Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Resúmen II

Ingeniería de Software I

Autor:
Arroyo, Joaquín

7 de agosto 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

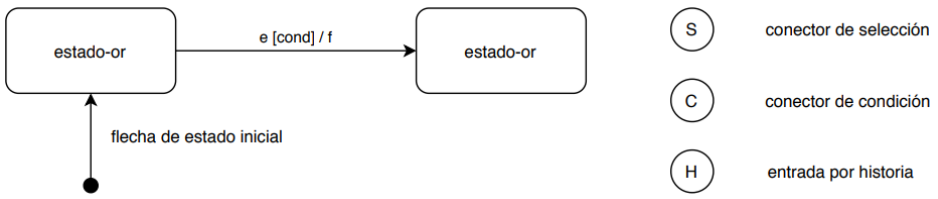
Lenguaje gráfico con semántica ejecutable. Se basa en el formalismo típico para describir FSM, aunque lo extiende de manera brillante. Imposible hablar de tipos.

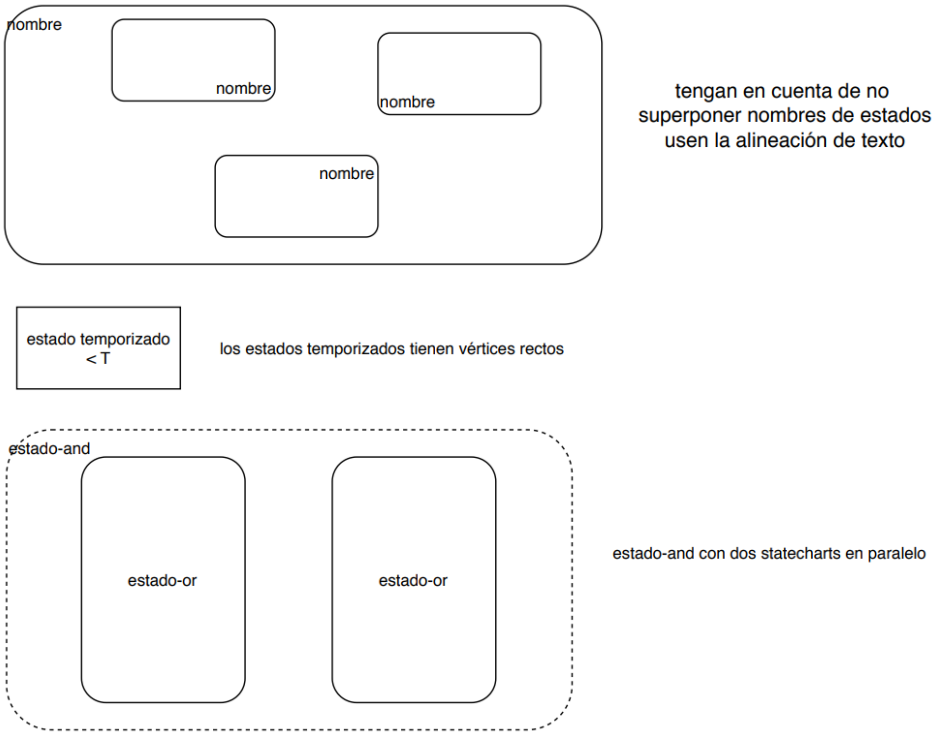
Los conceptos básicos son eventos y estados; a ellos se agregan condiciones para las transiciones, super-estados, concurrencia, temporización, historia, etc.

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:



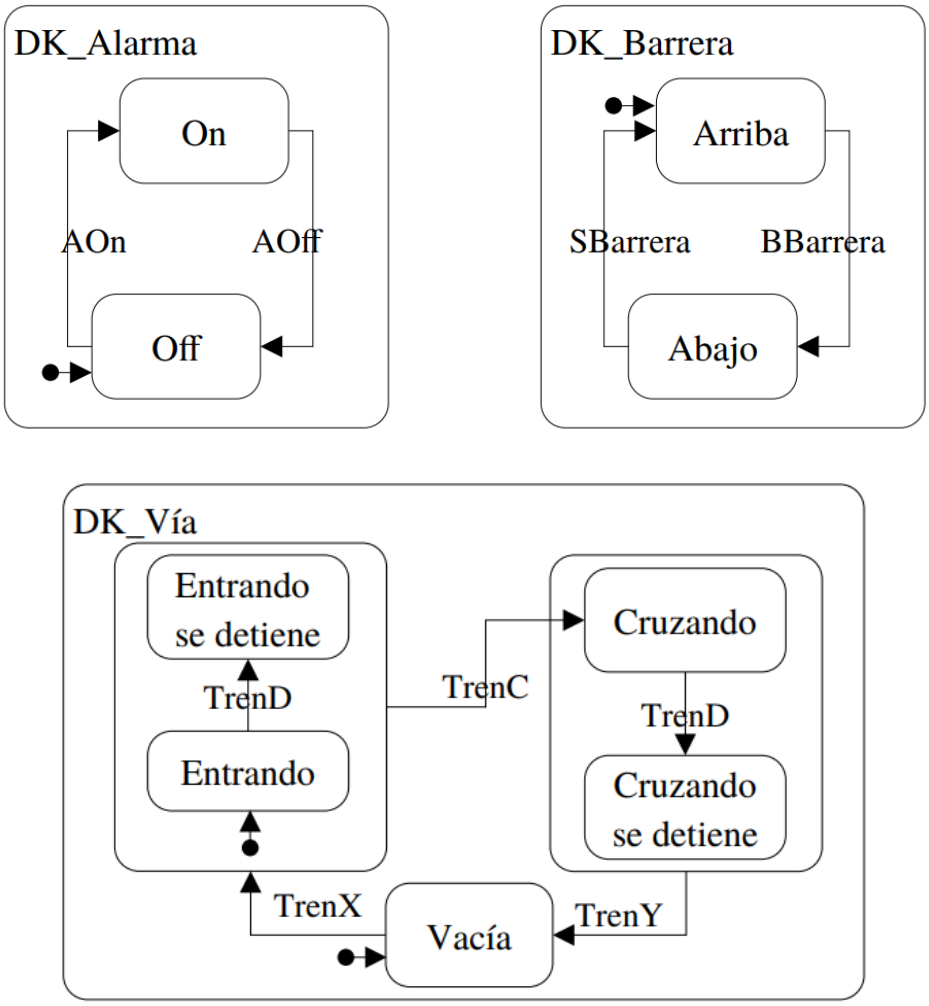


3.3. Ejemplos

Tomemos las siguientes designaciones:

