Especificación del Comportamiento Interactivo de Componentes mediante *Pew*

Silvia N. Amaro[1] **y Ernesto Pimentel**[2]

[1]*Departamento de Informática y Estadística, Universidad Nacional del Comahue*
Buenos Aires 1400, (8300) Neuquén, Argentina
e-mail: samaro@uncoma.edu.ar

[2]*Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga*
Complejo Tecnológico, Campus Teatinos, (29071) Málaga, España
e-mail: ernesto@lcc.uma.es

Resumen

El desarrollo de software basado en componentes es una disciplina en continuo crecimiento dentro del campo de la ingeniería de software. Consideramos un sistema basado en componentes como un conjunto de componentes de software que interactúan. A tal efecto los componentes deben ser diseñados para asegurar que un sistema construido a partir de ellos y sus interfaces satisfará especificaciones de desempeño claramente definidas. En este sentido los lenguajes de descripción de interfaces tradicionales, provistos por las plataformas orientadas a componentes del mercado describen los servicios ofrecidos por los componentes pero no dan información alguna respecto al orden relativo en el que los servicios pueden ser utilizados, de manera que no hay garantía de que los componentes interoperarán con éxito. Nuestra propuesta se orienta a enriquecer la información provista por los lenguajes de descripción de interfaces mediante la descripción de una abstracción del protocolo de interacción de los componentes. Proponemos y analizamos la utilización de *Pew*, un modelo de coordinación basado en cálculo de canales que posee una gran potencia expresiva, para especificar software basado en componentes. Presentamos un álgebra de procesos basada en las primitivas de comunicación y composición de *Pew* que luego aplicamos en algunos ejemplos para especificar el comportamiento interactivo de los componentes involucrados.

Palabras claves: Lenguajes de coordinación, software basado en componentes, cálculo de canales, interoperabilidad, álgebra de procesos.

1 Introducción

La Ingeniería de Software Basada en Componentes (ISBC) propone la creación de colecciones de componentes de software reutilizables que puedan ser adaptadas e interconectadas en forma dinámica en el desarrollo de nuevas aplicaciones, tendiente a responder a la creciente demanda de nuevas tecnologías para la construcción de aplicaciones abiertas y distribuidas. Para desarrollar sistemas de software a partir de la integración de componentes existentes es necesario que exista un mecanismo de composición que habilite su integración. Además la interoperación de componentes puede requerir la preservación de propiedades temporales y funcionales, como por ejemplo restricciones de orden sobre las operaciones o coordinación de las entradas desde múltiples flujos. Podemos encontrar componentes de distintos formatos, diseños e implementaciones. Los componentes pueden diseñarse para trabajar en conjunto o pueden obtenerse en las más diversas fuentes. Todos estos factores influencian en gran medida la tarea de composición de componentes. El desarrollo de sistemas abiertos debería basarse en el uso de componentes y la tecnología debería dar soporte a esa composición.

Un componente es una entidad que puede ser utilizada y compuesta sólo por medio de su interfaz, en la que se describe la entrada/salida y comportamiento observable de las instancias del

componente. La interfaz de un componente provee una abstracción que encapsula los detalles de implementación internos. En el proceso de construcción y uso de componentes [19] debemos tener en cuenta que la composición de objetos es un ingrediente esencial para los modelos con componentes, ya que son necesarios para crear las estructuras utilizadas en tiempo de ejecución; se debe enfatizar la separación entre el uso de componentes y el desarrollo de los mismos; y considerar que los componentes pueden requerir diferentes tipos de conectores. Cualquier composición debería asegurar que la aplicación resultante ha sido correctamente construida y en ese sentido, debería poder definirse qué es una composición correcta.

La investigación sobre lenguajes y modelos de coordinación tiene como meta principal la interoperación de componentes software. Tradicionalmente estos modelos y lenguajes [2, 16] han evolucionado alrededor de las nociones de espacio de datos compartido, por un lado, dando origen a la familia de lenguajes orientados a datos; y control y eventos, por otro lado, dando origen a la familia de lenguajes orientados a control. Dentro del primer grupo se encuentra LINDA, como su principal exponente. Se han desarrollado varios trabajos [12] extendiendo Linda para poder utilizar su máxima potencia expresiva en la especificación de protocolos de componentes [11]. Dentro del segundo grupo de lenguajes se encuentra Manifold, que es pionero en la aplicación del modelo IWIM (Idealized Worker Idealized Manager) [3]. Se ha demostrado en [20] que es muy potente para modelar sistemas distribuidos con reconfiguración dinámica de su topología. Con el propósito de resolver los problemas de interoperabilidad que pueden presentarse al trabajar con los Lenguajes de Descripción de Interfaces provistos por las plataformas orientadas a componentes del mercado (CORBA [7], COM/DCOM [21], EJB, etc [15]), que sólo permiten describir los servicios ofrecidos por los componentes, pero nada especifican respecto a los servicios requeridos, ni el orden relativo que deben respetar, se han presentado varias propuestas para extender las interfaces con información del comportamiento concurrente de los componentes, utilizando álgebras de procesos [10, 13] y representaciones basadas en roles [14]

En este sentido una alternativa que parece prometedora dentro del campo de los modelos de coordinación es trabajar con cálculo de canales. En el cálculo de canales la interfaz consiste de un conjunto de canales móviles a través de los cuales la instancia del componente envía y recibe valores. Que un canal sea móvil significa que las identidades de sus extremos pueden ser pasadas a través de otro canal y así cambiar físicamente la posición de uno de sus extremos. Que un componente sea móvil significa que puede moverse de una posición a otra dentro del sistema distribuido en que esta inmerso. Los canales y componentes móviles permiten la reconfiguración dinámica de la topología de un sistema.

