Universidad Nacional de Rosario

Departamento de Ciencias de la Computación

Un modelo semántico para Actores basado en CSP

Tesina de grado

Autor:
José Luis Diaz

Directores:
Maximiliano Cristiá
Hernán Ponce de León

7 de mayo de 2018

A mi viejo.

Resumen

Debido al incremento de la cantidad de núcleos por microprocesador, las aplicaciones hacen un uso más frecuente de la concurrencia. Una forma de programar este tipo de aplicaciones es utilizando el modelo tradicional de concurrencia que se basa en multi-hilos, variables compartidas, locks, etc. Este trabajo propone estudiar un enfoque diferente: el modelo de actores utilizado en la industria, particularmente en lenguajes como Erlang y Scala con la librería Akka.

El objetivo de este trabajo es comprender el modelo de actores y su semántica. Con este fin, luego de explorar las características del modelo de actores, se introduce un lenguaje simple de actores llamado *SAL* cuya semántica se formalizara en *CSP*. Con este modelo se realizan algunas pruebas utilizando la herramienta *FDR*, tales como revisión del árbol de ejecución y verificación de refinamiento utilizando el modelo de traza. Se presentan varios ejemplos con el fin de ayudar al lector a comprender el paradigma de programación concurrente basado en actores.

Índice general

Capítulo 1

Introducción

1.1. Motivación

Debido al incremento de la cantidad de núcleos por microprocesador, las aplicaciones hacen un uso más frecuente de la concurrencia. Una forma de programar este tipo de aplicaciones es utilizando el modelo tradicional de concurrencia que se basa en multi-hilos, variables compartidas, locks, etc. Este trabajo propone estudiar un enfoque diferente: el modelo de actores utilizado en la industria, particularmente en lenguajes como Erlang [4, 17] y Scala [11] con la librería Akka [18].

El objetivo de este trabajo es comprender el modelo de actores y su semántica. Con este fin, luego de explorar las características del modelo de actores, se introduce un lenguaje simple de actores llamado *SAL* cuya semántica se formalizara en *CSP*. Con este modelo se realizan algunas pruebas utilizando la herramienta *FDR*, tales como revisión del árbol de ejecución y verificación de refinamiento utilizando el modelo de traza. Se presentan varios ejemplos con el fin de ayudar al lector a comprender el paradigma de programación concurrente basado en actores.

La diferencia entre ambos modelos se puede notar mediante el problema del jardín ornamental. El enunciado del problema es el siguiente: supongamos que tenemos dos entradas a un parque y se requiere saber cuanta gente ingresa. Para eso se instala un molinete en cada entrada. Se utiliza una computadora para registrar la información de ingreso.

Siguiendo el modelo tradicional de concurrencia, se incluiría una variable global que guarde la cantidad de visitantes y dos hilos representando los molinetes que incrementan esta variable. Sin ningún tipo de protección en la región crítica planteada por la actualización de la variable global podrían perderse incrementos, ya que cada hilo carga localmente el valor de la variable global, efectúa un incremento y finalmente guarda el valor en la variable global.

El mismo problema se puede escribir utilizando el modelo de actores, que tiene como único mecanismo de comunicación entre entidades el paso de mensajes. En este

caso, el problema puede ser representado utilizando un actor que realiza la tarea de contador. Este incrementará su valor cuando reciba un mensaje *inc*. Otros dos actores emitirán los mensajes *inc* (los molinetes). En este caso el problema de la pérdida de la actualización no ocurre.

El modelo de actores fue originalmente propuesto por C. Heweeit [18]. Es un enfoque diferente a cómo estructurar programas concurrentes. Según este modelo un actor es una entidad computacional que puede:

- Enviar y recibir un número finito de mensajes a otros actores.
- Crear un número finito de actores.
- Designar un nuevo comportamiento a ser usado cuando se reciba el próximo mensaje.

Como señala Rob Pike en su charla titulada "Concurrencia no es paralelismo" [12], muchas veces se pasa por alto la diferencia conceptual entre la concurrencia y el paralelismo. En programación, la concurrencia es la composición de los procesos independientemente de la ejecución, mientras que el paralelismo es la ejecución simultánea de cálculos (posiblemente relacionados). El modelo de actores, mejora sustancialmente la composición.

1.2. Objetivo

El objetivo de este trabajo es comprender el modelo de actores y su semántica. Una buena herramienta para asistir a este proceso es utilizar métodos formales. Se propone modelar la semántica del modelo de actores en *CSP* y efectuar algunas pruebas utilizando la herramienta *FDR* [16].

CSP

Communicating Sequential Processes (*CSP*), fue propuesto por primera vez por C.A.R Hoare [13]. Es un lenguaje para la especificación y verificación del comportamiento concurrente de sistemas. Como su nombre lo indica, *CSP* permite la descripción de sistemas en términos de componentes que operan de forma independiente e interactúan entre sí únicamente a través de eventos sincrónicos. Las relaciones entre los diferentes procesos y la forma en que cada proceso se comunica con su entorno, se describen utilizando un álgebra de procesos.

FDR

Es una herramienta para el análisis de los modelos escritos en *CSP*. El lenguaje de entrada de *FDR*, *CSPm*, combina los operadores de *CSP* con un lenguaje de programación funcional. *FDR* originalmente fue escrito en 1991 por Formal Systems (Europe)

Ltd, que también lanzo la versión 2 a mediados de la década de 1990. La versión actual de la herramienta esta disponible gracias a la Universidad de Oxford. Se puede utilizar para fines académicos si necesidad de una licencia, siendo sí necesaria para fines comerciales.

Si bien tanto *CSP* como el modelo de actores apuntan al problema de la concurrencia, no lo modelos tienen entidades concurrentes que intercambian eventos o mensajes. Sin embargo, los dos modelos toman algunas decisiones fundamentalmente diferentes con respecto a las primitivas que proporcionan:

- Los procesos de *CSP* son anónimos, mientras que los actores tienen identidades.
- En *CSP* los procesos involucrados en el envío y la recepción de un evento deben sincronizar. Es decir, el remitente no puede transmitir un evento hasta que el receptor está dispuesto a aceptarlo. Por el contrario, en los sistemas de actores, el paso de eventos es fundamentalmente asíncrono, es decir, la transmisión y la recepción de eventos no tienen que suceder al mismo instante.
- *CSP* utiliza canales explícitos para el paso de datos, mientras que los sistemas de actores transmiten datos a los actores de destino mediante la dirección de su buzón.

Estos enfoques pueden ser considerados duales el uno al otro, en el sentido de que los sistemas basados en *sincronización* pueden utilizarse para construir comunicaciones que se comporten como sistemas de mensajería asíncrona, mientras que los sistemas asíncronos se pueden utilizar para construir las comunicaciones sincrónicas utilizando algún protocolo que permita el encuentro entre los procesos. Lo mismo ocurre con los canales.

1.3. Trabajos Relacionados

En su tesis doctoral Agha [1] define en detalle el modelo de actores. También define dos lenguajes *SAL* y *ACT*. Da una semántica denotacional a *SAL*. Este trabajo sigue de cerca este lenguaje, y construye una semántica en *CSP*.

En el trabajo *An Algebraic Theory of Actors and its Application to a Simple Object-Based Language* [2], Gul Agha y Prasanna Thati definen un modelo algebraico del modelo de actores, y sobre el final le dan semántica a *SAL*.

En su tesis doctoral Clinger [6] presenta una teoría para una clase de lenguajes no deterministas que incluyen paralelismo, enfocado específicamente en el modelo de actores.

Se puede ver en el trabajo *An algebra of actors* [9], que los autores definen un algebra de procesos y proveen una semántica operacional basada en un sistema de transición etiquetado.

En el trabajo *Formalising Actors in Linear Logic* [8], los autores definen un formalismo basado en actores en términos la deducción en lógica lineal.

Otro lenguaje que implementa el modelo de actores es Pony [3], utiliza memoria compartida para evitar duplicación de memoria cuando se envían mensajes. Esto puede llevar a una o varias *condiciónes de carreras*. Para evitar este tipo de problemas, tiene definido un sistema de permisos, con una prueba formal [5]. Este sistema de permisos leer, escribir y atravesar referencias únicas evitando este problema.

1.4. Organización de este trabajo

En el capítulo 2 se introduce el modelo de actores y se describe la sintaxis de *SAL*. En el capítulo 3 se presentan los conceptos necesarios de *CSP* y *CSPm*, que serán luego utilizados en el siguiente capítulo. El capítulo 4 construye un modelo de actores utilizando *CSP* donde, se presentan varios ejemplos. Se muestra una semántica de *SAL* en *CSP*. Se termina mostrando como este modelo puede ser utilizado en la herramienta *FDR*. Finalmente, el capítulo 5 presenta algunas conclusiones del trabajo y los posibles trabajos futuros.

Capítulo 2

Sistema de actores

En este capítulo se explica cómo funciona la estructura computacional en el modelo de actores. En la primera sección se introduce el funcionamiento de las comunicaciones y el comportamiento de los actores. En la segunda sección se define la sintaxis de un lenguaje mínimo de actores, para terminar con algunos ejemplos utilizando este lenguaje.

2.1. Describiendo un sistema de actores

Un sistema de actores consiste de configuraciones. Una configuración viene dada por una colección de actores concurrentes y una colección de comunicaciones en tránsito. Cada actor tiene un nombre único y un comportamiento. Se comunica con otros actores a través de mensajes asíncronos. Los actores son reactivos, es decir, ejecutan solo en respuesta a los mensajes recibidos. El comportamiento de un actor es determinista, ya que la respuesta está determinada por el contenido del mensaje que procesa.

2.1.1. Comunicaciones

Se puede decir que las comunicaciones que no son fueron procesadas son quienes mueven los cálculos en un sistema de actores. Las comunicaciones vienen representados por el par:

1. *destino*, la dirección de buzón a la que será entregada la comunicación.
2. *mensaje*, básicamente la información que estará disponible al actor que procesará la comunicación.

Un *mensaje* es una lista de valores. Estos valores pueden ser: direcciones de buzón, enteros, cadenas de caracteres, etc.

El *destino* debe ser una dirección de buzón válida, es decir, un actor antes de enviarle una comunicación a otro actor debe tener la dirección de buzón destino, y esta

debe ser válida. Existen tres formas en la cual un actor α, al aceptar una comunicación $\bar{k}$, puede conocer la dirección de buzón de otro actor:

- El *destino* era conocido por el actor α, antes de aceptar la comunicación.
- El *destino* estaba incluido como parte de la comunicación $\bar{k}$.
- El *destino* es una dirección de buzón creada como resultado de aceptar la comunicación $\bar{k}$.

Es probable que más de un actor al mismo tiempo quiera enviar una comunicación a un buzón. Para esto es necesario alguna estructura intermedia que guarde todas las comunicaciones creadas y que aún no fueron procesadas. Esta estructura tiene que tener la capacidad para guardar todas las comunicaciones hasta ser procesadas por el actor destino.

2.1.2. Actores

Toda computación en el modelo de actores es resultado de procesar comunicaciones. Un actor acepta un mensaje cuando este es procesado. Un actor sólo puede procesar comunicaciones que están dirigidas a su dirección de buzón. Como resultado de aceptar una comunicación, un actor puede: crear nuevas comunicaciones, crear nuevos actores y debe definir su comportamiento de reemplazo.

Un actor puede describirse especificando:

- Su dirección de buzón, a la que le corresponde un *buzón* lo suficientemente grande para almacenar las comunicaciones aún no procesadas.
- Su *comportamiento*, que es una función que tiene la comunicación que está siendo procesada como entrada y como salida nuevos actores, nuevas comunicaciones y su nuevo comportamiento de reemplazo.

Podemos pensar que un actor es un *buzón* donde llegan todas las comunicaciones, y un proceso que ejecutará su comportamiento. Este proceso apunta a una comunicación particular de este *buzón*.

Cuando el proceso X_n acepta la $(n) - ésima$ comunicación, eventualmente crea el proceso X_{n+1} el cual ejecutará el comportamiento de reemplazo definido por X_n. Este nuevo proceso apunta a la siguiente comunicación en el buzón en donde estará guardada la comunicación $(n+1) - ésima$. Esto puede verse en la figura 2.1

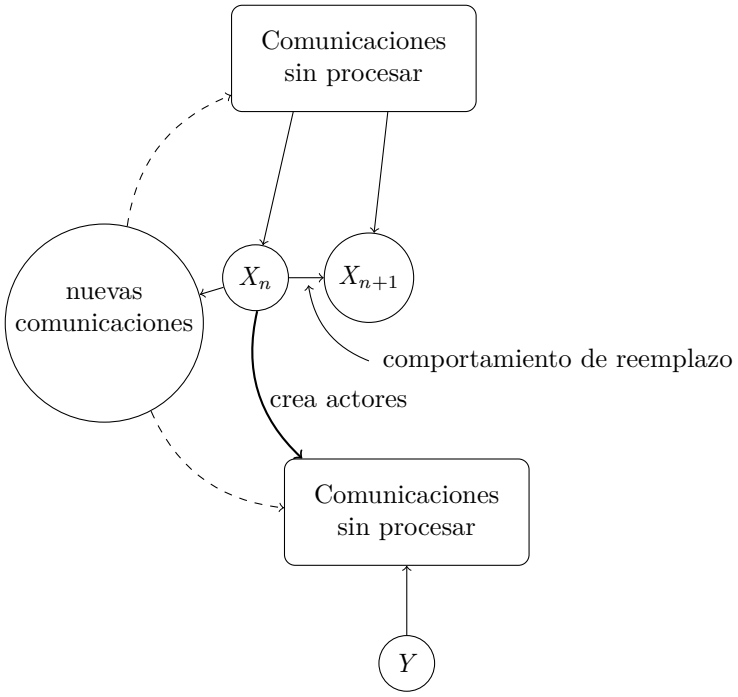

Figura 2.1: Una transición entre dos comportamientos

Los dos procesos X_n y X_{n+1} no interfieren entre sí, pero la creación de X_{n+1} depende de que X_n haya definido su comportamiento de reemplazo. X_n solo procesa la $(n) - ésima$ comunicación. Exceptuando el caso en el cual se envíe una comunicación a sí mismo, en este caso indirectamente estaría interfiriendo. Cada uno de estos procesos crean sus propias tareas, sus propios actores como esta definido en sus comportamientos. Antes que el proceso X_n cree el proceso X_{n+1}, X_n podría haber creado otros actores u otros trabajos. Es posible incluso, que X_n esté creando actores o trabajos al mismo tiempo que lo está haciendo X_{n+1}. Es importante notar que X_n no recibirá ninguna otra comunicación, ni tampoco especificará ningún otro comportamiento de reemplazo.

Esto parecería definir algún tipo de orden con respecto a cómo se van a procesar los mensajes. Sin embargo, el trabajo de Agha [1] no se define ningún orden específico sobre el procesamiento de comunicaciones. La única garantía que el modelo provee es sobre las comunicaciones, está relacionada con el eventual proceso de los mensajes. Tanto la implementación de Erlang [4] como la implementación de Akka [18], entregarán al menos una vez la comunicación a un actor, pero no garantizan la entrega.

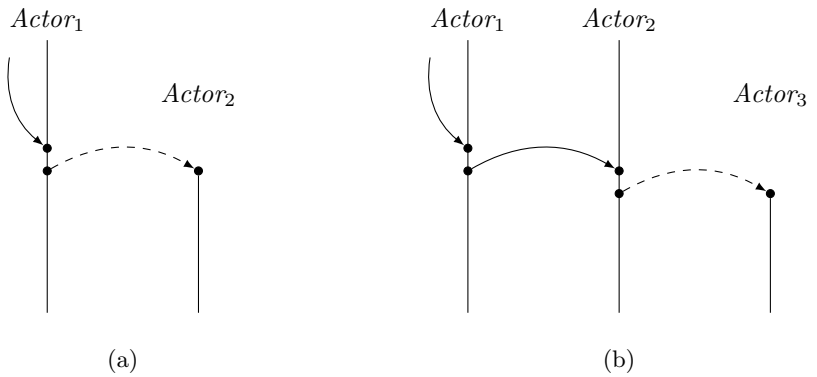

Figura 2.2: Las lineas verticales indican el paso del tiempo, las de punto indican creación de actores y las otras flechas envío de mensaje.

2.1.3. Dinámica del modelo de actores

Las comunicaciones están definidas por el par buzón destino y mensaje, de la siguiente forma:

$$\mathcal{K} = \mathcal{B} \times \mathcal{M} \tag{2.1}$$

donde $\mathcal{B}$ es el conjunto de las direcciones buzón, $\mathcal{M}$ es el conjunto de los mensajes.

La definición de los comportamiento viene dada de la siguiente forma:

$$\mathcal{C} : \mathcal{K} \to (\{\bar{k}_1, \bar{k}_2, \ldots, \bar{k}_m\}, \{\alpha_1, \alpha_2, \ldots, \alpha_n\}, c) \tag{2.2}$$

donde $\{\bar{k}_1, \bar{k}_2, \ldots, \bar{k}_m\}$ es el conjunto de las nuevas comunicaciones, estas tienen la forma que se muestra en la ecuación 2.1. $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ son los nuevos actores creados y el comportamiento de reemplazo esta definido por c.

