Un lenguaje de programación orientado a objetos activos

Claudia Pons Gabriel Baum

LIFIA, Universidad Nacional de La Plata, Argentina.
cpons@ada.info.unlp.edu.ar

Keywords: programación orientada a objetos, sistemas reactivos, lenguajes de programación, semántica formal.

Resumen

En este artículo definimos las bases para un lenguaje de programación orientado a objetos que permite expresar comportamiento activo en forma unificada. El comportamiento activo queda embebido dentro del comportamiento general de los objetos del sistema. Se respeta el encapsulamiento de los objetos y se aprovecha la jerarquía de clases para determinar el alcance de las reglas.

El comportamiento del sistema se va definiendo a lo largo de la ejecución, ya que cada objeto al desplegar su comportamiento (ejecutar un método) puede agregar y/o borrar mensajes activos del sistema. Es decir, cada objeto tiene la posibilidad de "legislar" (dictando y/o anulando reglas.) sobre el sistema. El lenguaje permite expresar distintas formas de comportamiento activo: general, particular, persistente y efímero.

1. Introducción

La crisis del software que se desencadenó en las últimas décadas obligó a los investigadores a buscar nuevas metodologías de trabajo para aumentar la productividad de los desarrolladores de software.

Se trabajó sobre el principio de que una buena metodologías de desarrollo de soft debe proveer como mínimo: • *modularidad*: dominio de la complejidad de los grandes sistemas mediante su adecuado particionamiento. • *reuso*: una pieza de soft definida para un contexto puede ser reutilizada en otro contexto diferente. • *mantenibilidad*: cambios en el sistema pueden ser localizados e implementados en forma fácil y económica.

En este contexto surgió el paradigma de orientación a objetos que permite modelar sistemas en términos de objetos, lo cual favorece la producción de programas mas legibles, reusables y mantenibles. El nuevo paradigma logró una rápida aplicación comercial mediante lenguajes como Smalltalk, Eiffel y C++.

Paralelamente a los sistemas orientados a objetos también cobraron importancia los sistemas basados en conocimiento y sistemas expertos. Emergió un nuevo estilo de programación - basada en reglas- que encuentra su expresión en lenguajes tales como Prolog y OPS5.

Hoy en día, tanto programación orientada a objetos como programación basada en reglas son paradigmas populares e igualmente aceptados. No es posible determinar cual de los dos paradigmas es mas adecuado, ya que esto depende de la tarea que se intente codificar. Sin embargo existen casos donde sería deseable poder usar ambos paradigmas en forma combinada, ya que ninguno de ellos aisladamente resulta de óptima aplicabilidad.

Básicamente existen dos formas de integrar sistemas orientados a objetos con sistemas basados en reglas:

- una posibilidad es permitir al programador incluir llamadas a un programa orientado a objetos dentro de un sistema basado en reglas. Sin embargo, con esta solución se pierden las principales ventajas de la programación orientada a objetos, ya que los lenguajes basados en reglas existentes en general no proveen encapsulamiento y modularidad.
- la otra alternativa es extender un lenguaje orientado a objetos para soportar reglas. Esto permite conservar todas las ventajas de la programación orientada a objetos y además ganar mecanismos declarativos para especificar restricciones y comportamiento guiado por eventos (es decir, comportamiento activo), tanto locales a un objeto como globales a un grupo de objetos.

Existen varias propuestas para integrar conceptos activos con conceptos orientados a objetos, por ejemplo: el sistema de reglas CERS-rules system [Miranker93] que soporta reglas de producción OPS5 dentro de un ambiente C++; el sistema TANGUY [Eich93] propone una extensión del lenguaje C++ para soportar reglas de producción reactivas (que reaccionan a eventos que ocurren durante la ejecución de un sistema, tales como envío de mensajes o cambios en los datos de objetos particulares). El sistema Opus [Atkinson87] y su extensión el sistema NéOpus [Pachet95] permiten embeber reglas de producción dentro del lenguaje Smalltalk-80 La integración de los dos paradigmas también ha recibido gran atención en el área de las bases de datos [Gatziu91, Gehani91, Medeiros91, Kappel94, Bertino94]

2. Planteo del Problema

Las soluciones existentes permiten integrar los dos paradigmas. Los resultados son sistemas híbridos consistentes de dos lenguajes diferentes, uno para los objetos y otro para las reglas. En estos sistemas los paradigmas conviven pero no se unifican. Por ejemplo, en Tanguy las clases se definen en C++, mientras que el comportamiento activo se define a través de reglas para las cuales se provee una sintaxis especial. Los diseñadores deben definir el comportamiento activo separadamente y no en forma integrada con las clases del sistema. En TriGS [Kappel94] las reglas activas y los eventos que hacen que esas reglas se disparen son parte de la jerarquía de clases de la aplicación. Esto conduce a una modelización donde un evento es un objeto de

primera clase. Sin embargo, conceptualmente un evento es un cambio de estado de un objeto y no un objeto en sí mismo. NéOpus elimina el "impedance mismatch" entre los lenguajes permitiendo expresar las reglas en términos de Smalltalk. Sin embargo, las reglas no están embebidas dentro del comportamiento de los objetos, sino que ellas constituyen en sí mismas un objeto distinguido dentro del sistema que gobiernan.