Tanto los modelos basados en canales móviles como los basados en espacio de tuplas compartido y en eventos comparten las características de comunicación anónima, y la separación de los conceptos de coordinación y computación. Sin embargo, la composición de componentes por medio de canales presenta algunas otras ventajas[4]: permite implementaciones más eficientes; provee seguridad -la comunicación punto a punto es una comunicación privada que evita interferencias accidentales o intensionales sobre la transmisión de los datos-, y tiene mayor expresividad arquitectural debido a que los canales son conexiones directas entre componentes, luego son altamente expresivos de la arquitectura del sistema, ya que en este modelo es claro ver qué entidades o componentes pueden ser afectadas por la modificación o reemplazo de algún otro componente.

Pew[1] surge como un lenguaje que permite la construcción composicional de sistemas a partir de componentes que interactúan y cooperan anónimamente a través de conectores. Los conectores se logran por la aplicación de operaciones primitivas en la composición de canales. Los canales proveen el desacople temporal y espacial de las partes que intervienen en una comunicación. ***Pew***

es un lenguaje que respeta el modelo IWIM propuesto en [3], y como tal abstrae los detalles de implementación de los componentes encargados de la computación y presenta un alto potencial para expresar, por medio de sus conectores y su semántica el comportamiento interactivo de componentes software.

El resto de este trabajo es organizado como sigue. La sección 2 introduce los conceptos del modelo propuesto en *P**ew***, dando la semántica del cálculo de canales y conectores, .y algunos ejemplos de construcción de conectores a partir de las operaciones primitivas ofrecidas por *P**ew***. En la sección 3 se propone un cálculo basado en las primitivas de *P**ew***, y se presentan las reglas de transición que modelan dicho cálculo. En la sección 4 se muestra como especificar el comportamiento interactivo de componentes mediante ejemplos. Por último se dan algunas conclusiones y se indica el trabajo futuro.

2 *Pew*: Un Modelo Basado en Canales

*P**ew*** [1] es un modelo de coordinación exógena, basado en canales, en el que a partir de un conjunto de canales provistos por el usuario con comportamiento bien definido, se construyen de manera composicional coordinadores mas complejos llamados conectores. Cada conector impone un patrón de coordinación específico sobre las entidades que conecta.

2.1 Conceptos Generales

Componentes y Canales

Un sistema consta de un número de instancias de componentes que se ejecutan en uno o más procesadores lógicos, comunicándose a través de conectores. Un componente es un tipo abstracto que describe las propiedades de sus instancias. Los conectores actúan como los componentes coordinadores del modelo IWIM y su función es coordinar las actividades de los demás componentes. La comunicación entre instancias de componentes toma lugar exclusivamente a través de los canales constituyentes de los conectores. Los conectores habilitan la comunicación entre instancias de componentes, refuerzan los patrones de coordinación exógenos, tienen semántica bien definida independiente de los componentes que coordinan, y son construidos como composición de conectores más simples. Un canal es el único medio primitivo de comunicación, y es considerado un conector atómico, permite comunicación anónima, punto a punto y de una via. Las entidades activas de una instancia de componente se comunican con las entidades del exterior solamente a través de un conjunto de operaciones de entrada/salida que efectúan sobre el conjunto de extremos de canales conectados a él. Cuando un extremo de canal es desconectado de una instancia de componente y conectado a otra, cambia dinámicamente la topología de conexiones del sistema.

*P**ew*** asume la disponibilidad de un conjunto de tipos de canales bien definidos, con un comportamiento dado por el número de extremos fuente y destino, el esquema de orden, el tamaño del buffer, un filtro asociado, etc. Por ejemplo, un canal síncrono tipo *SyncDrain* es un canal que tiene dos extremos fuente y se utiliza para sincronizar la actividad en los nodos conectados a sus extremos, sin importar el valor que se escriba en el canal. Todos los canales deben implementar un conjunto básico de operaciones con la misma semántica. Están definidas operaciones de entrada/salida características (*write, read, take*), operaciones de consulta (*wait*), y operaciones topológicas (*connect*, *disconnect*, *forget*). La única operación primitiva sobre canales que puede ser ejecutada directamente por una instancia de componente es la operación *create.* Cuando se crea un canal se le asocia un patrón que regula las operaciones de entrada/salida sobre el canal. El resto de las operaciones deben ser utilizadas internamente por *P**ew***.

Conectores

Un conector es un conjunto de extremos de canal y los canales que los conectan organizados en un grafo que tiene las siguientes características:

- Cada extremo de canal x solo puede estar conectado a un nodo N en cada momento
- Cada nodo puede tener varios extremos de canal incidentes en él
- Cada canal interviniente en el conector es representado por un arco en el grafo

Dependiendo de los extremos de canal incidentes, cada nodo en un conector puede ser de uno de los tres tipos siguientes:

- *Nodo fuente*, cuando sólo inciden en él extremos fuente de canales.
- *Nodo destino*, cuando sólo inciden en él extremos destino de canales.
- *Nodo mixto*, cuando se da cualquier combinación de extremos de canales incidentes en él.

