Sistemas Informáticos

Curso 2002/2003

Utilización de herramientas basadas en Métodos Formales para el desarrollo de sistemas de comercio electrónico

Luis Lorido Sánchez
Luis Palazuelo Bretón
Sergio Romero Córdoba
José Luis Serna Gil

Dirigido por:
Prof. David de Frutos Escrig
Dpto. Sistemas Informáticos y Programación

Facultad de Informática
Universidad Complutense de Madrid

Índice:

Índice exhaustivo : **Pag**

Resumen en español:

El comercio electrónico es, sin lugar a dudas uno de los campos de la informática aplicada de mayor actualidad y previsible crecimiento en los próximos años. El mismo exige la producción de un software con garantías de seguridad que facilite la confianza de los potenciales usuarios, y con ello su rentabilidad.
El comercio electrónico es un ejemplo de software distribuido. A su vez, los sistemas distribuidos se pueden ver como una evolución de los sistemas concurrentes. Es bien sabido que la programación concurrente exige un cierto soporte formal si se quiere garantizar el buen funcionamiento de los sistemas, pues en otro caso, la explosión combinatoria de posibles ejecuciones que genera el no-determinismo, hace totalmente imposible capturar 'a ojo' todas las posibles alternativas.
El trabajo aquí desarrollado trata sobre el estudio y el diseño de sistemas distribuidos y sistemas basados en agentes móviles. Este documento esta estructurado en consecuencia con esta idea, claramente diferenciado en dos bloques, un primer bloque dedicado a los sistemas distribuidos, y un segundo bloque destinado a los sistemas basados en agentes móviles. El primer bloque consta de una primera parte de estudio sobre los métodos formales, y en concreto en el lenguaje CCS y sus variantes y como segunda parte del bloque el uso de herramientas formales, que usaremos sobre un ejemplo más o menos complejo. Análogamente el segundo bloque también consta de una primera parte de estudio y una segunda de aplicación práctica, en la cual detallaremos mediante cálculo de ambientes, un protocolo de intercambio de claves privadas.

Resumen en ingles:

e-comerce is one of the most topical subjects of information tecnologies and probably it will grow in the future. We need the production of realiable software with security guarantees which inspire constumer confidence and market profits.

e-comerce is an example of distributed software. And distributed software could be seen as an evaluation of concurrent systems. We know thet concurrent software needs a formal support in order to grant a proper operation. On the other hand we could have too many potential different processes which are generated by the non-determiniss of the system.

This project is about the study and design of distributed systems and system based in mobility agents. The structure of these documents follows this idea.

We distinguish between the two blocks. The first of them is about distributed systems and the second one is about systems based in mobility ambients.

The first block has two parts, a first part ot theorical subjects about formal methods and specificly about CCS and their variations. And a second part about formal utilities, which we employ in orden to run and test an example made by ours.

The second block also has two parts, very similar to the first block. The first part is about the theorical subject of Ambient Calculus and the second one is about a security protocol which we described using Ambient Calculus.

Bloque 1:

Cálculo para sistemas de comunicaciones

1.- Introducción

En este documento se piensa abordar el lenguaje CCS (Calculus of Communicating Systems). Este lenguaje se basa en los métodos formales para especificar programas concurrentes. En este caso la especificación está muy unida a la codificación, de forma que en algunos casos vienen a ser equivalentes o se requiere de un mínimo esfuerzo para su transformación.

Hablaremos previamente en una pequeña introducción sobre que es la concurrencia, y los programas concurrentes. Trataremos también la especificación de sistemas concurrentes.

Dentro en materia nos centraremos en los lenguajes formales para la especificación de programas concurrentes, y en concreto hablaremos de CCS. Para que todo esto no se quede en mera teoría, daremos algunos ejemplos de programas concurrentes especificados en CCS y explicaremos algunas de las herramientas que se pueden emplear para probar estos programas.

1.1.- Notación y Sinónimos

Lenguaje CCS → Calculus of Communicating Systems (Cálculo para sistemas de comunicaciones)
Proceso → Agente
Comunicaciones → Acciones
Traza → Secuencia de acciones

Nuestro alfabeto básico va a ser acciones complementarias. Para la entrada escribiremos un nombre perteneciente a nuestro alfabeto, y para la salida complementaria escribiremos el mismo nombre subrayado. Ejemplo: m y m.
Para indicar el agente que no hace nada o terminador, escribiremos: 0.
Para representar las acciones ocultas utilizaremos la letra: r.
El resto de notación estándar lo podemos ver en el apartado de 'Breve introducción de un lenguaje concurrente'.

2.- Concurrencia

2.1.- ¿Qué es concurrencia?

Los primeros ordenadores se limitaban a ejecutar un conjunto de instrucciones secuencialmente. El programador introducía el programa en el ordenador y éste ejecutaba las instrucciones una a una, es decir, desde el principio hasta el final del programa, el ordenador no se ocupaba de ninguna otra tarea. En aquellos ordenadores no existía interacción con el usuario, y tampoco con otros dispositivos, el ordenador era simplemente un procesador que se dedicaba exclusivamente a una instrucción por vez.

Con el avance de la tecnología los ordenadores empezaron a interactuar con el usuario o con el entorno en tiempo real. Además el microprocesador empezó a encargarse de todo tipo de dispositivos. Más adelante el ordenador paso a ocuparse de la ejecución de varios programas simultáneos. Para conseguir que varios programas se ejecuten simultáneamente se puede emplear un microprocesador o un conjunto de ellos. Cuando ejecutamos un conjunto de programas en un mismo microprocesador, sólo conseguimos

la apariencia de que todos ellos se están ejecutando al mismo tiempo. Básicamente lo que hace el microprocesador es ir ejecutando secuencialmente pequeñas porciones de cada programa, con lo que la apariencia es que todos se están ejecutando al mismo tiempo.

La idea intuitiva es que para ejecutar un conjunto de programas a la vez, se necesita un conjunto de procesadores que los vayan ejecutando. Pues bien, al conjunto de procesadores se les conoce como sistemas multiprocesadores. Estos sistemas sí son capaces de ejecutar diferentes programas simultáneamente, es decir, concurrentemente. La concurrencia significa que el conjunto de programas se puede ejecutar a la vez. Con esta idea tan simple de concurrencia, empiezan a surgir unos cuantos problemas. Uno de los principales problemas es que los recursos son limitados, puede que no tengamos tantos procesadores como programas, lo que sería absurdo, o que sólo dispongamos de una unidad de un tipo de dispositivo al que quieran acceder dos programas al mismo tiempo.

Concurrencia implica que existe un conjunto de recursos que son compartidos, y que por tanto puede que se requiera su uso simultáneamente. Concurrencia implica que los programas deben comunicarse unos con otros.

Para ilustrar todo esto vamos a poner un ejemplo: La reserva de entradas de cine. Esta reserva esta prevista que se haga por internet o por teléfono, en cualquiera de los dos medios diferentes personas pueden estar reservando su entrada al mismo tiempo, es decir, concurrentemente. Es posible que el ordenador o el encargado de asignar las entradas, ofrezca el mismo asiento a dos clientes diferentes, y como pueden imaginar es un problema molesto.

2.2.- Problemas a raíz de la concurrencia

A la hora de abordar programas concurrentes tenemos que tener en cuenta que nos enfrentamos a un problema de acceso y a otro de seguridad. El problema de acceso es el mismo que hemos visto en el ejemplo de las entradas. Cuando dos acciones simultáneas requieren el mismo recurso debemos controlar el acceso para que lo hagan en orden, es decir que no interfieran la una en la otra y viceversa. En el caso de las entradas la consecuencia es que un cliente verá la película de pie, pero existen casos más graves y por tanto debemos conseguir un mecanismo formal y demostrable a priori que esto no puede suceder.
El segundo problema atañe a la seguridad, en el ejemplo de las entradas un cliente no tiene porque conocer la identidad del otro. En el acceso concurrente a un recurso las acciones implicadas deben conocer lo mínimo necesario sobre la otra para poder funcionar, y en algunos casos serán independientes y no deberán conocer la existencia de la otra. Para ilustrar el caso imaginemos un tienda por internet. No conviene que a la hora de ir a pagar dos usuarios simultáneamente el programa equivoque las tarjetas de crédito de ambos clientes, por no suponer errores más graves.

Como conclusión decir que los programas que estén destinados a ejecutarse en un entorno concurrente deben ser debidamente especificados formalmente de forma que se pueda demostrar a priori que cumplen estrictamente con su cometido.

2.3.- Breve introducción de un lenguaje concurrente

Agente nulo: '0'. Escribimos un cero.
Operador de secuencia: '.'. Escribimos un punto. Ejemplo: C1=in.out.0
Operador de elección: '+'. Ejemplo: D2=in.(outa.0 + outb.0)
Acciones complementarias: m y m. Sirven para comunicar dos procesos.
Operador de concurrencia o paralelismo: 'I'.

Ejemplo: C1=in.m.C1
C2=m.out.C2
System1=C1IC2

Operador de restricción: '\'. Ejemplo: Buff2=(C1IC2)\{m}. Indicamos la acción oculta. Para representar que hay una acción oculta escribiremos para este ejemplo: in.r.out.

3.- Lenguaje para la concurrencia

El estudio de un lenguaje formal básico para especificar programas concurrentes se va a tratar a través de una serie de pequeños, aunque prácticos ejemplos tales como buffers, semáforos y recursos compartidos. Con estos ejemplos veremos algunos de los conceptos vistos anteriormente. Posteriormente veremos como combinar estos ejemplos simples, en sistemas más complejos.

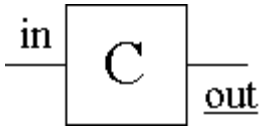
3.1.- Calculus of Communicating Systems

Calculus of communicating systems, que a partir de ahora vamos a denominar CCS, es un lenguaje que consiste en una serie de operadores de elección, secuencia, paralelismo y restricción.

Empezaremos definiendo un lenguaje basado solamente en una comunicación síncrona, sin considerar de momento el paso de valores, aunque más adelante veremos que esta lenguaje nos sirve para ello. Consideremos por tanto sólo el flujo de control de un sistema, sin tener en cuenta los datos que se emplean en la comunicación o con los que se opera. Tampoco tendremos en cuenta aspectos como el tiempo, pues se verán más adelante.

Para comenzar, veamos un simple semáforo que representaremos con la siguiente figura:

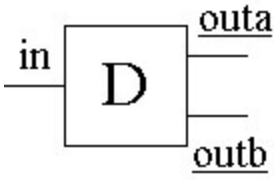
Este diagrama bien puede representar un proceso C que tiene dos conexiones con el entorno que hemos etiquetado como *in* and *out*.



Nos referiremos a estas etiquetas como acciones, y consideraremos las acciones no subrayadas como entradas y las subrayadas como salidas. El comportamiento esperado para este proceso es que espere hasta que reciba comunicación a través de la acción *in* y entonces establezca comunicación con la acción *out*. Recordar, que por el momento, no nos preocupa los datos que se comuniquen, sino el patrón de comunicación. También

hay que fijarse en que no hemos especificado el entorno en el que existe el proceso C. No hemos dicho con que otros procesos interactua C. A pesar de esto el diagrama muestra el patrón de comunicación que puede realizar C.

Miremos ahora este nuevo diagrama.



Ahora intentaremos identificar el comportamiento del proceso descrito por la figura anterior. El agente D tiene una acción *in* de entrada, pero dos acciones de salida *outa* y *outb*. Cómo interpretar el orden posible de las acciones. Suponer que deseamos que D no haga nada hasta que no reciba comunicación mediante su acción *in*, pero entonces cuál debe ser el orden de las comunicaciones hacia *outa* y *outb*. Quizás ambas acciones son simultáneas, o quizás una por vez en cualquier orden. O quizás una cada vez, pero en un orden determinado. Otro punto a tener en cuenta es que no hemos dicho que hacen C o D una vez que han terminado sus tareas. Quizás se quedan parados, o quizás vuelven a su estado inicial para empezar de nuevo. Observamos que estos diagramas no son suficientemente expresivos para indicar claramente el comportamiento de estos procesos particulares.

Volvamos a C y empecemos a construir un lenguaje que nos permita expresar suficientemente claro el comportamiento que esperamos de los procesos. El lenguaje que vamos a construir consiste en un conjunto de operadores que modelizan diferentes aspectos del comportamiento de los procesos. Los procesos, que vamos a llamar comúnmente agentes, en nuestro lenguaje estarán formados de acciones primitivas que a su vez construirán patrones de comportamiento según el significado de los operadores que definamos. El agente más simple que podemos imaginar es un agente que no haga nada, y que representaremos por el símbolo ‘0’.

El primer operador que vamos a introducir nos permitirá construir un agente que es capaz de realizar un secuencia de acciones y lo representaremos por el símbolo ‘.’. Todas las acciones serán atómicas y utilizaremos el conjunto de nombres *a*, *b*, *c*, etc... para representarlas. Ahora podemos representar uno de los posibles comportamientos del proceso C por el significado de la siguiente definición:

$$C1=in.\underline{out}.0$$

Esto significa que el agente llamado C1 esta definido por la expresión in.out.0. Informalmente podemos interpretar esta expresión como que el proceso C1 espera una comunicación de entrada a través de su acción *in*, y después realiza una comunicación de salida a través de su acción *out*, y partir de entonces no hace nada más. De aquí en adelante los nombres de los agentes los expresaremos en mayúsculas, al menos la primera letra. Los nombres de las acciones de forma parecida, pero empezando por una letra minúscula.

Otra posible interpretación del proceso C es que queremos que realice la acción *in*, a continuación la acción *out*, y así un número indeterminado de veces. La expresión que representa este comportamiento sería:

$$C2=in.\underline{out}.C2$$

Y vemos como en las definiciones de los agentes podemos referirnos a otros agentes en la parte derecha de la expresión. En este caso el agente se refiere a si mismo, y por tanto la definición es recursiva. Aclarar que cada agente sólo puede ser definido una vez. Sólo puede aparecer una vez en la parte izquierda entre todas las definiciones. Dicho de otra forma, C2 sólo puede tener una única definición, como por ejemplo la de arriba, para cualquier sistema en el que aparezca.

Volviendo al proceso D, miramos los posibles comportamientos que nos gustaría que realizase. Sin añadir más al lenguaje podemos definir D de la forma:

$$D1=in.\underline{outa}.\underline{outb}.D1$$

Que es un proceso que realiza la acción *in* seguida de la acción *outa*, a continuación la acción *outb*, para volver a llamar a D1 y repetir toda la secuencia de acciones. Sin duda podíamos haber cambiado el orden de las salidas. Para empezar a expresar otros posibles comportamientos de D, tenemos que introducir nuevos operadores. El primero de los operadores que vamos a incorporar, es el operador que nos permite elegir la siguiente acción, para ello utilizaremos el símbolo '+' para designarlo. Podemos llamarlo el operador de elección. Podemos ahora definir D de la siguiente manera:

$$D2=in.(\underline{outa}.0+\underline{outb}.0)$$

Define un proceso que primero se comunica son su acción *in* para elegir a continuación cualquiera de las dos comunicaciones a través de $\underline{outa}$ o $\underline{outb}$. Después de cualquiera de estas dos acciones el proceso D2 queda inactivo. Otra posibilidad para el comportamiento de D puede ser:

$$D3=in.(\underline{outa}.D3+\underline{outb}.D3)$$

Esta definición del proceso se comporta inicialmente como el anterior, pero esta vez puede repetir la elección de acciones tantas veces como quiera. Hay que fijarse en que la elección es entre dos agentes dependiendo de cual realice su acción primero. La elección no es sobre acciones, y por lo tanto no podemos escribir:

$$...\underline{outa}+\underline{outb}.0$$

En casos como el de arriba es el entorno el que determina cual de las acciones ofrecidas es la que realmente se realizará.

Existen otras posibles definiciones para D que incluyan también la secuencia y la elección. En la siguiente definición hay un elección no determinista que se hace cuando la acción *in* es realizada, que lleva a $\underline{outa}$ o $\underline{outb}$ como siguiente acción, pero no ambas.

$$D4=in.\underline{outa}.D4+in.\underline{outb}.D4$$

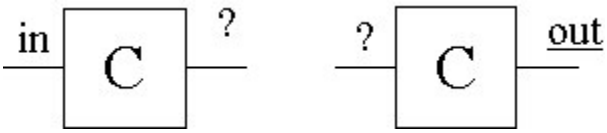
En este caso el entorno sólo tiene la acción *in* para elegir, pero no es suficiente para determinar el estado que será alcanzado como resultado. El agente definido como D4 es capaz de realizar las siguientes secuencias de acciones:

in.outa.in.outb.in.outb.in.outa...

in.outb.in.outb.in.outa.in.outb...

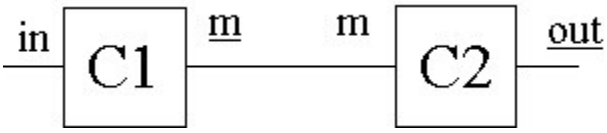
in.outa.in.outa.in.outa.in.outa...

Hasta aquí hemos considerado el comportamiento de un único proceso o agente y hemos visto como un simple diagrama pude tener una serie de posibles expresiones que lo definan. Nos gustaría tener una serie de procesos ejecutándose concurrentemente y siendo capaces de interactuar entre ellos. Suponer por ejemplo que queremos dos copias de C; volvamos a usar los nombres de C1 y C2 respectivamente, comunicándose entre ellos de forma que una copia, C1, dirá a la otra, C2, que ha recibido una comunicación. Un posible diagrama para el sistema lo podemos ver en el siguiente dibujo. En este caso C1 dirá a C2 que ha recibido una comunicación y se quedará esperando la siguiente.



Mientras tanto C2 será capaz de comunicar este hecho al entorno y después volver a esperar a que C1 informe se su siguiente comunicación.

De alguna manera deseamos tener formas de definir cuando dos procesos pueden comunicarse y por ello debemos ampliar nuestro lenguaje para incluir dos o más procesos que existan concurrentemente. Primero de todo permitiremos a los procesos comunicarse si ambos tienen acciones complementarias disponibles. Diremos que un par de acciones son complementarias si ambas tiene el mismo identificador, pero una esta subrayada y la otra no. Por ejemplo las acciones m y m en la siguiente figura son acciones complementarias.



Cuando C1 recibe comunicación a través de *in* entonces puede mandar una comunicación a través de su puerto *m*, y C2 puede recibir esa comunicación a través de su puerto *m*. De esta forma C1 y C2 tiene acciones siguientes complementarias. Definimos C1 y C2 de la siguiente manera:

C1=in.m.C1
C2=m.out.C2

Nos gustaría que ambos procesos realizasen estas acciones concurrentemente. Vamos a emplear el nuevo operador 'l' para representar la composición de dos procesos y definir entonces el sistema de la siguiente forma:

Sistema1=C1|C2

Intuitivamente deseamos que Sistema1 en su estado inicial permita que C1 y C2 puedan operar independientemente, o si sus acciones lo permiten entonces cooperar y operar concurrentemente. En este caso la cooperación puede tener lugar después de que C1 haya realizado su acción *in* y C2 siga esperando la comunicación por su puerto *m*.

Una manera de interpretar Sistema1 es que representa una posible especificación de un buffer de dos sitios que puede almacenar un máximo de dos comunicaciones a la vez. Sin embargo C2 puede seguir comunicándose independientemente con el entorno a través de m sin necesidad de esperar a que C1 le notifique que ha recibido una comunicación. De forma similar C1 no tiene porque sincronizarse con C2 cuando quiera hacer su acción $\underline{m}$. Para hacer que Sistema1 se comporte como un buffer real necesitamos asegurar que C2 sólo se comunica a través de m con C1 y con ningún otro proceso del entorno. Para alcanzar este objetivo tenemos que introducir el concepto de restricción, las acciones no pueden tener lugar cuando quieran y además son ocultas a cualquier observador externo. En el ejemplo actual deseamos que las acciones $\underline{m}$ y *m* de los procesos C1 y C2 respectivamente estén restringidas en el proceso Sistema1 y por tanto invisibles. Esencialmente el efecto de esto sobre C2 será que no puede realizar ninguna acción hasta que sincronice con C1 a través de *m*.

Introducimos un nuevo operador '\' que restringe las acciones observables de un proceso. La expresión final para nuestro buffer será:

Buff2=(C1|C2)\{m}

Que intuitivamente expresa que las acciones observables de las posibles acciones de C1 y C2 no incluye las acciones *m* o $\underline{m}$. Sin embargo nos gustaría registrar que Buff2 esta realizando tanto acciones observables como no observables. Hacemos esto empleando el símbolo 'r', tau, para registrar la presencia de acciones no observables, invisibles. Sería imposible tener *r* como un elemento de un conjunto de restricciones, y así las acciones *r* no serán afectadas por una restricción.

Ejemplos de posibles secuencias de acciones de Buff2 son:

in.r.$\underline{out}$.in.r.in.$\underline{out}$...
in.r.in.$\underline{out}$.r.$\underline{out}$.in...

Buff2 esta listo para aceptar hasta dos comunicaciones a través de la acción *in* y entonces tener al menos una comunicación con su acción $\underline{out}$ antes de aceptar más comunicaciones de entrada. Por las mismas razones cada comunicación a través de $\underline{out}$ deben ser precedidas por una comunicación por *in*.

A pesar de la simplicidad de este ejemplo, nos es útil para ilustrar la técnica o método para construir especificaciones de sistemas que incluyan procesos que se comunican concurrentemente. Podemos construir procesos más complejos a partir de otros más

simples utilizando la composición para permitir a los procesos comunicarse y restringir u ocultar comunicaciones internas con otros procesos. Utilizaremos estas ideas en ejemplos más adelante.

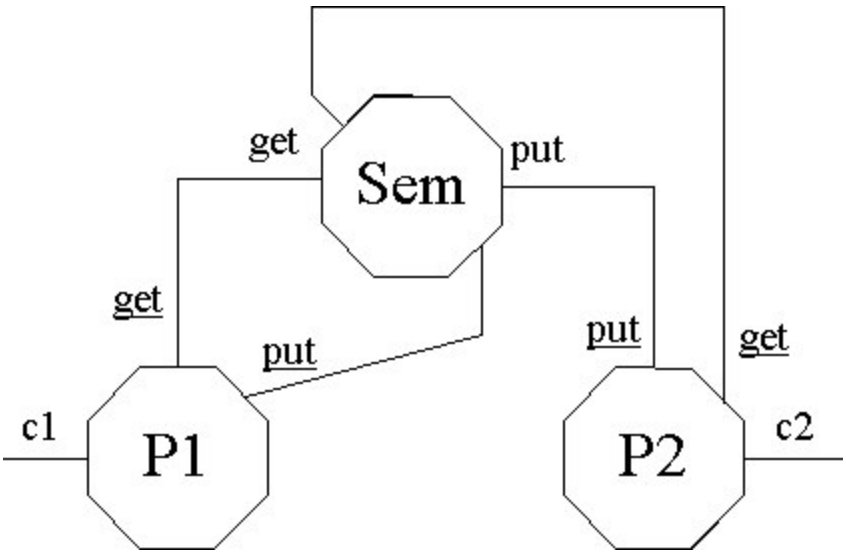
3.2.- Exclusión mutua

Nos gustaría especificar un sistema que sólo permitiese a un único proceso de entre dos, realizar una serie de tareas críticas a un tiempo. En otras palabras, queremos un sistema simple que muestre una forma de exclusión mutua. Un ejemplo sería un semáforo que controlase el acceso a las regiones críticas de los procesos. Vamos a volver a usar un proceso C, pero esta vez para representar un semáforo en nuestro sistema. Para evitar confusiones renombraremos el proceso C como Sem. Se define así:

Sem=get.put.Sem

Fijarse que Sem tiene un patrón de comportamiento parecido a C, excepto que los nombres de las acciones han cambiado. Ahora nos gustaría definir dos procesos que fuesen incapaces de realizar sus respectivas tareas críticas en paralelo. Llamaremos a estos procesos P1 y P2 y a sus respectivas acciones críticas c1 y c2. De momento no estamos interesados en la naturaleza de estas acciones pero si en que estas no puedan realizarse de forma simultánea. Ambos P1 y P2 pueden realizar su tarea crítica si consiguen el control del semáforo Sem.

El semáforo estará libre si la siguiente acción que puede realizar es *get* y de la misma forma estará ocupado si la siguiente acción que puede realizar es *put*. P1 y P2 deben competir por el control de Sem, y esto lo pueden conseguir mediante una comunicación satisfactoria a través de su puerto *get*. Teniendo el control de Sem, un proceso puede realizar su acción crítica. Las definiciones de P1 y P2 son por tanto simples:



P1=get.c1.put.P1
P2= get.c2.put.P2

Y el sistema entero quedaría definido como:

Mx=(P1|P2|Sem)\{get,put}

Un punto importante a tener en cuenta es que no permitimos sincronización múltiple entre las mismas acciones. Si esto no fuese así, no tendríamos la exclusión mutua que pretendíamos. La semántica de nuestro lenguaje debe reflejarlo así.

3.3.- Semántica informal

Antes de avanzar en ejemplos más complicados, resumamos la sintaxis de CCS vista hasta el momento para ver de forma más clara las expresiones que están permitidas. Informalmente definiremos el significado de los operadores que hemos introducido, antes de ver la definición formal en un capítulo posterior.

0 Este es el agente que ha terminado o esta bloqueado y no tiene comportamiento futuro.

a.E El agente E está precedido por la acción a. Este es el agente que primero realiza la acción a y luego el agente E.

E+F El agente puede comportarse como el agente E o como el agente F, pero no ambos.

E|F Este agente puede comportarse como E y F en paralelo y además puede comportarse de forma síncrona cuando E y F se comunican mediante acciones complementarias. Está sincronización sólo tiene lugar entre dos agentes y no más.

E\L El agente cuyas acciones observables, sean de entrada o salida, no deben aparecer en el conjunto L. Fijarse que las acciones r siendo invisibles no pueden ser restringidas.

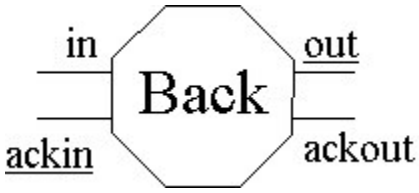
P Para cada agente llamado P debe haber una única definición de la forma P=E. El comportamiento de P es exactamente el comportamiento de E. Si E puede realizar la acción a para ser el agente E', entonces P puede realizar la misma acción para ser E'.

(E) El comportamiento del agente expresado entre paréntesis es exactamente el comportamiento del agente expresado sin paréntesis.

3.4.- Composición

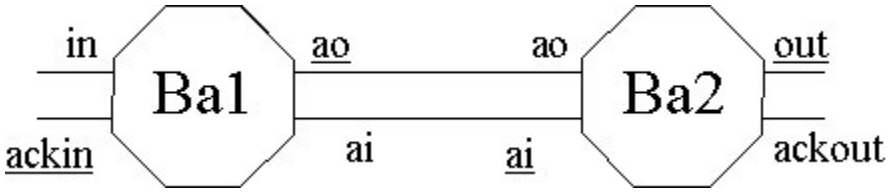
Parece que componer agentes en paralelo es directo y que siempre podemos componer objetos simples para conseguir otros más complicados y que se comporten como queremos. Este no es siempre el caso y hay que poner especial cuidado para asegurar que se conserva el comportamiento, y que ninguno adicional, normalmente indeseados, no aparezca. El objetivo del siguiente ejemplo es ilustrar el cuidado necesario que hay que tener cuando usamos la composición. También se busca demostrar la necesidad de ser precisos en la caracterización del comportamiento del sistema que estamos estudiando.

Consideremos un buffer que siempre reconoce las comunicaciones que recibe y espera reconocimiento de aquellas que envía. Lo llamaremos el agente Back.
Siendo una posible definición:



Back=in.out.ackout.ackin.Back

Todos los reconocimientos se hacen después de la recepción y envío de datos.
Suponer que queremos construir un buffer con capacidad 2, B2ack, que se comporte de forma similar excepto que puede almacenar hasta dos comunicaciones a un tiempo. Usando la misma técnica que utilizamos con Buff2, componemos dos copias de Back en paralelo con las acciones debidamente renombradas para que puedan comunicarse de la manera deseada. Definimos de la forma:



Ba1=in.ao.ai.ackin.Ba1
Ba2=ao.out.ackout.ai.Ba2
B2ack=(Ba1|Ba2)\{ai,ao}

Sin embargo, si quisiésemos investigar los posibles comportamientos de B2ack nos daríamos cuenta de que dista mucho de comportarse como una versión extendida de Buff2. En realidad se comporta exactamente como Back con el añadido de una serie de acciones r internas. Los comportamientos observables de B2ack y Back son exactamente los mismos, puesto que los dos son capaces de almacenar una sola comunicación a la vez. De hecho si compusiésemos tantas copias de Back como quisiésemos de la misma forma, terminaríamos con un sistema que se comportaría externamente como Back.

Es una simple cuestión de reorganizar las secuencias de eventos y llegar a una nueva definición con el comportamiento correcto, que nos permita especificar un verdadero buffer con capacidad n y que reconozca las comunicaciones.

B'ack=in.ackin.out.ackout.B'ack

3.5.- Ejemplo de los productores y consumidores

Nuestro siguiente ejemplo es un sistema que integra un cierto número de conceptos ya discutidos. En particular nos interesa la forma en que vamos a construir una especificación en CCS.

Requisitos:
Se nos pide especificar un sistema que use dos productores A y B, para almacenar objetos en un buffer de capacidad 4 que puede almacenar dos tipos de valores (ceros y

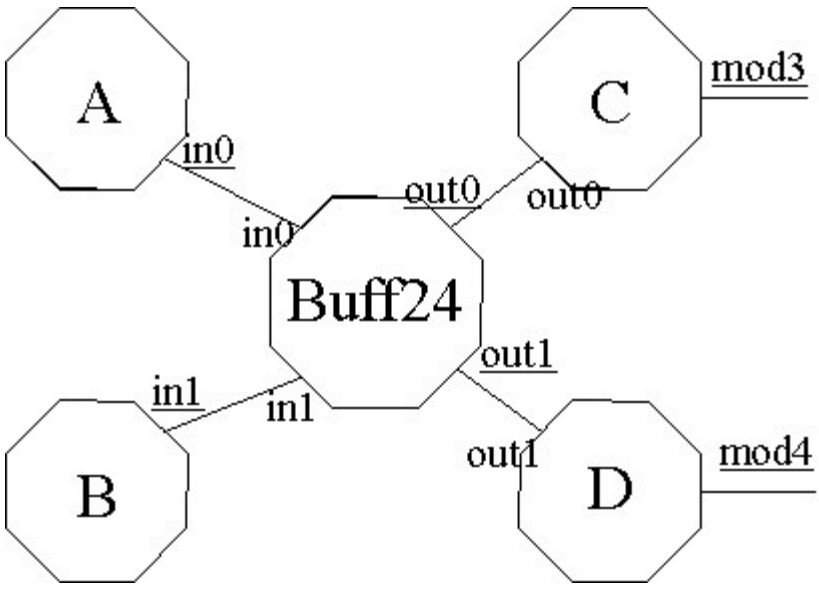
unos). Dos procesos más C y D consumen los objetos del buffer para poder realizar sus acciones particulares. El primero de estos pide tres ceros al buffer y el segundo pide cuatro unos al buffer. Los nombres de las dos acciones críticas son mod3 y mod4 respectivamente. No hay control sobre el orden en que los objetos son almacenados o leídos en el buffer, guardado en el estado del buffer mismo. Las únicas acciones observables del sistema en conjunto son las ocurrencias de las acciones mod3 y mod4.

Aproximaremos este problema usando el método introducido en este capítulo. Especificaremos un cierto número de agentes pequeños para luego combinarlos usando la composición y la restricción, para producir la especificación completa. Un breve estudio de los requerimientos es suficiente para darse cuenta que hay potencialmente cinco subagentes en el sistema, A, B, Buff24, C y D. Recordar que estamos hablando de la especificación del sistema y no todavía de la fase de implementación. Por esta razón podemos estructurar la especificación a nuestro gusto, siempre que cumpla los requisitos sin pensar todavía en cuestiones como la eficiencia.

De la información obtenida hasta ahora de los requisitos podemos hacer la siguiente tentativa como especificación:

PC=(A|B|Buff24|C|D)\L

Advertir que hemos definido | como un operador binario en la sintaxis y por tanto en la expresión de arriba deberíamos emplear paréntesis para expresar PC correctamente. Descubriremos más adelante que en estos casos, donde hay múltiples instancias del mismo operador, los paréntesis no afectan al comportamiento, y podemos omitirlos. Las comunicaciones entre estos agentes son bastante simples como muestra la siguiente figura.



Sin duda todavía no tenemos las definiciones de los subagentes y tampoco conocemos cuál debe ser el conjunto de restricciones L, pero siguiendo la buena costumbre descendente hemos reducido un problema grande en un conjunto de problemas más simples. Empecemos con los productores A y B. Todo lo que tienen que hacer es producir valores para alimentar el buffer. Sus definiciones son las siguientes:

A=in0.A
B=in1.B

Podemos continuar con los otros agentes y considerar los patrones de comunicación más adelante. De los agentes restantes los dos consumidores son los más fáciles puesto que sólo tienen que leer un predeterminado número de unos o ceros del buffer y producir las salidas *mod3* o *mod4* según corresponda. Aquí las tenemos:

C=out0.out0.out0.mod3.C
D=out1.out1.out1.out1.mod4.D

Ya sólo nos queda la definición del buffer de capacidad cuatro con posibilidad de manejar dos tipos de valores. Aproximaremos este problema prenguntándonos si hemos resuelto problemas parecidos con anterioridad, y la respuesta es sí. Anteriormente en este capítulo hemos especificado una serie de buffers de varios tipos. Simplemente tenemos que expandirlo a un buffer de capacidad 4. Para hacerlo necesitaremos cuatro elementos o celdas a componer en paralelo. Cada celda estará en uno de estos tres estados: cuando no tiene nada, cuando tiene un cero, o cuando tiene un uno. Semejante estado se define de la forma:

Cell=in0.out0.Cell+in1.out1.Cell

Inicialmente el agente Cell está en el estado donde no tiene ningún valor. Si comunica mediante *in0*, entonces pasará a tener un cero, y si establece comunicación por *in1* entonces tendrá un uno. En ambos casos el agente sólo puede comunicar el valor que ha recibido. Si componemos cuatro de estos agentes en paralelo dispondremos del buffer de capacidad cuatro que buscábamos. Para asegurarnos de que el buffer se comporta como es debido, tendremos que renombrar alguno de los puertos de forma que puedan interactuar y entonces restringir todo excepto los puertos *in0*,*in1*,*out0*,*out1*. Tenemos por tanto:

Cell1=in0.z1.Cell1+in1.o1.Cell1

Cell2=z1.z2.Cell2+o1.o2.Cell2

Cell3=z2.z3.Cell3+o2.o3.Cell3

Cell4=z3.out0.Cell4+o3.out1.Cell4

Y la especificación del buffer:

Buff24=(Cell1|Cell2|Cell3|Cell4)\{z1,z2,z3,o1,o2,o3}

Todo lo que queda por hacer es componer los diferentes agentes y asegurarnos que nada es visible excepto las acciones *mod3* y *mod4*, y esto se consigue de la siguiente manera:

PC=(A|B|Buff24|C|D)\{in0,in1,out0,out1}

Un punto interesante a tener en cuenta acerca de esta especificación es que Buff24 es un subsistema restringido anidado dentro de otro, así que hemos empleado nuestro método dos veces en este ejemplo. Advertir también que deliberadamente hemos reutilizado una especificación existente donde nos convenía y por tanto hemos empleado las técnicas de desarrollo tanto ascendente como descendente. CCS esta particularmente bien acondicionado a esta aproximación de resolver problemas.

4.- Definición formal

El objetivo principal de este capítulo es establecer una semántica formal para el lenguaje que acabamos de introducir. La semántica del lenguaje esta principalmente dirigida a caracterizar el comportamiento de los sistemas. Vamos a ver antes algunas posibles aproximaciones para después introducir la semántica estándar de CCS. Hemos introducido la noción de sistema de transiciones etiquetadas como la base para la definición formal del lenguaje que hemos estado usando. La definición formal en si misma viene dada en términos de una simple semántica operacional. Finalmente en este capítulo investigaremos algunas semánticas alternativas de CCS y las compararemos con la semántica estándar.

En la especificación de sistemas de comunicaciones concurrentes estamos interesados particularmente en los patrones de comunicaciones o sincronización entre los agentes de un sistema. Al igual que el sistema como un todo evoluciona, también los agentes individualmente lo hacen, aunque como es lógico evolucionan a distintas proporciones y de formas diferentes cada uno de ellos. Es importante como hemos visto encontrar alguna forma de modelar la evolución de un sistema, y en particular sus patrones de comunicación. Sistema en este contexto se refiere a expresiones de agentes que a su vez son especificaciones de sistemas. Como siempre tenemos que tomar la decisión de exactamente que es lo que queremos modelizar, y como lo vamos a hacer.

Un número de posibilidades se ofrecen por si mismas como modelos potenciales de sistemas concurrentes. Ciertamente queremos registrar los posibles estados a los que un sistema puede evolucionar a través de las acciones de sus diversos agentes. En un determinado momento puede haber un número de agentes, cada uno de ellos capaz de realizar una serie de acciones, mientras que otros pueden estar inactivos, posiblemente hasta que sean capaces de comunicarse con otros procesos. Una forma de modelar estos cambios de estado de un sistema sería modelar todos los cambios de un estado que puede ocurrir como resultado de alguna de las posibles permutaciones de las acciones concurrentes que el sistema es capaz de realizar. Si consideramos este agente simple:

$$Perm = a.0 | b.0 | c.0$$

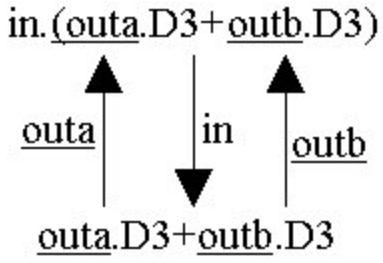
Es obvio que cualquier permutación del conjunto {a,b,c} resultará en un posible cambio de estado. Realizando las tres posibles acciones tendremos el sistema 0|0|0 mientras que realizando a y b resultará en el sistema 0|0|*c*.0. Podemos llevar registro de los efectos de otras permutaciones.

Otro acercamiento resulta de la observación de que las únicas acciones que podemos asegurar que tendrán lugar paralelamente son las sincronizaciones entre agentes. En el resto de los casos el orden de los eventos puede depender del observador con respecto al sistema. En este caso es posible para los observadores en distintas posiciones relativas

al sistema observar el mismo cambio de estado resultante de diferentes permutaciones del mismo conjunto de acciones. En esta aproximación representamos la ocurrencia paralela de un número de posibles eventos como un conjunto de sus posibles entrelazamientos. Si retomamos el pequeño ejemplo usado anteriormente podemos observar que cualquier orden de los eventos *a*,*b*,*c* dará lugar al mismo estado final. Así, realizando cualquier secuencia de acciones *a*.*b*.*c*, *b*,*a*,*c* o *c*.*a*.*b*, etc... dará lugar al estado final 0|0|0.

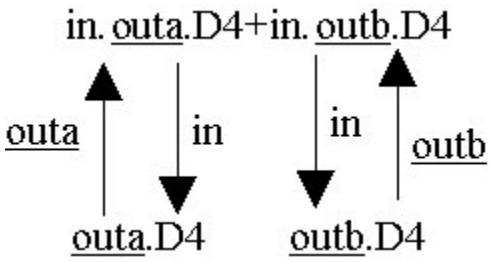
Es esta aproximación del entrelazamiento la que vamos a utilizar por varios motivos, el más importante de ellos es que nos permite definir una simple e intuitiva semántica para el lenguaje. Una forma de describir estos entrelazamientos del comportamiento de los agentes en nuestro lenguaje, es en términos de grafos de transición. Introduciremos los grafos de transición refiriéndonos a algunos de los sencillos agentes vistos con anterioridad, para volver más adelante al ejemplo de *Perm*.

Un grafo de transición para el agente *D3* se puede ver a continuación.



D3=in.(outa.D3+outb.D3)

Podemos ver que el grafo tiene dos estados representados por las expresiones de los agentes con las acciones que cada agente puede realizar con el fin de evolucionar a un nuevo estado etiquetando los arcos que conectan estos estados. Un grafo de transición es un conjunto de nodos, y cada cual es la expresión de un agente, junto con un conjunto de arcos que conectan esos nodos y donde cada arco esta etiquetado por una acción. Vamos a ver ahora el agente D4 que puede realizar la misma secuencia de acciones que D3, aunque tiene diferentes definiciones y aparentemente se comportan de distinto modo. Si construimos el grafo de transición para D4 veremos que ciertamente tiene un comportamiento distinto:

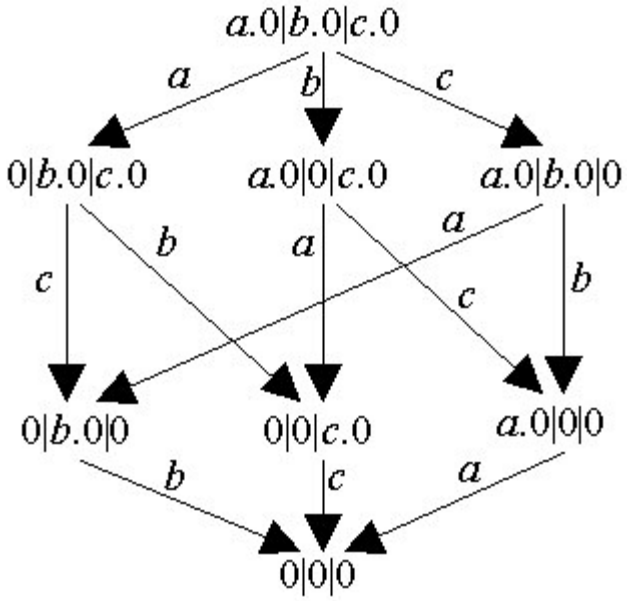


D4=in.outa.D4+in.outb.D4

D4 tiene tres estados en oposición con D3 que sólo tiene dos. Las posibilidades que pueden realizar los agentes en cada estado son también diferentes. Los grafos muestran claramente esa diferencia causada por elección no determinista hecha por D4 a la hora

de realizar su acción *in*. Los grafos de transición de estos dos agentes revelan ciertamente diferencias importantes que un simple listado de las posibles secuencias de acciones no muestra.

A continuación se muestra un grafo de transición para el agente $a.0|b.0|c.0$ introducido al comienzo de este capítulo:



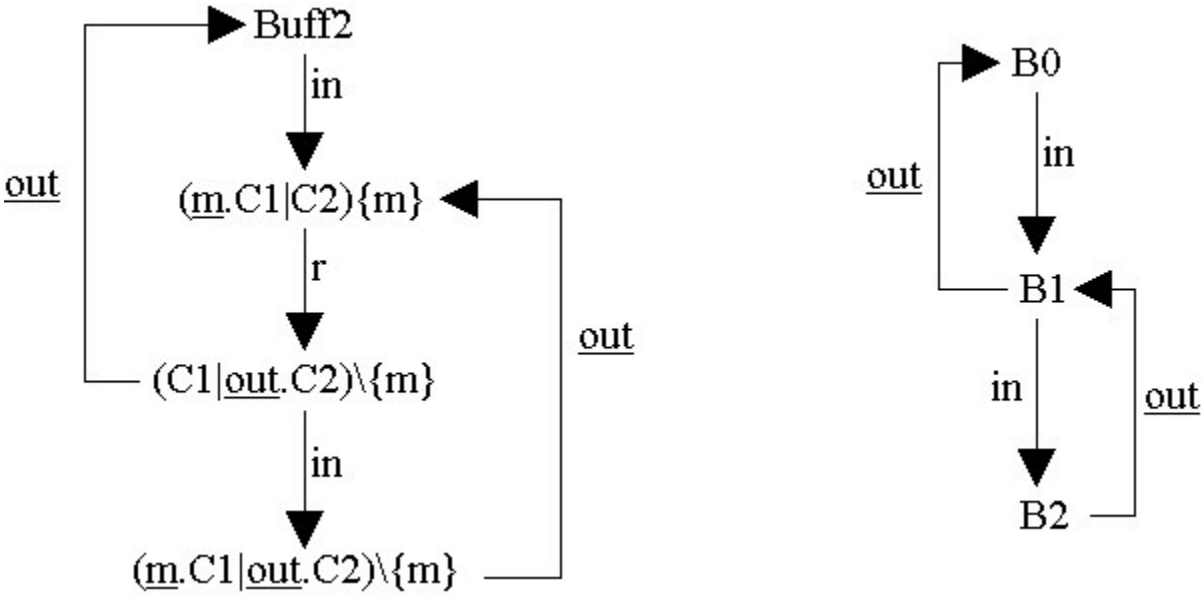
Este grafo muestra claramente como cualquier posible entrelazamiento de las acciones permitidas produce el mismo estado final. Tener presente también que los agentes 0 y 0|0|0 tendrían también los mismo grafos de transición puesto que ninguno de los dos es capaz de realizar ninguna acción. Se puede decir por tanto que hay expresiones equivalentes.

Para profundizar más sobre el tema vamos a ver un algoritmo que puede ser de utilidad para construir grafos de transición:

1. Elegir la expresión de un agente con la que queramos empezar. Si existe una que representa el estado inicial de un sistema, entonces coger esa.
2. Encontrar todas las posibles acciones para el agente escogido y dibujar un arco etiquetado con el nombre de la acción para cada una de las posibilidades. Recordar que acciones idénticas pueden generar diferentes siguientes estados, así algunos arcos del mismo nodo pueden tener la misma acción etiquetada.
3. Trabajar con las expresiones de los agentes que pueden aparecer como estados siguientes al final de cada uno de los arcos y añadir estos al grafo si es necesario. Esto es así porque es bastante posible que algunos de los estados puedan ya existir como un nodo del grafo y por tanto no hace falta añadir nuevos nodos para cada arco.
4. Comprobar si algún nodo tiene acciones siguientes sin sus apropiados arcos etiquetados. Si existe alguno, entonces volver al paso 2 para cada nodo en que esto sea cierto y repetir el proceso.
5. Todos los nodos deberían tener ya arcos para todas las posibles acciones que puedan derivarse de ellos. Puedes encontrar que es necesario redibujar el grafo un número de veces antes de conseguir una versión clara y simple del mismo. Suele ser mejor

dejar esta parte para el final cuando ya tienes todos los estados y arcos etiquetados necesarios.

Es interesante considerar que los grafos de transición para las dos versiones del buffer con capacidad dos utilizadas en el capítulo anterior, Buff2 y B0. Ambas versiones del ejemplo del buffer parecen tener diferentes grafos de transición, aunque supuestamente tiene el mismo comportamiento. La diferencia es la presencia del arco etiquetado por la acción *r*, que significa que un agente tiene que realizar una transición interna antes de poder continuar, mientras que el otro agente no tiene porque. Si ignoramos la *r* o la acción interna entonces el proceso tiene ciertamente el mismo comportamiento. Se muestran a continuación ambos grafos:



Los grafos de transición son muy útiles a la hora de reflejar claramente el espacio de estados de una especificación en particular.

Vale la pena comparar los grafos de transición usados en este capítulo con los grafos de flujo utilizados en el capítulo anterior. Los grafos de flujo nos dan una vista estática de las posibles comunicaciones entre los agentes de un especificación, pero no dicen nada acerca de la secuencia de estos eventos. Los grafos de flujo tienen similitudes con los diagramas de flujo de datos (DFDs). Un grafo de transición, por otra parte, da completa información sobre el orden de las acciones del sistema completo, pero lo que no nos da obviamente es el comportamiento individual de los agentes de ese sistema, aunque se podría obtener esta información si hiciera falta. Los grafos de transición tienen por tanto similitudes con los diagramas de transición de estados (STDs). Nuevamente vemos que estas técnicas son útiles en presentar la información de una forma fácil de asimilar, pero selectiva en cuanto a la información que actualmente ofrecen. Ciertamente uno puede considerar que semejante presentación selectiva de los datos es uno de los puntos fuertes de un buen diagrama. El diagrama nos dijera todo lo que queremos saber sobre un proceso o un sistema sería probablemente demasiado complejo y virtualmente inútil.

Una vez vistos los grafos de transición nuestro siguiente paso es definir esto formalmente.

4.1.- Semántica operacional

En esta sección vamos a definir formalmente los grafos de transición usados anteriormente. Sin embargo lo primero de todos es indicar los conjuntos que se usarán y las variables que se usarán bajo esos rangos.

A: Un infinito conjunto de nombres a, b, c,... sobre el alfabeto A. Unos ejemplos de nombres son *in*, *get*, *put*, etc.
$\underline{A}$: Un infinito conjunto de co-nombres $\underline{a}$, $\underline{b}$, $\underline{c}$,... sobre $\underline{A}$. Unos ejemplos de co-nombres son $\underline{get}$, $\underline{put}$, $\underline{out}$, $\underline{mod3}$.
L: El conjunto de etiquetas AU$\underline{A}$, l, l',... sobre L. Usaremos H y J para definir subconjuntos de L.
Act: El conjunto de todas las posibles acciones LU{r}, α y β definidas sobre Act.
K: El conjunto de los agentes constantes A, B,... sobre K.. y Mx, Buff2, B0 son unos ejemplos de esto.
E: El conjunto de las expresiones de los agentes dado por la siguiente notación BNF:

E ::= 0
| A
| E\H
| (E)
| α.E
| E|E
| E+E

donde los operadores están listados en orden descendente de prioridad.
X: El conjunto de las variables de agente Y, Z,... sobre X.

Insistimos en que no hay una ecuación que defina 0, mientras que para cada constante existe una única definición de la forma B=F. Denotar también el uso de H en la regla del operador \. Esto es así para asegurar que las acciones r no pueden ser incluidas en esas restricciones. Una expresión de la forma D+F|G equivale a D+(F|G). Sin embargo si queremos significar que la elección es entre D y F, y compuesto después en paralelo con G, entonces debemos escribir (D|F)+G. Lo mismo debemos aplicar para las expresiones de la forma D+F+G.

Con estos preliminares en mente podemos ahora adentrarnos en la definición formal de los operadores que hemos estado usando. Haremos esto en términos que se conocen como semántica operacional que formalizarán el comportamiento que queremos atribuir a cualquier agente CCS que construyamos. En otras palabras, describirá detalladamente que transiciones es capaz de realizar nuestro agente CCS, y por supuesto asociar los agentes CCS con sus grafos de transición. La semántica operacional puede ser pesada a la hora de definir un intérprete de CCS.

La semántica operacional para CCS consiste en un conjunto de reglas para cada operador. Cada regla tendrá cero o más hipótesis y una sola conclusión, con la hipótesis sobre una barra horizontal y las conclusiones debajo. En algunos casos una condición adicional será añadida entre paréntesis a la derecha de la regla. El significado de la condición forma parte obviamente de la regla misma. Además de una o más reglas para

cada operador habrá una regla que defina el significado de las constantes en las definiciones de las expresiones de los agentes.

Introduciremos las reglas para cada operador dando una breve explicación.
La primera regla se refiere al operador de prefijo que dice que un agente de la forma α.E puede realizar la acción α para convertirse en el agente E sin ninguna hipótesis.

$$\text{Pref}\ \frac{}{\alpha.E \xrightarrow{\alpha} E}$$

Hay dos reglas para la elección, dependiendo de que rama de la elección evalúe la expresión, y ambas son ligeramente más complicadas que la regla anterior. Estas reglas dicen que si un agente tiene una transición a través de α a E' entonces E+F tiene la misma transición que le lleva al mismo estado E'. La segunda regla de la elección dice los mismo para F en E+F:

$$\text{Sum1}\ \frac{E \xrightarrow{\alpha} E'}{E+F \xrightarrow{\alpha} E'} \qquad \text{Sum2}\ \frac{F \xrightarrow{\alpha} F'}{E+F \xrightarrow{\alpha} F'}$$

Existen tres reglas para la composición. Las dos primeras tratan con las posibles acciones observables para un agente de la forma E|F donde no hay involucrada ninguna sincronización. Siguen el mismo patrón que **Sum1** y **Sum2** excepto que el primer agente de la composición permanece después de que la transición ha ocurrido:

$$\text{Com1}\ \frac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F} \qquad \text{Com2}\ \frac{F \xrightarrow{\alpha} F'}{E \mid F \xrightarrow{\alpha} E \mid F'}$$

La tercera regla de la composición tiene que ver con la sincronización y lleva a la aparición de una acción r. Asumiremos el convenio de que una acción doblemente complementada es igual que la acción misma:

$$\text{Com3}\ \frac{E \xrightarrow{m} E' \quad F \xrightarrow{\overline{m}} F'}{E \mid F \xrightarrow{r} E' \mid F'}$$

La única regla para la restricción dice que si E puede realizar una acción α para llegar a E' entonces E\R puede hacerlo de la misma forma sólo si α no pertenece a R:

$$\text{Res}\ \frac{E \xrightarrow{\alpha} E'}{E\backslash R \xrightarrow{\alpha} E'\backslash R}\ (\alpha, \overline{\alpha} \notin R)$$

Las dos reglas de las constantes dicen que estas tiene el mismo comportamiento que las expresiones que las definen:

$$\text{Con1}\ \frac{E \xrightarrow{\alpha} E'}{A \xrightarrow{\alpha} E'}\ (A=E) \qquad \text{Con2}\ \frac{E \xrightarrow{\alpha} E'}{E \xrightarrow{\alpha} A}\ (A=E')$$

La última regla dice que ocurre cuando las expresiones están entre paréntesis. Hay dos casos dependiendo de la estructura de la expresión emparentizada. Básicamente la composición paralela de dos agentes mantiene los paréntesis después de una transición, mientras que la elección los descarta. La razón es que al evaluar una elección resulta en una simplificación en el sentido de que el + no aparece en el lado derecho de la regla y por tanto los paréntesis dejan de ser necesarios. En el caso de la composición paralela dicha simplificación no ocurre y por tanto se siguen necesitando los paréntesis:

$$\text{Brac1}\ \frac{E \xrightarrow{\alpha} E'}{(E) \xrightarrow{\alpha} E'}\ (E \equiv E1+E2) \qquad \text{Brac2}\ \frac{E \xrightarrow{\alpha} E'}{(E) \xrightarrow{\alpha} (E')}\ (E \equiv E1|E2)$$

Ahora tenemos un conjunto de reglas para cada uno de los operadores de CCS. Tenemos también definido como las constantes y los paréntesis afectan a las transiciones. Las reglas deberían corresponder con la compresión intuitiva de los operadores que desarrollamos en un capítulo previo. Una semántica operacional de este tipo no la única manera de establecer una semántica formal para un lenguaje como CCS, pero la experiencia muestra que es más fácil de aprender y muy práctico.

4.2.- Semántica concurrente

En este capítulo hemos adoptado lo que parece ser una semántica basada en el entrelazamiento de potenciales acciones concurrentes. Las sincronizaciones de los agentes, las cuales generan acciones r, nos dan la única concurrencia en el lenguaje. Aparte de esas sincronizaciones todas las acciones concurrentes observables son entrelazamientos, y por eso decimos que pueden realizarse en cualquier orden y que siguen alcanzando el mismo estado. La razón principal de adoptar esta restricción es que el lenguaje puede ser definido en términos de una semántica operacional muy simple. Naturalmente es de esperar que se tenga que pagar un precio por semejante compromiso de alguna forma, quizás por la perdida de poder expresivo. Ciertamente un lenguaje que busca modelar la concurrencia evitándola levanta muchas sospechas. Retomemos ahora esta decisión y considerémosla más detalladamente desarrollando una verdadera semántica concurrente para CCS considerando las ventajas y desventajas.

Intentaremos acortar el entrelazamiento de acciones concurrentes permitiendo transiciones que consistan en múltiples acciones atómicas. Si consideramos de nuevo el grafo de transición de Buff2 y en particular el nodo del grafo etiquetado por:

$$(in.\underline{m}.C1 \mid \underline{out}.C2)\backslash\{m\}$$

parece razonable que puede haber una transición etiquetada por ambos *in* y *out* que llevase al buffer directamente a :

$$(\underline{m}.C1 \mid C2)\backslash\{m\}$$

Sin duda, no queremos excluir la posibilidad de cualquiera de estas acciones tenga lugar cuando quiera de forma que el resultado sería producir un grafo de transiciones con un arco extra para la doble transición. No podemos sin embargo asumir que representemos ahora las transiciones por conjuntos de acciones atómicas. El agente:

a.E | a.F

demuestra que los conjuntos son inapropiados modelos para nuestras nuevas transiciones debido a que una única {*a*} generaría los tres posibles estados de E|*a*.F, *a*.E|F y E|F aun cuando el último de estos tres fuese generado por una doble transición de dos *a*. Las cosas empeorarían si las transiciones con sincronizaciones están involucradas. Para continuar con esta línea de desarrollo debemos usar bolsas, o multiconjuntos como también se los conoce, para modelar las transiciones concurrentes. Usaremos la notación [ai,... am] para representar las bolsas (donde los índices i,...,m no tienen porque ser distintos). Esencialmente un bolsa en el contexto actual es una función de el conjunto observable de acciones, L, y el conjunto de los números naturales, N. Esto nos llevará a que no necesitemos modelar las sincronizaciones mediante acciones r en nuestro nuevo lenguaje. La bolsa [in, out] será definida por la función {in → 1, out → 1}. En el ejemplo de arriba nosotros ahora tenemos dos posibles transiciones etiquetadas [a] y [a,a], las cuales marcan la distinción requerida. Usaremos la unión, intersección, diferencia y pertenencia seguidas de un + para representar estas mismas operaciones en las bolsas.

Construyamos ahora la semántica operacional para nuestro nuevo lenguaje que podemos llamar CCS+. La nueva regla para el prefijo queda de la siguiente forma:

$$\text{PreP}\ \frac{}{\alpha.E \xrightarrow{\{\alpha\}} E}$$

Simplemente incluimos la acción atómica α en una bolsa que contiene justamente a α. La regla de elección sigue igual con la única diferencia de que las etiquetas de las transiciones son ahora bolsas.

$$\text{SumP1}\ \frac{E \xrightarrow{\sigma} E'}{E+F \xrightarrow{\sigma} E'} \qquad \text{SumP2}\ \frac{F \xrightarrow{\sigma} F'}{E+F \xrightarrow{\sigma} F'}$$

Las dos primeras reglas **Com** también son copiadas directamente:

$$\text{ComP1}\ \frac{E \xrightarrow{\sigma} E'}{E \mid F \xrightarrow{\sigma} E' \mid F} \qquad \text{ComP2}\ \frac{F \xrightarrow{\sigma} F'}{E \mid F \xrightarrow{\sigma} E \mid F'}$$

La tercera regla **Com** requiere bastante más puesto que ahora tenemos múltiples sincronizaciones como resultado de la composición de dos transiciones. En los nuevos cálculos no necesitamos especificar un símbolo que represente a r, las acciones silenciosas como esta se representarán de forma natural por [], la bolsa vacía. La nueva **Com3**, que llamaremos **ComP3**, generará una transición combinada que consistirá en una bolsa de acciones que no contienen pares complementarios. Básicamente la regla dice que dados dos agentes cada uno con una transición concurrente, entonces la transición concurrente resultante de la composición será la unión de las bolsas de sus transiciones individuales menos la intersección de las bolsas. Podemos representar este calculo mediante la siguiente función:

$$T : \text{bag } L \times \text{bag } L \rightarrow \text{bag } L$$
$$T(x,y) = \text{let } z = x \cap\!+ \underline{y} \text{ in } (x \setminus\!+ z) \cup\!+ (y \setminus\!+ \underline{z})$$

Ver que si las transiciones afectadas son bolsas de una única acción como [$\underline{m}$] y [m] entonces el resultado de la transición concurrente será simplemente la bolsa vacía, que como hemos advertido, corresponde con la acción básica r de CCS. En nuestro nuevo lenguaje podemos trabajar con el conjunto L de las acciones atómicas observables y las bolsas de tipo L como etiquetas de las transiciones. La regla resultante para **Com3** es:

$$\text{Com3}\ \frac{E \xrightarrow{\sigma 1} E' \quad F \xrightarrow{\sigma 2} F'}{E|F \xrightarrow{T(\sigma 1,\sigma 2)} E'|F'}$$

Así en nuestro nuevo lenguaje el agente $\underline{a}$.E | $\underline{a}$.F tendrá una única transición $\underline{a}$.E | a.F → E | F, donde la interpretación de este agente usando las reglas operacionales del CCS básico daría lugar a tres posibles transiciones a través de $\underline{a}$, a, y r. Debido al hecho de que todas las posibles sincronizaciones son forzadas por la función T entonces definimos ahora una nueva versión del operador de restricción que especifica aquellas acciones que permitimos que sean observables:

$$\text{Perm}\ \frac{E \xrightarrow{\sigma} E'}{E \setminus L \xrightarrow{\sigma} E' \setminus L}\ (\text{dom } \sigma \subseteq (L \cup \underline{L}))$$

El resto de reglas son directas y solo hace falta adecuarlas al nuevo CCS:

$$\text{ConP1}\ \frac{E \xrightarrow{\sigma} E'}{A \xrightarrow{\sigma} E'}\ (A=E) \qquad \text{ConP2}\ \frac{E \xrightarrow{\sigma} E'}{E \xrightarrow{\sigma} A}\ (A=E')$$

$$\text{BracP1}\ \frac{E \xrightarrow{\sigma} E'}{(E) \xrightarrow{\sigma} E'}\ (E=E1+E2) \qquad \text{BracP2}\ \frac{E \xrightarrow{\sigma} E'}{(E) \xrightarrow{\sigma} (E')}\ (E=E1|E2)$$

Si usamos las reglas de CCS+ para dibujar el grafo de transiciones para Buff2 verás que ahora puedes añadir una nueva transición vía [in,$\underline{out}$]. En realidad todo este esfuerzo extra produce un cálculo con el que es fácil producir grafos de transiciones extremadamente complejos, porque hemos añadido todas la posibles transiciones concurrentes y todas las sub-bolsas de todas las transiciones concurrentes. En otras palabras, si un número de acciones observables están actuando concurrentemente entonces cualquier permutación de ellas puede tener lugar. Prueba a trabajar con el grafo de transición para a.0|b.0|c.0 del principio de este capítulo y lo descubrirás por ti mismo. La pregunta obvia es: ¿Podemos modelar cosas nuevas aparte de los arcos extras en los grafos de transiciones?. En otras palabras, ¿Esta nueva versión de concurrencia nos ofrece verdaderas nuevas transiciones?. En el contexto de CCS esta cuestión lleva a preguntar: Dada una expresión en la sintaxis CCS/CCS+, ¿la nueva interpretación permite transiciones etiquetadas con una bolsa que contiene más de una acción en un grafo tal que las acciones no puede ser entrelazadas separadamente para alcanzar el mismo estado?. La contestación a esta pregunta es simple: No! Esto se puede ver intentando especificar un sistema que ni CCS ni tampoco CCS+ puedan modelar.