Dado que un actor está compuesto de un buzón y su comportamiento, es necesario vincular las direcciones de buzones con un determinado comportamiento. Se usa una función parcial que tiene como dominio un subconjunto de las todas las direcciones de buzón, y como codominio un subconjunto de los comportamientos. La definición de la función esta dada por:

$$f_{actor} : \mathsf{B} \nrightarrow \mathsf{C} \tag{2.3}$$

donde B es un subconjunto de $\mathcal{B}$ y C es un subconjunto de $\mathcal{C}$. Esta función modela los actores que fueron creados ya que un actor, como por ejemplo los que se crean en la ecuación 2.2.

Con estas definiciones se establece a una configuración como la tupla (f_{actor}, κ). Donde κ es el conjunto de las comunicaciones no procesadas. La evolución de un sistema de actores ocurre cuando una comunicación es procesada, es decir, cuando se pasa de una configuración a otra configuración. Para que el sistema evolucione

de la configuración (f_{actor}, κ) a la configuración (f'_{actor}, κ'), los siguientes pasos están involucrados:

- Es necesario remover un elemento del conjunto de las comunicaciones no procesadas κ. La comunicación será de la forma (β, m), donde β es un buzón destino y m un mensaje.
- Al aplicar $f_{actor} + \beta$, obtendremos el comportamiento c_b.
- Al aplicar en el comportamiento c_b el mensaje m, se obtendrán los nuevos actores, los nuevos mensajes, y el comportamiento de reemplazo.
- Se los agregan en f_{actor} los nuevos actores $\alpha_1, \alpha_2, \ldots, \alpha_n$. Esto se hace de la siguiente manera: $f''_{actor} = f_{actor} \cup \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$
- Se incorporan las nuevas comunicaciones $\bar{k}_1, \bar{k}_2, \ldots, \bar{k}_m$ en κ. Esto se hace de la siguiente forma: $\kappa' = \kappa \cup \{\bar{k}_1, \bar{k}_2, \ldots, \bar{k}_m\}$
- Se tiene que cambiar en f_{actor} el comportamiento de reemplazo obtenido: c'_b. Se hace lo siguiente: $f'_{actor} = (f''_{actor} \oplus \{\beta \rightarrow c'_b\})$. Es decir, en la nueva función de comportamientos se reemplaza para el buzón β el comportamiento c_b por el comportamiento c'_b.

Estos pasos muestran cómo un sistema de actores va de una configuración a otra.

2.2. Programando con actores

En esta sección se explora un lenguaje que implementa los conceptos básicos del modelo de actores. El lenguaje *SAL* fue desarrollado con intenciones pedagógicas y tiene una sintaxis heredada de Algol. Un programa *SAL* en un sistema de actores esta compuesto por:

- *definición de comportamientos*: asocia un esquema de comportamiento con un identificador, no crea ningún actor.
- expresiones *new* para crear nuevos actores.
- comandos *send* para crear nuevas tareas.

Primero se explora la sintaxis de las expresiones, ya que las expresiones son utilizadas tanto por la definición de comportamientos como por los comandos. Luego se presentará como definir comportamientos. Para terminar esta sección mostrando la definición de comandos.

Se utiliza la notación **Backus-Naur** [10] para describir la gramática. Adjunto a la notación se utiliza el símbolo $\langle + \rangle$ cuando haya al menos una ocurrencia de un término.

2.2.1. Expresiones

Existen cuatro tipos primitivos: booleanos, enteros, cadenas, y dirección del buzón. Las operaciones posibles entre los booleanos son **or**, **and** y **not**.

La gramática de las expresiones booleanas es la siguiente:

⟨*bexp*⟩ ::= ⟨*bterm*⟩ '`or`' ⟨*bterm*⟩ | ⟨*bterm*⟩ '`and`' ⟨*bterm*⟩ | ⟨*exp*⟩ = ⟨*exp*⟩
⟨*bterm*⟩ ::= ⟨*bool*⟩ | '`not`' ⟨*bterm*⟩ | '`(`' ⟨*bexp*⟩ '`)`' | ⟨*id*⟩
⟨*bool*⟩ ::= '`TRUE`' | '`FALSE`'

donde *id*, es simplemente cualquier carácter entre la *A* y la *Z* tanto mayúscula como minúscula y representa las variables. Estas pueden ser recibidas en el momento de creación o cuando recibe una comunicación.

Los enteros se pueden operar utilizando `+`, `-`, `*` y `/`. La gramática de los enteros es la siguiente:

⟨*iexp*⟩ ::= ⟨*iterm*⟩ '`*`' ⟨*iterm*⟩ | ⟨*iterm*⟩ '`/`' ⟨*iterm*⟩
| ⟨*iterm*⟩ '`+`' ⟨*iterm*⟩ | ⟨*iterm*⟩ '`-`' ⟨*iterm*⟩
⟨*número*⟩ ::= '`1`' | '`2`' | '`3`' | '`4`' | '`5`' | '`6`' | '`7`' | '`8`' | '`9`' | '`0`'
⟨*iterm*⟩ ::= ⟨*número*⟩$^+$ | '`-`' ⟨*iterm*⟩ | '`(`' ⟨*iexp*⟩ '`)`' | ⟨*id*⟩

Las cadenas son constantes. La gramática para las cadenas es la siguiente:

⟨*id*⟩ ::= '`"`' ⟨*carácter*⟩$^+$ '`"`'

donde *carácter* es simplemente cualquier carácter entre la *A* y la *Z* tanto mayúscula como minúscula.

La gramática de todas las expresiones viene dada por:

⟨*exp*⟩ :: = ⟨*iexp*⟩ | ⟨*bexp*⟩ | ⟨*sexp*⟩ | ⟨*id*⟩

La dirección de un buzón es un identificador que es devuelto cuando se crea un nuevo actor; este tipo primitivo no tiene ningún operador asociado.

2.2.2. Definición de comportamientos

Cada vez que un actor acepta una comunicación, define un comportamiento de reemplazo. Cada comportamiento está parametrizado. Por ejemplo, si el comportamiento de una cuenta bancaria depende de su saldo, entonces se especifica el comportamiento de la cuenta como una función de su saldo. Cada vez que se crea una cuenta, o se define un comportamiento de reemplazo que usa la definición de una cuenta bancaria, se tiene que dar un valor específico de saldo.

Existen dos listas de parámetros que están involucradas en la definición de un comportamiento. La primera lista corresponde a los parámetros que son dados al momento de la creación de un actor, esta lista es llamada *acquaintance-list*. La segunda, que se obtiene cuando una comunicación es aceptada, es llamada *comunication-list*.

En el caso de *communication-list*, esta asume que todas las comunicaciones serán una secuencia de identificadores. Siguiendo con el comportamiento que modela una cuenta bancaria resulta útil que esta lista de identificadores esté dada en función de la operación que se vaya a ejecutar. Por ejemplo, si la operación fuera *'extracción'* solamente necesitaríamos un identificador *'monto'* el cual tendría el valor de la extracción en cuestión. En el caso que la operación fuera *'saldo'* esta no requiere identificadores.

Se presenta a continuación una gramática que contempla el caso en el cual *communication-list* sea sólo una lista de identificadores (se usará mas la sección A.2 el ejemplo del cálculo del factorial). También se presenta una forma de bifurcación ante diferentes ramas dependiendo del contenido de la comunicación (el ejemplo de la una pila de la sección 2.3.2 hace uso de esto). La gramática de los comportamientos es la siguiente:

⟨*BDef*⟩ :== '`def`' *NombreComportamiento* '`(`' ⟨*acquaiantence-list*⟩ '`)`' ⟨*body*⟩ '`end def`'
⟨*acquaiantence-list*⟩ :== ⟨*id*⟩ | ⟨*id*⟩ '`,`' ⟨*acquaiantence-list*⟩
⟨*body*⟩ :== ⟨*static-list*⟩ | ⟨*match-list*⟩
⟨*static-list*⟩ :== '`[`'⟨*comunication-list*⟩'`]`'
⟨*command*⟩
⟨*comunication-list*⟩ :== ⟨*id*⟩ | ⟨*id*⟩ '`,`' ⟨*comunication-list*⟩
⟨*match-list*⟩ :== '`match`' ('`case`' '`[`' ⟨*case-list*⟩ '`]:`' ⟨*command*⟩)+
⟨*case-list*⟩ :== ⟨*case-exp*⟩ | ⟨*case-exp*⟩ '`,`' ⟨*case-list*⟩
⟨*case-exp*⟩ :== ⟨*bool*⟩ | ⟨*numero*⟩ | ⟨*id*⟩

donde:

NombreComportamiento identifica un comportamiento. Tiene alcance a todo el programa.

static-list son completados al momento de procesar una comunicación, y su alcance es todo *command*.

comunication-list se utiliza para definir una lista de identificadores.

case-exp puede ser de tipo booleana, un número o un identificador.

case-list se utiliza para definir una lista de elementos de tipo *case-exp*.

match-list Se comparan las expresiones una a una, de haber coincidencia se ejecutan los comandos que están a continuación. De contener identificadores libres, estos se inicializarán con el valor contenido en el mensaje para esa posición.

acquaiantence-list los recibe al momento de inicialización y tiene alcance en todo *command*.

body puede ser una lista de argumentos estática o se puede utilizar el operador *match*. *command* se define en la siguiente sección.

Ambas listas, *comunication-list* y *acquaiantence-list*, contienen todos los identificadores libres que están en *command*. En el caso de *match-list* los identificadores libres tienen alcance a los *commands* asociados a cada *case*. Existe un identificador especial *self* que puede ser utilizado para hacer referencia al buzón del actor que se está definiendo.

La ejecución de *command* deberá contar a lo sumo con un solo comando *become*. Esta propiedad tiene que ser garantizada de manera estática; de no existir ningún comando *become*, el actor asumirá un comportamiento de tipo *bottom*, que es básicamente ignorar los mensajes que se le envíen.

2.2.3. Definición de comandos

Los comandos son las acciones que permiten a *SAL* crear nuevos actores, enviar mensajes y definir un nuevo comportamiento. Estos describen en esencia lo que un actor puede hacer dentro de un comportamiento.

command viene definido definido por la siguiente gramática:

⟨*command*⟩ :== ⟨*LET*⟩ | ⟨*SEND*⟩ | ⟨*BECOME*⟩ | ⟨*IFCOMP*⟩

Donde: *LET*, *SEND*, *BECOME* y *IFCOMP* se definen en las próximas secciones.

Creando actores

Los actores son creados usando un comando de tipo *new*, que devuelve una nueva dirección de buzón del actor recién creado. La sintaxis de las expresiones de tipo *new* es la siguiente:

⟨*expr_list*⟩ ::= ⟨*exp*⟩ | ⟨*exp*⟩ ‘`,`’ ⟨*expr_list*⟩
⟨*new_expr_list*⟩ ::= ⟨*new_expr*⟩ | ⟨*new_expr*⟩ ‘`,`’ ⟨*new_expr_list*⟩
⟨*new_expr*⟩ ::= *mexp* ‘`=`’ ‘`new`’ *NombreComportamiento* ‘`(`’ ⟨*expr_list*⟩ ‘`)`’
⟨*LET*⟩ ::= ‘`let`’ ⟨*new_expr_list*⟩ ‘`in`’ ⟨*command*⟩

NombreComportamiento hace referencia a un identificador vinculado con un comportamiento específico, declarado utilizando una definición de comportamiento, o sea una *BDef*. Se crea un nuevo actor con el comportamiento descripto en la definición del comportamiento y sus parámetros son instanciados con los valores de las expresiones entre paréntesis, o sea una $expr_list$. Utilizando el léxico de actores, corresponde a los valores denominados como *acquaintance-list*. El identificador *mexp* es el valor de la dirección del buzón. Este identificador puede ser destino de nuevas comunicaciones.

Los actores son creados de manera concurrente, estos pueden conocer entre sí la dirección de buzón. Esta es una forma de definición mutuamente recursiva que es perfectamente válida en el modelo de actores.

Creando comunicaciones

Una comunicación es creada especificando un actor destino y un mensaje. Las comunicaciones se pueden enviar a actores que ya fueron creados o actores creados por quien está enviando la comunicación. El destino es la dirección de buzón del actor al que le queremos enviar la comunicación. La sintaxis de este comando es ser la siguiente:

⟨*SEND*⟩ ::= '`send`' '`[`' ⟨*expr_list*⟩ '`]`' '`to`' ⟨*mexp*⟩

Donde *expr_list* es una lista de expresiones, que puede ser vacía. Las expresiones son evaluadas y se envían los valores en la comunicación. *mexp* es un identificador que tiene asociado una dirección de buzón de un actor.

Comportamiento de reemplazo

El propósito de los comandos es especificar las acciones que pueden ocurrir. Se mostraron los comandos para crear nuevos actores y para crear nuevas comunicaciones. También se necesita un comando para definir el comportamiento de reemplazo. La sintaxis para este último tiene la forma:

⟨*BECOME*⟩ ::= '`become`' *NombreComportamiento* '`(`' ⟨*expr_list*⟩ '`)`'
| '`become`' *mexp*

donde *mexp* es una dirección de buzón, en este caso reenvía todos los mensajes al nuevo buzón. Por ejemplo:

`become` *link*

`become` $Comp(1, 2, 3)$

donde *link* es alguna dirección de buzón, y $Comp(1, 2, 3)$ hace referencia al comportamiento *Comp*, y su *acquaiantence-list* son los valores 1, 2 y 3. La sintaxis del comportamiento se describe en la sección 2.2.2

Otros comandos

Para completar el lenguaje, se agregan la composición secuencial y un condicional.

⟨*IFCOMP*⟩ :== '`if` `<bexp>` '`then`' ⟨*command*⟩ '`else`' ⟨*command*⟩ '`end if`'
| ⟨*command*⟩ ; ⟨*command*⟩

donde:

if-then-else , después 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 Dos comandos que se ejecutan de manera secuencial.

2.3. Ejemplos

En esta sección mostraremos dos ejemplos escritos en *SAL* y algunas particularidades del lenguaje. Primero se presenta el código del ejemplo, a continuación una breve descripción línea por línea de la funcionalidad, y para terminar se mencionan notas sobre su funcionamiento.

2.3.1. Cálculo del factorial

Se usa este clásico ejemplo, para mostrar que el paso de mensaje se puede usar como una estructura de control. En un lenguaje imperativo una función recursiva está implementada utilizando una pila de llamadas. Usar esta pila implica que la función factorial solo puede procesar un único cálculo a la vez; una vez que termina el cálculo puede aceptar nuevamente otro. En los lenguajes secuenciales no existe ningún mecanismo que permita distribuir el cálculo del factorial o que permita concurrentemente procesar más de una petición.

La implementación utilizando el modelo de actores depende de crear uno o más trabajadores que están a la espera de un entero, multiplicarlo y pasarlo al siguiente actor en la cadena. El actor factorial está dispuesto a procesar concurrentemente la próxima comunicación. Esta incluye la dirección de buzón a cual se debe enviar el calculo del factorial.

Está implementación del factorial está adaptada de [1]. Depende de un actor Main que envía al actor Factorial el valor a calcular (en el caso del ejemplo el valor 3). La palabra reservada *self* hace referencia a este buzón, correspondiente al actor que está procesando la comunicación.

Como se observa en el ejemplo, las comunicaciones pueden venir tanto del actor Main como del actor Factorial .

```
1  def Factorial()[val, customer]
2    if val = 0 then
3      send [1] to customer
4    else
5      let cont = new FactorialWorker(val, customer)
6          in send [val - 1, cont] to self
7    end if
8    become Factorial()
9  end def
10
11 def FactorialWorker(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
```

Línea 1 *Factorial* no recibe ningún parámetro en el momento de ser inicializado, pero sí recibe dos parámetros cuando procesa una comunicación, la lista [val , customer] con un entero y una dirección de un buzón respectivamente.

Línea 3 Envía una lista con el valor 1 al actor con buzón customer.

Línea 5 Crea un actor de tipo FactorialWorker. En este caso se utiliza *new*, ya que se está creando un nuevo buzón, y éste se asigna a la variable *cont*. Cuando se asigna el nuevo comportamiento, éste recibe los parámetros *val* y *customer*.

Línea 6 Envía un mensaje a la dirección del buzón propio utilizando la palabra reservada *self*, con la lista $val - 1$ y la dirección del buzón que se acaba de crear.

Línea 8 Asigna como siguiente comportamiento a *Factorial*(), sin parámetros ya que *Factorial* no recibe ningún parámetro a la inicialización.

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

Línea 12 Envía la multiplicación $n * m$ como lista a la dirección del buzón *customer*

Líneas 15-18 Inicializa el actor *Factorial* y le envía a este la lista 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 que espera un mensaje con un número y multiplica este número por **n**, luego se envía el resultado al buzón de *customer*.

También, se envía un mensaje a sí mismo para evaluar el factorial de **n - 1**, y como dirección de cliente utiliza la dirección del buzón del actor recientemente creado. Es decir, el resultado de $fact(n-1)$ se le pasará al actor recientemente creado que lo multiplicará por n.

Esto establece una red de actores que multiplican el valor indicado y enviarán el cálculo al siguiente actor en la red. Es el último actor en la red el que lo enviará a quien originalmente lo pidió.

En la figura 2.3 se puede ver que el actor *Factorial*() recibe como comunicación la lista $[3, c]$, esto hace que ocurran dos cosas:

- Se cree un actor nuevo *FactorialWorker* con buzón $c1$, este recibe dos parámetros en la inicialización: el valor 3 y el buzón inicial que recibió como comunicación, es decir quien pidió originalmente la computación del factorial de 3.

- Se envíe a sí mismo el mensaje $[2, c1]$, este inicia el cálculo del factorial de 2, es decir el cálculo de $fact(n-1)$, el paso recursivo.

