CSP

José Luis Diaz

Contents

Introducción

Actores

Lenguaje de Actor Mínimo

Un programa en **SAL** no es mas que una secuencia de *comportamientos*. Como en la mayoría de los lenguaje de programación, tiene un nombre reservado **main**, este *Comportamiento* es el punto de entrada, será el responsable de crear otros actores y enviar mensajes.

$\langle Pgm \rangle :== BDef_1 ... BDef_n BDef_{main}$

la definición de *comportamientos* funcionan como una suerte de plantilla para el comportamiento que luego tendrán los actores.

3.1 Expresiones

Existen tres tipos primitivos, booleanos, enteros y dirección del buzón. Las operaciones posibles entre los booleanos son **or**, **and**, **not**. Con respecto a los enteros se pueden operar utilizando +, -, *, /. buzón es un identificador que es devuelto cuando se crea un nuevo actor.

3.2 Definición de comandos

La gramática de los comandos en SAL es la siguiente:

$\langle command \rangle$ ::= '`send`' $e_1, e_2, \ldots, e_n$ '`to`' $\langle actor \rangle$
| '`become`' $B(e_1, e_2, ..., e_n)$
| '`let`' x_1 = '`new`' $B_1(e_1, e_2, ..., e_{1n})$,
... x_k = '`new`' $B_k(e_1, e_2, ..., e_{kn})$
'`in`' $\langle command \rangle$
| '`if`'`<bool-expr>` '`then`' $\langle command \rangle$ '`else`' $\langle command \rangle$ '`end if`'
| $\langle command \rangle$ '`||`' $\langle command \rangle$

send Este comando permite enviar mensajes a otros actores, toma como parámetro una lista separada por coma de las expresiones a enviar, y el actor destino, el envío de mensajes es asincrónico. Cada expresión es evaluada antes de ser enviada.

become Este comando especifica el siguiente comportamiento del actor que está procesando la comunicación recibida. Como en el caso anterior se evalúan las expresiones antes de ser enviadas, y estas aparecerán como la listas de parámetros del comportamiento.

new Este comando sirve para crear nuevos actores. El alcance de los identificadores de los nuevos actores creados está sujeto al cuerpo de **let**.

condicional Luego de evaluar la expresión booleana, si es verdadera ejecuta lo que esta a continuación de **then**, en caso contrario lo que está a continuación de **else**. Funciona como cualquier condicional.

composición Estos dos comandos en **SAL** son ejecutados concurrentemente.

La ejecución de los comandos ocurre cuando el actor recibe un mensaje, y todos ellos ocurren concurrentemente, la composición no es secuencial.

3.3 Definición de comportamientos

La sintaxis de los comportamientos es la siguiente:

⟨*BDef*⟩ :== '`def`' ⟨*beh name*⟩ '`(`' ⟨*acquaiantence-list*⟩ '`)`' '`[`'⟨*input-list*⟩'`]`'
⟨*command*⟩*
'`end def`'
| ⟨*acquaiantence-list*⟩ :== ⟨*id*⟩ | ⟨*id*⟩'`,`' ⟨*acquaiantence-list*⟩
| ⟨*input-list*⟩ :== ⟨*id*⟩ | ⟨*id*⟩'`,`' ⟨*input-list*⟩

El identificador *beh name* esta atado a una abstracción y tiene alcance a todo el programa. Los identificadores *acquaiantence-list* los recibe el momento de instanciación y tiene alcance en todo *command*. Los identificadores *input-list* son completados al momento de procesar una comunicación mensaje, y su alcance es todo *command*.
Ambas listas, *input-list* y *acquaiantence-list* contienen todos los identificadores libres que están en *command*. Existe un identificador especial *self* que puede ser utilizado para hacer referencia al actor que está se definiendo.
La ejecución de *command* deberá contar a lo sumo con un solo comando *become*, esta propiedad tiene que ser garantizado de manera estática, de no existir ningún comando *become*, el actor asumirá un comportamiento de tipo *bottom*, es básicamente ignorar los mensajes que se le envíen.

