre R y S es que, R contiene las designaciones que NO son compartidas y referencias a futuro, en cambio S, es donde se reemplaza todas las designaciones NO compartidas, por compartidas, y las referencias a futuro, por temporizadores o contadores.

Para obtener el sistema final, se combinan los DKs junto a S.

En caso de no tener designaciones NO compartidas ni referencias a futuro, no se realiza R, se pasa directamente a S.

3. Statecharts

3.1. Introducción

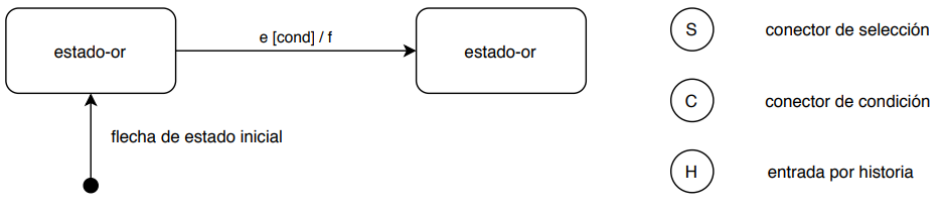
Lenguaje gráfico con semántica ejecutable. Se basa en el formalismo típico para describir FSM, aunque lo extiende de manera brillante. Imposible hablar de tipos.

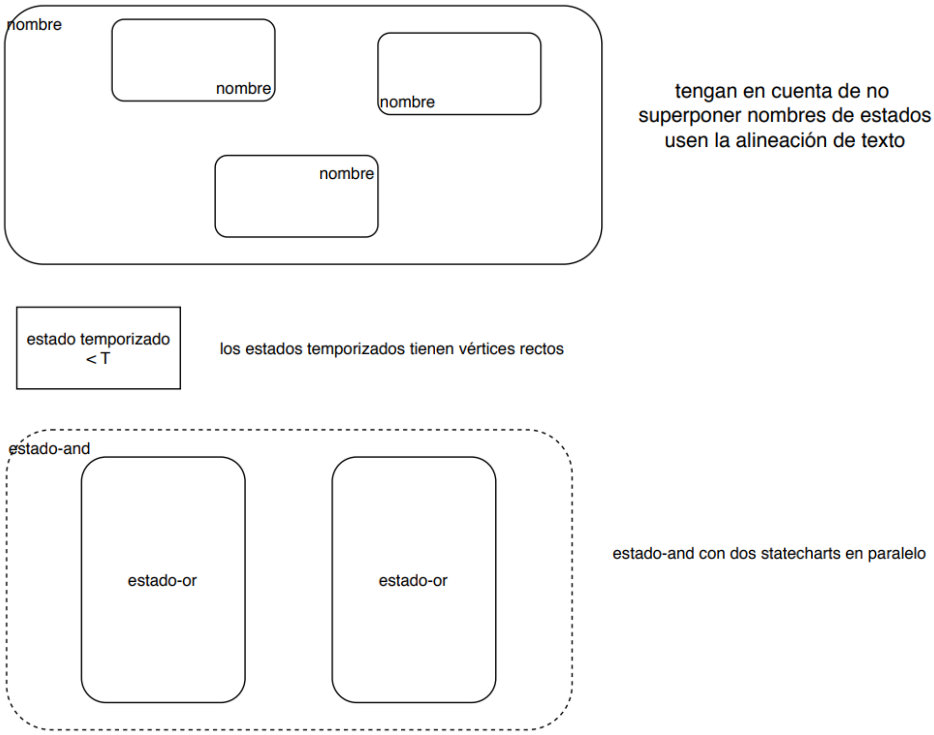
Los conceptos básicos son eventos y estados; a ellos se agregan condiciones para las transiciones, super-estados, concurrencia, temporización, historia, etc.

Utilizamos designaciones para representar sistemas en este lenguaje. Se comienza representando los DKs, luego R en caso de tener designaciones NO compartidas y referencias a futuro, y por último S convirtiendo las designaciones NO compartidas y referencias a futuro por nuevas designaciones que utilicen sensores, y temporizadores o contadores respectivamente.