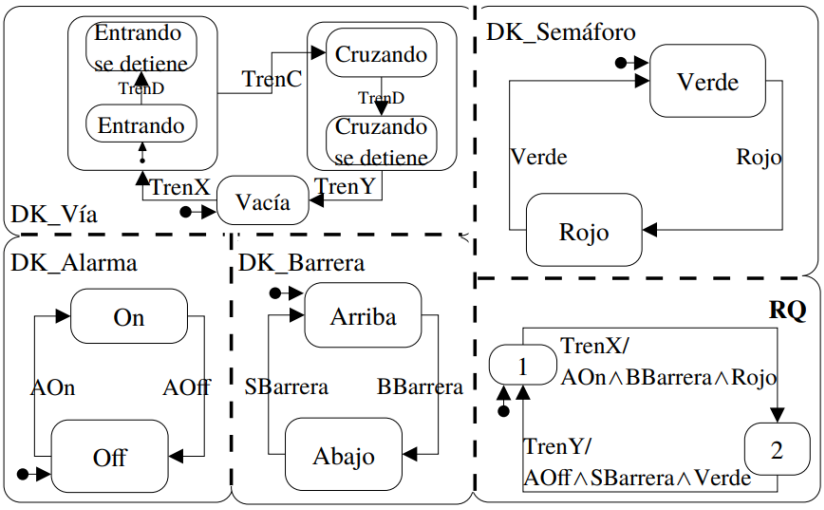
- La distancia de seguridad para bajar la barrera ≈ X
- La distancia de seguridad para subir la barrera ≈ Y
- Un tren cruza la distancia X acercándose al cruce por la vía v ≈ TrenX(v)
- Un tren alcanza la zona de carretera por la vía v ≈ TrenC(v)
- Un tren se detiene en la vía v en la zona de peligro ≈ 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

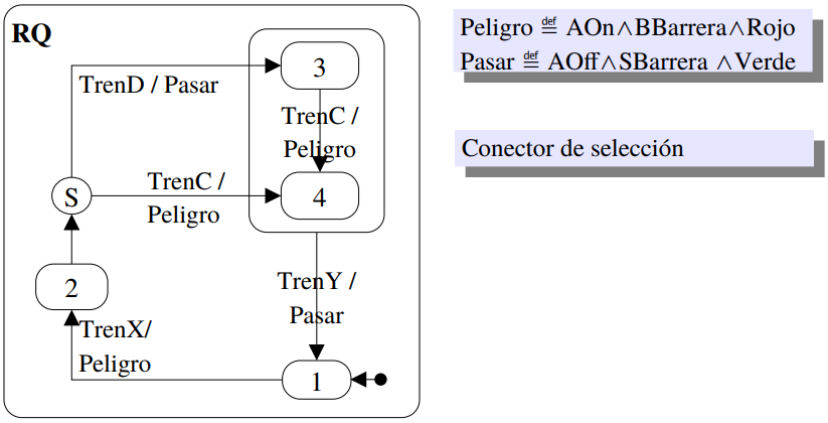

DK_Alarma
On
AOn
AOff
Off
DK_Barrera
Arriba
SBarrera
BBarrera
Abajo
DK_Vía
Entrando se detiene
TrenD
Entrando
TrenC
Cruzando
TrenD
Cruzando se detiene
TrenX
Vacía
TrenY

3.3.2. R

R simple para una sola vía:



R completo:

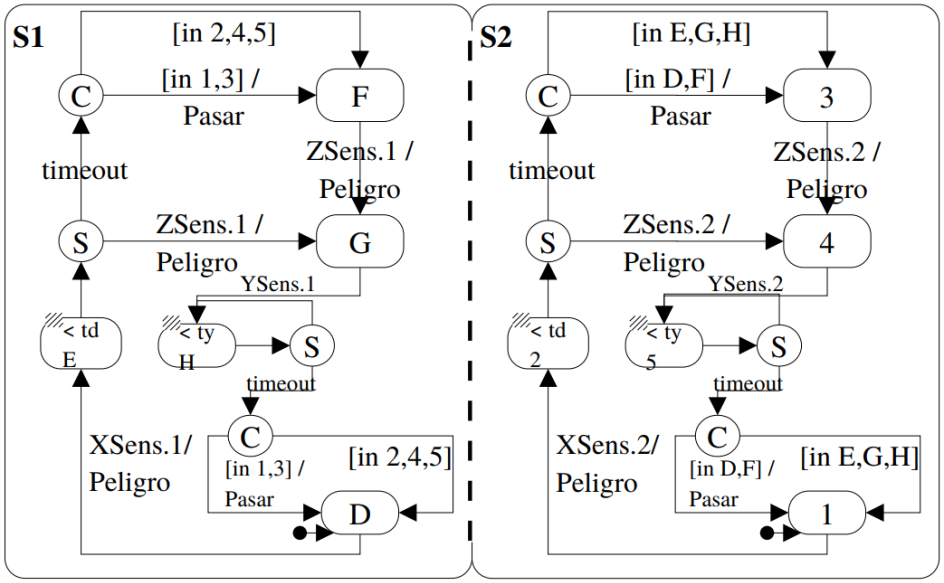


3.3.3. S

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

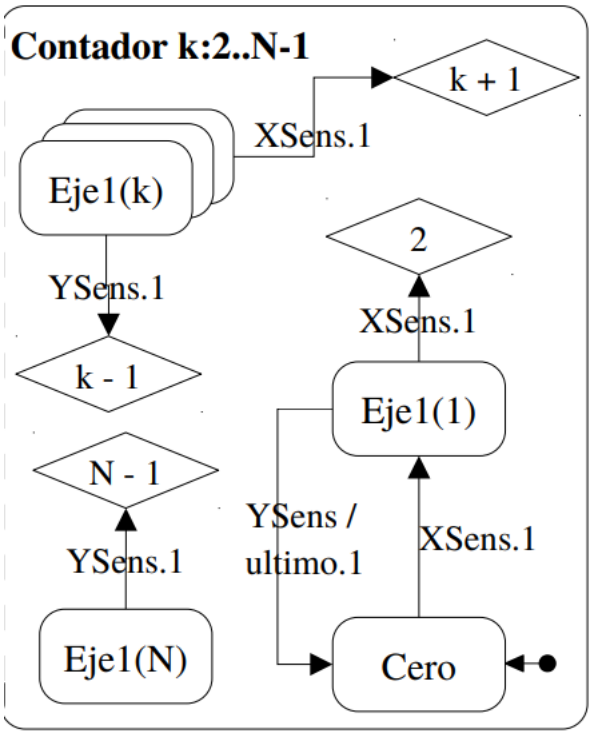
A partir de esto, desarrollamos la especificación S:



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. TLA

5.1. Introducción

Es un lenguaje no tipado, fuertemente orientado a concurrencia. Utiliza teoría de conjuntos y lógica de temporal.

Máquinas de estado: El estado se define por medio de variables. Se definen operaciones (acciones) que modifican ese estado y por lo tanto definen la relación de transición.

La lógica temporal, en principio, permite describir un sistema concurrente con una única fórmula (A. Pneuli, 1977).

Según Lamport, en la práctica, no resulta muy conveniente; por este motivo él introduce TLA.

La mayor parte de TLA no son fórmulas temporales, las cuales sólo aparecen en el momento necesario.

TLA por sí sola es matemática y lógica temporal, es el fundamento teórico del lenguaje.

TLA es algo así como Z sin los esquemas: lógica.

TLA + provee el azúcar sintáctico necesario para escribir especificaciones largas: Modularización, Parametrización e Instanciación.

5.2. Estado

Sea: Val, la colección de todos los valores posibles (de cualquier tipo), excepto TRUE y FALSE. Var, el conjunto de nombres de variables.

Un estado es una función de Var en Val.