3.4 Ejemplos

En esta sección mostraremos dos ejemplos de **SAL** y alguas particularidades del lenguaje. Primero se presentará el ejemplo de código, a continuación una breve descripción linea por linea de la funcionalidad, y para terminar notas sobre su funcionamiento.

3.4.1 Cálculo del factorial

Está implementación del factorial está adaptada de [**?**], esta depende de un actor *main* que le envía el valor a calcular. El factorial esta siempre disponible para procesar la siguiente comunicación, no bloquea con el calculo recursivo del factorial sino que lo delega en otros actores.

La palabra reservada *self* hace referencia a la dirección de buzón correspondiente al actor que está procesando la comunicación, este es inicializado cuando el actor es creado.

```
1  def Factorial()[val, customer]
2    become Factorial() ||
3     if val = 0 then
4    send [1] to customer
5    else
6      let cont = new FactorialCont(val, customer)
7          in send [val - 1, cont] to self
8    end if
9  end def
10
11 def FactorialCont(n, customer)[m]
12   send [n * m] to customer
13 end def
14
15 def Main()
16   let fact = new Factorial()
17     in send [3, self] to fact
18 end
```

Linea 1 *Factorial* no recive ningún parametro en el momento de ser inicializado, pero si recibe dos parametros cuando procesa una comunicación, *val* que es un entero, y *customer* que es la dirección de un buzón.

Linea 2 Asigna como siguiente comportamiento a *Factorial*(), el lector podrá notar que no tienen nigún parametro de tipo *input-list*. El operador || es la composición de SAL, es similar a el ; en *C*.

Linea 3 Envía una tupla con el valor 1 al actor con buzón *customer*.

Linea 6 Crea un actor de tipo *FactorialCont*. En este caso se utiliza *new*, ya que se está creando un nuevo buzón, y este se asigna a la variable *cont*. Cuando se asigna el nuevo comportamiento, este recive los parametros *val* y *customer*.

Linea 7 Envia un mensaje a la dirección del buzón propio utilizando la palabra reservada *self*, con la tupla $val - 1$ y la dirección del buzón que se acaba de crear.

Linea 11 *FactorialCont* recive dos valores cuando es instanciado: un entero n y la dirección de un buzón *customer*. Cuando procesa un mensaje, en su *acquaiantence-list* recive un entero m.

Linea 12 Envía la mutiplicación $n * m$ como tupla a la dirección del buzón *customer*

Lineas 15-18 Inicializa el actor *Factorial* y le envia a este la tupla con los valores 3 y la dirección del buzón actual.

Concretamente, el actor ante un entero distinto de cero ejecuta dos acciones, crea un actor con un comportamiento que será multiplicar por **n** el valor recibido y enviarlo al buzón de quien pidió el calculo del factorial de **n**. También, se envía un mensaje a si mismo para evaluar el factorial de **n - 1**, y como dirección de cliente utiliza la dirección del buzón del actor recientemente creado.
Esto establece una red de actores que multiplicaran por el valor indicado enviaran el cálculo al siguiente actor en la red, y el último actor en la red se lo enviará a quien originalmente lo pidió.
Este comportamiento se puede ver en la figura 3.1, En caso de que reciba **0** se le enviara **1** a quien pidió el calculo del factorial.