Ahora $c1$ es quien pide el calculo del factorial de 2. Esto se repite hasta que el factorial de 0 a calcular sea 0.

Cuando el primer elemento de la comunicación es cero, hace que se le envíe al actor cuya dirección de buzón fue recién recibida, la lista con el valor uno.

- a le envía el valor 1 a $c3$.
- $c3$ multiplica $1 * 1$ y se lo envía a $c2$.
- $c2$ multiplica $2 * 1$ y se lo envía a $c1$.
- $c1$ multiplica $3 * 2$ y se lo envía a c.

Recordamos que c había pedido el calculo del factorial 3 en primer lugar.

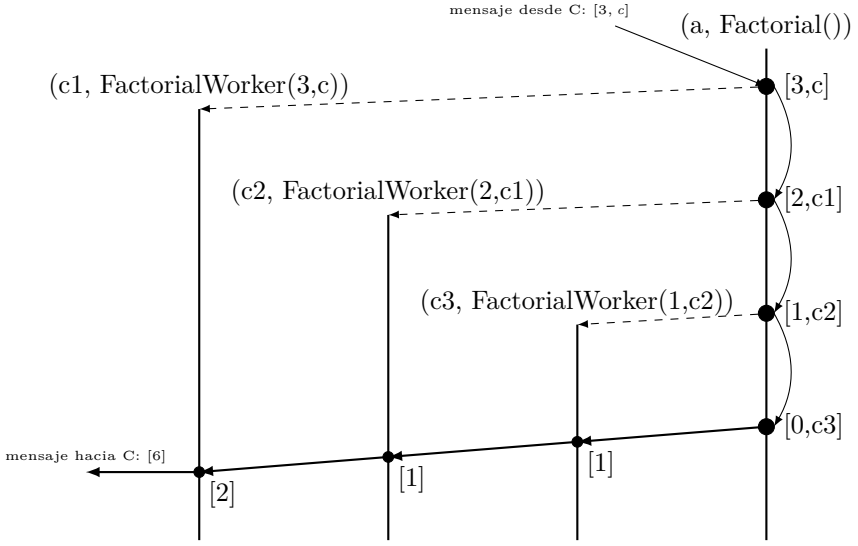

Figura 2.3: El diagrama ilustra el cálculo del factorial de 3. Todo el resultado es enviado al actor c. Las líneas verticales indican el paso del tiempo, las de punto indican creación de actores y las otras flechas envío de mensaje. La lista superior indica la dirección del buzón y el tipo de actor con los parámetros con los que fueron inicializados.

2.3.2. Una pila usando actores

Otro ejemplo que podemos encontrar en [1] es el de una pila, que está implementada con una lista enlazada. Se utiliza la dirección de un buzón como un puntero a un nodo de esta lista.

Tiene dos operaciones básicas: apilar (*push*), que coloca un nodo en la pila, y su operación inversa, sacar (*pop*), que remueve el último elemento agregado en la pila.

```
1  def Node(content, link)
2    case ['pop', customer]:
3      send content to customer;
4      become link;
5    case ['push', newcontent]:
6      let P = new Node(content, link)
7        in become Node(newcontent, P)
8  end def
9
10 def Main()
11   let stack = new Node(10, Nil)
12     in send [push, 20] to stack;
13        send [push, 30] to stack;
14   end
15 end
```

Línea 1 *Node* recibe dos parámetros, *content* que es un entero, el valor que tiene que guardar el nodo y *link* que es una dirección de buzón, es el siguiente actor en la pila.

Línea 2 Si la operación es *'pop'*, guarda en *customer* la dirección de buzón.

Línea 3 Envía el contenido del nodo a la dirección de buzón *customer*

Línea 4 La instrucción *become*, en este caso, hace que se reenvíen todos los mensajes a la dirección de buzón *link*.

Línea 5 Si la operación es *'push'*, guarda en *newcontent* el valor del entero recibido.

Línea 6 Crea un nuevo actor con los parámetros de inicialización *content* y *link*.

Línea 7 Asigna el siguiente comportamiento, como *Node* con los parámetros *newcontent* y la dirección de buzón del actor recién creado P.

Líneas 11-13 Crea una nueva pila con uno nodo con valor 10, y envía dos operaciones *push* con los valores 20 y 30.

El comportamiento *Node* funciona como una lista enlazada, donde en vez de tener direcciones de memoria tenemos direcciones de buzón. El primer parámetro (*content*) es el contenido a guardar y el segundo (*link*) es el actor siguiente en la pila, o sea el puntero al siguiente elemento.

Cuando el primer valor de la lista es *pop*, se envía el valor que contiene el nodo al buzón *customer* y se reenvían todos los mensajes a *link*. Todas las futuras operaciones *push* y *pop* las recibe este nodo, es decir que ahora es la "cabeza" de la pila. Esto guarda un parecido a mover un "puntero".

Cuando el primer valor de la lista es *push*, la pila crea un nuevo *node* que será el nodo que quedará siguiente en la red. Se puede ver que esto ocurre en las líneas 7 y 8. Se copia en P el nodo actual, y crea un nuevo nodo que es la nueva "cabeza".

Puede observarse en el ejemplo, que el primer nodo creado tiene como valor *Nil*, esto es simplemente una referencia nula.

Capítulo 3

Preliminares

En en la primera sección de este capítulo se exploran algunas particularidades del paralelismo en *CSP*, tales como, paralelismo sincrónico, alfabetizado, entrelazado y generalizado. En la segunda sección se muestra algunas construcciones en *CSPm* que resultan útiles.

La sección de *CSP* no pretende ser una introducción al lenguaje; se asume que el lector tiene cierta familiaridad con él. Para una introducción se puede consultar [7], para una referencia completa [14]

3.1. Algunas construcciones de CSP

En esta sección se muestran algunos de los operadores de paralelismo de *CSP*, seguido de algunos ejemplos de uso y su relación con el modelo de actores.

Paralelismo sincrónico

El operador más simple de *CSP* es el que está dispuesto a sincronizar todos los eventos. Es decir, ambos procesos compuestos por este operador avanzan cuando encuentran un evento que ambos están dispuestos a sincronizar. Por ejemplo:

$$\begin{aligned} P_1 &= a \rightarrow P_1 \\ P_2 &= a \rightarrow P_2 \\ SYSTEM &= P_1 \parallel P_2 \end{aligned}$$

donde P_1 y P_2 sincronizan en el evento a.

Cuando utilizamos procesos parametrizados muchas veces es útil enviar información. El siguiente ejemplo muestra esto:

$$\begin{aligned} P_1 &= canal!1 \rightarrow STOP \\ P_2 &= canal?x \rightarrow P(x) \\ SYSTEM &= P_1 \parallel P_2 \end{aligned}$$

donde $canal!1$ es quien envía el valor 1 y $canal?x$ lo recibe. Para entender un poco más cómo funciona la notación que involucra $\langle ? \rangle$ y $\langle ! \rangle$, supongamos que x puede tomar los valores 1, 2 y 3. La expresión $canal?x$ equivale a un proceso que está dispuesto a sincronizar con todos estos potenciales valores:

$$\begin{aligned} P_2 = & \ canal.1 \rightarrow STOP \\ \Box & \ canal.2 \rightarrow STOP \\ \Box & \ canal.3 \rightarrow STOP \end{aligned}$$

Como x es una variable libre, y el evento que termina sincronizando es $canal.1$ esta toma el valor 1. En realidad el paso de información es ficticio; todo el tiempo se está sincronizando en eventos.

Paralelismo alfabetizado

Mientras más procesos combinemos utilizando el operador $\parallel$, más procesos tienen que ponerse de acuerdo en los eventos a sincronizar si se pone en paralelo los procesos P y Q no necesariamente todas las comunicaciones de P son para Q.

Si X e Y son dos conjunto de eventos, $P \ {}_{\mathrm{X}} \parallel {}_{\mathrm{Y}} \ Q$ es la composición paralela en donde P tiene solo permitido comunicar los eventos X y donde Q tiene sólo permitido comunicar los eventos Y, y únicamente tienen que ponerse de acuerdo en la intersección $X \cap Y$. Por ejemplo:

$$(a \rightarrow b \rightarrow b \rightarrow STOP) \ {}_{\{\mathrm{a,\ b}\}} \parallel {}_{\{\mathrm{b,\ c}\}} \ (b \rightarrow c \rightarrow b \rightarrow STOP)$$

Se comporta como:

$$(a \rightarrow b \rightarrow c \rightarrow b \rightarrow STOP)$$

Entrelazado

Los operadores $\parallel$ y ${}_{\mathrm{X}} \parallel {}_{\mathrm{Y}}$ tienen la propiedad que todos los procesos involucrados tienen que sincronizar algún evento. Utilizando el operador de entrelazado ($|||$), cada proceso corre independiente de cualquier otro. Se nota $P \ ||| \ Q$.

Paralelismo generalizado

Existe una forma general de escribir todos los operadores vistos utilizando el operador de paralelismo generalizado $P \underset{X}{\|} Q$, donde P y Q solo tienen que ponerse de acuerdo en los eventos pertenecientes a X y los eventos que están por fuera de X se procesan independientemente.

Podemos escribir el operador de entrelazado usando la siguiente equivalencia:

$$P \,|||\, Q = P \underset{\{\}}{\|} Q$$

Podemos escribir el operador de paralelismo alfabetizado como:

$$P \;_{\mathrm{X}} \|_{\mathrm{Y}}\; Q = P \underset{X \cap Y}{\|} Q$$

Σ son todos los eventos posibles en un sistema dado podemos definir el operador de paralelismo sincrónico de la siguiente forma:

$$P \,\|\, Q = P \underset{\Sigma}{\|} Q$$

Cadena de caracteres

CSP no tiene soporte para cadena de caracteres, en esencia solo son procesos y eventos. Será util en el capitulo que se define el modelo poder utilizar dentro de las secuencias algún tipo de cadena de caracteres. Estas son inmutables y la única operación que se define sobre estas es la comparación. Se representa utilizando una secuencia alfanumérica, encerradas entre comillas simples. Por ejemplo:

$$'cadena1'$$
$$'cadena2'$$
$$\langle 'cadena1', ' cadena2' \rangle$$

Actores y CSP

Como vimos en el capítulo anterior, *CSP* es sincrónico, mientras que el paso de mensajes o envío de comunicaciones en el sistema de actores no lo es. Si se quiere transmitir información entre dos procesos en *CSP* lo escribimos (como vimos en el apartado de paralelismo sincrónico),de la siguiente forma:

$$\begin{aligned} P_1 &= canal!1 \rightarrow STOP \\ P_2 &= canal?x \rightarrow STOP \\ SYSTEM &= P_1 \;\|\; P_2 \end{aligned}$$

Para poder desacoplar el envío de la recepción del mensaje, se puede utilizar una estructura intermedia de *BUFFER*, cuya especificación es:

$$\begin{aligned} BUFFER =& enviar?x \rightarrow recibir!x \rightarrow BUFFER \\ P_1 =& enviar!1 \rightarrow STOP \\ P_2 =& recibir?x \rightarrow STOP \\ SYSTEM =& (P_1 \;|||\; P_2) \;\; || \; BUFFER \end{aligned}$$

Como la comunicación es desde P_1 hacia *BUFFER* y desde *BUFFER* hacia P_2, no hay ninguna comunicación entre P_1 y P_2. Por esto se utiliza el operador de entrelazado.

En *CSP* no existe el concepto de instancia, y se debe definir la red de procesos desde el comienzo. Es decir, no podemos crear nuevos procesos desde un proceso. Por lo tanto, simularemos la creación activando cada instancia mediante un evento especial. Para iniciar n procesos de tipo P se escriben los siguientes procesos en *CSP*:

$$\begin{aligned} P =& \texttt{comportamiento-de-P} \rightarrow STOP \\ P_1 =& iniciar_1 \rightarrow P \\ P_2 =& iniciar_2 \rightarrow P \\ & \dots \\ P_n =& iniciar_n \rightarrow P \\ SYSTEM =& P_1 \;|||\; P_2 \;|||\; \dots \;|||\; P_n \end{aligned}$$

donde $iniciar_i$ es el evento especial que da inicio a P. Se utiliza el operador de entrelazado para combinar los procesos $P_1, P_2, \dots, P_n$, ya que no tiene que haber ninguna comunicación entre ellos. En el caso de existir algún evento compartido, este debería ser desestimado y no generar un encuentro. Con estos elementos podemos crear un proceso y enviar una comunicación de manera asincrónica. Se puede ver esto en el siguiente ejemplo:

$$\begin{aligned} BUFFER_1 &= enviar.1?x \rightarrow recibir.1!x \rightarrow BUFFER_1 \\ BUFFER_2 &= enviar.2?x \rightarrow recibir.2!x \rightarrow BUFFER_2 \\ SUMA &= inicia_{suma} \rightarrow recibir.1?x \rightarrow enviar.2!(x+1) \rightarrow STOP \\ CLIENTE &= inicia_{suma} \rightarrow enviar.1!2 \rightarrow recibir.2?x \rightarrow STOP \\ BUFFER &= BUFFER_1 \;|||\; BUFFER_2 \\ SYSTEM &= (SUMA \underset{\{inicia_{suma}\}}{||} CLIENTE) \underset{Y}{||} BUFFER \end{aligned} \tag{3.1}$$

Es decir que *CLIENTE* inicia el proceso *SUMA*, y le envía un 2. Este envío es

asíncrono por $BUFFER_1$. Cuando *SUMA* recibe este 2, crea una nuevo mensaje y se lo envía a *CLIENTE* de manera asincrónica, con el valor que recibió incrementado en uno. En la composición de *SYSTEM* se puede ver que el único evento que se sincroniza entre *SUMA* y *CLIENTE* es $inicia_{suma}$. En el otro operador paralelo los valores de Y vienen dados por los eventos en los que la composición de *SUMA* y *CLIENT* sincronizan con *BUFFER*. Para esto deberíamos saber qué valores puede tomar x. Asumiendo que toma los valores 1, 2 y 3. Los eventos a sincronizar serían el conjunto generado por $\{m : [1,2], n : [1,2,3] \mid recibir.m.n\} \cup \{m : [1,2], n : [1,2,3] \mid enviar.m.n\}$, es decir todos los eventos inherentes a *BUFFER*.

Este último ejemplo muestra dos de los aspectos que se desarrollarán en el capítulo siguiente: cómo desacoplar el envío de mensajes y óomo simular la creación de un proceso. También puede verse el uso de los distintos operadores paralelo.

3.2. El lenguaje CSPm

CSPm es un lenguaje funcional, que tiene una integración para definir procesos de *CSP*. También permite realizar aserciones sobre los procesos de *CSP* resultantes. Este lenguaje es el que utiliza la plataforma *FDR*. En esta sección se describirán algunas de las construcciones de *CSP* en *CSPm* y algunas construcciones propias de *CSPm*.

Tipos algebraicos

Permite declarar tipos estructurados, son similares a las declaraciones de tipo *data* de Haskell. La más simple de las declaraciones es utilizando constantes.

```
datatype ColorSimple = Rojo | Verde | Azul
```

Esto declara `Rojo`, `Verde` y `Azul`=, como símbolos del tipo `ColorSimple`, y vincula `ColorSimple` al conjunto `{Rojo, Verde, Azul}`. Estos tipos de datos puede tener parámetros. Por ejemplo, se pude agregar un constructor de datos `RGB`, a saber:

```
datatype ColorComplejo = Nombre.ColorSimple | RGB.{0..255}.{0..255}.{0..255}
```

Esto declara `Nombre`, como un constructor de datos, de tipo `ColorSimple` ⇒ `ColorComplejo` y a `RGB` como un constructor de datos de tipo `Int` ⇒ `Int` ⇒ `Int` ⇒ `ColorComplejo` y `ColorComplejo` es el conjunto:

$$\{Nombre.c \mid c \leftarrow ColorSimple\} \cup \{RGB.r.g.b \mid r \leftarrow \{0 \ldots 255\}, g \leftarrow \{0 \ldots 255\}, b \leftarrow \{0 \ldots 255\}\}$$

Si se declara un tipo de datos T, entonces a T se adjunta el conjunto de todos los valores de tipo de datos posibles que se pueden construir.

Canales

Los canales de *CSPm* son utilizados para crear eventos, y se declaran de una manera similar a los tipos de datos. Por ejemplo:

```
channel estaListo
channel x, y : {0..1}.Bool
```

Declara tres canales, uno que no toma parámetros (listo es de tipo Event), y dos que tienen dos componentes. Cualquier valor del conjunto `{0,1}` y un booleano. El conjunto de los eventos definidos es el siguiente:

```
{ estaListo, x.0.false, x.1.false, x.0.true, x.1.true, y.0.false, y.1.false, y.0.true, y.1.tr
```

Estos eventos pueden ser parte de la declaración de procesos como por ejemplo `{P = x?a?b -> STOP}`.

Búsqueda de patrones

Es posible en *CSP* que los valores puedan ser buscados por coincidencia de patrones. Por ejemplo, la siguiente función toma un entero, se puede usar la búsqueda de patrones para especificar un comportamiento diferente dependiendo de este argumento:

```
f(0) = True
f(1) = False
f(_) = error("Error")
```

Funciona de manera similar a Haskell, también se pueden utilizar otras construcciones como secuencias o algún tipo algebraico.