Las operaciones sobre nodos se organizan según lo muestra la figura 1. Las operaciones *connect* y *disconnect* solo son aplicables con éxito a un nodo N que sea nodo fuente o nodo destino. En caso de que haya mas de un extremo de canal incidentes en N de los que se pueda tomar un valor, las operaciones *read* y *take* eligen de manera no determinista, pero la operación sólo tiene éxito si existe un item de dato disponible en algún extremo de canal. Por su parte la operación *write* siempre tiene éxito sobre un nodo destino. El parámetro *conds* en la operación *wait* es una expresión booleana que combina condiciones primitivas predefinidas sobre nodos. Una condición primitiva sobre un nodo N puede ser aplicable a todos los extremos de canales incidentes en N (*Allconnected(N)*, *Allempty(N)*, etc.) o puede exigir sólo la verificación para algún extremo de canal (*connected(N)*, *full(N)*, etc.). Las operaciones de composición de nodos se utilizan para la construcción de conectores y la reconfiguración topológica de los canales internos a un conector. En particular el efecto de la operación *join* es una mezcla destructiva de los nodos originales en un nuevo nodo.

Operaciones topológicas y de consulta

forget(N): produce la aplicación de la operación _forget a cada extremo de canal coincidente en el nodo N, produciendo la liberación de las referencias a extremos de canal por parte del componente que ejecutó la operación

connect([t,] N): conecta el extremo de cada canal coincidente en N a la instancia de componente que contiene la entidad activa que ejecuta la operación.

disconnect([t,] N): desconecta el extremo de canal coincidente en N de la instancia de componente que contiene la entidad activa que ejecuta la operación.

wait([t,] conds): suspende la entidad activa que ejecuta esta operación hasta que la condición especificada en el parámetro ***conds*** se haga verdadera.

Operaciones de entrada/salida

read([t,] N[, v[, pat]]): suspende la entidad activa que ejecuta esta operación en espera del ingreso de un valor que encaje con el patrón indicado por ***pat***, en algún extremo de canal incidente en N. Es una operación no destructiva, después de la lectura el valor se conserva en el canal.

take([t,] N[, v[, pat]]): suspende la entidad activa que ejecuta esta operación en espera del ingreso de un valor que encaje con el patrón indicado por ***pat***, en algún extremo de canal incidente en N. Es la versión destructiva de la operación de lectura. El valor en el extremo del canal es efectivamente consumido.

write([t,] N, v): suspende la entidad activa que ejecuta la operación hasta que se dan las condiciones para realizar la escritura del valor ***v*** en cada extremo de canal conectado a N, en forma atómica.

Operaciones de composición y abstracción

Join (N1, N2)*:* sólo tiene éxito si la instancia de componente que efectuó la operación tiene una conexión a alguno de los nodos indicados. El resultado de la operación es una mezcla destructiva y la generación de un nuevo nodo N que reunirá los extremos de canal coincidentes previamente en N1 y N2. La instancia de componente permanece conectada al nuevo nodo N sólo si estaba conectada a ambos nodos N1 y N2, y N no es un nodo mixto.

Split (N[, quoin]): genera un nuevo nodo N´ y divide los extremos de canal incidentes en N entre N y N´. La separación de los extremos de canales entre los nodos se realiza según lo especificado por el parámetro quoin.

Hide(N): es un mecanismo de abstracción. Se oculta el nodo N de manera que la topología de los canales coincidentes en N no pueda ser modificada. Como resultado de la aplicación de la operación sobre el nodo N, N no puede ser utilizado instancia de componente alguna.

Figura 1: Operaciones sobre nodos

Pew espera que cada tipo de canal pueda proveer una implementación razonable de las operaciones primitivas. Sólo los nodos fuente y destino pueden estar conectados a componentes. Los nodos mixtos tienen como función hacer fluir los datos desde los nodos fuente hacia los nodos destino, a través de los canales. Este comportamiento se logra a partir de la composición de canales en conectores y la semántica operacional de las operaciones de entrada/salida. En algunos casos, sin embargo, dependiendo del tipo de canal, algunos ítems pueden perderse.

2.2 Composición de Canales

Ejemplo 1: Regulador de Lectura

Figura 2: ejemplo de composición de canales y conectores

El ejemplo muestra como se puede construir un conector simple a partir de canales y operaciones sobre canales. La configuración mostrada en la figura 2.a representa un conector *Regulador de Lectura* (Take-Cue Regulator) que es una de las formas mas básicas de coordinación exógena. Asumimos que los canales <a, b> y <c, d> son de tipo FIFO, y que el canal <e, f> es de tipo Sync. El número de ítems de datos que fluyen del canal <a, b> al canal <c, d> es el mismo que el número de operaciones *take* que tienen éxito en el extremo destino f del canal <e, f>. En esta situación la entidad activa de la instancia de componente que está conectada al nodo N por el extremo de canal f (Node(f)=N) puede contar y regular el flujo de datos entre los canales <a, b> y <c, d> por el número de operaciones *take* efectuadas sobre N. Las instancias de componentes conectadas a Node(a) y Node(d) son ajenas al hecho de que su comunicación este siendo regulada o medida, y aún de que tal comunicación existe. En la figura 2.b se muestra la misma configuración, ahora encapsulada por efecto de la aplicación de la operación *hide* sobre el nodo común a los 3 canales. La abstracción lograda permite que la topología del conector sea inmutable, y pueda ser utilizado como un "componente conector" que provee sólo los puntos de conexión de sus límites. La figura 3 muestra el código *Pew* para generar el conector de la figura 3.b. El valor de retorno de una llamada a esta función contiene las identidades de los nodos primarios de entrada y salida, y el nodo que actúa como regulador.

```
ReguladorL
        <a, b> = create (FIFO)
        <c, d> = create (FIFO)
        <e, f> = create (Sync)
        connect (b)
        connect (c)
        join (Node(b), Node(e))
        join (Node(b), Node(c))
        hide (Node(e))
        return <Node(a), Node(d), Node(f)>
```