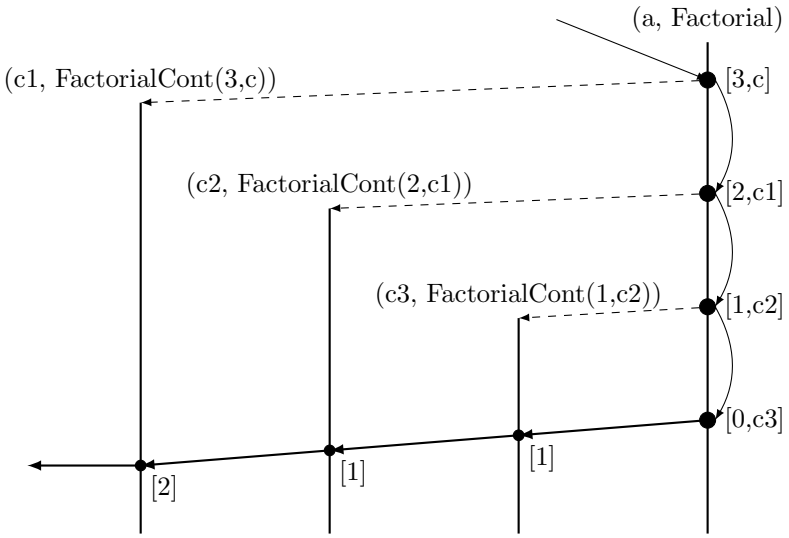

Figure 3.1: El diagrama ilustra el cálculo del factorial de 3, todo el resultado es enviado al actor c. Las lineas verticales indican el paso del tiempo, las de punto indican creación y las otras flechas envio de mensaje.

3.4.2 Una pila usando actores

Otro ejemplo que podemos encontrar en [**?**] es el de una pila, que está representada con una lista enlazada cada nodo de esta lista es un actor.

```
1  def node(content, link)
2    [ case operation of
3          pop : (customer)
4          push : (new-content)
5      end case ]
6    if operation = pop then
7      become link
8      send content to customer
9    enf if
10   if operation = push then
11     let P = new node(content, link)
12       in become new node(new-content, P)
13   end if
14 end def
15
16 def Main()
17   let stack = new node(10, Nil)
18     in send [push, 20] to stack ||
19        send [push, 30] to stack
20   end
21 end
```

El comportamiento *node* recibe dos valores al momento de creación el contenido a guardar y la referencia al siguiente nodo en la red.
Cuando *operation* es de tipo *push*, crea un nuevo *node* que sera el nodo que quedará segundo en la red, tomando los valores que anteriormente tenía la cabeza. Por otra parte, el comportamiento de reemplazo para el actor actual tendrá como contenido el valor recién recibido y como *link* al actor recién creado, de esta manera quedará primero en la red el valor recibido.
Cuando *operation* es de tipo *pop*, se envía el valor que contiene el nodo a la dirección recibida en *customer* y el comportamiento de reemplazo es *become link*, básicamente reenviá todos los mensajes que recibe en su buzón a *link*.
Puede observarse en el ejemplo, que el primer nodo creado tiene como valor *Nil*, esto es simplemente una referencia nula.

Un modelo en CSP

Para empezar a describir como se modeló en **CSP**, primero es importante hacer referencia a que se utilizó **CSPm** [**?**], que combina los operadores de **CSP** originalmente propuesto por Hoare [**?**], y un lenguaje funcional.
En el proceso de traducción de **SAL** a **CSPm**, fue necesario construir un pequeño *runtime* para emular como los actores corren. Recordemos que la naturaleza de **CSP** es sincrónica y los actores no lo son.
Fue necesario concretamente desacoplar la creación de nuevos actores, y el proceso de comunicaciones.

4.1 Estructuras de apoyo

4.1.1 Enteros pequeños

Para evitar la explosión de estados que causa utilizar todo el rango de enteros 64-bits, se generó una una representación propia para reducir está explosión de estados. El valor de MAX_INT es el entero mas grande que se quisiera representar.

$$datatype\ SmallInt = SI.\{0 \ldots MAX\_INT\} \mid Overflow$$

$$\begin{aligned}
add(SI.a, SI.b) &= let\ sum\ = a + b \\
&\quad within\ if\ sum <= MAX_INT\ then\ SI.sum\ else\ Overflow \\
sub(SI.a, SI.b) &= let\ sub\ = a - b \\
&\quad within\ if\ sub >= 0\ then\ SI.sub\ else\ Overflow \\
mult(SI.a, SI.b) &= let\ mult\ = a * b \\
&\quad within\ if\ mult <= MAX_INT\ then\ SI.mult\ else\ Overflow \\
eq(SI.a, SI.b) &= a == b \\
eq(_, _) &= false
\end{aligned}$$