3.2. Plantillas

Para desarrollar los statecharts, utilizamos las siguientes plantillas/conectores:



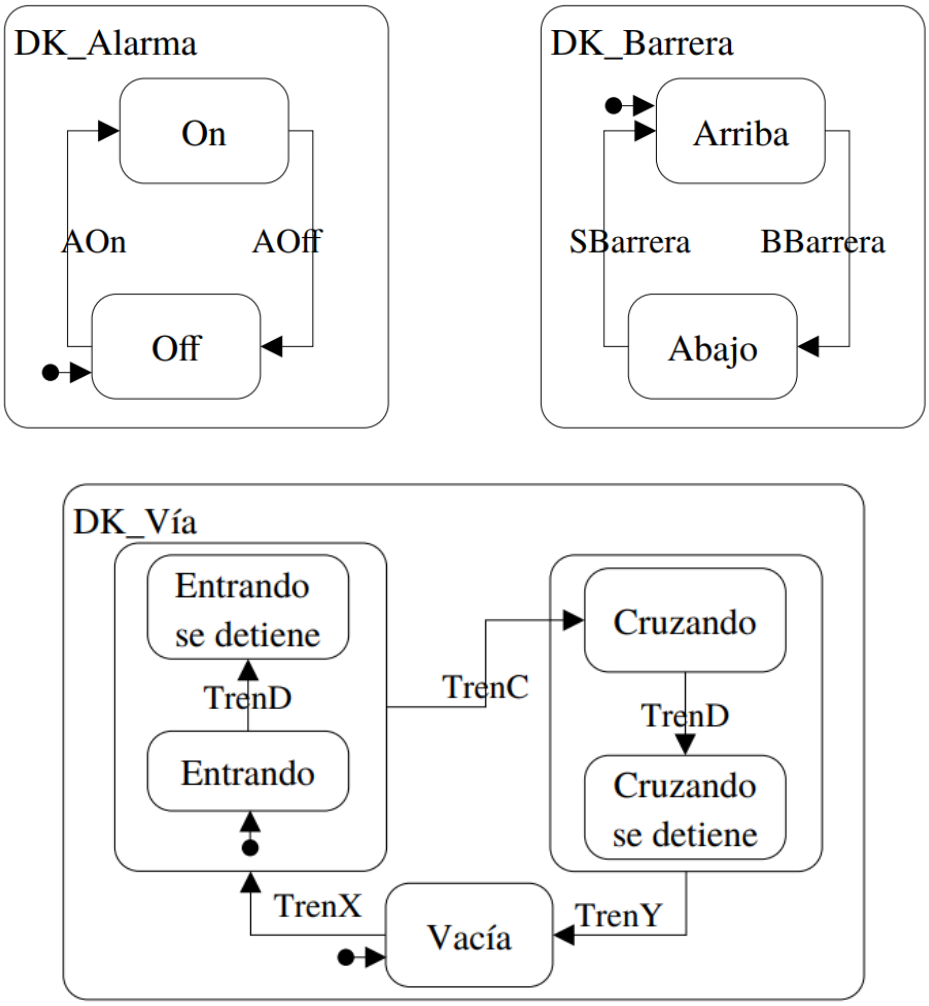


3.3. Ejemplos

Tomemos las siguientes designaciones:

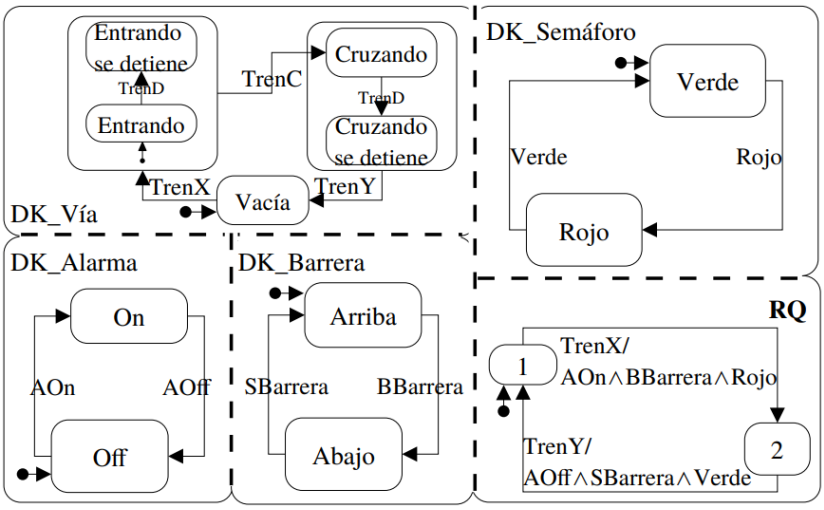
- La distancia de seguridad para bajar la barrera ≈ X
- La distancia de seguridad para subir la barrera ≈ Y
- Un tren cruza la distancia X acercándose al cruce por la vía v ≈ TrenX(v)
- Un tren alcanza la zona de carretera por la vía v ≈ TrenC(v)
- Un tren se detiene en la vía v en la zona de peligro ≈ TrenD(v)
- El último vagón de un tren pasa la distancia Y alejándose del cruce por la vía v ≈ TrenY(v)
- El sistema enciende la alarma ≈ AOn
- El sistema apaga la alarma ≈ AOff
- El sistema baja la barrera b ≈ BBarrera(b)
- El sistema sube la barrera b ≈ SBarrera(b)
- El sistema pone el semáforo s en rojo ≈ Rojo(s)
- El sistema pone el semáforo s en verde ≈ Verde(s)