Figura 3: código *Pew* para el conector Regulador de Lectura

Ejemplo 2: Conector Inhibidor

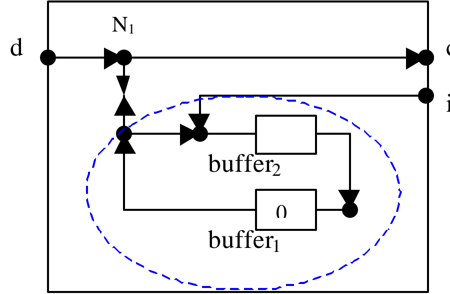

Figura 4: conector Inhibidor

El conector de la figura 4 tiene la función de permitir que los valores escritos en el extremo fuente *d* del canal fluyan libremente hacia el extremo destino *c* del otro canal, hasta que algún valor sea escrito en el extremo de canal *i*, lo que produce el corte del flujo de datos de *d* hacia *c*. Este comportamiento se logra en parte por la existencia del canal SyncDrain que sincroniza la escritura en *d* con el valor inicial en el $buffer_1$ del canal del conector de serie (señalado con un círculo en la figura), y en parte por la escritura en *i* que llena el $buffer_2$ y el valor en $buffer_1$ ya no puede circular por el conector de serie. Al bloquearse el valor en $buffer_1$, también se bloquea la operación de escritura en el nodo mixto N_1.

3 Un cálculo de procesos basado en *Pew*

Representamos un sistema basado en componentes en el que intervienen los componentes C1, C2, ..., Cn con la expresión Π: C1 || C2 || ... || Cn

Asumimos una configuración inicial de conexiones a canales que definen la topología inicial de conexiones del sistema. Consideramos un conjunto *C_ID* de identificaciones de canales, con elemento característico *c*, y c_s, c_t como sus extremos fuente y destino respectivamente. Cada nodo es especificado como un conjunto de los extremos de canal que inciden en él. Representamos con N_s a un nodo fuente y N_t a un nodo destino. Los nodos mixtos no se consideran por la imposibilidad de una conexión entre ellos y un componente externo. En este trabajo consideramos un conjunto básico de tipos de canales organizados en 2 grupos: *TSync* representa los tipos de canales síncronos y *TAsync* representa los tipos asíncronos ilimitados. Para representar la configuración inicial seguimos lo propuesto en [5]. La interfaz observable de un componente es una tupla de la forma $\langle C, r, I(z), \phi(r), \psi(r)\rangle$ en la cual $I(z)$ representa el invariante de bloqueo que por medio de aserciones especifica el posible comportamiento de bloqueo del componente *C*, y *z* puede ser el extremo destino de cualquier canal perteneciente al conjunto *C_ID*. *C* denota el componente que se desea especificar, *r* representa los nodos que intervienen en la configuración inicial. Inicialmente los nodos considerados están compuestos por un único extremo de canal de un tipo especificado en $TSync \cup TAsync$. La incorporación de nuevos extremos de canal a los nodos se logra dinámicamente a partir de la aplicación de la operación de composición. Las aserciones $\phi(r)$ y $\psi(r)$ denotan la precondición –contenidos iniciales de los buffers de los canales esternos- y post condición –secuencia de valores recibidos y entregados por los canales externos- del componente. Una aserción ϕ es definida como sigue:

$$\phi ::= p(e_1, ..., e_n) \mid \neg\phi \mid \phi \wedge \phi \mid \exists x(\phi)$$

dónde p denota un predicado múltiple, aplicado sobre las expresiones e_i, y x representa una variable. Una expresión e es definida de la siguiente forma:

$$e ::= c_s \mid c_k \mid x \mid e\downarrow \mid e\downarrow_n \mid f(e_1, ..., e_n)$$

f representa un operador, y las expresiones $e\downarrow$ y $e\downarrow_n$ denotan la secuencia de valores asociados al extremo de canal *e*.
Como un ejemplo simple la interfaz siguiente denota un componente nombrado *C*, inicialmente conectado al extremo fuente del canal *d* y al extremo destino del canal *e*. El componente repetidamente escribe un valor en el canal *d* y luego lee un valor del canal *e*. El invariante de bloqueo establece que el número de valores leídos del canal *e* es estrictamente menor que el número de valores escritos en el canal *d*. Aquí la operación $|w\downarrow|$ da la longitud de la secuencia *w* y ε denota la secuencia vacia.

$$\langle C, \{d_s, e_t\}, |d_s\downarrow| < |e_t\downarrow| \wedge z=\{e\} \wedge \forall z_1 \notin r\ (z_1\downarrow = \varepsilon),\ (d_s\downarrow = \varepsilon) \wedge (e_t\downarrow = \varepsilon),\ \psi(r) \rangle$$