Si hacemos una pequeña modificación en la regla del prefijo:

$$\text{PreP}\ \frac{}{\sigma.E \xrightarrow{\sigma} E}$$

Simplemente decimos con esto que los prefijos deben ser ahora bolsas de acciones atómicas. Sin duda podemos tener una bolsa unitaria, la bolsa que contiene una sola acción, y así seguimos modelando todos los sistemas existentes, sólo que ahora hemos alcanzado la sincronización de tres vías que perseguimos.

Bajo CCS*, como llamaremos a este nuevo lenguaje, sólo habrá una transición para el estado inicial. El sistema volverá repetidamente a este estado cuando cada agente complete una serie de acciones.

4.3.- Cálculo síncrono

Ahora hemos excluimos los arcos que no queremos del grafo en los casos donde deseamos forzar una transición concurrente en particular. Haciendo esto hemos quitado efectivamente la conexión entre la concurrencia y el entrelazamiento, pero hemos tenido que hacer un gran esfuerzo para ello. Un caso interesante ocurre si quitamos las dos reglas de **comP1** y **comP2** de la semántica operacional de CCS*. El resultado es una completa versión concurrente de CCS*, que llamaremos SCCS*. Con sólo **comP3**, SCCS* no permite a los agentes esperar a no ser que específicamente se les permita poniendo el prefijo de la bolsa vacía que ahora representaremos por ‘1’. La siguiente definición de la versión para el agente P que puede retrasarse todo el tiempo que quiera antes de realizar su primera acción. Usaremos ‘:’ para denotar el operador de prefijo en SCCS* debido a su naturaleza síncrona:

$$Pi = 1{:}Pi + P$$

CCS, CCS+, CCS* y SCCS* se distinguen principalmente en su semántica, y no es su sintaxis.

4.4.- Tabla resumen

A continuación mostramos una tabla resumen que recoge los lenguajes estudiados en este capítulo:

CCS	Semántica simple de entrelazamientos. Transiciones atómicas. Dos formas de sincronización.
CCS+	Semántica directa. Transiciones concurrentes que puede ser todas entrelazadas. No se asegura múltiples formas de sincronización.
CCS*	Semántica menos intuitiva. La concurrencia no esta definida en términos del entrelazamiento. Puede forzar sincronización múltiple.
SCCS*	Igual que CCS* pero completamente síncrono. Los agentes solo pueden esperar cuando les es permitido. Es un superconjunto de los otros lenguajes.

5.- Paso de parámetros

Hemos visto que CCS básico tiene un uso limitado en especificar sistemas concurrentes en los cuales los datos y sus comunicaciones juegan un papel importante. En este capítulo intentaremos extender CCS básico para que nos permita separar el comportamiento de los datos. Uno de los intencionados beneficios será simplificar las especificaciones de CCS ya que ahora los agentes serán usados para especificar los procesos y no las estructuras de datos, como hasta ahora eran forzados a hacer. El nuevo lenguaje nos permitirá también especificar sistemas de forma más concisa.

Primero veremos la notación adicional para el CCS básico y luego especificaremos la semántica formal. Todos esto resultará en que la semántica de este nuevo lenguaje puede darse en términos de una traducción en CCS básico. Para ilustrar los beneficios de este aparente CCS extendido, trabajaremos con ejemplos que nos son ya familiares, y los compararemos con sus iguales expresados en CCS básicos.

5.1.- Hacer los parámetros explícitos

Un buffer de dos lugares que puede almacenar bits puede ser definido como:

$$\text{Cell1} = in0.\underline{z}.\text{Cell1} + in1.\underline{o}.\text{Cell1}$$
$$\text{Cell2} = \underline{z}.out0.\text{Cell2} + \underline{o}.out1.\text{Cell2}$$
$$\text{Buff22} = (\text{Cell1}|\text{Cell2})\backslash\{z,o\}$$

Donde las acciones $in0$ y $in1$ están para representar la entrada de un cero y un uno respectivamente. Si nosotros intentáramos usar esta técnica para especificar explícitamente un buffer de dos lugares que almacenara números naturales, por ejemplo, entonces obviamente no seríamos capaces de escribir todos los casos. Podemos sin embargo usarlo como la base de una traducción entre el cálculo de paso de parámetros basado en CCS y el CCS básico.

El primer paso es permitir a las acciones tener parámetros que consistirán en variables tipadas. El buffer de dos lugares redefinido para los números naturales sería especificado de la siguiente manera:

$$\text{Cell1} = in(x).\underline{m}(x).\text{Cell1}$$
$$\text{Cell2} = m(x).\underline{out}(x).\text{Cell2}$$
$$\text{Buff2N} = (\text{Cell1}|\text{Cell2})\backslash\{m\}$$

Asumiremos que todas las variables son del tipo de números naturales en este ejemplo.

La especificación es obviamente mucho más simple que la versión previa y más fácil de entender por alguien acostumbrado a los lenguajes de programación. Hay una importante diferencia entre *a*(x) y $\underline{a}$(x). El antecedente liga todas las ocurrencias libres de la variable x en la subexpresión que *a*(x) actúa de prefijo, y por lo tanto actúa como un mecanismo de paso de parámetros. En contraposición la acción $\underline{a}$(x) es simplemente una salida de un valor x y por lo tanto no tiene poder de ligadura. Nosotros sólo consideraremos los casos donde la x esta encerrada por alguna expresión *e* antes de la acción $\underline{a}$(x) sea posible.

No sólo las acciones, sino también los agentes constante pueden tener parámetros. Podemos definir otra versión del buffer de la siguiente manera:

Buff2N(<>)=*in*(x).Buff2N(<x>)
Buff2N(<x>)=*in*(y).Buff2N(<y,x>)+$\underline{out}$(x).Buff2N(<>)
Buff2N(<x,y>)=$\underline{out}$(y).Buff2N(<x>)

De nuevo todas las variables son del tipo de los números naturales y usamos la notación <5> para representar la secuencia que contiene un único valor 5. Advertir que los nombres actuales de las variables usadas con los parámetros son menos importantes que el orden en el que son dadas. En la segunda definición el agente Buff2(<x,y>) es llamado, pero la definición para este caso tiene los nombres de las variables en orden inverso, así si se cheque a la parte derecha de la definición se puede asegurar que el primer valor de la entrada es ciertamente el primer valor de la salida. En una completa especificación formal del buffer deberíamos declarar los tipos de todas las variables usadas, y especificar los tipos abstractos de datos para cualquiera de las estructuras de datos, como la secuencia usada arriba. En este momento nos importa más la forma en que los valores son comunicados que su propia definición o cálculo. Volveremos a considerar estos temas más adelante en este capítulo.

El ámbito de las variables aparecidas en la parte izquierda de la definición de constante se extiende a todas las ocurrencias libres de esa variable en la parte derecha. Cuando la constante es llamada, sus parámetros deberían ser apropiadamente instanciados. Los siguientes puntos son importantes respecto al ámbito de las variables en este nuevo cálculo:

1. Si una variable aparece como parámetro en un agente constante entonces su ámbito es toda la expresión que define esa constante.
2. Las variables que aparecen en acciones de entrada de la forma *in*(x) tiene como ámbito la entera subexpresión que sigue a esta acción.
3. Las variables que aparecen como parámetros de las acciones de salida de la forma $\underline{out}$(x) deben estar siempre dentro del ámbito de alguna acción de entrada o definición constante. En otras palabras los parámetros de las acciones de salida deben recibir siempre un valor antes de que su acción se realice.
4. El ámbito de una variable nunca se extiende más allá de la ecuación en la que aparece. En las definiciones C=*in*(x).C' y C'=$\underline{out}$(x).C el ámbito de x termina con la llamada a C' y la aparición de x en la definición de C' es una variable libre que se mantiene sin encerrar a pesar de cualquier llamada a C'. Sin duda esta x puede ser encerrada si las definiciones formaran parte de una especificación más grande y en el ámbito de una acción de entrada con una variable como parámetro con ese nombre.

También vamos a introducir una sentencia condicional que nos permita hacer comprobaciones en los valores de entrada. Mediante un ejemplo, el siguiente agente recibe dos números naturales y devuelve el primero dividido por el segundo comprobando que el segundo no es cero, en cuyo caso se reportaría un error:

Div=*in*(x).*in*(y).(if y=0 then *error*.Div else *output*(x div y).Div)

Denotar que aparte de una apropiada definición de los números naturales asumimos que las definiciones de los operadores empleados también están apropiadamente definidos. De nuevo lo que debemos hacer es añadir las definiciones de los tipos de datos a nuestra especificación de CCS de paso de parámetros, así como también los mecanismos formales para tipar las variables.

Las extensiones al CCS básico nos proveen de una forma muy natural de especificar sistemas concurrentes que puedan manejar datos y estructuras de datos. Veremos en el siguiente apartado como podemos traducir todo este cálculo ampliado a CCS básico con el beneficio de que toda la semántica vista hasta ahora se mantiene con el paso de parámetros.

5.2.- Cálculo del paso de parámetros

Ahora veremos la sintaxis del cálculo del paso de parámetros en el contexto de un lenguaje que solo tiene un tipo V. Esto puede ampliarse fácilmente a un lenguaje de múltiples tipos tomando V como la unión de todos los tipos requeridos. Aquí *e* es una expresión de tipo V, x es una variable del mismo tipo, y *b* es una expresión booleana en V. Los operadores dados en orden descendiente de precedencia son:

```
ε+  ::=  0
       | ε\R
       | ε+[f]
       | l(x).ε+
       | l(e).ε+
       | a.ε+
       | ε+ | ε+
       | ε+ + ε+
       | if b then ε+
```

Los tres tipos de posibles expresiones prefijo tienen importantes implicaciones en nuestro nuevo lenguaje. En primer lugar, las acciones observables no son forzadas a tener parámetros pero deben ser atómicas como en CCS básico. Estipulamos que ninguna acción pueda ser a la vez parametrizada y lo contrario dentro de la misma especificación. Otra implicación es que las acciones r pueden no ser vistas para pasar valores. Por añadidura cada agente constante A con aridad n tiene una ecuación que lo define de la forma A(a1,....,an)=E donde las únicas variables libres en E son x1,...,xn.

La traducción se basa en la idea de que cada acción en el cálculo de paso de parámetros tiene asociada un conjunto de acciones del cálculo básico. Por ejemplo la acción *in*(x) de nuestro ejemplo del buffer donde x es de tipo de los número naturales, tiene un conjunto correspondiente {*in*x | x Є N} de acciones en el cálculo básico. Acciones con

ninguna variable no necesitan traducción. La traducción viene dada en términos de la función:

$$\|\text{-}\| \; \varepsilon+ \rightarrow \varepsilon$$

tal que por cada expresión E Є ε+ sin variables libres derivamos su forma traducida ||E|| Є ε. La función esta definida para los casos de la estructura de las expresiones de paso de parámetros. Haremos uso de la noción de suma indexada u operador de elección:

$$\Sigma \; Ei \;\; \text{para i Є I}$$

Donde I es el conjunto de índices y la expresión entera es sintácticamente equivalente al agente:

$$E1+...+En \; (\{1,...,n\}=I)$$

La suma indexada es una generalización de las sumas binarias que hemos estado utilizando hasta la fecha y están definidas por la siguiente regla:

$$\text{Sumj} \;\; \frac{Ej \xrightarrow{\alpha} E'j}{\sum_{i \in I} Ej \xrightarrow{\alpha} E'j} \; (j \in I)$$

Usando sumas indexadas podemos ahora definir la función semántica ||-|| ε+ → ε que traduce las expresiones de la sintaxis del paso de parámetros a sus expresiones equivalentes en la sintaxis de CCS básico, para el cual por supuesto tenemos una semántica operacional.

$$\|l(x).E\| = \Sigma_{v \in V} \, lv.\|E\{v\backslash x\}\|$$
$$\|\underline{l}(e).E\| = \underline{l}e.\|E\|$$
$$\|a.E\| = a.\|E\|$$
$$\|E+F\| = \|E\| + \|F\|$$
$$\|E \mid F\| = \|E\| \mid \|F\|$$
$$\|E\backslash R\| = \|E\|\backslash\{lv \mid l \in R \wedge v \in V\}$$
$$\|E[f]\| = \|E\|[\hat{}f] \; (\hat{}f(lv)=f(l)v)$$
$$\|\text{if b then E}\| = \|E\| \text{ if b=true, 0 en otro caso}$$
$$\|P(e1,...,en)\| = Pe1,...,en$$

Esencialmente hemos dado una semántica marcada para CCS en el sentido de que estamos llevando una expresión sintáctica a un objeto matemático que constituye el significado de la expresión. Advertir que sólo hemos considerado las acciones de salida que están encerradas dentro de apropiadas expresiones $\underline{l}(e).E$ y así asegurarnos que no se intenta traducir las variables libres a CCS básico. La ecuación de la restricción debería incluir el caso donde la acción en el conjunto de restricciones esta parametrizada, en cuyo caso no se ve afectada. Advertir también que hay varios posibles casos de terminación para esta función. Si el conjunto de índices en la elección generalizada es el conjunto vacío, entonces no hay Ei y simplemente devolvemos 0. Otro caso de terminación es cuando se encuentra con o sin parámetros con el agente constante. Un último caso de terminación es cuando una expresión booleana en una

condición evalúa a falso. Las ecuaciones para definir constantes serán traducidas de las siguiente manera:

$$\|P(x1,....,xn)=E\| = Pe1,...,en=\|E\{e1/x1,....,en/xn\}\|$$

para todas las posibles instancias e1,....,en de x1,...,xn. En otras palabras, habrá una definición separada de P para cada posible instanciación de sus parámetros.

Mediante un ejemplo redefiniremos el agente genérico Var y le aplicaremos la función de traducción. El agente tiene la siguiente definición:

$$Var=rd0.Var0+rd1.Var1$$
$$Var0=\underline{wrt0}.Var0+rd0.Var0+rd1.Var1$$
$$Var1=\underline{wrt1}.Var1+rd0.Var0+rd1.Var1$$

Podemos generar una versión de paso de parámetros sustancialmente más simple para esta especificación que ya de por si es simple.

$$VarVal=rd(x).VarVal(x)$$
$$VarVal(x)=\underline{wrt}(x).VarVal(x)+rd(y).VarVal(y)$$

Aplicaremos la función de traducción a esta definición de VarVal para ver como la función consigue su objetivo:

$$\|VarVal=rd(x).VarVal(x)\| = VarVal=\|rd(x).VarVal(x)\|$$

Esto nos deja a solo traducir la parte derecha de la definición, que haremos separadamente una vez que sepamos que valores han sido sustituidos por algún parámetro formal; en este caso no hay ninguno:

$$\|rd(x).VarVal(x)\| = rd0.\|VarVal(x)\{0/x\}\| + rd1.\|VarVal(x)\{1/x\}\| =$$
$$= rd0.VarVal0 + rd1.VarVal1$$

Comparando las definiciones para Var y ||VarVal|| debería convencerte de que son virtualmente idénticas en este caso. En la práctica la traducción debe hacerse aplicando primero la función a la definición de más alto nivel y después ir trabajando hacia abajo por los subagentes hasta llegar al final. Esto tiene el efecto de identificar que instancias de los parámetros de los agentes deben ser evaluadas.

6.- Tiempo

A lo largo de este documento sólo nos ha preocupado el orden de los eventos dentro de los sistemas concurrentes. Esto es una parte importante de cualquier sistema en tiempo real porque frecuentemente el tiempo de los eventos puede ser crítico. En este capítulo, como sugiere el título, consideraremos la importancia de hacer el tiempo explícito en las especificaciones. Ciertamente puede sorprender que nos haya llevado tanto tiempo llegar hasta aquí, dada la importancia del tiempo en los sistemas de tiempo real. De todas formas ahora debemos considerar la cuestión de permitir a nuestra especificaciones tener en cuenta el tipo de problemas con que normalmente se enfrentan los ingenieros.

6.1.- Concepto de tiempo

Hay varias formas de añadir el tiempo a las especificaciones. Una forma obvia en la cual el tiempo puede ser modelado es hacer que los agentes se sincronicen con una especie de reloj global. De hecho SCCS usa la idea de que cada proceso en un sistema debe realizar alguna acción cada tic de reloj a menos es este expresamente indicado que puede retrasarse hasta el tic siguiente. De esta forma ni el reloj global, ni el tiempo en si mismo esta modelado específicamente, pero el efecto es exactamente un reloj global que marca el tiempo constantemente y no puede ser ignorado.

Existen otras posibilidades; podríamos por ejemplo adjuntar a cada acción la porción de tiempo que le lleva completarse desde el principio hasta el final. Esto tendría el efecto de recordar el paso del tiempo en términos de la combinación de comportamientos de los subprocesos del sistema. Presumiblemente las acciones complementarias deberían requerir la misma cantidad de tiempo para completarse. Entonces nosotros deberíamos añadir una especie de acciones invisibles que no hiciesen nada excepto esperar durante un periodo de tiempo. Este es un procedimiento similar a lo que ocurre en los lenguajes de programación de tiempo real que requieren que el programador programe bucles que consuman tiempo para retrasar la realización de una instrucción.

Otra forma de introducir el tiempo es coger un lenguaje como CCS y añadirle un tipo de acción que modele la realización de un tiempo de espera especificado. En otras palabras hacemos que los agentes esperen un determinado tiempo antes de que puedan realizar su siguiente acción. La noción de tiempo se basa en la extraña idea de que las acciones atómicas no consumen tiempo. Además de esto existen otras formas de interpretar el retardo, dependiendo de si un proceso simplemente se convierte en disponible pasado un retardo de tiempo, o es forzado a realizar alguna acción en ese determinado momento. Los retardos pueden ser por tanto fuertes o débiles, y esto afectará a la definición semántica de estos lenguajes.

Hay por tanto varias posibilidades de añadir el tiempo a una especificación formal de sistemas en tiempo real. Vamos a estudiar un lenguaje que añade el tiempo a las especificaciones en forma de retardos. A pesar de las cuestiones planteadas en el párrafo anterior, veremos que hay unas cuantas ventajas que se obtienen de esta postura. Primero de todo no debemos cambiar las definiciones existentes de los operadores introducidos anteriormente y que ya han sido comprendidos. Nuestra noción del tiempo será mas o menos encajada dentro de lo que ya tenemos. En segundo lugar muchos de los usos importantes del tiempo en los sistemas de tiempo real son del tipo de esperas. La elección de lo que debe hacerse a continuación depende del tiempo que tenga que esperar un proceso para realizar su siguiente acción. La longitud del tiempo que le lleva a una acción realizarse puede ser modelada definiendo estas dos acciones: aStart y aEnd, que especifican el periodo de tiempo en el cual *a* esta activo y entonces colocan un retardo entre ellos. Por tanto modelar el tiempo en términos de retardos explícitos parece presentar la posibilidad de un lenguaje suficiente expresivo.

6.2.- CCS temporal (TCCS)

Vamos a estudiar un particular estilo de CCS temporal que usa retardos fuertes, en otras palabras retardos que fuerzan alguna acción en el momento en que el retardo expira.

Igual que con el paso de parámetros tomaremos lo sencillos ejemplos vistos anteriormente para ver como se comportan en presencia de los retardos.

Primero de todo consideraremos como los retardos interactuan con la composición paralela, para ello estudiaremos lo que ocurre cuando añadimos retardos al ejemplo de la exclusión mutua, Mx, que hemos visto con anterioridad. En lugar de acciones atómicas c1 y c2 para marcar las secciones críticas, ahora especificaremos puntos de comienzo y fin para cada una de las dos acciones de la siguiente forma:

Mx=(P1|P2|Sem)\{get,put}
NewP1=get.startc1.endc1.put.NewP1
NewP2=get.startc2.endc2.put.NewP2
Sem=get.put.Sem

El uso obvio del tiempo sería señalar cuanto tarda cada sección crítica añadiendo retardos a los procesos NewP1 y NewP2 para hacer clara la longitud de tiempo entre los puntos de inicio y de fin de las dos secciones críticas. Esto nos lleva al siguiente sistema:

NewMx=(TP1|TP2|Sem)\{get,put}
TP1=get.startc1.(5).endc1.put.TP1
TP2=get.startc2.(4).endc2.put.TP2
Sem=get.put.Sem

Esto nos dice ahora que las secciones críticas c1 y c2 tardan 5 ya 4 segundos en completarse respectivamente. Pero introduce un problema en que no hemos dicho lo que a los agentes se les permite hacer cuando están inactivos pero el tiempo pasa para otros agentes. En el ejemplo de NewMx con las nuevas definiciones de TP1 y TP2 el sistema puede evolucionar al siguiente estado realizando una acción r seguida de startc1:

((5).endc1.put.TP1|TP2|put.Sem)\{get,put}

En este estado TP1 tiene 5 segundos de retardo pero los otros dos agentes no tienen ningún retardo y ciertamente están temporalmente interbloqueados puesto que sus siguientes acciones están restringidas. Esto nos lleva a dictar cuales de los estados que un agente compone en paralelo pueden retrasarse solo si todos pueden retrasarse por el mismo periodo de tiempo. Hay dos posibilidades de cumplir este requerimiento: permitir que un agente sin un retardo especificado puede esperar todo el tiempo que quiera, o alternativamente no permitir que los agentes hagan un retardo a no ser que haya un retardo específico explícitamente. Si adoptamos la primera solución entonces puede ser difícil en algunas situaciones forzar una acción a realizarse por encima de un retardo no especificado. Si nos decantamos por la segunda solución entonces el sistema esta interbloqueado en ambos términos de tiempo y acción, lo que parece no ser un asunto deseable. A pesar de esto nuestra primera versión de CCS temporal adopta esta segunda aproximación.

Podemos intentar eliminar el interbloqueo temporal dando al semáforo un retardo de 5 segundos, que es correcto si esta siendo controlado por TP1 pero no si esta siendo controlado por TP2 que solo tiene 4 segundos de retardo. Otro interbloqueo ocurrirá y quizás deberíamos tener una elección de retardos para el semáforo. El problema para los

otros dos procesos es todavía peor debido a que el no determinismo en el sistema significa que TP2, por ejemplo, puede tener un retardo largo antes de que llegue a estar activo, o incluso puede que nunca llegue a estar activo. Introduciremos la noción de retardo arbitrario que especifica que un agente esta en un estado de espera. Sintácticamente identificaremos a ese agente poniéndolo en negrita y significará que ese agente puede retardarse indefinidamente. Advertir que para este documento utilizamos la notación de salida mediante el subrayado, y esta nueva función en negrita; a la hora de especificar sobre el papel, la notación recomendable es la salida indicarla mediante un superrayado o barra superior, y la nueva función mediante un subrayado. Volviendo al tema, el único modo que tiene un agente de liberarse de este estado de espera es realizar alguna acción. Usando esta nueva idea podemos completar la versión temporal de Mx:

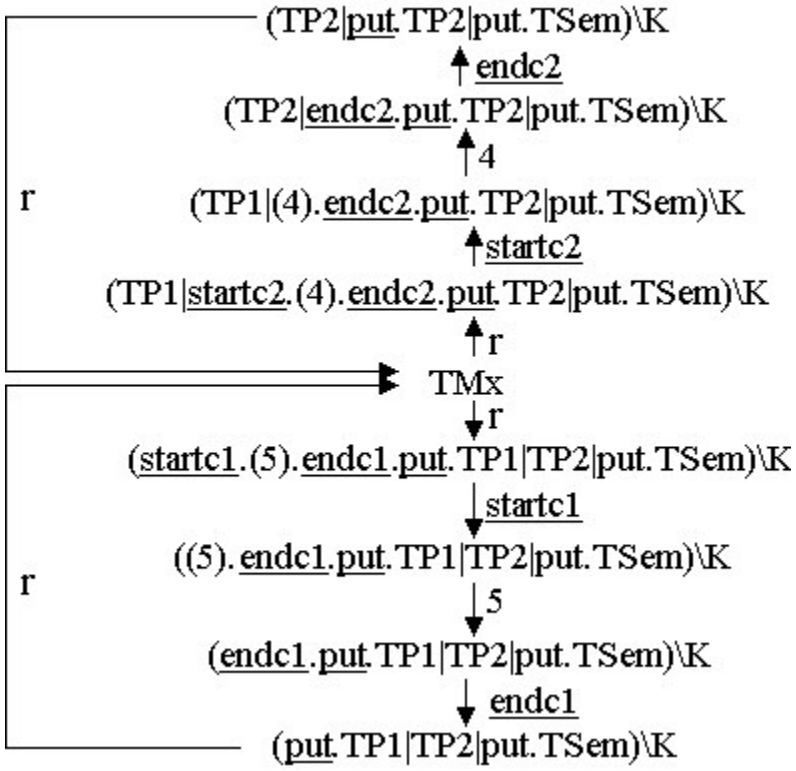
TMx=(TP1|TP2|Tsem)\{get,put}
TP1=**get**.startc1.(5).endc1.put.TP1
TP2=**get**.startc2.(4).endc2.put.TP2
Tsem=get.**put**.Tsem

Notar que Tsem puede solo retardarse cuando esta ocupado y no cuando esta libre. El resultado de esto es que no se permite ningún retardo entre las secciones críticas activas: o bien c1 o bien c2 deben progresar. Añadiendo un estado de espera al semáforo mediante:

IdleSem=**get**.**put**.IdleSem

permitimos la posibilidad de infinitos retardos entre las secciones críticas activas.

Igual que con el CCS básico podemos dibujar el grafo de transición para mostrar los posibles comportamientos de la especificación. El grafo se muestra a continuación, y fijarse que las flechas indican tanto acciones como una transición de retardo. Haremos distinción entre estas dos transiciones en la semántica formal.


(TP2|put.TP2|put.TSem)\K
endc2
(TP2|endc2.put.TP2|put.TSem)\K
4
r
(TP1|(4).endc2.put.TP2|put.TSem)\K
startc2
(TP1|startc2.(4).endc2.put.TP2|put.TSem)\K
r
TMx
r
(startc1.(5).endc1.put.TP1|TP2|put.TSem)\K
startc1
((5).endc1.put.TP1|TP2|put.TSem)\K
r
5
(endc1.put.TP1|TP2|put.TSem)\K
endc1
(put.TP1|TP2|put.TSem)\K

7.- Ejemplo del ascensor en CCS con parámetros

Diseño para el comportamiento de un ascensor descrito en CCS. Este ascensor se mueve de una manera racional, manteniéndose subiendo si hay peticiones por encima de la planta en la que está el ascensor o manteniéndose bajando si estaba bajando y quedan peticiones por debajo. Este ascensor tiene capacidad infinita y es posible que cuando esté moviéndose no atienda algunas peticiones hasta que no se detenga.
Para realizar este diseño se ha empleado CCS con paso de parámetros además del (if then else) instrucción que se puede traducir de forma sencilla a CCS normal.

Explicación de los estados considerados, así como de las diferentes acciones que se pueden realizar.

Especificaciones implementación en CCS del ascensor con capacidad infinita

- **BOTON DE LA PLANTA X**

Descripción:

Este proceso se encarga de simular el comportamiento de los botones de las diferentes plantas (botón de llamada al ascendor, están fuera del ascensor), el parámetro 'x' indica el número de planta en donde se encuentra el botón, habrá uno por planta.

BotonPlantaOff[x] = pulsaP[x].'request[x].llamada[x].BotonPlantaOn[x]

Descripción:

Este es el estado en el que se encuentra el ascensor cuando ningún usuario ha pulsado aún el botón de llamada.

Acciones:

1.- Si alguien pulsa el botón se envía una señal request al control de peticiones y se pone el botón como pulsado, actualizando la situación de peticiones mínima y máxima que debe atender el ascensor.

BotonPlantaOn[x] = pulsaP[x].BotonPlantaOn[x] + resetP[x].BotonPlantaOff[x]

Descripción:

Estado en el que se encuentra el ascensor cuando el botón ya ha sido pulsado y aun no se ha atendido la petición.

Acciones:

1.- Si alguien pulsa el botón, no se realiza ninguna acción, el estado no varía.

2.- Si recibimos la señal resetP[x] mandada por el control del ascensor después de haber atendido la petición de la planta, se deja el botón como apagado.

- **BOTONES DEL ASCENSOR PARA IR A X**

Descripción:

Este proceso es exactamente igual al anterior sólo que se encarga de atender las pulsaciones de los botones del panel interior del ascensor. Los estados y acciones son completamente equivalentes.

BotonAscensorOff[x] = pulsaA[x].'request[x].llamada[x].BotonAscensorOn[x]
BotonAscensorOn[x] = pulsaA[x].BotonAscensorOn[x] + resetA[x].BotonAscensorOff[x]

- **CONTROL DEL ASCENSOR**

Descripción:

Proceso encargado del control del ascensor, está dividido en varios estados y cada uno de ellos prepara al ascensor para llevar a cabo una serie de acciones.

En ControlX[x,y] la información que se almacena es la siguiente:

X puede tomar A,V,O. A determina arriba, o sea, que el ascensor tiende a subir, V tiende a bajar y O no da prioridad a ninguna elige la primera petición que reciba.

[x,y] guardan información de la mayor y la menor petición para saber si el ascensor debe seguir subiendo o bajando, o debe permanecer parado.

ControlA[x,y] = (‘planta?.en[z].’peticion?[z].(peticionOn.AbrirA[x,y,z] + peticionOff.if(x>z)then Subir[x,y] else ControlO[minimo,y]) + llamada[z].ActualizarA[x,y,z]

Descripción:

Este estado es en el que se encuentra el ascensor cuando está en una planta parado y lo último que hizo fue subir, con lo que se da prioridad a las peticiones que estén por encima de la planta en la que se encuentre el ascensor.

Acciones:

1.- Se pregunta al ascensor en que planta está y devuelve la planta z. Se pregunta si hay alguna petición en z y en caso de haberla se pasa al estado AbrirA.

AbrirA guarda con ControlA la información referida a la tendencia del ascensor a subir para que ésta no se pierda. Si no había petición entonces, si quedase alguna petición por encima de z, es decir x > z, se sube; sino se deja al ascensor parado y se quita la prioridad de subir y se deja para que atienda a la primera que reciba. Es necesario actualizar x al mínimo siendo mínimo el número más bajo de planta, para que no se tenga en cuenta este valor y se actualice correctamente si llega una petición.

2.- Si se recibe una llamada para una planta z, entonces se actualiza los valores x,y.

ControlV[x,y] = (‘planta?.en[z].’peticion?[z].(peticionOn.AbrirV[x,y,z] + peticionOff.if(y<z)then Bajar[x,y] else ControlO[x,maximo]) + llamada[z].ActualizarV[x,y,z]

Descripción:

Este estado es completamente equivalente al anterior sólo que en este caso, la prioridad es de bajada en vez de subida.

ControlO[x,y] = ‘planta?.en[z].’peticion?[z].(peticionOn.AbrirO[x,y,z] + peticionOff.if(x>z)then Subir[x,y] else if(y<z)then Bajar[x,y] else llamada[z].ActualizarO[x,y,z]

Descripción:

Este estado es en el que se encuentra el ascensor cuado ha atendido todas las peticiones según el movimiento que llevaba, es posible que, en general, no le queden peticiones. Lo que hace es ver si tiene que atender alguna petición y si no, se queda bloqueado esperando una señal del tipo llamada.

Acciones:

1.- Se consulta al ascensor por la planta en la que se encuentra. Se mira si hay alguna petición para esta planta y si la hay se pasa a AbrirO; sino, si quedan peticiones por arriba se sube, sino, si quedan peticiones se baja y sino, se espera a que llegue una llamada.

Estado auxiliares del control

Descripción:

Estos son los estados que se habían dejado sin definir en los estados anteriores

Subir[x,y] = ‘subir.ControlA[x,y]

Descripción:

Este estado simplemente hace subir una planta al ascensor.

Acciones:

1.- Manda una señal al ascensor para que suba una planta y pone el control con preferencia para la subida.

Bajar[x,y] = ‘bajar.ControlV[x,y]

Descripción:

Este estado simplemente hace bajar una planta al ascensor.

Acciones:

1.- Manda una señal al ascensor para que baje una planta y pone el control con preferencia para la bajada.

AbrirA[x,y,z] = ‘abrir.’vaciarAscensor[z].ascensorVacio[z].’vaciarPlanta[z].plantaVacia[z].’cerrar .ControlA[x,y]

Descripción:

Este estado implementa el protocolo de salida y entrada de usuarios que es sencillamente un hand-shaking con las colas del ascensor y la planta. Este se asegura de que todos los usuario del ascensor que tenían que salir en esta planta lo hagan, y que entren al ascensor todos los que lo estaban esperado en esta planta.

Accion:

1.- Se manda al ascensor una señal para que abra las puertas. Se indica a los usuarios del ascensor que quieran salir en la planta ‘z’ que pueden salir; se espera a la respuesta de éstos de que lo han hecho, entonces se da permiso para que entren a aquéllos que estaban esperando el ascensor, esperando a que no quede nadie. Por último, se cierra éste y se vuelve al estado de decisión del control; este estado recuerda que cuando vuelva a Control su estado será ControlA, es decir, con prioridad a la subida.

AbrirV[x,y,z] = ‘abrir.’vaciarAscensor[z].ascensorVacio[z].’vaciarPlanta[z].plantaVacia[z].’cerrar .ControlV[x,y]

Descripción:

Este estado es equivalente a anterior y al siguiente con la única diferencia de que da prioridad a la bajada, y cuando devuelve el control pone prioridad a la bajada.

AbrirO[x,y,z] = ‘abrir.’vaciarAscensor[z].ascensorVacio[z].’vaciarPlanta[z].plantaVacia[z].’cerrar .ControlO[x,y]

Descripción

Este estado es equivalente a las dos anteriores con la única diferencia de que cuando devuelva el control estará indeciso y atenderá cualquier petición tanto si fuera de bajada como si fuera de subida

ActualizarA[x,y,z] = ‘planta?.en[p].if(z>p)^(z>x)then ControlA[z,y] else if(z<p)^(z<y)then ControlA[x,z] else ControlA[x,y]

Descripción:

Cuando el control recibe una petición para una planta es posible que tenga que actualizar los limites max-min, es decir x,y. Finalmente, devuelve el control con prioridad de subida.

Acciones:

1.- Se consulta al ascensor en la planta en la que está y contesta ‘z’.

1.2.- Si la llamada está por encima de la planta del ascensor y es mayor que ‘x’, se actualiza x con el valor de ‘z’.

1.3.- Si la llamada está por debajo de la planta del ascensor y es menor que ‘y’, entonces se actualiza ‘y’ con el valor de ‘z’.

ActualizarV[x,y,z] = ‘planta?.en[p].if(z>p)^(z>x)then ControlV[z,y] else if(z<p)^(z<y)then ControlV[x,z] else ControlV[x,y]
ActualizarO[x,y,z] = ‘planta?.en[p].if(z>p)^(z>x)then ControlO[z,y] else if(z<p)^(z<y)then ControlO[x,z] else ControlO[x,y]

Descripción:

Estos dos estados son iguales que el anterior y sólo se diferencian en el estado en el que dejan al control al terminar la actualización.

- **ControlPlanta de cada piso**

Descripción:

El control de planta recuerda el hecho de que hay una petición para esta planta, es decir, se comporta como un flag para indicar que al pasar por esa planta debe pararse. Recibe peticiones tanto de los botones de la planta como de los botones del ascensor, y los trata por igual. Cuando la petición está activada da igual que lleguen otras peticiones, ya que las acepta y no hace nada más.

ControlPlantaOff[x] = (peticion?[x].’peticionOff + reset[x]).ControlPlantaOff[x] + request[x].ControlPlantaOn[x]

Descripción:

Este estado es en el que se encuentra el control de planta cuando no ha recibido ninguna petición.

Acción:

1.- Se recibe una señal que consulta por la existencia de peticiones en esta planta, y se devuelve que no hay petición por medio de la señal peticionOff’. Por último, se deja al proceso como estaba, es decir, en Off

2.- Se recibe una señal ‘reset’ para borrar las petición de esta planta, dejando al sistema l como estaba, o sea, en Off.

3.- Se recibe una petición ‘request’ y se ponemos el estado actual de la planta en On.

ControlPlantaOn[x] = (peticion?[x].’peticionOn + request[x]).ControlPlantaOn[x] + reset[x].ControlPlantaOff[x]

Descripción: Este estado es en el que se encuentra el control de planta cuando ya ha recibido ninguna petición.

1.- Se recibe una señal que hace las veces de pregunta para ver si hay peticiones en esta planta, devolviendo una respuesta afirmativa por medio de la señal 'peticionOn' y se deja al proceso en el mismo estado en que se encontraba, es decir, en On.

2.- Se recibe una petición 'request' dejando el estado de la planta en On.

3.- Se recibe una señal 'reset' para borrar las peticiones de esta planta, y se cambia el estado a Off.

- **ColaPlanta almacena en 'x' la planta y en 'y' el número de usuarios que hay esperando en la planta.**

ColaPlantaVacia[x] = vaciarPlanta[x].'plantaVacia.ColaPlantaVacia[x] + llegaCliente[x].ColaPlanta[x,1]

Descripción:

Este estado informa que no hay nadie esperando en la planta para entrar al ascensor.

Acciones:

1.- Se recibe una señal para vaciar la planta (para entrar en el ascensor), e inmediatamente se devuelve que la planta ya esta vacía y se permanece en el mismo estado.

2.- Llega un cliente a la cola y se pasa al estado ColaPlanta[x,1] indicando que hay un usuario esperando.

ColaPlanta[x,y] = vaciarPlanta[x].VaciandoPlanta[x,y] + llegaCliente[x].ColaPlanta[x,y+1]

Descripción:

Se está en este estado cuando hay 'y' clientes esperando en la planta 'x' para entrar en el ascensor.

Acciones:

1.- Se recibe una señal vaciarPlanta y se pasa al estado VaciandoPlanta.

2.- Llega un cliente a la cola y se incrementa en una unidad el número de clientes que están esperando.

VaciandoPlanta[x,y] = (if(y=0) then 'plantaVacia.ColaPlantaVacia[x] else 'entraCliente[x].VaciandoPlanta[x,y-1] + llegaCliente[x].VaciandoPlanta[x,y+1])

Descripción:

Este estado simula el momento en el cuál los clientes están entrando en el ascensor, esperando a que no quede ninguno.

Acciones:

1.- Si no quedan clientes, se devuelve plantaVacia y se pasa al estado plantaVacia.

2.- Si no, hay dos opciones:

a) Se manda una señal a un cliente para que entre en el ascensor y se decrementa 'y'.
b) Llega un cliente mientras están entrando los usuarios en el ascensor y se incrementa 'y'.

- **Cola del Ascensor**

Descripción:

Este proceso es completamente equivalente al del la cola de planta.

ColaAscensorVacia[x] =
vaciarAscensor[x].'AscensorVacio[x].ColaAscensorVacia[x] +
entraAscensor[x].ColaAscensor[x,1]

ColaAscensor[x,y] = vaciarAscensor[x].VaciandoAscensor[x,y] +
entraAscensor[x].ColaAscensor[x,y+1]

VaciandoAscesor[x,y] = (if(y=0) then 'AscensorVacio.ColaAscensorVacia[x] else
'saleCliente[x].VaciandoAscensor[x,y-1] +
entraAscensor[x].VaciandoAscensor[x,y+1])

- **Ascensor[x]**

Descripción:
Este proceso se encarga de simular la cabina del ascensor y guarda información en 'x' de la planta en la que se encuentra.

Ascensor[x] = subir.Ascensor[x+1] + bajar.Ascensor[x-1] +
planta?.'en[x].Ascensor[x] + abrir.cerrar.resetP[x].resetA[x]Ascensor[x]

Acciones:
1.- Si se recibe la señal de subir se sube una planta
2.- Si se rece la señal de bajar se baja una planta.
3.- Si se recibe una consulta para averiguar la planta, se contesta la señal en[x] siendo 'x' el número de planta y se regresa al estado original.
4.- Si se recibe la señal de apertura de puertas, se espera a que se puedan cerrar y se lleva a cabo las siguientes acciones: Se resetean las peticiones de la planta, se resetean las peticiones del ascensor y se retorna al estado original.

- **El proceso Usuario básicamente sigue una acción lineal yendo de la planta 'x' a la planta 'y'.**

Usuario[x,y] =
'llegaCliente[x].'pulsaP[x].vaciarPlanta[x].entraAscensor[x].'pulsaA[y].saleCliente[y].0

Lo que hace el usuario se describe a continuación:
El usuario llega a una planta y pulsa el botón de la planta para llamar al ascensor. Espera a que le den permiso para entrar en el ascensor(vaciarPlanta), entra en el ascensor, pulsa el botón del ascensor, aguarda a que le concedan el permiso para salir en la planta que desea(vaciarAscensor) y acaba.

Otra opción para el proceso Usuario que se ajusta más al resto de estados sería uno que pulsara o no pulsara los botones. Esto supondría que tendría que confiar en que alguien lo hiciera y podría quedar ignorado por el ascensor. El proceso sería así.

Usuario[x,y] = llegaCliente[x].(pulsaP[x].vaciarPlanta[x]+ vaciarPlanta[x])
.entraAscensor[x].(pulsaA[y].vaciarAscensor[y] + vaciarAscensor[y]) .0

7.1.- Ejemplo de simulación del ascensor

Supongamos que hay un usuario que está en la planta baja y que desea ir a la planta 2. El ascensor está parado en la planta 1 y no tiene que atender peticiones de otros usuarios, es decir, está ocioso.

Tendremos las siguientes señales:
Usuario02 = 'llegaCliente0.'pulsaP0.entraCliente0.'entraAscensor0.'pulsaA2. saleCliente 2.nil
'llegaCliente0 → Se activa ColaPlantaVacia0 = llegaCliente0.ColaPlanta01
Se añadirá a la cola de peticiones de la planta 0 al usuario. Se "consume" la señal llegaCliente0 y se puede lanzar el proceso ColaPlanta01.

Usuario02 = 'pulsaP0.entraCliente0.'entraAscensor0.'pulsaA2.saleCliente2.nil
'pulsaP0 → Activación del proceso BotonPlantaOff0 = pulsaP0.'request0.'llamada0.BotonPlantaOn0
El usuario ha pulsado el botón para llamar al ascensor activando dicho proceso.

'request → Se activa el control para la planta que tiene una petición de uso del ascensor
ControlPlantaOff0 = request0.'llamada0.ControlPlantaOn0
Se consume la señal request0 y se puede lanzar el proceso ControlPlantaOn0.

El ascensor era un proceso en paralelo a los procesos "persona" que podían hacer uso del ascensor. Se continúa ahora viendo cuál es el comportamiento del ascensor.
El proceso ControlPlantaOff0 se comunica con el controlador para que actualice el estado de las peticiones que ha de atender a través de la señal 'llamada0.
'llamada0 → ControlO02 = llamada0.Actualizar020

El control se comunica con el ascensor y viceversa mediante el envío de señales. Los caracteres (...) utilizados representan el resto de señales del proceso.
ControlO02 comienza teniendo este valor, ya que los parámetros 'x' e 'y' de este proceso almacenan las peticiones máxima y mínima respectivamente, y al inicializar deben tener los valores opuestos para garantizar un correcto funcionamiento del ascensor.

El control llama al proceso Actualizar

ActualizarO020 = 'planta.(...)

Este proceso se comunica con el ascensor para determinar la planta en la que está. Se comunican por medio de la señal 'planta, y aquél le responde a través de la señal enX. Recordamos que el ascensor estaba en planta 1.
'planta→ Ascensor1 = planta.'en1.Ascensor1
El ascensor le responde con la señal 'en1 y permanece en el mismo estado en que estaba.
'en1 → ActualizarO020 = en1.ControlO00.
El proceso actualizar consume la señal en1 y hace que ControlO pase del estado 02 a 00.

Una vez producido este cambio de estado, ControlO vuelve a comunicarse con el ascensor para atender la nueva solicitud.

ControlO00 = ‘planta.en.(...)
‘planta → Ascensor1 = planta.’en1.ascensor1
El ascensor responde al control dándole el número de planta en el que se encuentra y regresa a su estado anterior. Consume la señal planta.
‘en1 → ControlO00 = en1.’peticion1.(...)
Ahora el control se comunica con el control de la planta para determinar si hay alguna petición de la planta que deba ser atendida.
‘peticion1 → ControlPlantaOff1 = peticion1.’peticionOff.ControlPlantaOff1
El proceso le responde tras consumir la señal de entrada. La respuesta es negativa dado que no ha habido petición procedente de esta planta. Una vez ha respondido vuelve a su estado actual.

’peticionOff →ControlO02 = Bajar00. Una vez que ha consumido la señal que le mandó el proceso controlador de la planta, el ascensor baja ya que hay una petición en una planta inferior a la que se encuentra el ascensor y no hay ninguna por encima.

Bajar00 = ‘bajar.ControlV02. El ascensor lanza la señal de bajada y predispone al control para dar prioridad a las solicitudes para el uso del ascensor que sean de bajada. Esta señal comunica al ascensor, o sea, la cabina que baje un piso. Se produce comunicación entre este proceso y el ascensor.

Bajar → Ascensor1 = bajar.Ascensor0. El ascensor consume la señal y baja un piso, actualizando la posición de su estado decrementando una posición.

Al bajar una planta, el control de la planta se comunica con el ascensor preguntándole la planta, y a su vez, se comunica con el control de la planta para saber si hay en ésta una petición de uso del mismo.

ControlV02 = ‘planta.(...)
‘planta → Ascensor0 = planta.’en0.Ascensor0. Recibe la señal, le responde y regresa a su estado actual.
‘en0 → ControlV02 = en0.’peticion0.(...)
Se comunica ahora con el proceso controlador de la planta.
’peticion0 → ControlPlantaOn0 = peticion0.’peticionOn.ControlPlantaOn0. El proceso le comunica al controlV que hay una petición, la del usuario y vuelve a su estado.
‘peticionOn→ ControlV00 = peticionOn.Abrir000. Como consecuencia de esta petición, se abre la puerta del ascensor.

A partir de este momento se producirá la comunicación entre el proceso de apertura y la cabina para que se abran las puertas y entre el usuario. Se comunicará a los usuarios que quieran bajar en esta planta que lo hagan; como el ascensor está vacío no hay ninguno que lo hará, y el usuario de fuera podrá entrar. Para ello, se llevarán a cabo comunicaciones con procesos que representan la cola de personas que desean acceder al ascensor de la planta, y los que quieren salir del mismo en esa planta.

Abrir000=‘abrir.’vaciarAscensor0.ascensorVacio0.’vaciarPlanta0.plantaVacia0.’cerrar.ControlV00

‘abrir → Ascensor0 = abrir.cerrar.(...)
‘vaciarAscensor0 → ColaAscensorVacia0 =
vaciarAscensor0.’ascensorVacio.ColaAscensorVacia0
La cola recibe la señal, responde y regresa al estado en el que estaba.
’ascensorVacio → Abrir000 =
ascensorVacio0.’vaciarPlanta0.plantaVacia0.’cerrar.ControlV00

Ahora se efectúa la comunicación con el proceso colaPlanta01, activado con anterioridad.
‘vaciarPlanta0 → ColaPlanta01 = vaciarPlanta0.VaciandoPlanta01.
Una vez que recibe la señal de que todas las personas que deseaban bajar en esta planta lo han hecho, en este caso ninguna, se vacía la cola de personas que deseaban entrar en el ascensor, es decir, se vacía la planta.

VaciandoPlanta01 = ‘entraCliente0.VaciandoPlanta00. Comunica al usuario que puede entrar a la cabina, y actualiza su estado reduciendo el tamaño de personas que esperan de uno a cero.

VaciandoPlanta00 = ‘plantaVacia.ColaPlantaVacia0
La señal ‘entraCliente es consumida por el usuario
‘entraCliente0 → Usuario02 = entraCliente0.’entraAscensor0.’pulsaA2.
vaciarAscensor2.nil
Por otra parte, Abrir000 continúa su ejecución al recibir la señal plantaVacia de VaciandoPlanta00 que le indica que ya se pueden cerrar las puertas de la cabina, estableciendo para ello una comunicación con el proceso ascensor.

‘plantaVacia → Abrir000 = plantaVacia0.’cerrar.ControlV00
Ahora avisa al ascensor de que cierre sus puertas.
‘cerrar → Ascensor0 = cerrar.’resetP0.’resetA0.Ascensor0
Con las señales de reset se borran las peticiones que parar el ascensor en esta planta para bajar de él o subir a él.
‘resetP0 → BotonPlantaOn0 = resetP0.BotonPlantaOff0. Se deshabilita al botón de la planta 0, para la gente que quería usar el ascensor, ya que ya lo ha hecho.
‘resetA0 no tendrá ninguna acción ya que no ha había ninguna petición para bajar en esta planta procedente del interior del ascensor.

Ahora el proceso Usuario02 informa a la cola de usuario que han entrado al ascensor, la presencia de un usuario del ascensor, y ésta actualiza su estado.
Usuario02 = ’entraAscensor0.’pulsaA2. saleCliente2.nil
‘entraAscensor0 → ColaAscensorVacia0 = entraAscensor.ColaAscensor01

Con la pulsación del usuario del destino al que quiere ir, se produce una comunicación entre el ascensor y el proceso que controla los botones interiores del mismo.
‘pulsaA2 → BotonAscensorOff2 = pulsaA2.’request2.’llamada2.BotonAscensor2
La señal de request activa el proceso controlador de la planta destino
‘request → ControlPlantaOff2 = request.ControlPlantaOn2
mientras que la señal de ‘llamada2 actualiza el valor de las peticiones para el proceso controlador
‘llamada2 → ControlV00 = llamada2.ActualizarO002

El proceso Actualizar se comporta como sigue:
ActualizarO002 = ‘planta.(...)
Análogamente al caso de la comunicación en la primera solicitud se efectúa una comunicación con el ascensor para determinar la planta. El comportamiento es igual al caso anterior y lo omitimos.
‘en0 → Actualizar = en0.ControlO20
Como consecuencia de la nueva petición, el estado del controlador evoluciona de ControlO00 a ControlA20.
A partir de ahora, dado que el comportamiento es prácticamente análogo se hará una descripción más somera, sin entrar en tantos detalles de las señales.
El control se comunicará con el ascensor para determinar su planta a través de las señales planta y enX, como ya se ha visto. A continuación, se comunicará con el proceso controlador de planta (ControlPlanta), como salvo para el caso de la planta dos, todos los controladores de las plantas intermedias estarán apagados y su respuesta a la señal peticiónX será peticionOff.

Como consecuencia de las condiciones de la estructura de selección el piso subirá una planta, pasando al estado Subir20.
Este estado se comunica con el proceso Ascensor a través de la señal ‘subir, que hará que éste ascienda un piso, y que el estado del proceso controlador cambie su tendencia hacia arriba.
Subir20 = subir.ControlA20
‘subir → Ascensor0 = subir.Ascensor1

ControlA20 ahora se comunicará con el proceso controlador de la planta uno para ver si hay peticiones de usuarios que quieren bajarse o que hayan pulsado el botón de llamada fuera del ascensor.
ControlA20 = ‘planta.en1.’peticion1.peticionOff.Subir20

Lo que sucede ahora es análogo a lo que pasaba unas líneas antes, 8 exactamente, con lo que evoluciona Ascensor1 a Ascensor2.

De nuevo, se ejecuta el proceso ControlA20, sólo que ahora sí recibirá una petición de detenerse en la planta dos, que era la de destino del usuario del ascensor.
ControlA20 = ‘planta.en2.’peticion2.(...)
Aquí se repite lo mismo que antes, comunicación con el ascensor por medio de la señal ‘planta, éste responde con la señal en2. Posteriormente se comunica con el proceso ControlPlanta2
‘peticion2 → ControlPlanta2 = peticion2.’peticionOn
‘peticionOn →ControlA20 = peticionOn.AbrirA202

El proceso Abrir se comunicará con el ascensor para que abra las puertas, con las colas del ascensor de usuarios que están en el ascensor y desean bajar, al igual que con los usuarios que subieron para que bajen.

AbrirA202 =
abrir.’vaciarAscensor2.ascensorVacio2.’vaciarPlanta2.plantaVacia2.’cerrar
.ControlA20

‘abrir → Ascensor2 = abrir.cerrar.’resetP2.’resetA2.Ascensor2

Con dicha señal el ascensor abre sus puertas para entren o salgan los usuarios.

‘vaciarAscensor2 → ColaAscensor21 = vaciarAscensor2.VaciandoAscensor21
Una vez recibida la señal vaciarAscensor2 se llama al proceso VaciandoAscensor21, el cual, vaciará la cola con los usuarios presentes en el ascensor que deseen bajarse en la planta en cuestión.

VaciandoAscensor21 = ‘saleCliente2.VaciandoAscensor20
VaciandoAscensor20 = ‘ascensorVacio.ColaAscensorVacia2
‘saleCliente hace que salga el cliente del ascensor. Es una señal que captura el proceso Usuario02.

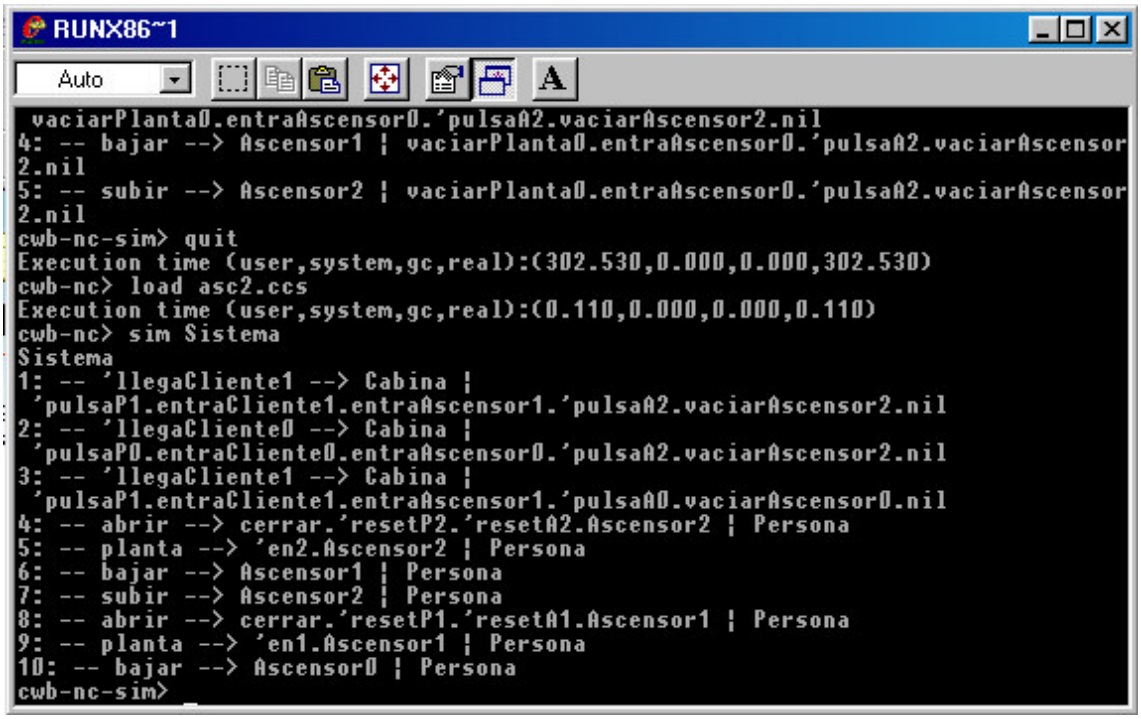
‘saleCliente → Usuario02 = saleCliente.nil
El Usaurio02 sale del ascensor y el proceso Usuario acaba.

Lanza ‘ascensorVacio → Abrir202 =
ascensorVacio2.’vaciarPlanta2.plantaVacia2.’cerrar .ControlA20
Una vez ha consumido dicha señal, se comunica a través de la señal ‘vaciarPlanta2, señal que recibe ColaPlantaVacia.

‘vaciarPlanta2 →ColaPlantaVacia2 = vaciarPlanta2.’plantaVacia.ColaPlantaVacia2
Este proceso se comunica con el anterior devolviéndole la señal ‘plantaVacia, que le indica que no hay nadie esperando a entrar en el ascensor en dicha planta
‘plantaVacia → Abrir202 = plantaVacia2.’cerrar .ControlA20
Abrir recibe la señal y manda al ascensor que cierre sus puertas.
‘cerrar →Ascensor2 = cerrar. ’resetP2.’resetA2.Ascensor2

Por último, manda a los botones que reseteen las peticiones de pulsación que estos recibieran y permanece al estado Ascensor2.
‘resetP2 → No activa el proceso ya que estaba apagado
‘resetA2 → BotonAscensorOn2 = resetA2.BotonAscensorOff2

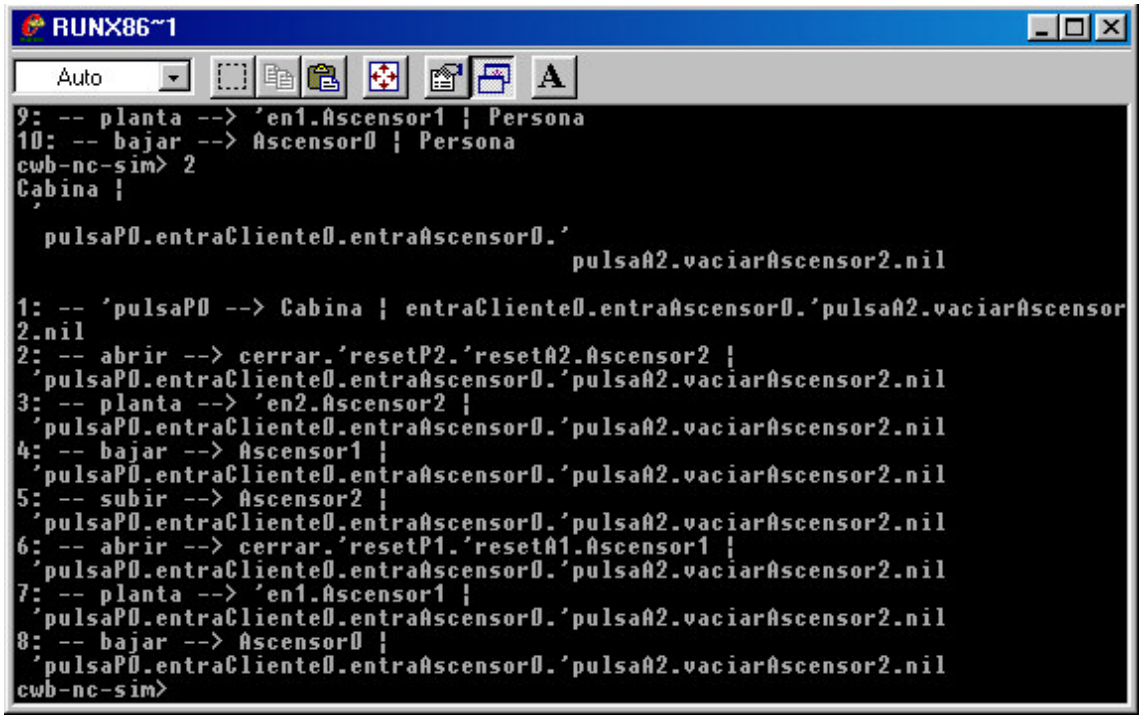
A continuación, se muestra la simulación hecha con Concurrency Work Bench de North Carolina, junto con algunas capturas de pantalla, que muestran el estado en un momento de la ejecución concreto.
En la ejecución se han simulado los dos procesos más importantes, el ascensor (la cabina) junto a unos usuarios. No se han simulado todos los procesos debido al número de los mismos, y al gran volumen de señales que participan en el sistema, las cuáles hubiesen impedido la captura de imágenes suficientemente explicativas.
Para la generación de los valores se ha utilizado el conversor de CCS con parámetros a CCS básico.