Entonces si s es un estado y x es una variable (cualquiera, es decir se haya usado o no para definir s), s(x) es el valor de x en s.

En TLA, un estado puede considerarse un estado del Universo.

5.3. Timer

- El timer debe detenerse o emitir una señal limit unidades de tiempo luego de haber sido iniciado ≈ Set(limit)
- Se enciende el timer ≈ Start
- El timer alcanza el límite de tiempo prefijado ≈ Timeout
- El tiempo real avanza una unidad de tiempo ≈ Tick
- El timer se detiene ≈ Stop

$$
\begin{array}{|l}
\hline
MODULE - DK\_RealTime \\
EXTENDS\ Naturals \\
VARIABLES\ now \\
TypeInv \mathrel{\widehat{=}} now \in Nat \\
\hline
InitRealTime \mathrel{\widehat{=}} now = 0 \\
Tick \mathrel{\widehat{=}} now' = now + 1 \\
RealTimeSpec \mathrel{\widehat{=}} InitRealTime \land \Box[Tick]_{now} \land WF_{now}(Tick) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
MODULE - Timer \\
EXTENDS\ Naturals, DK\_RealTime, RealTime \\
VARIABLES\ running, time, limit \\
TypeInv \mathrel{\widehat{=}} time, limit \in Nat \land running \in \{\overline{yes}, \overline{no}\} \\
sv \mathrel{\widehat{=}} \langle time, running, limit \rangle \\
av \mathrel{\widehat{=}} \langle time, running, limit, now \rangle \\
\hline
InitTimer \mathrel{\widehat{=}} limit = 0 \land running = \overline{no} \\
Set(l) \mathrel{\widehat{=}} running = \overline{no} \land l > 0 \land limit' = l \land UNCHANGED \langle now, time, running \rangle \\
Start \mathrel{\widehat{=}} running = \overline{no} \land limit > 0 \land time' = now \land running' = \overline{yes} \land UNCHANGED \langle now, limit \rangle \\
Timeout \mathrel{\widehat{=}} running = \overline{yes} \land now - time \geq limit \land running' = \overline{no} \land UNCHANGED \langle time, limit, now \rangle \\
Stop \mathrel{\widehat{=}} running = \overline{yes} \land running' = \overline{no} \land UNCHANGED \langle now, time, limit \rangle \\
TimerNext \mathrel{\widehat{=}} Start \lor Stop \lor Timeout \lor (\exists\, l \in \mathbb{N} \bullet Set(l)) \\
TimerSpec \mathrel{\widehat{=}} InitTimer \land \Box[TimeNext]_{sv} \land RTBound(Timeout, av, 0, 1) \\
\hline
THEOREM\ Spec \Rightarrow \Box TypeInv \\
\hline
\end{array}
$$

RTBound es un predicado definido en el módulo *RealTime* de la biblioteca estándar de TLA+ que se usa para especificar restricciones temporales sobre acciones. La signatura del predicado es $RTBound(A, v, \gamma, \epsilon)$ donde A es una acción, v es una o más variables de estado y *gamma* y *epsilon* son números no negativos.

RTBound dice que A debe ejecutarse dentro del intervalo $[\gamma, \epsilon]$ tomado desde la última vez que se ejecutó A siempre y cuando haya estado habilitada todo ese tiempo. En particular, si γ es cero entonces el sistema tiene ϵ unidades de tiempo para ejecutar la acción desde el momento en que queda habilitada. Lamport indica que las cotas γ y ϵ son necesarias porque es imposible medir el tiempo con precisión absoluta.

Entonces, para el caso de *Timeout*, con *RTBound* estamos especificando que se debe ejecutar antes de que pase una unidad de tiempo desde el momento en que queda habilitada, lo que tal vez sea exigir demasiada precisión.

Notar que en la definición de *TimeSpec*, cuando usamos *RTBound* usamos *av* y no *sv* porque de haber incluido *now* en *sv* hubiéramos prohibido el paso del tiempo.

5.4. Timers

$$
\begin{array}{l}
\textit{Module} - \textit{Timers} \\
\textit{EXTENDS Naturals, DK\_RealTime, RealTime} \\
\textit{VARIABLES timers} \\
\textit{TypeInv} \mathrel{\widehat{=}} \textit{timers} \in [\textit{Nat} \to [t, l : \textit{Nat}, r : \{\overline{no}, \overline{yes}\}]] \\
\hline
\textit{Init} \mathrel{\widehat{=}} \textit{timers} = [n \in \textit{Nat} \mapsto [T \mapsto \textit{now}, l \mapsto 0, r \mapsto \overline{no}]] \\
\textit{Set}(i, \textit{lim}) \mathrel{\widehat{=}} \land \textit{lim} > 0 \\
\qquad\qquad \land \textit{timers}[i].r = \overline{no} \\
\qquad\qquad \land \textit{timers}' = [\textit{timers EXCEPT}\ ![i] = [t \mapsto\bullet\ .t, l \mapsto \textit{lim}, r \mapsto\bullet\ .r]] \\
\qquad\qquad \land \textit{UNCHANGED}\langle \textit{now} \rangle \\
\textit{Start}(i) \mathrel{\widehat{=}} \land \textit{timers}[i].r = \overline{no} \\
\qquad\qquad \land \textit{timers}[i].l > 0 \\
\qquad\qquad \land \textit{timers}' = [\textit{timers EXCEPT}\ ![i] = [t \mapsto \textit{now}, l \mapsto\bullet\ .l, r \mapsto \overline{yes}]] \\
\qquad\qquad \land \textit{UNCHANGED}\langle \textit{now} \rangle \\
\textit{Timeout}(i) \mathrel{\widehat{=}} \land \textit{timers}[i].r = \overline{yes} \\
\qquad\qquad\qquad \land \textit{now} - \textit{timers}[i].t \geq \textit{timers}[i].l \\
\qquad\qquad\qquad \land \textit{timers}' = [\textit{timers EXCEPT}\ ![i] = [t \mapsto\bullet\ .t, l \mapsto\bullet\ .l, r \mapsto \overline{no}]] \\
\qquad\qquad\qquad \land \textit{UNCHANGED}\langle \textit{now} \rangle \\
\textit{Stop}(i) \mathrel{\widehat{=}} \land \textit{timers}[i].r = \overline{yes} \\
\qquad\qquad \land \textit{timers}' = [\textit{timers EXCEPT}\ ![i] = [t \mapsto\bullet\ .t, l \mapsto\bullet\ .l, r \mapsto \overline{no}]] \\
\qquad\qquad \land \textit{UNCHANGED}\langle \textit{now} \rangle \\
\textit{Next} \mathrel{\widehat{=}} \exists\, i \in \textit{Nat} : \textit{Start}(i) \lor \textit{Stop}(i) \lor \textit{Timeout}(i) \lor (\exists\, t \in \textit{Nat} : \textit{Set}(i, t)) \\
\textit{Spec} \mathrel{\widehat{=}} \textit{Init} \land \Box[\textit{Next}]_{\textit{timers}} \land (\forall\, i \in \textit{Nat} : \textit{RTBound}(\textit{Timeout}(i), \langle \textit{timers}, \textit{now} \rangle, 0, 1)) \\
\hline
\textit{THEOREM Spec} \Rightarrow \Box \textit{TypeInv}
\end{array}
$$