Para especificar el comportamiento interactivo de los componentes utilizamos un álgebra de procesos basado en las primitivas de comunicación y composición de ***Pew*** y los operadores estándar de prefijo, composición alternativa y composición paralela. La sintaxis del álgebra es definida de la siguiente manera:

$$\begin{aligned} P ::= &\ 0 \mid A.P \mid P+P \mid P \parallel P \\ A ::= &\ write(N_s, v) \mid take(N_t, v) \mid read(N_t, v) \mid connect(N) \mid disconnect(N) \\ &\mid forget(N) \mid \tau \mid join(N1, N2) \end{aligned}$$

dónde 0 denota el proceso vacío, y τ representa cualquier acción interna que deba ejecutar el componente. El agente τ P evolucionará a P sin interactuar con su entorno. Los demás prefijos son las primitivas de ***Pew*** que nos interesa considerar.

Las reglas de transición que modelan la semántica operacional del álgebra presentada se muestran en la figura 5. La regla 1 describe el comportamiento de las transiciones silenciosas y la regla 2 describe las transiciones basadas en operaciones topológicas. Las reglas 3, 4 y 5 describen el comportamiento de las primitivas de entrada/salida sobre canales de tipo asíncrono y síncrono que tengan al menos un extremo fuente en el caso de la operación *write* - los canales de tipo *AsyncSpout* y *SyncSpout* poseen dos extremos destino- y al menos un extremo destino en el caso de las operaciones *read* y *take* - los canales de tipo *AsyncDrain* y *SyncDrain* poseen dos extremos fuente-. La expresión B(c) se refiere al contenido del buffer asociado al canal c tal que $B(c) = \langle B_k, B_{k-1}, ..., B_2, B_1 \rangle$, B_1 representa el primer elemento ingresado al buffer, y $\circ$ es el operador de concatenación. En la regla 3 se especifica que la operación de concatenación se produce sobre todos los canales componentes del nodo, pues a diferencia de las operaciones *read* y *take*, que eligen de manera no determinista, una operación de escritura replica su valor y escribe atómicamente una copia del mismo en cada extremo de canal incidente en el nodo. Las reglas 6 y 7 son las reglas estándar para la composición alternativa y paralela de un sistema cerrado con respecto a la relación de equivalencia estructural. La regla 8 modela de manera general la sincronización entre dos procesos que ejecutan acciones complementarias de comunicación. Dependiendo del tipo específico de canal síncrono: *Sync*, *SyncDrain* y *SyncSpout*, las operaciones α y β se refieren a *write-take*, *write-write* y *take-take* respectivamente. En particular el tipo de canal syncdrain se utiliza sólo para sincronización, sin importar que suceda con los valores escritos en el canal.

$$1\quad \left[\tau\cdot P \xrightarrow{\tau} P\right] \qquad 2\quad [a\cdot P \xrightarrow{a} P] \quad a\in\{connect(N), disconnect(N), forget(N)\}$$

$$3\quad \left[\frac{(\forall c\cdot(c\alpha\ N_s \wedge type(c)\in TAsync), B(c))}{write(N_s,v)\cdot P} \xrightarrow[\forall c\alpha\ N_s\ (type(c)\notin\{AsyncSpout, SyncSpout\})]{write(N_s,v)} \frac{P}{\forall c\alpha\ N_s\cdot type(c)\in TAsync(B(c)\circ\langle v\rangle)}\right]$$

$$4\quad \forall c\alpha\ N_k\ (type(c)\notin\{AsyncDrain, SyncDrain\}) \quad \left[take(N_k,v)\cdot P \xrightarrow{take(N_k,d)} P\{d/v\} \ \wedge (B(c)-\langle d\rangle)\vee type(c)\in TSync \quad para\cdot algun\cdot c\alpha\ N_k\right]$$

$$5\quad \forall c\alpha\ N_k\ (type(c)\notin\{ASyncDrain, SyncDrain\}) \quad \left[read(N_k,v)\cdot P \xrightarrow{read(N_k,d)} P\{d/v\} \quad para\cdot algun\ \ c\alpha\ N_k\right]$$

$$6\quad \left[\frac{P\xrightarrow{\alpha}P'}{P+Q\xrightarrow{\alpha}P'}\right] \qquad 7\quad \left[\frac{P\xrightarrow{\alpha}P'}{P\|Q\xrightarrow{\alpha}P'\|Q}\right]$$

$$8\quad (\exists c\cdot c\to N\wedge\ type(c)\in TSync)\quad \left[\frac{P\xrightarrow{\alpha(N)}P' \quad Q\xrightarrow{\beta(N)}Q'}{P\|Q\xrightarrow{\gamma(\alpha(N),\beta(N))}P'\|Q'}\right]$$
$$(\alpha,\beta)\in\{(write,take),(write,write),(take,take)\}$$

$$9\quad connected(N1)\wedge connected(N2)\quad \left[join(N1,N2)\cdot P\xrightarrow{join(N1,N2)}P\{N1/N1\circ N2\} \quad \wedge connectedAll(N1)\right]$$

$$10\quad connected(N1)\wedge\neg connected(N2)\quad \left[join(N1,N2)\cdot P\xrightarrow{join(N1,N2)}P\{N1/N1\circ N2\} \quad \wedge disconnectedAll(N1)\right]$$

Figura 5: Reglas de Transición

Las reglas 9 y 10 modelan el join de dos nodos, que genera un nuevo nodo con la unión de todos los canales de los nodos originales. En la regla 9 se considera el caso en que el componente tiene conexión a los dos nodos, y el join preserva las conexiones. Cuando no existe conexión con alguno de los nodos, como resultado de la operación join el componente pierde las conexiones originales (regla 10).