3.3.1. DKs

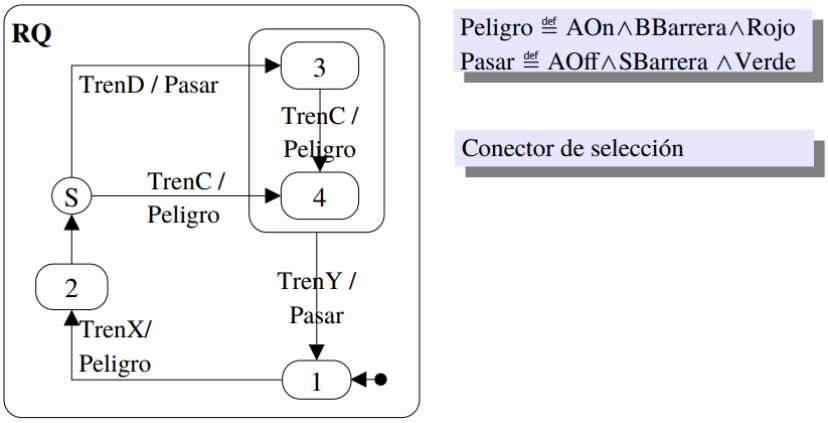

DK_Alarma
On
AOn
AOff
Off
DK_Barrera
Arriba
SBarrera
BBarrera
Abajo
DK_Vía
Entrando se detiene
TrenD
Entrando
TrenC
Cruzando
TrenD
Cruzando se detiene
TrenX
Vacía
TrenY

3.3.2. R

R simple para una sola vía:



R completo:

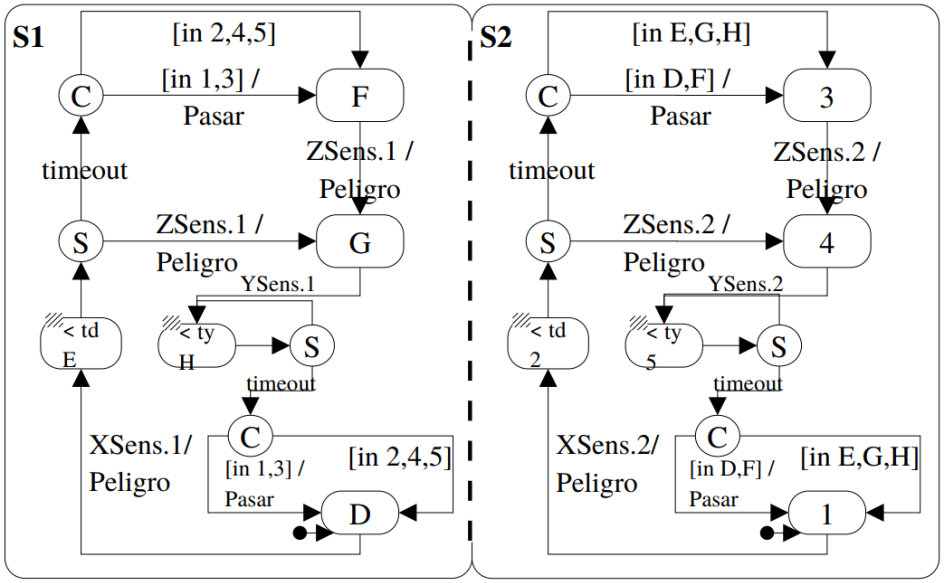


3.3.3. S

En este caso R no es S ya que tenemos varios fenómenos no compartidos(TrenX), y hay referencias a futuro(TrenY). Debemos crear nuevas designaciones para quitar estos problemas:

- El sensor de distancia X de la vía v envía una señal ≈ XSens(v)
- El sensor de distancia Y de la vía v envía una señal ≈ YSens(v)
- El sensor de zona de carretera de la vía v envía una señal ≈ ZSens(v)
- Tiempo de espera para determinar la detención de un tren ≈ td
- Tiempo de espera desde que se recibió la última señal de un sensor Y ≈ ty