Nuestra propuesta tiene por objetivo unificar los paradigmas de manera que el comportamiento activo quede embebido dentro del comportamiento general de los objetos del sistema. Deseamos que los desarrolladores del sistema tengan un marco de trabajo integral, donde puedan expresar todo el comportamiento de los objetos (ya sea activo o pasivo) dentro del mismo lenguaje y al mismo nivel. Para esto definimos un prototipo de lenguaje unificado y damos su semántica formal. En nuestro lenguaje el comportamiento activo se modeliza a través de diferentes tipos de reglas (o mensajes activos) escritas en el mismo lenguaje. Estas reglas tienen la misma jerarquía que los mensajes comunes del sistema y forman parte del comportamiento de los objetos.

3. Taxonomía de Mensajes en los Sistemas Orientados a Objetos

En los sistemas orientados a objetos un grupo de objetos interactúa para resolver un problema. La única forma de interacción es el envío de mensajes. Cada clase de objeto se caracteriza por el conjunto de mensajes que sus objetos pueden aceptar. Dado que las clases son el único ámbito para la definición de mensajes, resulta claro que todo mensaje que circula por el sistema tiene un objeto receptor. Sin embargo no todo mensaje que circula por el sistema tiene un objeto emisor (existen mensajes cuyo emisor no es un objeto). Basándonos en su forma de emisión clasificaremos a los mensajes que circulan por el sistema en distintas categorías

3.1. Los mensajes en un Sistema Orientado a Objetos Clásico

En un sistema orientado a objetos clásico (como C++ o Smalltalk) los objetos sólo reaccionan al recibir un mensaje, por lo tanto un objeto sólo puede enviar un mensaje como respuesta (reacción) a otro mensaje. Existen además mensajes contenidos en el programa principal (Main). El emisor de estos mensajes no es ningún objeto del sistema. Por lo tanto, en estos sistemas distinguiremos dos categorías de mensajes (fig.3.a):

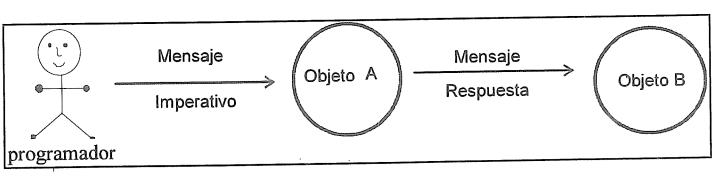

Fig.3.a Mensajes en un Sistema Orientado a Objetos.

Mensajes Imperativos: son los mensajes del programa principal, su emisor no es ningún objeto del sistema.

Mensajes Respuesta: su emisor es algún objeto del sistema, es decir son los mensajes que algún objeto envía como respuesta a otro mensaje.

3.2. Los mensajes en un Sistema Orientado a Objetos Activos

Los sistemas orientados a objetos tradicionales no son adecuados para representar comportamiento guiado por eventos; esto se debe a que los objetos tradicionales sólo reaccionan al recibir un mensaje y no pueden reaccionar automáticamente ante determinados cambios de estado del sistema. Es decir, los objetos tradicionales sólo responden a ordenes, no poseen capacidad para iniciar acciones por decisión propia. En consecuencia, para modelar comportamiento activo es necesario agregar una nueva categoría de mensajes:

Mensajes Activos: los mensajes activos no tienen un objeto emisor, se disparan automáticamente en respuesta a algún evento, es decir, cuando el sistema alcanza algún estado particular. La figura 3.b muestra la evolución de un evento dentro de un sistema activo.

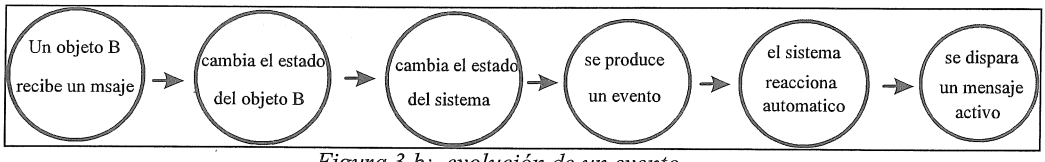

Figura 3.b: evolución de un evento

Utilizaremos la palabra **regla** para referirnos a estos mensajes activos. Una regla tiene la forma < **conditions** Exp **actions** Com>, siendo Exp una expresión que representa las condiciones que deben cumplirse para que ocurra el evento. Com es un comando que representa las acciones que se desencadenan en respuesta al evento.