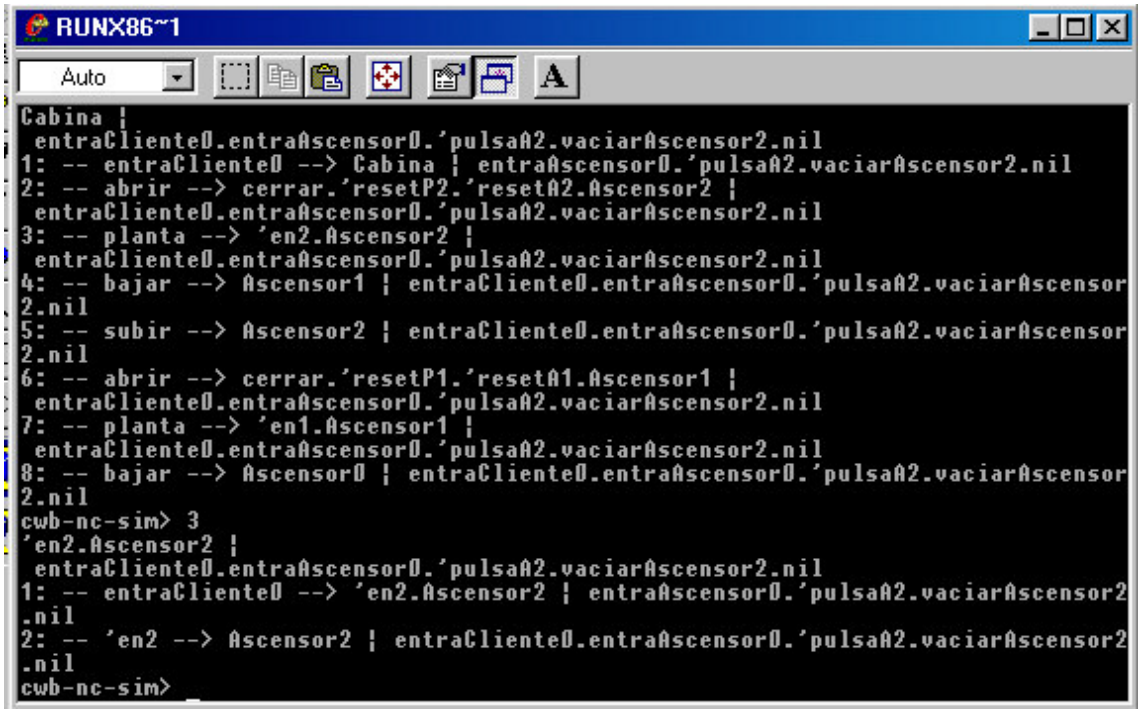
La primera imagen muestra como ha sido cargado el sistema. En esta captura se observan todas las señales correspondientes al ascensor y a los usuarios. El sistema cargado contaba con un proceso ascensor en la planta y otro proceso ascensor en la planta. Del mismo había tres procesos usuarios en las plantas que se dirigían respectivamente a las plantas.

En la siguiente imagen se aprecian las distintas señales que se dan en el sistema cuando un usuario ha llegado a la planta cero y desea utilizar el ascensor. Como se observa en la imagen, y al igual que se había explicado anteriormente, las señales que le quedan al proceso Usuario son
'pulsaP0.entraCliente0.'entraAscensor0.'pulsaA2.vaciarAscensor2.nil

Como se observa en la imagen el proceso Ascensor aún no ha realizado ninguna acción. Esto se observa en que el sistema cuenta con el proceso Usuario que ya ha realizado una serie de acciones (las ya explicadas), en paralelo con el proceso Cabina, que contenía varias elecciones sobre la posición inicial del ascensor.

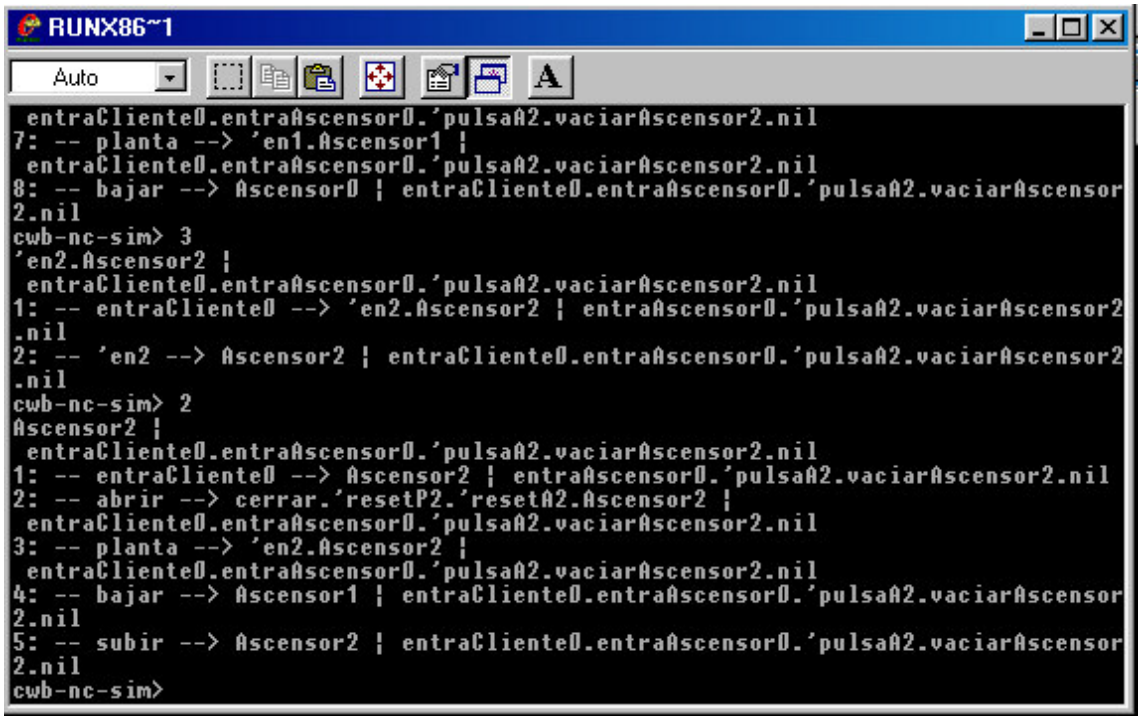


La siguiente captura muestra en la misma imagen dos acciones realizadas, de nuevo, por el proceso Usuario que corresponden con el aviso al sistema de que ha pulsado el botón de aviso de llamada al ascensor, lógicamente, el que está fuera del mismo, no se trata de un botón del panel del ascensor. La siguiente acción es el aviso por medio de una señal planta al proceso ascensor para que devuelva su posición actual.



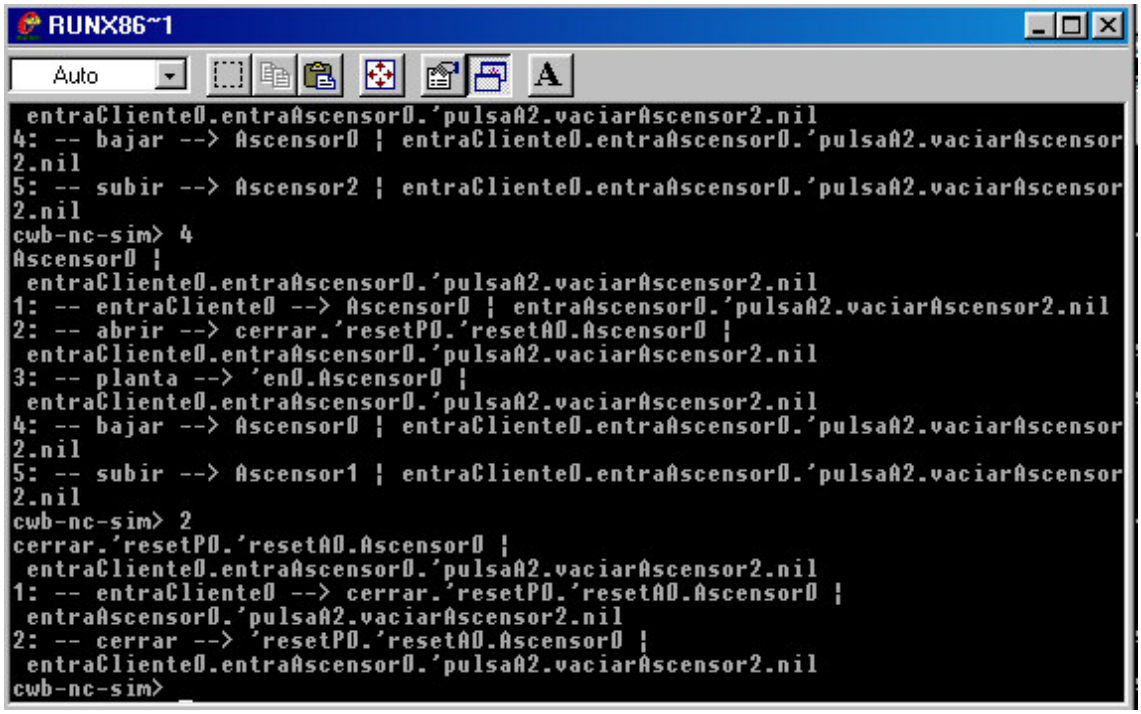
Esta comunicación se lleva a cabo entre un proceso controlador como ControlV y el ascensor, al igual que fue descrito en el pasado ejemplo.

La siguiente captura muestra la respuesta a la imagen anterior de lo ya dicho, es decir, la devolución de la señal utilizada para comunicar a los dos procesos citados anteriormente. El ascensor comunica al proceso controlador la planta en la que se encuentra actualmente. Una vez hecho esto, se pueden ver el resto de acciones posibles a realizar por parte del sistema.

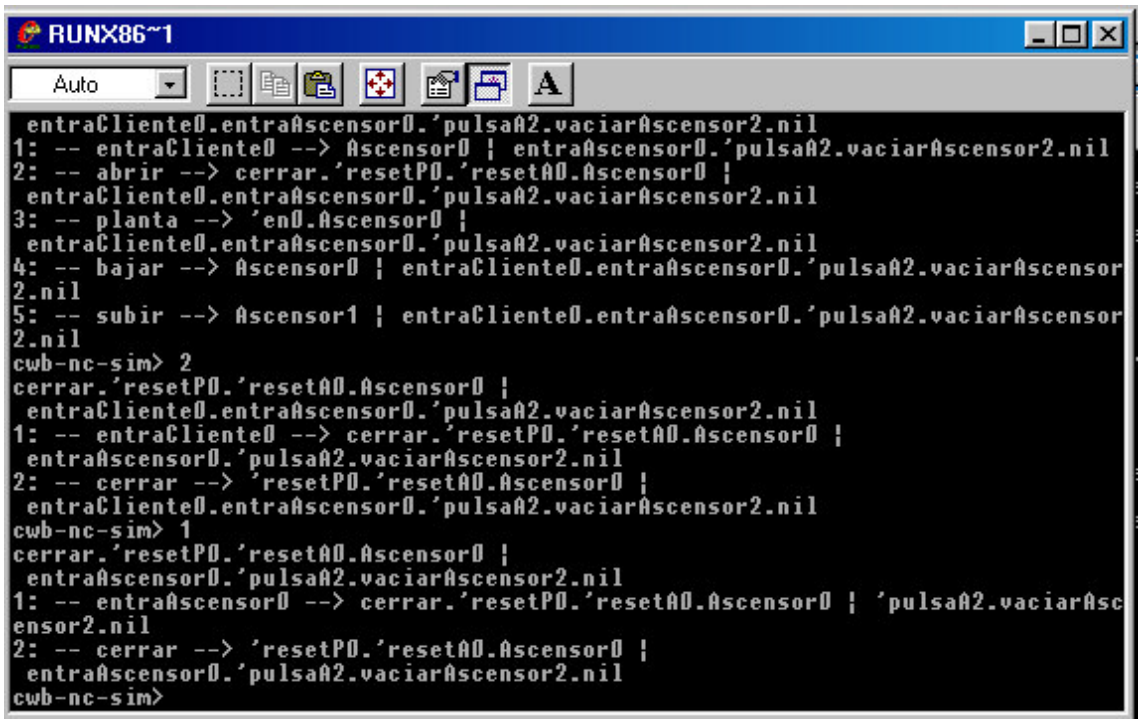


La captura siguiente contiene dos acciones realizadas por el sistema. La primera de ellas muestra la señal que recibe el ascensor para que baje una planta (se encontraba en nuestro ejemplo en la planta uno), mientras que la segunda corresponde con la apertura de las puertas del ascensor. En el ejemplo descrito anteriormente, la primera corresponde al momento en que el proceso controlador ControlV se comunicaba con el proceso controlador de los usuarios que querían tomar el ascensor en la planta1 (ControlPlantaOff1), pasado el cual ControlV se comunicaba con el proceso Bajar, el cual lanzaba una señal para que bajara una planta al ascensor.

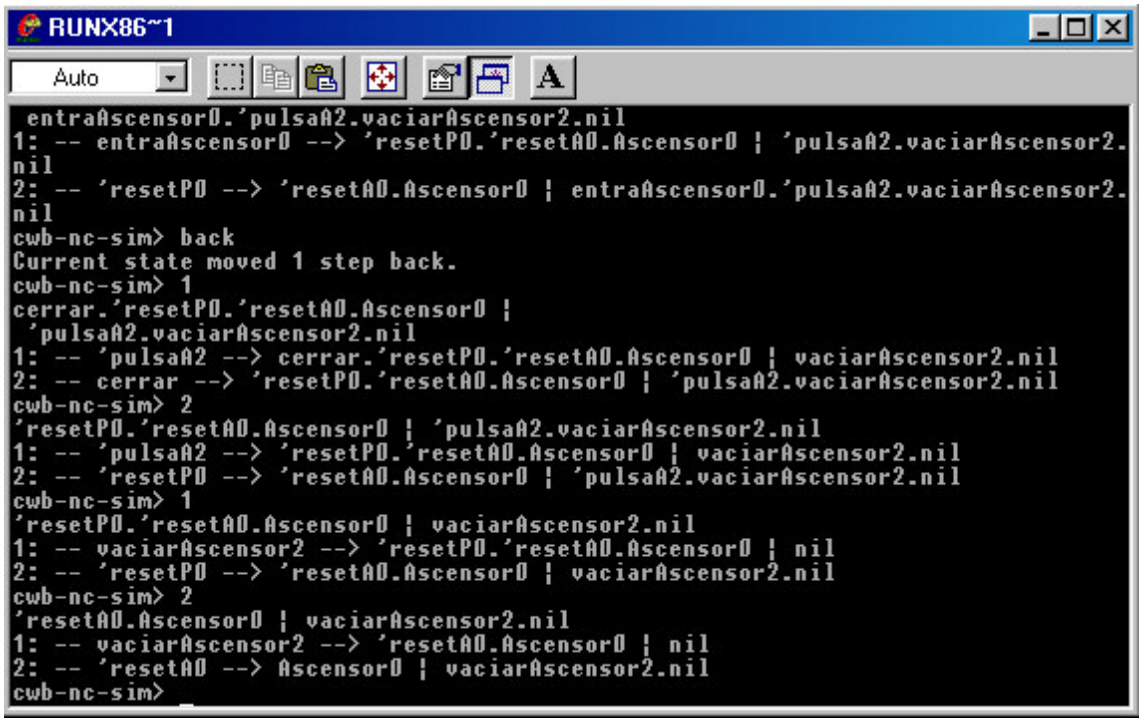
El segundo se produce como consecuencia entre el proceso AbrirV y el ascensor. El proceso AbrirV se da por el hecho de haber usuarios que quieren utilizar el ascensor en la planta cero.



La siguiente captura muestra la concesión del permiso de entrada al ascensor al usuario. Corresponde con la lectura o entrada en el proceso usuario de la señal entraCliente0. Esta señal era lanzada el proceso VaciandoPlanta en el momento en el que se pretendía que todos los usuarios que estaban aguardando para usar el ascensor, entraran en él y se vaciase la planta.

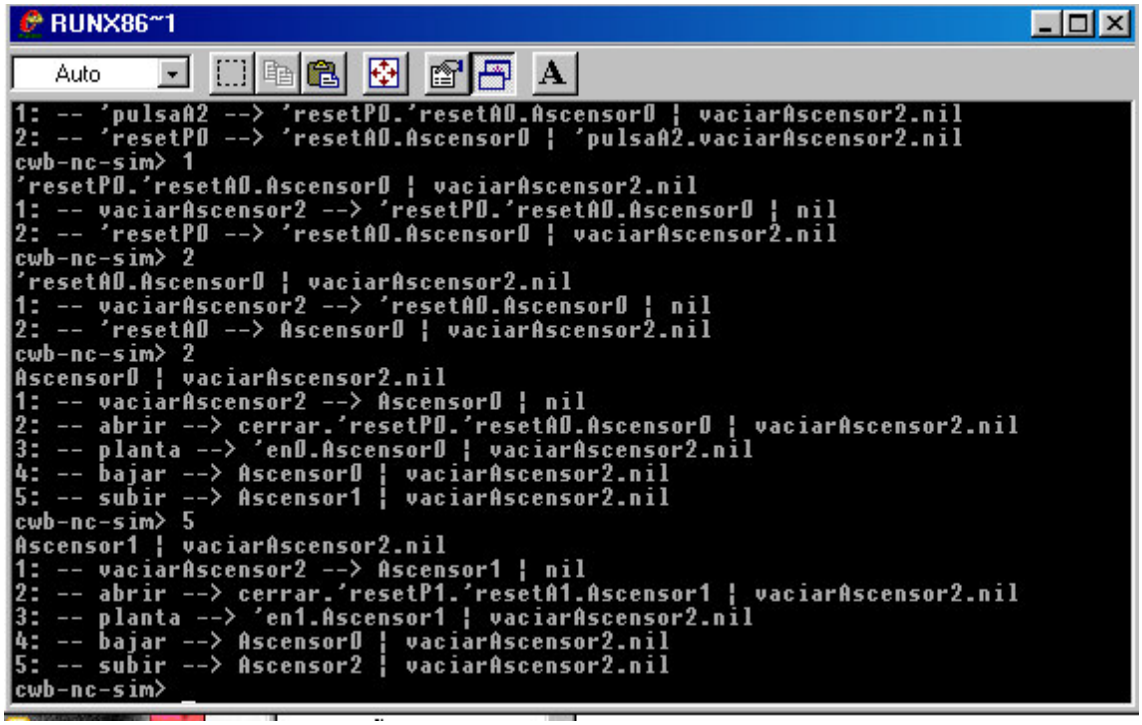


La siguiente captura muestra varias acciones. En la primera, el proceso Usuario advierte a los procesos interesados (ColaAscensor, que en un principio estaba vacía, y ahora pasará a tener como capacidad el número de usuarios que hayan subido al ascensor) que los usuarios ya han subido. A continuación, se cierran las puertas del ascensor, y por último, se borra la señal de petición de utilización del ascensor que habían hecho los usuarios que se acaban de montar en él para indicar su deseo de utilizarlo.



La imagen siguiente realiza dos acciones. La primera de ellas es borrar del ascensor, al igual que en el caso anterior la petición del ascensor de detenerse en la planta en la que lo ha hecho, sólo que en este caso se borra la detención procedente del pulsado de un botón del panel del interior del ascensor. En nuestro caso, dicho botón estaba apagado, ya que no había dentro del ascensor que pudiera haber hecho uso de él. Esto corresponde con la acción en la que el ascensor se comunica a través de la señal ‘resetA con el proceso BotónAscensorOff.

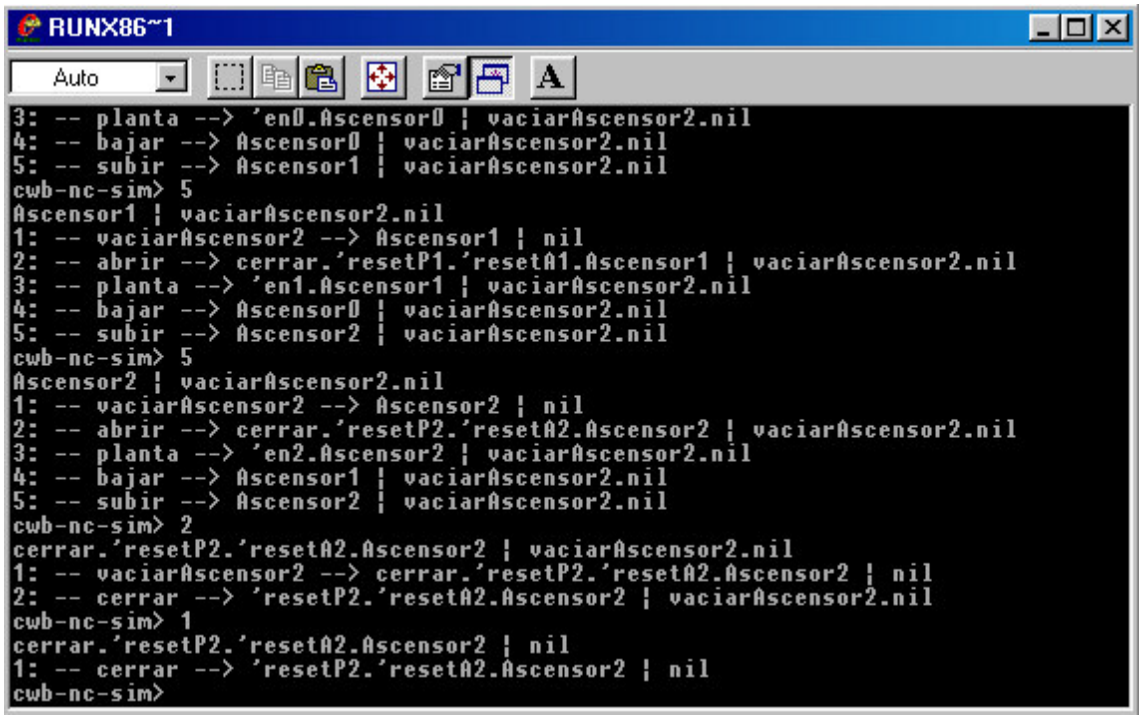
La siguiente acción corresponde al hecho de subir una planta, con lo que el ascensor pasa de la posición de planta cero a uno, esto es debido a que el ascensor iba a la planta dos debido a que es esta planta a la que se dirigía el usuario. Esta señal de subir procede del proceso análogo que es, a su vez, llamado por un de los procesos controladores. Realizada el envío de la señal de subida, se predispone al control a atender peticiones que vayan en su misma dirección con el paso al estado ControlA.



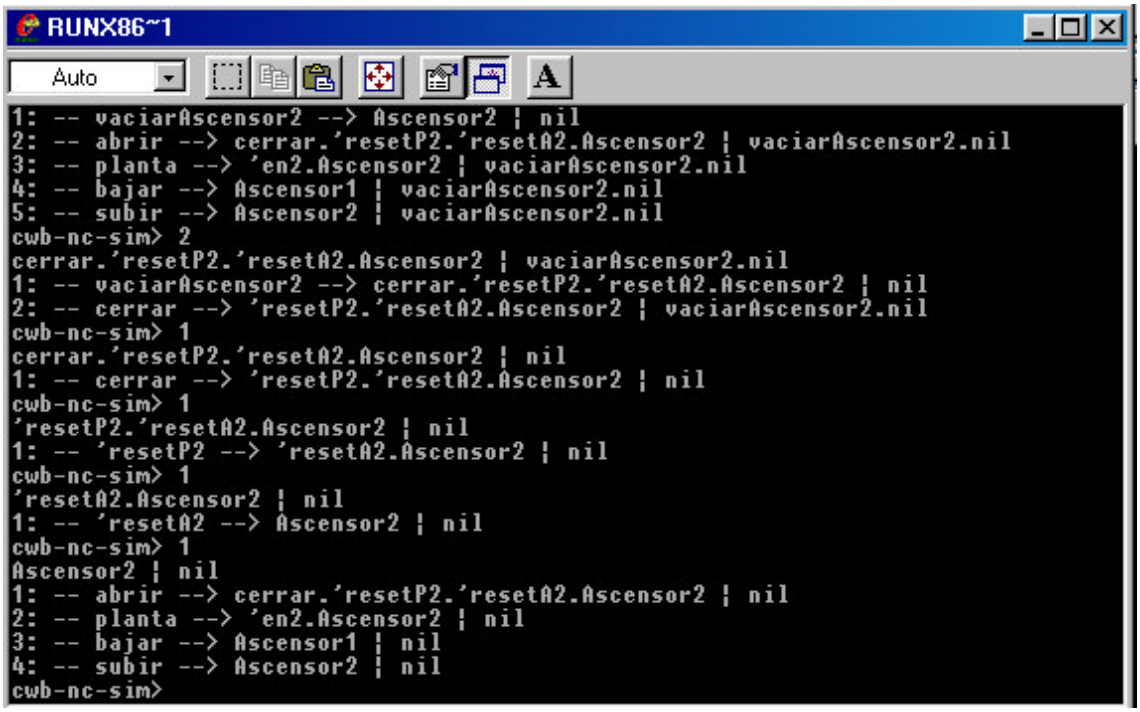
La siguiente captura muestra tres acciones realizadas. La ascensión de otro piso por parte del ascensor para alcanzar su destino. Todo lo ocurre es análogo a lo explicado en la imagen anterior para el caso en que el ascensor sube de la planta cero a la uno. Se puede observar el piso en que el está actualmente en la señal seleccionable número tres, en la que, a continuación, aparece enX, siendo X el número de planta actual.

Su siguiente acción es la apertura de puertas para que salgan las personas que iban a bajar en dicha planta y entren las que estén fuera. Esta acción de apertura de puertas de puertas como la posterior de cerrado suponen comunicaciones entre el ascensor y el proceso AbrirA.

Por último, bajan los usuarios del ascensor en dicha planta con lo que se vacía la cola existente de personas que almacenaba los usuarios que iban a bajar en aquélla. (Proceso ColaPlanta). Esta señal es enviada por el proceso de apertura AbrirA. Por un error al cargar el fichero de procesos, esta señal se llamó vaciarPlanta en vez del nombre utilizado antes, es decir, saleCliente.



La última captura muestra varias acciones que corresponde con el cerrado de puertas anteriormente citado, y las ya descritas acciones de borrado de las pulsaciones hechas desde dentro o fuera del ascensor para que bajara en dicha planta. En este caso, al revés que hace unas imágenes, el proceso BotonAscensor está encendido, mientras que el proceso que simula la llamada desde fuera está apagado (proceso BotonPlanta). La última acción muestra el fin del proceso Usuario a través de la señal de fin nil. Se puede comprobar que se pueden hacer más acciones correspondientes con las acciones de subir (permanecer en la planta actual) o bajar, las cuáles se dan como respuesta a una petición del proceso controlador ControlX.


RUNX86~1
Auto
1: -- vaciarAscensor2 --> Ascensor2 ¦ nil
2: -- abrir --> cerrar.'resetP2.'resetA2.Ascensor2 ¦ vaciarAscensor2.nil
3: -- planta --> 'en2.Ascensor2 ¦ vaciarAscensor2.nil
4: -- bajar --> Ascensor1 ¦ vaciarAscensor2.nil
5: -- subir --> Ascensor2 ¦ vaciarAscensor2.nil
cwb-nc-sim> 2
cerrar.'resetP2.'resetA2.Ascensor2 ¦ vaciarAscensor2.nil
1: -- vaciarAscensor2 --> cerrar.'resetP2.'resetA2.Ascensor2 ¦ nil
2: -- cerrar --> 'resetP2.'resetA2.Ascensor2 ¦ vaciarAscensor2.nil
cwb-nc-sim> 1
cerrar.'resetP2.'resetA2.Ascensor2 ¦ nil
1: -- cerrar --> 'resetP2.'resetA2.Ascensor2 ¦ nil
cwb-nc-sim> 1
'resetP2.'resetA2.Ascensor2 ¦ nil
1: -- 'resetP2 --> 'resetA2.Ascensor2 ¦ nil
cwb-nc-sim> 1
'resetA2.Ascensor2 ¦ nil
1: -- 'resetA2 --> Ascensor2 ¦ nil
cwb-nc-sim> 1
Ascensor2 ¦ nil
1: -- abrir --> cerrar.'resetP2.'resetA2.Ascensor2 ¦ nil
2: -- planta --> 'en2.Ascensor2 ¦ nil
3: -- bajar --> Ascensor1 ¦ nil
4: -- subir --> Ascensor2 ¦ nil
cwb-nc-sim>

8.-Comentario sobre el CWB-NC (concurrency workbench of North Carolina)

Este ha sido el programa de análisis y simulación de CCS más avanzado que hemos encontrado.

Vamos a detallar ahora objetivamente las diferentes ventajas y desventajas que hemos observado en el sistema.

8.1.- Ventajas y desventajas.

Ventajas:

- **Portabilidad**: El CWB-NC además de facilitar el código fuente, tenemos para distintan plataformas basadas en UNIX tales como linux red hat, mandrake linux y Solaris paquetes auto-instalables, y para WINDOWS un ejecutable auto-instalable también.
- **Sencillez de instalación**: Como acabo de decir tiene para casi todas las plataformas tenemos un auto-instalable.
- **Utilidades para varios lenguajes**: El CWB-NC permite no solo el lenguaje CCS sino también el TCCS (CCS con tiempos), PCCS (CCS con prioridades), Basic LOTOS y CSP.
- **Diversas funcionalidades**: La herramienta tiene funcionalidades para la simulación de procesos, la depuración de los mismos, demostrador de propiedades (expresadas en una cierta lógica) y demostraciones de igualdad, bisimulaciones y otras.
- **Interfaz sencilla**: La interfaz en modo consola nos permite dirigir el funcionamiento del sistema mediante el uso de comandos.
- **Comandos:** La lista de comandos, no es muy abultada y de rápida comprensión.
- **Documentación disponible:** El sistema tiene abundante documentación bastante bien estructurada, de libre distribución al igual que el sistema en sí, el único inconveniente es que solo se encuentra en ingles (lo cual en relealidad no es ningún problema. La documentación la tenemos en varios formatos y con diferentes contenidos según su finalidad.

Desventajas:

- **Dificultad en procesos complejos:** Aunque computacionalmente no supone ningún problema para el sistema, el tipo de interfaz (tan básico) en modo consola, no permite tener una idea global del sistema completo. No podemos visualizar el estado de muchos de los procesos (esto depende mucho de la resolución de la pantalla).
- **Falta de control en ciertos momentos:** El sistema cuando tiene la posibilidad de hacer una transición por una línea que está ocultada, la muestra como una transición ‘mu’ (m), con lo cual no podemos distinguir fácilmente unas de otras, a parte de que tampoco podemos distinguirlas de las mus (que también permite el sistema)
- **Falta de uniformidad:** Según la plataforma donde lo ejecutamos, aunque las ordenes con las mismas, la distribución de los diferentes elementos de la interfaz varia bastante, despistando un poco al usuario que cambia de plataforma.

- **No permite el tratamiento de CCS con parámetros:** Un problema grave con el que nos encontramos es que los sistemas más o menos complejos se expresan mediante parámetros, para hacer la descripción mas sencilla y más entendible por el usuario. Esto en realidad no hace que el sistema pierda completitud, ya que el CCS con parámetros, puede ser expresado de forma más o menos inmediata en CCS standar. Para sistemas medianamente grandes y con rango de valores de los diferentes parámetros, medianamente anchos, se hace aunque sencillo algorítmicamente, muy costoso en tiempo (y papel). Esto es lo que nos motivo a realizar un traductor automático.
- **Uso de paréntesis:** El sistema aunque si que considera el uso de paréntesis (lo cual es básico para las elecciones), no lo hace del todo bien, ya que algo del estilo, (x . Proc1 + z . Proc2) no lo admite a pesar de ser una expresión correcta en CCS, sin embargo 'Proc1 + Proc2' si que lo permite, así que podemos realizar una sencilla traducción a posteriori.

8.2.- Comparativa con otros sistemas.

Hemos encontrado otros programas aparte del de North Carolina, La universidad de Edimburgo tiene también un proyecto sobre concurrencia, muy parecido al de NC. Pero tenia algunos inconvenientes, tales como la dificultad de instalación; Solo tenía posibilidad de ejecutarse bajo linux, además no tenia paquete auto-instalable. Una vez instalado necesitaba de otros programas de linux tales como el emacs y el gimp, que debíamos de reconfigurar para que dieran servicio al programa. Sin embargo el sistema tenia una interfaz grafica ligerísimamente mejor que la de NC y sobre todo la capacidad de trabajar con el CCS con parámetros.

Otra herramienta de posible interés, era una herramienta desarrollada mediante el lenguaje ML, que también admitía CCS con parámetros, pero que sólo podía ser utilizada desde una plataforma Solaris.

El resto de los sistemas que hemos encontrado eran muy inferiores tanto en funcionalidades como en documentación y sencillez de instalación y manejo, al CBW de NC.

Además a la finalización de este proyecto, han sacado una segunda versión revisada en la cual problemas como el que describimos de los paréntesis han sido solucionados, además de haber progresado bastante en la parte de la amigabilidad de la interfaz.

Bloque 2:

Cálculo de ambientes

CÁLCULO DE AMBIENTES

1.- Introducción

Actualmente, una de las principales dificultades para diseñar aplicaciones distribuidas que exploten todo el poder computacional de Internet es la ausencia de abstracciones que apoyen la implementación de las mismas. El cálculo de ambientes un cálculo de procesos propuesto recientemente por Luca Cardelli y Adrew D. Gordon específicamente para este fin. El primer artículo publicado en por ambos describiendo este cálculo es de 1998 por lo que podemos ver que es realmente algo nuevo.

Existen dos áreas distintas para trabajar en movilidad: la informática móvil, que concierne a los cálculos que se llevan a cabo en máquinas móviles (portátiles, etc.), y el cálculo móvil, que concierne al código móvil que se mueve entre máquinas (applets, agentes, etc.). Lo que se pretende con el cálculo de ambientes es describir todos estos aspectos de movilidad con una única estructura que abarque agentes móviles, los ambientes donde estos agentes interactúan y la movilidad de los ambientes por sí mismos.

La principal dificultad para el cálculo móvil en la web no es la movilidad por sí misma, sino el manejo de los dominios administrativos. La movilidad requiere algo más que la tradicional noción de autorización a acceder a la información en ciertos dominios: implica la autorización a entrar y salir de ciertos dominios. En particular, no es real imaginar que un agente puede migrar desde un punto A hasta otro B en Internet. Un agente debe primero salir de su dominio administrativo (obteniendo permiso para hacerlo), entrar en otro dominio administrativo (de nuevo obteniendo permiso) y por último entrar en el área protegida de una máquina donde se le permita ejecutarse (después de haber obtenido el permiso para hacerlo). El acceso a la información está controlado a muchos niveles, por lo tanto, múltiples niveles de autorización deben verse implicados.

Con estas motivaciones adoptamos un paradigma de movilidad donde los ambientes están estructurados jerárquicamente, los agentes estas confinados a ambientes y los ambientes se mueven bajo el control de los ambientes. El reto está en hacer que los cálculos móviles aumenten proporcionalmente con los entornos de cálculos distribuidos, conectados intermitentemente y bien administrados.

A lo largo de este documento vamos a introducir el cálculo de ambientes en distintas fases, primero dando una sintaxis válida para expresar movilidad, posteriormente introduciremos la posibilidad de comunicación, y por último hablaremos de un posible sistema de tipos para este cálculo.

2.- Modelando la movilidad

El cálculo de ambientes se invento por Luca Cardelli y Andew D. Gordon como un nuevo paradigma, dado que nada de lo que ya existía se ajustaba exactamente a las necesidades que exigía la comunicación en Internet. Sin embargo, no se puede decir que partieran de 0, ya que tenían una amplia base de modelos distribuidos, dado que al fin y al cabo Internet es un sistema distribuido. La descripción de estos modelos nos puede

dar una breve visión para comprender luego mejor el cálculo de ambientes. Además, veremos también porque estos modelos no eran validos y porque se decidió por la implementación de un nuevo paradigma.

2.1 Formalismos de concurrencia, distribución y seguridad

El cálculo-π, junto con sus distintas variaciones, es un modelo de concurrencia que significa el punto de partida para el cálculo de ambientes. Se basa en la noción de comunicación de procesos a través de canales, con la habilidad de crear nuevos canales y de intercambiar canales sobre canales.

Vamos a discutir ahora porque algunos formalismos de concurrencia no satisfaces nuestro requisitos particulares. En general nos centraremos en el cálculo-π dado que como ya hemos dicho, es el punto de partida del cálculo de ambientes.

Todos los formalismos que se mencionan en este capítulo están descritos muy brevemente en el apéndice A. Aquí nos vamos a centrar en que es lo que estos formalismos no nos ofrecen.

Conectividad estática

Algunos de los primeros paradigmas de concurrencia permitía sistemas altamente dinámicos. Sin embargo, las principales descripciones formalizadas de concurrencia empezaron considerando únicamente conectividad estática. En el caso particular de CCS, que hemos descrito ampliamente en la primer parte de esta memoria, el conjunto de canales de comunicación que un proceso tiene permitidos no cambia durante la ejecución.

Estos modelos son insuficientes para modelar movilidad en un sentido general.

Conectividad dinámica

El cálculo-π es una extensión de CCS donde los canales pueden ser transmitido a través de otros canales, de tal manera que un proceso puede adquirir nuevos canales dinámicamente. La transmisión de canales (también llamada movilidad de canales) es una extensión muy poderosa del modelo básico de comunicación. La transmisión de un canal a través de otro canal da al receptor la capacidad de comunicación a través de ese canal.

Consideremos el final de un canal que se transmite a través de la frontera de un dominio sobre otro canal que se encuentra al otro lado de la frontera. Si el punto final que hemos transmitido permanece funcional, nos proporciona una conexión dinámica consolidada entre los dos lados de la frontera. Este es el tipo de conexiones que los firewalls típicamente prohíben. El nuevo canal que cruza el firewall puede verse como un túnel del firewall, pero el establecimiento de túneles de confianza implica algo más que el simple hecho de pasar un canal sobre otro, de lo contrario, el firewall perdería todo control de seguridad.

Un firewall debe ver el tráfico de comunicación de todo canal que lo cruze; esto es, debe actuar como un intermediario y remitente de mensajes entre el lado de fuera y

el de dentro del dominio. Si ve que el final de un canal lo traspasa, el firewall debe decidir si permite la comunicación en ese canal, y de ser así, debe crear un forwarder para ello. Por lo tanto, un canal a través de un firewall realmente debe ser manipulado conectados por un filtro.

Por eso, la capacidad de comunicar un canal depende no sólo de tener el comienzo de ese canal, sino también de donde se encuentra el final, y como llego hasta allí. Si este final fue mandado a través de un firewall, la capacidad de que realmente nos podamos comunicar depende de cual sea la actitud del firewall ante dicho canal.

Distribución

El cálculo-π no tiene ninguna noción inherente de distribución. Los procesos existen en una única dirección contigua, por defecto, porque no hay ninguna noción de construcción de distintas direcciones y sus efectos en los procesos. La interacción entre procesos se alcanza en global, nombres compartidos (los nombre del canal); por eso, se asume que todo proceso está fuera del alcance de los otros procesos. Esto hace las implementaciones distribuidas del cálculo-π original (y de CCS) algo duras, requiriendo alguna forma de consenso distribuido.

Se han hecho varias propuestas para hacer un cálculo-π que sea apropiado para las implementaciones distribuidas y para extenderlo con una conciencia de direcciones. El cálculo-π asíncrono se obtiene por un simple debilitamiento de las primitivas de sincronización del cálculo-π. Los mensajes asíncronos simplifican los requisitos para sincronizaciones distribuidas, pero no todavía no permiten las decisiones de comunicación. El cálculo-unión aproxima este problema arraigando cada canal a un proceso en particular; esto proporciona un único lugar donde se pueda resolver la sincronización. Llinda es una formalización de Linda usando técnicas de cálculo de procesos; como en las versiones distribuidas de Linda, Llinda tiene múltiples espacios de tuplas distribuidos, cada uno con su administrados de sincronización local.

Localidad

Por localidad entendemos conciencia de distribución: un proceso tiene alguna noción de dirección que ocupa, en sentido relativo o absoluto.

Un creciente cuerpo de literatura se está concentrando en la idea de añadir direcciones discretas a un cálculo de procesos y considerando el fallo de estas direcciones. Una noción de direcciones sólo no es suficiente, dado que las direcciones pueden estar todas realmente en el “mismo sitio”. Sin embargo, en presencia de fallos, uno puede observa que ciertas direcciones han dado fallo y otras no, y deduce que esas direcciones se encuentran en distintos sitios, pero puede que todas fallen a la vez. El cálculo-unión distribuido, por ejemplo, añade una noción de nombre de direcciones, y una noción de fallo distribuido; las direcciones forman un árbol, y los subárboles pueden migrar de una parte del árbol a otra.

Esta aproximación basada en fallos pretende modelizar entornos distribuidos tradicionales, y algoritmos tradicionales que toleran nodos de fallos. Sin embargo, en Internet, un fallo de un nodo es irrelevante comparado con la incapacidad de alcanzar dicho nodo. Los servidores web no suelen fallar de por vida, pero si que es frecuente

que fallen porque la red o un nodo se sobrecargan. Algunas veces vuelven en un lugar diferente, por ejemplo, cuando un web site cambie el proveedor de internet. Además la incapacidad de alcanzar un web site solo implica que un cierto camino no está permitido; esto no implica ni que exista un fallo del sitio ni que haya una incapacidad de alcance global. En este sentido, la percepción de un fallo de un nodo no puede asociarse simplemente con dicho nodo, sino que es una propiedad de toda la red, una propiedad que cambia a lo largo del tiempo.

En el cálculo de ambientes, la noción de localidad se induce no por fallos, sino por la necesidad de cruzar barreras. Las barreras producen una topología de direcciones no trivial y dinámica. Las direcciones se pueden diferenciar porque implica un esfuerzo moverse entre ellas, y porque algunas direcciones pueden estar ausentes y son distinguibles por direcciones que están presentes. El fallo sólo se representa convirtiendo la dirección en inalcanzable para siempre.

Movilidad

Hay varios sentidos en los que la gente habla de movilidad de procesos; vamos a tratar de distinguirlos.

Un canal del cálculo-π es móvil en el sentido en el que puede ser transmitido sobre otro canal. El conjunto de canales de un proceso influye en su localización, y un cambio de canales causa que el proceso cambie de localización. Esto se hace especialmente patente si un proceso tiene un único canal activo de comunicación en cada momento. La movilidad de canales puede ser interpretada como la causante de la movilidad de los procesos.

Sin embargo, la noción de movilidad de procesos que deseamos está unida a la idea de traspasar barreras de dominios. Esta es un muy discreta, on-off, forma de movilidad: un proceso puede o bien estar dentro de un dominio, o bien estar fuera. Representar esta forma de movilidad añadiendo o quitando canales no es inmediato. Por ejemplo, si un canal del cálculo-π cruza una barrera (esto es, se comunica con procesos que vienen a representar una barrera), sigue sin estar claro el sentido en el que el proceso a cruzado la barrera. De hecho, el mismo canal puede cruzar varias barreras de dominios que no tienen porque estar unidas, pero un proceso no debe existir en todos estos dominios a la vez. Por lo tanto, un proceso del cálculo-π puede haberse hecho para extenderse a lo largo de varios dominios, lo cual es algo que con el cálculo de ambientes se quiere borrar desde el principio.

Nuestra noción de procesos móviles está unida a la noción de dominios anidados. Cierta noción de anidamientos de dominios parece natural, y esto es difícil de representar adecuadamente en cálculos de procesos como el cálculo-π que están basados en conjuntos de procesos.

Otra forma de movilidad de procesos más directa consiste en que un proceso puede moverse siendo transmitido sobre canales a otro proceso. Este mecanismo no está presente en el cálculo-π básico, pero sí está presente en el llamado cálculo-π de alto nivel. Esta es una forma de movilidad de procesos que es el tipo de movilidad que podemos encontrar en los sistemas de agentes móviles donde los agentes se transmiten “enteros” y “vivos” a través de los canales de comunicación. De todas formas, hay un

par de detalles que debemos tener en cuenta y que de nuevo tienen que ver con las fronteras de dominios.

Primero, transmitir un proceso a través de un firewall sobre un canal depende de la existencia de un canal que ya ha cruzado el firewall. Esto es, todavía tenemos que modelar por separado es establecimiento de túneles en los firewalls.

Segundo, si un proceso tiene conexiones antes del movimiento, ¿se mantienen los otros al otro lado del firewall?. Si la respuesta es sí, entonces se pueden hacer canales arbitrarios para traspasar un firewall a través de la existencia de un canal. Si la respuesta es no necesariamente, cómo y por qué se rompen estos otro canales. Si el firewall actúa como un filtro, entonces tiene que analizar el proceso que esta pasando; esto es considerablemente más complicado que analizar simples mensajes.

Y tercero, trasmitir un proceso sobre un cana es una copia más que un operación de movimiento. Nada previene al proceso original de continuar, y nada previene al proceso trasmitido de ser rechazado más adelante.

En el cálculo de ambientes la movilidad no se representa como paso de canales sobre canales, ni como el paso de procesos sobre canales. Se representa como procesos saltando a través de fronteras. La identidad de el proceso que se mueve se preserva porque un proceso que cruza una frontera desaparece de su ubicación anterior.

Seguridad

Es posible extender un cálculo concurrente con primitivas de encriptación, como en el cálculo spi (extensión del cálculo-π).

En el cálculo de ambientes, la seguridad no está atada a las primitivas de encriptación sino a la capacidad o no de cruzar barreras, que se transmite por las habilidades. Dados los mecanismos necesarios para manipular fronteras, no nos aparece la necesidad de las primitivas de encriptación. Por ejemplo, un texto que está dentro de una frontera se puede ver como un texto encriptado en el sentido en que se necesita una habilidad para cruzar la barrera y poder leer el texto. Es decir, no es que el texto este encriptado, es que no lo podemos leer a no ser que tengamos la habilidad, por lo tanto, dicha habilidad se puede ver como la clave para desencriptar el texto.

2.2 Formalismos de reactividad

Una parte importante de interacción y computación en la web es ser capaz de reaccionar a las fluctuaciones del ancho de banda en tiempo real. Sin embargo este problema es menos significativo en el cálculo de ambientes que las otras observaciones computacionales que acabamos de ver, por lo que simplemente vamos a dejar constancia de él.

3.- Ambientes

La definición de ambiente hecha por Luca Cardelli y Andrew D. Gordon se basa en tres características:

- Un ambiente es un espacio acotado donde suceden los cálculos. La propiedad interesante aquí es la existencia de una frontera alrededor del ambiente. Si queremos mover cálculos fácilmente debemos ser capaces de determinar que es lo que debemos mover; una frontera determina que es lo que está dentro y lo que está fuera del ambiente. Algunos ejemplos de ambiente, en este sentido, son: una página web (acotada por un fichero), un espacio de dirección virtual (acotado por el rango de dirección), un sistema de ficheros Unís (acotado con un volumen físico), o un portátil (acotado por su caja y sus puertos de entrada/salida). Y algunos ejemplos de lo que no son ambientes son: threads (donde la frontera de lo que es "alcanzable" es difícil de determinar) o colecciones de objetos relacionados lógicamente.
- Un ambiente es algo que puede ser anidado con otro ambientes. Como ya discutimos, los dominios administrativos generalmente se organizan de forma jerárquica. Si queremos llevar una aplicación del trabajo a casa, la aplicación tiene que eliminarse de un ambiente (en este caso del trabajo) y ser insertada en otro ambiente diferente (en casa). Un portátil, por ejemplo, puede necesitar una acreditación para moverlo de su sitio de trabajo, o un pase gubernamental para introducirlo o sacarlo de un país.
- Un ambiente es algo que se puede mover como un todo. Si reconectamos un portátil a un red diferente, todo el espacio de direcciones y el sistema de ficheros se deben mover con él automáticamente. Si movemos un agente de un ordenador a otro, sus datos se deben mover también automáticamente.

Para ser más precisos, nosotros investigamos ambientes que tienen la siguiente estructura:

- Cada ambiente tiene un nombre. Este nombre se usa para controlar el acceso (entrada, salida, comunicación, etc.). En una situación real, el verdadero nombre del ambiente se protegería, y sólo se distribuirían habilidades específicas de cómo usar este nombre.
- Cada ambiente tiene una colección de agentes locales (también conocidos como threads, procesos, etc.). Estos son los cálculos que se ejecutan directamente con el ambiente y, en cierto sentido, controlan el ambiente. Por ejemplo, pueden mandar al ambiente que se mueva.
- Cada ambiente tiene una colección de subambientes. Cada subambiente a su vez, tiene su propio nombre, agentes, subambientes, etc.

En todo esto, los nombres son extremadamente importantes. Un nombre es:

- Algo que puede ser creado, pasado y usado para nombrar ambientes.
- Algo de lo que se pueden extraer habilidades.

4.- Movilidad

Vamos a continuación a comenzar describiendo un cálculo de ambientes mínimo, que incluye únicamente primitivas de movilidad. Aún así, veremos que este cálculo es bastante expresivo.

4.1 Primitivas de movilidad

La sintaxis del cálculo se define en la siguiente tabla. Las principales categorías sintácticas son procesos (incluyendo tanto a los ambiente como a los procesos que ejecutan acciones) y habilidades.

Primitivas de movilidad	
n	nombres
P, Q ::= (νn)P 0 P | Q !P n [P] M.P	procesos ocultamiento inactividad composición replicación ambiente acción
M ::= in n out n open n	habilidades puede entrar n puede salir n se puede abrir n

Nombres libres:

$$
\begin{aligned}
fn((\nu n)P) &\triangleq fn(P) - \{n\} \\
fn(0) &\triangleq \varnothing \\
fn(P \mid Q) &\triangleq fn(P) \cup fn(Q) \\
fn(!P) &\triangleq fn(P) \\
fn(n[P]) &\triangleq \{n\} \cup fn(P) \\
fn(M.P) &\triangleq fn(M) \cup fn(P) \\
\\
fn(\text{in } n) &\triangleq \{n\} \\
fn(\text{out } n) &\triangleq \{n\} \\
fn(\text{open } n) &\triangleq \{n\}
\end{aligned}
$$

Escribimos $P\{n \leftarrow m\}$ para la substitución de el nombre m para cada ocurrencia libre del nombre n en el proceso P. De la misma manera para $M\{n \leftarrow m\}$.

Convenciones sintácticas:

(νn)P | Q	se lee	((νn)P) | Q
!P | Q	se lee	(!P) | Q
M.P | Q	se lee	(M.P) | Q

Abreviaciones:

$(\nu n_1 ... n_m)P \quad \triangleq \quad (\nu n_1) \ldots (\nu n_m)P$

$n[] \quad \triangleq \quad n[0]$

$M \quad \triangleq \quad M.0$

4.2 Explicaciones

A continuación vamos a describir cada una de las primitivas que hemos introducido anteriormente en la sintaxis. Una reducción $P \rightarrow Q$ describe la evolución de un proceso P en un nuevo proceso Q.

Ocultamiento

El operador de ocultamiento:

$(\nu n)P$

crea un nuevo nombre n con un ámbito P. El nuevo nombre puede ser usado para nombrar ambientes y para operar en ambientes por nombre.

Como en el cálculo-π, el operador (νn) puede moverse hacia fuera para extender el ámbito de un nombre, o bien por el contrario, moverse hacia dentro para restringir su ámbito.

Regla de reducción de ocultamiento:

$P \rightarrow Q \quad \Rightarrow \quad (\nu n)P \rightarrow (\nu n)Q$

Inactividad

El proceso:

0

es el proceso que no hace nada. No se puede reducir.

Composición

La ejecución paralela se denota por un operador binario que es conmutativo y asociativo:

$P \mid Q$

Regla de reducción de composición:

$P \rightarrow Q \quad \Rightarrow \quad P \mid R \rightarrow Q \mid R$

Replicación

La replicación es una manera de representar iteración y recursión. El proceso:

!P

denota la creación de una replica del proceso P. Esto es, !P puede producir tantas replicas paralelas de P como sean necesarias, y es equivalente a P | !P. No hay reglas de reducción para !P; en particular, el termino P bajo ! no puede reducirse hasta que no se expanda fuera como P | !P.

Ambientes

Un ambiente se escribe:

n[P]

donde n es el nombre del ambiente y P es el proceso que se ejecuta dentro del ambiente.

En n[P], se sobreentiende que P se está ejecutando, y que P puede ser la composición en paralelo de varios procesos. Debemos enfatizar que P se está ejecutando aún cuando el ambiente que le rodea se está moviendo. Este hecho de tener un proceso dentro de un ambiente ejecutándose aún cuando el ambiente esté moviéndose, puede ser realista o no dependiendo de la naturaleza del ambiente y del medio de comunicación a través del cual se está moviendo el ambiente. Sin embargo, vamos a trabajar con esta idea dado que es consistente.

La regla de reducción de ambientes es la siguiente:

$$P \rightarrow Q \qquad \Rightarrow \qquad n[P] \rightarrow n[Q]$$

En general, un ambiente tiene una estructura de árbol inducida por el anidamiento de ambientes. Cada nodo de esta estructura puede contener una colección de procesos que se están ejecutando en paralelo, además de subambientes. Por lo tanto, la forma general de un ambiente es:

$$n[P_1 \mid ... \mid P_p \mid m_1[...] \mid ... \mid m_q[...]] \qquad (P_i \neq n_i[...])$$

Nada nos previene de la existencia de dos o más ambientes con el mismo nombre, ya sea anidados o al mismo nivel. Cuando un nombre es creado, se puede usar para nombrar múltiples ambientes. Además, !n[P] genera múltiples ambientes con el mismo nombre.

Acciones y habilidades

Las operaciones que cambian la estructura jerárquica de los ambientes son delicadas, por lo que están restringidas por las habilidades. Gracias a las habilidades, un ambiente puede permitir a otro ambiente llevar a cabo cierta operación sin desvelar su verdadero nombre.

El proceso:

M.P

Ejecuta una acción regulada por la habilidad M, y que continua como el proceso P. El proceso P no comienza a ejecutarse hasta que la acción ya se ha terminado de ejecutar. Para cada tipo de habilidad M, tenemos una regla específica para reducir M.P. Cada una de estas reglas las explicaremos a continuación según vayamos describiendo las habilidades una por una.

Tenemos tres tipos de habilidades: una para entrar en un ambiente, otro para salir y otra para abrir un ambiente. Las habilidades se consiguen a través de los nombres; dado un nombre m, la habilidad in m permite entrar en m. No obstante, la posesión de una o todas estas habilidades es insuficiente para reconstruir el nombre original m del que fueron extraídas.

Habilidad de entrada

Una habilidad de entrada, in m, puede ser usada en la acción:

in m.P

que ordena al ambiente que rodea a in m.P que entre en el ambiente cuyo nombre es m. Si no se encuentra ningún ambiente al mismo nivel con nombre m, la operación bloquea hasta que dicho ambiente exista. Si, por el contrario, existen más de un ambiente al mismo nivel con nombre m, podemos escoger cualquiera de ellos. La regla de reducción es la siguiente:

$$n[\text{in } m.P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q] \mid R]$$

Si tiene éxito, esta reducción transforma un ambiente, n, que se encuentra al mismo nivel que otro, m, en hijo de este último. Después de su ejecución, el proceso in m.P continúa con P, y tanto P como Q se encuentran en un nivel más bajo en el árbol de ambientes del que se encontraban anteriormente.

Habilidad de salida

Una habilidad de salida, out m, puede ser usada en la acción:

out m.P

que ordena al ambiente que rodea a out m.P que salga de su ambiente padre cuyo nombre es m. Si el padre no se llama m, la operación bloquea hasta que exista dicho padre con nombre m. La regla de reducción es la siguiente:

$$m[n[\text{out } m.P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R]$$

Si tiene éxito, esta reducción transforma el hijo n de un ambiente m en un hermano de este último. Tras la ejecución, el proceso out m.P continúa con P, y tanto P como Q se encuentran en un nivel superior del que se encontraban antes en el árbol de ambientes.

Habilidad de apertura

Una habilidad de apertura, open m, puede ser usada en la acción:

open m.P

Esta acción proporciona un a manera de disolver la frontera de un ambiente cuyo nombre es m y que se encuentra al mismo nivel que open, de acuerdo con la regla:

$$\text{open } m.P \mid m[Q] \rightarrow P \mid Q$$

Si no se encuentra el ambiente m, la operación bloquea hasta que dicho ambiente exista. Si existe más de un ambiente con nombre m, se puede elegir cualquiera de ellos para abrir.

Una operación de apertura puede ser decepcionante tanto para P como para Q. Desde el punto de vista de P, nadie le puede decir que es lo que va a hacer Q cuando se libere. Desde el punto de vista de Q, su entorno se está abriendo. De todas formas, esta operación se comporta relativamente bien porque: (1) la disolución se inicia por medio de un agente, open m.P, así que la apariencia de Q al mismo nivel que P no es del todo inesperada; (2) open m es una habilidad que se da afuera a través de m, así que m[Q] no se puede disolver si este no lo desea así.

4.3 Semántica operacional

Ahora daremos una semántica operacional para el cálculo explicado en la sección 4.2, basándonos en congruencias entre procesos y la relación de reducción que ya hemos mostrado.

Los procesos del cálculo se agrupan en clases de equivalencia denotando congruencia estructural.

Congruencias estructurales	
$P \equiv P$	Reflexión
$P \equiv Q \Rightarrow Q \equiv P$	Simetría
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	Transitividad
$P \equiv Q \Rightarrow (\nu n)P \equiv (\nu n)Q$	Ocultamiento

$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	Paralelismo
$P \equiv Q \Rightarrow !P \equiv !Q$	Réplica
$P \equiv Q \Rightarrow n[P] \equiv n[Q]$	Ambiente
$P \equiv Q \Rightarrow M.P \equiv M.Q$	Acción
$P \mid Q \equiv Q \mid P$	Conmutación de paralelismo
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	Asociación de paralelismo
$!P \equiv P \mid !P$	Paralelismo de réplica
$(\nu n)\ (\nu m)P \equiv (\nu m)(\nu n)P$	Restricción de ocultamiento
$(\nu n)(P \mid Q) \equiv P \mid (\nu n)Q$ si $n \notin fn(P)$	Ocultamiento de paralelismo
$(\nu n)(m[P]) \equiv m[(\nu n)P]$ si $n \neq m$	Ocultamiento de ambiente
$P \mid 0 \equiv P$	Paralelismo con 0
$(\nu n)0 \equiv 0$	Ocultamiento de 0
$!0 \equiv 0$	Réplica de 0

Además entendemos como iguales procesos cuya frontera se renombra:

$$(\nu n)P \quad = \quad (\nu n)P\{n \leftarrow m\} \quad \text{si } m \notin fn(P)$$

Esto no debe llevarnos a equívoco con los siguientes términos, que son distintos:

$! (\nu n)P \quad \neq \quad (\nu n)!P$	la réplica crea un nuevo nombre
$n[P] \mid n[Q] \quad \neq \quad n[P \mid Q]$	múltiples ambientes con nombre n tienen identidades separadas.

En la siguiente tabla, una tabla de reducción, se muestra el comportamiento de los procesos.

Reducciones	
$n[\text{in } m.P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q] \mid R]$	Entrada
$m[n[\text{out } m.P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R]$	Salida
$\text{open } n.P \mid n[Q] \rightarrow P \mid Q$	Apertura
$P \rightarrow Q \ \Rightarrow \ (\nu n)P \rightarrow (\nu n)Q$	Ocultamiento
$P \rightarrow Q \ \Rightarrow \ n[P] \rightarrow n[Q]$	Ambiente
$P \rightarrow Q \ \Rightarrow \ P \mid R \rightarrow Q \mid R$	Paralelismo
$P' \equiv P, P \rightarrow Q, Q \equiv Q' \ \Rightarrow \ P' \rightarrow Q'$	Congruencia
$\rightarrow^*$	Cierre reflexivo y transitivo de $\rightarrow$

Las tres primeras reglas consisten en hacer un paso de reducción para in, out y open. Las tres siguientes propagan las reducciones a través de los alcances de visibilidad, anidamiento de ambientes y paralelismo. La siguiente regla permite el uso de equivalencias durante la reducción y por último, la regla final es el encadenamiento de reducciones.