Operadores replicados

FDR tiene una versión replicada o indexada de alguno de sus operadores. Estos proveen una forma simple de construir un proceso que consiste en una serie de procesos compuestos utilizando el mismo operador. Por ejemplo se define P de la siguiente manera:, `P :: (Int) -> Proc` luego, `||| x: { 0..2 } @ P(x)` evalúa P para cada valor de x en el conjunto dado y los compone utilizando el operador de entrelazado. Por lo tanto, lo anterior es equivalente a `P (0) ||| P (1) ||| P (2)`.

La forma general de un operador replicado es:

```
op <declaraciones> @ P
```

donde op es un operador que puede ser el de entrelazado, selección interna, etc. P es la definición de un proceso, puede hacer uso de las variables definidas por las declaraciones. Cada uno de los operadores evalúa P para cada valor que toman las declaraciones antes de componerlas juntas usando op.

En *CSPm* las declaraciones puede ser construcciones por comprensión, por ejemplo las que generan nuevos conjuntos basados en conjuntos existentes. Por ejemplo, el conjunto por comprensión: `{x + 1 | x <- xs}` incrementa cada elemento del conjunto xs en 1.

Otra forma de construcción por compresión es utilizar la siguiente expresión: `{| CANAL |}`, que genera el conjunto de todos los elementos del canal. Por ejemplo, dada la siguiente definición de canal `channel X : {0..1}.Bool`, el conjunto generado por `{| X |}`es: `{X.0.false, X.0.true, X.1.false, X.1.true}`.

Otras construcciones

En esta sección se muestran algunas de las conversiones útiles para entender un programa en *CSPm*.

Tabla de conversión para secuencias:

seq a	`Seq(a)`
$\langle\rangle$	`< >`
$\langle 1,2,3\rangle$	`<1,2,3>`
$s = \langle\rangle$	`Null(s)`
$s \frown t$	`s^t`
$\#s$	`#s`
head s	`head(s)`
tail s	`tail(s)`
$\frown / s$	`concat(s)`
$x \in \text{ran } s$	`elem(x,s)`
ran s	`set(s)`

Tabla de conversión para la definición de procesos:

Stop	`STOP`
Skip	`SKIP`
$c \rightarrow p$	`c -> p`
$c?x \rightarrow p$	`c?x -> p`
$c!v \rightarrow p$	`c!v -> p`
$p \Box q$	`p [] q`
$p \sqcap q$	`p |~| q`
$p \mid\mid\mid q$	`p ||| q`
$p \parallel q$	`p || q`
$p \;{}_{\text{X}}\!\parallel_{\text{Y}}\; q$	`p |[|X|]| q`
$p \underset{X}{\parallel} q$	`p |[x||y]| q`

Capítulo 4

El modelo de actores en CSP

En este capítulo se modela el sistema de actores utilizando *CSP*. Este modela los diferentes funcionalidades definidas por *SAL*, Tales como crear nuevos actores, enviar mensajes, y definir un comportamiento de reemplazo.

Se comienza por una descripción detallada de cada componete, luego se introducen ejemplos escritos en el lenguaje de programación *SAL* y su equivalente en *CSP*. Se presenta una función que traduce de *SAL* a *CSP*. Para terminar mostrando algunas particularidades sobre el modelo propuesto al utilizar la herramienta *FDR*.

4.1. Describiendo el sistema de actores

En esta sección se describen los dos componentes fundamentales para construir el modelo. Primero se verá el buzón, luego dos ecuaciones necesarias para crear la red de actores y para terminar se mostrará cómo es posible describir comportamientos usando *CSP*.

4.1.1. Buzón

La naturaleza de *CSP* es sincrónica y los actores no lo son. Para esto se necesita desacoplar el envío de mensajes de la recepción. Se utiliza una estructura intermedia que actúa de buzón, y dos canales que sirven para comunicarse con ella.

Un buzón puede guardar más de una comunicación en su interior. Las comunicaciones se agregan al final, y se consumen sobre el principio. En este sentido, es una cola de tipo FIFO. [1].

La ecuación de un buzón es la siguiente:

[1] Del acrónimo inglés de First In, First Out ("primero en entrar, primero en salir")

$$
\begin{aligned}
&Mailbox(mailboxId, \langle\rangle) = \\
&\quad CommSend?mailboxId.x \rightarrow Mailbox(mailboxId, \langle x \rangle) \\
&Mailbox(i, \langle x \rangle \frown xs) = \\
&\quad CommRecv!mailboxId.x \rightarrow Mailbox(mailboxId, xs) \\
&\quad \Box \\
&\quad CommSend?mailboxId.y \rightarrow Mailbox(mailboxId, \langle x \rangle \frown xs \frown \langle y \rangle)
\end{aligned}
\tag{4.1}
$$

donde:

$Mailbox(i, \langle\rangle)$ es cuando el buzón está vacío, y $Mailbox(i, \langle x \rangle \frown xs)$ es cuando tiene al menos un mensaje.

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

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

en $mailboxId.x$ y $mailboxId.y$, *mailboxId* es la dirección del buzón, y es la información que se recibe desde un actor y x es la información que se envía hacia un actor.

El comportamiento del proceso *Mailbox* depende de su estado:

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

Por cada actor en la red existe un buzón con el mismo *mailboxId* asociado. Para esto utilizamos la siguiente ecuación:

$$Mailboxes = |||_{actor:MailboxIDS}\ Mailbox(actor, \langle\rangle) \tag{4.2}$$

donde *Mailboxes* representa todos los buzones puestos en paralelo utilizando el operador de entrelazado, ya que no existe comunicación entre buzones. *MailboxIDS* es un conjunto de enteros que representan un identificador único para los buzones. Tendrá que tener tantos elementos, como actores sean necesarios.

4.1.2. Crear nuevos actores

Dentro de las acciones que un actor efectúa está la de crear otro actor. En *CSP* los actores corresponden a procesos. Como todos los procesos tienen que estar definidos desde el comienzo, en *CSP* no existe la posibilidad de crear dinámicamente un nuevo proceso.

Para identificar qué actor se está queriendo crear, se utiliza el siguiente conjunto:

$$ActorID = \Big\{ ACTOR_1, ACTOR_2, \ldots, ACTOR_K, MAIN \Big\} \tag{4.3}$$

Para resolver el problema asociado a no poder instanciar nuevos procesos en *CSP*, se presentan a continuación dos abstracciones.

La primera abstracción es un preámbulo a un comportamiento que le asigna a este los parámetros *acquaiantence-list* y al mismo tiempo le otorga al actor su identificador único de buzón. Es en esta abstracción cuando se crea la red de procesos que van a estar a la espera de un evento que da inicio al actor. Utilizamos para esto un conjunto de procesos puestos en paralelo.

$$ks = |||_{i:\{1..N_k\}} Create.ACTOR_k?self? < p1, p2 > \rightarrow Comportamiento_K(self, p1, p2) \tag{4.4}$$

donde:

- N_k es un entero que define la cantidad de actores de tipo $ACTOR_K$ que van a estar disponibles.
- $Create.ACTOR_k?self? < p1, p2 >$, sincroniza en el canal $Create.ACTOR_k$, donde $ACTOR_k$ es parte del conjunto *ActorID*. Recibe el identificador de buzón *self* como parámetro y la lista $< p1, p2 >$. En este caso se supone que K tiene solo dos parámetros de tipo *acquaiantence-list*.
- $K(self, p1, p2)$ llama al proceso parametrizado definido como $Comportamiento_K$ con los parámetros *self*, $p1$ y $p2$. K representa el comportamiento del actor

En realidad, la creación es algo ficticio, ya que tenemos una red de procesos *CSP* esperando al evento $Create.ACTOR_k$ para arrancar con el comportamiento definido.

El proceso que representa a todos los actores que van a ser iniciados en paralelo, utiliza el operador de entrelazado. Una vez sincronizado en el mensaje *Create*, se ejecuta el comportamiento que obtiene el identificador del buzón y los parámetros recibidos.

La segunda abstracción es un proceso que vuelve asincrónico el mensaje de creación, de la creación como tal. Para esto se utiliza una estructura intermedia *Create* y un canal para comunicarse con ella. *Create* se define de la siguiente

$$\begin{aligned} &Create(mailboxId) = \\ &\qquad CreateAsk?actorId!mailboxId?m \rightarrow \\ &\qquad Create.actorId?mailboxId!m \rightarrow \\ &\qquad STOP \end{aligned} \tag{4.5}$$

donde:

$CreateAsk?actorId!mailboxId?m$ sincroniza en el canal *CreateAsk*. Recibe *actorId*, el identificador de actor a crear. Envía el identificador de buzón *MailboxID* y recibe la lista de valores m.

$Create.actorId!mailboxId!m$ sincroniza en el canal $Create.actorId$, visto en la ecuación (4.5). Envía el identificador de buzón *mailboxID* y la lista de valores m.

Es necesario crear tantos procesos como elementos existan en el conjunto *MailboxIDS*. Para esto se utiliza la siguiente ecuación:

$$Creates = |||_{mailboxId:MailboxIDS}\ Create(mailboxId) \tag{4.6}$$

4.1.3. Definición de comportamientos

La idea de comportamiento fue introducida en la sección 2.2.2. 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 qué esta procesando la comunicación. A continuación se muestra cómo realizar estas operaciones en *CSP* utilizando las construcciones anteriores.

Enviar nuevas comunicaciones

Para enviar la comunicación al actor con dirección de buzón d, y la lista de valores $< p_1, p_2, \ldots p_n >$, utilizamos la siguiente termino de *CSP*:

$$CommSend.d! < p_1, p_2, \ldots, p_n >$$

Este evento sincronizará con el evento correspondiente al proceso *Mailbox*, visto en la ecuación (4.1), el cual almacena $< p_1, p_2, \ldots, p_n >$ en una lista. Mas tarde *Mailbox* tomará este mensaje y lo enviará a otro actor.

Como se verá en los ejemplos de la sección siguiente, todos los comportamiento de los diferentes actores comienzan con una construcción *CommRecv*.

Crear nuevos actores

Para crear un actor con un comportamiento incial $Comportamiento_m$ con una lista de parametros $< p_1, p_2, \ldots p_m >$, se utiliza el siguiente termino de *CSP*:

$$CreateAsk.Actor_m?buzon! < p_1, p_2, \ldots, p_m >$$

donde *buzon* es la dirección de buzón recibida y que corresponde a la dirección del actor creado.

Comportamiento de reemplazo

Para definir un comportamiento de reemplazo $Comportamiento_k$ con una lista de parámetros $< p_1, p_2, \ldots p_m >$, se deberá utilizar el siguiente termino:

$$Comportamiento_k(self, p_1, p_2, \ldots, p_m)$$

donde $Comportamiento_k$ es un proceso *CSP* que define el comportamiento y *self* es una variable definida por el preludio visto en la ecuación (4.4). En caso de no querer

especificar comportamiento de reemplazo, se deberá usar el término *STOP* de *CSP* para indicar que este actor finalizó.

En caso que el comportamiento de reemplazo sea la dirección de buzón de otro actor, utilizamos el siguiente proceso:

$$fwd(o, d) = CommRecv.o?msg \rightarrow CommSend.d!msg \rightarrow fwd(o, d)$$

El proceso anterior, reenvía todas las comunicaciones desde el buzón o, al buzón d. Dada la dirección de buzón *nuevoBuzon*, el *fwd* se utiliza de la siguiente manera:

$$fwd(self, nuevoBuzon)$$

4.2. Ejemplos

En esta sección se muestran cuatro ejemplos. Los dos primeros son los vistos en la sección 2.3.1 y 2.3.2 modelados en *CSP* en vez de *SAL*. Los siguientes dos son nuevos, uno es la estructura de datos **cola** y el otro un ejercicio tomado del libro *Programming Erlang* [4].

4.2.1. Ejemplo: cálculo de factorial en CSP

En esta sección se describe el funcionamiento del factorial. Es una implementación en *CSP* del ejemplo antes visto en *SAL*. Está compuesto por dos comportamientos *Factorial* y *FactorialWorker*.

Continuando con la mecánica del capítulo anterior, primero se presenta el código en *CSP*, luego se comentan las líneas de interés, para terminar con un pequeño detalle del funcionamiento.

El primero de los comportamientos, es el de *Factorial* que viene dado por la siguiente formula:

$$
\begin{array}{l}
\textsf{process} \\
\quad Factorial(self) = \\
\qquad CommRecv?self.\langle mailboxClient, k\rangle \rightarrow \\
\qquad\quad (CommSend!mailboxClient.\langle 1\rangle \rightarrow \\
\qquad\quad Factorial(self)) \\
\qquad \lceil\ k == 0\ \rceil \\
\qquad\quad (CreateAsk.factorialWorker?pid!\langle k, mailboxClient\rangle \rightarrow \\
\qquad\quad CommSend!self.\langle pid, k-1\rangle \rightarrow \\
\qquad\quad Factorial(self))
\end{array}
$$

donde:

$CommRecv?self. < mailboxClient, k >$ espera recibir una comunicación donde el primer elemento es un buzón y el segundo un entero.

$CommSend!mailboxClient.\langle 1 \rangle$ envía una comunicación al buzón *mailboxClient* con un solo elemento: el entero 1.

$CreateAsk.factorialWorker?pid!\langle k, mailboxClient \rangle$ crea un nuevo actor de tipo *FactorialWorker*, guarda en *pid* la dirección del buzón. Inicializa los valores *acquaiantence-list* con el entero k y el buzón *mailboxClient*.

$CommSend!self. < pid, k-1 >$ se auto envía un mensaje con el valor de buzón del actor creado en la linea anterior, y el entero k decrementado en uno.

$Factorial(self)$ define como siguiente comportamiento el proceso *Factorial* para el buzón *self*.

Cuando recibe un entero distinto de cero ejecuta dos acciones, crea un actor *FactorialWorker* y se envía un mensaje a sí mismo para evaluar el factorial de $n-1$. En este caso el comportamiento de reemplazo para el buzón actual no cambia. Para una descripción mas detallada revisar la sección 2.3.1

El segundo de los comportamientos, es el de *FactorialWorker* que viene dado de la siguiente forma:

```
process
  FactorialWorker(self, k, mailboxClient) =
    CommRecv.self?⟨n⟩ →
      CommSend.mailboxClient!⟨n ∗ k⟩ →
      STOP
```

donde:

$CommRecv.self?\langle n \rangle$ espera una comunicación que contenga, un solo elemento, y que este sea un entero.

$CommSend.mailboxClient!\langle n*k \rangle$ envía el resultado de la multiplicación a la dirección de buzón *mailboxClient*

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 del valor recibido por k y se lo envía a *mailboxClient*. En este caso no cuenta con comportamiento de reemplazo, entonces termina con $STOP$.

Tanto para *Factorial* y para *FactorialWorker* faltan definir los procesos que van a dar inicio a los actores. Los cuales se encuentran definidos en la sección 4.1.2, en la ecuación (4.4).

4.2.2. Ejemplo: una pila

En este ejemplo se construye una estructura de datos de tipo **pila**, la cual está compuesta de un solo comportamiento *node* que es el que se encarga de recibir las operaciones $'push'$ y $'pop'$.

El comportamiento de *node* está definido de la siguiente forma:

```
process
  Node(self, content, link) =
    CommRecv.self?⟨'push', newContent⟩ →
    CreateAsk.node?newNode!⟨content, link⟩ →
    Node(self, newContent, newNode)
  □
    CommRecv.self?⟨'pop', client⟩ →
    CommSend.client!⟨content⟩ →
    fwd(self, link)
```

donde:

$CommRecv.self?\langle'push', newContent\rangle$ espera recibir un mensaje, donde el primer elemento de la lista es la constante $'push'$ y el segundo elemento es el contenido a guardar.

$CreateAsk.node?newNode!\langle content, link\rangle$ crea un actor de un tipo *Node* y guarda en *newNode* el valor de la dirección del buzón. Inicializa los valores *acquaiantencelist* con valor *content* y el buzón *link*.

$Node(self, newContent, newNode)$ define como comportamiento para el buzón *self*, el mismo comportamiento con los parámetros, *newContent* y el buzón creado en la linea anterior.

$CommRecv.self?\langle'pop', client\rangle$ espera recibir un mensaje, donde el primer elemento de la lista sea la constante $'pop'$, guarda en *client* el valor del segundo elemento de la lista.

$CommSend.client!\langle content\rangle$ envía una comunicación al buzón *client* la lista con el valor *content*.

$fwd(self, link)$ se comporta como el proceso *fwd*.

Cuando la operación es de tipo $'pop'$, se envía el valor que contiene el nodo al buzón *client* y se reenvían todos los mensajes a *link*, todas las futuras operaciones $'push'$ y $'pop'$ las recibe este nodo.

Cuando la operación es de tipo *'push'*, la pila crea un nuevo actor *node*. Se copia en *Node.newNode* el nodo actual, y se reemplaza el contenido del nodo actual con el contenido recibido. Esto puede verse como el reemplazo de la cabeza de la pila.

4.2.3. Ejemplo: una cola

En este ejemplo se explora cómo construir una estructura de datos de tipo *cola*, la cual se modela como si fuera una máquina de estados, dónde cada comportamiento corresponde a un estado y las comunicaciones son quienes disparan las transiciones de estados. Primero se muestra el ejemplo escrito utilizando *SAL*, y luego su equivalente en *CSP*.

Existen dos operaciones posibles para efectuarse en una *cola QUEUE* o encolar, y *DEQUEUE* o desencolar. La primera operación agrega un nodo al final, y la segunda lo remueve del principio.