Clasificación de los Mensajes Activos

Distinguimos diferentes categorías de reglas teniendo en cuenta su alcance con respecto a los objetos que involucra y con respecto a su tiempo de vida:

1 - <u>De acuerdo a los objetos que involucra</u>:

una regla puede referirse a un grupo particular de objetos o puede referirse a objetos genéricos

<u>Categoría 1.1</u>: ***Reglas Generales***

involucran un conjunto genérico de individuos. Por ejemplo, en un cierto Banco la política financiera es clausurar las cuentas cuando su saldo es inferior a un monto mínimo establecido. Esto puede representarse con la siguiente regla, válida para toda instancia c de la clase CuentaBancaria

Rule "Cuentas en Rojo":
conditions: la cuenta c está activa y el saldo de c es inferior al mínimo
actions: clausurar la cuenta c.

Categoría 1.2: ***Reglas Particulares***

involucran un conjunto constante de individuos. Por ejemplo: Mr.Perez es un cliente especial del Banco al cual se le efectúan préstamos automáticos cuando desciende el saldo de su cuenta, entonces para él definimos la siguiente regla:

Rule "Rojos de Mr.Perez":
conditions la cuenta de Mr.Perez está activa y su saldo es inferior al mínimo
actions depositar $10000 en la cuenta de Mr.Perez .

2 - De acuerdo a su tiempo de vida:

Categoría 2.1: ***Reglas Persistentes***

pueden ejecutarse múltiples veces (modelan eventos que se repiten). Por ejemplo las reglas "Cuentas en Rojo" y "Rojos de Mr.Perez".

Categoría 2.2: ***Reglas Transitorias***

se ejecutan sólo una vez y luego desaparecen (modelan un evento que ocurre sólo una vez). En la sección 6 describimos un ejemplo que incluye este tipo de reglas.

4. Dominios Semánticos para Objetos

En esta sección definimos dominios para interpretar objetos activos. Estos dominios brindarán la base para expresar la semántica de los programas que manipulan objetos.

- **Object = (Bvalue x Behavior x List_Rule)** , dominio para objetos.
- Bvalue, dominio primitivo para valores básicos (ej: bool, nat).
- Behavior=[Message → Method] dominio para comportamiento de objetos. Un comportamiento es una tabla que asocia cada mensaje con el método que el objeto computará para responder a ese mensaje.
- Message = String, dominio primitivo para nombres de mensajes. Este dominio está particionado en Observer_Message y Mutator_Message..
- Method = Exp + Com, dominio de métodos. Un método puede ser una expresión (método observador) o un comando (método mutador).
- List_Rule, dominio para listas de reglas.

Fig. 4.a -Dominio semántico para Objetos

El dominio donde los objetos toman significado es el conjunto de **registros** en forma similar a lo propuesto por Cardelli en [Cardelli85]. Más precisamente, modelamos un objeto activo como una terna, donde la primera componente es el estado interno, la segunda es el comportamiento

pasivo (el conjunto de todos los métodos que el objeto puede ejecutar (propios y heredados)) y la tercera son las reglas particulares que determinan su comportamiento activo individual. Consideramos que el conjunto de métodos de un objeto está particionado en dos subgrupos: *observadores* y *mutadores*. Un observador retorna un resultado sin alterar el estado interno de ningún objeto. Un mutador modifica uno o más estados de objetos. La Figura 4.*a* muestra la definición del dominio semántico Object. En un sistema orientado a objetos cada objeto posee un identificador único (generalmente denominado oid). Sin embargo, estos identificadores no son visibles y los programas manipulan los objetos mediante nombres simbólicos (o variables). En la figura 4.b definimos los dominios semánticos Env y Store. Un ambiente (environment) es una función que asocia nombres simbólicos con valores expresables. Un valor expresable puede ser un valor básico o un identificador de objeto. Estas funciones no son necesariamente inyectivas, lo cual permite que en un programa dos o mas nombres referencien al mismo objeto. Un almacenamiento(store) es una función que asocia oids con objetos. Estas funciones son inyectivas garantizando que cada objeto posee un único oid.

- Oids, dominio primitivo para identificadores de objetos.
- Vars, dominio primitivo para nombres de variables.
- Evalue = Bvalue + Oids, dominio para valores expresables.
- **Env = [Vars → Evalue]** dominio para ambientes de variables. Un ambiente es una función que liga nombres de variables con valores expresables.
- **Store = [Oids → Object]** dominio para almacenamientos de objetos. Un store es una función inyectiva que liga identificadores de objetos con objetos.

Fig. 4.b -Dominios semánticos para Sistemas de Objetos.