Podemos formular equivalencia contextual en términos de mirar la presencia de ambiente en niveles superiores. Así, tenemos las dos definiciones siguientes:

- Decimos que un proceso P exhibe un ambiente de nombre n, y lo escribimos por $P \downarrow n$, si P es un proceso que contiene un ambiente llamado n en un nivel superior.
- Decimos que un proceso P exhibe eventualmente un ambiente de nombre n, y lo escribimos por $P \Downarrow n$, si después de un cierto número de reducciones, P exhibe un ambiente de nombre n.

Estas dos definiciones las podemos expresar formalmente:

$$P \downarrow n \triangleq P \equiv (\nu\, m_1...m_i)(n[P'] \mid P'') \quad \text{donde } n \notin \{m_1...m_i\}$$
$$P \Downarrow n \triangleq P \rightarrow^* Q \text{ and } Q \downarrow n$$

Ahora podemos definir la equivalencia contextual sirviéndonos del predicado $P \Downarrow n$. Sea el contexto C() un proceso que contiene un número de agujeros mayor o igual que 0, y para cada proceso P, sea C(P) el proceso que se obtiene rellenando cada agujero de C con una copia de P. Definimos la equivalencia contextual ($P \simeq Q$) de la siguiente manera:

$$P \simeq Q \qquad \triangleq \qquad \text{para todo n y C(), } C(P) \Downarrow n \Leftrightarrow C(Q) \Downarrow n$$

Para terminar con la semántica, escribimos $P \rightarrow^* \simeq Q$ si existe un R tal que $P \rightarrow^* R$ y $R \simeq Q$.

4.4 Ejemplos

A continuación vamos a introducir con lo que ya sabemos de cálculo de ambientes, algunos ejemplos que utilizaremos posteriormente bien para explicar el protocolo de seguridad diseñado con ambientes o bien para explicar otras secciones posteriores.

Canales

En este ejemplo vamos a representar los canales de CCS que hemos visto en la anterior sección mediante cálculo de ambientes. Cada canal se compone de dos puertos, un puerto de "entrada" y otro de "salida". De lo que se trata es de sincronizar dos procesos a través de los dos canales (cada proceso intenta la sincronización en uno de los canales). Estos procesos serán por lo tanto de la forma k.P donde k será el nombre del canal en el que se quieren sincronizar y P el proceso en sí.

Para definir esto dos procesos nos va a ser útil pensar en la primitiva in de una manera distinta a como lo hemos hecho hasta ahora. Lo que hemos visto es que con la primitiva in, si se encontraba dentro de un ambiente, podía moverse a otro ambiente que estuviera en el mismo nivel y cuyo nombre coincidiera con el parámetro de la primitiva. Para este caso, nos interesa otro tipo de movimiento, en el que no sea necesario encontrarnos en un ambiente para movernos al ambiente hermano, sino que queremos una primitiva que nos permita movernos a un ambiente que este al mismo nivel que el

proceso, no que el ambiente en el que se encuentra este proceso. Lo mismo sucede con la primitiva out. Definimos este nuevo tipo de movimiento (indicándolo con el prefijo mv) de la siguiente manera:

$$mv\ in\ m.P \mid m[R] \rightarrow m[P \mid R]$$
$$m[mv\ out\ m.P \mid R] \rightarrow P \mid m[R]$$

Para codificar estas dos nuevas primitivas, vamos a utiliza otro nuevo operador que llamaremos acid y que lo que hace es disolver el ambiente en el que se encuentre el proceso, es decir, es como si abriera el cascarón para el proceso. Se define así:

$$m[acid.P \mid Q] \rightarrow P \mid Q$$

Con esto, ya podemos codificar los operadores para los nuevos movimientos de la siguiente manera:

$$mv\ in\ m.P \triangleq (\nu q)q[in\ n.acid.P]$$
$$mv\ out\ n.P \triangleq (\nu q)q[out\ n.acid.P]$$

Lo que hacemos es meter la primitiva in o out dentro de un ambiente que será el que se mueva y que posteriormente disolvemos.

Con esto, ya sólo nos queda pensar en los canales como una operación de sincronización, de manera que el proceso que llegue a intentar sincronizarse por canal de entrada, intenta adquirir un cerrojo que debe crear el otro proceso, es decir, el que intenta sincronizarse por el canal de salida, y una vez hecho esto, sucede lo mismo pero a la inversa, es el segundo proceso el que lanza el cerrojo y el primero el que lo coge. Las operaciones de adquirir y lanzar un cerrojo se codifican de la siguiente manera:

$$acquire\ n.P \triangleq open\ n.P$$
$$release\ n.P \triangleq n[] \mid P$$

Con esto, la codificación de los procesos necesario para la creación de un canal CCS son los siguientes:

$$n?.P \triangleq mv\ in\ n.\ acquire\ rd.\ release\ wr.\ mv\ out\ n.P$$
$$n?.P \triangleq mv\ in\ n.\ release\ rd.\ acquire\ wr.\ mv\ out\ n.P$$

Renombramiento

Con la operación de open podemos codificar la operación de renombramiento de ambientes a la que llamaremos be:

$$n\ be\ m.P \triangleq m[out\ n.\ open\ n.\ P] \mid in\ m$$

En lo que consiste esta operación es en sacar alguno de los procesos de n fuera de este ambiente pero dentro de otro ambiente m, donde n es el ambiente que queremos

renombrar y m es el nuevo nombre. Una vez hecho esto, metemos el resto de n dentro de n y por último solo nos queda abrir n.

Otra forma de hacerlo sería con el tipo de movimiento explicado en el ejemplo anterior en el que nos movíamos a un ambiente que estaba en el mismo nivel que el proceso. De esta forma quedaría algo así:

$$n \text{ be } m.P \triangleq m[\text{out } n.\ P] \mid \text{open } n \mid mv \text{ in } m$$

Un ejemplo de un renombramiento podría ser el siguiente:

$$\begin{aligned} n[n \text{ be } m.P \mid Q] \equiv n[m[\text{out } n.\text{open } n.P] \mid \text{in } m \mid Q] &\rightarrow m[\text{open } n.P] \mid n[\text{in } m \mid Q\ ] \\ &\rightarrow m[\text{open } n.P \mid n[Q]] \\ &\rightarrow m[P \mid Q] \end{aligned}$$

El problema es que la operación be no es una operación atómica, por lo tanto, puede darse el caso de que Q realice algún movimiento indeseado. Para solucionar esto, podemos utilizar un cerrojo antes del bloqueo sobre n que impida ningún movimiento de Q.

Radar

En lo que consiste esta operación que llamaremos see, es en detectar la presencia de un ambiente dado a través de las operaciones be y open. La codificación es la siguiente:

$$\text{see } n.\ P \triangleq (\nu\ r\ s)(r[\text{in } n.\ \text{out } n,\ r \text{ be } s.\ P] \mid \text{open } s)$$

Lo que hace esta operación es que si existe el ambiente n, lo que va a ocurrir es que r va a entra de él, salir de él y una vez ha hecho esto, que se da porque existe n, se renombra a s. Cuando ya esta renombrado se abre y por lo tanto P queda activo.

Esto dos últimos ejemplos, el renombramiento y el radar los usamos en la codificación del protocolo de seguridad, mientras que los canales los hemos incluido como ejemplo por su importancia y dado que esto nos permitía introducir la noción de movilidad subjetiva.

5.- Comunicación

Aunque lo que ya hemos visto del cálculo móvil es suficientemente potente para ser Turing-completo, no posee comunicación o operadores de restricción de variables. Estos operadores parecen necesarios para, por ejemplo, poder codificar plácidamente otros formalismos como pueden ser el cálculo-λ o el cálculo-π (ver apéndice A).

Por lo tanto, ahora lo que nos queda por ver es elegir un mecanismo de comunicación que usaremos para permitir a los ambientes intercambiar mensajes entre ellos. A la hora de pensar en dicho mecanismo de comunicación, muchos son los que podemos elegir, sin embargo es claro que la elección de este mecanismo es, de algún

modo, ortogonal con las primitivas de movilidad explicadas en el capítulo anterior. Sin embargo, debemos intentar que la comunicación no interfiera en las restricciones impuestas por las habilidades. Esto lo que nos sugiere es que una primitiva de comunicación debe ser absolutamente local, y que la transmisión de mensajes no locales debe estar restringida por las habilidades.

5.1 Primitivas de comunicación

Lo primero que haremos será mostrar la sintaxis del cálculo. Las primitivas de movilidad son esencialmente las mismas que vimos en el capítulo anterior, salvo algún detalle que tendremos que cambiar como consecuencia de introducir variables de comunicación. Con respecto a la sintaxis mostrada en la sección anterior, se añaden las primitivas input((x).P) y output(⟨M⟩), que son las primitivas que nos permiten la comunicación, y además enriquecemos las habilidades para incluir paths.

Movilidad y primitivas de comunicación	
P, Q ::=	procesos
(νn)P	restricción
0	inactividad
P | Q	composición
!P	replicación
M [P]	ambiente
M.P	acción de habilidad
(x).P	acción de entrada
⟨M⟩	acción asíncrona de salida
M ::=	habilidades
x	variable
n	nombre
in M	puede entrar en M
out M	puede salir de M
open M	se puede abrir M
ε	null
M.M'	path

Nombres libres (revisión y adiciones):

$$
\begin{aligned}
fn(M[P]) &\triangleq fn(M) \cup fn(P) \\
fn((x).P) &\triangleq fn(P) \\
fn(\langle M \rangle) &\triangleq fn(M) \\
\\
fn(x) &\triangleq \varnothing \\
fn(n) &\triangleq \{n\} \\
fn(\varepsilon) &\triangleq \varnothing \\
fn(M.M') &\triangleq fn(M) \cup fn(M')
\end{aligned}
$$

Variables libres:

$$
\begin{aligned}
fv((\nu n)P) &\triangleq fv(P) \\
fv(0) &\triangleq \varnothing
\end{aligned}
$$

$fv(P \mid Q) \triangleq fv(P) \cup fv(Q)$

$fv(!P) \triangleq fv(P)$

$fv(M[P]) \triangleq fv(M) \cup fv(P)$

$fv(M.P) \triangleq fv(M) \cup fv(P)$

$fv((x).P) \triangleq fv(P) - \{x\}$

$fv(\langle M \rangle) \triangleq fv(M)$

$fv(x) \triangleq \{x\}$

$fv(n) \triangleq \varnothing$

$fv(\text{in } M) \triangleq fv(M)$

$fv(\text{out } M) \triangleq fv(M)$

$fv(\text{open } M) \triangleq fv(M)$

$fv(\varepsilon) \triangleq \varnothing$

$fv(M.M') \triangleq fv(M) \cup fv(M')$

Escribimos $P\{x \leftarrow M\}$ para denotar la substitución de la habilidad M por cada ocurrencia libre de la variable x en el proceso P. De igual manera para $M\{x \leftarrow M'\}$.

Nuevas convenciones sintácticas:

$(x).P \mid Q$ se lee $((x).P) \mid Q$

5.2 Explicaciones

Valores comunicables

Las entidades que podemos comunicar son bien nombres o bien habilidades. En situaciones reales es bastante raro que se de la comunicación de nombres por la falta de seguridad que esto implica dado que el conocer el nombre de un ambiente nos proporciona un gran control sobre él. Lo que si será muy normal es la comunicación de habilidades para permitir una interacción controlada entre ambientes.

Llegados a este punto, parece útil el poder combinar múltiples habilidades en un path, especialmente cuando una o más de estas habilidades se representan por variables de entrada. Con este propósito se introducen los paths (M.M'). Por ejemplo, (in n. in m).

Además, para este propósito, se han añadido los nombres a la lista de habilidades. Lo podemos ver pensando que un nombre es la habilidad de crear un ambiente con dicho nombre.

I/O de ambientes

La comunicación más simple en la que podemos pensar es una comunicación anónima con un ambiente.

$(x).P$ acción de entrada

$\langle M \rangle$ acción asíncrona de salida

Una acción de salida libera una habilidad (posiblemente un nombre) en el ámbito local del ambiente que le rodea. Una acción de entrada captura una habilidad del ámbito local y lo ata a una variable que este en el ámbito de visibilidad. Tenemos la reducción:

$$(x).P \mid \langle M \rangle \quad \rightarrow \quad P\{x \leftarrow M\}$$

Esta mecanismo de comunicación local encaja bastante bien con la intuición que tenemos ya de los ambientes. En particular, la comunicación de larga distancia, como el movimiento de larga distancia, no debería suceder de forma automática porque los mensajes pueden tener que atravesar firewalls.

Una anomalía sintáctica

Para permitir tanto a los nombres como a las habilidades las acciones de entrada y de salida, hay un orden sintáctico que incluye a ambos. Por lo tanto, un termino sin sentido de la forma n.P puede presentarse, por ejemplo, del proceso $((x).x.P) \mid \langle n \rangle$. Esta anomalía está causada por el deseo de denotar habilidades de movimiento por variables, como en (x).x.P, y por el deseo de denotar nombres por variables, como en (x).x[P]. Permitimos n.P, sintácticamente, para hacer que las substituciones estén siempre bien definidas. Una posibilidad para eliminar esta anomalía consistiría en un sistema de tipos que distinguiera entre nombres y habilidades de movimiento.

5.3 Semántica operacional

Para el cálculo extendido, la congruencia estructural es básicamente la misma que mostramos en el capítulo anterior, concretamente el apartado 4.2. Las únicas variaciones son que ahora P y M se extienden a clases más grandes, y además, debemos añadir las siguientes equivalencias:

Congruencias estructurales	
$P \equiv Q \Rightarrow M[P] \equiv M[Q]$	Ambientes
$P \equiv Q \Rightarrow (x).P \equiv (x).Q$	Input
$\varepsilon.P \equiv P$	ε
$(M.M').P \equiv M.M'.P$	.

Ahora también tenemos que indentificar procesos con el renombramiento de variables de vínculo:

$$(x).P \quad = \quad (y).P\{x \leftarrow y\} \quad \text{si } y \notin fv(P)$$

Por último, tenemos una nueva regla de reducción:

Reducciones	
$(x).P \mid \langle M \rangle \rightarrow P\{x \leftarrow M\}$	Comm

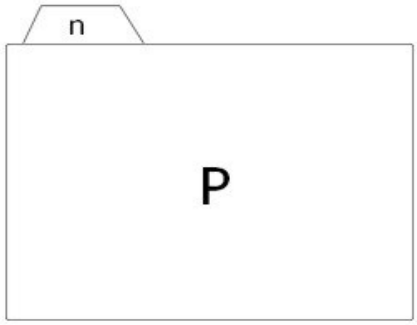
Necesitamos modificar la definición que dimos en el apartado 3.3 de equivalencia contextual: $P \simeq Q$ si y sólo si para todo n y para todo C() se tiene que $fv(C(P)) = fv(C(Q)) = \varnothing$, $C(P) \Downarrow n \Leftrightarrow C(Q) \Downarrow n$

6.- Sintaxis abstracta

Para hacer más comprensible la sintaxis textual, descrita anteriormente, de los ambientes, vamos a dar ahora una sintaxis visual que se corresponda con la anterior. Para ello, vamos a hacerlo con una metáfora de una oficina. Esta sintaxis nos va a permitir ver con mayor claridad el protocolo de seguridad desarrollado con cálculo de ambientes y que describiremos posteriormente con detalle.

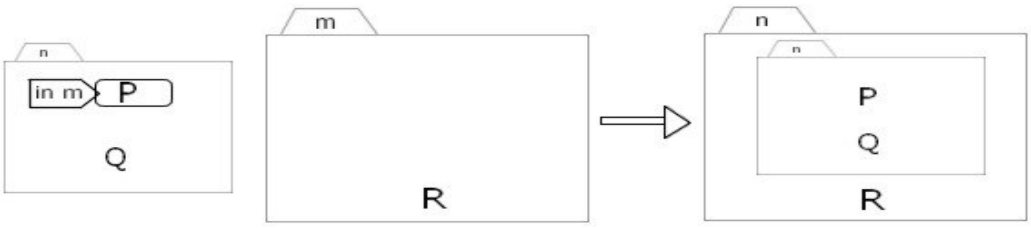
6.1 Representación de ambientes

En esta metáfora, un ambiente lo vamos a representar por una carpeta que guarda un contenido. Por lo tanto, cualquier cosa que podamos imaginar, estará o bien dentro o bien fuera de la carpeta. Las carpetas tienen un nombre, es decir, el nombre del ambiente, que se representa como una etiqueta en la carpeta, y en su interior podemos encontrar carpetas, por lo tanto, aquí tenemos la idea de ambientes anidados. Por último, también debemos decir que una carpeta nos la podemos llevar de la oficina a casa, es decir, las carpetas (ambientes) se pueden mover, no son algo estático.



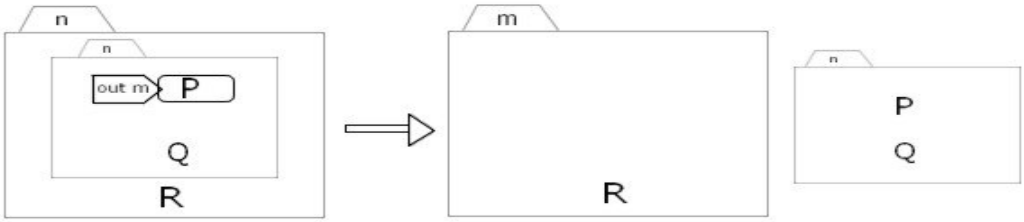
6.2.- Representación de la reducción de entrada

Veamos ahora siguiendo con nuestra metáfora, la reducción de entrada. Esta reducción, vista con la idea de carpetas, consistiría en tener dos carpetas: por un lado, una carpeta llamada n cuyo contenido es una habilidad de entrada en otra carpeta, m; y por otro lado una carpeta con etiqueta m. La reducción no es más que aplicar la habilidad de entrada de manera que lo que nos queda es una única carpeta llamada m cuyo contenido es el que tenía anteriormente más una carpeta que será n pero ya sin la habilidad de entrada. Visualmente tenemos:



6.3.- Representación de la reducción de salida

Ahora la situación inicial es la de una única carpeta que contiene otra carpeta con una habilidad de salida out m, siendo m la carpeta más externa. Si aplicamos esta habilidad, lo que tenemos son dos carpetas con el contenido que se muestra a continuación:

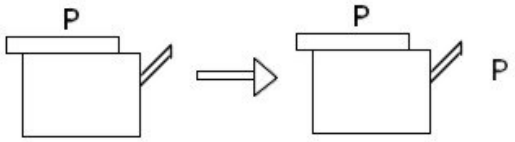


6.4.- Reducción de apertura

Igual que antes, partimos de una única carpeta que contiene una habilidad de apertura sobre una carpeta n, que se encuentra en la misma carpeta. La situación se muestra en la siguiente figura:

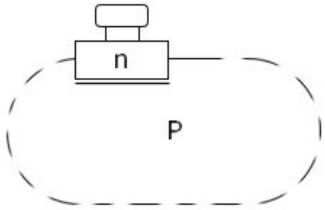
6.5.- Reducción de copia

La copia, en nuestra metáfora de una oficina con carpetas, la podemos ver como una fotocopiadora, de manera que teniendo esta fotocopiadora podemos hacer tantas copias como queramos de nuestra carpeta.



6.6.- Creación de nombres

La forma de crear un nombre la representamos con un sello que marque una carpeta con un nombre determinado.



7.- Extensiones al cálculo de ambientes básico.

Lo que hemos visto hasta ahora son los primeros pasos del cálculo de ambientes. Sin embargo, a partir de esta base podemos evolucionar sobre todo con la idea de proporcionar un sistema de tipos que nos facilite la codificación de sistemas seguros. Para ello, en esta sección vamos a centrarnos en cuatro posibles mejoras, como son la comunicación poliádica, una discusión sobre el tipo de movilidad que hemos estado usando, un simple sistema de tipos y grupos de ambientes.

7.1 Comunicación poliádica

Hasta ahora hemos tenido un tipo de comunicación que nos procuraba el intercambio de valores individuales, sin embargo, resulta mucho más cómodo la posibilidad de intercambiar tuplas de valores, lo que llamamos comunicación poliádica. Esta extensión nos facilita en gran manera la definición de un sistema de tipos que veremos en el apartado siguiente.

Esta extensión nos obliga únicamente al cambio de las acciones de entrada y de salida, de manera que lo que comuniquen estas acciones no sea sólo un valor sino una tupla.

7.2 Movilidad objetiva frente a movilidad subjetiva

La movilidad que hemos hemos venido describiendo para los ambientes, es aquella que conseguimos gracias a las primitivas in y out, y es lo que llamamos movilidad subjetiva, esto es, lo que hacen es “yo (ambiente) quiero moverme allí”. La

otra posibilidad que cabe pensar y que ya hemos introducido de manera rápida con el ejemplo de los canales en la sección 3.4, es la movilidad objetiva, “tú (ambiente) deberías moverte allí”. En el citado ejemplo hemos visto como podíamos codificar una primitiva nueva mv, con las primitivas de las que ya disponíamos. Ahora vamos a sugerir la introducción de una nueva primitiva enriqueciendo de esta manera el cálculo de ambientes. Esta nueva primitiva es la primitiva go, que tiene la forma go N.M[P] y consiste en activar el ambiente M[P] en la dirección a la que llegamos siguiendo el camino N, siempre que este camino sea posible.

A diferencia de in y out, la operación de go no mueve el ambiente en el que se encuentra esta operación. Podemos interpretar esta operación como trasladar código a una determinada dirección y una vez se encuentre allí, ejecutarlo.

La introducción de la movilidad objetiva nos va a ser de gran utilidad sobre todo a la hora de definir los tipos. Como todavía no tenemos la noción de los tipos, esto lo explicaremos mejor en el siguiente apartado.

7.3 Primer sistema de tipos simple

No vamos a avanzar mucho en cuanto a tipos se refiere en este apartado, dado que donde realmente veremos un sistema de tipos tampoco muy complejo pero bastante más potente es en el siguiente apartado.

Simplemente dejar constancia de la posibilidad de distinguir entre ambientes y habilidades a través de tipos. De esta manera, tendríamos únicamente dos tipos: Amb y Cap para designar a ambientes y habilidades respectivamente. Por otro lado, también es interesante distinguir el tipo de la información que se intercambia, de manera que con estas dos ideas tendríamos un pequeño sistema de tipo que consistiría en dos constructoras:

Amb[T]: describe el tipo de los ambientes que permiten el intercambio de información de tipo T.
Cap[T]: describe habilidades que pueden provocar el intercambio de información del tipo T en términos de la apertura de un subambiente en el ambiente actual.

El tipo T, es decir, el de la información a intercambiar puede tomar los valores Shh, lo que supone que no está permitido el intercambio, o bien una tupla formada por nombres o habilidades.

Este pequeño sistema de tipos nos permite controlar el intercambio de valores durante la comunicación, sin embargo, la mayor utilidad puede ser quizás el hecho de ver que el cálculo de ambiente sin tipos es tan expresivo como los procesos tipados u otros cálculos con funciones. Sin embargo, de momento nos quedamos en la garantía de que la comunicación está bien tipada, sin entrar en aspectos de movilidad de ambientes, que es algo que veremos en el siguiente apartado.

7.5 Generalización del tipo de ambientes mediante grupos.

Vamos a definir a continuación un sistema de tipos en el que se nos permita distinguir entre varios tipos de ambientes a través de lo que vamos a denominar grupos. La idea que subyace en todo esto es la de dar las mismas posibilidades con el cálculo de ambientes que tienen los programadores con los lenguajes de programación en los que los sistemas de tipos les permiten definir la claves de su código. Esto, como ya hemos dicho anteriormente es sobre todo útil por razones de seguridad.

De este modo, ahora lo que pretendemos es disponer de un abanico más amplio a la hora de definir el tipo de un ambiente. Por ejemplo, si queremos que el ambiente n tenga la posibilidad de entrar en el ambiente m, en principio, podríamos pensar en expresarlo diciendo que n tiene tipo canEnter(m). Sin embargo, con este tipo, nos surge el problema de la dependencia de un ambiente, es decir, el tipo canEnter(m) depende del ambiente m. Para solucionarlo, introducimos la idea de grupos, de manera que esta propiedad la formulamos de la siguiente manera:

m es un ambiente que pertenece al grupo G.
n es un ambiente que puede entrar en cualquier ambiente del grupo G.

De esta manera, los tipos que nos surgen son de la forma m:G y n:canEnter(G), donde ahora si que canEnter es un tipo genérico, ya que no depende únicamente de un ambiente, sino de otro tipo que es un grupo.

Dado que el cálculo de ambientes está pensado para el diseño de código móvil, la importancia de los grupos es considerable. Lo podemos ver pensando en el código de el típico sistema distribuido que se compone de varios nodos y hebras móviles. Con los grupos podríamos definir tipos tales como nodo, hebra, canal o paquete. Además podemos declarar propiedades como decir que este ambiente es una hebra y sólo puede cruzar ambientes que sean nodos. Y quizás, lo más importante de todo, que todas estas propiedades para cada uno de los ambientes estarían definidas estáticamente con lo que se reduce la posibilidad de errores intencionados o accidentales a la hora de codificar usando cálculo de ambientes.

7.6 Sintaxis y semántica operacional

La introducción de la todas estas extensiones nos obliga a realizar pequeñas modificaciones tanto en la sintaxis como en la semántica operacional explicadas en el apartado 4 así como a realizar adiciones. La nueva sintaxis es la siguiente:

Expresiones y procesos	
P, Q ::=	procesos
(νG)P	creación de grupos
(νn:W)P	restricción
0	inactividad
P | Q	composición
!P	replicación
M [P]	ambiente
M.P	acción de habilidad

$(x{:}W_1,\ldots,x_k{:}W_k).P$ $\langle M_1,\ldots,M_k\rangle$ go N.M[P]	acción de entrada acción asíncrona de salida movimiento objetivo
M ::= n in M out M open M ε M.M'	habilidades nombre entrar en M salir de M abrir M null path

Por otro lado, aparecen nuevas congruencias estructurales y reducciones modificándose además otras. Mostramos las tabla con las nuevas y las modificadas, el resto se pueden ver en las tablas de los apartados 3.3 y 4.3:

Reducciones	
$\langle M_1, \ldots, M_k\rangle \mid (x_1{:}W_1, \ldots, x_k{:}W_k).P \rightarrow P\{x_1 \leftarrow M_1, \ldots, x_k \leftarrow M_k\}$	(Red Comm)
go (in m.N).n[P] | m[Q] → m[go N.n[P] | Q]	(Red Go In)
m[go (out m.N).n[P] | Q] → go N.n[P] | m[Q]	(Red Go Out)
$P \rightarrow Q \Rightarrow (\nu n{:}W)P \rightarrow (\nu n{:}W)Q$	(Red Res)
$P \rightarrow Q \Rightarrow (\nu G)P \rightarrow (\nu G)Q$	(Red Gres)

Congruencias estructurales	
$P \equiv Q \Rightarrow (\nu n{:}W)P \equiv (\nu n{:}W)Q$	(Struct Res)
$P \equiv Q \Rightarrow (\nu G)P \equiv (\nu G)Q$	(Struct G Res)
$P \equiv Q \Rightarrow (x_1{:}W_1, \ldots, x_k{:}W_k).P \equiv (x_1{:}W_1, \ldots, x_k{:}W_k).P$	(Struct Input)
$P \equiv Q \Rightarrow$ go N.M[P] ≡ go N.M[Q]	(Struct Go)
$n_1 \neq n_2 \Rightarrow (\nu n_1{:}W)(\nu n_2{:}W)P \equiv (\nu n_2{:}W)(\nu n_1{:}W)P$	(Struct Res Res)
$n_1 \notin fn(P) \Rightarrow (\nu n{:}W)(P \mid Q) \equiv P \mid (\nu n{:}W)Q$	(Struct Res Par)
$n \neq m \Rightarrow (\nu n{:}W)m[P] \equiv m[(\nu n{:}W)P]$	(Struct Res Amb)
$(\nu G_1)(\nu G_2)P \Rightarrow (\nu G_2)(\nu G_1)P$	(Struct GRes GRes)
$G \notin fg(W) \Rightarrow (\nu G)(\nu n{:}W)P \equiv (\nu n{:}W)(\nu G)P$	(Struct Gres Res)
$G \notin fg(P) \Rightarrow (\nu G)(P \mid Q) \equiv P \mid (\nu G)Q$	(Struct GRes Par)
$(\nu G)m[P] \Rightarrow m[(\nu G)P]$	(Struct GRes Amb)
$(\nu n{:}W)0 \equiv 0$	(Struct Zero Res)
$(\nu G)0 \equiv 0$	(Struct Zero GRes)
go ε.M[P] ≡ M[P]	(Struct Go ε)

go (ε.M).N[P] ≡ go M.N[P]	(Struct Go ε .)
go (M.ε).N[P] ≡ go M.N[P]	(Struct Go . ε)
go ((M.M').M'').N[P] ≡ go (M.(M'M'')).N[P]	(Struct Go . Assoc)

Sin entrar mucho en los tipos, vamos a dar una tabla en la que mostramos los tipos introducidos hasta ahora:

Tipos	
W::= G[T] Cap[T]	Tipo de mensajes Nombre para ambientes del grupo G Habilidad
T::= Shh $W_1 \times ... \times W_k$	Tipo de intercambio Sin intercambio Tupla de intercambios

8.- Cálculo de ambientes tipado

Hasta ahora hemos visto un cálculo de ambientes en el que un ambiente podía representar conceptos diversos como el nodo de una red, un canal o un agente software. Con la introducción de un modelo tipado para los ambientes se pretende distinguir entre ambientes dependiendo de sus característica, bien porque un ambiente sea fijo, o porque no pueda ser eliminado por su entorno... De lo que se trata en definitiva es de intentar proporcionar un sistema de tipos que permita controlar el movimiento de los ambientes sobre otros ambientes. Para el protocolo de seguridad realizado con cálculo de ambientes no se ha utilizado este modelo con tipos, sin embargo, si que puede resultar de utilidad a la hora de desarrollar modelos de seguridad el no quedarnos simplemente en el modelo de ambientes visto hasta ahora.

En el apartado anterior ya nos hemos introducido un poco en un sistema de tipos para el cálculo de ambientes bastante sencillo. En este sistema de tipos distinguíamos entre ambientes según pertenecieran a un determinado grupo, pudiendo además controlar la comunicación. Sin embargo, no hemos hablado de la movilidad para nada.

Ahora pretendemos extender esta primera idea básica con la introducción de tres atributos: un atributo que nos permita el control de apertura (atributo de cerrojo), otro que nos permita el control sobre la movilidad objetiva (atributo de movilidad objetiva) y por último, otro que nos permita control sobre la movilidad subjetiva (atributo de movilidad subjetiva).

8.1 Control de apertura

La operación de apertura es una de las más delicadas en el cálculo de ambientes, dado que permite a un cliente tener código dentro de un servidor, pudiéndose dar una de las siguientes consecuencias:

- Que el cliente tiene la posibilidad de establecer comunicaciones dentro del servidor e incluso mover el servidor.
- Que el servidor es capaz de examinar el contenido del cliente, pudiendo ver los mensajes que mandados por este y pudiendo ver sus subambientes.

Por estas razones, es deseable tener control de en que ambientes podemos usar la habilidad open. Para ello, vamos a formalizar un atributo para los tipos de ambientes, el atributo de cerrojo, que puede tomar dos valores, bloqueado o no bloqueado, de manera que podemos indicar si un ambiente puede o no ser abierto.

Notación:

- Tipo para ambientes bloqueados $G^{\bullet}[T]$
- Tipo para ambientes no bloqueados $G^{\circ}[T]$

Donde T es el tipo de la información que se intercambia.

8.2 Control de movilidad

A la hora de pensar en un atributo para los tipos que nos controle la movilidad, debemos pensar en los dos tipos de movilidad de los que disponemos.

8.2.1 Control de movilidad subjetiva

Al igual que hicimos en el caso anterior para controlar la apertura de ambientes, vamos a enriquecer el sistema de tipos de manera que podamos controlar cuando un ambiente puede atravesar ambientes de un determinado grupo mediante movimientos subjetivos, es decir, con las primitivas in y out.

Notación:

Tipo para ambientes móviles $G[\searrow T]$
Tipo para ambientes no móviles $G[\forall T]$

Los procesos con efectos del tipo $[\searrow T]$ o $[\forall T]$ deben ejecutarse con ambientes del tipo correspondiente. Una habilidad puede producir ahora efectos de comunicación o de movilidad teniendo tipo $Cap[\forall T]$ y $Cap[\searrow T]$ respectivamente. Las habilidades in o out directamente producen efectos de movilidad mientras que open puede producir efectos de los dos tipos (la habilidad de open por si sola en realidad no produce ninguno de los dos efectos, si bien indirectamente si puede provocarlos).

8.2.2 Control de movilidad objetiva

Y ya por último, vamos a introducir un atributo que nos permita controlar la movilidad objetiva, es decir, que ambientes pueden atravesar ambientes de un determinado grupo mediante movimientos objetivos, es decir, con la primitiva go.

Notación:

Tipo para ambientes móviles	$G^{\searrow}$ [T]
Tipo para ambientes no móviles	$G^{\forall}$ [T]

8.3 El sistema de tipos

A continuación mostramos una tabla donde describimos la sintaxis de estos tipos:

Tipos	
Y::=	Atributo de cerrojo
•	Bloqueado
◦	No bloqueado
Z::=	Atributo de movilidad
$\searrow$	Móvil
$\forall$	Estático
W::=	Tipo de mensajes
$G^{Y}\{F\}^{Z'}\{F'\}[^{Z}\{F''\}T]$	Nombre de ambiente
$Cap[^{Z}\{F\}T]$	Habilidad
T::=	Tipo de intercambio
Shh	Sin intercambio
$W_1 \times ... \times W_k$	Tupla de intercambios

Donde F, F' y F'' son conjuntos de grupos de ambientes, y únicamente aparecen si el tipo es no bloqueado o móvil, de manera que indique que grupos pueden abrirle y que grupos puede atravesar.

Las reglas de tipos se describen a continuación. Tenemos tres tipos de sentencias: la primera para la construcción de entornos bien formados, la segunda para los tipos de mensajes y la tercera para los efectos de procesos.

Sentencias	
$E \vdash \Diamond$	Buen entorno
$E \vdash M : W$	Buena expresión de mensaje de tipo W
$E \vdash P : {}^{Z}T$	Buen proceso con movilidad Z e intercambio T

Buenos entornos	
(Env ∅)	$\dfrac{}{\varnothing \vdash \Diamond}$
(Env n)	$\dfrac{E \vdash \Diamond, n \notin dom(E)}{E, n : W \vdash \Diamond}$

Buenas expresiones de tipo W	
(Exp n)	$\dfrac{E', n:W, E'' \vdash \Diamond}{E', n:W, E'' \vdash n:W}$
(Exp ε)	$\dfrac{E \vdash \Diamond}{E \vdash \in: Cap\left[^{Z}\{F\}T\right]}$
(Exp .)	$\dfrac{E \vdash M : Cap\left[^{Z}\{F\}T\right], E \vdash M' : Cap\left[^{Z}\{F\}T\right]}{E \vdash M.M' : Cap\left[^{Z}\{F\}T\right]}$
(Exp In)	$\dfrac{E \vdash n : G^{Y}\{F\}^{Z'}\{F'\}\left[^{Z}\{F''\}T\right]}{E \vdash in \quad n : Cap\left[\rightarrow\{F'''\}T'\right]}$
(Exp Out)	$\dfrac{E \vdash n : G^{Y}\{F\}^{Z'}\{F''\}\left[^{Z}\{F''\}T\right]}{E \vdash \text{out n} : Cap\left[\rightarrow\{F'''\}T'\right]}$
(Exp Open)	$\dfrac{E \vdash n : G^{\circ}\{F\}^{Z'}\{F'\}\left[^{Z}\{F''\}T\right]}{E \vdash open \quad n : Cap\left[^{Z}\{F'''\}T\right]}$

Buenos procesos con movilidad Z e intercambio T	
(Proc Action)	$\dfrac{E \vdash M : Cap\left[^{Z}\{F\}T\right], E \vdash \text{P}:^{Z}\{F\}T}{E \vdash M.P:^{Z}\{F\}T}$
(Proc Amb)	$\dfrac{E \vdash \text{M} : \text{G}^{\text{Y}}\{F\}^{Z''}\{F'\}\left[^{Z}\{F''\}T\right], E \vdash \text{P} :\, ^{Z}\{F''\}T}{E \vdash M\left[P\right]:^{Z'}\{F'''\}T'}$
(Proc Res)	$\dfrac{E, n : \text{G}^{\text{Y}}\{F\}^{Z''}\{F'\}\left[^{Z}\{F''\}T\right] \vdash \text{P} :\, ^{Z'}\{F'''\}T'}{E \vdash (\text{vn} : \text{G}^{\text{Y}}\{F\}^{Z''}\{F'\}\left[^{Z}\{F''\}T\right])P:^{Z'}\{F'''\}T'}$
(Proc Par)	$\dfrac{E \vdash \text{P} :\, ^{Z}\{F\}T, E \vdash \text{Q} :\, ^{Z}\{F\}T}{E \vdash P \mid Q :\, ^{Z}\{F\}T}$
(Proc Repl)	$\dfrac{E \vdash \text{P} :\, ^{Z}\{F\}T}{E \vdash !\text{P} :\, ^{Z}\{F\}T}$
(Proc Input)	$\dfrac{E, \text{n}_1 : W_1, ..., \text{n}_n : W_n \vdash P :\, ^{Z}\{F\}W_1 \times ... \times W_k}{E \vdash (\text{n}_1 : W_1, ..., \text{n}_n : W_n).P :\, ^{Z}\{F\}W_1 \times ... \times W_k}$
(Proc Output)	$\dfrac{E \vdash \text{M}_1 : W_1, ..., E \vdash \text{M}_k : W_k}{E \vdash \langle M_1, ..., M_k \rangle :\, ^{Z}\{F\}W_1 \times ... \times W_k}$
(Proc Zero)	$\dfrac{E \vdash \Diamond}{E \vdash 0 :\, ^{Z}\{F\}T}$
(Proc Go)	$\dfrac{E \vdash \text{N}:\text{Cap}\left[^{Z'}\{F\}S\right], E \vdash \text{M}:\text{G}^{\text{Y}}\{F\}^{Z'}\{F\}\left[^{Z}\{F''\}T\right], E \vdash \text{P}:^{Z}\{F''\}T}{E \vdash \text{goN.M}\left[\text{P}\right]:^{Z''}\{F'''\}T''}$

Vamos a ilustrar los tipos introducidos con un ejemplo sencillo. Sean los siguientes ambientes ejecutándose en paralelo:

a[go (out a.in b).p[⟨M⟩]] | b[open p.(x:W).P]

A primera vista, vemos que lo que este proceso hace es ejecutar en primer lugar el go del ambiente a, con lo que seguimos el camino out a.in b. Una vez hecho esto tenemos el ambiente p[⟨M⟩] dentro del ambiente b. por lo que se puede hacer el open p y abrir por lo tanto y ya por último, lo único que podemos hacer es consumir la entrada (x).P con la salida ⟨M⟩. Vamos a tratar de ver el tipo de este proceso. Vamos a trabajar con las siguientes suposiciones: a es un ambiente perteneciente al grupo F, bloqueado y estático tanto objetiva como subjetivamente; p es un ambiente del grupo G no bloqueado para ambientes del grupo F, móvil objetivamente, pero fijo subjetivamente; b es un ambiente del grupo F, bloqueado y estático tanto objetiva como subjetivamente; W es el tipo del mensaje M. Definiendo esto formalmente, partimos de las siguientes hipótesis:

h1. E ⊢ a: Ch•∀ [∀Shh]
h2. E ⊢ p: Pk◦{Ch}↘{Ch} [∀W]
h3. E ⊢ b: Ch•∀ [∀W]
h4. E ⊢ M: W
h5. E, x: W ⊢ P: ∀W

Podemos demostrar que E ⊢a[go (out a.in b).p[⟨M⟩]] | b[open p.(x:W).P]: ≙Shh. Lo mostramos a continuación:

$$\text{(ProcPar)}\ \dfrac{\text{(ProcAmb)}\ \dfrac{\text{cierto por h1.} \quad \text{Continua en (1)}}{E \vdash a[go(out\ a.in\ b).p[\langle M\rangle]] : \forall Shh} \quad \text{(ProcAmb)}\ \dfrac{\text{continua en (3)}}{E \vdash b[open\ p.(x:W).P] : \forall Shh}}{E \vdash a[go(out\ a.in\ b).p[\langle M\rangle]] \mid b[open\ p.(x:W).P] : \forall Shh}$$

(1)

$$\text{(Proc Go)}\ \dfrac{\text{continua en (2)} \quad \text{cierto por h2.} \quad \text{(Proc Output)}\ \dfrac{\text{cierto por h4.}}{E \vdash M : \forall W}}{E \vdash go(out\ a.in\ b).p[\langle M\rangle] : \forall Shh}$$

(2)

$$(\text{Exp .})\dfrac{(\text{Exp Out})\dfrac{\dfrac{\text{cierto por h1.}}{\text{E} \vdash \text{a} : \text{Ch} \bullet \forall[\forall \mathit{Shh}]}}{\text{E} \vdash \text{out a} : \text{Cap}[\rightarrow \mathit{Shh}]} \quad (\text{Exp In})\dfrac{\dfrac{\text{cierto por h2.}}{\text{E} \vdash \text{b} : \text{Ch} \bullet \forall[\forall \text{W}]}}{\text{E} \vdash \text{in b} : \text{Cap}[\rightarrow \mathit{Shh}]}}{\text{E} \vdash \text{out a.in b} : \text{Cap}[\rightarrow \mathit{Shh}]}$$

(3)

$$(\text{Proc Amb})\dfrac{\dfrac{\text{cierto por h2.}}{\text{E} \vdash \text{b} : \text{Ch} \bullet \forall[\forall \text{W}]} \quad (\text{Proc Action})\dfrac{(\text{Exp Open})\dfrac{\dfrac{\text{cierto por h3.}}{\text{E} \vdash \text{p} : \text{Pk}^{\text{o}}\{\text{Ch}\} \rightarrow \{\text{Ch}\}[\forall W]}}{\text{E} \vdash \text{open p} : \text{Cap}[\forall W]} \quad (\text{Proc Input})\dfrac{\text{continua en (4)}}{\text{E} \vdash (\text{x} : \text{W}).\text{P} : \forall W}}{\text{E} \vdash \text{open p}.(\text{x} : \text{W}).\text{P} : \forall W}}{\text{E} \vdash \text{b}[\text{open p}.(\text{x} : \text{W}).\text{P}] : \forall \mathit{Shh}}$$

(4)

$$(\text{Proc Input})\dfrac{\dfrac{\text{cierto por h5.}}{\text{E}, \text{x} : \text{W} \vdash \text{P} : \forall W}}{\text{E} \vdash (\text{x} : \text{W}).\text{P} : \forall W}$$

8.4 Sistemas de tipos posteriores. Breve mención

Por supuesto, este sistema de tipos descrito, es bastante simple, y cabría pensar en sistemas de tipos mucho más complejos. Sin embargo, el objetivo de este apartado no es ese, sino simplemente el mencionar la existencia de trabajos posteriores realizados por Luca Cardelli, Andrew D. Gordon y Giorgio Ghelli, (que son precisamente las personas que realzaron el artículo del cual se ha extraído la información del sistema de tipos anteriormente mostrado), donde se describe un sistema de tipos ligeramente diferente. No entraremos a describirlo en detalle, ya que sería bastante redundante. Existen algunas variaciones en las sentencias y reglas de tipos, además de cambios en la notación ya que no indicamos tipos estáticos o no bloqueados, entendiéndose que si no es bloqueado es bloqueado y que si no es estático, es móvil. Otra diferencia es el control de bloqueo, que en este sistema de tipos indica no por que grupos puede ser abierto, sino que grupos le pueden abrir.

9.- Sistema de tipos polimórfico

Torben Amtoft, Assaf J. Kfoury y Santiago M. Preicas-Geertsen, todos ellos de la Universidad de Boston, proponen en un trabajo de Diciembre del 2000, un sistema de tipos para el cálculo de ambientes polimórfico. Sin entrar en detalles de este sistema de tipos, vamos a explicar la motivación que tiene con una serie de ejemplos.

Básicamente, son cuatro los casos para los que nos interesa tener un sistema de tipos polimórfico, es decir, tipos que no son estáticos. A continuación ilustramos con un ejemplo cada uno de estos casos.

9.1 Caso 1

Veamos el siguiente proceso:

p[in r.<impar, 3>] | q[in r.<not, true>] | r[(f, x).n[<f x> | P] | open p | open q]

Este proceso tiene tres ambientes, p, q y r en paralelo, además de un ambiente, n, dentro de r. Tanto p como q pueden moverse dentro de r y una vez dentro, pueden abrirse para poder utilizar sus salidas. El tipo del par de entrada (f, x) que se encuentra dentro de r puede ser bien (int → bool, int) o (bool → bool, bool) dependiendo de que la salida sea <impar, 3> o <not, true>. Este es el tipo de polimorfismo que más se ha estudiado y básicamente coincide con el polimorfismo del lenguaje ML.

9.2 Caso 2

El procedimiento para ilustrar este caso es una pequeña variación del caso anterior:

p[in r.<3, 2>] | q[in r.<3.6, 5.1>] | r[(x, y).n[<mult(x, y)> | P] | open p | open q]

En el proceso anterior, mult es una función que multiplica dos número reales: mult: (real, real) → real. Esto es así pues <3, 2> es de tipo (int, int) que es un subtipo de (real, real). Este es el tipo de polimorfismo típico de los lenguajes orientado a objetos.

9.3 Caso 3

Consideramos ahora otro procedimiento:

n[<true, 5> | <5, 6, 3.6> | (x, y).P | (x, y, z).Q]

Tenemos dos salida con distinta aridad (<true, 5> con aridad 2 y <5, 6, 3.6> con aridad 3). Esto nos permite determinar el tipo de los procesos (x, y) y (x, y, z), dado que estos pueden ejecutarse si las entradas son de tipo (bool, int) y (int, int, real) respectivamente. Este es el tipo de polimorfismo que se conoce con el nombre de polimorfismo de aridad.

9.4 Caso 4

Este último caso es el más delicado y el menos estudiado en cuanto a tipos de intercambio de datos que pueden variar con el tiempo. Veamos el siguiente procedimiento:

m[<7> | (x).open n.<x = 42> | n[(y).P]]

En principio, el ambiente m tiene como tipo int, sin embargo, al consumir la salida <7>, n se abre y m pasa a tener tipo bool debido al test de igualdad <x = 42>.

Esto es lo que se llama polimorfismo metódico y es el más raro y difícil de comprender de los cuatro.

10.- Trabajos encontrados

El cálculo de ambientes es bastante reciente y realmente la mayoría del trabajo que se puede desarrollar en torno a ello consiste en trabajo de investigación. Sin embargo, no hace falta buscar mucho por internet para encontrar ciertos trabajos ya desarrollados como implementaciones o simuladores. Algunas de las cosas más destacadas que hemos podido encontrar son las siguientes.

10.1 Simulador de cálculo de ambientes

Se trata de un simple simulador gráfico de cálculo de ambientes realizado en java. El autor de este simulador es Damien Pous y lo podemos encontrar en la dirección http://www-sop.inria.fr/mimosa/ambicobjs/.

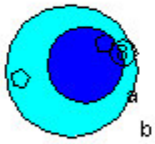
La representación gráfica difiere de la que hemos comentado anteriormente. Cada ambiente está representado por un círculo de color cuyo tamaño depende del contenido y cuyos limites actúan de frontera con el exterior.

Nos permite escribir ambientes con las habilidades in, out y open. Lo que no se nos permite es el ocultamiento, por lo que se nos hace bastante complicado simular cualquier protocolo de seguridad.

A continuación mostramos unas capturas de pantalla de una simulación simple realizada con este simulador. La simulación consta de un ambiente a con la siguiente expresión: `a[in b.sleep 100.out b]; y un ambiente b = b[].`

Esta es la primera captura de pantalla nada más crear ambos ambientes:

En la segunda captura, vemos que ya se ha ejecutado el in b y ya tenemos por tanto al ambiente a dentro del ambiente b.



Por último, tras los 10 segundo de espera, el ambiente a sale de b.

Como ya hemos dicho, al carecer del ocultamiento, es difícil que se puedan crear protocolos de seguridad. Esta simulación sirve más bien para entender un poco mejor los ambientes teniendo algo más visual.

10.2 Implementación de cálculo de ambientes

La única implementación, en términos de código, encontrada, es una realizada por unos alumnos de una universidad japonesa desarrollada en Java. Lo cierto es que no tenemos muchos datos de esta implementación, dado que no existía ningún manual y los comentarios escritos en el programa estaban en japonés (suponemos, es decir, bien pudieran estar en chino), de hecho, ni siquiera hemos conseguido llegar a ponerla en

funcionamiento. Sin embargo, la intención de poner está implementación como uno de los trabajos encontrados, es más bien el mostrar la idea de que aunque sea un trabajo muy moderno y la mayoría del trabajo este encaminado a labores de investigación, si que existen ya implementaciones del mismo. De hecho, el realizar una implementación es en cierto modo bastante sencillo.

10.3 Agentes móviles PESOS

La universidad de Melbourne está desarrollando un sistema de agentes móviles bajo cálculo de ambientes llamado PESOS (Portable Environment Small Operating System). Para este proyecto están usando el cálculo de ambientes básico, simplemente con comunicación y movilidad. Puede que no sea un trabajo realmente importante, sin embargo, lo mencionamos para indicar que la mayoría de las "críticas" que hacen con respecto al cálculo de ambientes, son problemas que se pueden solucionar mediante el sistema de tipos descrito los últimos apartados.

10.4 Trabajos de investigación

No mencionaremos ninguno en concreto, pero debemos mencionarlo como trabajos encontrados, ya que en realidad, la gran mayoría del avance que se hace en cálculo de ambientes hoy en día está orientado a este sentido. No hace falta buscar mucho por Internet para encontrar un gran número de artículos acerca de cálculo de ambientes. De hecho, el cálculo de ambientes nace como un proyecto de investigación de Microsoft.

11.- Conclusión

El cálculo de ambientes es un trabajo bastante reciente por lo que no está realmente desarrollado. Sin embargo con esta base se pueden hacer metodologías , librerías y lenguajes de programación. A modo de conclusión, vamos a tratar ahora una serie de discusiones sobre las posibilidades que tiene ahora y en un futuro el cálculo de ambientes.

11.1 Seguridad

Dado que la movilidad se representa como el traspaso de fronteras, cualquier movimiento debe obtener permiso por adelantado de la región a la que quiere pasar, así que la seguridad es una propiedad inherente. En la introducción hemos descrito un poco como funciona actualmente Internet, esto es, actualmente con los firewalls Internet se parte en dominios administrativos, de tal manera que si queremos ir desde un punto A hasta un punto B debemos primero obtener permiso para salir de nuestro propio dominio, obtener permiso para entrar en otro dominio y obtener permiso para poder ejecutarnos. Vemos que esta estructura se identifica muy bien con el cálculo de ambientes, por lo que el primer objetivo que era el de capturar las nociones de dominios, movilidad sobre estos y autorizaciones se consigue.

11.2 Complejidad

Los ambientes se definen recursivamente obteniendo una estructura jerárquica y moviéndose como un entorno anidado que lleva consigo sus ambientes y subentornos. Con este modelo ampliamente distribuido, los cálculos pueden llevarse a aplicaciones más complejas. Esta propiedad encaja perfectamente con la arquitectura de Internet donde los dominios administrativos están organizados en una jerarquía de mucho niveles. Tenemos accesos en ordenadores personales, redes de área local, de área medio o de grandes áreas. La autorización para movernos se requiere en todos y cada uno de los niveles de esta jerarquía por los que el agente navega.

11.3 Eficiencia en la comunicación

Para transmitir un valor a de un ambiente n a otro k, el mensajero, m, debe cruzar la frontera y luego dejar el mensaje mediante una comunicación local. En este paradigma de comunicación lo único móvil son los ambientes lo cual se viene a traducir por una gran eficiencia en la comunicación, aparte de una simplicidad en el control distribuido, tolerancia a fallos y alto poder de concurrencia.

11.4 Implementabilidad

El modelo del cálculo de ambientes está altamente abstraído, de tal manera que no puede ser aplicado directamente a sistemas o aplicaciones particulares. Sin embargo esto hace que la implementación del cálculo de ambientes no sea muy compleja, aunque sí su utilización.

12.- Representación de un protocolo mixto mediante el calculo de ambientes.

Descripción del protocolo de comunicación publico-privado:

Para entender lo que es un protocolo de comunicación mixto lo primero es entender que son la encriptación simétrica y asimétrica.

La encriptación simétrica, es la encriptación el la cual se encripta y se desencripta usando la misma clave, se descifra realizando la operación inversa a la de cifrado con valor de la clave igual a la de cifrado. Por lo tanto quien tiene la clave puede descifrar y cifrar. Por ejemplo cifrar podría ser sumar un cierto numero al valor del carácter y descifrar seria restar ese mismo numero.

Por el contrario las claves asimétricas, en vez de usar una misma clave tanto para cifrar como para descifrar, creamos un par de claves para cada sistema, una privada que no saldrá del sistema y una publica que se publicará para aquellos que quieran comunicarse con el sistema. La clave publica y la privada son complementarias, realizamos la misma operación sobre el texto tanto para cifrar como para descifrar, Los cifrados hecho con la clave privada solo se decodifican con la publica y viceversa. Lo mas importante es que el hecho de poseer la clave publica no permite decodificar mensajes cifrados con la clave publica.
Un cifrado con la clave pública de un sistema consigue que solo el sistema destinatario del mensaje sea capaz de leer el contenido, si embargo, un cifrado con la clave privada hace que todos los que tengan la clave publica puedan leer el mensaje (es decir todo el mundo por que la clave pública es pública) pero asegura dos cosas, primera que el

Comparativa de los cifrados simétricos, con los cifrados asimétricos:

Los cifrados simétricos cifran y descifran a mayor velocidad, menor coste y son más difíciles de cripto-analizar (obtener la clave a partir del texto cifrado), pero como desventaja esta la dificultad del intercambio de claves para establecer la comunicación.

Los cifrados simétricos, son más complejos, más lentos pero tienen X grandes ventajas:

1. El intercambio de claves es trivial, ya que podemos dar sin problema nuestra clave pública ya que es hecho de que alguien que no queremos conozca nuestra clave publica no afecta a la seguridad de la comunicación.
2. Podemos exigir y asegurar una identificación del emisor. Ya que si alguien cifra con su clave privada, es decir el texto se puede descifrar con la clave publica del mismo. Solo puede provenir del dueño de la clave.
3. Como cada sistema tiene su par publico-privado para que un atacante se entere de toda la comunicación debe de cripto-analizar dos claves, las claves privadas de lo sistemas que se comunican.

La forma que tenemos para aprovechar la velocidad de las claves simétricas y las propiedades de las claves asimétricas son los protocolos de comunicación público-privado.

La idea del protocolo mixto es que para la comunicación entre los sistemas usaremos una clave simétrica, pero para identificar los extremos y transmitir la clave simétrica usaremos los pares de claves asimétricas.

12.1.- Descripción paso a paso del protocolo mixto:

Condiciones iniciales:

- Tenemos un servidor con sus claves publica y privada, junto con las claves públicas del resto de sistemas(‘A’ y ‘B’).
- Los sistemas tienen sus pares de clave publico privada y la clave pública del servidor.

Pasos:

1. El sistema ‘A’ quiere comunicarse con ‘B’ y para eso obtiene del servidor la clave publica de ‘B’.
2. A envía una solicitud de comunicación a ‘B’ mediante un mensaje que contiene su identificador como sistema, cifrado con la clave publica de ‘B’ (solo lo podrá ver ‘B’)
3. ‘B’ lee el mensaje y si quiere comunicarse con el otro sistema obtiene de ‘A’ la clave publica de ‘A’
4. ‘B’ manda un mensaje con la clave simétrica que propone para la comunicación codificada primero con la clave privada de ‘B’ (la firma) y después con la publica de ‘A’ para que solo ‘A’ lo pueda leer.
5. ‘A’ recibe la clave y devuelve, cifrado con la clave publica de ‘B’ la clave que recibió que servirá como confirmación.
6. ‘B’ recibe la confirmación que indica que le otro acepto la conexión y fin.

Notas sobre la seguridad del protocolo:

Estudiemos la seguridad de los datos paso a paso.

- La solicitud de la clave publica de ‘B’ desde ‘A’ al servidor:
  - El mensaje para pedir al servidor la clave no necesita cifrado ya que cualquiera puede tener la clave.
  - La contestación del servidor con la clave, estará cifrado con la clave privada del servidor para identificar al emisor es decir al servidor y asegurarnos de que la clave es la correcta.

- ‘A’ envía a ‘B’ el mensaje de solicitud de la conexión
  - El mensaje contiene el nombre como se identifica el sistema, cifrado con la clave publica de ‘B’, esto hace que nadie sepa que va a iniciar una comunicación ni con quien, aunque podría ser un mensaje sin encriptar ya que la confirmación del solicitante se realizará en el paso 5
- ‘B’ Obtiene la clave publica del sistema cuyo nombre estaba en la solicitud de conexión (‘A’).
  - Con esta clave protegeremos el envió de la clave simétrica para el canal de comunicación y para confirmar que el nombre recibido del

emisor corresponde con el emisor, por que hemos obtenido de forma segura la clave publica que corresponde a esa cable y solo el sistema con ese nombre tiene la clave privada para decodificarla.

- 'B' Manda la clave simétrica del canal que queremos establecer con 'A' cifrada con la clave privada de 'B' y con la pública de 'A'.
  - Al cifrar con la clave privada de 'B' autentificamos nuestra identidad a 'A', además aseguramos que nadie va poder modificar la clave que enviamos
  - El cifrado con la clave publica de A consigue que solamente A pueda leer el mensaje.
- 'A' manda la misma clave que recibió de B cifrada con la clave publica de 'B'
  - Como solamente el dueño de la clave privada correspondiente a la clave publica de 'A' puede leer la clave simétrica propuesta (es decir solo 'A') el hecho de conocer la clave simétrica es una forma de autentificación.
  - Además para que al contestar nadie pueda conocer la clave que se usará, la ciframos con la clave publica de 'B'.
- Cuando 'B' recibe la confirmación de 'A' el canal se forma con la clave simétrica que se estableció. La confirmación no tiene porque estar cifrada.

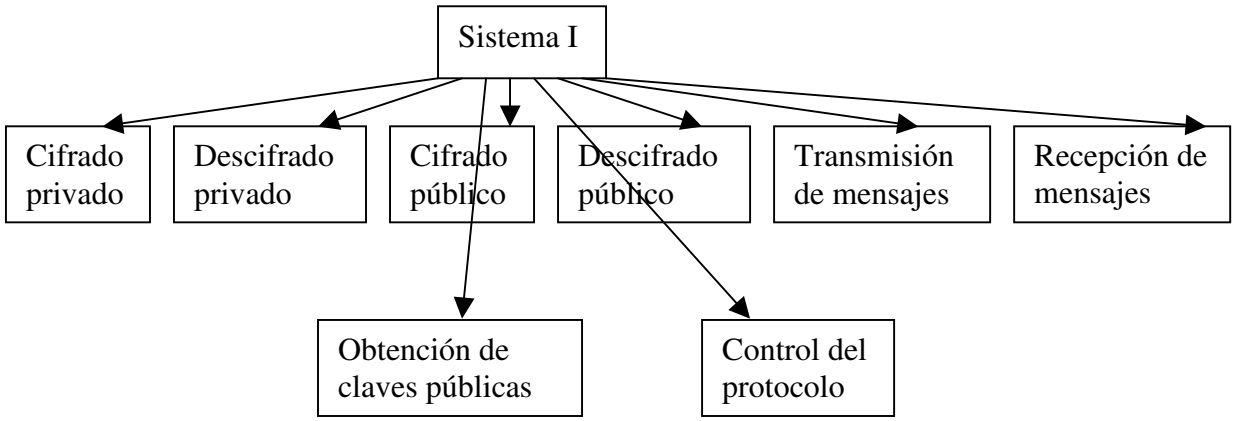
Este protocolo asegura que si se forma el canal se forma de forma segura, pero la ausencia de hand-shaking hace que si se pierden mensajes, el protocolo no termine y no se pueda crear el canal.

Cuando este protocolo termina los dos sistemas tienen una clave común con la cual se pueden comunicar de forma segura, la comunicación a parir de ese punto es trivial y no la trataremos aquí.