Tanto el ejemplo en *SAL* como el de *CSP* tienen cuatro comportamientos:

node Guarda el contenido y una referencia al siguiente nodo en la *cola*; son los eslabones de construcción de una suerte de lista enlazada.

queue Quien se encarga de gestionar los nodos. Tiene referencia a dos nodos, el primero y el último, para poder remover el primero, y agregar sobre el final.

emptyQueue Este es el comportamiento cuando la *cola* no tiene ningún nodo en ella.

waitDelete Un estado intermedio, se utiliza cuando se elimina un nodo de la *cola* y se esperan los valores de esta eliminación.

En el resto de la sección se detallaran cada uno de los cuatro comportamientos antes enumerados, para terminar con una descripción de cómo estos funcionan en conjunto.

Comportamiento de node

Este comportamiento es el que está encargado de guardar el contenido que se quiere guardar en la *cola*. Consta de dos operaciones *'delete'* e *'point_to'*. La primer operación envía todo su contenido a una dirección de buzón y termina su ejecución. El segundo, cambia el valor de *link*.

Código en *SAL*:

```
1 def Node(content, link) match
2   ['delete', mailbox]:
3     send [content, link] to mailbox
4   ['point_to', newLink]:
5     become Node(content, newLink);
6 end def
```

donde:

Líneas 2-3 Si la operación es $'delete'$, envía su contenido a *mailbox*. Como no hay comportamiento de reemplazo, este nodo termina su ejecución en este momento.

Líneas 4-5 Si la operación es $'point\_to'$, el comportamiento de reemplazo tiene un nuevo nodo al que apunta. Esta operación básicamente cambia el nodo que está siguiente en la *cola*.

Código en *CSP*:

$$
\begin{array}{l}
\textsf{process} \\
\quad Node(self, content, link) = \\
\qquad CommRecv.self?\langle 'delete', mailbox\rangle \rightarrow \\
\qquad CommSend.mailbox!(content, link) \rightarrow \\
\qquad STOP \\
\quad \Box \\
\qquad CommRecv.self?\langle 'point\_to', newLink\rangle \rightarrow \\
\qquad Node(self, content, newLink)
\end{array}
$$

donde:

$CommRecv.self?\langle 'delete', mailbox\rangle$ espera recibir un mensaje, donde el primer elemento de la lista sea la constante $'delete'$. Guarda en *mailbox* el valor del segundo elemento de la lista, que es una dirección de buzón.

$CommSend.mailbox!\langle content, link\rangle$ Envía una comunicación al buzón *mailbox*, con la lista *content* y *link*.

$STOP$ Al no haber comportamiento de reemplazo, el actor termina su ejecución.

$CommRecv.self?\langle 'point\_to', newLink\rangle$ Espera recibir un mensaje, donde el primer elemento de la lista sea la constante $'point\_to'$. Guarda en *mailbox* el valor del segundo elemento de la lista, que es una dirección de buzón.

$Node(self, content, newLink)$ Define el comportamiento de reemplazo, utiliza el parámetro *newLink* recibido en la línea como reemplazo del anterior *link*. Es decir reemplaza al nodo que apunta.

Comportamiento de emptyQueue

Este comportamiento corresponde a la *cola* cuando no tiene ningún nodo. La única operación posible es $'enqueue'$, que agrega un nodo. El comportamiento de reemplazo es *Queue*.

Código en *SAL*:

```
1 def EmptyQueue() match
2   ['enqueue', value]:
```

```
3       let P = new Node(value, nil) in
4         become Queue(P, P)
5   end def
```

donde:

Línea 3 Si la operación fue *enqueue*, entonces crea un nuevo nodo con el valor recibido. Como es el primer nodo que va a tener la cola, el parámetro del siguiente nodo en la pila es *nil*.

Línea 4 El comportamiento de reemplazo para este actor es *Queue*. Como es el único nodo que tiene la *cola* el primer y el último actor es el actor creado en la linea anterior.

Código en *CSP*:

```
process
  EmptyQueue(self) =
    CommRecv.self?⟨'enqueue', value⟩ →
    CreateAsk.node?pid!⟨value, Null⟩ →
    Queue(self, pid, pid)
```

$CommRecv.self?\langle'enqueue', value\rangle$ espera recibir un mensaje, donde el primer elemento de la lista sea la constante $'enqueue'$, guarda en *value* el valor del segundo elemento.

$CreateAsk.node?pid!\langle value, Null\rangle$ crea un nuevo actor de tipo *node*, guarda en en *pid* el buzón. Inicializa los valores *acquaiantence-list* con el entero *value* y *Null*.

$Queue(self, pid, pid)$ define a *Queue* como el comportamiento de reemplazo, para el buzón *self*. Le pasa como parámetro la dirección del buzón del actor antes creado, se repite porque tanto el primer como el último nodo es el mismo cuando la *cola* tiene un solo nodo.

Comportamiento de queue

Este comportamiento es cuando la cola tiene al menos un nodo. Tiene dos operaciones $'enqueue'$ y $'dequeue'$. La primera agrega un nodo y la segunda lo remueve. Al remover un nodo, es necesario obtener la referencia al nuevo primer nodo, es decir, al que apuntaba el nodo que esta por ser removido. Para esto se utiliza el comportamiento *waitDelete*.

Código en *SAL*:

```
1   def Queue(first, last) match
2     ['enqueue', value]:
3       let newLast = new node(value, nil) in
```

```
4         send [insert, newLast] to last
5         become queue(first, newLast)
6     ['dequeue', client]:
7       send [delete, self] to first
8       become waitDelete(last, client)
9   end def
```

donde:

Línea 1 *first* y *last* son respectivamente, el primer y último nodo de la *cola*.

Línea 3 Si la operación fue *enqueue*, entonces crea un nuevo nodo con el valor recibido. Como este nodo será el último, no apunta a nadie.

Línea 4 Le envía un mensaje a *last* para que intercambie el valor al que apunta por el valor del nodo recién creado.

Línea 5 El comportamiento de reemplazo es el mismo, lo único que cambia es el valor del buzón del último nodo (*last*) por el nodo recién creado.

Línea 7 Borra el primer nodo en la lista.

Línea 8 El nuevo comportamiento es un estado intermedio llamado *waitDelete*.

Código en *CSP*:

```
process
  Queue(self, first, last) =
    CommRecv.self?⟨'enqueue', value⟩ →
    CreateAsk.node?newLast!⟨value, Null⟩ →
    CommSend.last!⟨'insert', newLast⟩ →
    Queue(self, first, newLast)
  □
    CommRecv.self?⟨'dequeue', client⟩ →
    CommSend.first!⟨'delete', self⟩ →
    WaitDelete(self, client, last)
```

donde:

CommRecv.self?⟨*'enqueue'*, *value*⟩ espera recibir un mensaje, donde el primer elemento de la lista sea la constante *'enqueue'*. Guarda en *value* el valor del segundo elemento de la lista.

CreateAsk?*node.pid*!⟨*value*, *Null*⟩ crea un nuevo actor de tipo *node*, guarda en en *newLast* el buzón. Inicializa los valores *acquaiantence-list* con el valor *value* y *Null*.

CommSend.first!⟨*'insert'*, *newLast*⟩ envía una comunicación al buzón *first*, con la lista *'insert'* y el actor creado en la línea anterior.

Queue(*self*, *first*, *newLast*) define a *Queue* como el comportamiento de reemplazo, para el buzón *self*. Pasa como parámetro la dirección del buzón del actor antes creado como reemplazo del último nodo de la pila.

CommRecv.self?⟨*'dequeue'*, *client*⟩ espera recibir un mensaje, donde el primer elemento de la lista sea la constante *'dequeue'* y el segundo es el buzón a quien enviarle el valor que se está desencolando.

CommSend.first!⟨*'delete'*, *self*⟩ envía una comunicación al buzón *first*, con la lista *'delete'* y la dirección del buzón de la actor *cola*.

WaitDelete(*self*, *client*, *last*) el comportamiento de reemplazo es *WaitDelete*. Este comportamiento será descripto a continuación.

Comportamiento de waitDelete

Este comportamiento es un estado intermedio. Está a la espera de los datos del nodo que está siendo removido. Una vez que llegan estos datos, se envían a quien originalmente pidió remover el nodo. Si el nodo que se borró era el último, el nodo al que apuntaba será *nil*, entonces se tiene que comportar como si la cola estuviera vacía. En caso contrario se comporta como la cola con al menos un elemento.

Código en *SAL*:

```
1 def waitDelete(last, client)[content, newFirst]
2   send [content] to client
3   if (newFirst = nil) then
4     become emptyQueue()
5   else
6     become queue(newFirst, last)
7 end def
```

donde:

Línea 2 Reenvía a *client* el valor *content* recibido

Línea 4 Si el valor del primer nodo es nulo, el comportamiento de reemplazo es *emptyQueue*

Línea 6 El comportamiento de reemplazo es *queue*. Utiliza como primer nodo el nodo recibido en la comunicación.

Código en *CSP*:

```
process
```

$$
\begin{array}{l}
WaitDelete(self, client, last) = \\
\quad CommRecv.self?\langle newFirst, content\rangle \rightarrow \\
\quad CommSend.client?\langle content\rangle \rightarrow \\
\quad\quad EmptyQueue(self)\lceil\ newFirst == Null\ \rceil Queue(self, newFirst, last)
\end{array}
$$

donde:

$CommRecv.self?\langle newFirst, content\rangle$ espera recibir un mensaje, donde el primer elemento es una dirección de buzón. Lo guarda en *newFirst*. El segundo elemento lo guarda en *content*

$CommSend.client!\langle content\rangle$ envía una comunicación al buzón *client*, la lista con el valor *content*.

$EmptyQueue(self)$ si el valor recibido como el nuevo primer nodo de la cola es *Null*, el comportamiento de reemplazo es *EmptyQueue*. La lista vuelve al estado vacío.

$Queue(self, newFirst, last)$ el comportamiento de reemplazo es *Queue*. Cambia el valor del primer nodo por *newFirst*.

La *cola* tiene dos transiciones, cuando está vacía y se agrega un nodo cambia del comportamiento *EmptyQueue* al comportamiento *Queue*. Cuando tiene un único nodo y se lo remueve, cambia del comportamiento *Queue* a *EmptyQueue*. También tiene un funcionamiento habitual, es decir cuando se agregan y quitan nodos y hay más de un nodo en la *cola*.

Supongamos que se quiere insertar un nodo ($'enqueue'$), con el valor 42 en una *cola* con ningún nodo en ella. La interacción entre los actores tendría la siguiente forma:

- El buzón del actor que tiene el comportamiento *EmptyQueue*:

  Recibe una comunicación con la lista $'enqueue'$ y 42.

  Crea un nuevo nodo y guarda la dirección del buzón en P.

  Como es el primer nodo en la *cola*, el siguiente nodo es el nodo vacío.

  El comportamiento de reemplazo es *Queue*, como es el único nodo en la lista, el primer y el último nodo coinciden y es el nodo recién creado.

Supongamos que queremos insertar un nodo ($'enqueue'$), con el valor 42 en una *cola* con al menos un nodo en ella, la interacción entre los actores tendría la siguiente forma:

- El buzón del actor que tiene el comportamiento *Queue*:

  Recibe una comunicación con la lista $'enqueue'$ y 42.

  Crea un nuevo nodo, y guarda la dirección del buzón en *newLast*.

Reenvía un mensaje al último nodo en la *cola* (*last*) para que cambie al nodo que apunta.

Cambia cual es el último nodo en *Queue*, ahora es *newLast*.

- El buzón del nodo que hasta ese momento era el último (*last*):

  Recibe una comunicación con la lista $'insert'$ y *mailbox*.

  En el comportamiento de reemplazo, cambia el valor de *link* por *mailbox*. Esto hace que apunte a un nuevo nodo, el que fue insertado.

Supongamos que queremos remover el primer nodo ($'dequeue'$), y enviarle el contenido a un actor *client*. La interacción entre los actores tendría la siguiente forma:

- El buzón del actor que tiene el comportamiento *Queue* recibe una comunicación con la lista $'dequeue'$ y una dirección de buzón (*client*) para enviar el contenido del primer nodo.

  Le envía al actor que está primero en la cola (*first*) un mensaje para que se borre, con una dirección de buzón a quien enviarle su contenido.

  Cambia en un comportamiento intermedio, el cual esperará el contenido del nodo que será borrado.

- El buzón del nodo que hasta ese momento era el primero (*first*):

  Recibe una comunicación con la lista *delete* y *mailbox*.

  Envía su contenido, tanto *link* que es a la dirección de buzón al que apunta (el siguiente nodo en la *cola*), como el contenido que guarda al buzón *mailbox*. Como no tiene comportamiento de reemplazo, este actor termina su ejecución en este momento.

- El buzón del actor que tiene el comportamiento *WaitDelete* recibe una comunicación con la lista *content* y una dirección de buzón. Esta dirección ea a la que el primer nodo apunta, es decir el siguiente nodo.

  Se envía el contenido a quien originalmente lo pidió (*client*).

  Si la dirección a la que apuntaba era *Null*, esto quiere decir que era el último nodo en la lista. Se comporta como *emptyQueue*.

  En caso contrario, cambia al comportamiento *Queue*, con un nuevo nodo como el primer nodo en la *cola*.

4.2.4. Ejemplo: un anillo de actores

Este ejemplo está propuesto como ejercicio en el libro *Programming Erlang* [4, Página 115, Ejercicio 4-2: The Process Ring]. El ejercicio propone crear una red de n procesos como muestra la figura 4.1. Una vez terminada de establecerse, el primero de estos

nodos envía un mensaje al siguiente nodo en su red. Una vez enviados m mensajes dentro del anillo, los nodos deberían terminar su ejecución.

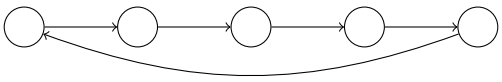

Figura 4.1: Anillo de procesos con $n = 5$

Para resolver este problema, se plantean tres comportamientos:

Node modela cada nodo del anillo.

Ring espera un mensaje con una cantidad de nodos a crear, y una cantidad de mensajes a enviar.

BuildingRing cumple la función de estructura de control, crea el anillo de nodos.

Comportamiento de Node

Node reacciona ante dos comunicaciones diferentes: $'point\_to'$ y $'msg'$. En el caso de $'point\_to'$, cuando lo procesa, cambia al nodo que apunta. Cuando procesa un mensaje de tipo $'msg'$, siempre reenvía al siguiente nodo en la lista la comunicación $'msg'$. Si el valor del contador m es mayor que cero, se disminuye el contador en uno y sigue con la ejecución. Si es cero, termina la ejecución en ese momento.

Código en *SAL* para **Node**:

```
1  def Node(m, next) match
2    case ['point_to', newNext]:
3      become Node(m, newNext)
4    case ['msg']:
5      if ( m = 0 ) then
6        send ['msg'] to next
7      else
8        send ['msg'] to next
9        become Node(m - 1, next)
10     end if
11 end def
```

Código en *CSP*:

$$
\begin{array}{l}
\textsf{process} \\
\quad Node(self, m, next) = \\
\qquad CommRecv.self?\langle 'point\_to', newNext\rangle \rightarrow \\
\qquad Node(self, m, newNext) \\
\quad \square
\end{array}
$$

$$
\begin{array}{l}
\quad CommRecv.self?\langle 'msg' \rangle \rightarrow \\
\qquad (CommSend!next.\langle 'msg' \rangle \rightarrow \\
\qquad STOP) \\
\quad \lceil\ m == 0\ \rceil \\
\qquad (CommSend!next.\langle 'msg' \rangle \rightarrow \\
\qquad Node(self, m-1, next))
\end{array}
$$

Comportamiento de Ring

Cuando **Ring** recibe una comunicación con una lista que tiene un par de enteros, el primero de los enteros es la cantidad de nodos a crear, y el segundo es la cantidad de comunicaciones a mandar. Crea el primer nodo que va a tener el anillo y el actor que va a estar encargado de crear el resto del anillo. Le envía un mensaje a *builder* con el número de nodos que tiene que crear. Como el primero fue inicialiado este es decrementado en uno.

Código en *SAL*:

```
1 def Ring()[n, m]
2   let first = new Node(m, nil)
3       builder = new BuildingRing(first, first)
4   in
5     send [n - 1, m] to builder
6     become Ring()
7 end def
```

Código en *CSP*:

$$
\begin{array}{l}
\quad \text{process} \\
\quad\ Ring(self) = \\
\qquad CommRecv.self?\langle n, m \rangle \rightarrow \\
\qquad CreateAsk.node?first!\langle m, Null \rangle \rightarrow \\
\qquad CreateAsk.buildingRing?builder!\langle first, first \rangle \rightarrow \\
\qquad CommSend.builder!\langle n-1, m \rangle \rightarrow \\
\qquad Ring(self)
\end{array}
$$

Comportamiento de BuildingRing

Esta es una estructura de control que siempre cuando procesa una comunicación con n mayor a uno, crea un nuevo nodo. Le envía al nodo que creó en la iteración anterior un mensaje para que apunte al nuevo nodo creado. Se auto envía un mensaje decrementando en uno el contador de nodos a crear. El comportamiento de reemplazo cambia el nodo *lastCreated*.

Si el contador n era cero, le envía un mensaje al nodo creado en la iteración anterior para que apunte al primer nodo creado (*first*). De esta manera cierra el anillo. Le envía una comunicación al primer nodo, que da inicio al envío de mensajes.