5. Un lenguaje de programación orientado a objetos activos

En esta sección consideramos un simple lenguaje de programación que provee las construcciones mínimas para soportar programación orientada a objetos (tal como definición de clases y envío de mensajes en paralelo) y además permite expresar *mensajes activos* (reglas) dentro de un marco uniforme con los restantes mensajes del sistema.

Sintaxis:

1- Categorías Sintácticas:

D in Declarations
meth in Meth
o in Observer_Message
m in Mutator_Message
bv in Bvalue
s in String
ch in Channel
e in Exp
c in Com p in Prog
list in List f in Fun
r in Rule x in Vars

2. Definiciones:

D ::= **Class** s **Methods** list_meth **Rules** list_r

meth ::= **obs_method** o(x) e | **mut_meth** m(x) c

e::= bv | $f(e_1,..., e_k)$ | x| *state*(x) | $e_1.o(e_2)$

c ::= **skip** | *update*(x,e) | ch?x | ch!e | x:= e | x:=**new**(s)
$e_1.m(e_2)$ | $c_1;c_2$ | $c_1||c_2$ | **insRule**(r) | **delRule**(r)

r::= (**conditions** e **actions** c)

p ::= **Program** list_D c

Utilizamos negrita para indicar los símbolos terminales del lenguaje (por ejemplo **Class**, **Methods, skip**). List_x es una abreviatura genérica que representa listas finitas de elementos de tipo x, es decir, list_x:: = x | x:list_x
Un programa en este lenguaje incluye declaraciones de clases activas y comandos. La declaración de una clase incluye un nombre para la clase y una secuencia de métodos (que representan el comportamiento pasivo tradicional de los objetos). Cada método puede ser un observador o un mutador. El cuerpo de un método observador es una expresión, mientras que el cuerpo de un método mutador es un comando. La declaración de una clase también incluye una lista de **reglas generales**, aplicables a todas las instancias de la clase. Las **reglas particulares** no se declaran sino que se crean dinámicamente en tiempo de ejecución.
Las expresiones comprenden: - valores básicos,- términos de función, - las variables (incluyendo a la variable especial *self* que representa al objeto en sí mismo), - el término *state*(x) que representa el estado interno oculto del objeto ligado a la variable x, - el término $e_1.o(e_2)$ que representa el envío de un mensaje observador con parámetro e_2 al objeto resultante de evaluar la expresión e_1.
Los comandos incluyen: - skip, el comando que no realiza ninguna acción, - el comando *update*(x,e) que representa la modificación del estado interno del objeto ligado a la variable x, asignándole la expresión e (en programas sintácticamente correctos la única variable que puede ocurrir como parámetro de *state* y *update* es la variable especial *self*) , - entrada y salida de datos, - asignación, - *new*(s) que simboliza la creación de un nuevo objeto de clase s, - envío de un mensaje mutador, - el operador ; que indica composición secuencial de comandos, - el operador || que indica composición paralela, - **insRule**(r) para agregar una nueva regla particular en el sistema y **delRule**(r) para eliminar una regla particular.

Reglas Particulares y Reglas Generales

El lenguaje permite estructurar las reglas de la siguiente forma: las reglas generales están asociadas con las declaraciones de clases y las reglas particulares quedan encapsuladas dentro de los objetos, como se muestra en la figura 5.a.. El alcance de las reglas (individuos sobre los cuales cada regla es potencialmente aplicable) está determinado por la jerarquía de clases del sistema. Una regla general es aplicable sobre todas las instancias de la clase donde está definida y también sobre todas las instancias de sus subclases. Evaluar la condición de una regla general tiene complejidad de O(n), siendo n la cantidad de objetos en la clase (incluyendo subclases). Los reglas particulares se refieren a objetos individuales. La evaluación de la condición de una regla particular tiene complejidad de O(1).

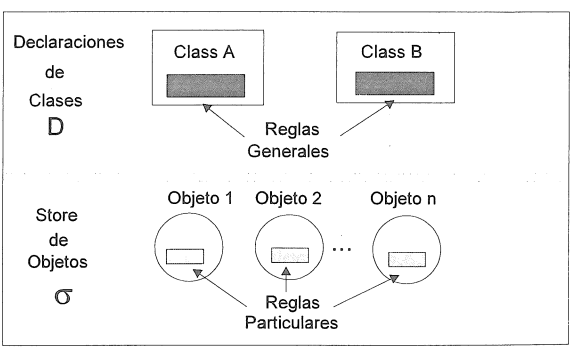

Fig 5.a: Distribución de las reglas dentro del sistema.

El comportamiento del sistema se va definiendo a lo largo de la ejecución, ya que cada objeto al desplegar su comportamiento (ejecutar un método) puede agregar y/o borrar mensajes activos del

sistema. Es decir, cada objeto tiene la posibilidad de "legislar" sobre el sistema (dictando y/o anulando reglas).