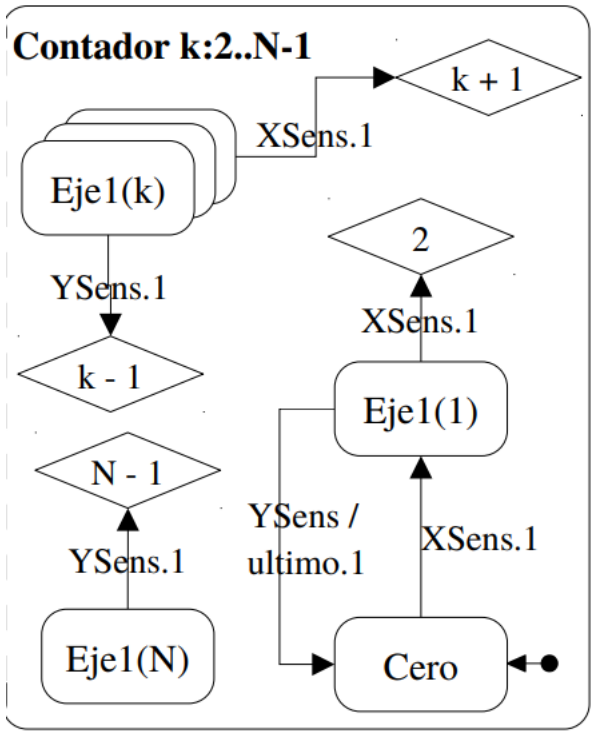
A partir de esto, desarrollamos la especificación S:



Notar que S se realizó para DOS vías, en este caso el R que realizamos no nos sirve, hay que hacerlo para dos vías también.

3.4. Contador

Se presenta un ejemplo de un contador



4. CSP

4.1. Introduccion

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.

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.

4.2. Eventos, procesos y recursividad

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 $NOMBREPROCESO = definicion$, 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:

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

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 $PRESS1$. A su vez $PRESS1$ en algún momento emitirá el evento *free* luego de lo cual se comportará como el proceso $PRESS2$. 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 que el entorno emita *remove*, de forma tal que finalmente se comportará como el proceso $PRESS1$.

En otro orden, la definición de DK_PRESS es recursiva en tanto que alguno de sus componentes ($PRESS1$ 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 semantica de $\overline{e}$ es que e y $\overline{e}$ son el mismo evento solo que se conviene que el proceso donde aparece $\overline{e}$ es el que lo emite, y los demas lo esperan.

Hay varias formar de escribir los procesos presentados mas arriba, y 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.

4.3. 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 $\triangledown$, llamado interrupción. Veamos cada uno de ellos:

- El proceso $P = e \rightarrow Q \mid f \rightarrow R$ ofrece las alternativas e y f a su entorno. 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. 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 $P = e_1 \to P_1 \mid ... \mid e_n \to P_n$.

- 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 $P = e \to Q \Box R$, $e \to P \Box e \to Q$, $P = \Box_{i=1}^{n} e_i \to P_i$, o $P = \Box i \in [1, n] \bullet e_i \to P_i$.

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

- $P = Q \triangledown R$ 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.

En consecuencia a estas definiciones, el verdadero comportamiento de la prensa se expresa por medio de:
$DK\_PREES = start \to PRESS1 \triangledown abort \to DK\_PRESS$

4.4. Primera aproximación a las leyes algebraicas de CSP

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:

- Nos permiten mejorar la comprensión e intuición del significado deseado para los operadores.
- Son muy útiles a la hora de hacer pruebas de propiedades de procesos CSP (como veremos más adelante).
- 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).

A continuación enunciaremos solo algunas de las leyes que verifican los operadores vistos hasta el momento (recordar que $\to$ tiene máxima prioridad excepto por los paréntesis):

1. $e \to P \Box f \to Q = e \to P \mid f \to Q$
2. $P \Box P = P$
3. $P \Box Q = Q \Box P$
4. $P \Box (Q \Box R) = (P \Box Q) \Box R)$
5. $e \to P \triangledown Q = Q \Box e \to (P \triangledown Q)$
6. $P \triangledown (Q \triangledown R) = (P \triangledown Q) \triangledown R)$
7. $P \triangledown (Q \Box R) = (P \triangledown Q) \Box (P \triangledown R)$

Las leyes enunciadas expresan claramente que:
| y $\Box$ son idénticos si se dan explícitamente los primeros eventos de cada proceso. es idempotente, conmutativo y asociativo.
$\triangledown$ es asociativo y es distributivo respecto de $\Box$.