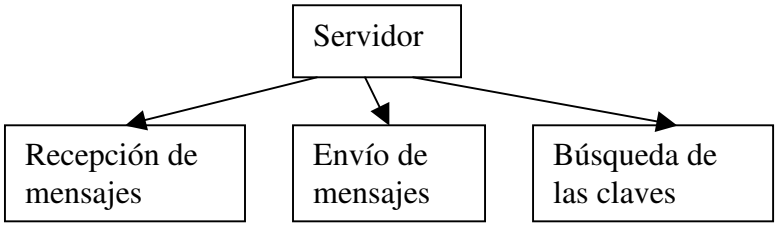
13.- Descripción de un protocolo de comunicación mixto con el calculo de ambientes.

Lo primero para realizar una descripción con el calculo de ambientes es numerar y definir que funcionalidades diferentes tienen los distintos sistemas, cada sistema será un proceso que en calculo de ambientes equivaldrá a un ambiente, y cada funcionalidad de cada sistema será otro proceso dentro del proceso general, que en calculo de ambientes se representará como un ambiente dentro de otro mas grande que los englobe (el sistema).

Vamos a indicar las distintas funcionalidades de cada sistema organizadas de forma jerárquica mediante un diagrama de árbol.



Estos son los procesos básicos para un sistema transmisor / receptor, pero no son los procesos para el sistema servidor, que se encarga de la distribución firmada de claves publicas de los distintos sistemas.



En realidad el servidor lo único que hace es atender a un tipo de mensajes de solicitud de claves de un sistema Y, desde un cierto ambiente X y devuelve firmado la clave.

Suponemos que todos los sistemas con sus claves públicas están registradas en el servidor al principio de la comunicación.

Una vez que tenemos descritos de forma general las distintas funcionalidades de cada sistema, vamos a detallarlas una a una y a representarlas mediante el calculo de ambientes.

Para describir el sistema de una forma incremental vamos a ir construyendo modularmente el sistema haciendo así más fácil demostrar la seguridad.

13.1.- Descripción del cifrado y descifrado simétrico.

Esta parte la vamos a obviar, ya que no llegaremos a usarla, pero el cifrado y descifrado simétrico es trivial en ambientes, de la siguiente manera.

Para cifrar un mensaje o lo que sea con una clave simétrica, lo conseguimos envolviendo el contenido a cifrar en un ambiente con nombre igual a la clave. Este nombre estará protegido y solo aquellos sistemas que tengan la clave conocerán el nombre del ambiente (la clave).

Para descifrar es aun más sencillo, lo único que tenemos que hacer es tener el permiso “open clave” donde clave e el nombre protegido del ambiente que “cifra” el mensaje.

13.2.- Descripción del cifrado y descifrado asimétrico.

Esto ya no es tan sencillo, ya no solo se trata de conocer un nombre que sirve tanto para cifrar, como para descifrar. Ya que tenemos que diferenciar entre las claves privadas y las claves publicas. Ya no usamos una clave sino dos, que son complementarias es decir se descifran una a la otra. Pero no debemos de permitir que alguien que tenga la clave publica sea capaz de sacar de hay la clave privada, ni viceversa.

Para esto podríamos pensar en varias soluciones, pero la que hemos tomado como mejor es que la claves sean en realidad pares de claves.

Descripción:

Clave privada:
“*(v acido)(open acido | ClavePriv[open ClavePub.acido[out ClavePriv.open ClavePriv]])”*

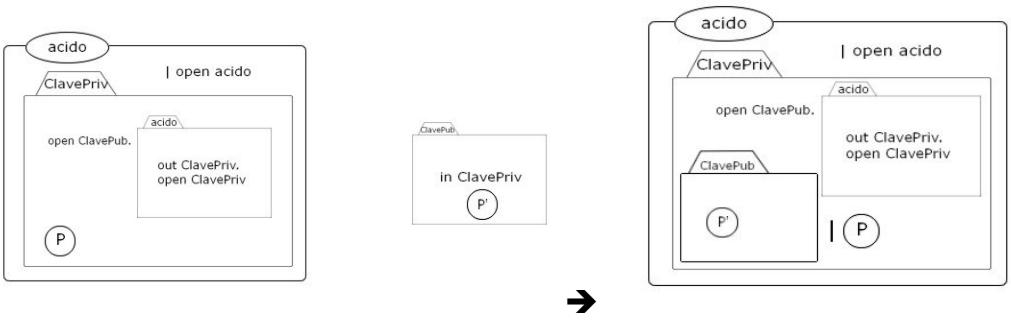
Clave pública:
“ClavePub[in ClavePriv]”

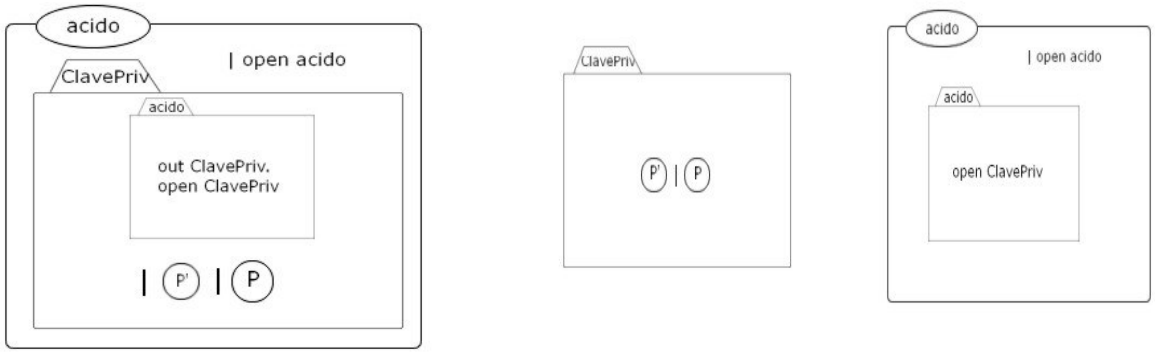
Tanto ClavePriv como ClavePub son nombre protegidos e iguales para que funcionen como una clave completamente segura. El mensaje cifrado estará en el interior del ambiente ClavePub o clavePriv según sea el cifrado.

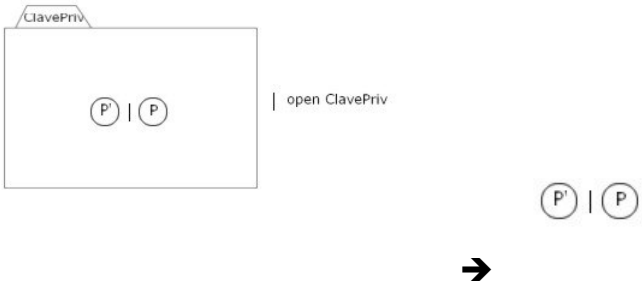
Veamos paso a paso como los cifrados se deshacen mutuamente al enfrentarse, Supongamos que contienen mensajes ClavePriv y ClavePub llamados P y P’ y veamos que como resultado final tenemos efectivamente P y P’ solamente.

Punto inicial.
*(v acido)(open acido | ClavePriv[open ClavePub.acido[out ClavePriv.open ClavePriv] | P]) | ClavePub[**in ClavePriv** | P’]*
=>
*(v acido)(open acido | ClavePriv[**open ClavePub**.acido[out ClavePriv.open ClavePriv] | P | ClavePub[P’]])*
=>

*(v acido)(open acido | ClavePriv[acido[**out ClavePriv**.open ClavePriv] | P | P'])*
=>
*(v acido)(**open acido** | acido[open ClavePriv] | ClavePriv[P | P'])*
=>
***open ClavePriv** | ClavePriv[P | P'])*
⇨ *P | P'*







Esta es la única cadena de pasos posibles, y como se ve termina con el descifrado de los mensajes en realidad nunca llegaremos a algo así porque una de las claves, la que actuará como descifrador, no tendrá mensaje P.

Viendo esto está claro que el hecho de tener solamente una de las claves de un par de claves, no puede obtener la información suficiente para construir su par.

En el calculo de ambientes no hay forma de obtener el nombre de un ambiente y devolverlo, ni tampoco de un permiso. Así pues estas estructuras de claves son teóricamente invulnerables. Es evidente que el hecho de tener permiso de salida de un cierto ambiente X (out X) no implica que podamos obtener el nombres y usarlo para construir permisos de entrada, apertura, etc... Si el calculo de ambientes lo permitiera no tendría sentido diferencias entre los diferentes permisos.

Esta descripción de los pares de claves asimétricos implica una serie de funcionalidades que debemos de implementar, como introducir el mensaje dentro del cifrado y como elegir la clave con la que ciframos o desciframos.

13.3.- Introducción del mensaje dentro de la clave.

Un ambiente por si solo, no puede introducirse dentro de un ambiente con un nombre protegido, necesita que le den el permiso. Para esto usaremos el uso de "agentes" que se encargaran de transportar ambientes a través de otros ambientes con nombre protegidos.

En este caso la clave para cifrar tendrá al ser creada un proceso (ambiente) para interiorizar un mensaje.

AgenteClave[out **clave**.in **mensaje**.in **clave**]

Clave → Puede ser ClavePriv o ClavePub según sea la clave.
Mensaje → Será el nombre del ambiente al que dejaremos pasar y que debe de tener un permiso "open AgenteClave" para que el permiso in clave sea efectivo. Para simplificar siempre sera '**Mensaje'**, es decir solo ciframos ambiente cuyo nombre sea mensaje, el cual debe de ser un nombre no protegido al igual que AgenteClave.

Veamos como funciona un agente transportador con estas condiciones.

(Supongamos que usamos la clave pública de un cierto par y P el contenido del mensaje)

ClavePub[in ClavePriv | AgenteClave[**out ClavePub**.in Mensaje.in ClavePriv]] | Mensaje[open AgenteClave | P]
=>
ClavePub[in ClavePriv] | AgenteClave[**in Mensaje**.in ClavePriv] | Mensaje[open AgenteClave | P]
=>
ClavePub[in ClavePriv] | Mensaje[**open AgenteClave** | AgenteClave[in ClavePriv] | P]
=>
ClavePub[in ClavePriv] | Mensaje[**in ClavePriv** | P]
=>
ClavePub[in ClavePriv | Mensaje[P]]

Vemos que al final tenemos el ambiente Mensaje con su contenido cifrado, está será entonces la forma de cifrar.

13.4.- Elección de la clave de cifrado o descifrado

Este problema no es trivial. Si tenemos que realizar un cifrado con la clave privada, pues esta claro que será con la nuestra, por ser la única clave privada que poseemos, pero si el cifrado ha de realizars con la clave publica del sistema destinatario del mensaje, como saber que clave es y como hacer que solo con esa clave pueda cifrar el mensaje. Este mismo problema nos encontramos al descifrar si el cifrado es privado por que tendremos que elegir igual que antes entre las diferentes claves públicas registradas.

Por ultimo interesa saber que es lo que se pretende hacer si cifrar o descifrar, ya que como vimos para cifrar usamos un "agenteClave" que para descifrar no tiene sentido. Esto justifica que diferenciemos para una misma clave, dos descripciones diferentes según sea para descifrar o para cifrar.

Clave pública para cifrar:
ClavePub[in ClavePriv | AgenteClave[**out ClavePub**.in Mensaje.in ClavePriv] | AgenteClave[out ClavePub.in Mensaje.in ClavePub]]
Clave pública para descifrar:
ClavePub[in ClavePriv | AgenteClave[**out ClavePub**.in Mensaje.in ClavePriv]]
Clave privada para cifrar:
(v acido)(open acido | ClavePriv[open ClavePub.acido[out ClavePriv.open ClavePriv]| AgenteClave[out ClavePub.in Mensaje.in ClavePriv]]
Clave privada para descifrar:
(v acido)(open acido | ClavePriv[open ClavePub.acido[out ClavePriv.open ClavePriv]| AgenteClave[out ClavePub.in Mensaje.in ClavePriv] | AgenteClave[out ClavePub.in Mensaje.in ClavePub]]

Para eso almacenaremos los cifradores y descifradores de cada clave por separado, y transportando el mensaje hacia el ambiente apropiado podremos cifrar/descifrar adecuadamente

13.5.- Descripción de los cifradores y descifradores

Supongamos que queremos cifrar un cierto mensaje con la clave privada del sistema. Ese mensaje debe de decir de alguna forma que tiene que ser cifrado con esa clave y no otra, además debemos de devolver un ambiente con la información de que clave hemos usado para cifrar el mensaje para que podamos distinguir entre las distintas claves para decodificar.

Como sabemos los nombres ClavePiv y ClavePub están protegidos, y esto hace que no sepamos que son ni que tienen en su interior. Por esto necesitamos que tanto clavePriv como ClavePriv se recubran de un cierto ambiente con un nombre público y suficiente información para que podamos recuperar la información.

La función de estos ambientes es la de "tipar" el mensaje codificado, para que según el "tipo" se obre en consecuencia al descifrar. Los nombre que hemos decidido darles son "priv" para el cifrado privado y "pub" para el público.

Y para indicar que queremos que un cierto ambiente sea cifrado, siguiendo la misma idea, será encerrar el ambiente Mensaje dentro de un Ambiente SolicitudPriv o SolicitudPub según queramos un cifrado privado o público. Si el cifrado que queremos es público, no tenemos toda la información necesaria por que podemos tener varias claves públicas registradas. Para solucionar esto tenemos dos opciones igualmente validas:

1. Que dentro de SolicitudPub tengamos un permiso in “nombreSistema” donde nombreSistema indica el nombre del sistema cuya clave pública vamos a usar para cifrar.
2. Que dentro de SolicitudPub tengamos el nombre del sistema cuya clave pública vamos a usar para cifrar como <nombre> (comunicación de salida asincrona), que debería ser recogida para ser convertida en un permiso de entrada

Usaremos la primera, por simplicidad y por ser más rápida y directa.

Con estos requisitos vemos que no es suficiente la forma de cifrar que teníamos ya que tenemos que tipar los mensajes y además dar información para el descifrado posterior, necesitamos unos cifradores y descifradores. Estos cifradores y descifradores son los que se encargan de implementar toda la funcionalidad referente al cifrado y descifrado, salvo la elección del cifrador (problema análogo a la elección de clave).

13.5.1.- Cifrador privado:

Descripción textual:

Cuando se encuentra con un ambiente nombrado “SolicitudPriv” realiza el cifrado del mensaje que contiene dicho ambiente con la clave privada del sistema. Dando como resultado final un mensaje tipado con el nombre priv.

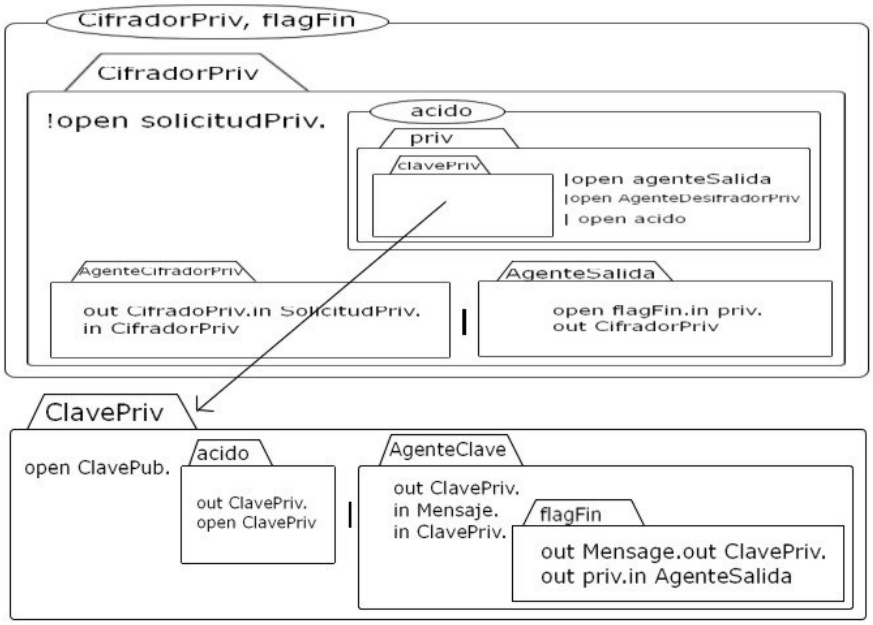
Realiza tres acciones:

1.- Interiorizar la solicitud mediante un AgenteCifradorPriv.
2.- Cifrar el mensaje .
3.- Sacar fuera el mensaje cifrado mediante un AgenteSalida.

Descripción mediante el calculo de ambientes:

(v CifradorPriv,flagFin)CifradorPriv[!(open solicitudPriv.(v acido)(priv[ClavePriv[*1*] | open acido | open AgenteSalida | open AgenteDescifradorPriv]) | AgenteCifradorPriv[out CifradorPriv. In solicitudPriv.in CifradorPriv] | AgenteSalida[open flagFin.in priv.out CifradorPriv]]

1
ClavePriv[open ClavePub.acido[out ClavePriv.open ClavePriv] | AgenteClave[out ClavePriv.in Mensaje.in ClavePriv.flagFin[out Mensaje.out ClavePriv.out priv.in AgenteSalida]]]



Aspectos que han motivado el diseño del cifrador de esta forma.

El open AgenteSalida que se encuentra dentro del ambiente priv esta motivado para que el AgenteSalida pueda sacar a priv del cifrador.

El flagFin es necesario para poder informar al AgenteSalida de que el mensaje ya esta cifrado y puede ser sacado del cifrador. Sin esto podriamos sacar el ambiente priv antes de que en contenido del mensaje esté dentro de ClavePriv y por tanto al salir podría ser vulnerable. Además protegemos el nombre del flag para que no se confunda con otro.

El AgenteSalida de dentro del cifrador, junto con el open AgenteSalida del ambiente priv, implementan la salida del los mensajes una vez que están cifrados.

13.5.2.- Cifrador público:

Descripción textual:

Es completamente análogo al cifrador privado salvando las diferencias de nombres y que usamos la clave pública para cifrar. El resultado final está tipado con el nombre priv.

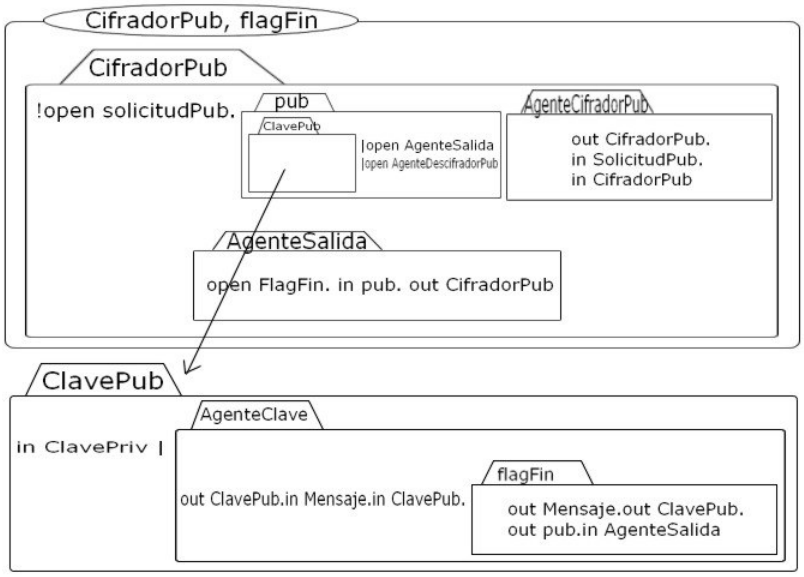
Realiza las mismas tres acciones que el cifrador privado:

1.- Interiorizar la solicitud mediante un AgenteCifradorPriv.
2.- Cifrar el mensaje .
3.- Sacar fuera el mensaje cifrado mediante un AgenteSalida.

Descripción mediante el calculo de ambientes:

(v CifradorPub,flagFin)CifradorPub[!(open solicitudPub.pub[ClavePub[*1*] | open AgenteSalida | open AgenteDescifradorPub] | AgenteCifradorPub[out CifradorPub. In solicitudPub.in CifradorPub] | AgenteSalida[open flagFin.in pub.out CifradorPub]]

1
ClavePub[in ClavePriv | AgenteClave[out ClavePub.in Mensaje.in ClavePub.flagFin[out Mensaje.out ClavePub.out pub.in AgenteSalida]]]



No vamos a explicar nada ya que como se puede ver es igual que el cifrados privado

13.5.2.- Descifrador público (descifra cifrados públicos usando la clave privada del sistema)

Descripción textual:

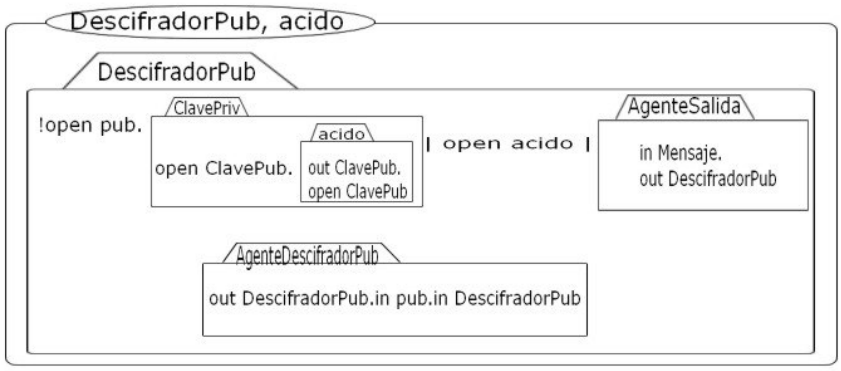
Siguiendo la misma idea de los cifradores y copiando la estructura vemos que los cifradores hacen prácticamente las mismas operaciones.

1.- Captura del mensaje cifrado
2.- Descifrado del mensaje
3.- Salida del mensaje en texto claro del descifrador.

Descripción mediante el calculo de ambientes:

(v DescifradorPub)DescifradorPub[open pub.ClavePriv[*1*] | open acido | AgenteDescifradorPub[out DescifradorPub.in pub.in DescifradorPub] | AgenteSalida[in Mensaje.out DescifradorPub]]

1
ClavePriv[open ClavePub.acido[out ClavePub.open ClavePub]]



Notas:

Ya no necesitamos el flag para saber cuando está descifrado, ya que cuando el ambiente con el nombre mensaje esté a la vista, ya estará descifrado.

El resto del descifrador el igual al cifrador salvo que la clave que usamos no tiene un AgenteClave ya que no necesita interiorizar el mensaje (no cifra).

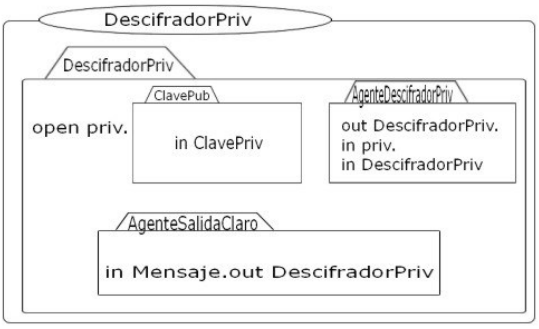
13.5.4.- Descifrador privado (descifra cifrados privados usando una clave pública)

Descripción textual:

No vamos a decir nada de este descifrador, más que es igual al descifrador público solo que usa la clave pública de descifrado en vez de la privada.

Descripción mediante el calculo de ambientes

(v DecifradorPriv)DescifradorPriv[open priv.ClavePub[in ClavePriv] |
AgenteDescifradorPriv[out DescifradorPriv.in priv.in DescifradorPriv] |
AgenteSalidaClaro[in Mensaje.out DescifradorPriv]]



13.6.- Selección de clave para el cifrado y descifrado.

Según el protocolo mixto, y la naturaleza misma de los cifrados asimétricos, nosotros tendremos las siguientes claves:

- Nuestra clave privada
- La clave pública del servidor
- Las claves públicas de los sistemas con los que queremos comunicarnos

Y para cada una de estas claves podemos usarlas o para descifrar o para cifrar.

Vemos que claves privadas solo tenemos una, la nuestra, con lo cual si queremos cifrar un mensaje con la clave privada, o descifrar un mensaje cifrado con la clave pública (se utiliza la clave privada para descifrar) solo tenemos una posibilidad así pues no tenie sentido estratificar nuestra clave privada.

Si embargo cuando queremos usar una clave pública debemos de elegir entre las que tengamos registradas para eso vamos a almacenar los cifradores en una ambiente llamado Cerrojo y los descifradores en un ambiente llamado Llavero.

13.6.1.- Cerrojo y Llavero

Descripción de la estructura:

Para decidir entre las diferentes claves públicas con las que podemos cifrar, tendremos que tener en cuenta quien va a decodificar el mensaje (quien es el destinatario del mensaje) y cifrar con la clave que tenemos registrada a su nombre. Ya que los nombres de los ambientes cifradores y descifradores están protegidos fuera del propio cifrador o descifrador así pues este nombre no puede servir para diferenciar ya que es inaccesible. Por tanto los cifradores y descifradores deben de tener un ambiente que los contenga y con un nombre que identifique el sistema que tiene la clave privada.

Los nombre Cerrojo y llavero estarán protegidos para que nadie pueda mirar siquiera que claves tenemos registradas. Esto implica que las solicitudes a cifrar y los mensajes a descifrar tienen que atravesar más capas y por tanto necesitaremos mas agentes transportadores además de los permisos para que las solicitudes y los textos cifrados se dejen transportar (permisos “open”).

- ***LLAVERO***

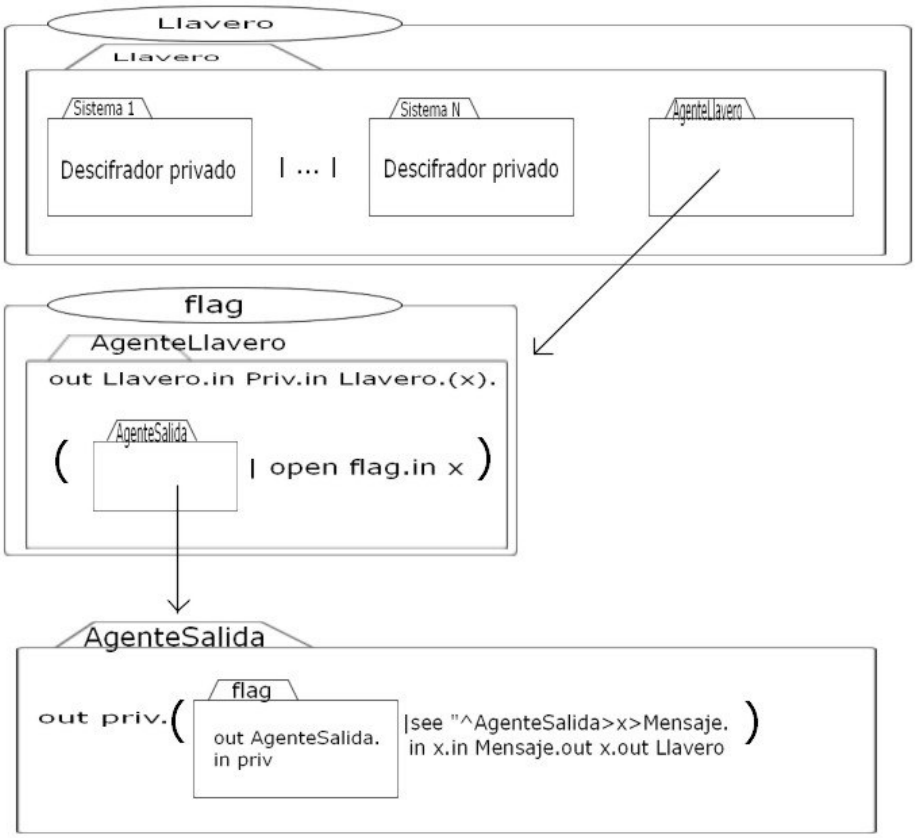
(v Llavero)Llavero[Sistema1[*1*] |...| SistemaN[*1*] | AgenteLlavero[*2*]]

1 Dentro de los distintos ambientes con los nombres de los propietarios de las claves tenemos los descifradores privados (descifran mensajes privados con la clave pública)

2 El agente llavero además de introducir el mensaje crea el AgenteSalida que sacará el mensaje del ambiente SistemaI y del Llavero, para esto lee del mensaje el nombre del sistema del que tenemos que usar la clave.

(v flag)AgenteLlavero[out Llavero.in priv.in Llavero.(x).(AgenteSalida[*3*] | open flag.in x)]

3 AgenteSalida[out priv.(flag[out AgenteSalida.in priv] | see "^AgenteSalida>x>Mensaje".in x.in Mensaje.out x.out Llavero)]



Notas:

La razón de que construyamos el AgenteSalida dentro del AgenteLlavero, es que por que el AgenteLlavero tiene que leer el nombre del sistema dueño de la clave para que pueda elegir entre los diferentes sistemas de los que tenemos registrada la clave. Por

la misma razón el AgenteSalida tiene que tener la misma información para saber donde buscar el mensaje cifrado que tiene que sacar fuera.

El flag que tenemos lo usamos para asegurarnos de que el AgenteSalida ya ha salido del ambiente priv donde lo metió el AgenteLlavero, mediente este flag conseguimos una cierta ordenación en la evolución del sistema. Tal y como esta diseñado el AgenteClave tiene que salir de priv cuando el Mensaje a descifrar ha entrado en el Llavero pero aun no ha entrado en el ambiente SistemaI, esto lo conseguimos con este flag que regula el orden.

Nuevas estructuras creadas:

Para esta descripción hemos inventado dos estructuras nuevas, la orden see "ruta" y los paréntesis.

Los paréntesis:

El uso que vamos a darle a los paréntesis es conseguir que arranquen en paralelo varias líneas de permisos o ambientes

a.(P | Q | ...) donde P y Q pueden ser cualquier estructura valida en calculo de ambientes.

La equivalencia entre los paréntesis que usamos y el calculo de ambientes puro es:

a.(P | Q | ...) ⇔ (v n)a.n[P | Q | ...] | open n

Como vemos es un ambiente auto-destructivo que según se crea se deshace

La orden "see <ruta>":

Con esta estructura lo que pretendemos es bloquear comandos posteriores hasta que no se encuentre el ambiente indicado por la ruta. Esta estructura es una extensión de la orden "see n" donde lo que pretendemos es que este en paralelo un ambiente nombrado 'n',

"see n.P" ⇔ (v a,b) a[in n.out n. a be b.P] | open b

La orden "x be y " (renombramiento) ya la describimos también con anterioridad

Bueno entonces pasemos a describir la ruta o path, la escribiremos con la siguiente notación '>' para indicar que el siguiente paso es entrar en un ambiente '^' si hay que salir fuera de un cierto ambiente. Una vez completada la ruta tenemos que hacer el camino inverso para volver.

Esta orden "see" no asegura que no se quede bloqueado si la ruta no es unívoca y completamente determinista en cada paso, hemos de tener cuidado con el uso de esta estructura ya que puede dejar bloqueado al sistema.

Veamos como queda es caso concreto que hemos usado aquí

see "^AgenteSalida>x>Mensaje" ⇔ (v n,m)n[out AgenteSalida.in x.in Mensaje.out Mensaje.out x. In AgenteSalida.n be m] | open m

- ***CERROJO***

Descripción textual:

La estructura cerrojo es completamente análoga al anterior, bajo las salvedades de los nombres y de que en vez de esperar el AgenteSalida a ver un ambiente Mensaje, espera a Tener un ambiente pub, que estará ya cifrado cuando salga del cifrador (que será cuando lo podamos ver)

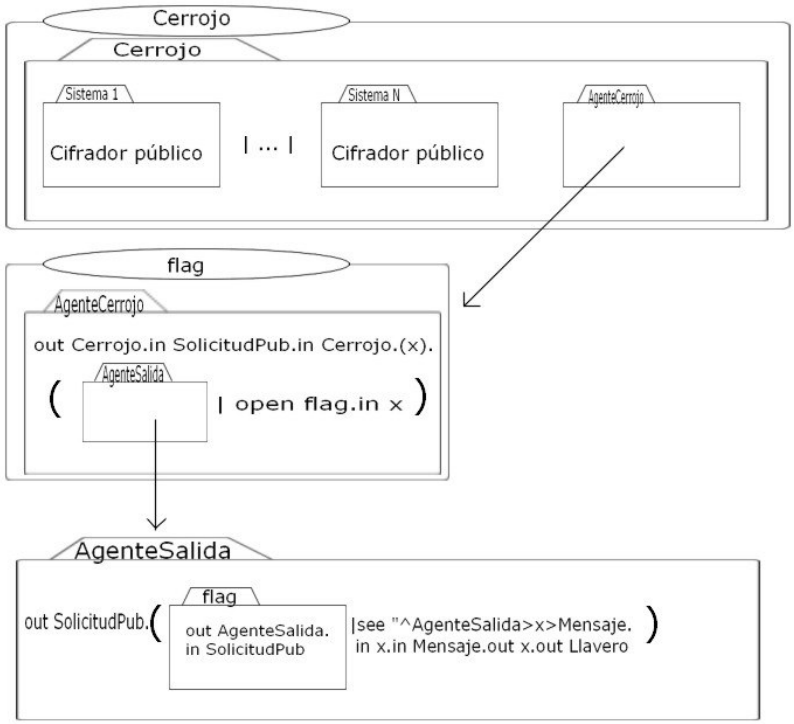
Descripción mediante el calculo de ambientes:

(v Cerrojo)Cerrojo[Sistema1[*1*] |...| SistemaN[*1*] | AgenteCerrojo[*2*]]

1 Dentro de los distintos ambientes con los nombres de los propietarios de las claves tenemos los cifradores públicos.
2 El agente Cerrojo además de introducir el mensaje a cifrar crea el AgenteSalida que sacará el mensaje cifrado del ambiente SistemaI y del Cerrojo, para esto lee de la solicitud de cifrado el nombre del sistema del que tenemos que usar la clave.

(v flag)AgenteCerrojo[out Cerrojo.in SolicitudPub.in Cerrojo.(x).(AgenteSalida[*3*] | open flag.in x)]

3 AgenteSalida[out SolicitudPub.(flag[out AgenteSalida.in SolicitudPub] | see "^AgenteSalida>x>pub".in x.in pub.out x.out Llavero)]



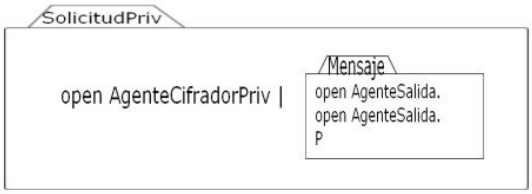
Bueno hasta ahora hemos ido paso a paso describiendo como son los elementos que realizan en cifrado y descifrado, pero como hemos visto necesitamos que para que

una cierta información sea cifrada necesitamos que mantenga una cierta estructura y que contenga una serie de permisos.

Vamos ahora a describir como tendrían que estar formados estos ambientes abstrayendo el contenido con el nombre 'P'.

13.7.- Solicitud de cifrado con la clave privada.

SolicitudPriv[open AgenteCifradorPriv | Mensaje[open AgenteSalida.open AgenteSalida | P]]



Explicación de las parte de la estructura.

Open AgenteCifradorPriv:

Para que la solicitud Pueda ser introducida dentro del cifrador privado.

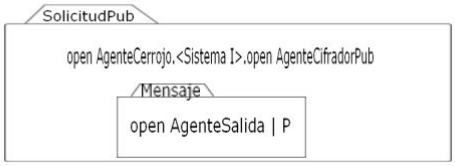
El nombre del ambiente Mensaje a de ser "Mensaje" por que es lo que esperan las Claves con las que ciframos otro nombre distinto no sería cifrado.

Open AgenteSalida.open AgenteSalida

Vemos que dentro de 'Mensaje' tenemos dos permisos 'open AgenteSalida' esto es porque el Ambiente Mensaj tras ser descifrado deberá salir del Descifrador privado y del Llavero, ya que el descifrador asociado al usar una clave pública, como dijimos antes, estará dentro del ambiente 'Llavero'

13.8.- Solicitud de cifrado con la clave pública.

SolicitudPub[open AgenteCerrojo.<SistemaI>.open AgenteCifradorPub | Mensaje[open AgenteSalida | P]]



Explicación de las parte de la estructura.

open AgenteCerrojo.<SistemaI>.open AgenteCifradorPub

Para el cifrador privado solo necesitábamos un 'open AgenteCifradorPriv', pero ahora el cifrador está dentro del Agente Cerrojo, así que, por orden, necesitaremos

Open AgenteCerrojo, para introducir la solicitud dentro del cerrojo

<SistemaI> Para elegir entre las distintas claves (lo lee el AgenteCerrojo)

Open AgenteCifradorPub, para que pueda ser cifrado.

El nombre 'Mensaje' al igual que antes debe de ser fijo.

Open AgenteSalida

Esta vez solo necesitamos uno, ya que será descifrado con una clave privada y el mensaje, ya en claro, solo tendrá que salir del cifrador.

También es imprescindible que al igual que SolicitudPriv y SolicitudPub tienen una estructura y permisos fijos para que el mensaje que contienen sean cifrados, los ambientes pub y priv resultantes, también deben de tener una estructura fija para que el cifrado que llevan dentro, pueda ser descifrado.

Realmente ya los describimos cuando definimos los cifradores, pero veamos que tenemos que hacer algunos cambios ya que en ese punto aun no habíamos tenido en cuenta que 'CifradorPub' y 'DescifradorPriv' estarían en los ambientes 'Cerrojo' y 'Llavero' respectivamente.

13.9.- Ambiente Priv (resultado de un cifrado con la clave privada)

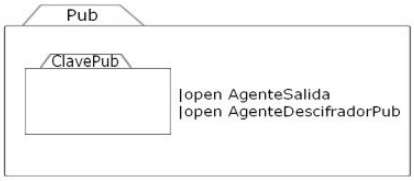
Priv[ClavePriv[....] | open acido | open AgenteSalida | open AgenteLlavero.<SistemaI>.open AgenteDescifradorPriv]

Modificaciones:

Ahora necesitamos un 'Open AgenteLlavero' para que 'priv' pueda ser capturado por el Llavero. Además necesitamos leer cual es la clave que debemos usar para descifrar, por tanto después tenemos que añadir <SistemaI> que será el nombre del sistema que lo cifró. Por último el permiso 'open AgenteDescifradorPriv' que se mantiene igual que antes.

13.10.- Ambiente Pub (resultado de un cifrado con la clave privada)

Pub[ClavePub[...] | open AgenteSalida | open AgenteDescifradorPub]



Este ambiente no cambia, ya que el descifrador público al usar una clave privada (por tanto ser única) no tiene que elegir entre varias y podemos dejar el descifrador tal cual estaba.

Hasta ahora hemos descrito perfectamente como se cifran y descifran mensajes. Nos quedan por ver como se comunican los sistemas entre ellos (paso de mensajes) y como llevamos el control del protocolo de intercambio de claves.

Paso de mensajes entre sistemas.

Puntos a tener en cuenta para el diseño:

1. Queremos que los mensajes se ruten unívocamente, es decir no queremos dejar un mensaje en un medio inseguro y esperar que quien lo tiene que recibir lo capture. Lo que queremos es que el mensaje conozca de su destinatario
2. Queremos que el sistema que recibe el mensaje pueda de la forma más sencilla posible de quien procede el sistema, para poder elegir si coger o no un mensaje.
3. Queremos que todo lo que se realice dentro del sistema esté lo más oculto posible para otros sistemas ajenos.

Soluciones mediante el calculo de ambientes.

1. Para que el mensaje sepa a donde ir debemos de hacer que los sistemas tengan un nombre publico (no protegido) de forma que con un simple permiso ‘in SistemaI’ dentro del mensaje pueda rutarse.
2. Para identificar el mensaje, lo más facil es que el mensaje que recibe un sistema sea un ambiente con el nombre igual al nombre del sistema remite.
3. Para que todos los procesos internos de los sistemas estén protegidos deberíamos de proteger el nombre del sistema (usando la protección de los ambientes (v SistemaI)) y solo coger mensajes mediante el uso de agente transportadores.

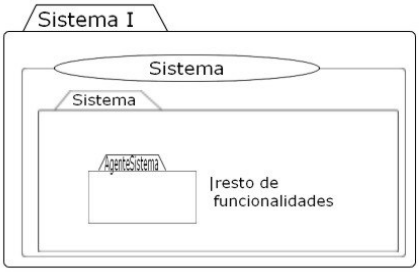
Si observamos los puntos 1 y 3 vemos que entramos en una contradicción, por una parte queremos que todo el mundo conozca el nombre del sistema (que el nombre

sea público) y por otra que el sistema este protegido con una especie de ‘firewall’ para evitar que cualquiera acceda al sistema desde el exterior (que el nombre del sistema esté protegido).

Sin embargo no es ninguna contradicción si lo pensamos bien, ya que podemos tener un ambiente con un nombre público recubriendo a otro con un nombre protegido. Con esto conseguimos que los mensajes sepan a donde ir, entraran dentro del ambiente con el nombre público (no protegido) correspondiente, pero sin llegar a entrar en el sistema ya que tiene protegido el nombre y no podremos entrar salvo que un ‘agente’ nos de los permisos necesarios.

Entonces los sistemas quedarían de la siguiente manera:

SistemaI[(v Sistema)Sistema[AgenteSistema[...] | (resto de funcionalidades del sistema)]]



Una cosa importante es que el AgenteSistema no coge cualquier ambiente, sino que coge el ambiente nombrado como espera(el resto los ignora).

Estas nuevas funcionalidades llevan consigo una serie de complicaciones nuevas, tales como la forma de sacar los mensajes de los Sistemas y rutarlos hasta los sistemas destinatarios.

Tenemos un problema nuevo y es que no sabemos cual será la acción que debemos de realizar sobre un mensaje, podemos cifrarlos con la clave privada o bien con una publica o simplemente enviarlo. Este problema es mayor si lo que tenemos es un mensaje cifrado, podemos re-cifrarlo con una clave publica o privada, podemos enviarlo o podemos descifrarlo.

Para solucionar este problema vamos a crear una nueva estructura: Ordenes.

Las ordenes se encargaran de dar los permisos necesarios para realiza una cierta acción a los ambientes ‘Mensaje’, ‘priv’ y ‘pub’. Además de particularizar los cifrados y descifrados indicando, en caso de usar claves públicas, que clave usar exactamente.

La solución que se nos ocurre es tener ambientes ordenes que crearemos específicamente para cada ambiente y cada acción. Por tanto los ambientes Mensajes, pub y priv no tendrán todos los permisos necesarios para cifrarse y descifrarse sino que se los darán mediante ordenes.

Describamos una a una las posibles ordenes, para cada ambiente.

13.11.- Ordenes para el Mensaje.

Cifrado privado

Punto de partida: Mensaje[open ordenes | P]

Queremos conseguir: SolicitudPriv[open AgenteCifradorPriv | Mensaje[open AgenteSalida.open AgenteSalida | P]]

Orden: (ordenes[in Mensaje.in esperando | open AgenteSalida.open AgenteSalida] | preparando[open AgenteCifradorPriv | see Mensaje.esperando be SolicitudPriv])

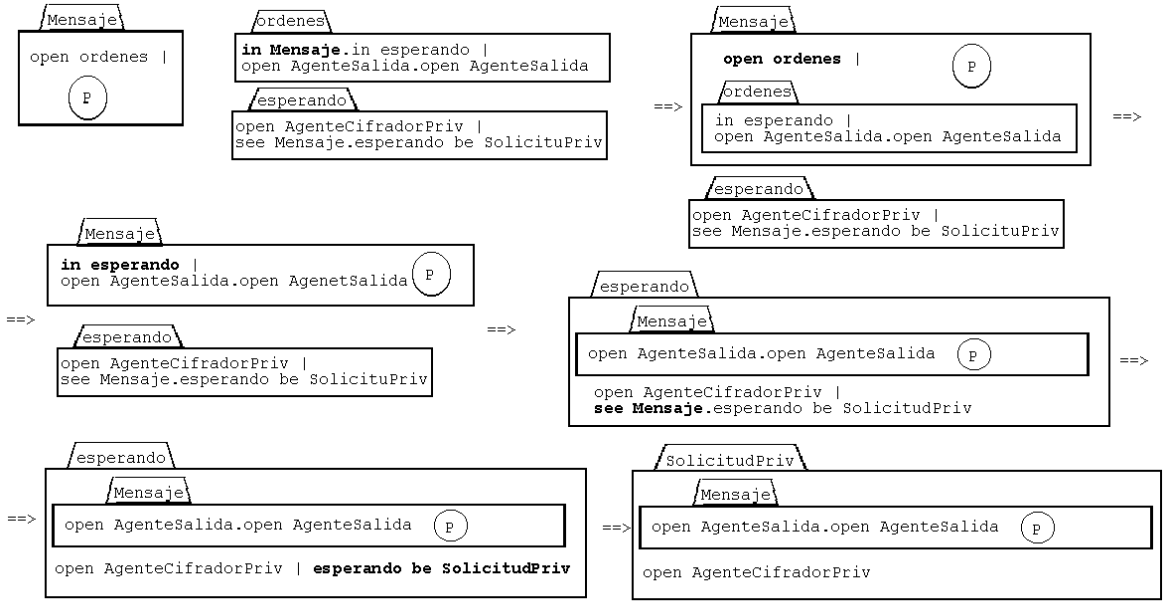
Las estructuras no simples como “esperando be SolicitudPriv” y los paréntesis ya los definimos antes.

Con esto preparamos el cifrado y el descifrado

Veamos el proceso paso a paso.

Mensaje[open ordenes | P] | (ordenes[in Mensaje.in esperando | open AgenteSalida.open AgenteSalida] | preparando[open AgenteCifradorPriv | see Mensaje.esperando be SolicitudPriv])
=>
Mensaje[open ordenes | P] | ordenes[**in Mensaje**.in esperando | open AgenteSalida.open AgenteSalida] | preparando[open AgenteCifradorPriv | see Mensaje.esperando be SolicitudPriv]
=>
Mensaje[**open ordenes** | P | ordenes[in esperando | open AgenteSalida.open AgenteSalida]] | preparando[open AgenteCifradorPriv | see Mensaje.esperando be SolicitudPriv]
=>
Mensaje[**open ordenes** | P | ordenes[in esperando | open AgenteSalida.open AgenteSalida]] | preparando[open AgenteCifradorPriv | see Mensaje.esperando be SolicitudPriv]
=>
Mensaje[P | **in esperando** | open AgenteSalida.open AgenteSalida] | preparando[open AgenteCifradorPriv | see Mensaje.esperando be SolicitudPriv]
=>
preparando[Mensaje[P | open AgenteSalida.open AgenteSalida] | open AgenteCifradorPriv | see Mensaje.esperando be SolicitudPriv]
=>
preparando[Mensaje[P | open AgenteSalida.open AgenteSalida] | open AgenteCifradorPriv | **see Mensaje**.esperando be SolicitudPriv]
=>
preparando[Mensaje[P | open AgenteSalida.open AgenteSalida] | open AgenteCifradorPriv | **esperando be SolicitudPriv**]
=>

SolicitudPriv [Mensaje[P | open AgenteSalida.open AgenteSalida] | open AgenteCifradorPriv]



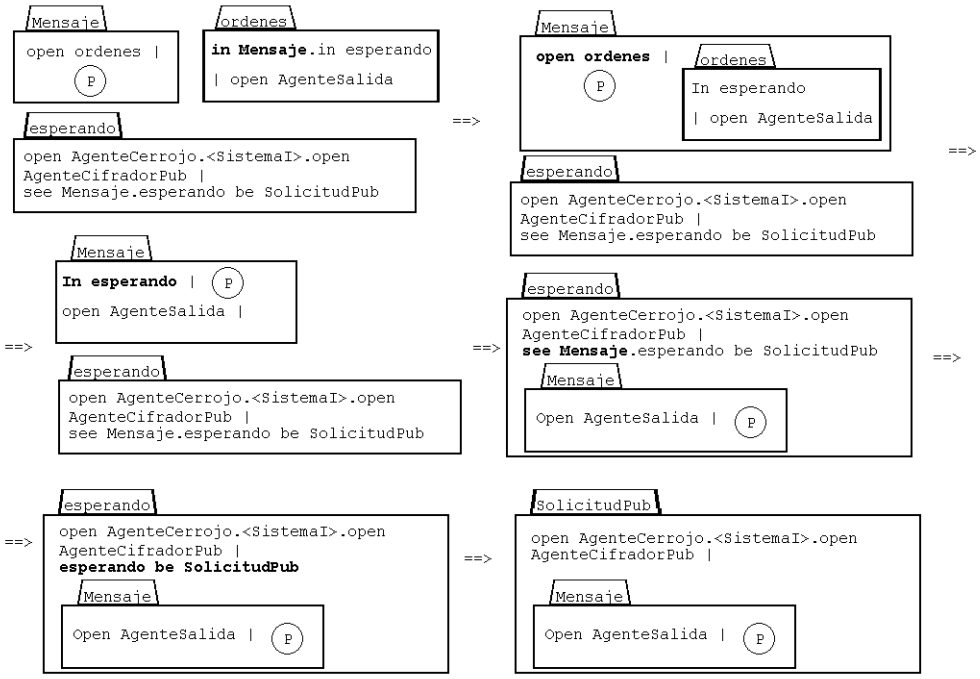
Y con esto ya tenemos exactamente lo que queríamos una solicitud de cifrado completa lista para ser cifrada.

Cifrado público.

Punto de partida: Mensaje[open ordenes | P]

Queremos conseguir: SolicitudPub[open AgenteCerrojo.<SistemaI>.open AgenteCifradorPub | Mensaje[open AgenteSalida | P]]

Orden: (ordenes[in Mensaje.in esperando | open AgenteSalida] | preparando[open AgenteCerrojo.<SistemaI>.open AgenteCifradorPub | see Mensaje.esperando be SolicitudPub])



Es ahora cuando decidimos el nombre del sistema cuya clave utilizaremos para cifrar, y que será el destinatario del mensaje.

Envío de mensajes:

Para enviar un mensaje debemos de:

1.- Sacarlo del ambiente actual “remite”
2.- Introducirlo en el ambiente destino “destino”
3.- Renombrar el ambiente con el nombre del remite, “remite”

Punto de partida: Mensaje[open ordenes | P]

Queremos conseguir: Mensaje[out Sistema.out “remite”.in “destino”.Mensaje be “remite”.open AgenteSistema | P]

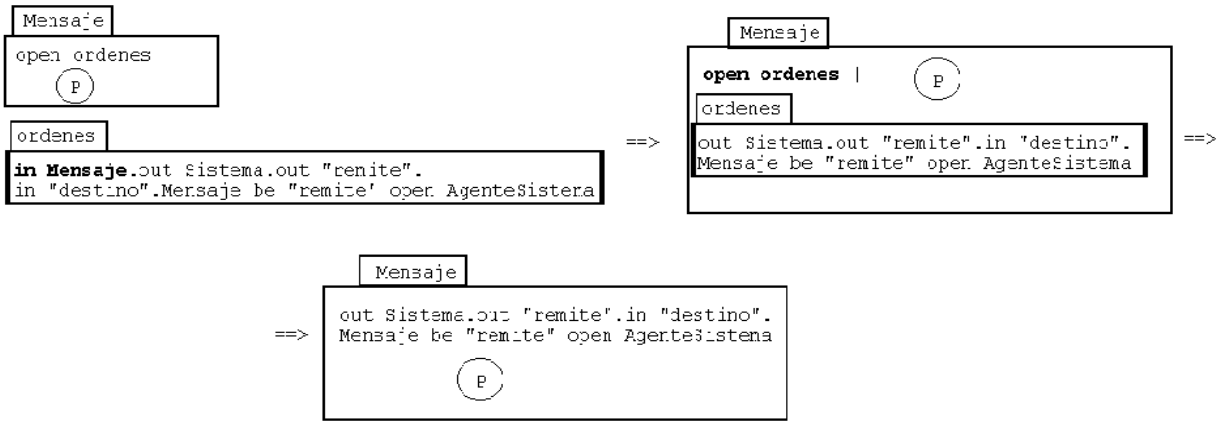
Orden:

Ordenes[in Mensaje.out Sistema.out “remite”.in “destino”.Mensaje be “remite”.open AgenteSistema]

Pasos:

Mensaje[open ordenes | P] | ordenes[**in Mensaje.**out Sistema.out “remite”.in “destino”.Mensaje be “remite”.open AgenteSistema]
=>
Mensaje[**open ordenes** | ordenes[out Sistema.out “remite”.in “destino”.Mensaje be “remite”.open AgenteSistema] | P]
=>

Mensaje[out Sistema.out “remite”.in “destino”.Mensaje be “remite”.open AgenteSistema | P]



Con estas ordenes terminamos las diferentes acciones posibles sobre los mensajes. Vamos ahora a definir las posibles ordenes para ‘pub’ y para ‘priv’

Las posibles ordenes que tienen sentido sobre un cifrado publico son:

- Un cifrado privado
- Descifrar el mensaje
- Enviarlo a otro sistema

13.12.- Ordenes sobre pub

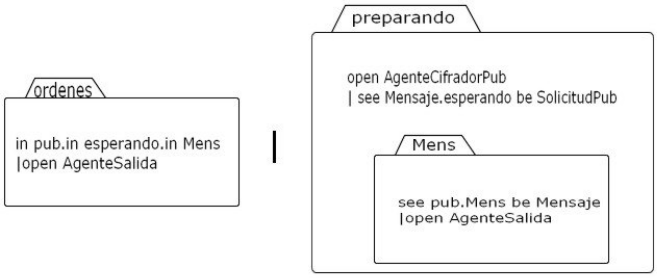
Cifrado privado:

Sabemos que para poder cifrar un mensaje el ambiente debe de llamarse ‘Mensaje’ y esto no ocurre aquí, así que el ambiente ‘pub’ lo tendremos que meter dentro de un ambiente nombrado ‘Mensaje’.

Punto de partida: pub[ClavePub[....] | open ordenes]

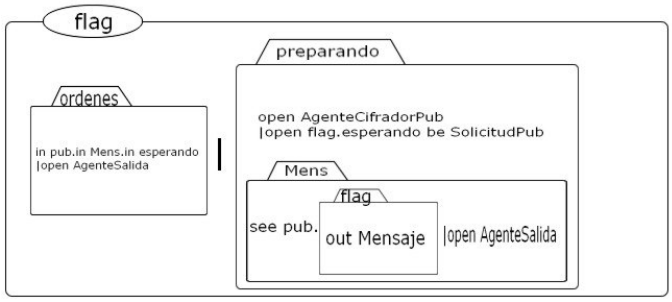
Queremos conseguir: SolicitudPriv[open AgenteCifradorPriv | Mensaje[open AgenteSalida.open AgenteSalida | pub[ClavePub[....]]]

Orden: (ordenes[in pub.in esperando.in Mens | open AgenteSalida] | preparando[open AgenteCifradorPub | see Mensaje.esperando be SolicitudPub | Mens[see pub.Mens be Mensaje | open AgenteSalida])



También podemos cambiar el nombre de ‘esperando’ a ‘SolicitudPub’ mediante el uso de un flag.

Orden: (v flag) (ordenes[in pub.in Mens.in esperando | open AgenteSalida] | preparando[open AgenteCifradorPub | open flag.esperando be SolicitudPub | Mensaje[see pub.flag[out Mensaje] | open AgenteSalida])



Cualquiera de las dos opciones son igualmente validas.

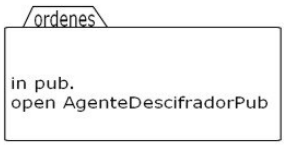
Descifrado publico:

La ordenes han de dar los permisos necesarios para que el mensaje pueda entrar en el descifrador público, ya que los permisos para salir los incluimos en las ordenes de cifrado.

Punto de partida: pub[ClavePub[....] | open ordenes]

Queremos conseguir: pub[ClavePub[....] | open AgenteDescifradorPub]

Orden: ordenes[in pub.open AgenteDescifradorPub]



Pasos:

pub[ClavePub[....] | open ordenes] | ordenes[**in pub**.open AgenteDescifradorPub]
=>
pub[ClavePub[....] | **open ordenes** | ordenes[open AgenteDescifradorPub]]
=>
pub[ClavePub[....] | open AgenteDescifradorPub]

Envío de mensaje codificado con clave pública:

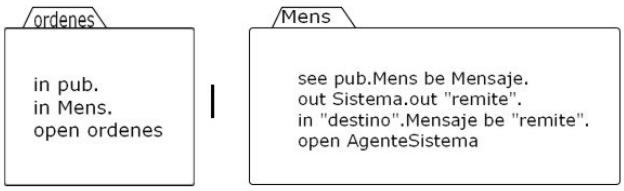
Necesitamos los 3 puntos necesarios para el paso de mensajes, con la diferencia de que ahora no podemos simplemente cambiar el nombre ‘pub’ por el nombre del

sistema remite, ya que perderíamos la información del tipo de cifrado que lleva (si privado o público), así pues podemos, siguiendo ideas anteriores, introducir el ambiente 'pub' en un ambiente 'Mensaje' y enviar este mensaje.

Punto de partida: pub[ClavePub[....] | open ordenes]

Queremos conseguir: Mensaje[out Sistema.out "remite".in "destino".Mensaje be "remite".open AgenteSistema | pub[open ordenes.ClavePub[....]]]
(necesitamos mantener un permiso open ordenes para que al llegar al destino pueda ser descodificado.)

Orden: (ordenes[in pub.in Mens.open ordenes] | Mens[see pub.Mens be Mensaje.out Sistema.out "remite".in "destino".Mensaje be "remite".open AgenteSistema])



Pasos: pub[ClavePub[....] | open ordenes] | (ordenes[in pub.in Mens.open ordenes] | Mens[see pub .Mens be Mensaje.out Sistema.out "remite".in "destino".Mensaje be "remite".open AgenteSistema])
=>
pub[ClavePub[....] | open ordenes] | ordenes[**in pub**.in Mens.open ordenes] | Mens[see pub.Mens be Mensaje.out Sistema.out "remite".in "destino".Mensaje be "remite".open AgenteSistema]
=>
pub[ClavePub[....] | **open ordenes** | ordenes[in Mens.open ordenes]] | Mens[see pub.Mens be Mensaje.out Sistema.out "remite".in "destino".Mensaje be "remite".open AgenteSistema]
=>
pub[ClavePub[....] | **in Mens** .open ordenes] | Mens[see pub.Mens be Mensaje.out Sistema.out "remite".in "destino".Mensaje be "remite".open AgenteSistema]
=>
Mens[**see pub**.Mens be Mensaje.out Sistema.out "remite".in "destino".Mensaje be "remite".open AgenteSistema | pub[ClavePub[....] | open ordenes]]
=>
Mens[**Mens be Mensaje.**out Sistema.out "remite".in "destino".Mensaje be "remite".open AgenteSistema | pub[ClavePub[....] | open ordenes]]
=>
Mensaje[out Sistema.out "remite".in "destino".Mensaje be "remite".open AgenteSistema | pub[ClavePub[....] | open ordenes]]

13.13.- Ordenes para el ambiente priv:

Este vez puede tener sentido el re-cifrar con la clave pública pero no con la privada de nuevo.

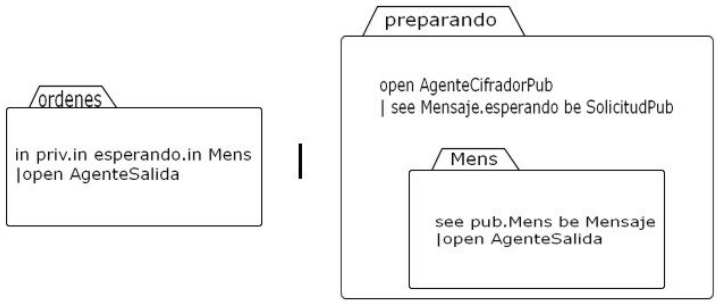
Cifrado publico:

Sabemos que para poder cifrar un mensaje el ambiente debe de llamarse 'Mensaje' y esto no ocurre aquí, así que el ambiente 'pub' lo tendremos que meter dentro de un ambiente nombrado 'Mensaje'.

Punto de partida: (v acido)priv[ClavePriv[....] | open acido | open ordenes]

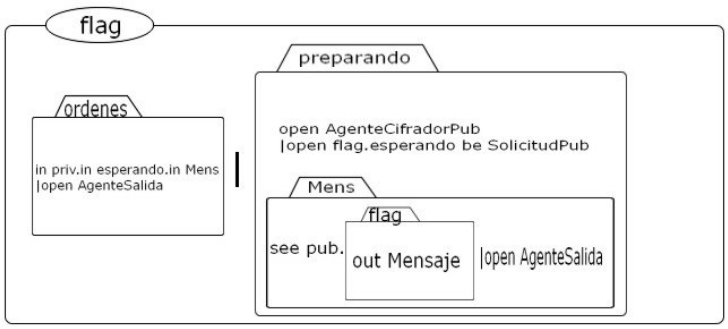
Queremos conseguir: SolicitudPub[open AgenteCerrojo.<SistemaI>.open AgenteCifradorPub | Mensaje[open AgenteSalida.open AgenteSalida | (v acido)priv[ClavePriv[....] | open acido]]

Ordenes: (ordenes[in priv.in esperando.in Mens | open AgenteSalida] | preparando[open AgenteCifradorPub | see Mensaje.esperando be SolicitudPub | Mens[see pub.Mens be Mensaje | open AgenteSalida]])



También podríamos usar un flag.