5.5. Super Timer/s

- Se setea el timer con *limit* y se enciende ≈ SetStart
- Se hace stop y start del timer ≈ Restart

$$
\begin{array}{l}
\textit{MODULE} - \textit{SuperTimer} \\
\textit{EXTENDS Timer} \\
\textit{VARIABLES start} \\
\textit{STypeInv} \mathrel{\widehat{=}} \textit{TypeInv} \land \textit{start} \in \textit{Boolean} \\
\textit{vars}1 \mathrel{\widehat{=}} \textit{sv} \frown \langle \textit{start} \rangle \\
\textit{vars}2 \mathrel{\widehat{=}} \textit{av} \frown \langle \textit{start} \rangle \\
\hline
\textit{Init} \mathrel{\widehat{=}} \textit{start} = \textit{False} \\
\textit{SetStart}(l) \mathrel{\widehat{=}} \bar{}\textit{start} \land \textit{Set}(l) \land \textit{start}' = \textit{True} \\
\textit{Restart} \mathrel{\widehat{=}} \bar{}\textit{start} \land \textit{Stop} \land \textit{start}' = \textit{True} \\
\textit{CheckStart} \mathrel{\widehat{=}} \textit{start} \land \textit{Start} \land \textit{start}' = \textit{False} \\
\textit{Next} \mathrel{\widehat{=}} \textit{Restart} \lor \textit{CheckStart} \lor (\exists\, l : \textit{Nat} \bullet \textit{SetStart}(l)) \\
\textit{Spec} \mathrel{\widehat{=}} \textit{Init} \land \Box[\textit{Next}]_{\textit{vars}1} \land \textit{WF}(\textit{CheckStart})_{\textit{vars}2} \\
\hline
\textit{THEOREM Spec} \Rightarrow \Box \textit{STypeInv}
\end{array}
$$

- Se setea el timer i con limite l y se enciende ≈ $\textit{SetStart}(i, l)$

- Se hace stop y start del timer $i \approx$ Restart(i)

$MODULE - SuperTimers$

$EXTENDS\ Timers, Sequences$
$VARIABLES\ starts$
$STypeInv \triangleq starts \in Seq\ Nat$
$vars \triangleq \langle timers, starts \rangle$

$SInit \triangleq starts = \langle\rangle$
$SetStart(i, l) \triangleq \barwedge(i \text{ in } starts)$
$\quad \wedge Set(i, l)$
$\quad \wedge starts' = starts \frown \langle i \rangle$
$Restart(i) \triangleq \barwedge(i \text{ in } starts)$
$\quad \wedge Stop(i)$
$\quad \wedge starts' = starts \frown \langle i \rangle$
$CheckStart \triangleq \wedge starts \neq \langle\rangle$
$\quad \wedge Start(head(start))$
$\quad \wedge starts' = tail(starts)$
$SafeStart(i) \triangleq starts' = starts \frown \langle i \rangle$
$SNext \triangleq (\exists\, i, l : Nat \bullet SetStart(i, l) \vee Restart(i) \vee SafeStart(i)) \vee CheckStart$
$SSpec \triangleq SInit \wedge \Box[SNext]_{vars} \wedge WF(CheckStart)_{vars}$

$THEOREM\ SSPec \Rightarrow \Box STypeInv$

5.6. Módulos

La unidad de especificación de TLA+ son los módulos; llevan un nombre para futuras referencias. Un módulo puede incluir:

- variables: VARIABLE, VARIABLES
- constantes: CONSTANT, CONSTANTS
- hipótesis sobre las constantes: ASSUME
- definiciones: $\triangleq$
- teoremas: THEOREM
- otros módulos. EXTENDS, INSTANCE

5.6.1. EXTENDS e INSTANCE

EXTENDS permite utilizar todas las definiciones efectuadas en los módulos que se extienden. Excepto BOOLEAN, todos los demás "tipos"deben ser incluidos usando esta cláusula.

$T1 \triangleq INSTANCE\ Timer\ WITH\ time \leftarrow t1, limit \leftarrow Tp, running \leftarrow r1$

Define una instancia de Timer donde *time*, *Limit* y *running* son renombradas; $t1$, Tp y $r1$ deben estar definidas. De esta forma, $T1!Timeout$ es:

$r1 = \overline{yes} \wedge now - t1 \geq Tp \wedge r1' = \overline{no} \wedge UNCHANGED\langle t1, Tp, now \rangle$

5.6.2. VARIABLE

Esta cláusula se utiliza para definir las variables de estado que interesan al módulo.

Las variables no tienen tipo. Esto significa que si definimos la variable n, luego podemos escribir cosas como $n = Tail(s)$ o $n = 5$.

Sin embargo, es común que se incluya un predicado sobre los valores posibles que pueden tomar las variables y luego se 'obligue' al especificador a probar que ese predicado es un invariante. Como por ejemplo: $VARIABLES\ now$, $TypeInv \triangleq now \in Nat$

5.6.3. Definiciones

Las definiciones permiten estructurar la lógica que se utiliza dentro de un módulo.

El nombre de una definición se puede utilizar en otras definiciones; y como vimos, las definiciones de un módulo pueden accederse desde otros.

Las definiciones pueden ser paramétricas:

$Set(l) \triangleq running = \overline{no} \wedge l > 0 \wedge limit' = l \wedge UNCHANGED\langle now, realtime, running\rangle$

5.6.4. Acciones

Las acciones son las operaciones del sistema. Formalmente, una acción es un predicado que depende de dos estados; por tanto, contiene variables primadas y no primadas.

Si A es una acción y s y t son dos estados tales que $s[\![A]\!]t$ entonces el par (s, t) es un paso-A.

• $s[\![A]\!]t \triangleq A(\forall\, v : s(v)/v, t(v)/v'$

Si $\langle s_0, s_1, ...\rangle$ es una ejecución, se define el valor de verdad de la acción de A como: $\langle s_0, s_1, ...\rangle[\![A]\!] \triangleq s_0[\![A]\!]s_1$

Las acciones son atómicas. Es decir, no existen estados ntermedios entre el estado de partida y el de llegada de una acción.

Sean A y B acciones que se inician en s y finalizan en t y v, respectivamente. Entonces, el sistema pasa de s a t o de s a v, pero no ambos ni a ningún otro.

Esto significa que si se espera que las ejecuciones del sistema verifiquen cierta propiedad, habrá que considerar ambas.

5.7. *Interleaving model of concurrency*

La semántica de TLA está orientada a abordar el problema de la concurrencia por medio del IMC.

Dado un estado s y n acciones que pueden iniciarse en s, se deben considerar las n ejecuciones que se originan partiendo de s.

Si se espera que una especificación verifique cierta propiedad, deben tenerse en cuenta todas las ejecuciones producto de intercalar de todas las formas posibles la ejecución de sus acciones.

5.8. La forma de una especificación

En TLA las especificaciones tiene una forma sintáctica restringida: $Init \wedge \Box[Next]_{vars} \wedge Fairness_{vars}$ donde:

- *Next* es por lo general la disyunción de las acciones permitidas en el sistema.
- *vars* son todas las variables del sistema.
- *Fairness* es una conjunción numerable de fórmulas de WF y/o SF.