4.5. Alfabeto de un proceso

Conjunto de los eventos que participan en la defínción de un proceso P. Se nota como αP.
Si $P = a \to f \to p \to Q$ y $Q = e \to b \to P$, entonces $\alpha P = \{a, f, p, e, b\} = \alpha Q$

4.6. Concurrencia e intercalación: procesos secuenciales

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 4 leyes fundamentales que gobiernan el significado de ||, por ahora se presenta solo la primera:

> 1. $e \notin \alpha P, f \notin \alpha Q \Rightarrow f \to P \,||\, e \to Q = f \to (P \,||\, e \to Q) \square e \to (f \to P \,||\, Q)$ (interleaving). Esta ley 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.

Se puede ver que aunque utilicemos el operador || TODOS los procesos terminan siendo secuenciales. Para demostrar esto, debemos expandir las definiciones de procesos que utilicen || hasta que estas desaparezcan.

4.7. Renombramiento

Permite renombrar los eventos de un proceso, y hay dos formas de hacerlo: Renombramiento funcional y Renombramiento indexado.

4.7.1. Renombramiento funcional

Estos conjuntos asocian el nombre de un proceso a un nuevo nombre.
$R = \{tobelt \to tbr, take \to tr, topress \to tpr, release \to rr, p \to pr\}$
$L = \{tobelt \to tbl, take \to tl, topress \to tpl, release \to rl, p \to pl\}$.
Es decir, definimos eventos para lado derecho e izquierdo.
A partir de esto se pueden definir procesos, como por ejemplo:
$ARMR = R[DK\_ARM], ARML = L[DK\_ARM]$
Donde $C[P]$ toma el proceso P y realiza todos los renombramientos definidos en C.

Algunas leyes de renombramiento:

- $e \in domF \Rightarrow F[e \to p] = F(e) \to F[P]$
- $e \notin domF \Rightarrow F[e \to p] = e \to F[P]$
- $F[P \square Q] = F[P] \square F[Q]$. (Lo mismo para ||)

4.7.2. Renombramiento indexado

Se utiliza cuando se quiere renombrar muchas veces un proceso, por ejemplo:

$ROBOT = ||_{i=1}^{1}\, 0i : DK\_ARM$ donde $i : DK\_ARM = F_i[DK\_ARM]$ donde $F_i = \{tobelt \to i.tobelt, take \to i.take, topress \to i.topress, release \to i.release\}$

Por lo tanto, si $i \neq j, \alpha(i : DK\_ARM) \cap \alpha(j : DK\_ARM) = \emptyset$

Es decir, se toman todos los eventos en la definicion de DK_ARM y los indexa. Vamos a ver que existen casos en los que no queremos que indexe todo, por lo que se puede combinar renombramiento indexado y funcional.

4.7.3. Combinar renombramiento funcional e indexado

En el ejemplo anterior, se indexaban los eventos *start* y *abort*, lo cual no queremos que pase. Para evitar esto, hacemos lo siguiente:

$H = \{x.start \to start \mid x \in [1, 10]\} \cup \{x.abort \to abort \mid x \in [1, 10]\}$
$ROBOT = H[||_{i=1}^{1}\, 0i : DK\_ARM]$

4.8. El operador de selección interna

$P \sqcap Q$ es un proceso que decide si se comporta como P o como Q, hay un NO determinismo, ya que el entorno no puede controlar al proceso.

Veamos el siguiente ejemplo:

El sistema debe inicializar cada brazo del robot, llevandolo a la cinta, y liberando lo que haya tomado. ¿Cual de los dos brazos se inician primero?

$INITL = tbl \to rl \to tbr \to rr \to WORK$

$INITL = tbr \to rr \to tbl \to rl \to WORK$

$INIT = INITL \sqcap INITR$

A partir de esto, introducimos un NO determinismo en la definición de $INIT$.

4.9. Leyes fundamentales restantes

2. $e \to P \;||\; e \to Q = e \to (P \;||\; Q)$ Ley de sincronización.

3. $e \in \alpha Q, f \notin \alpha P \Rightarrow e \to P \;||\; f \to Q = f \to (e \to P \;||\; Q)$ Ley de eventos independientes. e evento compartido, f evento independiente.

4. $e \in \alpha Q, f \in \alpha P \Rightarrow e \to P \;||\; f \to Q = STOP$ Ley de Deadlock. STOP es un proceso que forma parte del lenguaje, es como una constante. Este proceso no ejecuta ningún evento, no hace nada, indica terminación anormal.