4.1.2 Identificadores de actores

Esta construcción nombra cada uno de los actores que van ser utilizados, también guarda la cantidad de actores de un tipo dado.
Para la definición usa tipos algebraicos similares a los de Haskell soportados por **CSPm**.

$$\begin{aligned} datatype\ ActorID = {} & ACTOR_1.\{1 \dots N_1\} \mid \\ & ACTOR_2.\{1, \dots, N_2\} \mid \\ & ACTOR_k.\{1, \dots, N_k\} \mid \\ & Main.\{1\} \end{aligned}$$

Ya que no existe en **CSP** el concepto de instancia es necesario contar con todos los procesos que van a ser parte de la red definidos desde el principio, concretamente el valor que está entre llaves corresponde a la cantidad de elementos de este tipo que van a ser necesarios.

4.1.3 Valores primitivos

Representaremos las cadenas de caracteres utilizando *Atoms*, estas cadenas son inmutables y no existe ninguna operación sobre ellas.

$$datatype\ Atoms = ATOM_1 \mid ATOM_2 \mid \dots \mid ATOM_n$$

Finalmente, *VALUE* es un tipo de datos que nos permite tener cierta flexibilidad al momento de enviar mensajes o crear actores.

$$datatype\ VALUE = ACTOR.ActorID \mid INT.SmallInt \mid ATOM.Atoms \mid None$$

4.2 Buzón

Recodemos que la naturaleza de **CSP** es sincronica y los actores no lo son para esto necesitamos desacoplar el envío de mensajes de la recepción. Para esto utilizamos una estructura intermedia que actúa de *buzón*, y dos canales que sirven para comunicarse con ella.

$$\begin{aligned} &channel\ CommSend : ActorID.(VALUE, VALUE) \\ &channel\ CommRecv : ActorID.(VALUE, VALUE) \end{aligned}$$

Un *buzón* puede guardar más de una comunicación en su interior. Esto introduce un tipo de no determinismo ya que un actor puede sincronizar con cualquiera de las comunicaciones disponibles en su *buzón*.
El comportamiento del proceso buzón depende de su estado, si no tiene ningún mensaje, si tiene algunos mensajes, o si está completo. Podríamos modelar un

buzón que no tenga una cota superior, pero tendríamos problemas de explosión de estados.

- Si no tiene ningún mensaje, solo sincroniza mensajes por el canal *CommSend*.
- Si tiene algunos mensajes, por los canales *CommSend* y *CommRecv*.
- Si está completo, solo por el canal *CommRecv*.

Puede que los nombre de los canales suenen poco intuitivos, es importante notar que provienen de la acciones vista desde los actores.

CommSend Canal utilizado para comunicar desde cualquier actor hacia el buzón.

CommRecv Canal utilizado para comunicar del buzón hacia el actor asociado.

$$mailbox(i, \langle\rangle) = CommSend?i.x \rightarrow mailbox(i, \langle x \rangle)$$

$$mailbox(i, \langle x \rangle) =$$

$$CommRecv!i.x \rightarrow mailbox(i, \langle\rangle) \,\Box\, CommSend?i.y \rightarrow mailbox(i, \langle x, y \rangle)$$

$$mailbox(i, \langle x, y \rangle) =$$

$$CommRecv!i.x \rightarrow mailbox(i, \langle y \rangle) \,\Box\, CommRecv!i.y \rightarrow mailbox(i, \langle x \rangle)$$

Por cada actor en la red, existe un *buzón* con el mismo **ActorID** asociado. Un actor se bloquea esperando en *CommRecv*, cuando una nueva comunicación está dispuesta a ser sincronizada, desde el *buzón* hacia el actor esta es procesada.