Semántica:

Utilizaremos conceptos del formalismo de semántica operacional estructural [Hennessy90].

Las variables δ, σ, bv, ev, obj y oid denotan elementos de los conjuntos Env, Store, Bvalue, Evalue, Object y Oids respectivamente.

Semántica de las expresiones :

Las expresiones denotan valores básicos o identificadores de objetos (llamamos valores expresables, Evalue, al dominio semántico de las expresiones).

Asumimos que no es de interés observar el proceso de computación de expresiones; por lo tanto definimos una función que asocia directamente cada expresión con el valor final que esta denota.

Definimos la función $\Rightarrow_e$ mediante reglas, de la siguiente forma:

$$\Rightarrow_e :: < \text{Exp} , \text{Env} , \text{Store} > \rightarrow <\text{Evalue} >$$

Regla BR

$$(bv,\delta,\sigma) \Rightarrow_e bv$$

Regla VarR

$$(x,\delta,\sigma) \Rightarrow_e \delta(x)$$

Regla StateR

$$(state(x),\delta,\sigma) \Rightarrow_e \Pi_1(\sigma\ (\delta(x)))$$

La regla BR indica el significado de un valor básico. La regla VarR se refiere a la evaluación de variables. De acuerdo con la regla StateR, state(x) denota el estado interno del objeto que en el momento de la evaluación se halla ligado a la variable x. En nuestro dominio los objetos son ternas. Π_1 representa la primera proyección de la terna, es decir el estado interno del objeto.

Regla FunR

$$(e_1,\delta,\sigma) \Rightarrow_e bv_1$$
$$\ldots\ldots\ldots$$
$$(e_k,\delta,\sigma) \Rightarrow_e bv_k$$

$$(f(_k,\ldots, e_k),\delta,\sigma) \Rightarrow_e Apply(f, bv_1 , \ldots, bv_k)$$

Regla ObsR

$$(e_1,\delta,\sigma) \Rightarrow_e oid$$
$$(e_2,\delta,\sigma) \Rightarrow_e ev$$
$$(B(o), \delta[oid/self]\ [ev/x],\sigma\ ) \Rightarrow_e ev'$$

$$(e_1.o(e_2),\delta,\sigma) \Rightarrow_e ev'$$

donde, $B = \Pi_2(\sigma(oid))$.

FunR describe la evaluación de funciones primitivas (tal como suma de enteros).

ObsR describe la evaluación de un mensaje observador. La expresión e_1 denota al identificador del objeto receptor del mensaje o. Llamamos B al comportamiento de este objeto. Entonces B(o) es el método que el objeto computa como respuesta al mensaje o. Para la evaluación del método (que es una expresión) la variable especial self se instancia con el objeto receptor y el parámetro formal x se instancia con el parámetro real ev.

Semántica de los comandos :

Llamamos **Act** al conjunto de acciones primitivas,
Act= { c?bv / c∈Chan , bv∈Bvalue } ∪ { c!bv / c∈Chan , bv∈Bvalue } ∪ { ε }
Este conjunto está integrado por las acciones de entrada y salida de datos y por la acción interna ε. Una acción interna es una acción no observable desde el exterior, por ejemplo las modificación del store o la comunicación interna de dos comandos corriendo en paralelo.
La semántica de computación de nuestro lenguaje está dada por las relaciones $\overset{a}{\Rightarrow}c$, una para cada acción ***a***, de tipo $\overset{a}{\Rightarrow}c$ ::: <Com,Env, Store> → <Com,Env, Store>,
donde la sentencia (c,δ, σ) $\overset{a}{\Rightarrow}c$ (c',δ', σ') significa que el comando c puede ejecutar una acción primitiva ***a*** sobre el ambiente δ y el store σ transformándolos en δ' y en σ' respectivamente, quedando en c' lo que aún resta por ejecutar del comando c. Además, para dar el significado del lenguaje necesitamos tener en cuenta las declaraciones de clases definidas por el usuario; concretamente vamos a definir relaciones parametrizadas sobre declaraciones de clases.
Las reglas para comandos usan un predicado de terminación ✓, cuya definición es la clásica [Hennessy90]. Por limitaciones de espacio sólo presentamos algunas de las reglas que definen la semántica de los comandos.

Regla InR

$$D \vdash (ch?x,\delta,\sigma) \overset{ch?v}{\Rightarrow} c(skip, \delta[bv/x] , \sigma)$$

La regla InR describe el siginificado del comando ch?x que permite leer del canal ch y colocar el valor leído dentro de la variable x. El store de objetos no se modifica.