Esta sintaxis obedece a seis criterios:

- Sirve para la mayoría de los sistemas.
- Dice que el primer estado de cualquier ejecución debe verificar Init, y cualquier par de estados de una ejecución deben estar relacionados por una de lasacciones definidas.
- Las acciones controladas por la máquina se ejecutarán equitativamente.
- Siempre se obtienen máquinas cerradas.
- La fórmula resultante es invariante con respecto a los pasos intrascendentes.
- Respeta el teorema de Alpern-Schneider.

5.8.1. $Fairness_{vars}$

Debe ser una conjunción numerable de fórmulas de la forma:

- $WF_{vars}(A) \mathrel{\widehat{=}} \Box(\Box ENABLED\langle A\rangle_{vars} \Rightarrow \langle A\rangle_{vars})$
- $SF_{vars}(A) \mathrel{\widehat{=}} \Box ENABLED\langle A\rangle_{vars} \Rightarrow \Box\langle A\rangle_{vars}$

Donde: A es una subacción de $Next$, es decir $\Rightarrow Next$, y $\langle A\rangle_{vars}$ es igual a $\bar{}\Box[\bar{}A]_{vars}$

5.9. Regla de conjunción para WF

Si A_1, ... A_n son acciones tales que, para $i \neq j$, siempre que la acción A_i está habilitada, A_j no puede habilitarse hasta que se de un $paso - A_i$, entonces $WF_v(A_1) \wedge ... \wedge WF_v(A_n)$ es equivalente a $WF_v(A1 \vee ... \vee A_n)$

Como la conjunción es una forma particular de la cuantificación universal, existe una regla similar para una cantidad numerable de acciones.

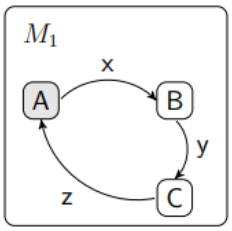
Hay una regla igual para SF.

6. Conceptos avanzados: Seguridad, Vitalidad y Equidad

6.1. Pasos de ejecución repetitivos

El concepto de pasos de ejecución repetitivos (stuttering steps) está relacionado con la idea de dar la especificación de un sistema como la composición de las especificaciones de sub-sistemas o componentes.

Consideremos el sistema descripto por la siguiente máquina de estados (donde el estado en gris es el estado inicial)

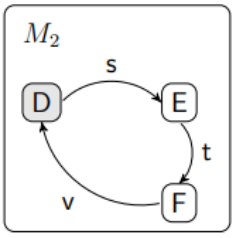


Definimos la semántica de este sistema como el conjunto de todas las sucesiones infinitas de estados que surgen de ejecutar las transiciones. Si el sistema permanece infinito tiempo en un estado (porque la transición de salida no se da nunca) este se repite al infinito en la sucesión. A una sucesión infinita de estados la llamaremos ejecución o comportamiento.

Por consiguiente, las siguientes son algunas de las ejecuciones de M_1 (donde $\hat{\cdot}$ significa que $\cdot$ se repite al infinito):

$$\langle \widehat{\mathsf{A}} \rangle \tag{1}$$
$$\langle \mathsf{A}, \widehat{\mathsf{B}} \rangle \tag{2}$$
$$\langle \mathsf{A}, \mathsf{B}, \widehat{\mathsf{C}} \rangle \tag{3}$$
$$\langle \mathsf{A}, \mathsf{B}, \mathsf{C}, \widehat{\mathsf{A}} \rangle \tag{4}$$
$$\langle \mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{A}, \widehat{\mathsf{B}} \rangle \tag{5}$$

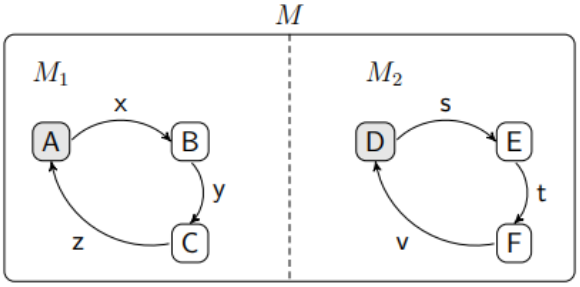
Ahora consideremos el sistema descripto por M_2:



algunas de cuyas ejecuciones son:

$$\langle \widehat{\mathsf{D}} \rangle \tag{6}$$
$$\langle \mathsf{D}, \widehat{\mathsf{E}} \rangle \tag{7}$$
$$\langle \mathsf{D}, \mathsf{E}, \widehat{\mathsf{F}} \rangle \tag{8}$$
$$\langle \mathsf{D}, \mathsf{E}, \mathsf{F}, \widehat{\mathsf{D}} \rangle \tag{9}$$
$$\langle \mathsf{D}, \mathsf{E}, \mathsf{F}, \mathsf{D}, \widehat{\mathsf{E}} \rangle \tag{10}$$

Finalmente, consideremos el sistema M que resulta de componer en paralelo M_1 y M_2.



algunas de cuyas ejecuciones son:

$$\langle \widehat{(\mathsf{A}, \mathsf{D})} \rangle \tag{11}$$
$$\langle (\mathsf{A}, \mathsf{D}), \widehat{(\mathsf{B}, \mathsf{D})} \rangle \tag{12}$$
$$\langle (\mathsf{A}, \mathsf{D}), (\mathsf{B}, \mathsf{D}), \widehat{(\mathsf{B}, \mathsf{E})} \rangle \tag{13}$$
$$\langle (\mathsf{A}, \mathsf{D}), (\mathsf{B}, \mathsf{D}), (\mathsf{B}, \mathsf{E}), \widehat{(\mathsf{C}, \mathsf{E})} \rangle \tag{14}$$

Consideremos ahora la proyección de la ejecución (14) a M_1:

$$\mathcal{P}_{M_1}(\langle (\mathsf{A}, \mathsf{D}), (\mathsf{B}, \mathsf{D}), (\mathsf{B}, \mathsf{E}), \widehat{(\mathsf{C}, \mathsf{E})} \rangle) = \langle \mathsf{A}, \mathsf{B}, \mathsf{B}, \widehat{\mathsf{C}} \rangle \tag{15}$$

El problema con (15) es que no es una de las ejecuciones admitidas por M_1 puesto que B se repite un número finito de veces en el medio de la ejecución (mientras que en todas las ejecuciones de M_1, si B se repite, lo hace infinitas veces y al final de la ejecución). Por lo tanto, el comportamiento de M contempla ejecuciones cuyas proyecciones resultan en ejecuciones que el componente correspondiente no contempla. En otras palabras, si tratamos de descubrir los componentes que componen M mirando sus ejecuciones, no llegaremos a las especificaciones de M_1 y M_2.