$$mailboxes = \Big|\Big|\Big|_{actor:\{|ActorID|\}} mailbox(actor, \langle\rangle)$$

mailboxes representa todos los buzones puestos en paralelo utilizando el operador *Interleave*. Como no existe comunicación entre buzones, siempre la comunicación es desde un actor hacia un buzón el operador de *Interleave* [**?**, Cap. 2, p. 65].

$$datatype\ ActorID = ACTOR_1.\{1,2,3\} \mid ACTOR_2.\{1,2\} \mid Main.\{1\}$$
$$\{|\ ActorID\ |\} \equiv \{ACTOR_1.1, ACTOR_1.2, ACTOR_2.3, ACTOR_2.1, ACTOR_2.2, MAIN.1\}$$
$$\{|\ ACTOR_2\ |\} \equiv \{ACTOR_2.1, ACTOR_2.2\}$$

Figure 4.1: Ejemplo de notación

4.3 Crear nuevos actores

Antes de empezar, un ejemplo para entender como funciona un aspecto fundamental de los canales de **CSPm**.

$V = \{1, 2, 3\}$
$channelC : V.V$
$P1 = C?x!2 \rightarrow Stop$
$P2 = C!1?y \rightarrow Stop$
$SYSTEM = P1 \,\|\, P2$

Del ejemplo anterior podemos ver que el canal **C** genera el alfabeto de comunicación $\{C.1.1, C.1.2, C.1.3, C.2.1, C.2.2, C.2.3, C.3.1, C.3.2, C.3.3\}$. En $P1$ tenemos la variable libre x y en $P2$ tenemos la variable libre y, es simple notar que al utilizar **'?'** podemos introducir el uso de una variable libre y no así cuando utilizamos **'!'**. Esto genera la sensación de poder enviar un mensaje utilizando **!** y recibirlo usando **?** [**?**, chap. 1,p. 27].

Recordemos que en **CSP** no existe el concepto de instancia, y debemos tener definida la red de procesos desde el comienzo. Para resolver este problema, se presentan tres abstracciones.

La primera, es un conjunto de canales *CreateAsk* y *Create* el primero es utilizando por quien está queriendo crear un nuevo actor y el otro por el actor que está por iniciar.

$channel\ CreateAsk : ActorID.(VALUE, VALUE)$
$channel\ Create : ActorID.(VALUE, VALUE)$

La segunda abstracción es un preámbulo que ocurre antes de un comportamiento, quien dota a este de los parámetros *acquaiantence-list* y al mismo tiempo el otorga al actor su identificador único de buzón. Esto se comporta como una especie de maquina de actores.

$$actors\_machine_k = |||_{id:\{|ACTOR_k|\}} Create!id?(p1, p2) \rightarrow \tag{4.1}$$

$$comportamiento_k(id, p1, p2) \tag{4.2}$$

En realidad, la creación es algo ficticio ya que tenemos una red de procesos **CSP** esperando al evento *Create* para arrancar con el comportamiento definido. El conjunto definido por $ACTOR_k$ es equivalente a los elementos definidos en *ActorID*, esto se puede ver en el ejemplo 4.2.
Por esto es que decimos que no solo define el nombre, sino que al mismo tiempo está estableciendo cuantos actores del tipo $ACTOR_k$ vamos a tener. Siguiendo con esta idea, $actors_m achine_k$, representa todos los actores que van ser iniciados en paralelo utilizando el operador de *Interleave*. Una vez sincronizado en el mensaje *Create*, se ejecuta el comportamiento que obtiene el identificador del buzón y los parámetros recibidos.

La tercer abstracción es un proceso que conecta esto dos mensajes.

$$create(actorId) = CreateAsk!actorId?m \rightarrow Create.actorId!m \rightarrow STOP$$
$$creates = |||_{actor:ActorID}\ create(actor)$$

Como puede observarse, tenemos tantos procesos en paralelo como *ActorID* existan, tal como muestra el ejemplo en 4.2.
Tal vez esta abstracción podría haber sido omitida, pero juega un papel fundamental en la construcción total del sistema, esto tiene que ver con en **CSP** se puede elegir los eventos [**?**, chap. 2,p. 55] que se van a sincronizar, cuando veamos como compone todo el sistema esta idea quedará mas clara.