Regla NewR

$$D \vdash (x:= new(s),\delta,\sigma) \overset{\varepsilon}{\Rightarrow} c(skip,\ \delta[oid/x] , \sigma[obj/oid])$$

donde, existe en D una declaración de clase de la forma **Class** s **Methods** m **Rules** r

La regla NewR denota la creación de un nuevo objeto. La función next_oid retorna un identificador no usado dentro del store, sea oid= next_oid(σ). Este identificador se liga a la variable x dentro del ambiente. La función Sem construye una nueva instancia de la clase s a partir de su declaración. Su tipo es Sem:: Declarations → Object. Para una descripción detallada de la función (incluyendo el tratamiento de definiciones de clases utilizando herencia) puede consultarse [Pons95]. Sea obj = Sem(Class s Methods m Rules r) esa nueva instancia, obj se liga al nuevo identificador dentro del store.

Regla Par2aR

$$\frac{D \vdash (c_1,\delta,\sigma) \overset{ch?v}{\Rightarrow} c\ (c'_1,\delta',\sigma') \qquad D \vdash (c_2,\delta,\sigma) \overset{ch!v}{\Rightarrow} c\ (c'_2,\delta,\sigma)}{D \vdash (c_1 \| c_2,\delta,\sigma) \overset{\varepsilon}{\Rightarrow} c\ (c'_1 \| c'_2,\ \delta',\sigma')}$$

Regla Par1aR

$$\frac{D \vdash (c_1,\delta,\sigma) \overset{a}{\Rightarrow} c(c',\delta',\sigma')}{D \vdash (c_1 \| c_2,\delta,\sigma) \overset{a}{\Rightarrow} c(c' \| c_2,\delta',\sigma')}$$

Las Reglas **Par1aR**, **Par2aR** (y sus dos reglas simétricas Par1bR y Par2bR) definen la semántica de la composición paralela. Los comandos c y c' actúan en forma independiente intercalando sus acciones primitivas. La regla Par2R permite comunicación entre comandos. Si c_1 puede recibir un valor del canal ch y c_2 puede enviar el mismo valor al mismo canal, entonces c_1 y c_2 pueden comunicarse (comunicación handshake). Esta comunicación es una acción interna.

Regla InsR

$D \mid\!- (\text{insRule}(r),\delta,\sigma) \overset{\varepsilon}{\Rightarrow} c(\text{skip}, \delta', \sigma')$
donde $< c, \delta' > = \text{const}(r, \delta)$; $\sigma' = \text{insert}(\sigma, \delta', c)$

La regla InsR define la creación de una nueva regla particular. La regla r, por ser particular, involucra a objetos individuales. Las variables contenidas en r en el momento de la creación de la regla denotan a estos individuos. Dado que las variables pueden ser modificadas (por asignaciones posteriores a la creación de la regla y anteriores a su ejecución), resulta necesario reemplazarlas por constantes. Esto es realizado por la función const. La función insert agrega la regla constante en cada uno de los objetos involucrados en ésta.

Regla delR

$D \mid\!- (\text{delRule}(r),\delta,\sigma) \overset{\varepsilon}{\Rightarrow} c(\text{skip}, \delta, \sigma')$
donde $\sigma' = \text{delete}(\sigma, r)$

La regla delR indica el significado del comando delRule que elimina una regla particular del sistema. La función delete elimina la regla r de los objetos correspondientes dentro del almacenamiento σ .

Semántica de los programas :

La semántica de computación de los programas está dada por las relaciones $\overset{\lambda}{\Rightarrow} P$, una para cada secuencia de acciones primitivas λ, con λ en Act*. Estas relaciones están definidas en la figura 5.b mediante la regla ProgramR.

Notación : $P \overset{\lambda^*}{\Rightarrow} Q$ si $\lambda = a_1 a_2 ... a_n$ y $P \overset{a_1}{\Rightarrow} p_1 \overset{a_2}{\Rightarrow} p_2 \overset{a_3}{\Rightarrow} ... p_{n-1} \overset{a_n}{\Rightarrow} Q$

$\overset{\lambda}{\Rightarrow} P :: < \text{Prog} > \rightarrow < \text{Env} , \text{Store} >$

Regla ProgramR

$D \mid\!- (c , \delta_0 , \sigma_0) \overset{\lambda_1^*}{\Rightarrow} c(c', \delta, \sigma)$
(c', δ, σ) ✓
$D \mid\!- (\delta, \sigma) \overset{\lambda_2^*}{\Rightarrow} R(\delta', \sigma')$
$D \mid\!-$ (δ', σ') end

$< \text{Active\_Program AD } c > \overset{\lambda}{\Rightarrow} P (\delta', \sigma')$

donde λ es la concatenación de λ_1 y λ_2

Figura 5.b : Semántica de los programas

$\overset{\lambda}{\Rightarrow} R :: < \text{Env} , \text{Store} > \rightarrow < \text{Env} , \text{Store} >$

Regla ForwardR

$D \mid\!-$ (δ, σ) choose (r, δ', σ')
$D \mid\!-$ $(c, \delta', \sigma') \overset{\lambda_1^*}{\Rightarrow} c$ $(c' , \delta'', \sigma'')$
(c', δ'', σ'') ✓