4.10. Comuniación de datos y eventos compuestos

Eventos de la forma $c?x$ y $c!x$ definen un canal de comunicación, donde c es el canal y x el parámetro que recibe. $c?x$ es la entrada del canal, y $c!x$ la salida.

Los canales se pueden utilizar con múltiples parámetros: $c?x.y.z$, $c!5.(4,3).\langle 3\rangle$. No está especificado que tipos se pueden utilizar, podemos utilizar los típicos, como listas, conjuntos, pares ordenados, etc.

$\bullet c?x \to P(x) \;||\; c!y \to Q = c!y \to (P(y) \;||\; Q)$ Ley de comunicación.

Los canales de proceso son siempre unidireccionales(Hoare).

$c?x \to c!y \to P$ No es correcto (según Hoare).

$cin?x \to cout!y \to P$ Define dos canales.

4.11. Procesos parametrizados

Si se quiere parametrizar un proceso P, se escribe la definición de cada uno de los casos que querramos representar, por ejemplo: $P(0) = definition$, $P(n) = definition$.

Veamos el siguiente ejemplo, sea $BUFFER$ el siguiente proceso:

$BUFFER = long?n + 1 \to B(n+1, \langle\rangle)$

$B(m+1, \langle\rangle) = left?n.N \to B(m, \langle n\rangle)$

$B(m+1, s \frown \langle y\rangle) = left?n.N \to B(m, \langle n\rangle \frown s \frown \langle y\rangle)$
$\qquad\qquad\qquad\qquad\qquad\qquad \mid right!y \to B(m+2, s)$

$B(0, s \frown \langle y\rangle) = right!y \to B(1, s)$

$B(0, \langle\rangle)$ no hace falta definirlo, ya que como mínimo el proceso B recibe un 1 para el tamaño del buffer.

4.12. Procesos $STOP$ y $SKIP$

El proceso $STOP$ indica una terminación incorrecta de un proceso.

Al utilizar el proceso $STOP$ es conveniente hacerlo con su alfabeto A, es decir $STOP_A$, esto quiere decir que tenemos infinitos $STOP$ que dependen de un alfabeto. Cuando en el contexto del proceso queda claro, o cuando se quiere enunciar un resultado genérico que no dependa del alfabeto, no hace falta utilizarlo.

Algunas leyes de $STOP$:

1. $STOP \Box P = P$
2. $P \,||\, STOP_A = STOP_{A \cup \alpha P}$ sii $\alpha P \cap A \neq \emptyset$
3. $P \,||\, STOP_A = P$ sii $\alpha P \cap A = \emptyset$

El proceso *SKIP* indica una terminación exitosa de un proceso. $SKIP = \checkmark \rightarrow STOP$ donde $\checkmark$ es un evento reservado.
Algunas leyes de *SKIP*:

1. $SKIP;\ P = P$
2. $P;\ SKIP = P$
3. $STOP;\ SKIP = STOP$
4. $P \Box SKIP = (P \sqcap STOP) \Box SKIP$
5. $SKIP \,||\, e \rightarrow P = e \rightarrow (P \,||\, SKIP)$
6. $SKIP \,||\, SKIP = SKIP$

4.13. Operador condicional

Supongamos que tenemos el evento e, y los procesos Q y R.
$P = e \rightarrow (Q \lceil c \rceil R)$
donde c es una determinada condición. La semántica nos dice que, si se cumple c entonces ejecutamos Q, caso contrario ejecutamos R.

4.14. Especificación de requisitos temporales

Se pueden utilizar:

- TimedCSP: Tiempo continuo, es otro modelo semántico.
- Modelar el tiempo con un evento cuya interpretación es que ha pasado una unidad de tiempo. Tiempo discreto → mismo modelo semántico. El tiempo avanza una unidad ≈ tick.

Vamos a modelar el tiempo como en Statecharts. El estado temporizado funciona como un timer.
En CSP no tenemos estados, pero podemos modelar un *TIMER*, luego usamos las primitivas del timer para simular estados temporizados.
Se configura e inicia el *TIMER* para que cuente n unidades de tiempo ≈ $start?n$. EC, S
Se detiene el timer ≈ *stop*. EC, S.
El timer alcanza la cota de tiempo estipulada ≈ *timeout*. MC, S

$TIMER = start?n \rightarrow (TIMER'(n) \lceil n > 0 \rceil TIMER) \triangledown stop \rightarrow TIMER$
$TIMER'(0) = \overline{timeout} \rightarrow TIMER$
$TIMER'(n+1) = tick \rightarrow TIMER'(n)$