4 Comportamiento Interactivo de Componentes en *Pew*

En esta sección describiremos como *Pew* puede ser efectivamente utilizado para especificar el comportamiento interactivo de componentes. En *Pew* los componentes se conectan a nodos fuente y destino, y un nodo representa el conjunto de extremos de canal que confluyen en él. Por razones de simplicidad en los casos en que el nodo consta de solo un extremo de canal lo identificamos directamente con el nombre del extremo del canal; sólo nombramos al nodo cuando consta de más de un extremo de canal como resultado, por ejemplo, de la aplicación de una operación *join*. Es decir identificamos con c_t al nodo destino cuyo único extremo de canal incidente es c_t, y con $N_{t-c,d}$ al nodo destino en el que inciden los extremos de canal c_t y d_t. Las precondiciones son expresadas en términos de las condiciones primitivas aplicables sobre canales y nodos definidas por *Pew*. No consideramos las postcondiciones dado que sólo nos interesa representar la topología inicial del sistema.

Como primer ejemplo consideramos un escenario en el que intervienen un componente autor, un componente editor, y un componente director. El autor es responsable de la escritura del documento y la realización al mismo de cambios significativos. Al editor le corresponde chequear la validez del documento. El director puede aprobar el documento recibido desde el editor ó

rechazarlo, lo que significa que será devuelto al editor, quien debe solicitar los cambios al autor. No hay comunicación directa entre autor y director.

Consideramos dos canales para la comunicación autor-editor, y dos canales para la comunicación editor-director. El director además dispone de una conexión a un tercer canal por el cual enviará los documentos aprobados. Elegimos canales de tipo Sync para la comunicación desde el editor al autor y al director para representar comportamiento adicional respecto a la "entrega en mano" al autor de las modificaciones requeridas sobre el documento, y al director del documento para su evaluación.

Especificamos el sistema como sigue:

Π: ⟨ *Autor*, {aut_s,, $edit_t$}, *empty* (aut_s) **Ù** *empty* ($edit_t$) ⟩
|| ⟨ *Editor*, {dir_t,, $edit_s$, aut_t,, ed_s }, *empty* (dir_t) **Ù** *empty* (ed_s) ∧ *empty* (aut_t) **Ù** *empty* ($edit_s$) ⟩
|| ⟨ *Director*, {ed_t,, dir_s , $accept_s$ }, *empty* (ed_t) **Ù** *empty*(dir_s) ⟩

C_ID = {aut: FIFO, edit: Sync, dir: FIFO, ed: Sync, accept: FIFO}

A continuación describimos el comportamiento de cada agente:

AUTOR (aut_s, $edit_t$) = GENERAR (doc) . write(aut_s, doc) . ESPERAR (aut_s, $edit_t$, doc)
ESPERAR (aut_s, $edit_t$, doc) = 0
+
take ($edit_t$, doc) Corregir(doc) . write(aut_s, doc) . ESPERAR(aut_s, $edit_t$, doc))

Al chequear la validez del documento el editor puede llevar a cabo pequeñas correcciones y enviarlo al director; pero si es necesario realizar mayores cambios, debe regresar el documento al autor.

EDITOR(aut_t, $edit_s$, ed_s, dir_t, doc) = (take (aut_t, doc) . VERIFICAR(doc) . (write(ed_s, doc)
+
write($edit_s$, doc1))
+
take (dir_t, doc) . write($edit_s$, doc))
EDITOR(aut_t, $edit_s$, dir_t, ed_s)

DIRECTOR(dir_s, ed_t, $accept_s$, doc) = take (ed_t, doc) . (write($accept_s$, doc)
+
write(dir_s, doc)) .
DIRECTOR(dir_s, ed_t, $accept_s$, doc)

Como segundo ejemplo consideramos la siguiente situación: se desea utilizar agentes para buscar información específica, por ejemplo precios de vuelos por internet. Los agentes consultan diferentes fuentes de información. Cada fuente de información tiene un punto de entrada dónde se puede establecer los requisitos. Un agente conoce la ubicación de la fuente de información. La identidad de los puntos de requerimiento de las fuentes puede estar en una lista, o ser pasada como datos a través de canales, etc. El componente cliente interactúa con el agente solicitando información y recibiendo resultados. El agente por alguna acción interna recupera la identidad del canal por medio del cual debe comunicarse con la primera fuente de información para obtener resultados. El proceso continúa con el agente recorriendo las distintas fuentes de información que conoce. Cada fuente de información al ser consultada devuelve sus resultados.
La especificación del sistema para un cliente, un agente y dos fuentes de información es:

Π: ⟨ *Cliente*, {e_s,, s_t}, *empty* (e_s) **Ù** *empty* (s_k) ⟩ || ⟨ *Agente*, {e_t,, s_s}, *empty* (e_k) **Ù** *empty* (s_s) ⟩ ||
⟨ *Fuente1*, {$f1_t$,, $i1_s$ }, *empty* ($f1_k$) **Ù** *empty*($i1_s$) ⟩ || ⟨ *Fuente2*, {$f2_t$,, $i2_s$}, *empty* ($f2_k$) **Ù** *empty*($i2_s$) ⟩

En el conjunto C_ID de los identificadores de canales, indicamos el tipo de cada canal. Entonces

C_ID = {e: FIFO , s:FIFO , f1:Sync , f2:Sync, i1:Sync , i2:Sync }

Las conexiones iniciales son las que permiten la comunicación entre el cliente y el agente, y las propias de cada fuente de información. Las conexiones que permiten la comunicación entre el agente y las distintas fuentes de información son dinámicas y serán realizadas por parte del agente a medida que sea necesario. Se debe diferenciar los canales de entrada y salida de datos, identificados por su extremo destino y fuente respectivamente, debido a que en el cálculo de canales se consideran canales de un sentido.