Los pasos repetitivos (stuttering steps) permiten evitar esta inconsistencia. Entonces, definimos una ejecución de un cierto sistema M como el conjunto de sucesiones infinitas de estados que surgen de ejecutar las transiciones pero donde se admiten repeticiones finitas de estados llamadas pasos repetitivos. O sea que para la máquina M_1 las ejecuciones (1)-(5) son (donde $\cdot^*$ significa que $\cdot$ puede repetirse un número finito de veces):

$$\langle \widehat{\mathsf{A}} \rangle \tag{16}$$
$$\langle \mathsf{A}^*, \widehat{\mathsf{B}} \rangle \tag{17}$$
$$\langle \mathsf{A}^*, \mathsf{B}^*, \widehat{\mathsf{C}} \rangle \tag{18}$$
$$\langle \mathsf{A}^*, \mathsf{B}^*, \mathsf{C}^*, \widehat{\mathsf{A}} \rangle \tag{19}$$
$$\langle \mathsf{A}^*, \mathsf{B}^*, \mathsf{C}^*, \mathsf{A}^*, \widehat{\mathsf{B}} \rangle \tag{20}$$

aunque normalmente los pasos repetitivos no se explicitan sino que se utiliza la notación usada en (1)-(5) donde se asume que cada estado puede repetirse un número finito de veces.

6.2. El concepto de estado

Supongamos que el sistema M_1 depende de la variable de estado x que toma valores en N. La única transición permitida en M_1 es $TM_1 \mathrel{\widehat{=}} x' = x + 1$ y el estado inicial de M_1 es $x = 0$. Entonces las ejecuciones de M_1 pueden caracterizarse de la siguiente forma:

$\langle x = 0, x = 1, x = 2, ... \rangle$

Ahora supongamos que tenemos otro sistema M_2 que depende de la variable de estado y que toma valores en N. La única transición permitida en M_2 es $TM_2 \mathrel{\widehat{=}} y' = y + 2$ y el estado inicial de M_2 es $y = 0$. Entonces las ejecuciones de M_2 pueden caracterizarse de la siguiente forma:

$\langle y = 0, y = 2, y = 4, ... \rangle$

Finalmente supongamos que el sistema M es el resultado de componer M_1 con M_2 de forma tal que la única transición permitida es $TM \mathrel{\widehat{=}} TM_1 \vee TM_2$. Entonces una de las ejecuciones típicas de M puede caracterizarse de la siguiente forma:

$\langle (x = 0, y = 0), (x = 1, y = 5), (x = 10, y = 3), ... \rangle$

puesto que entre $(x = 0, y = 0)$ y $(x = 1, y = 5)$ se ejecutó TM_1 la cual establece el valor de x pero no establece el valor de y por lo que el paso es perfectamente compatible con TM; y entre $(x = 1, y = 5)$ y $(x = 10, y = 3)$ se ejecutó TM_2 la cual establece el valor de y pero no establece el valor de x por lo que el paso es perfectamente compatible con TM. Claramente este tipo de comportamientos no son los que teníamos en mente cuando dimos la especificación de TM.

La especificación de TM que en realidad queremos es esta:

$TM \mathrel{\widehat{=}} (TM_1 \wedge y' = y) \vee (TM_2 \wedge x' = x)$

pero el problema es que ni $TM_1 \wedge y' = y$ es una especificación para la máquina M_1 ni $TM_2 \wedge x' = x$ lo es para M_2 dado que M_1 no predica sobre y ni M_2 sobre x.

Por este motivo Lamport define el concepto de estado de la siguiente forma. Sea Var el conjunto de todos los nombres posibles de variables y sea Val la colección de todos los valores que las variables pueden tomar (notar que algunos elementos de Val son $1, (2, 3), 4, a, 6, N$, etc.). Entonces el estado s se define como una función de Var en Val: $s : Var \to Val$.

Es decir, que si s es un estado del sistema M_1 o de M_2 o de M, tanto $s(x)$ como $s(y)$ están definidas. Por lo tanto, si:

$$e(0)(x) = 0 \wedge e(1)(x) = 1 \wedge e(2)(x) = 2 \wedge \ldots \tag{21}$$

y lo es de M_2 sii:

$$e(0)(y) = 0 \wedge e(1)(y) = 2 \wedge e(2)(y) = 4 \wedge \ldots \tag{22}$$

Definir el concepto de estado de esta forma surge de la necesidad de poder especificar un sistema como la composición de las especificaciones de sub-sistemas o componentes. De esta forma, cualquier estado depende potencialmente de todas las variables posibles solo que cada componente especifica ciertas condiciones para algunas de las variables. Podemos decir que al definir el concepto de estado de esta forma estamos definiendo el estado de un universo discreto.

6.3. El teorema de Alpern-Schneider

De aquí en más consideraremos que una propiedad de un sistema concurrente es un conjunto de ejecuciones. De la misma forma, como ya vimos, un sistema concurrente queda caracterizado por el conjunto de ejecuciones que admite. En consecuencia decimos que un sistema M verifica una propiedad P sii $M \subseteq P$ (es decir, si el conjunto de ejecuciones que caracteriza a M está incluido o es igual al conjunto de ejecuciones que define a la propiedad P).

Las siguientes son algunos ejemplos de propiedades que se pueden expresar como conjuntos de ejecuciones: ausencia de abrazo mortal (deadlock freedom), corrección parcial (si un programa termina, verifica su especificación), exclusión mutua, primero en llegar primero en ser atendido, ausencia de inanición (starvation freedom), terminación (o sea parte de la corrección total) y garantía de servicio. En 1985 Bruce Alpern y Fred Schneider, en su artículo "Defining Liveness", postulan y demuestran el siguiente teorema:

Teorema 1 Toda propiedad P es el resultado de la intersección de una propiedad de seguridad (safety) con una propiedad de vitalidad (liveness).

Como los sistemas también están caracterizados como conjuntos de ejecuciones, el teorema implica que todo sistema puede expresarse como la intersección de una propiedad de seguridad (safety) con una propiedad de vitalidad (liveness). Por lo tanto, a continuación veremos con cierto detalle las propiedades de seguridad y vitalidad.

6.3.1. Seguridad

Las propiedades de seguridad postulan que nada malo puede pasar durante una ejecución del sistema. Toda propiedad de seguridad proscribe o prohíbe algo 'malo'. Algunos ejemplos de propiedades de seguridad son: ausencia de abrazo mortal, corrección parcial, exclusión mutua y primero en llegar primero en ser atendido. Entonces:

- En exclusión mutua lo 'malo' es dos procesos ejecutando en la sección crítica al mismo tiempo.
- En ausencia de abrazo mortal lo 'malo' es que haya abrazo mortal,
- En corrección parcial lo 'malo' es que el programa termine en un estado que no satisface la post-condición habiendo comenzado a ejecutar en un estado que satisfacía la pre-condición.
- En primero en llegar primero en ser atendido, lo 'malo' es atender un pedido antes que otro que llegó primero.

Usualmente la forma más fácil de dar una propiedad de seguridad es indicando solo las transiciones de estado que están permitidas. O sea que una propiedad de seguridad normalmente se da indicando solo los comportamientos permitidos y de esta forma, implícitamente, se prohíben todos los demás.