4.4 Definición de comportamientos

La idea de comportamiento fue introducida en la sección **??**, podemos pensar a un comportamiento como una función que procesa una comunicación y tiene como salida, nuevas comunicaciones, nuevos actores y el comportamiento de reemplazo para el actor que esta procesando la comunicación.

$CommSend.actor_{buzon}!(p_1, p_2)$	(*Enviar Comunicaciones*)
$CreateAsk!Actor_m.pid?(p_1, p_2)$	(*Crear nuevos actores*)
$actor_{running}(self, p_1, p_2)$	(*Comportamiento de reemplazo*)

Enviar Comunicaciones En este caso le enviaremos al actor con buzón $actor_{buzon}$ la tupla (p_1, p_2).

Crear nuevos actores Obtendríamos mediante $Actor_m.pid$ el identificador de buzón del actor creado, y le enviaríamos los parámetros (p_1, p_2) como *acquaiantence-list*.

Comportamiento de reemplazo En este caso el comportamiento sería $actor_{running}$, de no contar con uno sería simplemente Stop.

4.5 Ejemplo: cálculo de factorial en CSPm

Se utilizará en esencia el mismo ejemplo del factorial antes visto en sal **SAL**, está compuesto por dos comportamientos *Factorial* y *FactorialWorker*.

Comportamiento de Factorial

$$factorial = Create.Factorial.1?(None, None) \rightarrow factorial_{running}(Factorial.1)$$

En el caso de la formula anterior, espera a sincronizar con un mensaje de creación. Esto no solamente simula la creación, sino que al mismo tiempo se

asigna el buzón con nombre *Factorial*.1 al actor que está corriendo. Dentro de la definición del comportamiento hace referencia a *self* como nombre de su propio buzón.

Notar que en este caso solo se necesita un actor de tipo *Factorial*.

$$
\begin{array}{l}
factorial_{running}(self) = CommRecv?self.(ACTOR.mailboxClient, INT.k) \rightarrow \\
\textbf{if}(eq(k, SI.0)) \\
\quad then \\
\quad\quad CommSend!mailboxClient.(INT.SI.1, None) \rightarrow factorialRunning(self) \\
\quad \textbf{else} \\
\quad\quad \textbf{let} \\
\quad\quad\quad newK = sub(k, SI.1) \\
\quad\quad \textbf{within} \\
\quad\quad\quad CreateAsk?FactorialWorker.pid!(INT.k, ACTOR.mailboxClient) \rightarrow \\
\quad\quad\quad CommSend!self.(ACTOR.FactorialWorker.pid, INT.newK) \rightarrow \\
\quad\quad\quad factorial_{running}(self)
\end{array}
$$

Cuando recibe un entero distinto de cero ejecuta dos acciones, crea un actor **FactorialWorker** y se envía un mensaje a si mismo para evaluar el factorial de **n - 1**.
En este caso el comportamiento de reemplazo para el buzón actual no cambia.

Comportamiento de FactorialWorker

En el caso del comportamiento de **FactorialWorker** vamos a necesitar mas de un actor de este tipo, puede verse a continuación la notación antes vista, para poner en paralelo tantos procesos como elementos existan en el conjunto $\{|\ FactorialWorker\ |\}$.

$$
\begin{array}{l}
factorialWorker = |||_{x:\{|FactorialWorker|\}} Create!x?(k, mailboxClient) \\
\quad \rightarrow factorialWorker_{running}(x, k, mailboxClient) factorialWorker_{running}(self, INT.k, ACTOR.mailboxClie \\
CommRecv.self?(INT.n, None) \rightarrow \\
\quad \textbf{let} \\
\quad\quad val = mult(n, k) \\
\quad \textbf{within} \\
\quad\quad CommSend.mailboxClient!(INT.val, None) \rightarrow \\
\quad Stop
\end{array}
$$