Veamos el timer en un ejemplo:
Supongamos que se entra a un estado a partir de un evento a, luego dicho estado tiene un temporizador en t, y podemos salir por un evento b, o por un *timeout*.
$P = a \rightarrow start!T \rightarrow (b \rightarrow STOP \rightarrow Q \mid timeout \rightarrow R) \,||\, TIMER$

Para el parcial, utilizar el timer *RTR*, se combinan el DK junto al timer definido anteriormente:
$DKRT = \overline{tick} \rightarrow DKRT$
$RTR = DKRT \,||\, TIMER$
$RTR(n) = DKRT \,||\, H[||_{i=1}^{n}\ i : TIMER]$

donde $H = \{x.tick \rightarrow tick \mid x \in [1, n]\}$

4.14.1. Operadores temporales: Espera inactiva

$WAIT(t)$: Es un proceso que no ejectura ningún evento durante t unidades de tiempo.
$WAIT(t) = (start!t \rightarrow timeout \rightarrow SKIP) \;||\; TIMER$

$SKIP$: Proceso de CSP que indica terminación exitosa.

4.14.2. Operadores temporales: Composición secuencial de procesos

$Q;\ R$, significa que R comienza a ejecutarse luego de Q sii Q terminó de manera existosa, es decir, con un $SKIP$.
Algunas leyes:

1. $c?x \rightarrow P;\ Q = c?x \rightarrow (P;\ Q)$ si x no esta libre en Q
2. $(P \sqcap Q);\ R = (P;\ R) \sqcap (Q;\ R)$
3. $P;\ (Q \sqcap R) = (P;\ Q) \sqcap (P;\ R)$
4. $P;\ (Q;\ R) = (P;\ Q);\ R$

4.14.3. Operadores temporales: Prefijación temporizada

$e \rightarrow^t P$ significa que la transición entre e y P demora t unidades de tiempo.
$e \rightarrow^t P = e \rightarrow start!t \rightarrow timeout \rightarrow P$

4.14.4. Operadores temporales: timeout

$e \rightarrow P \rhd^t Q$ significa que si e aparece antes de t unidades de tiempo, se comporta como P, caso contrario, se comporta como Q.
$e \rightarrow P \rhd^t Q = start!t \rightarrow (e \rightarrow stop \rightarrow P \mid timeout \rightarrow Q)$
$e \rightarrow^t P = e \rightarrow start!t \rightarrow timeout \rightarrow P$

4.14.5. Funciones subespecificadas

Son funciones sobre las cuales decimos su comportamiento pero no lo especificamos. Por ejemplo, una función que nos devuelva la hora actual: $\underline{horaactual()}$, una función que calcule el angulo para llegar de un punto x a un punto y: $\underline{grados(x, y)}$.
El subrayado indica que esta subespecificada.

4.15. Modelo semántico de fallas y divergencias

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

- Operacional: Máquina de estados.
- Denotacional: Teoría de conjuntos.
- Algebraica: Teoría axiomática. (Similar a la teoría de grupos, anillos, etc.)

Vamos a utilizar la forma denotacional.
Tenemos por un lado las Fallas, y por otro lado las Divergencias.

4.15.1. Fallas

Las fallas se dividen en **trazas** y **rechazos**.

Trazas: Secuencias de eventos comunicados por un proceso. Conjunto de trazas de P $\rightarrow tP$ o $t(P)$. Pueden ser secuencias finitas o infinitas.
$P = a \rightarrow b \rightarrow STOP \Rightarrow tP = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle\}$
$P = e \rightarrow P \Rightarrow tP = \{\langle\rangle, \langle e\rangle, \langle e, e\rangle, ...\}$

De ahora en más solo consideramos trazas finitas, es más que suficiente.
Los conjuntos de trazas de los procesos se construyes de la siguiente forma:

- $\langle\rangle \in tP$
- $s \frown t \in tP \Rightarrow s \in tP$ (tP es cerrado por prefijos)

Se presenta la semántica de las trazas:

1. $t(STOP) = \{\langle\rangle\}$
2. $t(a \rightarrow P) = \{\langle\rangle\} \cup \{\langle a\rangle \frown t \mid t \in tP\}$
3. $t(P \Box Q) = tP \cup tQ$
4. $t(P \sqcap Q) = tP \cup tQ$
5. $t(P \mid\mid Q) = \{t \in (\alpha P \cup \alpha Q)^* \mid t \uparrow \alpha P \in tP \wedge t \uparrow \alpha Q \in tQ\}$ donde $\uparrow$ significa que, dado $t = \langle a, b, c\rangle$, si $a, b \in \alpha Q$ entonces $t \uparrow \alpha Q = \langle a, c\rangle$ (restricción)
6. $t(SKIP) = \{\langle\rangle, \checkmark\}$

Veamos que 3 y 4 son iguales, es decir, necesitamos más riqueza semántica en el modelo para distinguir comportamientos deterministas de no deterministas. Para esto introducimos los rechazos.