En este sentido, una propiedad de seguridad no indica que el sistema hará algo; solo indica lo que tiene permitido hacer. Hasta el momento, todo lo que hemos hecho en este curso es especificar propiedades de seguridad sin saberlo (por ejemplo, una especificación Z o un modelo Statecharts, normalmente, indican solo la propiedad de seguridad que se espera verifique el sistema que se está especificando). Lamport dice que normalmente el 90 % del esfuerzo de especificación está dedicado a propiedades de seguridad.

El primero en formalizar el concepto de seguridad fue Lamport. Sin embargo, antes de dar la definición formal de seguridad introducimos la siguiente notación.

Notación Sea E un conjunto de estados. $E*$ denota el conjunto de todas las sucesiones finitas de elementos de E (llamadas ejecuciones parciales), y E^∞ el de todas las sucesiones infinitas (o sea las ejecuciones). Si $e \in E^\infty$ y $n \in \mathbb{N}$ entonces, e_n denota el prefijo de e hasta n y $e(n)$ el n-ésimo elemento de la sucesión (notar que el primer elemento es $e(0)$). Es decir, $e_n = \langle e(0), e(1), ..., e(n)\rangle$. Si e y t son dos sucesiones de estados ($e \in E*$), et denota la concatenación de e con t.

Definición 1 S es una propiedad de seguridad respecto de E sii se verifica lo siguiente: $e \notin S$ *sii* $\exists\, n \in \mathbb{N} : \forall\, t \in E^\infty : e_n \circ t \notin S$ para toda $e \in E^\infty$.

Observar que esta definición define una propiedad de seguridad por las ejecuciones que no pertenecen a ella. Es decir, la propiedad de seguridad es el complemento (respecto de E^∞) de las ejecuciones

que verifican la Definición 1. Precisamente, e no pertenece a S cuando en un punto (n) lo 'malo' prohibido por S tuvo lugar y en consecuencia esto no se puede remediar (lo 'malo' ocurrió y ya no se puede volver atrás). En otras palabras, haga lo que haga el sistema luego de n (simbolizado por t) no podrá remediar el hecho de que lo 'malo' ya ocurrió en e. Por ejemplo, si S es "primero en llegar primero en ser atendido" y en e hay un pedido que es servido primero que otro que llegó antes, entonces hay un punto en e donde esto ocurrió y cualquier cosa que siga en n o podrá ocultar este hecho, por lo tanto $e \notin S$.

A raíz de la existencia de un punto preciso a partir del cual la ejecución no cumple con S, Alpern y Schneider dicen que las propiedades de seguridad son discretas en el sentido de que hay un punto preciso donde se puede ver la violación de la propiedad. Además, de la Definición 1 surge que una propiedad de seguridad jamás puede estipular que algo eventualmente ocurrirá, solo que algo nunca ocurrirá.

6.3.2. Vitalidad

Las propiedades de vitalidad estipulan que algo 'bueno' ocurrirá durante la ejecución del sistema. Toda propiedad de vitalidad asegura que algo eventualmente ocurrirá en el sistema. Algunos ejemplos de propiedades de vitalidad son: ausencia de inanición, terminación y garantía de servicio. Entonces:

- En ausencia de inanición (que establece que un proceso progresa infinitamente a menudo) lo 'bueno' es que el proceso progresa.
- En terminación lo 'bueno' es ejecutar la ´ultima instrucción.
- En garantía de servicio lo 'bueno' es que el proceso es atendido.

Una propiedad de vitalidad dice que ninguna ejecución parcial es irremediable: siempre existe la posibilidad de que más adelante la ejecución cumpla con la propiedad. Es decir que una propiedad de vitalidad indica los estados que obligatoriamente deberán alcanzarse durante una ejecución.

Jackson señala que las propiedades de vitalidad del tipo $MC - S - DK$ no parecen ser posibles en ningún sistema (aunque sí son posibles las de seguridad de este tipo). (Piense en una justificaci´on para esta conjetura de Jackson).

Definición 2 L es una propiedad de vitalidad respecto de E sii se verifica lo siguiente: $\forall\, e \in E* : \exists\, t \in E^{\infty}\, e \circ t \in L$

Es decir que L es una propiedad de vitalidad si toda ejecución parcial puede ser extendida a una ejecución donde lo 'bueno' estipulado por L ocurre. En otras palabras, un sistema verifica L de vitalidad si toda ejecución parcial del sistema puede ser extendida a una ejecución donde lo estipulado por L se da. Notar que al contrario de las propiedades de seguridad, no necesariamente hay un punto preciso donde la propiedad de vitalidad se haga verdadera. Más aun, ciertas propiedades de vitalidad requieren que muchos estados la verifiquen (por ejemplo en ausencia de inanición el proceso debe avanzar infinitamente a menudo por lo que debe haber infinitos estados donde cierta condición se cumple). Es decir que las propiedades de vitalidad no son discretas según la idea comentada en la sección anterior. En este sentido 'bueno' y 'malo' parecen ser conceptos fundamentalmente diferentes. Además, una propiedad de vitalidad no estipula que algo 'bueno' siempre ocurre, solo indica que eso eventualmente ocurrirá. En general hay un acuerdo universal sobre la definición de seguridad; no así con la definición de vitalidad (por lo menos al momento en que Alpern y Schneider publicaron su famoso artículo). Entonces cabe preguntarse, ¿es razonable la definición de vitalidad? Supongamos que L es de vitalidad pero no verifica la Definición 2. Entonces existe una ejecución parcial e tal que:

$\forall\, t \in E^{\infty} : et \notin L$

lo cual sugiere que e es algo prohibido por L por lo que L es, al menos en parte, de seguridad (comparar con la Definición 1). Es decir que, al relajar la Definición 2 se obtienen propiedades que prohíben cosas lo que es el dominio de las propiedades de seguridad. Por lo tanto, parecería darse que si una propiedad no verifica la definición dada es en parte de seguridad. Entonces se considera que la Definición 2 incluye a todas las propiedades (puramente) de vitalidad.

6.4. El concepto de máquina cerrada

Según el teorema de Alpern-Schneider la especificación de un sistema debería estar dada por una propiedad de seguridad y otra de vitalidad, donde la propiedad de seguridad restringe el compor-

tamiento del sistema a aquellas ejecuciones que son seguras. Sin embargo, consideremos el siguiente sistema:

1. El estado del sistema está dado por la variable x que toma valores en $\mathbb{N}$.
2. El estado inicial es $x = 0$.
3. Propiedad de seguridad: la única transición posible es $x' = x + 1$.
4. Propiedad de vitalidad: si en algún estado x vale 1 entonces eventualmente se debe alcanzar un estado donde x valga 0.

Por consiguiente, la única ejecución posible de este sistema es $\widehat{x = 0}$. Es decir, el sistema permanecerá siempre en el estado inicial puesto que si la transición permitida se ejecutara, x valdría 1 pero no habría forma de cumplir con la propiedad de vitalidad. Entonces la única forma de cumplir con las dos propiedades es permaneciendo en el estado inicial.

Este ejemplo demuestra que no necesariamente todas las restricciones están dadas por la propiedad de seguridad. Según Abadi y Lamport este problema se origina al permitir propiedades de vitalidad arbitrarias. Por lo tanto ellos estudiaron el problema de encontrar propiedades de vitalidad que no agregaran restricciones al sistema que no estuvieran dadas en su propiedad de seguridad. El primer paso fue definir el concepto de máquina cerrada para el cual es preciso definir primero la clausura de una propiedad.