Código en *SAL*:

```
1  def BuildingRing(m, first, lastCreated)[n, m]
2    if (n == 0) then
3      send ['poins_to', first] to lastCreated,
4      send ['msg'] to first
5    else
6      let newNode = new Node(m, null) in
7        send ['point_to', newNode] to lastCreated
8        send [ n - 1, m ] to self
9        become BuildingRing(first, newNode)
10   end if
11 end def
```

Código en *CSP*:

process
$BuildingRing(self, first, lastCreated) =$
$CommRecv.self?\langle n, m\rangle \rightarrow$
if $(n == 0)$ then
$CommSend!lastCreated.\langle'point\_to', first\rangle \rightarrow$
$CommSend!first.\langle'msg'\rangle \rightarrow$
$STOP$
else
$CreateAsk.node?newNode!\langle m, Null\rangle \rightarrow$
$CommSend!lastCreated.\langle'point\_to', newNode\rangle \rightarrow$
$CommSend!self.\langle n-1, m\rangle \rightarrow$
$BuildingRing(self, first, newNode)$

4.3. La semántica en CSP

En esta sección se describe cómo traducir las expresiones, los comandos y los comportamientos desde *SAL* a *CSP*. Para esto se utiliza como punto de partida la gramática definida en la sección 2.2. Se utilizará como mecanismo de traducción unas funciones que estan definidas de manera inductiva.

Se utiliza en la descripción de las funciones de traducción la notación $x \equiv H$ cuando fuese necesario especificar que x tiene la forma gramatical H.

Expresiones

En la sección 2.2.1 se definió la gramática para las expresiones. Para traducir estas expresiones se utiliza la función exp_{tr}, definida de la siguiente forma:

$$exp_{tr}[\![exp]\!] = \begin{cases} iexp_{tr}[\![exp]\!] & \text{si } exp \equiv iexp \\ bexp_{tr}[\![exp]\!] & \text{si } exp \equiv bexp \\ sexp_{tr}[\![exp]\!] & \text{si } exp \equiv sexp \\ idexp_{tr}[\![exp]\!] & \text{si } exp \equiv id \end{cases}$$

La función para las expresiones de enteros, $iexp_{tr}$, viene dada de la siguiente forma:

$$\begin{array}{lcl} iexp_{tr}[\![iexp_1 + iexp_2]\!] & = & iexp_{tr}[\![iexp_1]\!] + iexp_{tr}[\![iexp_2]\!] \\ iexp_{tr}[\![iexp_1 - iexp_2]\!] & = & iexp_{tr}[\![iexp_1]\!] - iexp_{tr}[\![iexp_2]\!] \\ iexp_{tr}[\![iexp_1 * iexp_2]\!] & = & iexp_{tr}[\![iexp_1]\!] * iexp_{tr}[\![iexp_2]\!] \\ iexp_{tr}[\![iexp_1 / iexp_2]\!] & = & iexp_{tr}[\![iexp_1]\!] / iexp_{tr}[\![iexp_2]\!] \\ iexp_{tr}[\![- iexp]\!] & = & -iexp_{tr}[\![iexp]\!] \\ iexp_{tr}[\![id]\!] & = & id \end{array}$$

La función para traducir las expresiones booleanas, $bexp_{tr}$, es finalmente:

$$\begin{array}{lcl} bexp_{tr}[\![bexp_1 \textbf{ or } bexp_2]\!] & = & bexp_{tr}[\![iexp_1]\!] \vee bexp_{tr}[\![iexp_2]\!] \\ bexp_{tr}[\![bexp_1 \textbf{ and } bexp_2]\!] & = & bexp_{tr}[\![iexp_1]\!] \wedge bexp_{tr}[\![iexp_2]\!] \\ bexp_{tr}[\![\textbf{not } bexp]\!] & = & \neg bexp_{tr}[\![bexp]\!] \\ bexp_{tr}[\![exp_1 == iexp_2]\!] & = & exp_{tr}[\![exp_1]\!] == exp_{tr}[\![exp_2]\!] \\ bexp_{tr}[\![id]\!] & = & id \end{array}$$

Tanto $sexp_{tr}$ como $idexp_{tr}$ no tienen una traducción asociada, podrían definirse como la función identidad.

Comportamientos

En la sección 2.2.2 se definió la gramática para los comportamientos. Para traducir estas expresiones se utilizan las funciones $beha_{tr}$ y $body_{tr}$.

La función $beha_{tr}$ viene dada por la forma:

$$beha_{tr}[\![\textbf{ def } BehName(p_1, p_2, \ldots, p_n)\ body \textbf{ end def }]\!] = \\ BehName(self, p_1, p_2, \ldots, p_n) = body_{tr}[\![body]\!]$$

La función $body_{tr}$ viene dada por la forma:

$$body_{tr}[\![\ [p_1, p_2, \ldots, p_n]\ command\ ]\!] = \\ CommRecv.self?\langle p_1, p_2, \ldots, p_n \rangle \rightarrow cmd_{tr}[\![command]\!]$$

$$body_{tr}[\![\ \textbf{case}\ [p_{11}, p_{12}, \ldots, p_{1n}] : command_1 \ldots\ \textbf{case}\ [p_{n1}, p_{n2}, \ldots, p_{nk}] : command_n\ ]\!] =$$
$$CommRecv.self?\langle p_{11}, p_{12}, \ldots, p_{1n}\rangle \rightarrow cmd_{tr}[\![command_1]\!] \,\Box \ldots \Box$$
$$CommRecv.self?\langle p_{n1}, p_{n2}, \ldots, p_{nk}\rangle \rightarrow cmd_{tr}[\![command_n]\!]$$

Comandos

En la sección 2.2.3 se definió la gramática para los comandos. Para traducir los comandos se utiliza la función cmd_{tr}, que se define de la siguiente forma:

$$cmd_{tr}[\![\ \textbf{send}\ exp_1, exp_2, \ldots\ , exp_n \textbf{to}\ mexp\ ]\!] =$$
$$CommSend.mexp.\langle exp_{tr}[\![exp_1]\!], exp_{tr}[\![exp_2]\!], \ldots, exp_{tr}[\![exp_m]\!]\rangle$$

$$cmd_{tr}[\![\ \textbf{become}\ B(exp_1, exp_2,\ \ldots, exp_n)\ ]\!] =$$
$$B(exp_{tr}[\![exp_1]\!], exp_{tr}[\![exp_2]\!], \ldots, exp_{tr}[\![exp_n]\!]\rangle$$

$$cmd_{tr}[\![command_1 \textbf{;} command_2]\!] = cmd_{tr}[\![command_1]\!] \rightarrow cmd_{tr}[\![command_2]\!]$$

$$cmd_{tr}[\![\ \textbf{if}\ bexp\ \textbf{then}\ command_1\ \textbf{else}\ command_2\ \textbf{end if}\ ]\!] =$$
$$cmd_{tr}[\![command_1]\!]\ \lceil bexp_{tr}[\![bexp]\!]\rceil\ cmd_{tr}[\![command_2]\!]$$

$$cmd_{tr}[\![\ \textbf{let}\ id_1 = \textbf{new} B_1(exp_{11}, exp_{12},\ \ldots, exp_{1m}), \ldots,$$
$$id_n = \ \textbf{new}\ B_n(exp_{n1}, exp_{n2}, \ldots, exp_{nk})\ \textbf{in}\ command\ ]\!] =$$
$$CreateAsk.b_1?id_1!\langle exp_{tr}[\![exp_{11}]\!], exp_{tr}[\![exp_{12}]\!],\ \ldots, exp_{tr}[\![exp_{1m}]\!]\rangle \rightarrow \ldots \rightarrow$$
$$CreateAsk.b_N?id_n!\langle exp_{tr}[\![exp_{n1}]\!], exp_{tr}[\![exp_{n2}]\!],\ \ldots, exp_{tr}[\![exp_{nk}]\!]\rangle \rightarrow$$
$$cmd_{tr}[\![command]\!]$$

Pasos finales

El sistema final consiste de todos los componentes del sistema de actores en paralelo. Los componentes a poner en paralelo son: los buzones vistos en la sección 4.1.1, los procesos que separan la intención de crear de la creación propiamente dicha vistos en la sección 4.1.2 y los preámbulos vistos en la misma sección.

La comunicación que va desde estos preámbulos hacia los buzones es una de las comunicaciones de interés. Los eventos en este caso son todos los generados por los buzones. Es decir el conjunto generado por $CommSend.i.j$ donde i son todos los posibles actores y j son los mensajes a enviar. A estos eventos se le debe unir el conjunto generado por $CommRecv.i.j$ donde i son todos los posibles actores y j son los men-

sajes a enviar. Este es el conjunto de todos los eventos posibles de los buzones. Este conjunto se denomina: *Comm*.

Otra forma de comunicación a tener en cuenta es desde estos preámbulos hacia los procesos que separan la intención de crear de la creación. Es decir el conjunto generado por *CreateAsk.i.j* donde i son todos los posibles actores y j son todos los posibles valores de *acquaiantence-list*. A estos eventos se le debe unir el conjunto generado por *Create.i.j* donde i son todos los posibles actores y j son todos los posibles valores de *acquaiantence-list*. Este conjunto se denomina *Init*.

Ahora se supone que tenemos dos redes concurrentes de actores *ks* y *js*, definidos de la siguiente manera:

$$ks = |||_{i:\{0..N_k\}} Create.k?self? < p1, p2 > \rightarrow Comportamiento_K(self, p1, p2)$$
$$js = |||_{i:\{0..N_j\}} Create.j?self? < p1, p2 > \rightarrow Comportamiento_J(self, p1, p2)$$

donde $Comportamiento_K$ y $Comportamiento_J$ son dos definiciones de comportamiento. Para integrar todo el sistema de actores se escribe la siguiente ecuación:

$$SYSTEM = (ks \,|||\, js) \underset{Comm \cup Init}{\|} (Mailboxes \,|||\, Creates)$$

donde *Mailboxes* está definido en la ecuación 4.1 y *Creates* en la ecuación 4.6 . Entre *ks* y *js* nunca existe ningún tipo de comunicación, por eso se usa el operador de entrelazado. Lo mismo ocurre con *Mailboxes* y *Creates*.

4.4. Corriendo los modelos en FDR

FDR es un verificador de refinamiento para *CSP*. Permite definir procesos de *CSP* utilizando el lenguaje *CSPm*. Se pueden verificar varias afirmaciones sobre estos procesos. A continuación se cuentan algunas de estas características.

La herramienta cuenta de la palabra reservada *probe*, está sirve para explorar manualmente las transiciones de un proceso como si fuera un árbol, es muy útil para al depurar una definición de proceso. Esto fue particularmente útil al explorar la creación del modelo antes propuesto. Se mostrará un caso donde se puede ver el gráfico de todas las ejecuciones de uno de los ejemplos.

También es posible verificar refinamiento utilizando el modelo de trazas o el modelo de fallas y divergencias. Se verá un ejemplo de cómo es posible verificar este tipo de afirmaciones.

Tanto *probe* como el chequeo de refinamiento son algunas de las posibles pruebas que se pueden hacer utilizando la herramienta. Este conjunto de pruebas no pretende comprender un uso exhaustivo de la herramienta, para mayor información sobre la herramienta se puede consultar su manual [15].

4.4.1. Restricciones sobre el modelo

Una de las ventajas de representar el modelo de actores en *CSP* es que se pueden utilizar herramientas de verificación de especificaciones *CSP* como *FDR*.

Es necesario imponer restricciones al utilizar *FDR*. Estas restricciones están relacionas a cómo la herramienta explora los estados posibles del modelo. Por ejemplo, si se utiliza el rango completo de enteros intentará revisar cada uno de estos enteros, haciendo cada exploración exponencial en cada estado que se visite. En el resto de esta sección se exploran estas restricciones. Las modificaciones que incluyen estas restricciones pueden verse en los ejemplos en el apéndice A.

Enteros pequeños

Para evitar la explosión de estados que causa utilizar el rango de enteros de *64-bits*, se genera una representación propia para reducirla. Para esto se utiliza un tipo algebraico que representa estos enteros:

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

donde, MAX_INT es el entero más grande a representar.

Aparte de esta representación de los enteros, se construyeron las operaciones básicas sobre ellos:

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

donde *add* es la suma, *sub* la resta, *mult* la multiplicación y *eq* la igualdad. Si alguna operación excede el entero máximo el resultado de esta es *Overflow*.

Listas de parámetros y valores

En en modelo se utilizan listas para representar tanto los parámetros de *acquaiantence-list* y los de *communication-list*. Dado que las listas son potencialmente infinitas, para

evitar que la herramienta entre en un ciclo infinito, se limita su tamaño. Con este fin se utilizan tuplas de tamaño fijo. Para esto es necesario conocer el tamaño máximo del mensaje que se va a enviar; esta es una limitación con respecto al modelo presentado.

Para poder tener cierta flexibilidad en el momento de enviar mensajes, se crea una unión de tipos llamada $VALUE$. Este tipo codifica todos los posibles valores que son posible utilizar en el modelo: buzones, enteros, booleanos y cadenas.

$$\begin{aligned} datatype\ VALUE = {} & ACTOR.ActorID \mid \\ & INT.SmallInt \mid \\ & BOOL.Bool \mid \\ & ATOM.Atoms \mid \\ & None \end{aligned}$$

Como las tuplas se definen de tamaño fijo, es necesario agregar elemento que permite marcar que un elemento de la lista no se usa. Por ejemplo si las listas son de tamaño tres, no necesariamente todos los mensajes que intercambien los actores serán de esta longitud. Por caso podría existir un mensaje que sea una lista de tamaño dos ($[1, client]$). La tupla de tamaño tres sería $(1, None, None)$.

Ya qué las cadenas son inmutables, estas son representadas utilizando un tipo enumeración, donde cada elemento de este nuevo tipo de datos es una cadena. La definición viene dada de la siguiente forma:

$$datatype\ Atoms = CADENA_1 \mid CADENA_2 \mid \ldots \mid CADENA_n$$

donde $Atoms$ representa el conjunto de cadenas en uso.

Para simplificar como se especifica la cantidad de actores que se necesitan se extiende la definición (4.3). Al conjunto de nombres de actores se agrega el conjunto de enteros que representa la cantidad, por ejemplo:

$$datatypeActorID = j.1, 2 \mid k.1, 2, 3 \mid main.1$$

donde $ActorID$ equivale al conjunto $\{j.1, j.2, k.1, k.2, k.3, main\}$. Este conjunto esta, al mismo tiempo, nombrando el conjunto de actores y definiendo la cantidad máxima de actores que existen.

Cota en el buzón

Se puede modelar un buzón que no tenga una cota superior, pero se tendría nuevamente problemas de explosión de estados ya que FDR intentaría explorar todas las combinaciones posibles de buzón. Este caso es similar al de la lista en la comunicación.

La ecuación de buzón, con cota, es la siguiente:

process

$MAILBOX\_SIZE = 4$

$Mailbox(i, \langle\rangle) =$

$\quad CommSend?i.x \rightarrow Mailbox(i, \langle x \rangle)$

$Mailbox(i, msgs) =$

$\quad\quad MailboxWithSpace(i, msgs)$

$\quad \lceil\ length(msgs) < MAILBOX\_SIZE - 1\ \rceil$

$\quad\quad MailboxFull(i, msgs)$

$MailboxWithSpace(i, \langle x \rangle \frown xs) =$

$\quad CommRecv!i.x \rightarrow Mailbox(i, xs)$

$\quad \square$

$\quad CommSend?i.y \rightarrow Mailbox(i, \langle x \rangle \frown xs \frown \langle y \rangle)$

$MailboxFull(i, \langle x \rangle \frown xs) =$

$\quad CommRecv!i.x \rightarrow Mailbox(i, xs)$

La ecuación anterior le agrega a la ecuación (4.1), un límite definido por *MAILBOX_SIZE*. El comportamiento del proceso buzón depende de su estado de la siguiente forma:

- Si no tiene ningún mensaje, solo sincroniza mensajes por el canal *CommSend*.
- Si tiene algunos mensajes, por los canales *CommSend* y *CommRecv*.
- Si llegó a su capacidad máxima *MAILBOX_SIZE* lo hace solo por el canal *CommRecv*.

4.4.2. Algunos resultados

A continuación se muestran dos resultados obtenidos utilizando el modelo propuesto en *CSPm*. Uno explora el árbol de procesos y el otro verifica una afirmación utilizando el modelo de trazas.

Árbol de interacciones

Utilizando la función *probe* de la herramienta *FDR*, permite ver árbol de interacciones de los distintos procesos. En el caso de este modelo, permite ver las diferentes interacciones de los actores.

En el proceso de desacoplar los componentes vistos en las secciones 4.1.1 y 4.1.2, fue necesario analizar la interacción que tienen los diferentes procesos para entender que efectivamente funcionaba de la forma esperada. Se puede ver en el ejemplo del Apéndice A.1, una versión extendida para funcionar en *FDR* del ejemplo de la ecuación 3.1. Este ejemplo muestra como es posible desacoplar los componentes utilizando dos eventos,

uno para la intensión de crear y otro para iniciar el actor, se puede desacoplar el inicio del proceso suma, y mejorar sustancialmente la composición en paralelo.

Esto es muy útil, ya que cada proceso que intente iniciar otro debería tener como evento de interés este evento de inicio. Al ir agregando procesos, el cálculo de qué eventos de interés deberían utilizarse al ponerse en paralelo se dificulta.

En el ejemplo, `SYSTEM2`, utiliza un proceso intermedio como fue visto en la sección 4.1.2. Se puede ver en la Figura 4.2 el árbol de procesos capturado utilizando `:probe SYSTEM1`. Este sistema pone en paralelo `CLIENTE` y `SUMA` usando paralelismo alfabetizado. Se puede ver en la Figura 4.3 el árbol de procesos capturado utilizando `:probe SYSTEM2`. Este sistema pone en paralelo `CLIENTE2` y `SUMA` usando entrelazado, y un proceso auxiliar `INICIA`. Este ultimo proceso reduce a cero los eventos de interés entre `CLIENTE2` y `SUMA`.