Este comportamiento es muy simple, en el momento de creación recibe dos parámetros, un entero **k** y un dirección de un buzón, al momento de recibir una comunicación, efectúa la multiplicación y se lo envía a *mailboxClient*. En este caso no cuenta con comportamiento de reemplazo, entonces termina con *Stop*

4.6 Ejemplo: Una pila en CSPm

$$
\begin{aligned}
&node = |||_{x:\{|Node|\}} Create.actorId?(INT.content, ACTOR.link) \\
&\quad \rightarrow node_{running}(actorId, content, link) \\
&node_{running}(self, content, link) = \\
&\quad CommRecv.self?(ATOM.PUSH, INT.newContent) \rightarrow \\
&\quad CreateAsk?Node.newNode!(INT.content, ACTOR.link) \rightarrow \\
&\quad nodeRunning(self, newContent, Node.newNode) \\
&\Box \\
&\quad CommRecv.self?(ATOM.POP, ACTOR.client) \rightarrow \\
&\quad CommSend.client!(INT.content, None) \rightarrow \\
&\quad fwd(self, link)
\end{aligned}
$$

$$fwd(in, out) = CommRecv.in?msg \rightarrow CommSend.out!msg \rightarrow fwd(in, out)$$

4.7 Ejemplo: Un una cola en SAL

Introducir el código en sal, y explicar como funciona.

4.8 Ejemplo: Un una cola en CSPm

Introducir el código en CSPm equivalente y explicar las particularidades.

4.9 Ejemplo: Un cliente-servidor de chat en SAL

Introducir el código en sal, y explicar como funciona.

4.10 Ejemplo: Un cliente-servidor de chat en CSPm

Introducir el código en CSPm equivalente y explicar las particularidades.

Formaliznado la semantica en CSP

TODO: Definir **translateExp**

La clase **Cmnd** con elementos de tipo S está dada por:

```
S :== S_1 ; S_2 | if b then S_1 else S_2 | send [e1, .., e_i] to a | become new
E(e_1, .. ,e_i) | let a_1 = new E_1(e_1,..,e_i) and ... a_j = new
E_1(e_1,..,e_i) { S }
```

definimos la funcion **translateCmd** de la siguiente forma:

```
translateCmd (S_1 S_2) = translateCmd(S_1) -> translateCmd(S_2)

translateCmd(if b then S_1 else S_2) =
   if (translateExp(b)) then
       translateCmd(S_1) else
       translateCmd(S_2)

translateCmd(send[e_1, ..., e_i] to a) = CommSend.a.
         (translateExp(e_1), ...,
          translateExp(e_i))

translateCmd(become new Beh(e_1, ..., e_n)) = runningBeh(self, e_1, ..., e_n)
```

newEnv es el resultado de agregar a el entorno de las variables de mailbox
$a_1 = E_1.pid_1$ .. $a_n = E_n.pid_n$

```
translateCmd(let a_1 = new E_1(e_1, ..., e_j) and
         ... and a_n = new E_N(e_1, ..., e_j) { S } =

CreateAsk?E_1.pid_1!(translateExp(e_1), ...,translateExp(e_j)) ->
CreateAsk?E_N.pid_n!(translateExp(e_1), ...,translateExp(e_j)) ->
translateCmd(S, newEnv)
```

La clase **Beha** con elementos de tipo S está dada por:

```
def behName(a_1 .. a_i)[n_1 ... n_j]
  S
end def
```

Tendria como equivalente en CSP:

```
behName = ||| actorId : {|BehName|} @ Create.actorId?(a_1, ..., a_i) ->
behNameRunning(actorId, a_1, .., a_n)

behNameRunning(self, a_1, .., a_n) = CommRecv.self(n_1 ... n_j) -> translateCmd(S)
```

Bibliography