El comportamiento del componente cliente puede ser descripto por el protocolo siguiente:

```
CLIENTE (salida, entrada) = write(salida, qry) . ESPERAR(entrada)  (0
                                                                    +
                                                                    CLIENTE(salida, entrada))
ESPERAR(entrada) = PROCESAR  ||  take(entrada, ans) . VERIFICAR_ANS (write(salida, akn)
                                                                     +
                                                                     ESPERAR(entrada))
```

El cliente puede seguir realizando acciones internas mientras espera los resultados solicitados. Una vez recibidas las respuestas por parte del agente, puede decidir hacer una nueva consulta o terminar el proceso. El cliente permanece en espera de mas respuestas a su consulta hasta que recibe una marca de fin.

El componente agente espera el ingreso de algún pedido por su canal de entrada y se conecta a las fuentes de información, una por vez, para obtener la información que le ha sido solicitada. Al terminar con cada fuente de información libera sus puntos de requerimientos de manera que, por ejemplo, puedan ser utilizados por otro agente (BUSCAR_INFO).

AGENTE(entrada, salida) = take(salida, qry) . BUSCAR_INFO(entrada, qry) . take(salida, akn) . AGENTE(entrada, salida)

BUSCAR_INFO(entrada, qry) = ELEGIR(f_s, i_k) . (write(salida, ‘FIN’)
+
connect(f_s) . write(f_s, qry) . forget(f_s). connect(i_t) . take(i_t, ans) . forget(i_t) . write(entrada, ans) . BUSCAR_INFO(entrada, qry))

El comportamiento de una fuente de información consiste sólo en esperar consultas y entregar una respuesta para cada consulta recibida.

FUENTE(f_t, i_s) = take(f_t, qry) . BUSCAR(qry, ans) . write(i_s, ans) . FUENTE(f_t, i_s)

Consideramos que dos componentes son compatibles si para cada acción posible ofrecida por uno de ellos existe una respuesta del otro y viceversa. Intuitivamente la noción de compatibilidad entre componentes encierra la noción de ausencia de deadlock en todas las ejecuciones alternativas de los procesos. Sobre el ejemplo podemos verificar, por ejemplo, que el proceso CLIENTE y el proceso AGENTE son compatibles. Por una primera transición aplicada sobre cada proceso obtenemos los nuevos procesos

CLIENTE1= ESPERAR(entrada) CLIENTE(salida, entrada)

AGENTE1= BUSCAR_INFO(entrada, qry). take(salida, akn) AGENTE(entrada, salida),

para los que debemos evaluar seguidamente su compatibilidad. Mostramos la aplicación de ésta noción.

1 $CLIENTE \xrightarrow{write(e_s,qry)} CLIENTE1$

2 $AGENTE \xrightarrow{take(e_t,qry)} AGENTE1$

3 $AGENTE1 \Rightarrow^* \xrightarrow{write(s_s,ans)} AGENTE2 \quad (BUSCAR_INFO(entrada, qry))$

4 $CLIENTE1 \xrightarrow{take(s_t,ans)} \xrightarrow{t} CLIENTE2 \quad (ESPERAR(entrada))$

5 $AGENTE2 \Rightarrow^* \xrightarrow{write(s_s,ans)} AGENTE2$

6 $CLIENTE2 \xrightarrow{take(s_t,ans)} \xrightarrow{t} CLIENTE2$

7 $AGENTE2 \xrightarrow{t} \xrightarrow{take(s_t,'fin')} AGENTE3 \quad (take(salida, akn) \cdot AGENTE(entrada, salida))$

8 $CLIENTE2 \xrightarrow{take(s_t,ans)} \xrightarrow{t} CLIENTE3 \quad (write(salida, akn))$

9 $CLIENTE3 \xrightarrow{write(e_s,akn)} CLIENTE$

10 $AGENTE3 \xrightarrow{take(e_t,akn)} AGENTE$

En la transición $\Rightarrow^*$ están representadas las acciones que compatibilizan al AGENTE con la FUENTE de información: realizar las conexiones, solicitar la información, recibir la respuesta. Observemos que el tipo de canal elegido en cada caso afecta el comportamiento de los componentes. Con una misma especificación y solo variando el tipo de algún canal se puede llegar a condiciones adversas, contra condiciones favorables previas. En particular en nuestro ejemplo, podemos analizar que sucede con el canal *s* por el que el agente comunica cada respuesta al cliente. Con el tipo de canal FIFO especificado sería posible cambiar el orden de algunas transiciones sin generar inconvenientes. Por ejemplo se podría realizar las transiciones 3, 5 y 7 de manera continuada, antes de que se produzca la transición 4, dado que el cliente puede realizar tres aplicaciones de la operación *take* seguidas sin problemas. Sin embargo si el tipo de canal especificado fuera Sync, dicho orden entre las transiciones nos conduciría a una posible situación de bloqueo. En las últimas dos transiciones, el componente CLIENTE elige dejar abierta la comunicación, y el AGENTE reacciona de manera acorde.