$D \mid\!- (\delta, \sigma) \overset{\lambda}{\Rightarrow} R(\delta'', \sigma'')$

La regla seleccionada r es de la forma (**conditions** e **actions** c)

Figura 5.c.: La relación $\overset{\lambda}{\Rightarrow} R$,

La Regla ProgramR indica que el programa comienza ejecutando los comandos contenidos en el main a partir de un almacenamiento inicial σ_0 y un ambiente inicial δ_0. Generalmente estos comandos crearán los objetos del sistema, dictarán algunas reglas particulares y enviarán mensajes imperativos a los objetos para que comiencen a desplegar sus actividades.

Luego de ejecutado el main, el sistema evoluciona al estilo de un sistema de producción. Cada paso de computación involucra la selección de una regla (general o particular) y su ejecución. El

programa termina cuando ninguna regla posee una condición que evalúa a true. Esto es indicado por el predicado de terminación "end" (que por limitaciones de espacio no describiremos aqui).

La relación $\overset{\lambda}{\Rightarrow}$R representa un paso de computación. Cada paso de computación consiste en la selección de una regla y su ejecución completa (hasta llegar a un estado donde el predicado de terminación ✓ es verdadero).

La relación $\overset{\lambda}{\Rightarrow}$R, parametrizada por las declaraciones, está definida en la figura 5.c.

La relación choose retorna una regla cuya condición evalúa a true. Presenta un comportamiento no determinístico ya que más de una regla puede evaluar a true. Choose retorna además el ambiente conteniendo una posible instanciación de las variables que ocurren en la regla y retorna el store modificado cuando la regla es transitoria. Cuando la regla seleccionada es particular el ambiente no se modifica ya que la regla no contiene variables y cuando la regla seleccionada es persistente el store no se modifica ya que la regla permanece almacenada.

6. Un Ejemplo

Cuando un cliente de un Banco se coloca en la cola de espera de un cajero, decide (en función de su paciencia y tiempo disponible) cuanto tiempo permanecerá en la cola. Si luego de transcurrido ese tiempo el cliente aún no ha sido atendido, renunciará a la espera abandonando la cola.
Este ejemplo describe un evento, que llamaremos ***TopeDeEspera***. Este evento desencadena un mensaje activo que llamaremos ***Impacientarse***. El evento ***TopeDeEspera*** se produce cuando la siguiente condición se vuelve verdadera: " Ha transcurrido al tiempo límite de permanencia del cliente en la cola de espera". El mensaje activo ***Impacientarse***, cuyo objeto receptor es una instancia de la clase Cliente, provoca que el objeto abandone la Cola de Espera.

Implementación en NéOpus

El comportamiento reactivo de nuestro ejemplo se puede modelar con la regla TopeDeEspera de la fig 6.a. NéOpus evalúa esta regla para todas las instancias de la clase cliente, estén o no colocados en la cola de espera; por ello resulta necesario colocar en la regla la condición de que el cliente C pertenezca a la cola Q.

```
Regla TopeDeEspera
| Cliente C. ColaDeEspera Q |
conditions
Q Pertenece (C)
C HoraEncolado + C TiempoLimite > = Reloj
HoraActual
actions
C Impacientarse(Q)
```

Figura 6.a: Regla en NéOpus

Implementación en nuestro lenguaje unificado

Al ingresar al banco el cliente recibe el mensaje Encólese. En respuesta a este mensaje realiza las siguientes acciones: a) se coloca en la cola de espera, b) genera una regla transitoria para modelar el evento potencial de que decida abandonar la cola antes de ser atendido. El mensaje activo ***Impacientarse*** queda

```
mut_meth Encólese ( Q : ColaDeEspera)
   Q.Encolar (self);
   self.HoraEncolado := Reloj.HoraActual ;
   insRule ( conditions self.HoraEncolado +
             self. TiempoLimite = Reloj.HoraActual
             actions self.Impacientarse(Q ) );

mut_meth Impacientarse(Q:ColaDeEspera)
   Q.Desencolar(self);
   self.AbandoneElBanco
```

embebido dentro del método Encólese, como puede verse en la figura 6.b.