Rechazos: Eventos que un proceso puede no ejecutar sin importar por cuanto tiempo el entorno los ofrezca.
$a \rightarrow P \Box b \rightarrow Q$, el entorno ofrece el eventoe a o b, y el proceso debe ejecutar.
$a \rightarrow P \sqcap b \rightarrow Q$, el entorno ofrece el evento a o b, y el proceso puede ejecutar, o puede rechazar. $\{a\}$ y $\{b\}$ son rechazos iniciales del proceso.
Rechazo $\Rightarrow$ no determinismo.

Se define el conjunto de conjuntos de rechazos iniciales de un proceso P como rP o $r(P)$. Estos conjunto se contruyen de la siguiente forma:

- $\emptyset \in rP$
- $A \in rP, B \subseteq A \Rightarrow B \in rP$ (rP es cerrado por subconjuntos)

Se presenta la semántica de los rechazos iniciales:

1. $r(STOP_A) = \mathbb{P}\, A$
2. $r(SKIP) = \mathbb{P}\, A$
3. $t(a \rightarrow P) = \{X \mid X \subseteq \alpha P \setminus \{a\}\}$
4. $r(P \Box Q) = rP \cap rQ$
5. $r(P \sqcap Q) = rP \cup rQ$
6. $r(P \mid\mid Q) = \{A \cup B \mid A \in rP \wedge B \in rQ\}$

Fallas(Trazas $\times$ Rechazos): Definimos P/s como el proceso P luego de ejecutar la traza $s \in tP$.
$r(P/s)$ es el conjunto de conjuntos de rechazos iniciales de P/s.

Las fallas de un proceso P se definen como fP o $f(P)$, y $fP = \{(s, A) \mid s \in tP \wedge A \in r(P/s)\}$, luego, esta es la semántica de las fallas.

Un proceso P es determinista sii $\forall\, a \in \alpha P, s \in tP, s \frown \langle a \rangle \in tP \Rightarrow (s, \{a\}) \notin fP$

Un proceso P esta libre de deadlock sii $\forall\, s \in tP, (s, \alpha P) \notin fP$

4.15.2. Divergencias:

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 $r(P)$).

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. 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 $t \in d(P) \wedge s \in \alpha P^*$ entonces $t \frown s \in d(P)$.+

En CSP dos procesos, P y Q, son equivalentes sí y sólo sí $(\alpha P, f_\perp(P), d(P)) = (\alpha Q, f_\perp(Q), d(Q))$
donde $f_\perp(P) = f(P) \cup \{(t, X) \mid t \in d(P)\}$ (fallas inestables)

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

1. $d(STOP_A) = \emptyset$
2. $d(a \to P)\{\langle a \rangle \frown t \mid t \in d(P)\}$
3. $d(P \Box Q) = d(P) \cup d(Q)$
4. $d(P \sqcap Q) = d(P) \cup d(Q)$
5. $d(SKIP) = \emptyset$
6. $d(P;\ Q) = d(P) \cup \{s \frown t \mid s \frown \langle \checkmark \rangle \in t_\perp(P) \wedge t \in d(Q)\}$ donde $t_\perp(P) = t(P) \cup d(P)$

5. Referencias

- D. Harel, "Statecharts: A visual formalism for complex systems," Science of Computer Programming, vol. 8
- C. A. R. Hoare, Communicating Sequential Processes. Prentice-Hall, 1985.
- R. Allen, "A formal approach to software architecture," Ph.D. dissertation, Carnegie Mellon School of Computer Science, 1997.
- M. Jackson, Software requirements & specifications: a lexicon of practice, principles and prejudices. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1995.
- P. Zave and M. Jackson, "Four dark corners of requirements engineering," ACM Transactions on Software Engineering and Methodology, vol. 6, no. 1, Jan. 1997.
- M. G. Hinchey and S. A. Jarvis, Concurrent systems: formal development in CSP. McGraw-Hill International Series in Software Engineering, 1995.
- A. W. Roscoe, The Theory and Practice of Concurrency. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997