Ordenes: (v flag) (ordenes[in priv.in esperando.in Mens | open AgenteSalida] | preparando[open AgenteCifradorPub | open flag.esperando be SolicitudPub | Mensaje[see pub.flag[out Mensaje] | open AgenteSalida])



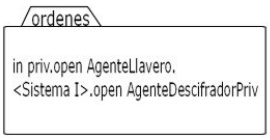
Descifrado privado:

Es análogo al descifrador público del ambiente pub, excepto que ahora davemos dar permiso al llavero ademas del cifrador y dar el nombre del sistema del que usar la clave.

Punto de partida: (v acido)priv[ClavePriv[....] | open acido | open ordenes]

Queremos conseguir: priv[ClavePriv[....] | open acido | open AgenteLlavero. <SistemaI>.open AgenteDescifradorPriv]

Orden: ordenes[in priv. open AgenteLlavero.<SistemaI>.open AgenteDescifradorPriv]



Pasos:
priv[ClavePriv[....] | open acido | open ordenes] | ordenes[**in pub**.open AgenteLlavero. <SistemaI>.open AgenteDescifradorPriv]
=>
priv[ClavePriv[....]| open acido | **open ordenes**| ordenes[open AgenteLlavero. <SistemaI>.open AgenteDescifradorPriv]]
=>
priv[ClavePriv[....]| open acido | open AgenteLlavero.<SistemaI>.open AgenteDescifradorPriv]

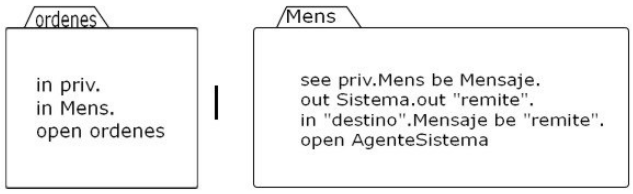
Envío del ambiente priv:

La orden y el procedimiento de envío es prácticamente igual que enviar un cifrado público.

Punto de partida: priv[ClavePriv[....] | open acido | open ordenes]

Queremos conseguir: Mensaje[out Sistema.out "remite".in "destino".Mensaje be "remite".open AgenteSistema | priv[ClavePub[....] | open acido | open ordenes]]

Orden: (ordenes[in priv.in Mens.open ordenes] | Mens[see priv.Mens be Mensaje.out Sistema.out "remite".in "destino".Mensaje be "remite".open AgenteSistema])



Pasos:
priv[ClavePriv[....] | open acido | open ordenes] | ordenes[**in priv**.in Mens.open ordenes] | Mens[see priv.Mens be Mensaje.out Sistema.out “remite”.in “destino”.Mensaje be “remite”.open AgenteSistema]
=>
priv[ClavePriv[....] | open acido | **open ordenes** | ordenes[in Mens.open ordenes]] | Mens[see priv.Mens be Mensaje.out Sistema.out “remite”.in “destino”.Mensaje be “remite”.open AgenteSistema]
priv[ClavePriv[....] | open acido | **open ordenes** | ordenes[in Mens.open ordenes]] | Mens[see priv.Mens be Mensaje.out Sistema.out “remite”.in “destino”.Mensaje be “remite”.open AgenteSistema]
=>
priv[ClavePriv[....] | open acido | **in Mens**.open ordenes] | Mens[see priv.Mens be Mensaje.out Sistema.out “remite”.in “destino”.Mensaje be “remite”.open AgenteSistema]
=>
Mens[priv[ClavePriv[....] | open acido | open ordenes] | **see priv**.Mens be Mensaje.out Sistema.out “remite”.in “destino”.Mensaje be “remite”.open AgenteSistema]
=>
Mens[priv[ClavePriv[....] | open acido | open ordenes] | **Mens be Mensaje**.out Sistema.out “remite”.in “destino”.Mensaje be “remite”.open AgenteSistema]
=>
Mensaje[priv[ClavePriv[....] | open acido | open ordenes] | out Sistema.out “remite”.in “destino”.Mensaje be “remite”.open AgenteSistema]

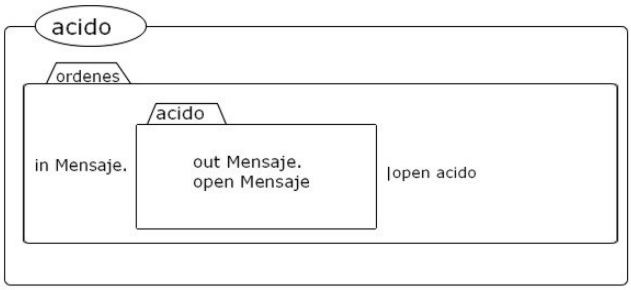
Ya tenemos todas las posibles acciones sobre los ambientes ‘Mensaje’, ‘priv’ y ‘pub’. Pero nos queda por definir los agentes que capturarán mensajes hacia dentro de sistemas y además unas nuevas ordenes para leer un Mensaje, o lo que es lo mismo deshacer la envoltura dada por el ambiente priv.

13.14.- Lectura del ambiente Mensaje.

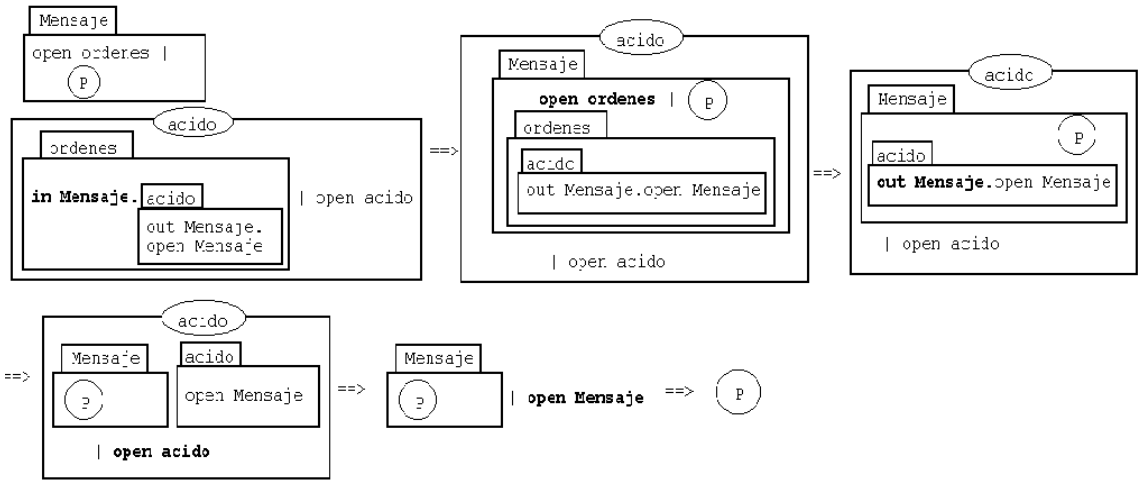
Punto de partida: Mensaje[open ordenes | P]

Queremos conseguir: P

Orden: (v acido) (ordenes[in Mensaje.acido[out Mensaje.open Mensaje]] | open acido)



Pasos:
(Mensaje[open ordenes | P] | (v acido) ordenes[**in Mensaje**.acido[out Mensaje.open Mensaje]] | open acido)
=>
(v acido) Mensaje[**open ordenes** | P | ordenes[acido[out Mensaje.open Mensaje]]] | open acido
=>
(v acido) Mensaje[P | acido[**out Mensaje**.open Mensaje]] | open acido
=>
(v acido) acido[open Mensaje] | Mensaje[P] | **open acido**
=>
open Mensaje | Mensaje[P]
=>
P



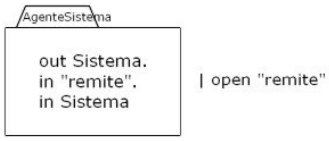
13.15.- Captura de un cierto Mensaje.

Vemos que no podemos hacer un ambiente general para capturar cualquier mensaje, por que según provengan de un sistema u otro se llamarán de una forma u otra ("remite").

Punto de partida: "remite"[open AgenteSistema | P]
Según el valor de "remite" tendremos un AgenteSistema u otro.

Queremos conseguir: (que dentro del Sistema tengamos P es decir)
SistemaI[(v Sistema)Sistema[P | ....]

Agente: (AgenteSistema[out Sistema.in “remite”.in Sistema]|open “remite”)



Pasos: (‘SistemaI’ será el nombre público del sistema que captura el mensaje)
SistemaI[“remite”[open AgenteSistema | P] | (v Sistema)Sistema[AgenteSistema[**out Sistema**.in “remite”.in Sistema]| open “remite” | ....]
=>
(v Sistema)SistemaI[“remite”[open AgenteSistema | P] | AgenteSistema[**in “remite”**.in Sistema] | Sistema[open “remite” | ....]
=>
(v Sistema)SistemaI[“remite”[**open AgenteSistema** | AgenteSistema[in Sistema] | P] | Sistema[open “remite” | ....]
=>
(v Sistema)SistemaI[“remite”[**in Sistema** | P] | Sistema[open “remite” | ....]
=>
(v Sistema)SistemaI[Sistema[“remite”[P] | **open “remite”** | ....]
=>
(v Sistema)SistemaI[Sistema[P | ....]

Así pues la funcionalidad del paso de mensajes rutados entre diferentes sistemas, recae sobre dos estructuras: Las ordenes y el AgenteSistema.

Ya solo queda un pequeño detalle y es la asimilación de nuevos registros de clave el los ambientes Llavero y Cifrador. Pero no tiene ninguna dificultad añadida sobre lo que hemos hecho hasta ahora.

Necesitamos dos nuevos agentes que introduzcan las claves recibidas en los ambientes correspondientes, ‘Llavero’ y ‘Cerrojo’. Estos Agentes capturaran como claves ambientes llamados ClaveCifrado y ClaveDescifrado, ambos ambientes en su interior con una salida asíncrona con el nombre del sistema dueño de la clave (<SistemaI>).

Aún no hemos determinado como será el sistema Servidor, y por tanto no sabemos que es lo que nos devolverá ni como pedir la clave así pues vamos a definir el Servidor

13.16.- Descripción del sistema servidor.

Realmente podemos considerarlo como una simplificación de los sistemas descritos hasta ahora.

El cifrador recibe mensajes en texto claro, con la información necesaria para identificar la clave que quiere y a quien se la tendremos que enviar. Después coge la clave del sistema indicado, cifra esta información con su clave privada y la envía al sistema que la pidió.

Externamente el sistema Servidor será igual que cualquier otro sistema, con un ambiente externo con nombre "Servidor", no protegido, y dentro de este otro ambiente con nombre protegido (p.e. Sistema). La diferencia respecto a los otros sistemas es que este acepta cualquier solicitud que se le envíe, por lo tanto tiene "activados" un agente transportados para cada sistema en todo momento.

Está claro que solo tenemos que tener nuestra clave privada para cifrar y no necesitamos descifrar. Aunque tenemos que tener almacenadas las claves públicas de cifrado y descifrado preparadas para ser enviadas.

Podemos crear un sistema "Servidor" muy sencillo y que cumpla todas las condiciones necesarias:

- **Descripción del Servidor mediante el calculo de ambientes.**

Servidor[v(Sistema)[(*1* lista de agentes) | CifradorPriv[*2*] | (*3* Lista de copiadores de Claves)]

Esta era una descripción funcional pero veamos que esta muy cerca ya de la descripción final.

La lista de agentes no solo se encargara de capturar los mensajes sino de interpretarlos. Acto seguido avisa al replicador de claves correspondiente que genere un mensaje con la clave que será auto-cifrable, es decir, que ya tendrá los permisos oportunos para ser cifrado. El cifrador además de cifrar con la clave privada correspondiente también dará al resultado los permisos necesarios para que el mensaje sea enviado.

Veamos uno a uno cada proceso y que resulta después de cada operación.

Vamos a describir este sistema hacía atrás, es decir, partiendo de la forma del mensaje que debe de recibir un Sistema para que se pueda dar de alta la clave pública de un cierto sistema, definiendo los ambientes y permisos que necesita para llegar hasta el sistema.

La ordenes que recibirá este ambiente priv no será solamente para que se descifre, sino tambien para que se ruten automáticamente lo sub-productos a los ambientes 'cerrojo' o 'llavero' según corresponda

(v acido)priv[ClavePriv[....] | open acido | open ordenes]

Para evitar que el mensaje al descifrarse salga del cifrador sin que las correspondientes claves esten ya rutadas y por tanto no puedan ser asimiladas por los ambientes correspondientes, el ambiente que tendremos dentro de ClavePriv, no tendrá los permisos 'open AgenteSalida' desde el principio, para evitar que salga fuera del

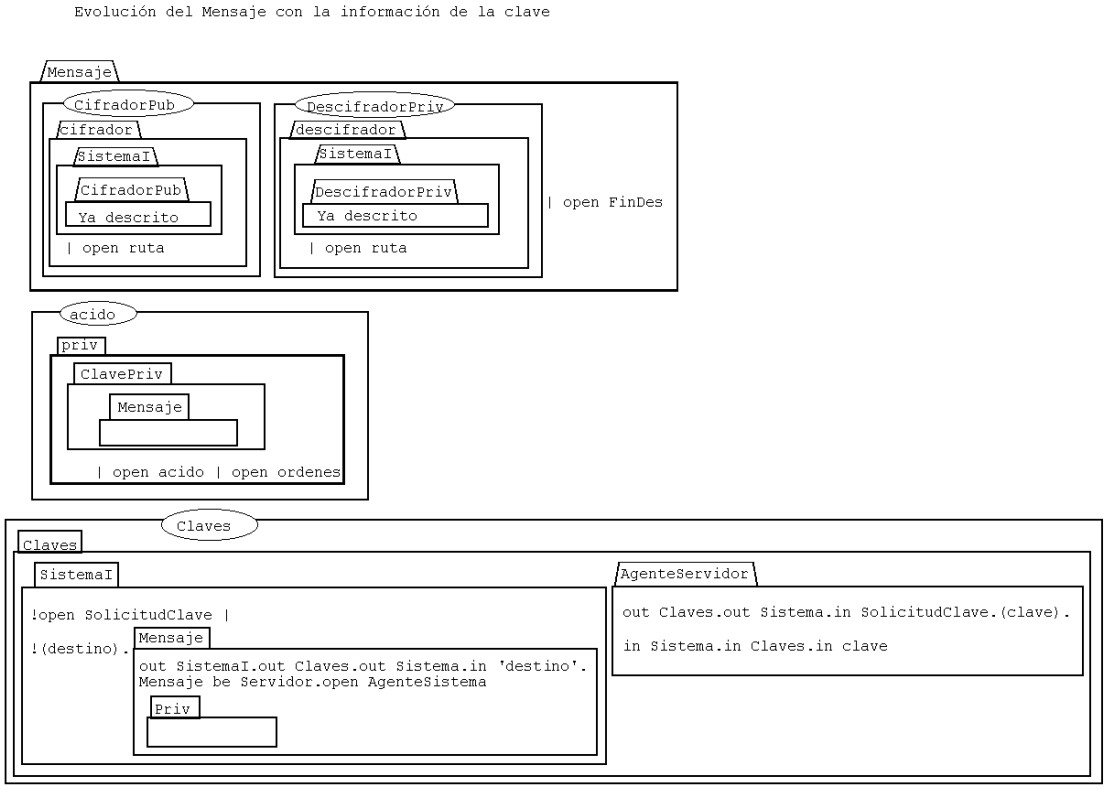
descifrador antes de que se hayan introducido las rutas para las claves. Por otra parte será necesario que los ambientes cifrador y descifrador que distinguen la clave para cifrar de la de descifrar, contengan un permiso ‘open ruta’ para que puedan ser conducidos dentro de los ambientes que las almacenarán. Además dentro de los ambientes cerrojo y llavero necesitaremos permisos ‘open cifrador’ y ‘open descifrador’ respectivamente para que las claves se almacenen correctamente.

Mensaje[cifrador[open ruta | SistemaI[CifradorPub[….]]] | descifrador[open ruta | SistemaI [DescifradorPriv[….]]] | open finDes]

ordenes[in priv. open AgenteLlavero.<Servidor>.open AgenteDescifradorPriv | ruta[in Mensaje.in cifrador.finCif[out cifrador.open AgenteSalida.open AgenteSalida] | in Cerrojo] | ruta[in Mensaje.in descifrador.finDes[out descifrador.open finCif] | in Llavero] | open finDes]

Viendo la forma del mensaje que recibimos del servidor, hemos de recordar que estaba cifrado cuando llegó, así pues al llegar tendría esta forma.

(v acido)priv[ClavePriv[Mensaje[...]] | open acido | open ordenes]
donde Mensaje es el ambiente descrito arriba.



Este mensaje debe de ser capturado por el sistema, por lo tanto antes de entrar en el sistema tendría la forma

Servidor[open AgenteSistema | (v acido)priv[...]] donde priv es el ambiente anterior.

Cuando fue enviado desde Servidor tendría la siguiente forma de acuerdo con los estándares de envío y recepción de mensajes que describimos antes, tendría esta forma:

Mensaje[out Sistema.out Servidor.in SistemaI.Mensaje be Servidor.open AgenteSistema | (v acido)priv[...]]

Donde SistemaI es el nombre que identifica al sistema destino.

Este mensaje puede ser prácticamente guardado así, salvo por el echo de que hay algo que no tienen todos los mensajes enviados por el servidir, que son:

La clave que vamos a enviar.

El nombre del sistema a quien enviar la clave. (ServidorI)

Para la primera cuestión almacenaremos en lugares diferentes según el nombre del propietario, de la clave que contiene el mensaje. Y para el segundo problema haremos que el replicador que crea estos mensajes, lea antes el nombre del Sistema destinatario de la clave.

Tendremos entonces que el almacenamiento de clave quedaría así:

(v Claves)Claves[AgenteServidor[*1*] | SistemaI [¡open solicitudClave | ¡(destino). Mensaje[out SistemaI.out Claves.out Sistema.out Servidor.in (destino).Mensaje be Servidor.open AgenteSistema | (v acido)priv[...]]]]

El permiso open SolicitudClave lo usaremos para que la información del sistema que

Falta por describir el comportamiento del AgenteClaves, que se encarga de recoger las solicitudes de clave que le llegan al cifrador.

Las solicitudes tendrán información en forma de salida asíncrona del sistema del cual queremos la clave, y del sistema que lo solicita

El mensaje de solicitud de la clave del ‘SistemaI’ desde un ‘SistemaJ’ será el siguiente.

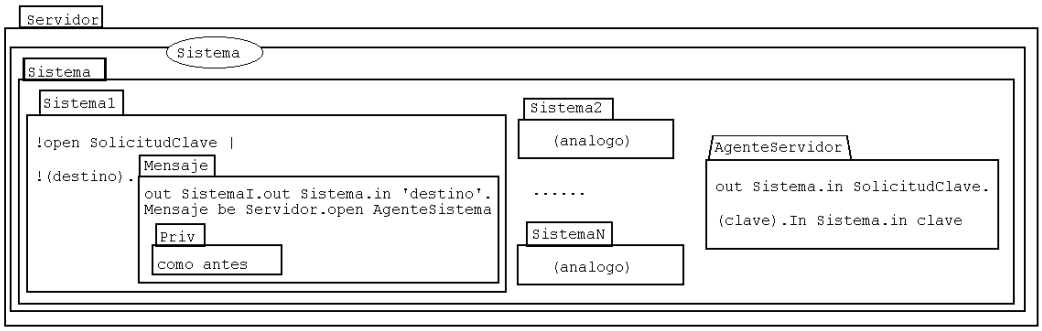
solicitudClave[out Sistema.out SistemaJ.in SistemaI.open AgenteServidor.<SistemaJ>.<SistemaI>]

AgenteServidor[out Claves.out Sistema.in solicitudClave.(clave).in Sistema.in Claves.in clave]

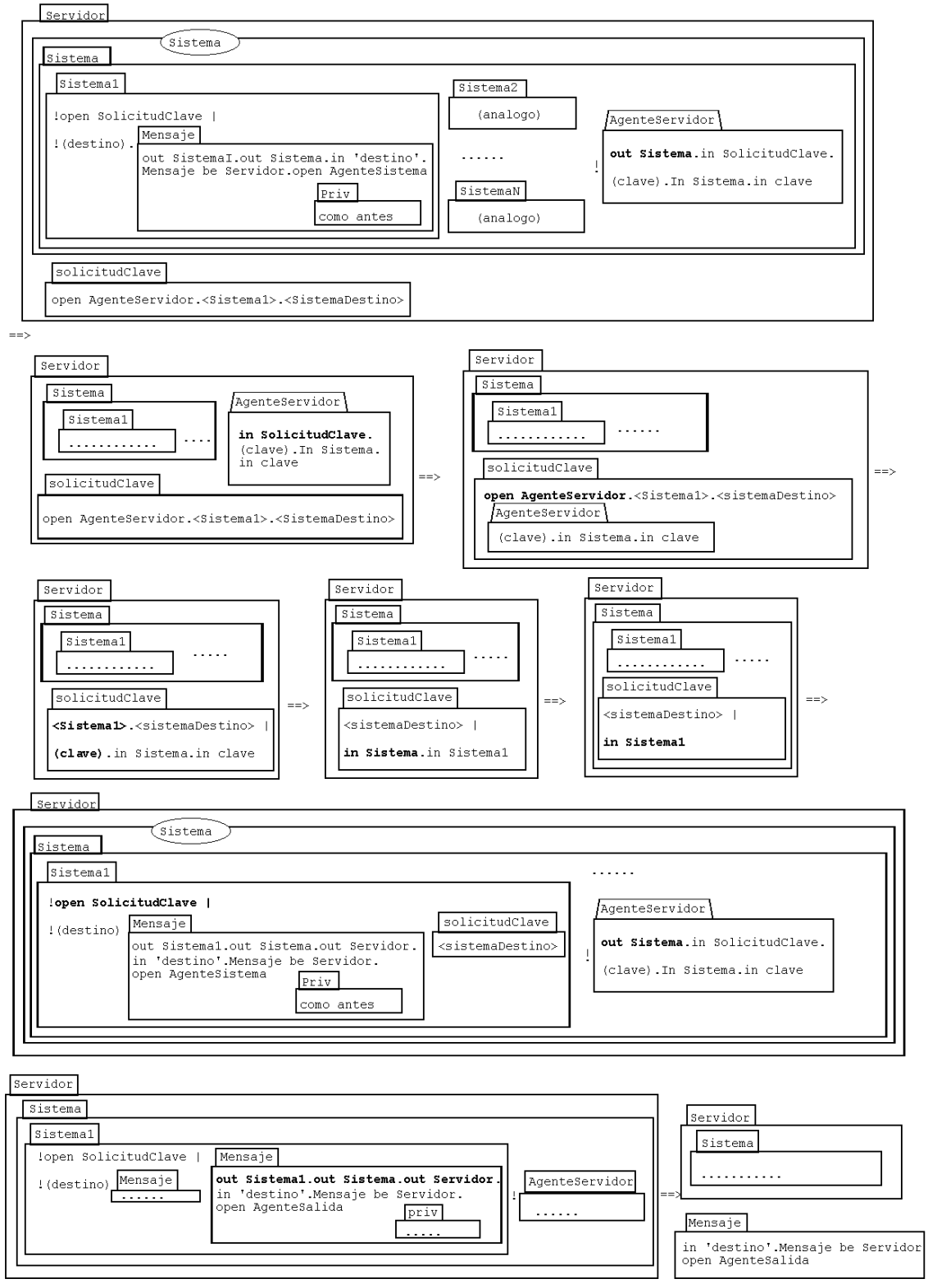
De esta forma ya tenemos completamente definido como se solicitan y reciben las claves públicas (firmadas por el servidor)

Realmente podemos fusionar los ambientes Sistema y claves en un solo ambiente Sistema, cuyo interior es igual al antiguo de Claves. Quedando el Servidor de la siguiente forma:

Absorción de una clave



Veamos ahora como el sistema Servidor reacciona ante una solicitud de clave, devolviendo la clave ya rutada hacia el sistema que la pidió.



13.17.- Conclusiones:

Hemos visto como se puede desarrollar un sistema bastante complejo de forma más o menos incremental, la seguridad que pretendíamos demostrar, la hemos demostrado en cada paso. Ya que hablar en conjunto de la seguridad en el sistema completo se nos hace practicamente inabordable.

La descripción del sistema, la hemos realizado así precisamente para que sea seguro, es decir, cuando justificabamos un tipo de descripción u otra, estabamos ya comentando la seguridad de esa funcionalidad.

Lo único que vamos a comentar además de lo ya dicho es que el sistema es seguro en la idea de que si llegamos al final del protocolo, es decir, intercambiamos con alguien una clave para establecer un canal de comunicación, este canal será seguro, o lo que es lo mismo la clave privada utilizada no será conocida por ningún sistema fuera de la comunicación. Lo que no podemos asegurar es que el protocolo termine, es decir un atacante esterior puede conseguir que no se establezca la comunicación pero nunca espiarla.

Apéndice A: Otros formalismos

El cálculo de ambientes ha sido desarrollado por Luca Cardelli y Andrew D. Gordon, los cuales se han provisto de otros muchos cálculos ya existentes. Algunos de esto son los siguientes:

- La máquina química abstracta: es un framework semántico más que un formalismo. Sus nociones básicas de reacción en una solución de una membrana que aísla subsoluciones recuerda la noción de ambiente. Sin embargo, las membranas no implican una protección fuerte, y no se da una idea de movilidad de subsoluciones.
- El cálculo-π: es un cálculo de procesos donde los canales pueden moverse a lo largo de otros canales. El movimiento de procesos se representa como el movimiento de canales que se refieren a los procesos. Por lo tanto, no existe una indicación clara de que sean los procesos por si mismos los que se muevan. Por ejemplo, si un canal cruza un firewall (es decir, si se comunica con un proceso que se supone que representa un firewall), no se tiene un sentido claro de que el proceso ha cruzado el firewall. De hecho, el canal puede cruzar algunos firewalls independientes, pero un proceso no puede estar en todos esos sitios a la vez. De todas manera, muchos conceptos y técnicas fundamentales del cálculo-π subyacen en este trabajo.
- Enriquecimiento del cálculo-π con direcciones: esto es algo que se ha estudiado con el interés de capturar nociones de computación distribuida. En su forma más simple, se añade un espacio de direcciones, y las operaciones pueden ser indexadas por la dirección por la que son ejecutados. Riely y Hennesy y Sewell propusieron versiones del cálculo-π extendido con primitivas que permitían a los cálculos migrar entre distintos nombres de direcciones. En este trabajo, el énfasis se ponía en desarrollar sistemas de tipos para el cálculo móvil basados en la existencia de sistemas de tipos para el cálculo-π. El sistema de tipos de Riley y Hennesy regulaba el uso de nombres de canales de acuerdo con el permiso representado por tipo. El sistema de tipos de Sewell diferenciaba entre canales locales y remotos para la implementación eficiente de la comunicación
- El cálculo-unión: es una reformulación del cálculo-π con una noción más explicita de los lugares de interacción; con esto se ayuda bastante en la construcción de implementaciones distribuidas de mecanismos de canales. El cálculo-unión distribuido añade una noción de dirección de nombres, con el mismo propósito, en esencia, que en el cálculo, de ambientes, y una noción en el fallo en la distribución. Las direcciones en el cálculo-unión distribuido forman un árbol, y los subárboles puede migrar de una parte del árbol a otra. Una diferencia significativa con respecto al cálculo de ambientes es que el movimiento puede tener lugar directamente de cualquier dirección activa a cualquier otra dirección conocida.
- Llinda: es una formalización de Linda usando técnicas de cálculo de procesos. Como en las versiones distribuidas de Linda, Llinda tiene múltiples espacios espacio de tuplas distribuidos. Esta multiplicidad es

muy similar en espíritu a los múltiples ambientes, pero los espacios de tuplas de Linda no se anidan, y no existen restricciones para el acceso a a un espacio de tuplas desde otro espacio de tuplas cualquiera.

- Un creciente grupo de literatura se está concentrando en la idea de añadir direcciones discretas al cálculo de procesos y considerando fallos para estas direcciones. Esta aproximación pretende modelar los tradicionales entornos distribuidos junto con algoritmos que toleren fallos en nodos.
- El cálculo spi es una extensión del cálculo-π con primitivas de encriptación. Sin embargo, estás primitivas no parecen ser necesarias para el cálculo de ambientes dado que algunas de las motivaciones del cálculo spi ya se cubren con la noción de encapsulamiento de un ambiente.

Apéndice B: Formalización del cálculo-π asíncrono

A lo largo de la parte de cálculo de ambientes y más concretamente en el capítulo 2, hemos visto que la base de la que se partía era del cálculo-π.

El cálculo-π es un pequeño pero extremadamente expresivo lenguaje de programación. Fue diseñado como una base para la programación concurrente, así como el cálculo-λ lo es para la programación secuencial. El cálculo-π se fundamenta en la idea de que en principio, cualquier programa distribuido se puede explicar en términos de un intercambio de nombres sobre canales de comunicación de nombres.

Los programa del cálculo-π son sistemas de procesos independientes y paralelos que se sincronizan a través de mensajes que se pasan en estos canales de comunicación de nombres. La capacidad de comunicación de un proceso depende por lo tanto de los canales de comunicación que dicho proceso conozca. Los canales se pueden restringir de manera que sólo ciertos procesos se puedan comunicar en ellos.

La principal diferencia existente entre el cálculo-π y otros cálculos que ya existían anteriormente es que el ámbito de una restricción puede cambiar durante el proceso. Cuando un proceso manda un nombre restringido como un mensaje a un proceso que se encuentra fuera del ámbito de la restricción, el ámbito de esta restricción aumenta de manera que ahora también abarca el proceso que a recibido el mensaje. Por lo tanto la capacidad de comunicación de un proceso, es decir, los distintos canales que conoce, pueden cambiar a lo largo de la vida de dicho proceso por el método que acabamos de comentar.

Con estas ideas ya mostradas, podemos ver que una de las utilidades que podemos encontrar en el cálculo-π es la de la criptografía. La idea es que simplemente con las restricciones sobre canales y la posibilidad de incrementar el ámbito de un nombre, podemos formular un modelo de posesión y comunicación de secretos.

Consideramos una formalización donde los nombres n, acotados por restricciones, son distintos de las variables x acotadas por prefijos de entrada. Tenemos funciones separadas fn y fv para nombres libres y variables libres respectivamente.

Cálculo-π asíncrono	
P, Q ::=	procesos
(νn)P	ocultamiento
P | Q	composición
!P	replicación
M(x).P	acción de entrada
M⟨M'⟩	acción asíncrona de salida
M ::=	expresiones
x	variable
n	nombre

Nombres libres y variables libres	
$fn((\nu n)P) \triangleq fn(P) - \{n\}$	$fv((\nu n)P) \triangleq fv(P)$
$fn(P \mid Q) \triangleq fn(P) \cup fn(Q)$	$fv(P \mid Q) \triangleq fv(P) \cup fv(Q)$
$fn(!P) \triangleq fn(P)$	$fv(!P) \triangleq fv(P)$
$fn(M(x).P) \triangleq fn(M) \cup fn(P)$	$fv(M(x).P) \triangleq fv(M) \cup (fv(P) - \{x\})$
$fn(M\langle M' \rangle) \triangleq fn(M) \cup fn(M')$	$fv(M\langle M' \rangle) \triangleq fv(M) \cup fv(M')$
$fn(x) \triangleq \varnothing$	$fv(x) \triangleq \{n\}$
$fn(n) \triangleq \{n\}$	$fv(n) \triangleq \varnothing$

Congruencias estructurales	
$P \equiv P$	Reflexión
$P \equiv Q \Rightarrow Q \equiv P$	Simetría
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	Transitividad
$P \equiv Q \Rightarrow (\nu n)P \equiv (\nu n)Q$	Ocultamiento
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	Paralelismo
$P \equiv Q \Rightarrow !P \equiv !Q$	Réplica
$P \equiv Q \Rightarrow M(x).P \equiv M(x).Q$	Input
$P \mid Q \equiv Q \mid P$	Conmutación de paralelismo
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	Asociación de paralelismo
$!P \equiv P \mid !P$	Paralelismo de réplica
$(\nu n)\ (\nu m)P \equiv (\nu m)(\nu n)P$	Restricción de ocultamiento
$(\nu n)(P \mid Q) \equiv P \mid (\nu n)Q$ si $n \notin fn(P)$	Ocultamiento de paralelismo
$(\nu n)P \equiv P$ si $n \notin fn(P)$	Ocultamiento de fn

Reducciones	
$n\langle m \rangle \mid n(x).P \rightarrow P\{x \leftarrow M\}$	Comm
$P \rightarrow Q \Rightarrow (\nu n)P \rightarrow (\nu n)Q$	Ocultamiento
$P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R$	Paralelismo
$P' \equiv P, P \rightarrow Q, Q \equiv Q' \Rightarrow P' \equiv Q'$	$\equiv$

Apéndice C: Codificando el cálculo-π

En el apéndice B hemos visto una formalización del cálculo-π asíncrono dada la importancia que tenía como base del cálculo de ambientes. Pues bien, podemos ver que el cálculo-π se puede codificar a partir del cálculo de ambientes. Esta codificación es relativamente sencilla dadas las primitivas de entrada/salida. Esto da muestra de la expresividad del cálculo de ambientes. Primero discutiremos como representar el nombre de canales: esta es la idea clave de toda la traducción.

Un canal lo podemos representar simplemente como un ambiente: el nombre del canal es el nombre del ambiente. La comunicación en un canal se representa por la comunicación local dentro del ambiente. Un nombre convencional, io, se usa para transporta entrada y salida requerida en el canal. El canal abre todas estas peticiones y las deja interactuar.

$$
\begin{array}{lll}
\text{buf n} & \triangleq \text{ n[!open io]} & \text{buffer del canal} \\
\text{(ch n)P} & \triangleq (\nu n)(\text{buf n} \mid P) & \text{nuevo canal} \\
\text{n(x).P} & \triangleq (\nu n)(\text{io[in n.(x).p[out n.P]]} \mid \text{open p}) & \text{canal de entrada} \\
\text{n}\langle M\rangle & \triangleq \text{ io[in n. } \langle M\rangle] & \text{canal asíncrono de salida}
\end{array}
$$

Estas definiciones satisfacen la reducción esperada $n(x).P \mid n\langle M\rangle \rightarrow^* P\{x \leftarrow M\}$ en la presencia de un buffer del canal buf n:

$$
\begin{array}{l}
\text{buf n} \mid \text{n(x).P} \mid \text{n}\langle M\rangle \\
\equiv (\nu n)(\text{n[!open io]} \mid \text{io[in n.(x).p[out n.P]]} \mid \text{open p} \mid \text{io[in n.}\langle M\rangle]) \\
\rightarrow^* (\nu n)(\text{n[!open io} \mid \text{io[(x).p[out n.P]]} \mid \text{io[}\langle M\rangle]] \mid \text{open p}) \\
\rightarrow^* (\nu n)(\text{n[!open io} \mid \text{(x).p[out n.P]} \mid \langle M\rangle] \mid \text{open p}) \\
\rightarrow (\nu n)(\text{n[!open io} \mid \text{p[out n.P}\{x \leftarrow M\}]] \mid \text{open p}) \\
\rightarrow (\nu n)(\text{n[!open io]} \mid \text{p[P}\{x \leftarrow M\}] \mid \text{open p}) \\
\rightarrow (\nu n)(\text{n[!open io]} \mid \text{P}\{x \leftarrow M\}) \\
\equiv \text{buf n} \mid \text{P}\{x \leftarrow M\}
\end{array}
$$

Podemos usar las definiciones de arriba de canales para fijar la comunicación en los canales con nombre con el cálculo de ambientes (proveer el nombre io no se usa con otro propósito). La comunicación en estos canales con nombre sólo funciona con un único ambiente. En otras palabras, desde el punto de vista del cálculo de ambientes, un proceso del cálculo-π siempre habita un único ambiente. Por eso, la noción de movilidad en el cálculo-π (comunicación de nombres sobre canales con nombre) es diferente de la noción de movilidad que tenemos con el cálculo de ambientes.

A continuación mostramos la codificación del cálculo-π asíncrono con el cálculo de ambientes. Cada proceso de alto nivel se traduce en un contexto de un conjunto de nombres S que, en particular, puede ser tomado como un conjunto de nombres libres de un proceso. Para entender mejor esta traducción es conveniente ver el apéndice B donde se da una formalización del cálculo-π asíncrono.

Codificación de cálculo-π asíncrono			
$\langle\langle\ P\ \rangle\rangle S$	$\triangleq$	$\langle\langle\ S\ \rangle\rangle \mid \langle\langle\ P\ \rangle\rangle$	donde S es un conjunto de nombres
$\langle\langle\ \{n_1, \ldots, n_k\ \rangle\rangle$	$\triangleq$	$n_1[!\text{open io}] \mid ... \mid n_k[!\text{open io}]$	
$\langle\langle\ (\nu n)P\ \rangle\rangle$	$\triangleq$	$(\nu n)\ (n[!\text{open io}] \mid !\langle\langle\ P\ \rangle\rangle)$	
$\langle\langle\ P \mid Q\ \rangle\rangle$	$\triangleq$	$\langle\langle\ P\ \rangle\rangle \mid \langle\langle\ Q\ \rangle\rangle$	
$\langle\langle\ !P\ \rangle\rangle$	$\triangleq$	$!\langle\langle\ P\ \rangle\rangle$	
$\langle\langle\ M(x).P\ \rangle\rangle$	$\triangleq$	$(\nu n)\ (\text{io}[\text{in M.}(x).p[\text{out M.}\langle\langle\ P\ \rangle\rangle]] \mid \text{open p})$	
$\langle\langle\ M\langle M'\rangle\ \rangle\rangle$	$\triangleq$	$\text{io}[\text{in M.}\langle M\rangle]$	

Apéndice D: Método de generación de los distintos resultados posibles por parte del conversor

Una vez se han tratado los tipos declarados y los distintos parámetros que puede usar cada proceso llega la hora de generar todos los posibles para cada proceso.

Para el caso de los parámetros, ya sean los que se declaren en la cabecera de un proceso, bien los que ya aparezcan por primera vez en el cuerpo del proceso se realiza una vuelta atrás para generar todos los valores posibles para un parámetro dado, siempre y cuando este parámetro no haya sido declarado antes, o salga del ámbito de visibilidad del anterior y haya que generar todos sus posibles valores de nuevo. Una vez que ha generado un nuevo valor se llama recursivamente al mismo método para generar los valores del parámetro siguiente, esto en caso de que lo siguiente que venga en el cuerpo del proceso sea un parámetro, si no lo es, se llamará al método correspondiente para seguir generando el resultado final.
Cuando se regrese de esa llamada, se generará un nuevo valor para el parámetro, y se continuará tratando el código del cuerpo que le sigue.
Sin embargo, puede suceder que tal valor haya sido generado antes por un parámetro igual al actual, y solamente haya que tomar su valor. Eso supone la necesidad de almacenar los valores generados para una iteración en el proceso recursivo concreta. Dicho almacenamiento de valores se lleva a cabo por medio de una estructura como un array o un vector de valores.

Puede darse el caso de que el parámetro haya aparecido antes (el parámetro es igual al que se va a tratar ahora) pero fuera del ámbito de visibilidad del actual, por lo que, en tal caso, se han de generar de nuevo los valores. Para resolver este problema se hace uso de la comprobación del nivel en el que han aparecido.
Si el parámetro que ahora se procesa está en un nivel superior al que se encontraba el parámetro igual a éste previo, se generan sus valores. Esto es debido a que el parámetro anterior no tiene vigencia fuera de su ámbito, y el parámetro actual al estar por encima sí lo está. Para el caso en que dicho parámetro hubiera aparecido con anterioridad, y que el nivel de éste esté por debajo del anterior o igual, habría que consultar si el anterior le corresponde viendo si pertenece a un ancestro común en el camino que lleva del nivel actual al más alto.

El tratamiento de las estructuras de selección se hace como sigue:
El código de la condición evaluada como cierta se añade a la secuencia de señales que compondrán el proceso. El código de la condición evaluada como cierta antecederá a las señales que siguen a la estructura de selección en el "nivel" de código en la que ésta se encuentra.
Para el caso en que ninguna condición se cumpla (eso implica que la estructura no consta de un else), se continúa generando el código del proceso tratando a la estructura de selección como si no existiese.

A la hora de recorrer el código del proceso para generarlo se hace lo siguiente. Se recorre el cuerpo del proceso de forma normal generando valores de los parámetros si se encuentra una señal con parámetros y fuera necesario (todo esto por medio de una vuelta atrás para generar todos los valores), y si se encuentra una estructura de selección se profundiza en ella. Antes de esto, guardamos información sobre la selección en la que nos encontrábamos, así como el término o elección dentro de la

misma y el elemento siguiente a tratar dentro de la elección en una pila. Una vez que se entra en el if se recorre de forma normal recorriendo todas las selecciones que lo componen, y para cada selección todas las posibles elecciones de la misma. Cuando hemos llegado al final de una elección dada para una selección cualquiera, hemos de desapilar la información de por dónde tenemos que continuar en el nivel de código superior al actual, lo que nos quedaba por tratar donde lo dejamos una que se entró en la estructura de selección. Cuando se ha llegado al final del código de una elección que se encuentra en el nivel más alto de todos los existentes, se escribe el texto generado, y se regresa recursivamente para ir generando todas las distintas posibilidades.

Estructuras de almacenamiento utilizadas

La estructura de datos utilizada para almacenar el cuerpo de un proceso es una estructura que almacena todo el cuerpo asociado a una estructura de selección.
Se considera el cuerpo de un proceso como un if sin condición que tiene como cuerpo el código del proceso.
Esta estructura empleada es un registro que tiene los siguientes campos:
_El número de selecciones que consta cada estructura de selección.
_ Para cada una de éstas se almacena el código de su condición, salvo que no la tenga (caso del else).
_El número de elecciones o términos
_Para cada selección y cada término se almacena el orden en que aparece cada elemento de dicho término.
_Para cada selección y cada término se almacena el texto de las señales "simples", es decir, señales que no tienen parámetros.
_Para cada selección y cada término se almacena los parámetros de la señal no simple iésima; y para ésta, los parámetros que la componen.
_Para cada selección y cada término se almacena una estructura como la que ahora se describe para almacenar las estructuras de selección que aparezcan.

Al margen de ésta, la estructura clave para almacenar el código de los procesos, se utilizan otras (arrays) para guardar los distintos tipos y valores correspondientes declarados por el usuario, la información de los procesos relativa a sus nombres, parámetros de sus cabeceras, tipos y valores de los mismos, si es necesario definirlos, así como, los parámetros, tipos de los mismos y valores que utilizan de aquéllos declarados al margen de la cabecera del proceso.

Apéndice E: Notas del conversor versión 1.1

Características del conversor

El conversor hace la conversión de ficheros de texto (extensión del fichero .txt) escritos en CCS no básico, esto es, en el que hay declarados parámetros y estructuras de selección a CCS básico, a un fichero con extensión .ccs, que pueden utilizar herramientas como North Carolina Concurrency WorkBench. Para ello, en el caso de las estructuras de selección se irá añadiendo el código de la misma al que se encuentra en niveles superiores a tal estructura si la condición se cumple; mientras que para el caso de los parámetros, se realiza la concatenación de los valores de los parámetros que éstos tienen en un instante dado al nombre de la señal que utiliza dichos parámetros.

Dado que la herramienta anteriormente mencionada realiza la distinción entre letras en mayúscula y minúscula, este conversor también la realizará.

Todas las palabras, así como caracteres aislados, que aparecen en el documento en letra cursiva corresponden a palabras reservadas o símbolos reservados o marcas de obligada utilización.

Declaración de tipos

El conversor permite la declaración de tipos. Cuenta con los tipos predefinidos int, char y bool.

En el caso de que se vayan a emplear otros tipos, aparte de los predefinidos habría que declararlos; para ello se haría uso de la sintaxis definida a continuación:

Sintaxis

type Nombre_Tipo = *{*valor1*, ... ,*valorN*}*

Todo el conjunto de tipos declarados irá precedido de la palabra reservada TYPE y seguido de la palabra reservada de fin de declaración de tipos FTYPE.

Así, en caso de declararse tipos se tendrá lo siguiente:

TYPE

type Nombre_Tipo1 = *{*valor1*, ... ,*valorN*}*
.....
type Nombre_TipoN = *{*valor1*, ... ,*valorN*}*

FTYPE

Declaración de procesos

Se tienen que declarar todos aquellos procesos que tengan parámetros en su cabecera, así como todos aquéllos que utilicen variables o párametros que pueden ser

recibidos o enviados a otros procesos y que sean distintos de los declarados en la cabecera; todo esto, en el caso de que el proceso cuente con ella.

La sintaxis para la declaración de un proceso que cuente con cabecera es la siguiente:

proc Nombre_Proceso*(*tipoParámetro1*, ... ,*tipoParámetroN*)*

Los tipos int y char exigen la declaración del rango de valores que pueden tomar.

Si el proceso define en su declaración de cabecera tipos int o char hay que declarar su rango. Para ello, se utilizará la sintaxis descrita a continuación:

*/(*valorMínimoParámetroConRango1 - valorMáximoParámetroConRango1*, ... ,*
valorMínimoParámetroConRangoM - valorMáximoParámetroConRangoM*)*

donde valorMínimoConRango determina cuál es el extremo inferior del rango de valores que va a tomar un parámetro con tipo int o char, mientras que ValorMáximoConRango determina el máximo. Ambos extremos van separados por un guión ('-').

Los pares de valores van encerrados entre paréntesis precedidos por el carácter '*/*'.

Si, además, el proceso utiliza otros parámetros declarados dentro del cuerpo del mismo distintos de los de la cabecera hay que declararlos también, y especificar su rango de valores si su tipo es int o char.
Al efecto se ha de emplear la sintaxis que vendrá a continuación del proceso. Si no declara parámetros en su cabecera, sólo aparecerá el nombre del mismo; por el contrario, si sí la tiene, aparecerá todo lo anteriormente descrito, esto es:

proc Nombre_Proceso*(*tipoParámetro1*, ... ,* tipoParámetroN*)*
*/(*valorMínimoParámetroConRango1 -
valorMáximoParámetroConRango1*, ... ,*
valorMínimoParámetroConRangoM –
valorMáximoParámetroConRangoM*)*

La sintaxis para la declaración de estos nuevos parámetros es la siguiente:

*; (*nombreParámetro1*:*tipo_Parámetro1*, ... ,*nombreParámetroN*:*tipo_ParámetroN*)*

Importante: No dejar espacios entre el nombre del parámetro y la marca ':', así como entre ésta y el tipo del parámetro. Igualmente, no dejar espacios entre el paréntesis que abre y el nombre del primer parámetro declarado, así como entre el tipo del último parámetro y el paréntesis que cierra.

Estas exigencias de la sintaxis serán rebajadas en una versión posterior.

Si alguno de estos tipos exige la definición del rango de valores que puede tomar se hace igual que para el caso de la lectura de rango para un parámetro de cabecera. La sintaxis es la siguiente:

*/(*valorMínimoParámetroConRango1-valorMáximoParámetroConRango1*, ... ,*
valorMínimoParámetroConRangoM-valorMáximoParámetroConRangoM*)*

Importante: Para este caso, es imprescindible, al igual que antes, que no aparezcan espacios entre ningún carácter; es decir, entre los valores de los rangos y el guión ('-'), al igual que entre los valores extremos y los paréntesis, y entre el carácter slash *'/'* y el paréntesis de apertura *'('*.

Estas exigencias de la sintaxis serán rebajadas en una versión venidera.

Como para el caso de la declaración de tipos todo el bloque de procesos irá precedido de la palabra reservada *HEAD* y finalizado por la palabra reservada *FHEAD*.

Un ejemplo de código de lo explicado hasta ahora es el siguiente:

TYPE

type Dia = *{*L,M,X,J,V,S,D*}*
type Vocales = *{*a,e,i,o,u*}*

FTYPE

HEAD

proc cero*(*Dia*,*Vocales*)*
proc uno*(*int*,*char*,*Dia*) / (*7 – 5*,* m – t*) ; (*numero*:*int*)/(*15-21*)*
proc dos *; (*num*:*int*,*vocal*:*Vocales*)/(*15-21*)*

FHEAD

Cuerpo de los procesos

Una vez acabada la declaración de los procesos se pasa a la escritura del código del proceso. Esto se hará como en CCS básico.

Dentro del código del proceso, en los parámetros se pueden pasar parámetros declarados en la cabecera, o fuera de la misma que aparezcan en la parte de declaración anterior explicada, de otros parámetros aparte de los de la cabecera.

También pueden ser pasadas constantes, que irán encerradas entre corchetes, [literal] ,o hacerse la suma o sustracción de una literal y un parámetro. No se permiten en esta versión, la suma o diferencia de dos parámetros o de más de dos elementos.

Para el caso de que la suma o diferencia exceda el rango de valores se permiten dos políticas, la operación módulo y la operación Techo/Suelo, que asigna el máximo valor del rango, en caso de que la operación exceda el rango por encima, y el mínimo si es por debajo.

La estructuras de selección pueden contar con una o más selecciones, pudiendo ser la última, si el total de éstas es mayor que uno, bien *else*, bien *elsif*.

Las condiciones precedidas de *if* o *elsif* exigen una condición, mientras que *else* puede no tenerla.

La utilización de estructuras de selección exige el uso del carácter '[' al abrir la estructura y ']' al cerrarla. Así, se podrían tener las siguientes estructuras:

```
[if (condición)                    [if (condición)
 cuerpo de la condición]             cuerpo de la condición
                                   else
                                     cuerpo de la condición]
```

```
[if (condición)                      [if (condición)
  cuerpo de la condición]              cuerpo de la condición
 elsif (condición)          // elsif 1  elsif (condición)          // elsif 1
  cuerpo de la condición                cuerpo de la condición
 .....                                 .....
elsif (condición)           // elsif N  elsif (condición)          // elsif N
  cuerpo de la condición]               cuerpo de la condición
                                        else
                                        cuerpo de la condición]
```

Es necesario dejar un espacio entre la marca *if/elsif* y las condiciones.

Se recuerda que todas las señales han de ir separadas por un punto '.', y que el fin de proceso es nil, en North Carolina Concurrency WorkBench.

Las condiciones han de ser escritas en notación prefija, es decir, primero el operador y luego los operandos.
Los operadores lógicos son los siguientes: & (and), | (or) y ¡ (not, símbolo de apertura de exclamación). El operador not es unario.
Los operadores relacionales son binarios, y son los siguientes: <, <=, > y >=. Como se observa, no se permite utilizar operadores como la adición o sustracción (+,-).

Un ejemplo del cuerpo de código de un proceso es el siguiente:

```
proc uno(num,letra,dia) = 'envia(dia).entrada(numero).[if (< numero num)
                                                        cadena1 + cadena2
                                                      elsif (& (= letra r) (¡ (= num 5) ) )
                                                        otro.texto(numero)
                                                      else
                                                         'envia(letra)]. 'otrasalida(dia,numero)
                                  + entrada2(numero + [2], letra + [A]).adios([cadena]).nil
```

La forma de contar los elementos que constituyen una elección se muestra en el siguiente ejemplo:

```
‘envia(dia).salida(numero,letra).[if ...].texto.texto2(num).[if...].nil   +  otro.texto
_____ __  ____ __________  ___  ___ ____ ___  ___ __      ___ ___
  1   2   3        4        5    6   7    8    9  10       1   2
```

La primera elección cuenta con 2 estructuras de selección, 3 fragmentos de código con parámetros y 5 señales simples, de las cuáles 3 emplean parámetros.

Mejoras respecto de la versión 1.0

Se permite la utilización de la suma/resta de una constante y una literal, o la asignación de una constante a una señal que emplee parámetros.

Subsanados errores de la versión anterior como la liberación incorrecta de la memoria en algunas ocasiones, tratamiento de procesos cuyo único “elemento” es una estructura de selección o problemas al identificar el proceso en la estructura almacenada.

Posibilidad de definir el número máximo de tipos, de procesos, valores por tipo, numero de elecciones por proceso, número de selecciones por estructura de selección..., es decir, definir todas las constantes utilizadas que antes valían 10.

Limitaciones de esta versión

La suma/ resta anteriormente comentada sólo permite realizarse sobre dos elementos, es decir, no se permite que en un mismo parámetro aparezcan más de dos operadores.

No se admite la utilización de paréntesis en el código del proceso, salvo para el uso de parámetros, por lo que expresiones como la siguiente:

```
‘salida(adios + hola)
```

deben ser transformadas por el usuario en la siguiente:

```
‘salida.adios + ‘salida.hola
```

Se recomienda la utilización de esta segunda forma al usar la herramienta ya citada con anterioridad.

Este imposibilidad de la utilización de los paréntesis salvo para el caso de los parámetros, impide implicíta y explícitamente la utilización del operador de paralelo ‘|’.

El número máximo de condiciones de una estructura de selección es 10, mientras que la condición no puede tener, según el caso más de 7 condiciones anidadas, de modo que no se puede tener:

```
( < ( &( ¡ ( < ( >= ( | ( = ...  ))))))
```

El programa no es especialmente robusto ante la presencia de errores.

Se recomienda que en caso de utilización de un parámetro declarado con anterioridad en dos lugares distintos del texto de una misma elección, y que se encuentre en un nivel inferior o igual al actual, se declare otra variable distinta del mismo tipo y rango.

Las condiciones de las estructuras de selección no permiten hacer operaciones frecuentes, como la suma, dentro de la misma.

Características que incorporarán las versiones venideras

Mayor flexibilidad de la sintaxis.

Explicación más detallada de los errores que se puedan dar, que se añaden a los ya existentes. Mayor robustez.

Realizar operaciones como el de adición o sustracción dentro de una condición.

Posibilidad de leer restricciones o conjuntos de las mismas, como permite North Carolina Concurrency WorkBench, y añadirlas al código generado.

Tratamiento de paréntesis al margen de los parámetros, y, por tanto, la utilización del operador de paralelo 'l'.

Pruebas realizadas

Las pruebas realizadas hasta ahora emplean sólo dos niveles de texto, es decir, el texto del código principal y el de una estructura de selección. No se ha podido probar para estructuras de selección anidadas, por lo que no se garantiza su funcionamiento. A veces se suceden errores a la hora de liberar los recursos solicitados al sistema, por lo que tras una ejecución correcta se puede dar fallo del programa en intentos sucesivos, por lo que se recomienda cerrar el programa e intentarlo de nuevo. Si aun así se produce el error, se realizará la depuración del código y se intentará subsanarlo en una versión venidera.

El resultado de la ejecución de los ficheros adjuntos ha sido correcta para la versión actual.

Defectos a nivel de código y posterior resolución

Falta de comentarios en determinados lugares del código del programa.

No elección de nombres adecuados o que no coinciden con el del trabajo que efectúan los métodos.

Utilización del código de métodos declarados, en vez de la llamada a los mismos, correspondiente con la época en que éstos no fueron creados y que no se utilizaron posteriormente por falta de tiempo.

Dificultad a la hora de entender las condiciones de control de ciertos métodos debido a su complejidad.

CODIGO C++ DEL CONVERSOR CCS

A continuación mostramos el archivo de cabecera de el conversor CCS

```
//---------------------------------------------------------------------------

#ifndef UFormConvH
#define UFormConvH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Dialogs.hpp>
#include <Menus.hpp>
//---------------------------------------------------------------------------
class TForm1 : public Tform {
__published:   // IDE-managed Components
        TMainMenu *MainMenu1;
        TMenuItem *Covertirfichero1;
        TOpenDialog *OpenDialog1;
        TLabel *Label1;
        TSaveDialog *SaveDialog1;
        TComboBox *ComboBox1;
        TLabel *Label2;
        TLabel *Label3;
        TGroupBox *GroupBox1;
        TEdit *EditTipos;
        TLabel *LabelTipos;
        TLabel *LabelProcesos;
        TEdit *EditProcesos;
        TLabel *Label6;
        TEdit *EditValoresTipo;
        TEdit *EditElecciones;
        TLabel *Label7;
        TLabel *LabelSenales;
        TEdit *EditSenales;
        TLabel *Label10;
        TEdit *EditNumIfs;
        TLabel *Label11;
        TEdit *EditSelecciones;
        TLabel *Label12;
        TEdit *EditParam;
        TLabel *Label13;
        TEdit *EditParamBis;
        TLabel *Label4;
        TEdit *EditNumParamCabecera;
        TLabel *Label5;
        TEdit *EditNumConstantes;
        TLabel *Label8;
        TEdit *EditParamFueraCabecera;
```

```
        void __fastcall Covertirfichero1Click(TObject *Sender);
        void __fastcall ComboBox1Change(TObject *Sender);
private:        // User declarations

    // Estructura que almacena información de los procesos en el código principal
        struct Proceso{
          String nombre;
          int numParametros;
          bool simple;
          String * tipos;
          String * nombreParametro;
        };
        Proceso* estructura;
    // Estructura que almacena información de los tipos
        struct Tipo{
          String nombre;
          String* valores;
          int numValores;
        };
        Tipo* tipos;
    // Estructura que almacena información de los parámetros y características
    // declarados al margen de la cabecera
        struct LectDentro{
          String* nombreVar;
          String* tipoVar;
          String* rangoVal;
          int total;
        };
    // Estructura que almacena información de los procesos leídos en la parte
    // de declaración de cabeceras
        struct Cabecera{
          String proceso;
          String* parametros;
          String* rangos;
          int numParametros;
          LectDentro resto;
          bool hayResto;
        };
        Cabecera* cabeceras;

// Estructura que almacena sobre la posición por la que estamos recorriendo una
// elección concreta
        struct Indice{
          int indiceIf;
          int indiceParam;
          int indiceTP;
        };

        struct ListaDoble{
          String valor;
```

```
  String nombre;
  String tipo;
  int nivel;
};


struct Linea{
  String* partes;
  int numeroPartes;
};

struct LineaDeLinea {
  Linea* termino;
  int numeroPartes;
};
struct Orden {
  int* orden;
  int tamanio;
};

struct condicion {
  String operador;
  bool esNot;
  bool hayOperandos;
  String operando1;
  String operando2;
};
// Estructura que almacena el cuerpo de una estructura de selección
struct seleccion {
  Linea** textoPlano;
  LineaDeLinea** parametros;
  int numeroSelecciones;
  condicion** condiciones;
  int* numFragSelec;        // Número de fragmentos que tiene una selección
  Orden** ordenacion;
  seleccion*** ifs;
};

// Estructura que almacena información sobre la posición de retorn cuando se ha
// llegado al final de una elección y queda por recorrer en niveles superiores
struct Continuacion{
  seleccion nodo;
  int numTermino;
  int numComponente;
  int numSeleccion;
  int nivel;
};

struct ListaCont{
  Continuacion * cont;
```

```
        int tamanio;
      };

      seleccion raiz;
      int numTipos, numCabeceras, numProcesos,numIf, posCabecera, eof,
numConstantes;
      int TamArrayTipos,
TamArrayProcesos,TamArrayTP,TamArrayParam,TamArrayParamBis,
        TamArrayIfs,TamArraySelecciones,TamArrayValoresTipo,
TamArrayTerminos,

TamArrayNumParamCabecera,TamArrayOrdenacion,TamArrayParamFueraCabecera;
      bool hayTipos, hayCabeceras, finMas, primera, primeroEnEscribir;
      bool operacionModular,proceso_CCS_Basico;
      FILE* ficheroDestino;
      char* nombreFicheroDestino;

public:          // User declarations
      __fastcall TForm1(TComponent* Owner);
      bool leerNombre(FILE* fichero, int numeroProcesos, int &eof);
      bool leerParametros(FILE* fichero, int numeroProcesos, int &eof);
      void tomarNombreParametros(String parametros,int numeroProcesos);
      void leerTipos(FILE * fichero);
      void leerCabeceras(FILE * fichero);
      void leerDatosTipo(FILE* fichero,int numTipos);
      char leerDatosCabecera(FILE* fichero,int numCabeceras);
      void comprobarTipos();
      void leerRangos(FILE* fichero,int numCabeceras, int numParametros, bool*
arrayBool);
      bool esPorDefecto(String nombreTipo);
      bool analizarProceso(FILE* fichero,seleccion nodo, bool nodoPadre, int indSel,
char &ultCaracter, int &eof);
      char QuitarBlancos(FILE* fichero);
      char encuentraCaracter(FILE* fichero, char caracterBuscado, bool omitirCaracter
= false);
      void InsertaCaracter(String &cadena, char caracter, int limite = 0);
      char tomarSecuencia(FILE* fichero, String cadena, bool primera, seleccion
nodo,int indSel,int indiceTermino,int indIf,int posTermino);
      void leerParametroSecuencia(FILE* fichero,seleccion nodo, int indSel,int
indiceTermino,int indiceParam);
      char tratarIf(FILE* fichero,seleccion nodo,int indSel,int indiceTermino,int
indiceIf, int &eof);
      void vaciarCadena(String &cadena);
      void hacerComprobacion(int indice,int numProcesos);
      void escribirFragmento(int numProcesos, int numPartes, bool primera);
      void escribirParametros2(int numParametros, ListaDoble* valorParametro,int
indice,seleccion nodo);
      void escribirCodigo2(ListaDoble* valorParametro,int totalElemArrayVP,seleccion
nodo);
      char tomarResto(FILE* fichero, int numCabeceras);
```

```
        char QuitarBlancosCarro(FILE* fichero);
        bool esProceso(FILE* fichero,String &cadena);
        void escribirCadena(FILE* fichero, String cadena);
        void escribirParametros(FILE* fichero, int numProcesos, int numParte, int
numParametros, ListaDoble * valorParametro, int posCabecera,int indice);
        void escribirCodigo(FILE* fichero, int numeroProceso, int numPartes, ListaDoble
* valorParametro, int posCabecera,int totalElemArrayVP);
        void escribirProceso(seleccion nodo);
        int encuentraParametro(int numCabecera, int indice);
        void tomarValoresRango(int numCabecera, int posArrayRangos, int
posArrayTipos, int &extremo1, int &extremo2,bool &esInt);
        String cogerHasta(String cadena, char caracter);
        String cogerDesde(String cadena, char caracter);
        char tomarCaracter(String cadena, int pos);
        bool estaEn(ListaDoble* listaCadenas,int numElementos, String cadena, int
&pos,int nivel);
        bool generarIfs(seleccion nodo, int seleccionEscogida, int indiceTermino, int
indiceIf, ListaDoble* valorParametro,int totalElemArrayVP, ListaCont* continuacion,
String cadResultado, int nivel);
        void recorrerIf(seleccion nodo,int seleccionEscogida, ListaDoble*
valorParametro,int totalElemArrayVP,ListaCont* continuacion,String cadResultado,int
nivel);
        void analizarComponentes(seleccion nodo, int seleccionEscogida, int termino, int
contador, ListaCont* continuacion, String cadResultado, ListaDoble* valorParametro,
int totalElemArrayVP,Indice* indices,int nivel);
        void generarParteCodigoParametrizado2(seleccion nodo,int seleccionEscogida,int
termino,int contadorFrag,ListaDoble* valorParametro,int totalElemArrayVP,String
cadResultado,Indice* indices, int paramPasados, int nivel, ListaCont* continuacion);
        void tomarValoresRango2(int numCabecera, int posArrayRangos, int &extremo1,
int &extremo2,bool &esInt);
        int encontrarCabecera(int numeroProceso, bool salir = true);
        bool encontrarPosicion(String variable, int &posicion);
        bool buscarPosicionTipo(String* lista,String nombre, int tamanio,int& pos);
        void actualizarIndices(Indice* indices, seleccion nodo, int numSeleccion, int
numTermino, int numComponente);
        bool buscaElemEnFragmento(int posInicio,Orden ordenacion,int &distancia,int
codigoElem);
        bool hayIfEnFragmento(int posInicio,Orden ordenacion,int &distancia);
        void salidaProgramaError(int codigoError,String param1 = " ", String param2 = "
");
        String concatenar(String cadena1, String cadena2);
        void leerCondicion(FILE* fichero,seleccion nodo, int numSel, int posArray, bool
quitarParentesis);
        bool evaluarCondicion(seleccion nodo, int numeroIf, int posArray, ListaDoble*
lista, int tamLista);
        bool or(String cadena1,String cadena2);
        bool and(String cadena1,String cadena2);
        int posicionValorEnTipo(String valor, String nombreTipo);
        bool mayorIgual(String cadena1,String cadena2,String tipo);
        bool menorIgual(String cadena1,String cadena2,String tipo);
```

```
        bool mayor(String cadena1,String cadena2,String tipo);
        bool menor(String cadena1,String cadena2,String tipo);
        bool esSeleccion(FILE* fichero);
        bool esLetra (String cadena);
        bool esNumero (String cadena);
        bool pertenece(char caracter, String cadena);
        String cogerLiteral(String cadena);
        char cogerOperador(String cadena);
        String cogerParametro(String cadena);
        void liberarRecursos(seleccion nodo,bool todaEstructura,bool esRaiz);
        String operacion(String valorParametro, char operador, String literal, String tipo,
int extremoInf, int extremoSup);
        int encuentraPosParam(int posCabecera, String parametro,bool dentro);
        bool hayLiteral(String cadena);
        void escribirProcesoCCSBasico();
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

A continuación, se muestra el código del archivo .c del conversor.