6. Conclusiones y trabajo futuro

Pew es un modelo de coordinación exógena, en el cual la comunicación entre procesos puede ocurrir solamente a través de canales, que se componen para generar conectores que actúan como coordinadores. En este trabajo se presenta la fase inicial del análisis de la aplicación de ***Pew*** en la interoperabilidad de componentes. Con algunos ejemplos se muestra la potencia expresiva del lenguaje y se da su significado formal por medio del álgebra de procesos presentada. Además se da una noción intuitiva de compatibilidad de componentes.

Nuestro trabajo futuro estará destinado a:

- Extender el álgebra de procesos para capturar todas las posibilidades ofrecidas por ***Pew*** y definir formalmente una relación de compatibilidad que sea decidible. Se tiende a lograr demostrar cuando el comportamiento de dos componentes especificados en ***Pew*** es compatible y su interoperación es posible.
- Analizar la influencia de los distintos tipos de conectores y canales sobre la relación de compatibilidad y la especificación de protocolos de interacción; y la influencia del cambio de conectores sobre una relación ya establecida entre componentes.

La intención final de nuestra investigación es la aplicación práctica de los resultados por medio de herramientas de chequeo automático.

7. Referencias

1. F. Arbab, *A Channel-based Coordination Model for Component Composition*, Reporte de CWI, Centrum voor Wiskunde en Informatica, 16th European Conference on Object-Oriented Programming, 2002.
2. F. Arbab, *What do you mean, coordination?*, Bulletin of the Dutch Association for theoretical Computer Science, NVTI, pages 11-22, 1998.
3. F. Arbad, *The IWIM Model for Coordination of Concurrent Activities,* Proceedings First International Conference on Coordination Models, Languages and Applications (Coordination'96), pp 34-56.
4. F. Arbab, F.S. de Boer, M.M. Bonsangue, J.V. Guillen Scholten, *A channel-based coordination model for components*, Reporte de CWI, Centrum voor Wiskunde en Informatica, 16th European Conference on Object-Oriented Programming, 2001.
5. F. Arbab, F.S. de Boer, M.M. Bonsangue, *A logical interfaz description language for components*, Proceedings of Coordination 2000, Vol 1906 of Lecture Notes in Computer Science, pages 249-266, 2000.
6. F. Arbab, M.M. Bonsangue, F.S. de Boer, *A coordination Language for Mobile Components*, Proceedings of SAC 2000, ACM Press, pages 166-173, 2000.
7. H. Balen, *Distributed Object Architectures with CORBA*, Managing Object Technologies Series, Cambridge University Press 2000
8. K. Bergner, R. Grosu, A. Rausch, A. Schmidt, P. Scholz, M. Broy*Focusing on Mobility,* http://www4.informatik.tu-muenchen.de
9. J.A. Bergstra, P. Klint, *The TOOLBUS –a component interconnection architecture*
10. A. Bracciali, A. Brogi, F. Turini, *Interaction patterns*, V Jornadas Iberoamericanas de Ingeniería de Software, 2001.
11. A. Brogi, E. Pimentel, A.M. Roldán, *Interoperabilidad de Componentes Software en Linda*, IDEAS 2002
12. A. Brogi, J.M. Jacquet, *On the expressiveness of coordination models*, Coordination Languages and Models:3rd. International Conference, Vol. 1594 of Lecture Notes, pages 134-149,1999.
13. C. Canal, *Un lenguaje para la especificación y validación de arquitecturas de software*, Ph.D. thesis, Depto Lenguajes y Ciencias de la Computación, Universidad de Málaga, 2001.
14. C. Canal, L. Fuentes, J.M. Troya, and A. Vallecillo, *Extending CORBA interfaces with pi-calculus for protocol compatibility,* Proceedings of the Technology of Object-Oriented Languages and Systems - TOOLS Europe 2000, IEEE Press 2000, pp. 208-225
15. W. Emmerich, *Engineering Distributed Objects*, London University, John Wiley & Sons, Ltd. 2000.
16. D. Gelernter and N. Carriero, *Coordination Languages and their significance,* Communications of the ACM, No. 35, pp. 97-107. 1992.
17. R. Grosu, K. Stoelen, *A model for mobile point-to-point data-flow networks without channel sharing.* Lecture Notes in Computer Science, 1996.
18. G.T. Leavens, M. Staraman, *Foundations of component based systems*, Cambridge University Press, 2000.
19. T. D. Maijler and O. Nierstrasz, *Beyond Objects: Components,* En Cooperative Information Systems: Current Trends and Directions, M.P. Papazoglou, G. Schlageter (Ed), Academic Press, Nov. 1997
20. G. Papadopoulos and F. Arbab, *Dynamic Reconfiguration in Coordination Languages*, Advances in Computers 46, Marvin V. Zelkowitz, Academic Press, 1998, pp. 329-400
21. W. Rubin, M. Brain, *Understanding DCOM*, Prentice Hall Series on Microsoft Technologies. 1999.