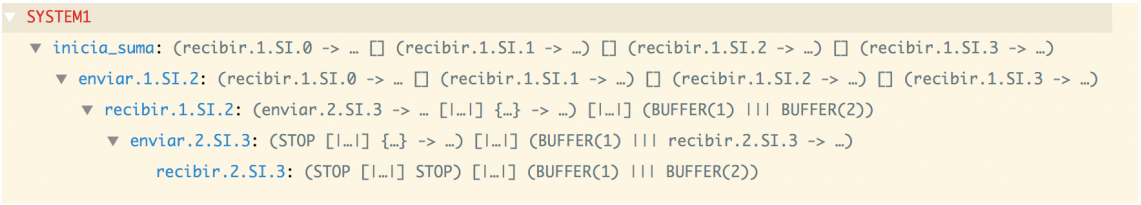

Figura 4.2: Árbol de interacciones de SYSTEM1 del apéndice A.1

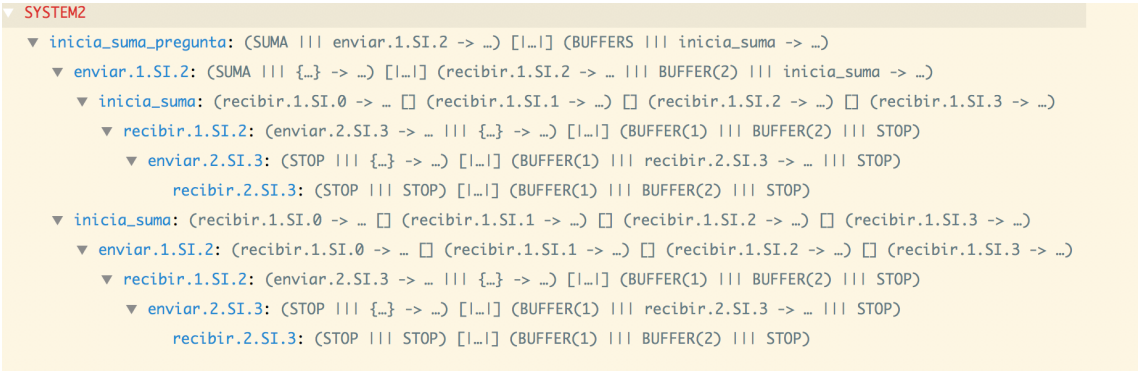

Figura 4.3: Árbol de interacciones de SYSTEM2 del apéndice A.1

Un ejemplo mas complejo puede verse en la Figura 4.4, fue utilizado para comprobar que el ejemplo estaba realmente calculando el factorial de 3. La captura fue realizada utilizando el comando `:probe SYSTEM` en *FDR* utilizando el código de factorial del apéndice A.2.

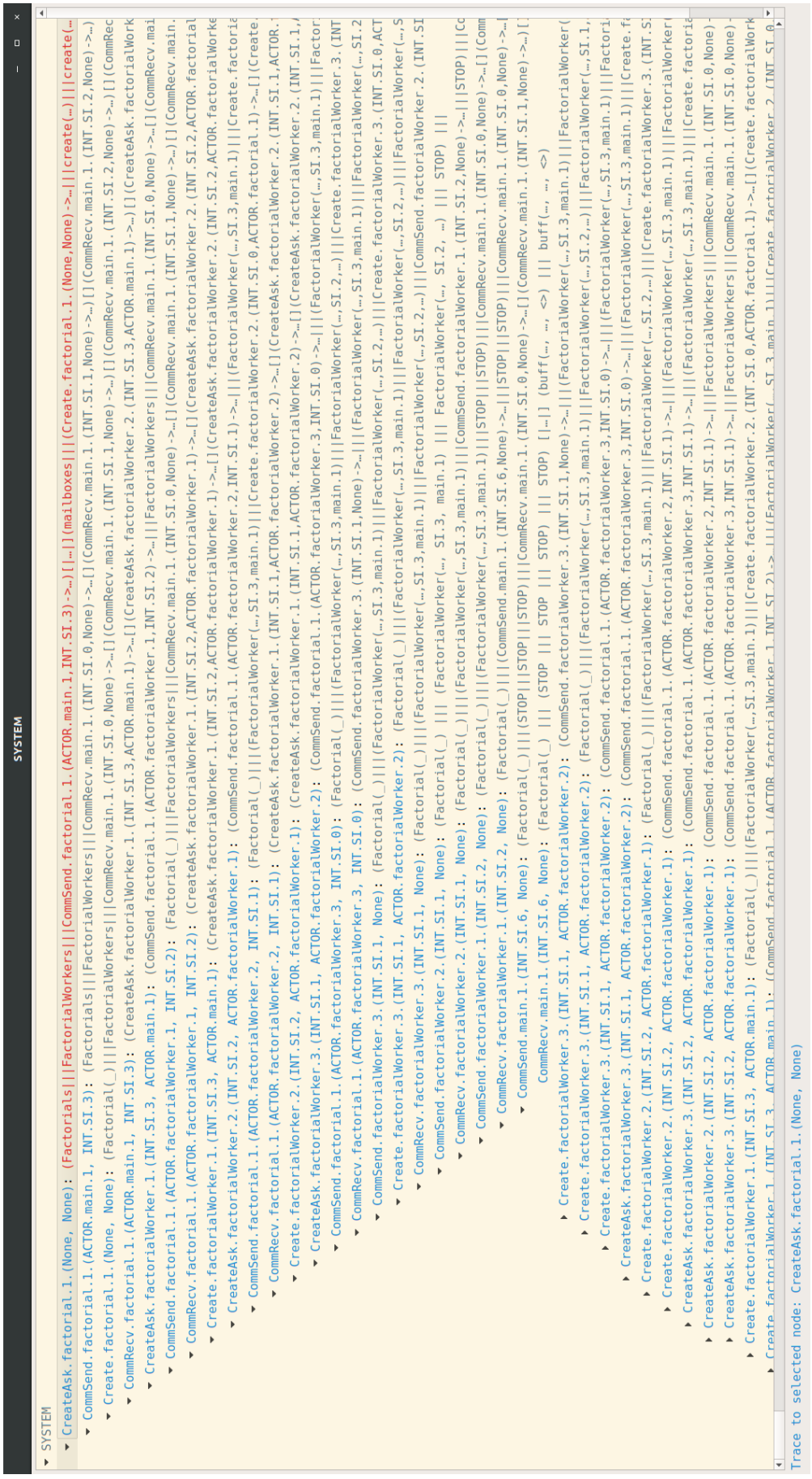

Figura 4.4: Calculo del factorial de 3 visto en la seccion 2.3

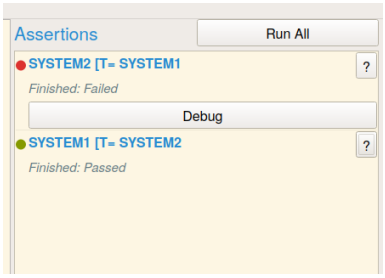

Figura 4.5: Resultado de las afirmaciones

Refinamiento

En *FDR* para probar si el conjunto de trazas de `P` es un refinamiento del conjunto de trazas `Q` se escribe `assert P [T= Q`. En caso de querer utilizar fallas y divergencias se debería notar `assert P [FD= Q`. Para hacer una prueba de refinamiento se utiliza el ejemplo de la cola visto en la sección 4.2.3. El código en *CSPm* de este se encuentra en el Apéndice A.3. El mismo cuenta con dos actores "main", `actor_main1` y `actor_main2`, donde: el actor `actor_main1`, después de crear un actor `Queue`, hace uso de la selección interna para:

- enviar al actor `Queue` un mensaje $['enqueue', 1]$
- enviar al actor `Queue` un mensaje $['enqueue', 2]$
- enviar al actor `Queue` un mensaje $['dequeue', buzonDeMain]$, seguido de esperar recibir un entero.

El actor `actor_main2`, después de crear un actor *Queue*, hace uso de la selección interna para:

- enviar al actor `Queue` un mensaje $['enqueue', 1]$, seguido de enviar al actor *Queue* un mensaje $['enqueue', 2]$
- enviar al actor `Queue` un mensaje $['dequeue', buzonDeMain]$, seguido de esperar recibir un entero.

El resultado de poner en paralelo todos los componentes del sistema utilizando `actor_main1` se llama `SYSTEM1`. En el caso de `actor_main2` se llama `SYSTEM2`. Esto puede verse en el código de *CSPm* en el Apéndice A.3.

Para probar si `SYSTEM1`, refina en `SYSTEM2` se utiliza la siguiente afirmación: `assert SYSTEM1 [T= SYSTEM2`. Para probar si `SYSTEM2`, refina en `SYSTEM1` se utiliza la siguiente afirmación: `assert SYSTEM2 [T= SYSTEM1`.

Puede verse en la Figura 4.5 el resultado de correr ambas afirmaciones. Utilizando la herramienta se encuentra un contraejemplo donde `SYSTEM2` no refina en `SYSTEM1` como muestra la Figura 4.6.

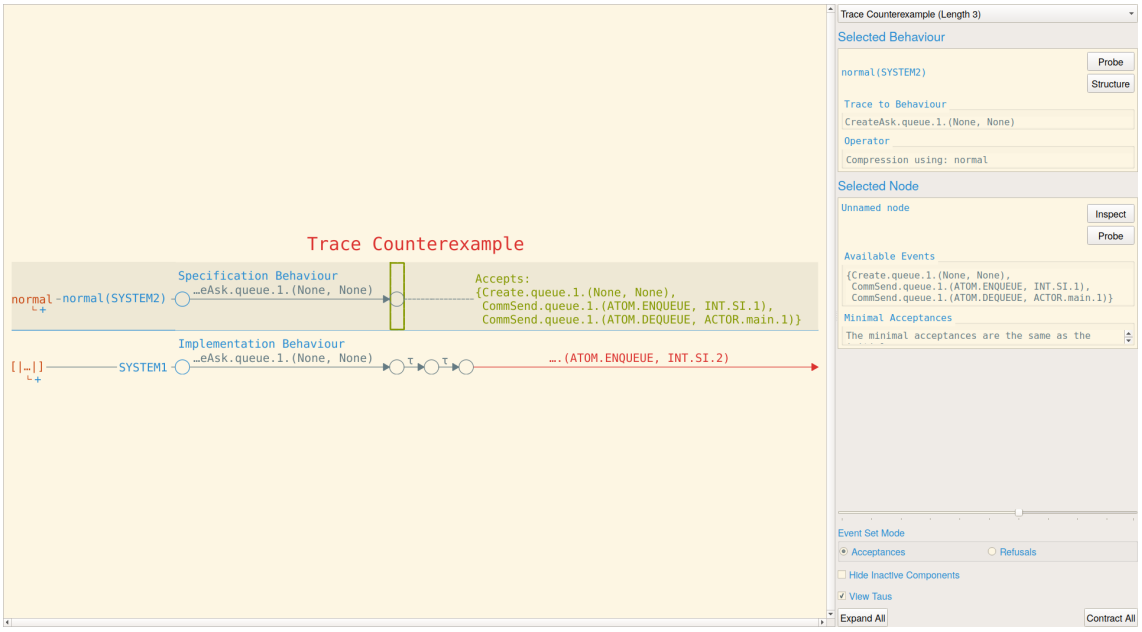

Figura 4.6: Contraejemplo de SYSTEM2 no refina en SYSTEM1

La intuición detrás de por qué `SYSTEM1` refina en `SYSTEM2`, y que `SYSTEM2` no refina en `SYSTEM1` tiene que ver con que `SYSTEM1` es menos restrictivo que `SYSTEM2`. El actor `actor_main1` puede elegir entre entre dos opciones para desencolar, y el actor `actor_main2` solo una y con un orden preestablecido.

Si bien la descripción anterior solo contempla el caso del chequeo de refiniamiento utilizando el modelo de trazas, en el Apéndice A.3 pueden verse ambas aserciones.

Capítulo 5

Conclusiones

La primera motivación de este trabajo consistió comprender los elementos básicos del modelo de actores. En segunda instancia fue generar un modelo en *CSP*, y hacer con este algunas pruebas en *FDR*. Fue interesante explorar los distintos mecanismos de paralelismo que tiene *CSP* a la hora de componer procesos.

El mayor esfuerzo involucrado tiene que ver con lograr que el modelo de actores corriera en *FDR*, de ahí vienen las restricciones enunciadas en el capítulo anterior. Sin dudar el aporte de *FDR* permitió entender que el modelo realmente funcionaba. Fue interesante poder observar el árbol de ejecución utilizando el comando `probe` de *FDR*.

El trabajo original de Agha [1], modelaba los mensajes como una 3-tupla. Además del actor destino y el mensaje este agregaba un *TAG* que después utilizaba en el modelo denotacional que construyó para prefijar la creación de las nuevas direcciones de buzón, esto servia para crear una dirección de buzón que no existía antes. Como crear una dirección no existente, no fue un problema simple de resolver en este modelo. Diferentes modelos fueron construidos y probados hasta llegar el propuesto en la sección 4.1.2. El momento de creación y el paso inicial de mensajes es uno de los puntos fuertes del modelo de actores.

Otro de los puntos más interesantes a resaltar del modelo, es la conclusión de que la fuerza que impulsa el modelo son los mensajes sin procesar. Este concepto es fundamental para cualquier trabajo relacionado con la exploración del grafo de entrelazado ya que dado que los actores son deterministas, recorrer este grafo está relacionado con el orden en el que se procesan los mensajes.

Trabajos futuros

Se podría utilizar el modelo propuesto para implementar con otros lenguajes que implementan el modelo de actores, como Erlang [**?**, 17], PonyLang [3] o la biblioteca akka [18].

El modelo de buzón que se utilizó tiene solo disponible para ser consumido el mensaje que está primero en el buzón. Se podría usar el presente modelo para analizar

las diferencias con el modelo en el cual todos los mensajes que están dentro del buzón están disponibles para ser consumidos.

Apéndice A

Codigo CSPm

A.1. Suma

```
MAX_INT = 3
datatype SmallInt = SI.{0 .. MAX_INT} | Overflow
--
add(SI.a, SI.b) =
  let sum = a + b within if sum <= MAX_INT then SI.sum else Overflow
add(_, _) = Overflow

channel inicia_suma, inicia_suma_pregunta
channel enviar, recibir: {1,2}.SmallInt

BUFFER(x) = enviar.x?msg -> recibir.x!msg -> BUFFER(x)

SUMA = inicia_suma -> recibir.1?x -> enviar.2!add(x, SI.1) -> STOP
CLIENTE = inicia_suma -> enviar.1!SI.2 -> recibir.2?x -> STOP
BUFFERS = BUFFER(1) ||| BUFFER(2)

SYSTEM1 = (SUMA [|{ inicia_suma }|] CLIENTE) [|{| enviar, recibir |}|] BUFFERS

INICIA = inicia_suma_pregunta -> inicia_suma -> STOP
CLIENTE2 = inicia_suma_pregunta -> enviar.1!SI.2 -> recibir.2?x -> STOP

COMUNICACIONES = {| enviar, recibir, inicia_suma, inicia_suma_pregunta |}

SYSTEM2 = (SUMA ||| CLIENTE2) [|COMUNICACIONES|] ( BUFFERS ||| INICIA )
```

A.2. Factorial

```
-- Small Int representation

MAX_INT = 6
datatype SmallInt = SI.{0 .. MAX_INT} | Overflow
--
add(SI.a, SI.b) =
  let sum = a + b within if sum <= MAX_INT then SI.sum else Overflow
add(_, _) = Overflow

sub(SI.a, SI.b) =
  let sub = a - b within if sub >= 0 then SI.sub else Overflow
sub(_, _) = Overflow

mult(SI.a, SI.b) =
  let mult = a * b within if mult <= MAX_INT then SI.mult else Overflow
mult(_, _) = Overflow

eq(SI.a, SI.b) = a == b
eq(_, _) = false

-- Possible actor names
datatype ActorID = factorial | factorialWorker | main
N_FACTORIAL = 1
N_FACTORIAL_WORKER = 3

MailboxIDS = { 0 .. N_FACTORIAL + N_FACTORIAL_WORKER }

-- Possible types for actors
datatype VALUE = ACTOR.MailboxIDS | INT.SmallInt | None

-- Actor creation decoupling
channel CreateAsk:ActorID.MailboxIDS.(VALUE, VALUE)
channel Create:ActorID.MailboxIDS.(VALUE, VALUE)

create(mailboxId) = CreateAsk?actorId!mailboxId?m -> Create.actorId!mailboxId!m -> STOP
creates = ||| mailboxId: MailboxIDS @ create(mailboxId)

-- Send ( actor to mailbox ) and Recv ( mailbox to actor ) communication
channel CommSend:MailboxIDS.(VALUE, VALUE)
channel CommRecv:MailboxIDS.(VALUE, VALUE)
```

```
MAILBOX_SIZE = 3

buff(left, right, <>) = left?msg -> buff(left, right, <msg>)

buff(left, right, xs) = if (length(xs) < MAILBOX_SIZE - 1) then
    buff_with_space(left, right, xs)
  else
    buff_full(left, right, xs)

buff_with_space(left, right, <x> ^ xs) =
  right!x -> buff(left, right, xs)
  []
  left?y -> buff(left, right, <x> ^ xs ^ <y>)

buff_full(left, right, <x> ^ xs) = right!x -> buff(left, right, xs)

mailboxes = ||| mailboxId: MailboxIDS @ buff(CommSend.mailboxId, CommRecv.mailboxId, <>)

-- Factorial actor
Factorials = Create.factorial?mailboxId?(None, None) -> Factorial(mailboxId)
Factorial(self) = CommRecv?self.(ACTOR.mailboxClient, INT.k) ->
if (eq(k,SI.0))
  then
    CommSend!mailboxClient.(INT.SI.1, None) -> Factorial(self)
  else
    let
      newK = sub(k, SI.1)
    within
      CreateAsk.factorialWorker?pid!(INT.k, ACTOR.mailboxClient) ->
      CommSend!self.(ACTOR.pid, INT.newK) ->
      Factorial(self)

-- FactorialWorker actor
FactorialWorkers =
  ||| i : { 1 .. N_FACTORIAL_WORKER } @ Create.factorialWorker?actorId?(INT.k, ACTOR.client)
  FactorialWorker(actorId, k, client)

FactorialWorker(self, k, client) = CommRecv.self?(INT.n, None) ->
    let
      val = mult(n, k)
    within
```

```
      CommSend.client!(INT.val, None) ->
      STOP

-- Main
Main =
  CreateAsk!factorial?pid!(None, None) ->
  CommSend!pid.(ACTOR.0, INT.SI.3) ->
  CommRecv?0.(INT.v, None) ->
  STOP
--
COMM = {|CommSend, CommRecv|}
INIT = {|Create, CreateAsk|}
--

SYSTEM =
  (Factorials ||| FactorialWorkers ||| Main)
    [|union(COMM,INIT)|]
  (mailboxes ||| creates)
```