```
//---------------------------------------------------------------------------

#include <vcl.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <ctype.h>

#pragma hdrstop

#include "UFormConv.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"

#define TiposPorDefecto 3
#define TPDExigenRangos 2
#define CaracterIgual '='
#define OmitirCaracter true
#define NoOmitirCaracter false

TForm1 *Form1;

//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
```

```
  ComboBox1->Items->Add("Operación Techo/Suelo");
  ComboBox1->Items->Add("Operación Módulo");
  operacionModular = false;
  proceso_CCS_Basico = true;
 }
//---------------------------------------------------------------------------

void __fastcall TForm1::Covertirfichero1Click(TObject *Sender) {
   // Lectura de los valores de las constantes especificadas por el usuario
   TamArrayTipos = (StrToInt) (EditTipos->Text);
   TamArrayTipos = max (TamArrayTipos,3); //3 = número de tipos por defecto
   TamArrayProcesos = (StrToInt) (EditProcesos->Text);
   TamArrayTP = (StrToInt) (EditSenales->Text);
   TamArrayParam = (StrToInt) (EditParam->Text);
   TamArrayParamBis = (StrToInt) (EditParamBis->Text);
   TamArrayIfs = (StrToInt) (EditNumIfs->Text);
   TamArraySelecciones = (StrToInt) (EditSelecciones->Text);
   TamArrayValoresTipo = (StrToInt) (EditValoresTipo->Text);
   TamArrayTerminos = (StrToInt) (EditElecciones->Text);
   TamArrayNumParamCabecera = (StrToInt) (EditNumParamCabecera->Text);
   TamArrayOrdenacion = TamArrayTP + TamArrayIfs + TamArrayParam;
   TamArrayParamFueraCabecera = (StrToInt) (EditParamFueraCabecera->Text);
   numConstantes = (StrToInt) (EditNumConstantes->Text);

   FILE *forigen, *fdestino;
   char caracter;
   int posicion = 0;
   numProcesos = 0;
   numIf = 0;
   numTipos = 3;
   finMas = false;
   primera = true;
   primeroEnEscribir = true;
   numCabeceras = 0;
   String cadena = " ";
   hayTipos = hayCabeceras = false;
   estructura = new Proceso[TamArrayProcesos];
   tipos = new Tipo[TamArrayTipos];
   tipos[0].nombre = "char";
   tipos[1].nombre = "int";
   tipos[2].nombre = "bool";
   cabeceras = new Cabecera[TamArrayProcesos];
   bool parametros;

   // Inicialización del nodo raíz
   raiz.numeroSelecciones = 1;
   raiz.ifs = new seleccion**[1];
   raiz.textoPlano = new Linea*[1];
   raiz.numFragSelec = new int[1];
   raiz.ordenacion = new Orden*[1];
```

```
raiz.parametros = new LineaDeLinea*[1]; // Damos diez fragmentos al proceso
// Fin de la inicialización del nodo raíz

OpenDialog1->Title = "Cargar fichero CCS";
OpenDialog1->Filter = "TXT files (*.txt)|*.txt";
OpenDialog1->FilterIndex = 1;
OpenDialog1->DefaultExt = "txt";
// Apertura del fichero fuente
if (OpenDialog1->Execute()) {
  FILE *ficheroOrigen, *ficheroDestino;
  char* nombreFicheroOrigen = OpenDialog1->FileName.c_str();
  SaveDialog1->Title = "Fichero destino CCS";
  SaveDialog1->Filter = "CCS files (*.ccs)|*.ccs";
  SaveDialog1->FilterIndex = 1;
  SaveDialog1->DefaultExt = "CCS";
  // Apertura del fichero destino
  if (SaveDialog1->Execute())
    nombreFicheroDestino = SaveDialog1->FileName.c_str();
  if ((ficheroOrigen = fopen(nombreFicheroOrigen, "rt")) == NULL)
    ShowMessage("Error al tratar de abrir el fichero origen.");
  else {
   if ((ficheroDestino = fopen(nombreFicheroDestino, "w+t")) == NULL)
     ShowMessage("Error al tratar de abrir el fichero destino.");
   else {
     //Lectura de procesos
     Label1->Caption = "Trabajando";
     bool leido = false;
     int num = cadena.Length();
     eof = 1; // EOF
     fread(&caracter, sizeof(caracter), 1, ficheroOrigen);
     int fin = 0;
     while (eof != 0 && fin == 0) {
       cadena[cadena.Length()] = caracter;
       if (cadena.Length() == 4) {
         // Si se lee proc, se trata el cuerpo del proceso en el código principal
         if (cadena == "proc" || cadena == "Proc") {
           bool continuar = true;
           while (continuar && eof != 0) {
             // Mientras haya más procesos se genera su cuerpo
             continuar = leerNombre(ficheroOrigen,numProcesos,eof);
             if (continuar)
               numProcesos = numProcesos + 1;
             ShowMessage(AnsiString(numProcesos) + estructura[numProcesos-
                                 1].nombre);
           }
           leido = true;
         }
         else if (cadena == "TYPE") {
           // Si viene la marca de tipos se leen
           leerTipos(ficheroOrigen);
```

```
            }
            else if (cadena == "HEAD") {
              // Si viene la marca de declaración de cabeceras se tratan
              leerCabeceras(ficheroOrigen);
              comprobarTipos();
            }
            else
              // Se añade el carácter a la cadena haciéndola avanzar de
              // derecha a izquierda
              for (int i = 1; i < cadena.Length(); i++) {
                caracter = cadena[i+1];
                cadena.Delete(i,1);
                cadena.Insert(caracter,i);
              }
          }
          else
            cadena.SetLength(cadena.Length() + 1);
          // Se añade el carácter a la cadena haciéndola avanzar de derecha a
          // izquierda
          if (leido && eof != 0) {
            leido = false;
            num = cadena.Length();
            for (int i = 1; i < cadena.Length(); i++) {
              caracter = cadena[i+1];
              cadena.Delete(i,1);
              cadena.Insert(caracter,i);
            }
          }
          if (eof != 0) {
            num = cadena.Length();
            eof = fread(&caracter, sizeof(caracter), 1, ficheroOrigen);
            num = cadena.Length();
          }
        }
        // Se cierra los ficheros utilizados
        fclose(ficheroOrigen);
        fclose(ficheroDestino);
        Label1->Caption = "Conversión satisfactoria. Esperando entrada de fichero";
      }
    } // Primer fichero abierto sin problemas
  }// Se ejecuta la apertura de fichero (OpenDialog)
}
//---------------------------------------------------------------------------
// Método que lee el nombre del proceso en el "código" del programa
bool TForm1:: leerNombre(FILE* fichero, int numeroProcesos, int &eof) {
  char caracter;
  String cadena = " ";
  // Nos posicionamos al principio del nombre
  fread(&caracter, sizeof(caracter), 1, fichero);
  // nos saltamos los blancos
```

```
 while (caracter == ' ')
   fread(&caracter, sizeof(caracter), 1, fichero);
 bool parar = false;
 while (caracter != ' ' && caracter != '(' && caracter != '[' && !parar) {
  cadena[cadena.Length()] = caracter;
  cadena.SetLength(cadena.Length() + 1);
  fread(&caracter, sizeof(caracter), 1, fichero);
 }
 if (caracter == ' ') {
  while (caracter == ' ')
   fread(&caracter, sizeof(caracter), 1, fichero);
  if (caracter == '=') {
   // Si leemos igual ha acabado el nombre
   estructura[numeroProcesos].simple = true;
   estructura[numeroProcesos].nombre = cadena;
   parar = analizarProceso(fichero,raiz,true,0,caracter,eof);
  }
  else {
   // si no, es que hay parámetros en la cabecera...
   estructura[numeroProcesos].nombre = cadena;
   estructura[numeroProcesos].simple = false;
   // y se leen
   parar = leerParametros(fichero,numeroProcesos,eof);
  }
 }
 else {
  estructura[numProcesos].nombre = cadena;
  estructura[numProcesos].simple = false;
  parar = leerParametros(fichero,numProcesos, eof);
 }
 return parar;
}

//---------------------------------------------------------------------
// Método que lee el número de parámetros de un proceso en el "código principal"
// Ya ha leído el carácter '(' o '['
bool TForm1:: leerParametros(FILE* fichero, int numeroProcesos,int &eof)
{
 char caracter;
 String cadena = " ";
 int numParametros = 1;
 fread(&caracter, sizeof(caracter), 1, fichero);
 while (caracter != ')' && caracter != ']') {
   cadena[cadena.Length()] = caracter;
   cadena.SetLength(cadena.Length() + 1);
   if (caracter == ',')
     numParametros = numParametros + 1;
   fread(&caracter, sizeof(caracter), 1, fichero);
 }
 estructura[numeroProcesos].numParametros = numParametros;
```

```
 estructura[numeroProcesos].tipos = new String[numParametros];
 estructura[numeroProcesos].nombreParametro = new String[numParametros];
 tomarNombreParametros(cadena, numeroProcesos);
 // Se encuentra el carácter =, se tiene proc ... = ...
 encuentraCaracter(fichero,CaracterIgual,NoOmitirCaracter);
 // y se analiza el código del proceso
 return analizarProceso(fichero,raiz,true,0,caracter,eof);
}

//---------------------------------------------------------------------
// Método que toma el nombre de los parametros en el código del programa
void TForm1:: tomarNombreParametros(String parametros,int numeroProcesos)
{
  int indice = 1;
  int indicePalabra = 1;
  int proceso = 0;
  String cadena = " ";
  char caracter;
  while (indice < parametros.Length()) {
    caracter = parametros[indice];
    // Si aparece una coma lo leído anteriormente se guarda, es el nombre del tipo
    if (caracter == ',') {
       estructura[numeroProcesos].nombreParametro[proceso] = cadena;
       indicePalabra = 0;
       proceso = proceso + 1;
       cadena.SetLength(1);
    }
    else {
      // Si no, se añade el carácter a la cadena
      cadena[indicePalabra] = caracter;
      cadena.SetLength(indicePalabra + 1);
    }
    indice = indice + 1;
    indicePalabra = indicePalabra + 1;
  }
  // Se guarda el nombre del último tipo
  estructura[numeroProcesos].nombreParametro[proceso] = cadena;
}

//---------------------------------------------------------------------
// Método que toma el nombre de los tipos en la declaración de los mismos
void TForm1 :: leerTipos(FILE * fichero)
{
 char caracter;
 fread(&caracter, sizeof(caracter), 1, fichero);
 String cadena = " ";
 // Mientras no aparezca la señal de fin de declaración de tipos
 bool parar = false;
 while (!parar) {
   cadena[cadena.Length()] = caracter;
```

```
    if (cadena.Length() == 5) {
      // si tenemos esta marca leemos los datos del tipo
      if (cadena == "type ") {
        leerDatosTipo(fichero,numTipos);
        numTipos = numTipos + 1;
      }
      // Si aparece la señal de fin de dedclaración se acaba
      else if (cadena == "FTYPE")
        parar = true;
      //  Se mueven los caracteres de la cadena de lectura de derecha
      // a izquierda
      else
        for (int i = 1; i < cadena.Length(); i++) {
          caracter = cadena[i+1];
          cadena.Delete(i,1);
          cadena.Insert(caracter,i);
        }
    }
    else
      cadena.SetLength(cadena.Length() + 1);
    fread(&caracter, sizeof(caracter), 1, fichero);
  }
}
//---------------------------------------------------------------------
// Método que lee los tipos de los parámetros de las distintas "cabeceras"
// de los procesos
void TForm1 :: leerCabeceras(FILE * fichero)
{
  char caracter;
  fread(&caracter, sizeof(caracter), 1, fichero);
  String cadena = " ";
  bool parar = false;
  // Mientras no aparezca la señal de fin de declaración de Cabecera
  while (!parar) {
    cadena[cadena.Length()] = caracter;
    if (cadena.Length() == 5) {
      // si tenemos esta marca leemos el proceso
      if (cadena == "proc ") {
        caracter = leerDatosCabecera(fichero,numCabeceras);
        InsertaCaracter(cadena,caracter,5);
        numCabeceras = numCabeceras + 1;
      }// Si aparece la señal de fin de declaración se acaba
      else if (cadena == "FHEAD")
        parar = true;
      else
      //  Se mueven los caracteres de la cadena de lectura de derecha
      // a izquierda
        for (int i = 1; i < cadena.Length(); i++) {
          caracter = cadena[i+1];
          cadena.Delete(i,1);
```

```
          cadena.Insert(caracter,i);
        }
    }
    else
      cadena.SetLength(cadena.Length() + 1);
    fread(&caracter, sizeof(caracter), 1, fichero);
  }// while (!parar)
}

//---------------------------------------------------------------------
// Método que toma el nombre del tipo y toma los valores de éste
void TForm1 :: leerDatosTipo(FILE* fichero,int numTipos)
{
  char caracter;
  int numValores = 0;
  fread(&caracter, sizeof(caracter), 1, fichero);
  String cadena = " ";
  // Mientras no aparezca el nombre del tipo se saltan los blancos
  while (caracter == ' ')
     fread(&caracter, sizeof(caracter), 1, fichero);
  // Se lee el nombre del tipo
  while (caracter != ' ') {
    cadena[cadena.Length()] = caracter;
    cadena.SetLength(cadena.Length() + 1);
    fread(&caracter, sizeof(caracter), 1, fichero);
  }
  tipos[numTipos].nombre = cadena;
  cadena.SetLength(1);
  //Nos posicionamos al principio de la enumeración
  while (caracter != '{')
    fread(&caracter, sizeof(caracter), 1, fichero);
  // Nos quitamos el carácter
  fread(&caracter, sizeof(caracter), 1, fichero);
  tipos[numTipos].valores = new String[TamArrayValoresTipo];
  // Leemos los valores
  while (caracter != '}') {
    // Si el carácter es una coma, almacenamos la cadena que hemos ido tomando
    if (caracter == ',') {
      tipos[numTipos].valores[numValores] = cadena;
      numValores = numValores + 1;
      cadena.SetLength(1);
    }
    // Si no, se añade el carácter a la cadena
    else {
      cadena[cadena.Length()] = caracter;
      cadena.SetLength(cadena.Length() + 1);
    }
    fread(&caracter, sizeof(caracter), 1, fichero);
  }
  tipos[numTipos].valores[numValores] = cadena;
```

```
  numValores = numValores + 1;
  tipos[numTipos].numValores = numValores;
  hayTipos = true;
 }

//---------------------------------------------------------------------
// Método que lee el nombre y los parámetros de la cabecera
char TForm1 :: leerDatosCabecera(FILE* fichero,int numCabeceras)
{
  char caracter;
  int numParametros = 0;
  bool tipoDefEnc = false; //tipoDefectoEncontrado
  bool * arrayBool = new bool[TamArrayNumParamCabecera];
  fread(&caracter, sizeof(caracter), 1, fichero);
  String cadena = " ";
  // Mientras no aparezca el nombre del proceso se saltan los blancos
  while (caracter == ' ')
     fread(&caracter, sizeof(caracter), 1, fichero);
  // Se lee el nombre del proceso
  while (caracter != ' ' && caracter != '(' && caracter != '[') {
   cadena[cadena.Length()] = caracter;
   cadena.SetLength(cadena.Length() + 1);
   fread(&caracter, sizeof(caracter), 1, fichero);
  }
  cabeceras[numCabeceras].proceso = cadena;
  cabeceras[numCabeceras].numParametros = 0;
  cabeceras[numCabeceras].resto.total = 0;
  cadena.SetLength(1);
  // Si el caracter era vacío avanzamos hasta el principio
  if (caracter == ' ')
    //Nos posicionamos al principio de la cabecera
    while (caracter != '(' && caracter != '[' && caracter != ';')
      fread(&caracter, sizeof(caracter), 1, fichero);
  // Si tiene parámetros o rangos para la cabecera aparecen
  // antes de la señal de parámetros declarados al margen
  // de la cabecera
  if (caracter != ';') {
    //Leemos la marca de comienzo de la cabecera
    fread(&caracter, sizeof(caracter), 1, fichero);
    cabeceras[numCabeceras].parametros = new String[TamArrayNumParamCabecera];
    // Leemos los tipos de los parametros
    while (caracter != ')' && caracter != ']') {
      // Si el carácter es coma, lo leído se guarda --> es un parámetro
      if (caracter == ',') {
        cabeceras[numCabeceras].parametros[numParametros] = cadena;
        arrayBool[numParametros] = esPorDefecto(cadena);
        // Si es un tipo por defecto, guardamos para la lectura posterior
        // del rango
        if(!tipoDefEnc)
          tipoDefEnc = arrayBool[numParametros];
```

```
        numParametros = numParametros + 1;
        cadena.SetLength(1);
       }
       // Si no, se añade el carácter a la cadena
       else {
        cadena[cadena.Length()] = caracter;
        cadena.SetLength(cadena.Length() + 1);
       }
       fread(&caracter, sizeof(caracter), 1, fichero);
     }// fin while
     // Se guarda el último parámetro
     cabeceras[numCabeceras].parametros[numParametros] = cadena;
     arrayBool[numParametros] = esPorDefecto(cadena);
     if (esPorDefecto(cadena))
       tipoDefEnc = true;
     numParametros = numParametros + 1;
     cabeceras[numCabeceras].numParametros = numParametros;
     hayCabeceras = true;
     // Si hay tipos por defecto que necesitan rangos se leen
     if (tipoDefEnc)
       leerRangos(fichero, numCabeceras, numParametros,arrayBool);
     caracter = QuitarBlancos(fichero);
   }
   // Si el carácter es ';', es que se leen o escriben cosas dentro del proceso
   // y se toman sus nombres de variables, tipos y rangos
   if (caracter == ';') {
     cabeceras[numCabeceras].hayResto = true;
     caracter = tomarResto(fichero,numCabeceras);
   }
   else
     cabeceras[numCabeceras].hayResto = false;
   delete [] arrayBool;
   return caracter;
}


//---------------------------------------------------------------------
// Método que lee los rangos para los parámetros no acotados, i.e., int y char
// de la cabecera de un proceso
void TForm1 :: leerRangos(FILE* fichero,int numCabeceras, int numParametros, bool*
                                    arrayBool)
{
   String cadena = " ";
   char caracter;
   int numParadas = 0;
   // Vemos cuántos rangos hemos de leer
   for( int i = 0; i < numParametros; i++)
     if (arrayBool[i])
        numParadas = numParadas + 1;
```

```
    cabeceras[numCabeceras].rangos = new String[numParadas];
    // Leemos del flujo hasta que encontramos la marca '('
    fread(&caracter, sizeof(caracter), 1, fichero);
    while (caracter != '(')
      fread(&caracter, sizeof(caracter), 1, fichero);
    // Nos quitamos de en medio el delimitador '('
    fread(&caracter, sizeof(caracter), 1, fichero);
    // Ya estamos con los valores de los rangos
    int paradasHechas = 0;
    while (paradasHechas < numParadas) {
      // Leemos los blancos
      while (caracter == ' ')
        fread(&caracter, sizeof(caracter), 1, fichero);
      // Leemos el valor del primer extremo
      while (caracter != ' ' && caracter != '-') {
        cadena[cadena.Length()] = caracter;
        cadena.SetLength(cadena.Length() + 1);
        fread(&caracter, sizeof(caracter), 1, fichero);
      }

      cabeceras[numCabeceras].rangos[paradasHechas] = cadena;

      // Leemos los blancos hasta la marca
      while (caracter != '-')
        fread(&caracter, sizeof(caracter), 1, fichero);
      // Leemos la marca '-'
      fread(&caracter, sizeof(caracter), 1, fichero);
      // Leemos los blancos hasta la marca
      while (caracter == ' ')
        fread(&caracter, sizeof(caracter), 1, fichero);
      // Leemos el valor del segundo extremo
      int longitud = cabeceras[numCabeceras].rangos[paradasHechas].Length();
      String marca = "|";
      cabeceras[numCabeceras].rangos[paradasHechas] =
cabeceras[numCabeceras].rangos[paradasHechas].Insert(marca,longitud);
      // Ajustamos la longitud de la cadena a 0
      cadena.SetLength(1);
      // Ya no hay comas pero sí marcas, queda un último rango
      if (numParadas - 1 == paradasHechas) {
        while (caracter != ' ' && caracter != ')') {
          cadena[cadena.Length()] = caracter;
          cadena.SetLength(cadena.Length() + 1);
          fread(&caracter, sizeof(caracter), 1, fichero);
        }
        if (caracter == ' ')
          encuentraCaracter(fichero,')',NoOmitirCaracter);
      } // fin if (numParadas - 1 == paradasHechas)
      else
      // queda más de un rango por leer
        while (caracter != ' ' && caracter != ')' && caracter != ',') {
```

```
          cadena[cadena.Length()] = caracter;
          cadena.SetLength(cadena.Length() + 1);
          fread(&caracter, sizeof(caracter), 1, fichero);
        }
      cabeceras[numCabeceras].rangos[paradasHechas] =
cabeceras[numCabeceras].rangos[paradasHechas].Insert(cadena,longitud + 1);
      // Leer hasta la coma
      if (numParadas - 1 != paradasHechas) {
        while (caracter != ',')
          fread(&caracter, sizeof(caracter), 1, fichero);
        // Quitamos la coma
        fread(&caracter, sizeof(caracter), 1, fichero);
      }
      cadena.SetLength(1);
      paradasHechas = paradasHechas + 1;
    }// fin   while (paradasHechas < numParadas)
}

//---------------------------------------------------------
// Método que comprueba que los tipos declarados en la cabecera
// o fuera de ella existen, ya sean tipos por defecto o creados
// en la declaración de tipos
void TForm1 :: comprobarTipos()
{
  int indiceT = 0;
  int indiceC = 0;
  int indiceP = 0;
  String tipo, cabecera;
  bool parar = false;
  bool coincidencia = false;
  int marca;

  while (!parar && indiceC < numCabeceras) {
    marca = cabeceras[indiceC].numParametros;
    while (!parar && indiceP < cabeceras[indiceC].numParametros) {
      coincidencia = false;
      indiceT = 0;
      while (indiceT < numTipos && !coincidencia) {
        // Si el tipo de esa cabecera existe se finaliza la comprobación
        // para este parámetro.
        tipo = tipos[indiceT].nombre;
        cabecera = cabeceras[indiceC].parametros[indiceP];
        if (tipos[indiceT].nombre.AnsiCompare(cabeceras[indiceC].parametros[indiceP])
            ==
            0)
          coincidencia = true;
        else
          indiceT = indiceT + 1;
      }
      // Si no hay coincidencia, es que el tipo de ese parámetro no existe
```

```
      if (!coincidencia)
        parar = true;
      indiceP = indiceP + 1;
    }
    indiceP = 0;
    indiceC = indiceC + 1;
  }
  // Si na ha habido coincidencia se abandona el programa
  if (parar)
    salidaProgramaError(17,cabeceras[indiceC].parametros[indiceP],
                                cabeceras[indiceC].proceso);

  // Se miran ahora los parámetros fuera de la cabecera de un proceso
  indiceT = 0;
  indiceC = 0;
  indiceP = 0;
  parar = false;
  coincidencia = false;

  while (!parar && indiceC < numCabeceras) {
    marca = cabeceras[indiceC].numParametros;
    // Si no hay parámetros adicionales no se hace la búsqueda
    while (!parar && indiceP < cabeceras[indiceC].resto.total) {
      coincidencia = false;
      indiceT = 0;
      while (indiceT < numTipos && !coincidencia) {
        // Si el tipo declarado fuera de la cabecera existe se
        // finaliza la comprobación para este parámetro.
        tipo = tipos[indiceT].nombre;
        cabecera = cabeceras[indiceC].resto.tipoVar[indiceP];
        if
(tipos[indiceT].nombre.AnsiCompare(cabeceras[indiceC].resto.tipoVar[indiceP]) == 0)
          coincidencia = true;
        else
          indiceT = indiceT + 1;
      }
      // Si no hay coincidencia, es que el tipo de ese parámetro no existe
      if (!coincidencia)
        parar = true;
      indiceP = indiceP + 1;
    }
    indiceP = 0;
    indiceC = indiceC + 1;
  }
  // Si na ha habido coincidencia se abandona el programa
  if (parar)

salidaProgramaError(18,cabeceras[indiceC].resto.tipoVar[indiceP],cabeceras[indiceC].p
roceso);
}
```

```
//---------------------------------------------------------------------
// Método que comprueba si el nombre de un tipo es de los creados por
// defecto y necesitan rangos de valores
bool TForm1 :: esPorDefecto(String nombreTipo)
{
  int indice = 0;
  bool parar = false;
  while (indice < TPDExigenRangos && !parar) {
     if (nombreTipo.AnsiCompare(tipos[indice].nombre) == 0)
       parar = true;
     else
       indice = indice + 1;
  }
  return parar;
}

//---------------------------------------------------------------------------
// Método que lee los blancos de una cadena de caracteres hasta encontrar un
// carácter distinto de blanco
char TForm1 :: QuitarBlancos(FILE* fichero)
{
   char caracterLeido;
   fread(&caracterLeido, sizeof(caracterLeido), 1, fichero);
   while (caracterLeido == ' ')
     fread(&caracterLeido, sizeof(caracterLeido), 1, fichero);
   return caracterLeido;
}
//---------------------------------------------------------------------------
// Método que dado un flujo de un fichero y un carácter marca, lee del flujo
// hasta encontrarlo
char TForm1 :: encuentraCaracter(FILE* fichero, char caracterBuscado, bool
omitirCaracter)
{
   char caracterLeido;
   fread(&caracterLeido, sizeof(caracterLeido), 1, fichero);
   while (caracterLeido != caracterBuscado)
     fread(&caracterLeido, sizeof(caracterLeido), 1, fichero);
   if (omitirCaracter)
     fread(&caracterLeido, sizeof(caracterLeido), 1, fichero);
   return caracterLeido;
}
//---------------------------------------------------------------------------
// Método que inserta un carácter en un cadena actualizando su longitud
// Si se alcanza la longitud máxima dada como parámetro se desplazan todas las
// letras de la cadena a la izquierda y se inserta como última la pasada como
// parámetro.
// SI LIMITE DISTINTO DE 0, EL TAMAÑO MAXIMO DE LA CADENA ES
LIMITE - 1
void TForm1 :: InsertaCaracter(String& cadena, char caracterParam, int limite)
```

```
{
  char caracter;
  int num = cadena.Length();
  if (limite == 0 || cadena.Length() < limite) {
    cadena[cadena.Length()] = caracterParam;
    cadena.SetLength(cadena.Length() + 1);
  }
  else {
     for (int i = 1; i < cadena.Length(); i++) {
        caracter = cadena[i+1];
        cadena.Delete(i,1);
        cadena.Insert(caracter,i);
     }
     cadena.Delete(cadena.Length()-1,1);
     cadena.Insert(caracterParam,cadena.Length());
  }
}

//---------------------------------------------------------------
// Método que analiza el código de un proceso y lo almacena
bool TForm1 :: analizarProceso(FILE* fichero,seleccion nodo, bool nodoPadre, int
indSel, char &ultCaracter, int &eof)
{
  char caracter;
  int indiceIf = 0;
  int posTermino = 0;
  int indiceTermino = 0;  // Número de partes/miembros que componen el proceso
  bool primera = true;
  bool vieneDeIf = false;

  //---INICIALIZACION DE LA INFORMACION DE LA ESTRUCTURA-------------
//
  // ENTRAMOS SABIENDO EN QUE SELECCION ESTAMOS (INDICE)
  // Asignamos 10 fragmentos dentro de cada selección
  nodo.textoPlano[indSel] = new Linea[TamArrayTerminos];
  //Array que tiene el orden para cada fragmento de los elementos que lo componen
  nodo.ordenacion[indSel] = new Orden[TamArrayTerminos];
  // Suponemos que el fragmento tiene no más de 10 partes
  nodo.ordenacion[indSel][indiceTermino].orden = new int[TamArrayOrdenacion];
  // Array que almacena los parametros con su información para cada fragmento
  nodo.parametros[indSel] = new LineaDeLinea[TamArrayTerminos];
  //Creas espacio para los ifs de cada fragmento
  nodo.ifs[indSel] = new seleccion*[TamArrayTerminos];

  //---------------------------------------------------------------//

  String cadenaCod = " ";
  caracter = QuitarBlancosCarro(fichero);
  //Estamos leyendo cosas del proceso
  bool continuar = true;
```

```
bool fallo = false;

while (continuar && !fallo) {
  if (caracter == '\\')
    continuar = false; //--->EOF
  if (vieneDeIf) {
     if (caracter == 'p' && nodoPadre) {
        // Vemos si se trata de un nuevo proceso
       vaciarCadena(cadenaCod);
       InsertaCaracter(cadenaCod,caracter);
       if (esProceso(fichero,cadenaCod)) {
          continuar = false;
          nodo.ordenacion[indSel][indiceTermino].tamanio = posTermino;
          indiceTermino = indiceTermino + 1;
       }
       else
         salidaProgramaError(19);
     }
     else if (caracter == ']')
     //Se ha acabado la estructura de selección y salimos, volvemos al padre
        continuar = false;
     else if (caracter != '.' && caracter != ' ')
     // hay algo como [if ...]salida falta el punto
       salidaProgramaError(14);
     else if (caracter == '.') {
        fread(&caracter, sizeof(caracter), 1, fichero);
        if (caracter != '[') {
          vaciarCadena(cadenaCod);
         primera = true;
        }
     }
    else if(caracter == ' ') {
      vieneDeIf = false;
      indiceTermino = indiceTermino + 1;
    }
  } // fin VienIF

  //---------------------//
  if (continuar) {
    if (caracter == '[')  // Se trata de un if {
      // Se toma el código de la esrtcutura de selección
      //tenemos un if como primer elemento del fragmento de linea
      caracter = tratarIf(fichero,nodo,indSel,indiceTermino,indiceIf,eof);
              if (caracter == '\\')
        continuar = false;  //---> EOF
      else if (caracter == '\n') // Si es retorno de carro
        caracter = QuitarBlancosCarro(fichero);
      indiceIf = indiceIf + 1;
      nodo.ordenacion[indSel][indiceTermino].orden[posTermino] = 0;
      posTermino = posTermino + 1;
```

```
        if (caracter == '\\') {  //---> EOF
          vieneDeIf = false;
          nodo.ordenacion[indSel][indiceTermino].tamanio = posTermino;
          indiceTermino = indiceTermino + 1;
        }
        else
          vieneDeIf = true;
        proceso_CCS_Basico = false;
      }
      else { // No empieza por un if
        vieneDeIf = false;
        InsertaCaracter(cadenaCod,caracter);
        // Tomamos el código de la elección
        caracter =
tomarSecuencia(fichero,cadenaCod,primera,nodo,indSel,indiceTermino,indiceIf,posTer
mino);
        // Al salir actualizamos las variables para tratar un nuevo fragmento
        indiceIf = 0;
        posTermino = 0;
        indiceTermino = indiceTermino + 1;
        if (caracter == '+')  { // --> Va a haber más términos en el proceso
          continuar = true;
          caracter = ' ';
          primera = false;
          // Se inicializa la estructura para el siguiente fragmento
          //Suponemos no más de 10 parámetros por fragmento
          nodo.parametros[indSel][indiceTermino].termino = new Linea[TamArrayParam]
          nodo.ordenacion[indSel][indiceTermino].orden = new int[TamArrayOrdenacion]
          //Creas espacio para los ifs de cada fragmento
          nodo.ifs[indSel][indiceTermino] = new seleccion[TamArrayIfs];
        }
        else if (caracter ==']') //Fin de una selección
          continuar = false;
        else if (caracter == 'p') { //---> Puede venir otro proceso distinto
          vaciarCadena(cadenaCod);
          InsertaCaracter(cadenaCod,caracter);
          if (esProceso(fichero,cadenaCod))
            continuar = false;
          else
            salidaProgramaError(3);
        }
        else if (caracter == 'e') //---> puede que venga un else/elsif
           continuar = false;
        else if (caracter == '\n') {
        // ---> Tenemos una nueva línea del fichero y la analizamos
          caracter = QuitarBlancosCarro(fichero);
          // Tenemos otra elección
          if (caracter == '+') {
            // Se inicializa la estructura para el siguiente fragmento
```

```
            nodo.parametros[indSel][indiceTermino].termino = new
Linea[TamArrayParam]; //Suponemos no más de 10 parámetros por fragmento
            nodo.ordenacion[indSel][indiceTermino].orden = new
int[TamArrayOrdenacion];
            //Creas espacio para los ifs de cada fragmento
            nodo.ifs[indSel][indiceTermino] = new seleccion[TamArrayIfs];
            primera = false;
           }
           else if (caracter == 'p' && nodoPadre) { // Vemos si se trata de un nuevo proceso
            vaciarCadena(cadenaCod);
            InsertaCaracter(cadenaCod,caracter);
            // Si es el proceso siguiente paramos si no es un fallo
            if (esProceso(fichero,cadenaCod))
              continuar = false;
            else
              salidaProgramaError(19);
           }
           else if (caracter == ']')  // SE CIERRA UN IF
            continuar = false;       // Salimos y se encarga tratarIf de volver al método
           else if (caracter == 'e')  // PUEDE QUE TENGAMOS UN ELSIF/ELSE
            continuar = false;       // Salimos y que se encargue tratarIf al volver al método
           else if (caracter == '\\') // TENEMOS FIN DE FICHERO
            continuar = false;
           else
            salidaProgramaError(3);
          }
          else
            salidaProgramaError(3);
        }// else '['
        primera = false;
      }
    }
    // Una vez que se tiene todo el proceso, se escriben los fragmentos
    // del mismo en el flujo de salida
   nodo.numFragSelec[indSel] = indiceTermino;
   // Si era el nivel superior, se genera el código
   if (nodoPadre)
     escribirProceso(nodo);
   if (!continuar && caracter == '\\')
     eof = 0;
   ultCaracter = caracter;
   return !continuar;
}
//---------------------------------------------------------------------------
// Método encargado de leer el código de una estructura de selección
char TForm1 :: tratarIf(FILE* fichero,seleccion nodo,int indSel,int indiceTermino,int
indiceIf,int &eof)
{
  // Si hemos entrado es porque ya hemos leído el caracter '['
  int numSelecciones = 0;
```

```
  // Si es el primer if de este fragmento para el elemento de selección indSel,
  // entonces creamos el espacio para todos los ifs de este fragmento
  if (indiceIf == 0)
    nodo.ifs[indSel][indiceTermino] = new seleccion[TamArrayIfs];
  // Inicialización del nodo correspondiente
  nodo.ifs[indSel][indiceTermino][indiceIf].ifs = new
                          seleccion**[TamArraySelecciones];
  nodo.ifs[indSel][indiceTermino][indiceIf].textoPlano = new
                          Linea*[TamArraySelecciones];
  nodo.ifs[indSel][indiceTermino][indiceIf].numFragSelec = new
                          int[TamArrayTerminos];
  nodo.ifs[indSel][indiceTermino][indiceIf].ordenacion = new
                          Orden*[TamArrayOrdenacion];
  nodo.ifs[indSel][indiceTermino][indiceIf].parametros = new
                          LineaDeLinea*[TamArraySelecciones];
nodo.ifs[indSel][indiceTermino][indiceIf].condiciones = new
                          condicion*[TamArraySelecciones];
  char caracter;
  char ultCaracter;
  bool parar = false;
  String cadena = " ";
  caracter = QuitarBlancosCarro(fichero);
  if (caracter == '\\')
    salidaProgramaError(31); //---> If vacío EOF
  // Leemos el principio de la estructura hasta que aparezca un blanco
  while (caracter != ' ' && caracter != '\n') {
     InsertaCaracter(cadena,caracter);
     fread(&caracter, sizeof(caracter), 1, fichero);
  }
  if (cadena.AnsiCompare("if") != 0)
    salidaProgramaError(10);  // No aparece la marca if y salimos
  encuentraCaracter(fichero,'(');

  // Reservamos espacio para los operadores para la condición numSelecciones
  nodo.ifs[indSel][indiceTermino][indiceIf].condiciones[numSelecciones] = new
condicion[32];
  // leemos la condición del if
  leerCondicion(fichero,nodo.ifs[indSel][indiceTermino][indiceIf],numSelecciones,1,
                        false);

  //Una vez leída la condicion hay que tratar el código

analizarProceso(fichero,nodo.ifs[indSel][indiceTermino][indiceIf],false,numSelecciones
                         ,ultCaracter,eof);
  numSelecciones = numSelecciones + 1;
  while (!parar) {
     if (eof == 0)
       parar = true;
     // Si el if ya indicó que no había más selecciones salimos del bucle
     if(ultCaracter == ']')
```

```
        parar = true;
      else if (ultCaracter != 'e')
        salidaProgramaError(15); // Hemos encontrado un caracter distinto de fin
       // de estructura de selección y no empieza por e --> no es elsif/else
      else { // Es una 'e'. Ahora vemos si es un elsif o un else
        if (esSeleccion(fichero)) { //Es un else ---> no hay condición
          analizarProceso(fichero, nodo.ifs[indSel][indiceTermino][indiceIf], false,
                          numSelecciones, ultCaracter, eof);
          nodo.ifs[indSel][indiceTermino][indiceIf].condiciones[numSelecciones] =
                                                                  NULL;
        }
        else {  // Es un elsif ---> Leemos la condición
          // Reservamos espacio para los operadores para la condición numSelecciones
          nodo.ifs[indSel][indiceTermino][indiceIf].condiciones[numSelecciones] = new
condicion[32];
          //Leemos la condición
leerCondicion(fichero,nodo.ifs[indSel][indiceTermino][indiceIf],numSelecciones,1,true
);

analizarProceso(fichero,nodo.ifs[indSel][indiceTermino][indiceIf],false,numSelecciones
                  ,ultCaracter,eof);
        }
        numSelecciones = numSelecciones + 1;
      }
      // Almacenamos el total de selecciones que tenía esa estructura de selección
      nodo.ifs[indSel][indiceTermino][indiceIf].numeroSelecciones = numSelecciones;
    }
    // Se devuelve el caracter que viene a continuación.
    // Si se encontró el fin de fichero lo notificamos devolviendo '\\'
    if (eof == 0)
      return '\\';
    if (fread(&caracter,sizeof(caracter),1,fichero) == 0)
      return '\\';  // Si es el fin de fichero lo notificamos devolviendo '\\'
    else
      return caracter;
}


//---------------------------------------------------------------------
// Método que devuelve true si en una estructura de selección se ha leído else
// false si se trata de elsif
bool TForm1 :: esSeleccion(FILE* fichero)
{
  //Se supone leída la letra 'e' de else o elsif
  int indice = 0;
  bool parar = false;
  char caracter;
  String cadena = " ";
  while(indice < 3) {  // Se van a leer tres caracteres para ver si viene lse
    if (fread(&caracter,sizeof(caracter),1,fichero) == 0)
```

```
      salidaProgramaError(16); //Falta código en la marca de selección
    else
      InsertaCaracter(cadena,caracter);
    indice = indice + 1;
  }
  if (cadena.AnsiCompare("lse") == 0)
    return true;
  else if (cadena.AnsiCompare("lsi") == 0) {
    if (fread(&caracter,sizeof(caracter),1,fichero) == 0)
      salidaProgramaError(16); //Falta código en la marca de selección
    if (caracter == 'f')    // Se trata de elsif
      return false;
    else
      salidaProgramaError(17); // No es ni else ni elsif. Fallo en la sintaxis
  }
  else
    salidaProgramaError(17); // No es ni else ni elsif. Fallo en la sintaxis
}


//---------------------------------------------------------------------
// Método que lee un fragmento (porción de código) de un proceso, el cual va
// precedido de un '+' si no es el primer "fragmento" del proceso
// IndIf: Necesario pq el primer fragmento del término puede haber sido una estructura
// de selección. Por esa misma razón se pasa posTermino que vale 1 si se dió la
// selección citada anteriormente

char TForm1 :: tomarSecuencia(FILE* fichero, String cadena, bool primera, seleccion
nodo,int indSel,int indiceTermino,int indIf,int posTermino)
{
   bool comienzoSenial = true;
   bool inicioBucle = true;
   if (!primera)
     vaciarCadena(cadena);
   int indiceTexto = 0;
   int indiceParam = 0;
   int indiceIf = indIf;
   int indPos = posTermino;
   char caracter;
   // Se crea espacio para el texto "plano" del fragmento
   nodo.textoPlano[indSel][indiceTermino].partes = new String[TamArrayTP];
   // Se crea espacio para los parámetros que se lean
   nodo.parametros[indSel][indiceTermino].termino = new Linea[TamArrayParam];
   //Suponemos no más de 10 parámetros por fragmento
   // Quitamos caracteres blancos y saltos de línea si es necesario
   caracter = QuitarBlancosCarro(fichero);
   bool parar = false;
   while(caracter != ' ' && caracter != '+' && caracter != '\n' && caracter != '\\' &&
            !parar) {
      if (caracter == '[' && comienzoSenial && inicioBucle)
```

```
        salidaProgramaError(18);
// Al comienzo de la línea hay un if y ahora viene otro sin un punto que los separe
        // Tenemos [if (...)][(...)] cuando debería ser [if (...)].[(...)]
      else if (caracter == '[' && comienzoSenial) { //Viene un if
        caracter = tratarIf(fichero,nodo,indSel,indiceTermino,indiceIf,eof);
        // Indicamos que en el fragmento ahora viene una estructura de selección
        nodo.ordenacion[indSel][indiceTermino].orden[indPos] = 0;
        // Se incrementa la posición del índice de posición de elemento dentro del
        // fragmento
        indPos = indPos + 1;
        // Se incrementa el índice de selecciones dentro del fragmento que estamos
        // tratando
        indiceIf = indiceIf + 1;
      }
      else if (caracter == '(' && !comienzoSenial) { //Viene un parámetro
        nodo.textoPlano[indSel][indiceTermino].partes[indiceTexto] = cadena;
        // Se incrementa el índice de texto "plano", i.e., el que no es ni estructura
        // de selección ni parámetrico
        indiceTexto = indiceTexto + 1;
        vaciarCadena(cadena);
        // Se añade a la estructura que almacena el orden de las apariciones
        // de elementos el texto "plano" que habíamos leído hasta ahora
        nodo.ordenacion[indSel][indiceTermino].orden[indPos] = 2;
        // Se incrementa la posición del índice de posición de elemento dentro del
fragmento
        indPos = indPos + 1;
        // Se leen los parámetros
        leerParametroSecuencia(fichero,nodo,indSel,indiceTermino,indiceParam);
        // Una vez leídos, se incrementa el índice de parámetros...
        indiceParam = indiceParam + 1;
        // Y se añade a la estructura que almacena el orden de las apariciones
        // de elementos que ha aparecido un/os parámetro/s
        nodo.ordenacion[indSel][indiceTermino].orden[indPos] = 1;
        // y se incrementa la posición del índice de posición de elemento dentro del
        // fragmento
        indPos = indPos + 1;
        hacerComprobacion(indiceParam,numProcesos);
        // Si hay fin de fichero --> salir
        if (fread(&caracter,sizeof(caracter),1,fichero)==0)
          caracter = '\\';
      }
      else if (caracter == '.') {
        // Otra señal/if
        // Si la cadena no era vacía se guarda la señal como texto plano, señal simple
        if (cadena.AnsiCompare(" ") != 0) {
          nodo.textoPlano[indSel][indiceTermino].partes[indiceTexto] = cadena;
          // Se incrementa el índice de texto "plano", i.e., el que no es ni estructura
          // de selección ni parámetrico
          indiceTexto = indiceTexto + 1;
          vaciarCadena(cadena);
```

```
          // Se añade a la estructura que almacena el orden de las apariciones
          // de elementos el texto "plano" que habíamos leído hasta ahora
          nodo.ordenacion[indSel][indiceTermino].orden[indPos] = 2;
          // Se incrementa la posición del índice de posición de elemento dentro del
fragmento
          indPos = indPos + 1;
         }
        comienzoSenial = true;
        if (fread(&caracter,sizeof(caracter),1,fichero) == 0)
          caracter = '\\';
      }
      else if (caracter == ']') { //fin de un if
        parar = true;
      }
      else {
        // Se añade el carácter leído a la cadena
        InsertaCaracter(cadena,caracter);
        comienzoSenial = false;
        if (fread(&caracter,sizeof(caracter),1,fichero) == 0)
          caracter = '\\';
      }
      inicioBucle = false;
    }
    if (cadena.AnsiCompare(" ") != 0) {
    //No es vacía y tenemos otra señal simple que se guarda
      nodo.textoPlano[indSel][indiceTermino].partes[indiceTexto] = cadena;
      indiceTexto = indiceTexto + 1;
      nodo.ordenacion[indSel][indiceTermino].orden[indPos] = 2;
      indPos = indPos + 1;
    }
    // Guardamos los datos característicos del fragmento
    nodo.textoPlano[indSel][indiceTermino].numeroPartes = indiceTexto;
    nodo.ordenacion[indSel][indiceTermino].tamanio = indPos;
    nodo.parametros[indSel][indiceTermino].numeroPartes = indiceParam;
    if (caracter == ' ')
      caracter = QuitarBlancosCarro(fichero);
    if (parar)
      return ']';
    else
      return caracter;
}

//---------------------------------------------------------------------
// Método que lee los parámetros que aparecen en el código de un proceso
// Por ejemplo, dado in(x,tiempo,altura) toma x,tiempo y altura
void TForm1 :: leerParametroSecuencia(FILE* fichero,seleccion nodo, int indSel,int
indiceTermino,int indiceParam)
{
  char caracter;
  String cadena = " ";
```

```
  int numParametros = 0;
  proceso_CCS_Basico = false;
  //Suponemos no más de 10 parámetros por fragmento
  nodo.parametros[indSel][indiceTermino].termino[indiceParam].partes =
                                                    new String[TamArrayParamBis];

  fread(&caracter, sizeof(caracter), 1, fichero);
  //  while (caracter != ')' && caracter != ']')
  while (caracter != ')') {
    // Si aparece coma se almacena el parámetro
    if (caracter == ',') {
nodo.parametros[indSel][indiceTermino].termino[indiceParam].partes[numParametros]
=
cadena;
      numParametros = numParametros + 1;
      vaciarCadena(cadena);
    }
    else
    // Se añade el carácter a la cadena
      InsertaCaracter(cadena,caracter);
    fread(&caracter, sizeof(caracter), 1, fichero);
  }
  //  Guardamos los datos característicos de la lista de parámetros

nodo.parametros[indSel][indiceTermino].termino[indiceParam].partes[numParametros]
=
cadena;
  numParametros = numParametros + 1;
  nodo.parametros[indSel][indiceTermino].termino[indiceParam].numeroPartes =
                                                                    numParametros;
}
//---------------------------------------------------------------
// Método que dada una cadena, le asigna longitud 1 y almacena un blanco
void TForm1 :: vaciarCadena(String &cadena)
{
  cadena = " ";
  cadena.SetLength(1);
}
//---------------------------------------------------------------
void TForm1 :: hacerComprobacion(int indice,int numeroProcesos)
{

}
//---------------------------------------------------------------
// Método que toma el nombre de las variables, tipo y rango de aquellas variables
// que se reciban o devuelvan durante un proceso y que estén al margen de
// las de la cabecera
char TForm1 :: tomarResto(FILE* fichero, int numCabeceras)
{
   char caracter;
```

```
String cadena = " ";
int indice = 0;
// Se reserva espacio para los datos que se tomen
cabeceras[numCabeceras].resto.nombreVar =
                                        new String[TamArrayParamFueraCabecera];
cabeceras[numCabeceras].resto.tipoVar =
                                        new String[TamArrayParamFueraCabecera];
cabeceras[numCabeceras].resto.rangoVal =
                                        new String[TamArrayParamFueraCabecera];
// Se lee del flujo hasta alcanzar el carácter '('
encuentraCaracter(fichero,'(',NoOmitirCaracter);
caracter = QuitarBlancos(fichero);
InsertaCaracter(cadena,caracter);
fread(&caracter, sizeof(caracter), 1, fichero);
// mientras que no aparezca el carácter de fin de parámetros se leen
while (caracter != ')') {
  // Se lee el nombre del parámetro...
  while (caracter != ' ' && caracter != ':') {
     InsertaCaracter(cadena,caracter);
     fread(&caracter, sizeof(caracter), 1, fichero);
  }
  // y se guarda
  cabeceras[numCabeceras].resto.nombreVar[indice] = cadena;
  vaciarCadena(cadena);
  // Se quitan los espacios...
  caracter = QuitarBlancos(fichero);
  // y se lee el tipo
  while (caracter != ','&& caracter != ' ' && caracter != ')') {
     InsertaCaracter(cadena,caracter);
     fread(&caracter, sizeof(caracter), 1, fichero);
  }
  if (caracter == ' ')
    QuitarBlancos(fichero);
  cabeceras[numCabeceras].resto.tipoVar[indice] = cadena;
  indice = indice + 1;
  vaciarCadena(cadena);
  if (caracter != ')')
  fread(&caracter, sizeof(caracter), 1, fichero);
}
// Ahora se tratan los rangos
cabeceras[numCabeceras].resto.total = indice;
indice = 0;
caracter = QuitarBlancosCarro(fichero);
if (caracter == '/') { //--> Hay rangos que leer
  caracter = QuitarBlancosCarro(fichero);
  // se avanza hasta el principio de los rangos de los mismos
  if (caracter != '(')
    salidaProgramaError(13,IntToStr(posCabecera));
  else
  // Quitamos la marca '('
```

```
      caracter = QuitarBlancosCarro(fichero);
    while (caracter != ')') {
      // Se lee el primer valor
      while (caracter != ' ' && caracter != '-') {
        InsertaCaracter(cadena,caracter);
        fread(&caracter, sizeof(caracter), 1, fichero);
      }
      // Se saltan los blancos
      if (caracter == ' ')
        caracter = QuitarBlancos(fichero); // Se supone que lee '-'
      // Se almacena el valor en la estructura
      InsertaCaracter(cadena,'|');
      cabeceras[numCabeceras].resto.rangoVal[indice] = cadena;
      vaciarCadena(cadena);
      //Se avanza hasta encontrar el siguiente
      caracter = QuitarBlancos(fichero);
      while (caracter != ' ' && caracter != ',' && caracter != ')') {
        InsertaCaracter(cadena,caracter);
        fread(&caracter, sizeof(caracter), 1, fichero);
      }
      // Una vez leído el segundo valor se guarda
      int longitud = cabeceras[numCabeceras].resto.rangoVal[indice].Length();
      cabeceras[numCabeceras].resto.rangoVal[indice] =
cabeceras[numCabeceras].resto.rangoVal[indice].Insert(cadena,longitud);
      vaciarCadena(cadena);
      indice = indice + 1;
      if (caracter == ' ')
        caracter = QuitarBlancos(fichero);
      // Si se encuentra la coma es que viene otro par
      else if (caracter == ',')
        fread(&caracter, sizeof(caracter), 1, fichero);
    }
  } // fin if (caracter == '/')
  return caracter;
}

//---------------------------------------------------------------
// Método que dado un flujo de entrada va leyendo del mismo hasta encontrar
// un carácter distinto al espacio o el retorno de carro.
// Si encuentra el fin de fichero mientras lee devuelve un slash '\\'
char TForm1 :: QuitarBlancosCarro(FILE* fichero)
{
  char caracterLeido;
  fread(&caracterLeido, sizeof(caracterLeido), 1, fichero);
  bool parar = false;
  while (caracterLeido == ' ' || caracterLeido == '\n' && !parar) {
    // Si aparece el fin de fichero salimos de bucle
    if (fread(&caracterLeido, sizeof(caracterLeido), 1, fichero) == 0) {
      parar = true;
      break;
```

```
    }
   }
   if (parar)
    return '\\';
   else
    return caracterLeido;
}
//---------------------------------------------------------------------
// Método que comprueba que el flujo que empieza por 'p' es otro proceso
// que se ha de tratar en la lectura de los mismos en el "código principal"
bool TForm1 :: esProceso(FILE* fichero,String &cadena)
{
   char caracter;
   char proceso[] = "roc ";
   int tam = 4;
   int indice = 0;
   fread(&caracter, sizeof(caracter), 1, fichero);
   // Si se han leído los 4 caracteres o no coincide alguno salimos
   while (indice < tam && caracter == proceso[indice]) {
    InsertaCaracter(cadena,caracter);
    indice = indice + 1;
    if (indice != tam)
     fread(&caracter, sizeof(caracter), 1, fichero);
   }
   if (indice == tam)
    return true;
   else
    return false;
}
//-----------------------------------------------------------------------------
// Método que dado un flujo de salida y una cadena, la escribe en dicho flujo
void TForm1 :: escribirCadena(FILE* fichero, String cadena)
{
   char caracter;
   for (int i = 1; i <= cadena.Length(); i++) {
     caracter = cadena[i];
     if (caracter != '\0')
      fwrite(&caracter,sizeof(caracter),1,fichero);
   }
}
//---------------------------------------------------------------------------
// Método que escribe a CCS básico en un fichero el código de un proceso
// que ya ha sido tratado en el método analizarCodigo
void TForm1 :: escribirProceso(seleccion nodo)
{
   // Se abre el fichero destino
   if ((ficheroDestino = fopen(nombreFicheroDestino, "a+t")) == NULL)
    ShowMessage("Error al tratar de abrir el fichero destino.");
   // Si ya estaba en CCS básico, i.e., no tiene parámetros ni ifs
   if (proceso_CCS_Basico && encontrarCabecera(numProcesos,false) == -1)
```

```
    escribirProcesoCCSBasico();
  // Si no, sí tiene parámetros
  else {
    // Se localiza la cabecera en la estructura que almacena los procesos
    posCabecera = encontrarCabecera(numProcesos);
    int anadido = 0;
    int indice = 0;
    // Si hay parámetros extra al margen de los declarados en la cabecera
    // se reserva espacio para el valor que tengan en una ejecución concreta
    if (cabeceras[posCabecera].hayResto)
      anadido = cabeceras[posCabecera].resto.total;
    // Si tiene parámetros que se leen o escriben dentro del proceso distintos
    // de los de la cabecera se añadirán a la lista. * 2 por si acaso
    int cantidad = estructura[numProcesos].numParametros + 2 * anadido +
                   numConstantes;
    ListaDoble* valorParametro = new ListaDoble[cantidad];
    // Se mira si el proceso tiene parámetros en su cabecera o no
    if (estructura[numProcesos].simple) {
      // Se escribe el nombre del proceso
      escribirCadena(ficheroDestino,"\nproc ");
      escribirCadena(ficheroDestino,estructura[numProcesos].nombre);
      escribirCadena(ficheroDestino," = ");
      // Se escribe el código del proceso, su cuerpo.
      escribirCodigo2(valorParametro,indice,nodo);
    }
    else {
      // Se escriben los parámetros de la cabecera del proceso

escribirParametros2(estructura[numProcesos].numParametros,valorParametro,indice,no
do);
    }
    delete [] valorParametro;
  }
  // Se cierra el flujo de salida...
  fclose(ficheroDestino);
  // y sel liberan los recursos
  liberarRecursos(nodo,false,true);
  primera = false;
  primeroEnEscribir = true;
  proceso_CCS_Basico = true;
}

//---------------------------------------------------------------------------
// Método que genera todas las combinaciones posibles de valores para los parámetros
// de la cabecera y se escribe ésta
void TForm1 :: escribirParametros2(int numParametros, ListaDoble*
valorParametro,int indice,seleccion nodo) {
  // Si ya hemos escrito todos los parámetros de la cabecera del método,
  // entonces escribimos el código del proceso
  int i = 0;
```

```
  String cadenaTrue = "true";
  String cadenaFalse = "false";
  // Si se han escrito todos los parámetros entonces escribimos el código del proceso
  // INDICE VA RECORRIENDO EL NUMERO DE PARAMETROS DE LA
CABECERA DEL PROCESO
  if (indice == numParametros) {
    escribirCadena(ficheroDestino,"\n");
    // Se escribe el nombre del proceso
    escribirCadena(ficheroDestino,"proc ");
    escribirCadena(ficheroDestino,estructura[numProcesos].nombre);
    // Ahora se escriben los valores de los parámetros
    for (int k = 0; k < numParametros; k++)
      escribirCadena(ficheroDestino,valorParametro[k].valor);
    escribirCadena(ficheroDestino," = ");
    primeroEnEscribir = true;
    // Una vez escrito esto, se escribe el código del cuerpo del proceso
    escribirCodigo2(valorParametro,indice,nodo);
  }
  // Si no, se toma el parámetro que indica índice y se escribe para cada posible
  // valor del mismo
  else {
     // Hay que tomar los valores correspondientes qur puede tomar el parámetro
     // número índice
     // Se almacena su valor y características del parámetro
     valorParametro[indice].nombre =
                                    estructura[numProcesos].nombreParametro[indice];
     valorParametro[indice].tipo = cabeceras[posCabecera].parametros[indice];
     valorParametro[indice].nivel = 0;
     // Si es de tipo int o char, se lee su rango de valores y los tomamos
     if (cabeceras[posCabecera].parametros[indice].AnsiCompare("int") == 0 ||
         cabeceras[posCabecera].parametros[indice].AnsiCompare("char") == 0) {
       int pos = 0;
       pos = encuentraParametro(posCabecera,indice);
       // Tras encuentraParametro, ya sé cuál es el valor del índice para
       // indexar el array de rangos de valores correspondiente al tipo
       // que ahora estoy tratando
       int valor1 = 0;
       int valor2 = 0;
       bool esInt;
       char car;
       // Se toman los valores y se generan
       tomarValoresRango(posCabecera,pos,indice,valor1,valor2,esInt);
       for (int j = valor1; j <= valor2; j++) {
         if (esInt)
           valorParametro[indice].valor = IntToStr(j);
         else {
           car = (char) j;
           valorParametro[indice].valor = car;
         }
         // y se llama recursivamente
```