Si el cliente es atendido antes de que expire su tiempo máximo de permanencia en la cola, entonces la regla anterior debe ser descartada. Esta acción queda embebida dentro del método PaseElSiguiente de la cola de espera, como puede verse en la figura 6.c.

```
mut_meth PaseElSiguiente(Y : Cajero)
        ( not self.vacía )
        ifTrue C:= self.Desencolar;
                delRule ( conditions C.HoraEncolado +
                                C.TAguante = Reloj.HoraActual
                                actions C.Impacientese(Q ) )
        ifFalse skip
```

Figura6.c.: método de la clase Cola de Espera

Análisis de las diferencias

• En NéOpus las reglas activas no forman parte de los métodos, sino que residen dentro de una base de reglas independiente. Con nuestro lenguaje es posible definir reglas en forma integrada dentro de los métodos. • En NéOpus existe una única regla general para todos los clientes, en nuestro modelo se crea una regla particular para cada cliente que ingresa en la cola de espera. Por lo tanto para evaluar si ha ocurrido el evento sólo serán analizados los clientes incluidos en la cola y no todos los clientes del banco. Por la misma razón tampoco resulta necesario colocar en la regla la condición de que C pertenezca a Q. • En NéOpus las reglas son persistentes, por lo tanto la regla TopeDeEspera continuará siendo evaluada aún cuando ningún cliente permanezca en la cola. En nuestro modelo la regla es efímera, luego de ejecutarse desaparece. Este comportamiento es adecuado ya que cuando el cliente abandona la cola ya no puede impacientarse. Si el mismo cliente decidiera regresar al banco y colocarse nuevamente en la cola de espera entonces el método Encólese generará una nueva regla que le permitirá impacientarse nuevamente.

7. Conclusiones

La programación basada en reglas es más adecuada que la programación orientada a objetos para la codificación de tareas reactivas. Existen propuestas que permiten a los programadores orientados a objetos usar ventajas de la programación basada en reglas dentro de un ambiente orientado a objetos, el resultado es un producto donde los dos paradigmas conviven (se integran) pero no se unifican

Hemos definido las bases para un lenguaje de programación orientado a objetos que permite expresar comportamiento activo en forma unificada. El comportamiento activo queda embebido dentro del comportamiento general de los objetos del sistema. Se respeta el encapsulamiento de los objetos y se aprovecha la jerarquía de clases para determinar el alcance de las reglas.

El comportamiento del sistema se va definiendo a lo largo de la ejecución, ya que cada objeto al desplegar su comportamiento (ejecutar un método) puede agregar y/o borrar mensajes activos del sistema. Es decir, cada objeto tiene la posibilidad de "legislar" (dictando y/o anulando reglas.) sobre el sistema.

El lenguaje permite expresar distintas formas de comportamiento activo: general, particular, persistente y efímero. Esto facilita el desarrollo de determinadas aplicaciones, por ejemplo, las reglas efímeras permiten modelizar eventos en forma natural, evitando de esta manera la necesidad de recurrir a un diseño artificioso (donde los eventos son tratados como objetos de primera clase, como en [Haythorn94]).

Por otra parte, la especificación formal provee una descripción precisa de las construcciones del lenguaje , facilitando el desarrollo de software confiable y dando las bases para testear su correctitud.

Bibliografía

[Atkinson87] Atkinson,R and J. Laursen , Opus: a Smalltalk production system, SigPlan Notices 22, 1987.

[Bertino94] E.Bertino, G.Guerrini and D Montesi, Deductive Object Databases, Proceedings ECOOP'94, July 1994, LNCS 821.

[Cardelli85] Cardelli, L.,Wegner P. On Understanding Types, Data Abstraction and Polymorphism. Computing Surveys, 17(4). 1985.

[Eick94] C. Eick and B. Czejdo, "Reactive Rules for C++ ", Journal Object Oriented Programming October 1993.

[Gatziu91] S. Gatziu, A. Geppert, K.Dittrich, "Integrating Active concepts into an object-oriented database system", Proceedings of the 3rd Workshop on database programming Languages, Nafplion 1991.

[Gehani91] N. Gehani, H.Jagadish, "Ode as an active database: constraints and reglas", Proceedings of the 17th International Conference on VLDB, Barcelona 1991.

[Haythorn94] Wayne Haythorn, "What is Object Oriented Design", Journal O O P April. 1994.

[Hennessy90] M.Hennessy, "The Semantics of Programming Languages: An elementary introduction using structural operational semantics" Copyright 1990 by J Wiley@Sons. England. Chapter six.

[Kappel94] G. Kappel, Rausch-Schott, W.Retschitzegger and S. Vieweg, "TriGS: Making a passive object-oriented database system active", Journal OO Programming, August 1994.

[Medeiros91] C. Medeiros and P.Pfeffer, "Object integrity using rules", Proceedings of ECOOP 1991, LNCS 512.

[Miranker93] Miranker, D, Burke, Steele, et al. "The C++ embeddable rule system", International Journal On Artificial Intelligence Tools, 2(1), 1993.

[Pachet95] F.Pachet, On the embeddability of production rules in object-oriented languages, Journal Object Oriented Programming, August 1995.

[Pons95] C.Pons, R.Giandini,G.Baum , "Modelos matemáticos para Objetos", Memorias de Panel'95 - XXI Conferencia Latinoamericana de Informática, Brasil, Julio 1995.