A.3. Cola

```
-- Small Int representation
MAX_INT = 6
datatype SmallInt = SI.{0 .. MAX_INT} | Overflow

add(SI.a, SI.b) =
  let sum = a + b within if sum <= MAX_INT then SI.sum else Overflow
add(_, _) = Overflow

sub(SI.a, SI.b) =
  let sub = a - b within if sub >= 0 then SI.sub else Overflow
sub(_, _) = Overflow

mult(SI.a, SI.b) =
  let mult = a * b within if mult <= MAX_INT then SI.mult else Overflow
mult(_, _) = Overflow

eq(SI.a, SI.b) = a == b
eq(_, _) = false

-- Possible actor names
```

```
datatype ActorID = queue.{1} | node.{1,2,3} | main.{1} | NoId

-- Possible Strings
datatype Atoms = ENQUEUE | DEQUEUE | INSERT | DELETE

-- Possible actor values
datatype VALUE = ACTOR.ActorID | INT.SmallInt | ATOM.Atoms | None

-- Actor creation decoupling
channel CreateAsk:ActorID.(VALUE, VALUE)
channel Create:ActorID.(VALUE, VALUE)

create(actorId) = CreateAsk!actorId?m -> Create.actorId!m -> STOP
creates = ||| actor: ActorID @ create(actor)

-- Send ( actor to mailbox ) and Recv ( mailbox to actor ) communication
channel CommSend:ActorID.(VALUE, VALUE)
channel CommRecv:ActorID.(VALUE, VALUE)
MAILBOX_SIZE = 3

buff(left, right, <>) = left?msg -> buff(left, right, <msg>)

buff(left, right, xs) = if (length(xs) < MAILBOX_SIZE - 1) then
    buff_with_space(left, right, xs)
  else
    buff_full(left, right, xs)

buff_with_space(left, right, <x> ^ xs) =
  right!x -> buff(left, right, xs)
  []
  left?y -> buff(left, right, <x> ^ xs ^ <y>)

buff_full(left, right, <x> ^ xs) = right!x -> buff(left, right, xs)

mailboxes = ||| actor: ActorID @ buff(CommSend.actor, CommRecv.actor, <>)

-- become (communication forwarding)
fwd(in, out) = CommRecv.in?msg -> CommSend.out!msg -> fwd(in, out)

-- Node actor
Nodes = ||| actorId : {|node|} @ Create.actorId?(INT.content, ACTOR.link) ->
```

```
  Node(actorId, content, link)

Node(self, content, link) =
  CommRecv.self?(ATOM.DELETE, ACTOR.client) ->
  CommSend.client!(ACTOR.link, INT.content) ->
  STOP
  []
  CommRecv.self?(ATOM.INSERT, ACTOR.newLink) ->
  Node(self, content, newLink)

-- Queue Actor
Queues = ||| actorId : {|queue|} @ Create.actorId?(None, None) -> EmptyQueue(actorId)

EmptyQueue(self) =
  CommRecv.self?(ATOM.ENQUEUE, INT.value) ->
  CreateAsk?node.pid!(INT.value, ACTOR.NoId) ->
  Queue(self, node.pid, node.pid)

Queue(self, first, last) =
  CommRecv.self?(ATOM.ENQUEUE, INT.value) ->
  CreateAsk?node.newLast!(INT.value, ACTOR.NoId) ->
  CommSend.first!(ATOM.INSERT, ACTOR.node.newLast) ->
  Queue(self, first, node.newLast)
  []
  CommRecv.self?(ATOM.DEQUEUE, ACTOR.client) ->
  CommSend.first!(ATOM.DELETE, ACTOR.self) ->
  CommRecv.self?(ACTOR.newFirst, INT.value) ->
  CommSend.client!(INT.value, None) ->
  if (newFirst == NoId) then
    EmptyQueue(self)
  else
    Queue(self, newFirst, last)

actor_main1 = CreateAsk?queue.pid!(None, None) ->
              actor_main1_r(queue.pid)

actor_main1_r(pid) =
    CommSend.pid!(ATOM.ENQUEUE, INT.SI.1) -> actor_main1_r(pid) |~|
    CommSend.pid!(ATOM.ENQUEUE, INT.SI.2) -> actor_main1_r(pid) |~|
    CommSend.pid!(ATOM.DEQUEUE, ACTOR.main.1) ->
    CommRecv.main.1?(INT.V, None) ->
```

```
    actor_main1_r(pid)

actor_main2 = CreateAsk?queue.pid!(None, None) ->
              actor_main2_r(queue.pid)

actor_main2_r(pid) =
      CommSend.pid!(ATOM.ENQUEUE, INT.SI.1) ->
      CommSend.pid!(ATOM.ENQUEUE, INT.SI.2) ->
      actor_main2_r(pid) |~|
      CommSend.pid!(ATOM.DEQUEUE, ACTOR.main.1) ->
      CommRecv.main.1?(INT.V, None) ->
      actor_main2_r(pid)

--
COMM = {|CommSend, CommRecv|}
INIT = {|Create, CreateAsk|}
--

SYSTEM1 =
  (Queues ||| Nodes ||| actor_main1)
    [|union(COMM,INIT)|]
  ( mailboxes ||| creates )

SYSTEM2 =
  (Queues ||| Nodes ||| actor_main2)
    [|union(COMM,INIT)|]
  ( mailboxes ||| creates )

assert SYSTEM2 [T= SYSTEM1
assert SYSTEM1 [T= SYSTEM2

assert SYSTEM2 [FD= SYSTEM1
assert SYSTEM1 [FD= SYSTEM2
```

A.4. Pila

```
-- Small Int representation
MAX_INT = 6
datatype SmallInt = SI.{0 .. MAX_INT} | Overflow
--
add(SI.a, SI.b) =
```

```
  let sum = a + b within if sum <= MAX_INT then SI.sum else Overflow
add(_, _) = Overflow

sub(SI.a, SI.b) =
  let sub = a - b within if sub >= 0 then SI.sub else Overflow
sub(_, _) = Overflow

mult(SI.a, SI.b) =
  let mult = a * b within if mult <= MAX_INT then SI.mult else Overflow
mult(_, _) = Overflow

eq(SI.a, SI.b) = a == b
eq(_, _) = false

-- Possible actor names
datatype ActorID = node.{1,2,3} | main.{1} | NoId

-- Possible Strings
datatype Atoms = PUSH | POP

-- Possible types for actors
datatype VALUE = ACTOR.ActorID | INT.SmallInt | ATOM.Atoms | None

-- Actor creation decoupling
channel CreateAsk:ActorID.(VALUE, VALUE)
channel Create:ActorID.(VALUE, VALUE)

create(actorId) = CreateAsk!actorId?m -> Create.actorId!m -> STOP
creates = ||| actor: ActorID @ create(actor)

-- Send ( actor to mailbox ) and Recv ( mailbox to actor ) communication
channel CommSend:ActorID.(VALUE, VALUE)
channel CommRecv:ActorID.(VALUE, VALUE)

MAILBOX_SIZE = 3

buff(left, right, <>) = left?msg -> buff(left, right, <msg>)

buff(left, right, xs) = if (length(xs) < MAILBOX_SIZE - 1) then
    buff_with_space(left, right, xs)
  else
```

```
    buff_full(left, right, xs)

buff_with_space(left, right, <x> ^ xs) =
  right!x -> buff(left, right, xs)
  []
  left?y -> buff(left, right, <x> ^ xs ^ <y>)

buff_full(left, right, <x> ^ xs) = right!x -> buff(left, right, xs)

mailboxes = ||| actor: ActorID @ buff(CommSend.actor, CommRecv.actor, <>)

-- become <buzón>
fwd(in, out) = CommRecv.in?msg -> CommSend.out!msg -> fwd(in, out)

-- Node actor
Nodes =
  ||| actorId : {|node|} @ Create.actorId?(INT.content, ACTOR.link) ->
  Node(actorId, content, link)

Node(self, content, link) =
  CommRecv.self?(ATOM.PUSH, INT.newContent) ->
  CreateAsk?node.newNode!(INT.content, ACTOR.link) ->
  Node(self, newContent, node.newNode)
  []
  CommRecv.self?(ATOM.POP, ACTOR.client) ->
  CommSend.client!(INT.content, None) ->
  fwd(self, link)

-- Main actor
Main =
    CreateAsk?node.pid!(INT.SI.3, ACTOR.NoId) ->
    CommSend.node.pid!(ATOM.PUSH, INT.SI.2) ->
    CommSend.node.pid!(ATOM.PUSH, INT.SI.1) ->
    CommSend.node.pid!(ATOM.POP, ACTOR.main.1) ->
    CommRecv.main.1?(INT.V, None) ->
    CommSend.node.pid!(ATOM.POP, ACTOR.main.1) ->
    CommRecv.main.1?(INT.V, None) ->
    CommSend.node.pid!(ATOM.POP, ACTOR.main.1) ->
    CommRecv.main.1?(INT.V, None) ->
    STOP
```

```
--
COMM = {|CommSend, CommRecv|}
INIT = {|Create, CreateAsk|}
--

SYSTEM =
  ( Nodes ||| Main )
    [|union(COMM, INIT)|]
  ( mailboxes ||| creates )
```

A.5. Anillo de actores

```
-- Small Int representation

MAX_INT = 6
datatype SmallInt = SI.{0 .. MAX_INT} | Overflow
--
add(SI.a, SI.b) =
  let sum = a + b within if sum <= MAX_INT then SI.sum else Overflow
add(_, _) = Overflow

sub1(SI.a) = sub(SI.a, SI.1)
sub(SI.a, SI.b) =
  let sub = a - b within if sub >= 0 then SI.sub else Overflow
sub(_, _) = Overflow

mult(SI.a, SI.b) =
  let mult = a * b within if mult <= MAX_INT then SI.mult else Overflow
mult(_, _) = Overflow

eq(SI.a, SI.b) = a == b
eq(_, _) = false

-- Possible actor names
datatype ActorID =
  ring.{1} | buildingRing.{1} | node.{1,2,3} | main.{1} | NoId

-- Possible Strings
datatype Atoms = POINT_TO | MSG

-- Possible types for actors
```

```
datatype VALUE = ACTOR.ActorID | INT.SmallInt | ATOM.Atoms | None

-- Actor creation decoupling
channel CreateAsk:ActorID.(VALUE, VALUE)
channel Create:ActorID.(VALUE, VALUE)

create(actorId) = CreateAsk!actorId?m -> Create.actorId!m -> STOP
creates = ||| actor: diff(ActorID, {main.1}) @ create(actor)

-- Send ( actor to mailbox ) and Recv ( mailbox to actor ) communication
channel CommSend:ActorID.(VALUE, VALUE)
channel CommRecv:ActorID.(VALUE, VALUE)
MAILBOX_SIZE = 3

buff(left, right, <>) = left?msg -> buff(left, right, <msg>)

buff(left, right, xs) = if (length(xs) < MAILBOX_SIZE - 1) then
    buff_with_space(left, right, xs)
  else
    buff_full(left, right, xs)

buff_with_space(left, right, <x> ^ xs) =
  right!x -> buff(left, right, xs)
  []
  left?y -> buff(left, right, <x> ^ xs ^ <y>)

buff_full(left, right, <x> ^ xs) = right!x -> buff(left, right, xs)

mailboxes = ||| actor: ActorID @ buff(CommSend.actor, CommRecv.actor, <>)

-- Ring actor
Rings =
  ||| self : {|ring|} @ Create!self?(None, None) ->
  Ring(self)

Ring(self) = CommRecv.self?(INT.n, INT.m) ->
  CreateAsk?node.nodePid!(INT.m, ACTOR.NoId) ->
  CreateAsk?buildingRing.brPid!(ACTOR.node.nodePid, ACTOR.node.nodePid) ->
  CommSend.buildingRing.brPid!(INT.(sub1(n)), INT.m) ->
  Ring(self)
```

```
-- Building Ring actor
BuildingRings =
  ||| self : {|buildingRing|} @ Create!self?(ACTOR.first, ACTOR.lastCreated) ->
  BuildingRing(self, first, lastCreated)

BuildingRing(self, first, lastCreated) = CommRecv.self?(INT.n, INT.m) ->
  if (eq(n, SI.0)) then
    CommSend.lastCreated!(ATOM.POINT_TO, ACTOR.first) ->
    CommSend.first!(ATOM.MSG, None) ->
    STOP
  else
    CreateAsk?node.nodePid!(INT.m, ACTOR.NoId) ->
    CommSend.lastCreated!(ATOM.POINT_TO, ACTOR.node.nodePid) ->
    CommSend.self!(INT.sub1(n), INT.m) ->
    BuildingRing(self, first, node.nodePid)

-- Node  actor
Nodes =
  ||| self : {|node|} @ Create!self?(INT.m, ACTOR.next) ->
  Node(self, m, next)

Node(self, m, next) =
  (CommRecv.self?(ATOM.POINT_TO, ACTOR.newNext) -> Node(self, m, newNext) )
[]
  ( CommRecv.self?(ATOM.MSG, None) ->
    if (eq(m, SI.0)) then
      CommSend.next!(ATOM.MSG, None) ->
      STOP
    else
      CommSend.next!(ATOM.MSG, None) ->
      Node(self, sub1(m), next)
 )
-- Main
Main =
  CreateAsk?ring.pid!(None, None) ->
  CommSend!ring.pid.(INT.SI.3, INT.SI.1) ->
  STOP
--
COMM = {|CommSend, CommRecv|}
INIT = {|Create, CreateAsk|}
--
```

```
SYSTEM =
  (Rings ||| BuildingRings ||| Nodes ||| Main)
    [|union(COMM, INIT)|]
  (mailboxes ||| creates)
```

Bibliografía

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] Gul Agha and Prasanna Thati. An algebraic theory of actors and its application to a simple object-based language. In Olaf Owe, Stein Krogdahl, and Tom Lyche, editors, *From Object-Orientation to Formal Methods*, volume 2635 of *Lecture Notes in Computer Science*, pages 26–57. Springer Berlin Heidelberg, 2004.

[3] Ponylang authors. *What is Pony?*, 2018.

[4] Francesco Cesarini and Simon Thompson. *ERLANG Programming*. O'Reilly Media, Inc., 1st edition, 2009.

[5] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2015, pages 1–12, New York, NY, USA, 2015. ACM.

[6] William Douglas Clinger. Foundations of actor semantics. 1981.

[7] Maximiliano Cristiá. *Introducción a CSP*. 2011.

[8] John Darlington and Yi-ke Guo. Formalising actors in linear logic. In Dilip Patel, Yuan Sun, and Shushma Patel, editors, *OOIS'94*, pages 37–53, London, 1995. Springer London.

[9] Mauro Gaspari and Gianluigi Zavattaro. An algebra of actors. In Paolo Ciancarini, Alessandro Fantechi, and Robert Gorrieri, editors, *Formal Methods for Open Object-Based Distributed Systems*, pages 3–18, Boston, MA, 1999. Springer US.

[10] Daniel D. McCracken and Edwin D. Reilly. Backus-naur form (bnf). In *Encyclopedia of Computer Science*, pages 129–131. John Wiley and Sons Ltd., Chichester, UK.

[11] Martin Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.

[12] Rob Pike. Concurrency is not parallelism. `http://blog.golang.org/concurrency-is-not-parallelism`, 2013.

[13] Hoare C. A. R. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

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

[15] Alexandre Boulgakov Thomas Gibson-Robinson, Philip Armstrong and A.W. Roscoe. *Failures Divergences Refinement (FDR) Version 3*, 2013.

[16] et al. Thomas Gibson-Robinson, Philip Armstrong. Fdr3 — a modern refinement checker for csp. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.

[17] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG (2Nd Ed.).* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[18] Derek Wyatt. *Akka Concurrency.* Artima Incorporation, USA, 2013.