```
          escribirParametros2(numParametros,valorParametro,indice + 1,nodo);
        }
      }
      else { // Si no es de estos tipos
      // Si es bool se escribe true y false
        if (cabeceras[posCabecera].parametros[indice].AnsiCompare("bool") == 0) {
          for (int j = 0; j < 2; j++) {
            if (j == 0)
              valorParametro[indice].valor = cadenaTrue;
            else
              valorParametro[indice].valor = cadenaFalse;
            // y se llama recursivamente
            escribirParametros2(numParametros,valorParametro,indice + 1,nodo);
          }
        }
        else { // Lo ha declarado el usuario
          // Buscamos el tipo en el array de tipos para tener sus valores
          while (i < numTipos &&
          tipos[i].nombre.AnsiCompare(cabeceras[posCabecera].parametros[indice]) != 0)
            i = i + 1;
          for (int j = 0; j < tipos[i].numValores; j++) {
            // Se escribe el valor j-ésimo de este tipo y se llama a este método
            // para que haga lo mismo con los valores de los parámetros posteriores
            valorParametro[indice].valor = tipos[i].valores[j];
            // y se llama recursivamente
            escribirParametros2(numParametros,valorParametro,indice + 1,nodo);
          }
        }
      }
    }
  }
//---------------------------------------------------------------------------
//
// Método encargado de la escritura de todas las distintas posibilidades del
// cuerpo de código leído anteriormente
// INDICE INDICA EL NUMERO DE PARAMETROS QUE TIENE EL PROCESO
EN LA CABECERA
void TForm1 :: escribirCodigo2(ListaDoble* valorParametro,
                                    int totalElemArrayVP,seleccion nodo)
{
  ListaCont * continuacion = new ListaCont;
  // Se reserva espacio para la pila de valores que nos indicarán por donde continuar
  // una vez que hayamos llegado al final de una elección en un nivel distinto
  // del principal
  continuacion->cont = new Continuacion[20];
  continuacion->tamanio = 0;
  Indice* indices = new Indice;
  // Indice indica qué número de parámetros,ifs y señales simples revisadas
  indices->indiceTP = 0;
  indices->indiceParam = 0;
```

```
  indices->indiceIf = 0;
  // Para cada elección generamos su código
  for (int i = 0; i < nodo.numFragSelec[0]; i++) {
    analizarComponentes(nodo,0,i,0,continuacion," ",
                                valorParametro,totalElemArrayVP,indices,0);
    indices->indiceTP = 0;
    indices->indiceParam = 0;
    indices->indiceIf = 0;
  }
  delete [] continuacion->cont;
  delete continuacion;
  delete indices;
}
//---------------------------------------------------------------------------
// Método que genera todas las posibles ramas de una estructura de selección
// Continuación no se ve alterado aquí.
bool TForm1 :: generarIfs(seleccion nodo, int seleccionEscogida, int indiceTermino,
                                int indiceIf, ListaDoble* valorParametro,
                                int totalElemArrayVP, ListaCont* continuacion,
                               String cadResultado, int nivel)
{
  int totalIfs =
nodo.ifs[seleccionEscogida][indiceTermino][indiceIf].numeroSelecciones;
  bool algunaCondicionCierta = false;
  for (int i = 0; i < totalIfs; i++)
    // Si no es la última hay evaluar las condiciones. Su condición no vale NULL
    // Si sí lo es, se entra siempre que no se haya entrado en otra condición
    if (i == totalIfs - 1 && !algunaCondicionCierta &&
        nodo.ifs[seleccionEscogida][indiceTermino][indiceIf].condiciones[i] == NULL)
    {
      recorrerIf(nodo.ifs[seleccionEscogida][indiceTermino][indiceIf],i,valorParametro,
                totalElemArrayVP,continuacion,cadResultado,nivel);
      algunaCondicionCierta = true;
    }

    // Si la condidión no es nula se entra en ella y se evalúa
    else if (nodo.ifs[seleccionEscogida][indiceTermino][indiceIf].condiciones[i] !=
             NULL) {
      // Si la condición es cierta exploramos en la selección
      if (evaluarCondicion(nodo.ifs[seleccionEscogida][indiceTermino][indiceIf],
                                i,1,valorParametro,totalElemArrayVP)) {
      // Se recorre la rama
        recorrerIf(nodo.ifs[seleccionEscogida][indiceTermino][indiceIf],i,
                  valorParametro,totalElemArrayVP,continuacion,cadResultado,nivel);
        algunaCondicionCierta = true;
      }
    }
  // Si no hay ninguna condición cierta se ejecuta el resto que viene a continuación del if
  return algunaCondicionCierta;
}
```

```
//---------------------------------------------------------------------------
// Método que recorre cada rama de una estructura de selección y llama al método
// generador del código de la misma
void TForm1 :: recorrerIf(seleccion nodo,int seleccionEscogida,
                          ListaDoble* valorParametro,
                          int totalElemArrayVP,ListaCont* continuacion,
                          String cadResultado,int nivel)
{
  // Se recorren todas las elecciones
  for (int numFrag = 0; numFrag < nodo.numFragSelec[seleccionEscogida];
       numFrag++) {
    ListaCont * contAux = new ListaCont;
    contAux->cont = new Continuacion[20];
    Indice* indices = new Indice;
    // Se ponen los índices de elementos recorridos a cero
    indices->indiceTP = 0;
    indices->indiceParam = 0;
    indices->indiceIf = 0;
    contAux->tamanio = continuacion->tamanio;
    // Se copia la pila de posiciones de regreso...
    for (int i = 0; i < continuacion->tamanio; i++) {
      contAux->cont[i].nodo = continuacion->cont[i].nodo;
      contAux->cont[i].numTermino = continuacion->cont[i].numTermino;
      contAux->cont[i].numComponente = continuacion->cont[i].numComponente;
      contAux->cont[i].numSeleccion = continuacion->cont[i].numSeleccion;
      contAux->cont[i].nivel = continuacion->cont[i].nivel;
    }
    // y se llama al método generador

analizarComponentes(nodo,seleccionEscogida,numFrag,0,contAux,cadResultado,valor
Parametro,totalElemArrayVP,indices,nivel);
    // Se libera la memoria utilizada
    delete [] contAux->cont;
    delete contAux;
    delete indices;
  }

}
//---------------------------------------------------------------------------
// Método que trata cada elemento de una elección para generar el cuerpo del proceso
void TForm1 :: analizarComponentes(seleccion nodo, int seleccionEscogida,
                                   int termino, int contador,
                                   ListaCont* continuacion, String cadResultado,
                                   ListaDoble* valorParametro,
                                   int totalElemArrayVP,Indice* indices,int nivel)
{
 String cadenaParcial;
 int posicion;
 // Si se han recorrido todos los elementos de la elección
```

```
if(contador >= nodo.ordenacion[seleccionEscogida][termino].tamanio) {
  // Si no quedan más partes de código de diferentes niveles se escribe la cadena
  // resultado generada
  if (continuacion->tamanio == 0) {
  // No hay que recorrer trozo alguno de ningún padre y se escribe la cadena
    if (primeroEnEscribir)
      primeroEnEscribir = false;
    else
      escribirCadena(ficheroDestino,"\n + ");
    escribirCadena(ficheroDestino,cadResultado);
  }
  else {
     // Hay que recorrer lo que queda de algún padre
     continuacion->tamanio = continuacion->tamanio - 1;
     // Se decrementa una posición de la pila
     posicion = continuacion->tamanio;
     int indIf = indices->indiceIf;
     int indParam = indices->indiceParam;
     int indTP = indices->indiceTP;
     // Se actualiza el valor de los índices para el fragmento de código a donde
     // vamos a ir a parar
     actualizarIndices(indices,continuacion->cont[posicion].nodo,
                               continuacion->cont[posicion].numSeleccion,
                               continuacion->cont[posicion].numTermino,
                               continuacion->cont[posicion].numComponente);
     // Se trata el código de tal fragmento saltando al mismo
     analizarComponentes(continuacion->cont[posicion].nodo,
                                    continuacion->cont[posicion].numSeleccion,
                                    continuacion->cont[posicion].numTermino,
                                    continuacion->cont[posicion].numComponente,
                                    continuacion,cadResultado, valorParametro,
                                    totalElemArrayVP,indices,
                                    continuacion->cont[posicion].nivel);
     // Se restaura el valor de los índices y el tamaño de la pila
     indices->indiceIf = indIf;
     indices->indiceParam = indParam;
     indices->indiceTP = indTP;
     continuacion->tamanio = posicion + 1;
  }
}
// Si no, es que quedan más elementos por tratar dentro de la elección
else {
   if (nodo.ordenacion[seleccionEscogida][termino].orden[contador] == 0) {
     // Tenemos un if. nivel + 1
     int posAnteriorComponente = continuacion->cont[continuación
                                                    ->tamanio].numComponente;
     // Se almacena en la pila, la posición de regreso, que será el siguiente elemento
     // a tratar dentro de esta elección
     continuacion->cont[continuacion->tamanio].numTermino = termino;
     continuacion->cont[continuacion->tamanio].numComponente = contador + 1;
```

```
      //La siguiente a revisar
      continuacion->cont[continuacion->tamanio].numSeleccion = seleccionEscogida;
      continuacion->cont[continuacion->tamanio].nodo = nodo;
      continuacion->cont[continuacion->tamanio].nivel = nivel;
      continuacion->tamanio = continuacion->tamanio + 1;
      bool algCondCierta;
      // Se genera el código de las ramas de la estructura
      algCondCierta = generarIfs(nodo,seleccionEscogida,termino,
                                 indices->indiceIf,valorParametro,
                                 totalElemArrayVP,continuacion,cadResultado,
                                 nivel + 1);
      continuacion->tamanio = continuacion->tamanio - 1;  // ACHTUNG
      indices->indiceIf = indices->indiceIf + 1;
      // Si no ha habido ninguna cierta, se genera el código del resto de elección
      if (!algCondCierta)
        analizarComponentes(nodo,seleccionEscogida,termino,
                            contador + 1,continuacion,cadResultado,
                            valorParametro,totalElemArrayVP,indices,nivel);
      // Se restauran los valores
      indices->indiceIf = indices->indiceIf - 1;
      if (nivel != 0)       //ACHTUNG
        continuacion->cont[continuacion->tamanio].numComponente =
                                                          posAnteriorComponente;
    }
    else if (nodo.ordenacion[seleccionEscogida][termino].orden[contador] == 1) {
      // Tenemos parámetros y hay que generar todos los valores de todos los parámetros
      generarParteCodigoParametrizado2(nodo,seleccionEscogida,termino,contador,
                                       valorParametro,totalElemArrayVP,
                                       cadResultado,indices,0,nivel,continuacion);
    }
    else {
      // Texto plano ---> Llamamos recursivamente concatenando al resultado parcial
dicho texto plano
      String cadAuxiliar = cadResultado;
      // Si la cadena actual no es vacía
      if (cadAuxiliar.AnsiCompare(" ") != 0)
        cadAuxiliar = concatenar(cadAuxiliar,".");
      cadenaParcial = nodo.textoPlano[seleccionEscogida][termino].partes[indices-
>indiceTP];
      cadAuxiliar = concatenar(cadAuxiliar,cadenaParcial);
      indices->indiceTP = indices->indiceTP + 1;
      analizarComponentes(nodo,seleccionEscogida,termino,contador +
1,continuacion,cadAuxiliar,valorParametro,totalElemArrayVP,indices,nivel);
      indices->indiceTP = indices->indiceTP - 1;
    }
  }// fin else. Quedan elementos por tratar en la elección actual
}

//---------------------------------------------------------------------------
// Método que genera los distintos valores para parámetros que aparecen en el
```

```
// cuerpo del proceso
void TForm1 :: generarParteCodigoParametrizado2(seleccion nodo,
                                                int seleccionEscogida,
                                                int termino,
                                                int contadorFrag,
                                                ListaDoble* valorParametro,
                                                int totalElemArrayVP,
                                                String cadResultado,
                                                Indice* indices, int paramPasados,
                                                int nivel, ListaCont* continuacion)
{
  int pos;
  int valor1 = 0;
  int valor2 = 0;
  bool esInt;
  String cadAuxiliar = cadResultado;
  // Si no se han recorridos todos, se recorren todos los parámetros que faltan
  if (paramPasados < nodo.parametros[seleccionEscogida][termino].termino[
                                                indices->indiceParam].numeroPartes)
  {
    // Tomamos el parámetro que indica numParte. Vemos si forma parte de la cabecera
    // del proceso. Si es así, el valor está en el array de cadenas, si no, hay que
    // generarlos.
    // Si no es simple y el parámetro pertenece a la lista de parámetros vistos
    String parametro = nodo.parametros[seleccionEscogida][termino].termino[indices
                                                ->indiceParam].partes[paramPasados];
    char operador;
    String literal;
    bool hayOperador = pertenece('+',parametro) || pertenece('-',parametro);
    // Si no hay operador y hay literal, se toma y se escribe en la cadena resultado
    if (!hayOperador && hayLiteral(parametro)) {
      // Se almacena el valor y características...
      valorParametro[totalElemArrayVP].valor = cogerLiteral(parametro);
      valorParametro[totalElemArrayVP].nombre = "nombreLiteral" +
                                                valorParametro[totalElemArrayVP].valor;
      valorParametro[totalElemArrayVP].tipo = "tipoLiteral" +
                                                valorParametro[totalElemArrayVP].valor;
      valorParametro[totalElemArrayVP].nivel = nivel;
      cadAuxiliar = concatenar(cadAuxiliar,valorParametro[totalElemArrayVP].valor);
      // y se llama recursivamente al mismo método

generarParteCodigoParametrizado2(nodo,seleccionEscogida,termino,contadorFrag,valo
rParametro,totalElemArrayVP,cadAuxiliar,indices,paramPasados +
1,nivel,continuacion);
    }
    else { // Si hay operador se toma, junto al parámetro y la literal
      if (hayOperador) {
        operador = cogerOperador(parametro);
        literal = cogerLiteral(parametro);
        parametro = cogerParametro(parametro);
```

```
      }
      // Si está el parámetro en el array de valores temporales, se toma su
      // valor actual
      if (estaEn(valorParametro,totalElemArrayVP,parametro,pos,nivel)) {
        if (hayOperador) {
          String cadAuxiliar2;
          if (valorParametro[pos].tipo.AnsiCompare("int") == 0 ||
                              valorParametro[pos].tipo.AnsiCompare("char") == 0) {
           // El tipo del parámetro es int o char
           int posNueva;
           if (cabeceras[posCabecera].numParametros >= pos) {
             // El parámetro es de la cabecera del proceso y no han sido
             // generado sus valores
             posNueva = encuentraParametro(posCabecera,pos);
             int posTipo =
encuentraPosParam(posCabecera,valorParametro[pos].tipo,true);
             tomarValoresRango(posCabecera,posNueva,posTipo,valor1,valor2,esInt);
           }
           else { //Parámetro generado dentro del código
             encontrarPosicion(parametro,posNueva);
             tomarValoresRango2(posCabecera,posNueva,valor1,valor2,esInt);
           }
           cadAuxiliar2 = operacion(valorParametro[pos].valor,operador,literal,
                                    valorParametro[pos].tipo,valor1,valor2);
           cadAuxiliar = concatenar(cadAuxiliar,cadAuxiliar2);
          }
          else {
          // El tipo del parámetro no es ni int ni char. Se supone que tampoco es bool
            cadAuxiliar2 = operacion(valorParametro[pos].valor,operador,
                                     literal,valorParametro[pos].tipo,0,0);
            cadAuxiliar = concatenar(cadAuxiliar,cadAuxiliar2);
          }
        }
        else   // No hay operador en el parámetro
          cadAuxiliar = concatenar(cadAuxiliar,valorParametro[pos].valor);

generarParteCodigoParametrizado2(nodo,seleccionEscogida,termino,contadorFrag,
                                 valorParametro,totalElemArrayVP,cadAuxiliar,
                                 indices, paramPasados + 1,nivel,continuacion);
      } // fin (estaEn(valorParametro,totalElemArrayVP,parametro,pos,nivel))
      // Si no, hay que generar los valores para este parámetro
      else  {
        // Se añade al array de parámetros el nombre de éste
        int pos = 0;
        if (!encontrarPosicion(parametro,pos))
          salidaProgramaError(24);
        valorParametro[totalElemArrayVP].nombre = parametro;
        valorParametro[totalElemArrayVP].tipo =
                                          cabeceras[posCabecera].resto.tipoVar[pos];
        valorParametro[totalElemArrayVP].nivel = nivel;
```

```
        char car;
        String cad;
        String cadenaTrue = "true";
        String cadenaFalse = "false";
        encontrarPosicion(parametro,pos);
        // Se mira si especifica rango de valores o no
        if (cabeceras[posCabecera].resto.tipoVar[pos].AnsiCompare("int") == 0 ||
            cabeceras[posCabecera].resto.tipoVar[pos].AnsiCompare("char") == 0) {
           // Si sí lo hacen, se toma su rango de valores
           tomarValoresRango2(posCabecera,pos,valor1,valor2,esInt);
           for (int j = valor1; j <= valor2; j++) {
             if (hayOperador) {
              if (esInt) {
                 cad = operacion(IntToStr(j),operador,literal,"int",valor1,valor2);
                 valorParametro[totalElemArrayVP].valor = IntToStr(j);
              }
              else {
                 cad = (char) j;
                 cad = operacion(cad,operador,literal,"char",valor1,valor2);
                 valorParametro[totalElemArrayVP].valor = cad;
              }
             }
             else {
              // Si no hay operador, se generan simplemente
              if (esInt) {
                 cad = IntToStr(j);
                 valorParametro[totalElemArrayVP].valor = cad;
              }
              else {
                 cad = (char) j;
                 valorParametro[totalElemArrayVP].valor = cad;
              }
             }
             cadAuxiliar = concatenar(cadAuxiliar,cad);
             // y se continua generando el valor para el resto de parámetros

generarParteCodigoParametrizado2(nodo,seleccionEscogida,termino,contadorFrag,
                                  valorParametro,totalElemArrayVP+1,cadAuxiliar,
                                  indices,paramPasados + 1,nivel,continuacion);
             cadAuxiliar = cadResultado;
           }
        } // fin de igualdad de tipos con int o char
        else { // Si no es de estos tipos
        // Si es bool se escribe true y false
          if (cabeceras[posCabecera].resto.tipoVar[pos].AnsiCompare("bool") == 0) {
            for (int j = 0; j < 2; j++) {
              if (j == 0)
                valorParametro[totalElemArrayVP].valor = cadenaTrue;
              else
                valorParametro[totalElemArrayVP].valor = cadenaFalse;
```

```
            String anadido = valorParametro[totalElemArrayVP].valor;
            cadAuxiliar = concatenar(cadAuxiliar,anadido);
            // y se continua generando el valor para el resto de parámetros

generarParteCodigoParametrizado2(nodo,seleccionEscogida,termino,contadorFrag,
                                 valorParametro,totalElemArrayVP+1,cadAuxiliar,
                                 indices,paramPasados + 1,nivel,continuacion);
            cadAuxiliar = cadResultado;
           }// fin for
         }  // fin if
         else { // Lo ha declarado el usuario
          // Buscamos el tipo en el array de tipos para tener sus valores
          int i = 0;
          int posTipoVariable;
          String anadido;
          while (i < numTipos &&
           tipos[i].nombre.AnsiCompare(cabeceras[posCabecera].resto.tipoVar[pos]) !=
                                       0)
            i = i + 1;
          for (int j = 0; j < tipos[i].numValores; j++) {
           // Se escribe el valor j-ésimo de este tipo y se llama a este método
           // para que haga lo mismo con los valores de los parámetros posteriores
           if (hayOperador) {
             anadido = operacion(tipos[i].valores[j],operador,literal,tipos[i].nombre,0,0);
             valorParametro[totalElemArrayVP].valor = anadido;
           }
           else {
             valorParametro[totalElemArrayVP].valor = tipos[i].valores[j];
             anadido = valorParametro[totalElemArrayVP].valor;
           }
           cadAuxiliar = concatenar(cadAuxiliar,anadido);
            // y se continua generando el valor para el resto de parámetros

generarParteCodigoParametrizado2(nodo,seleccionEscogida,termino,contadorFrag,
                                 valorParametro,totalElemArrayVP+1,cadAuxiliar,
                                 indices,paramPasados + 1,nivel,continuacion);
            cadAuxiliar = cadResultado;
           }// fin for
          }//fin else. Tipo declarado por el usuario.
        }//fin else. Sus tipos no especifican rangos.
       }// fin else. Se han generado todos los valores para el tipo.
     }//No se trata de un literal solo, sino que hay seguro un parámetro.
   }
   else { // Se han recorrido todos los parámetros y se actualiza la estrucura indice
    indices->indiceParam = indices->indiceParam + 1;
    // Se genera el resto de código para lo que falta de elección
    analizarComponentes(nodo,seleccionEscogida,termino,contadorFrag +
                        1,continuacion,cadResultado,valorParametro,
                        totalElemArrayVP,indices,nivel);
    indices->indiceParam = indices->indiceParam - 1;
```

```
   }
}

//---------------------------------------------------------------------------
// Método que dado el numero de cabecera del array de cabeceras y una posicion
// determina el número de tipos por defecto que requiere la especificación de rangos
// que hay hasta esa posición pasada como parámetro
int TForm1 :: encuentraParametro(int numCabecera, int indice)
{
  String tipoInt = "int";
  String tipoChar = "char";
  int numPorDefecto = 0;
  int posicion = 0;
  while (posicion < indice) {
     if (cabeceras[numCabecera].parametros[numPorDefecto].AnsiCompare(tipoInt)
         == 0
        || cabeceras[numCabecera].parametros[numPorDefecto].AnsiCompare(tipoChar)
         == 0)
       numPorDefecto = numPorDefecto + 1;
     posicion = posicion + 1;
  }
  return numPorDefecto;
}
//---------------------------------------------------------------------------
// Método que dado un indicador del array que almacena las cabeceras de los
// procesos, y una posición del array que contiene los rangos devuelve los
// valores extremos dados para ese parámetro y si es de tipo int o char
void TForm1 :: tomarValoresRango(int numCabecera, int posArrayRangos,
                                 int posArrayTipos, int &extremo1,
                                 int &extremo2,bool &esInt)
{
  String cadExtremo1 =
                 cogerHasta(cabeceras[numCabecera].rangos[posArrayRangos],'|');
  String cadExtremo2 =
                 cogerDesde(cabeceras[numCabecera].rangos[posArrayRangos],'|');
  int aux1;
  int aux2;
  char car1;
  char car2;
  // Si es de tipo int
  if (cabeceras[numCabecera].parametros[posArrayTipos].AnsiCompare("int") == 0) {
    aux1 = StrToInt(cadExtremo1);
    aux2 = StrToInt(cadExtremo2);
    esInt = true;
  }
  // Si es de tipo char
  else if (cabeceras[numCabecera].parametros[posArrayTipos].AnsiCompare("char")
           == 0) {
    car1 = tomarCaracter(cadExtremo1,1);
    car2 = tomarCaracter(cadExtremo2,1);
```

```
    aux1 = (int) car1;
    aux2 = (int) car2;
    esInt = false;
   }
   // Ahora se ordenan de modo que el primer parámetro devuelva el más bajo
   // y el otro el más alto
   if (aux1 >= aux2) {
     extremo1 = aux2;
     extremo2 = aux1;
   }
   else {
     extremo1 = aux1;
     extremo2 = aux2;
   }
}
//---------------------------------------------------------------------
// Método que coge parte de una cadena hasta alcanzar una carácter dado
String TForm1:: cogerHasta(String cadena, char caracter)
{
  String resultado = " ";
  int indice = 1;
  while (caracter != cadena[indice]) {
     InsertaCaracter(resultado,cadena[indice]);
     indice = indice + 1;
  }
  return resultado;
}
//---------------------------------------------------------------------
// Método que coge el resto de una cadena a partir de una carácter dado
String TForm1:: cogerDesde(String cadena, char caracter)
{
  String resultado = " ";
  int indice = 1;
  while (caracter != cadena[indice])
     indice = indice + 1;
  indice = indice + 1;
  while (indice <= cadena.Length()) {
    InsertaCaracter(resultado,cadena[indice]);
    indice = indice + 1;
  }
  return resultado;
}
//---------------------------------------------------------------------------
// Método que toma un carácter a partir de una posición dada
char TForm1:: tomarCaracter(String cadena, int pos)
{
   if (pos > cadena.Length())
     return ' ';
   else
     return cadena[pos];
```

```
}
//---------------------------------------------------------------------------
// Método que comprueba si un parámetro está en la lista de valores temporales,
// siempre y cuando el nivel actual esté por debajo o al mismo nivel que el
// encontrado
bool TForm1 :: estaEn(ListaDoble* listaCadenas,int numElementos,
                      String cadena, int &pos, int nivel)
{
  int indice = 0;
  bool parar = false;
  while (indice < numElementos && !parar) {
    if (cadena.AnsiCompare(listaCadenas[indice].nombre)== 0 &&
listaCadenas[indice].nivel <= nivel)
      parar = true;
    else
      indice = indice + 1;
  }
  if (parar)
    pos = indice;
  return parar;
}
//---------------------------------------------------------------------------
// Método que toma el rango de valores para parámetros declarados al margen de los
// de la cabecera
void TForm1 :: tomarValoresRango2(int numCabecera, int posArrayRangos,
                                  int &extremo1, int &extremo2,bool &esInt)
{
  // Se cogen los valores de los rangos
  String cadExtremo1 =
cogerHasta(cabeceras[numCabecera].resto.rangoVal[posArrayRangos],'|');
  String cadExtremo2 =
cogerDesde(cabeceras[numCabecera].resto.rangoVal[posArrayRangos],'|');
  int aux1;
  int aux2;
  char car1;
  char car2;
  // Si es de tipo int
  if (cabeceras[numCabecera].resto.tipoVar[posArrayRangos].AnsiCompare("int") == 0)
 {
    aux1 = StrToInt(cadExtremo1);
    aux2 = StrToInt(cadExtremo2);
    esInt = true;
  }
  // Si es de tipo char
  else if
(cabeceras[numCabecera].resto.tipoVar[posArrayRangos].AnsiCompare("char") == 0)
  {
    car1 = tomarCaracter(cadExtremo1,1);
    car2 = tomarCaracter(cadExtremo2,1);
    aux1 = (int) car1;
```

```
    aux2 = (int) car2;
    esInt = false;
   }
   // Ahora se ordenan de modo que el primer parámetro devuelva el más bajo
   // y el otro el más alto
   if (aux1 >= aux2) {
    extremo1 = aux2;
    extremo2 = aux1;
   }
   else {
    extremo1 = aux1;
    extremo2 = aux2;
   }
}
//---------------------------------------------------------------------------
// Método que encuentra en qué posición del array de cabeceras se encuentra
// la que coincide con la del proceso actual
int TForm1 :: encontrarCabecera(int numeroProceso,bool salir) {
  int i = 0;
  bool parar = false;
  while (!parar && i < numCabeceras) {
     if (estructura[numeroProceso].nombre.AnsiCompare(cabeceras[i].proceso) == 0
        && estructura[numeroProceso].numParametros == cabeceras[i].numParametros)
       parar = true;
     else
       i = i + 1;
  }
  if (!parar && salir)
    salidaProgramaError(4);
  else if (!parar && !salir)
    return -1;
  else
    return i;
}
//---------------------------------------------------------------------------
// Método la posición de un parámetro declarado al margen de la cabecera en la
// estructura corespondiente
bool TForm1 :: encontrarPosicion(String variable, int &posicion)
{
  int indice = 0;
  bool parar = false;
  while (!parar && indice < cabeceras[posCabecera].resto.total) {
    if (variable.AnsiCompare(cabeceras[posCabecera].resto.nombreVar[indice]) == 0)
      parar = true;
    else
      indice = indice + 1;
  }
  posicion = indice;
  return parar;
}
```

```
//---------------------------------------------------------------------------
// Método que busca la posición de un tipo en la estrcutura que los almacena
bool TForm1 ::buscarPosicionTipo(String* lista,String nombre, int tamanio,int &pos)
{
  int indice = 0;
  bool parar = false;
  while (!parar && indice < tamanio) {
    if (nombre.AnsiCompare(lista[indice]) == 0)
      parar = true;
    else
      indice = indice + 1;
  }
  return parar;
}
//---------------------------------------------------------------------------
// Método que recibe dos cadenas y devuelve la concatenación de ambas
String TForm1 :: concatenar(String cadena1, String cadena2)
{
  for (int i = 1; i < cadena2.Length(); i++)
    InsertaCaracter(cadena1,cadena2[i]);
  if (cadena2[cadena2.Length()] != '\0')
    InsertaCaracter(cadena1,cadena2[cadena2.Length()]);
  return cadena1;
}
//---------------------------------------------------------------------------
// Método que sale del programa cuando se da alguna situación de error advirtiendo
// del mismo. MENSAJES INACABADOS.
void TForm1:: salidaProgramaError(int codigoError, String param1, String param2)
{
   String cadena = estructura[numProcesos].nombre;
   if (!estructura[numProcesos].simple) {
     InsertaCaracter(cadena,'(');
     for (int i = 0; i < estructura[numProcesos].numParametros - 1; i++) {
       cadena = concatenar(cadena,estructura[numProcesos].nombreParametro[i]);
       InsertaCaracter(cadena,',');
     }
     cadena =
concatenar(cadena,estructura[numProcesos].nombreParametro[estructura[numProcesos]
.numParametros - 1]);
     InsertaCaracter(cadena,')');
   }

   switch (codigoError) {
   // 3: Analizando el código de un proceso tras un término aparace un
   // carácter que no es '+' ni EOF ni es otro proceso
        case 3:
             ShowMessage(AnsiString("Se encontró un carácter que no es '+' ni fin de
                                    fichero ni es otro proceso")
                                    +
                                    AnsiString(" distinto \nen el código del proceso: ")
```

```
                                +
                                AnsiString(cadena));
            break;
  //Operador desconocido
      case 6:
            ShowMessage(AnsiString("Se encontró el operador desconocido") + param1 +
                              AnsiString(" en el código de una condición\n de selección del
                                          proceso ") +
                              cadena);
            break;
  //Operador desconocido lógico, distinto de & (and) |(or) !(not)
      case 7:
            ShowMessage(AnsiString("Se encontró el operador desconocido") + param1 +
                               AnsiString(" en el código de una condición\n de selección del
                                           proceso ")  +
                               cadena);
            break;

  // Se ha encontrado un tipo no existente cuando se evalúa una condición
      case 9:
            ShowMessage(AnsiString("El tipo") + param1 + AnsiString("encontrado en el
                               proceso")+ cadena + AnsiString(" es desconocido"));
            break;
  // Al evaluar una condición se pide comparar con un valor que no existe para ese tipo
      case 10:
            ShowMessage(AnsiString("El tipo") + param1 + AnsiString("encontrado en el
                              proceso")+ cadena + AnsiString(" no tiene el valor") + param2 );
            break;
      default:
             ShowMessage(AnsiString("Hubo un error al realizar la conversión") +
                               codigoError);
            break;
      //case 4:
             // En la lectura dee una condición de un if se ha encontrado
             // un operador desconocido
     // case 5: En un if hay un único operando para un operador binario
     // case 6: Hay un operador aritmético y se esperaba uno lógico
     // case 7: Hay un operador lógico y se esperaba uno aritmético
     // case 8 : No existe ese tipo
     // case 9: Ese valor no pertenece a ese tipo
  }
  liberarRecursos(raiz,true,true);
  exit(0);
}
//---------------------------------------------------------------------------
// Método que almacena la condición de una selección en una estructura de tipo
seleccion
void TForm1 :: leerCondicion(FILE* fichero,seleccion nodo, int numSel,
                                    int posArray, bool quitarParentesis)
{
```

```
// Operadores lógicos: AND &, OR |, NOT !, igual =  ;
// Operadores aritméticos: MAYOR >, MAYOR O IGUAL >=, MENOR <, MENOR
//  IGUAL <=
String cadena = " ";
char caracter;
if (quitarParentesis)
  caracter = encuentraCaracter(fichero,'(');
caracter = QuitarBlancosCarro(fichero);
InsertaCaracter(cadena,caracter);
// Leo un operador y lo almaceno
if(caracter == '<' || caracter == '>') {
  fread(&caracter, sizeof(caracter), 1, fichero);
  nodo.condiciones[numSel][posArray].esNot = false;
  if (caracter == '=') {
     InsertaCaracter(cadena,caracter);
     fread(&caracter, sizeof(caracter), 1, fichero);
   }
  nodo.condiciones[numSel][posArray].operador = cadena;
  vaciarCadena(cadena);
}
// Si no, es un operador lógico
else {
  if (caracter == '!')
    nodo.condiciones[numSel][posArray].esNot = true;
  else if (caracter == '|' || caracter == '&' || caracter == '=')
    nodo.condiciones[numSel][posArray].esNot = false;
  else
    salidaProgramaError(4);
  // se almacena el operador
  nodo.condiciones[numSel][posArray].operador = caracter;
  vaciarCadena(cadena);
  fread(&caracter, sizeof(caracter), 1, fichero);
}

if (caracter == ' ' || caracter == '\n')
  caracter = QuitarBlancosCarro(fichero);

if (caracter == '(') //---> Viene otra expresión (operador ...)
// viene oper op1 op2) (oper op1 op2) {
   nodo.condiciones[numSel][posArray].hayOperandos = false;
   leerCondicion(fichero,nodo,numSel,2*posArray,false);
   // Si no es un not viene (oper op1 op2)
   if (!nodo.condiciones[numSel][posArray].esNot)
     leerCondicion(fichero,nodo,numSel, 2*posArray + 1,true);
   encuentraCaracter(fichero,')');
}
else {
  // Viene un operando
  nodo.condiciones[numSel][posArray].hayOperandos = true;
  while (caracter != ' ' && caracter != '\n') {
```

```
        InsertaCaracter(cadena,caracter);
        fread(&caracter, sizeof(caracter), 1, fichero);
      }
      // Ya tenemos el primer operando. Ahora por el segundo
       nodo.condiciones[numSel][posArray].operando1 = cadena;
      vaciarCadena(cadena);
      caracter = QuitarBlancosCarro(fichero);
      // VER QUE EL USUARIO NO HAYA PUESTO LOS DOS OPERANDOS
JUNTOS
      // Viene el 2 operando
      while (caracter != ' ' && caracter != '\n' && caracter != ')') {
        InsertaCaracter(cadena,caracter);
        fread(&caracter, sizeof(caracter), 1, fichero);
      }
      if (cadena.AnsiCompare(" ") == 0)
        salidaProgramaError(5);
      //Ahora leemos hasta encontrar el caracter de cierre ')'
      if (caracter != ')')
        encuentraCaracter(fichero,')');
      nodo.condiciones[numSel][posArray].operando2 = cadena;
    }// fin else. Viene un operando
      // EN EL CODIGO DESDE DONDE LO LLAMES VIENE LA INSTRUCCION
QUE SIGUE
}
//---------------------------------------------------------------------------
// Método que evalúa la condición de una estructura de selección. Puede ser de un
// if o un else
bool TForm1 :: evaluarCondicion(seleccion nodo,int numeroIf,
                                        int posArray, ListaDoble* lista, int tamLista)
{
  bool return1 = false;
  // Si hay operandos se toman
  if(nodo.condiciones[numeroIf][posArray].hayOperandos) {
      int posicion;
      String operando1;
      String operando2;
      bool esta1 = false;
      bool esta2 = false;
      // Se comprueba si está el primer operando en la lista de valores temporales
      // y se toma de aquí
      if (estaEn(lista,tamLista,nodo.condiciones[numeroIf][posArray].operando1,
           posicion,INT_MAX)) {
        operando1 = lista[posicion].valor;
        esta1 = true;
      }
      // Si no, está el valor en la estructura de condición
      else
        operando1 = nodo.condiciones[numIf][posArray].operando1;
      // Idem para el segundo operando
      if (estaEn(lista,tamLista,nodo.condiciones[numeroIf][posArray].operando2,
```

```
        posicion,INT_MAX)) {
      operando2 = lista[posicion].valor;
      esta2 = true;
    }
    else
      operando2 = nodo.condiciones[numeroIf][posArray].operando2;
    // Si no había ningún parámetro se mira el tipo de los valores
    // leídos. SE SUPONE INT O CHAR
    if (!esta1 && !esta2) {
      if (esLetra(operando1) && esLetra(operando2))
        lista[posicion].tipo = "char";
      else if (esNumero(operando1) && esNumero(operando2))
        lista[posicion].tipo = "int";
      else
        salidaProgramaError(22);
    }
    // Ahora se ve de qué operador se trata y se hace la operación
    if (nodo.condiciones[numeroIf][posArray].operador.AnsiCompare("<") ==0)
      return1 = menor(operando1,operando2,lista[posicion].tipo);
    else if (nodo.condiciones[numeroIf][posArray].operador.AnsiCompare("<=") ==0)
      return1 = menorIgual(operando1,operando2,lista[posicion].tipo);
    else if (nodo.condiciones[numeroIf][posArray].operador.AnsiCompare(">") ==0)
      return1 = mayor(operando1,operando2,lista[posicion].tipo);
    else if (nodo.condiciones[numeroIf][posArray].operador.AnsiCompare(">=") ==0)
      return1 = mayorIgual(operando1,operando2,lista[posicion].tipo);
    else if (nodo.condiciones[numeroIf][posArray].operador.AnsiCompare("&") ==0)
      return1 = and(operando1,operando2);
    else if (nodo.condiciones[numeroIf][posArray].operador.AnsiCompare("|") ==0)
      return1 = or(operando1,operando2);
    else if (nodo.condiciones[numeroIf][posArray].operador.AnsiCompare("=") == 0)
      return1 = operando1.AnsiCompare(operando2) == 0;
    else
       salidaProgramaError(7,nodo.condiciones[numeroIf][posArray].operador);
  } // fin   if(nodo.condiciones[numeroIf][posArray].hayOperandos)
  else
  // Si no hay operandos, se trata de una operación lógica aplicada a dos hijos
  // de la estructura
    if (nodo.condiciones[numeroIf][posArray].operador.AnsiCompare("|") == 0)
      return1 = evaluarCondicion(nodo,numeroIf,posArray * 2, lista, tamLista) ||
                evaluarCondicion(nodo,numeroIf,posArray*2+1, lista, tamLista);
    else if (nodo.condiciones[numeroIf][posArray].operador.AnsiCompare("&") == 0)
      return1 = evaluarCondicion(nodo,numeroIf,posArray * 2, lista, tamLista) &&
                evaluarCondicion(nodo,numeroIf,posArray*2+1, lista, tamLista);
    else if (nodo.condiciones[numeroIf][posArray].operador.AnsiCompare("!") == 0)
      return1 = (!evaluarCondicion(nodo,numeroIf,posArray * 2, lista, tamLista));
    else
      salidaProgramaError(6,nodo.condiciones[numeroIf][posArray].operador);
  return return1;
}
//---------------------------------------------------------------------------
```

```
// Método que realiza la operación and. Se supone que recibe dos booleanos
bool TForm1 :: and(String cadena1,String cadena2)
{
  if(cadena1.AnsiCompare("false")== 0 || cadena2.AnsiCompare("false")== 0)
    return false;
  else
    return true;
}
//---------------------------------------------------------------------------
// Método que realiza la operación or. Se supone que recibe dos booleanos
bool TForm1 :: or(String cadena1,String cadena2)
{
  if(cadena1.AnsiCompare("false")== 0 && cadena2.AnsiCompare("false")== 0)
    return false;
  else
    return true;
}
//---------------------------------------------------------------------------
// Método que comprueba dadas dos cadenas y un tipo cuál es menor. Si es un entero
// o un carácter se compara su valor transformándolo a entero o viendo su posición
// en el código ASCII. Si es de otro tipo, se mira su posición dentro de los valores
// de los mismos. Se supone que no recibe valores de tipo bool
bool TForm1 :: menor(String cadena1,String cadena2,String tipo)
{
 if (tipo.AnsiCompare("int") == 0)
   return StrToInt(cadena1) < StrToInt(cadena2);
 else if (tipo.AnsiCompare("char") == 0)
   return cadena1 < cadena2;
 else {
   int posEnTipo1 = posicionValorEnTipo(cadena1,tipo);
   int posEnTipo2 = posicionValorEnTipo(cadena2,tipo);
   return posEnTipo1 < posEnTipo2;
 }

}
//---------------------------------------------------------------------------
// Idem al anterior sólo que con la operación menorIgual.
bool TForm1 :: menorIgual(String cadena1,String cadena2,String tipo)
{
 if (tipo.AnsiCompare("int") == 0)
   return StrToInt(cadena1) <= StrToInt(cadena2);
 else if (tipo.AnsiCompare("char") == 0)
   return cadena1 <= cadena2;
 else {
   int posEnTipo1 = posicionValorEnTipo(cadena1,tipo);
   int posEnTipo2 = posicionValorEnTipo(cadena2,tipo);
   return posEnTipo1 <= posEnTipo2;
 }

}
```

```
//---------------------------------------------------------
// Idem al anterior sólo que con la operación mayor.
bool TForm1 :: mayor(String cadena1,String cadena2,String tipo)
{
  if (tipo.AnsiCompare("int") == 0)
    return StrToInt(cadena1) > StrToInt(cadena2);
  else if (tipo.AnsiCompare("char") == 0)
    return cadena1 > cadena2;
  else {
    int posEnTipo1 = posicionValorEnTipo(cadena1,tipo);
    int posEnTipo2 = posicionValorEnTipo(cadena2,tipo);
    return posEnTipo1 > posEnTipo2;
  }
}
//---------------------------------------------------------------------------
// Idem al anterior sólo que con la operación mayor o igual.
bool TForm1 :: mayorIgual(String cadena1,String cadena2,String tipo)
{
  if (tipo.AnsiCompare("int") == 0)
    return StrToInt(cadena1) >= StrToInt(cadena2);
  else if (tipo.AnsiCompare("char") == 0)
    return cadena1 >= cadena2;
  else {
    int posEnTipo1 = posicionValorEnTipo(cadena1,tipo);
    int posEnTipo2 = posicionValorEnTipo(cadena2,tipo);
    return posEnTipo1 >= posEnTipo2;
  }
}
//---------------------------------------------------------------------------
// Método que encuentra la posición de un valor en un tipo declarado por el usuario
int TForm1 :: posicionValorEnTipo(String valor, String nombreTipo)
{
  bool parar = false;
  int indice = 0;
  while (indice < numTipos && !parar) {
     if (nombreTipo.AnsiCompare(tipos[indice].nombre) == 0)
       parar = true;
     else
       indice = indice + 1;
  }
  // TIPO NO ENCONTRADO
  if (!parar)
    salidaProgramaError(9,nombreTipo);
  int indice2 = 0;
  parar = false;
  while (indice2 < tipos[indice].numValores && !parar) {
     if (valor.AnsiCompare(tipos[indice].valores[indice2]) == 0)
       parar = true;
     else
       indice2 = indice2 + 1;
```

```
  }
  // VALOR NO EXISTENTE
  if (!parar)
    salidaProgramaError(10,nombreTipo,valor);
  return indice2;
}
//---------------------------------------------------------------------------
// Método que recibe una cadena y mira si es una letra o carácter
bool TForm1 :: esLetra (String cadena)
{
   //Una cadena consta de caracter + '\0' = Fin de cadena, si tiene sólo un carácter
   if (cadena.Length() > 2)
     return false;
   else
     return true;
}
//---------------------------------------------------------------
// Método que dada una cadena devuelve el entero, en caso de que ésta contenga
// sólo un número
bool TForm1 :: esNumero (String cadena)
{
  int indice = 1;
  bool parar = false;
  while (indice < cadena.Length() && !parar) {
    if (!isdigit(cadena[indice]))
      parar = true;
    else
      indice = indice + 1;
  }
  return !parar;
}

//-----------------------------------------------------------------------------------------
// Método que comprueba que haya una estructura de selección en un término
bool TForm1 :: buscaElemEnFragmento(int posInicio,Orden ordenacion,
                                                  int &distancia,int codigoElem)
{
   bool parar = false;
   int indice = posInicio;
   while (! parar && indice < ordenacion.tamanio) {
      if (ordenacion.orden[indice] == codigoElem)
        parar = true;
      indice = indice + 1;
   }
   if (parar) {
     distancia = indice + posInicio;
     return true;
   }
   else
     return false;
```

```
}
//---------------------------------------------------------------------------------------------
// Método que actualiza los índices indicándole a qué nodo va a ir y en qué posición
void TForm1 :: actualizarIndices(Indice* indices, seleccion nodo, int numSeleccion,
                                 int numTermino, int numComponente)
{
  int indice = 0;
  indices->indiceParam = 0;
  indices->indiceTP = 0;
  indices->indiceIf = 0;
  if (nodo.ordenacion[numSeleccion][numTermino].tamanio > numComponente) {
    while (indice < numComponente) {
      if (nodo.ordenacion[numSeleccion][numTermino].orden[indice] == 0)
        indices->indiceIf = indices->indiceIf + 1;
      else if (nodo.ordenacion[numSeleccion][numTermino].orden[indice] == 1)
        indices->indiceParam = indices->indiceParam + 1;
      else
        indices->indiceTP = indices->indiceTP + 1;
      indice = indice + 1;
    }
  }
}

//---------------------------------------------------------------------------------------------
// Método que libera los recursos solicitados para almacenar la información del
// cuerpo de un proceso
void TForm1 :: liberarRecursos(seleccion nodo,bool todaEstructura,bool esRaiz)
{
 int posIf = 0;
 int posTP = 0;
 int posParam = 0;
 // Se recorren todas las selecciones de un if
 for (int i = 0; i < nodo.numeroSelecciones; i++) {
   // Se recorren todas las elecciones
   for (int j = 0; j < nodo.numFragSelec[i]; j++) {
     // Se recorren todos los elemnetos de cada elección
     for (int k = 0; k < nodo.ordenacion[i][j].tamanio; k++) {
       if (nodo.ordenacion[i][j].orden[k] == 0) { // --> es un if
         // Se liberan los recursos llamando recursivamente
         liberarRecursos(nodo.ifs[i][j][posIf],true,false);
         posIf = posIf + 1;
       }
       if (nodo.ordenacion[i][j].orden[k] == 1 ) { //--> parámetro
         nodo.parametros[i][j].termino[posParam].numeroPartes = 0;
         nodo.parametros[i][j].termino[posParam].numeroPartes = 0;
         delete [] nodo.parametros[i][j].termino[posParam].partes;
         nodo.parametros[i][j].termino[posParam].partes = NULL;
         posParam = posParam + 1;
       }
       if (nodo.ordenacion[i][j].orden[k] == 2) // --> señal simple
```

```
            posTP = posTP + 1;

        }// fin for k
        nodo.ordenacion[i][j].tamanio = 0;
        // Si había parámetros se borra la estructura reservada para los mismos
        if (posParam != 0) {
          delete [] nodo.parametros[i][j].termino;
          nodo.parametros[i][j].termino = NULL;
        }
        nodo.textoPlano[i][j].numeroPartes = 0;
        delete [] nodo.ordenacion[i][j].orden;
        nodo.ordenacion[i][j].orden = NULL;
        // Idem para las señales simples
        if (posTP != 0) {
          delete [] nodo.textoPlano[i][j].partes;
          nodo.textoPlano[i][j].partes = NULL;
        }
        // Idem para los ifs
        if (posIf > 1) {
          delete [] nodo.ifs[i][j];
          nodo.ifs[i][j] = NULL;
        }
        posIf = 0;
        posParam = 0;
        posTP = 0;
      } // fin for j
      // Si no es la raíz y no es un else se borran las condiciones
      if (!esRaiz && nodo.condiciones[i] != NULL) { // no es un else
        delete [] nodo.condiciones[i];
        nodo.condiciones[i] = NULL;
      }
      delete [] nodo.parametros[i];
      nodo.parametros[i] = NULL;
      delete [] nodo.textoPlano[i];
      nodo.textoPlano[i] = NULL;
      delete [] nodo.ordenacion[i];
      nodo.ordenacion[i] = NULL;
      delete [] nodo.ifs[i];
      nodo.ifs[i] = NULL;
    }// fin for i
    if (todaEstructura) {
      nodo.numeroSelecciones = 0;
      delete[] nodo.textoPlano;
      nodo.textoPlano = NULL;
      if (!esRaiz) {
        delete[] nodo.condiciones;
        nodo.condiciones = NULL;
      }
      delete[] nodo.numFragSelec;
      nodo.numFragSelec = NULL;
```

```
    delete[] nodo.ordenacion;
    nodo.ordenacion = NULL;
    delete[] nodo.ifs;
    nodo.ifs = NULL;
    delete[] nodo.parametros;
    nodo.parametros = NULL;
   }
}

//-----------------------------------------------------------------------------------
// Método que comprueba que hay un carácter dado en una cadena pasada
bool TForm1 :: pertenece(char caracter, String cadena)
{
  bool parar = false;
  int indice = 1;
  // Mientras no se haya encontrado y la longitud sea menor que el tamaño
  while (!parar && indice < cadena.Length()) {
     if (cadena[indice] == caracter)
       parar = true;
     else
       indice = indice + 1;
  }
  return parar;
}
//-----------------------------------------------------------------------------------
// Método que coge un operador en una cadena correspondiente a un parámetro
// Operadores posible + -
char TForm1 :: cogerOperador(String cadena)
{
  int indice = 1;
  bool parar = false;
  while (!parar && indice < cadena.Length()) {
     if (cadena[indice] ==  '+' || cadena[indice] ==  '-')
       parar = true;
     else
       indice = indice + 1;
  }
    return cadena[indice];
}

//-----------------------------------------------------------------------------------
// Método que toma el literal de una cadena correspondiente a un parámetro
// Literales representados de la forma [valor litera]
String TForm1 :: cogerLiteral(String cadena)
{
  int indice = 1;
  bool parar = false;
  String resultado = " ";
  while (cadena[indice] !=  '[')
    indice = indice + 1;
```

```
  //Se salta la marca
  indice = indice + 1;
  while (cadena[indice] !=  ']') {
    InsertaCaracter(resultado,cadena[indice]);
    indice = indice + 1;
  }
  return resultado;
}
//---------------------------------------------------------------------------------------
// Método que toma el parámetro en una cadena correspondiente a un parámetro que
tiene
// un operador y una literal
// El parámetro aparece en la forma: param op [literal], o al revés
String TForm1 :: cogerParametro(String cadena)
{
  String resultado = " ";
  int indice = 1;
  if (cadena[1] == '[') { //---> Primero viene el literal
    char caracter;
     //Saltamos hasta el operador
    while (cadena[indice] != '+' && cadena[indice] != '-')
      indice = indice + 1;
    // Saltamos el operador
    indice = indice + 1;
    // Saltamos los blancos
    while (cadena[indice] == ' ') {
      indice = indice + 1;
      caracter = cadena[indice];
    }
    //Cogemos el parámetro
    while (cadena[indice] != ' ') {
      caracter = cadena[indice];
      InsertaCaracter(resultado,cadena[indice]);
      indice = indice + 1;
      if (indice == cadena.Length()+ 1)
        break; // Ha acabado la cadena  salimos del bucle
    }
  }
  else {  //---> Primero viene el parámetro
    while (cadena[indice] != '+' && cadena[indice] != '-' && cadena[indice] != ' ') {
      InsertaCaracter(resultado,cadena[indice]);
      indice = indice + 1;
    }
  }
  return resultado;
}
//---------------------------------------------------------------------------------------
// Método que efectúa la operación entre el valor del parámetro y el literal
String TForm1 :: operacion(String valorParametro, char operador, String literal,
                           String tipo, int extremoInf, int extremoSup)
```

```
{
  // Si no es int o char --> lo ha declarado el usuario, y se accede al array de
  // valores y se comprueba su posición
  if (tipo.AnsiCompare("int") != 0 && tipo.AnsiCompare("char") != 0) {
    // Se ecncuentra el tipo
   int indice = 0;
   while (indice < numTipos &&
         tipos[indice].nombre.AnsiCompare(tipo) != 0)
     indice = indice + 1;
   int indice2 = 0;
   bool parar = false;
   // Se encuentra la posición del valor
   while (indice2 < tipos[indice].numValores && !parar) {
     if (literal.AnsiCompare(tipos[indice].valores[indice2]) == 0)
       parar = true;
     else
       indice2 = indice2 + 1;
   }
   if (!parar)
     salidaProgramaError(27); //Ese valor no pertenece al parámetro
   int indice3 = 0;
   parar = false;
   // Idem para el segundo valor
   while (indice3 < tipos[indice].numValores && !parar) {
     if (valorParametro.AnsiCompare(tipos[indice].valores[indice3]) == 0)
       parar = true;
     else
       indice3 = indice3 + 1;
   }
   // Se devuelve la suma dependiendo de que se haya seleccionado la operación
   // módulo o la techo/suelo
   if (operador == '+') {
     if (operacionModular)
       return tipos[indice].valores[(indice2 + indice3 + 1)% tipos[indice].numValores];
     else
       return tipos[indice].valores[min(indice2 + indice3 + 1,tipos[indice].numValores -
1)];
   }
   else { // Idem para la resta
     if (operacionModular)
       return tipos[indice].valores[(indice3 - indice2 - 1 + tipos[indice].numValores)%
tipos[indice].numValores];
     else
       return tipos[indice].valores[max(indice3 - indice2 - 1,0)];
   }
 }
 // Si es un entero...
 else if(tipo.AnsiCompare("int") == 0) {
   if (!esNumero(literal)) // Si no es un número --> Salir
     salidaProgramaError(31);
```

```
    else
    // Se hace la operación
      if (operacionModular) {
        if (operador == '+')
          return IntToStr((StrToInt(valorParametro) - extremoInf + StrToInt(literal)) %
                          (extremoSup - extremoInf + 1) + extremoInf);
        else {
          int modulo = (StrToInt(valorParametro) - extremoInf - StrToInt(literal)) %
                       (extremoSup - extremoInf + 1);
          if (modulo < 0)
            modulo = modulo + extremoSup - extremoInf + 1;
          return IntToStr(modulo + extremoInf);
        }
      }
      // Aquí se hace la operación Techo/Suelo
      else {
        if (operador == '+')
          return IntToStr(min(StrToInt(valorParametro) + StrToInt(literal), extremoSup));
        else
          return IntToStr(max(StrToInt(valorParametro) - StrToInt(literal), extremoInf));
      }
  }// fin if (tipo.AnsiCompare("int") == 0)
  // Se hace lo mismo para los carácteres
  // LA OPERACION MODULO SE COMPORTA DE FORMA DIFERENTE A LA
DE LOS ENTEROS
  // ACHTUNG ECHAR UN VISTAZO
  else if(tipo.AnsiCompare("char") == 0) {
    if (!esLetra(literal))
      salidaProgramaError(32);
    else {
      char letraVP = valorParametro[1];
      char letraLiteral = literal[1];
      int numLetraVP = (int)(letraVP) - extremoInf;
      int numLetraLiteral = (int)(letraLiteral) - extremoInf;
      if (operacionModular) {
        if (operador == '+')
          return (char)((numLetraVP + numLetraLiteral + 1) % (extremoSup - extremoInf
+ 1) + extremoInf);
        else {
          int modulo = (numLetraVP - numLetraLiteral) % (extremoSup - extremoInf + 1);
          if (modulo < 0)
            modulo = modulo + extremoSup - extremoInf + 1;
          return (char) (modulo + extremoInf);
        }
      }
      else {
        if (operador == '+')
          return (char)(extremoInf + min(numLetraVP + numLetraLiteral, extremoSup –
                        extremoInf + 1));  //bien
        else
```

```
          return (char)(max(numLetraVP - numLetraLiteral + extremoInf,
                            extremoInf));//bien
       }                                        //-1
     }
   }
   else
     salidaProgramaError(30);
}
//---------------------------------------------------------------------------------------------
void __fastcall TForm1::ComboBox1Change(TObject *Sender)
{
    if (ComboBox1->Text.AnsiCompare("Operación Techo/Suelo") == 0)
      operacionModular = false;
    else
      operacionModular = true;
}
//---------------------------------------------------------------------------
// Método que
int TForm1 :: encuentraPosParam(int posCabecera, String parametro,bool dentro)
{
  int posicion = 0;
  bool parar = false;
  if (dentro)
  {
     while (cabeceras[posCabecera].parametros[posicion].AnsiCompare(parametro) != 0
              && !parar) {
       if (posicion == cabeceras[posCabecera].numParametros)
         parar = true;
       else
        posicion = posicion + 1;
     }
  }
  else {
     while (cabeceras[posCabecera].resto.nombreVar[posicion].AnsiCompare(parametro)
               != 0 && !parar) {
       if (posicion == cabeceras[posCabecera].resto.total)
         parar = true;
       else
        posicion = posicion + 1;
     }
  }
  if (parar)
     salidaProgramaError(33);
  else
    return posicion;
}

//---------------------------------------------------------------------------
// Método que comprueba si hay una literal en una cadena correspondiente
// a un parámetro
```

```
bool TForm1 :: hayLiteral(String cadena)
{
  int indice = 1;
  while (indice < cadena.Length() + 1 && cadena[indice] != '[')
    indice = indice + 1;
  if (indice == cadena.Length() + 1)
    return false;  // La cadena no contiene un literal

  while (indice < cadena.Length() + 1 && cadena[indice] != ']')
    indice = indice + 1;

  if (indice == cadena.Length() + 1)
  // La cadena no contiene el carácter ']', y, por tanto, un literal
    return false;
  else
    return true;
}
//---------------------------------------------------------------------------------------------
// Método que escribe el proceso y el cuerpo de un proceso CCS básico sin parámetros
void TForm1 :: escribirProcesoCCSBasico()
{
   bool primeroEnEscribir = true;
   escribirCadena(ficheroDestino,"\n proc ");
   escribirCadena(ficheroDestino,estructura[numProcesos].nombre);
   escribirCadena(ficheroDestino," = ");
   // Recorremos todas las elecciones
   for (int numElecciones = 0; numElecciones < raiz.numFragSelec[0];
         numElecciones++) {  // Sólo tenemos señales simples, no hay ifs ni parámetros
     for (int termino = 0; termino < raiz.textoPlano[0][numElecciones].numeroPartes;
           termino++) {
       if(!primeroEnEscribir)
         escribirCadena(ficheroDestino,".");
       else
         primeroEnEscribir = false;

escribirCadena(ficheroDestino,raiz.textoPlano[0][numElecciones].partes[termino]);
     }
     primeroEnEscribir = true;
     if (numElecciones !=  raiz.numFragSelec[0] - 1)
       escribirCadena(ficheroDestino,"\n + ");
   }
}
//---------------------------------------------------------------------------------------------
```

Bibliografía para el CCS

Básicamente las fuentes empleadas han sido los libros, por orden de prioridad:
[1] Clive Fencott. Formal Methods for Concurrency. International Thomson Computer Press. 1996.
[2] Glenn Bruns. Distributed Systems Analysis with CCS. Prentice Hall International. 1997.

Bibliografía para el cálculo de ambientes

Las fuentes empleadas son los siguientes artículos:
[1] Mobile ambients: Luca Cardelli, Andrew D. Gordon.
[2] Abstraction for mobile computation: Luca Cardelli.
[3] Mobility types for mobile ambients: Luca Cardelli, Giorgio Ghelli, Andrew D. Gordon.
[4] Types for the ambient calculus: Luca Cardelli, Giorgio Ghelli, Andrew D. Gordon.
[5] Finite control mobile ambients: Witold Charatonik, Andrew D. Gordon, Jan-Marc Talbot.
[6] Expressing Dynamics of Mobile Agent Systems Using Ambient Calculus
Peter Stañski, Arkady Zaslavsky.
[7] Orderly Communication in the Ambient Calculus: Torben Amtoft, Assaf J. Kfoury, Santiago M. Pericas-Geertsen
[8] What are PolymorphicallyTyped Ambients?: Torben Amtoft, Assaf J. Kfoury, Santiago M. Pericas-Geertsen