Definición 3 Si A es una propiedad cualquiera, $C(A)$, denominada clausura de A, es la menor propiedad de seguridad que contiene a A.

Definición 4 Si S es una propiedad de seguridad y L es una propiedad cualquiera, entonces el par (S, L) es una máquina cerrada sii $C(S \cap L) = S$

Observar que si (S, L) es máquina cerrada, todas las restricciones del sistema definido por $S \cap L$ están dadas por S ya que no existe S' de seguridad tal que $S \cap L \subseteq S' \subset S$. Si tal S' existiera, querría decir que L agrega restricciones.

Abadi y Lamport demostraron que si las propiedades de vitalidad se escriben como propiedades de equidad entonces se obtienen máquinas cerradas.

6.5. Equidad

Las propiedades de equidad (fairness) son una clase de propiedades de vitalidad que tienen una forma muy particular. Para definir formalmente las propiedades de equidad necesitamos introducir algunos conceptos simples de lógica temporal.

Sea P un predicado de estado (es decir un predicado que depende de las variables de estado), entonces $\Box P$ (que se lee 'siempre P') es un predicado de la lógica temporal que depende de una ejecución:

$\Box P$ es verdadero en $e \in E^\infty$ sii $\forall\, i \in \mathbb{N} : P(e(i))$

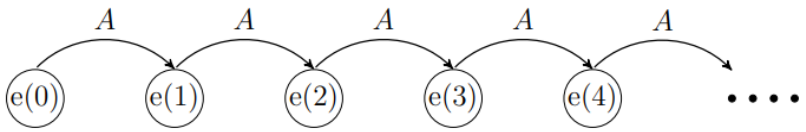
Esta es una forma temporal de expresar que P es un invariante de estado en la ejecución e.

Sea A un predicado que depende de dos estados (es decir A es una transición de estados), entonces A es un predicado de la lógica temporal que depende de una ejecución:

$\Box A$ es verdadero en $e \in E^\infty$ sii $\forall\, i \in \mathbb{N} : A(e(i), e(i+1))$

En este caso el sistema pasa del estado $e(i)$ al $e(i+1)$ (para cualquier i) siempre debido a A. Cuando el sistema pasa del estado $e(i)$ al $e(i+1)$ debido a A decimos que se da un paso-A ($A - step$).

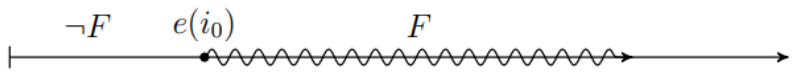
Gráficamente $\Box A$ se puede representar de la siguiente forma:



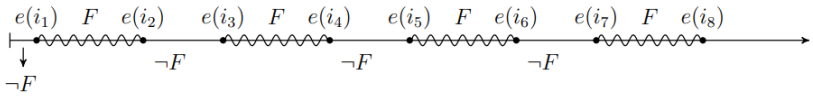
Si F es un predicado cualquiera, $\diamond F$ se define como $\bar{}\Box\bar{}F$ y se lee 'eventualmente F'. Precisamente:
$\diamond F$ es verdadero en $e \in E^\infty$ sii $\exists\, i \in \mathbb{N} : F(e(i))$, o $F(e(i), e(i+1))$ si F es una transición.

Hay dos combinaciones de $\Box$ y $\diamond$ que son importantes para definir las propiedades de equidad:

- $\diamond\Box F$ se lee 'eventualmente siempre F' y es verdadero si existe un estado a partir del cual F es siempre verdadero. Gráficamente:



- $\Box \diamond F$ se lee 'infinitamente a menudo F' y es verdadero si existen infinitos estados donde F es verdadero (por ejemplo todos los estados con ´ındice par o todos los estados con índice terminado en 0, etc.). Gráficamente se puede expresar de la siguiente forma donde se asume que los sectores donde F vale se repiten al infinito:



Si A es una transición de estados, el predicado $enabled(A)$ es verdadero cuando la precondición de A se satisface (es decir, cuando es posible ejecutar la transición A o cuando A está habilitada para ejecutar).

Con todos estos elementos podemos dar la definición de las propiedades de equidad. Existen dos versiones de equidad: débil (weak fairness) y fuerte (strong fairness).

Definición 5 La transición de estados A verifica weak fairness sii: $WF(A) \mathrel{\widehat{=}} \diamond\Box enabled(A) \Rightarrow \Box\diamond A$.

Definición 6 La transición de estados A verifica strong fairness sii: $SF(A) \mathrel{\widehat{=}} \Box \diamond enabled(A) \Rightarrow \Box \diamond A$.

Observar que la diferencia entre las dos fórmulas se da en el antecedente: en WF se pide que a partir de cierto punto A esté siempre habilitada mientras que en SF es suficiente con que A esté habilitada en infinitos estados aunque puede no estarlo de forma continua. En cualquiera de los dos casos se exige que A sea ejecutada infinitas veces.

$WF(A)$ y $SF(A)$ no son equivalentes cuando existe algún evento externo tal que puede deshabilitar a A.

Finalmente enunciamos el teorema de Abadi-Lamport.

Teorema 2 Si S es una propiedad de seguridad y L es una conjunción numerable de fórmulas $WF(A_i)$ o $SF(A_i)$ tales que cada $A_i (i \in I)$ puede ejecutarse satisfaciendo S, entonces (S, L) es una máquina cerrada.

Una forma de cumplir con la condición "A_i puede ejecutarse satisfaciendo S", es definiendo S como $\bigvee_{i \in I} A_i$. Es decir, S es la disyunción de todas las transiciones permitidas, lo que además implica que S es de seguridad.

7. 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.
- L. Lamport, Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
- M. Abadi and S. Merz, "On TLA as a logic," in Proceedings of the NATO Advanced Study Institute on Deductive program design. Secaucus, NJ, USA: Springer-Verlag New York Inc., 1996, pp. 235–271.
- B. Alpern and F. B. Schneider, "Defining livennes," Information Processing Letters, vol. 21, no. 7, pp. 181–185, Oct. 1985.
- M. Abadi, B. Alpern, K. R. Apt, N. Francez, S. Katz, L. Lamport, and F. B. Schneider, "Preserving liveness: Comments on "safety and liveness from a methodological point of view"," Inf. Process. Lett., vol. 40, no. 3, pp. 141–142, 1991. [Online]. Available: https://doi.org/10.1016/0020-0190(91)90168-H
- L. Lamport, "The temporal logic of actions," ACM Trans. Program. Lang. Syst., vol. 16, no. 3, pp. 872–923, 1994. [Online]. Available: http://doi.acm.org/10.1145/177492.177726
- M. Abadi and L. Lamport, "Composing specifications," ACM Trans. Program. Lang. Syst., vol. 15, no. 1, pp. 73–132, 1993. [Online]. Available: http://doi.acm.org/10.1145/151646.151649
- ——— , "The existence of refinement mappings," Theor. Comput. Sci., vol. 82, no. 2, pp. 253–284, 1991. [Online]. Available: https://doi.org/10.1016/0304-3975(91)90224-P