Un Modelo Abstracto de Ejecución para Animar Modelos Conceptuales*

Patricio Letelier **Pedro Sánchez** **Isidro Ramos**
Departamento Sistemas Informáticos y Computación
Universidad Politécnica de Valencia (España)
email {letelier | ppalma | iramos}@dsic.upv.es
http://www.dsic.upv.es/users/oom

Resumen

El modelo conceptual establece los requisitos funcionales del software. La prototipación es una de las técnicas usadas frecuentemente en la validación del modelo conceptual. Sin embargo, los resultados del proceso de validación, con o sin prototipación, distan de ser totalmente satisfactorios. La animación automática de modelos conceptuales emerge como un enfoque de prototipación que puede ayudar en la validación del modelo conceptual. Para realizar animación automática el principal obstáculo lo constituye la posibilidad de conseguir la ejecutabilidad de una especificación. En este sentido los métodos formales proveen ventajas significativas. Además del rigor y precisión provisto por un método formal para la especificación y verificación del modelo conceptual, respecto de su validación, puede definirse en mejores condiciones un proceso de animación automática que respete la semántica del modelo conceptual. $\mathcal{OASIS}$ es un enfoque formal para el modelado conceptual orientado a objeto. En este trabajo se presenta cómo se ha abordado la animación de especificaciones $\mathcal{OASIS}$ y en particular el modelo de ejecución utilizado. Dicho modelo es fiel a la semántica de $\mathcal{OASIS}$, por lo que constituye una base sólida para trabajos en validación de especificaciones $\mathcal{OASIS}$ mediante animación.

Palabras clave: modelo conceptual, especificación ejecutable, modelo de ejecución

1 Introducción

El modelo conceptual, representando los requisitos funcionales de un sistema de información, es la pieza clave para establecer el vínculo entre el espacio del problema y el espacio de la solución. La construcción del modelo conceptual es un proceso de descubrimiento, no sólo para el analista, sino también para el usuario. Durante el modelado conceptual se llevan a cabo esencialmente cuatro tareas: captura de requisitos, modelado o especificación, verificación de criterios de calidad y consistencia, y finalmente, validación. El grado de exhaustividad en la validación del modelo conceptual es un factor determinante para conseguir que el producto final se ajuste a lo esperado.

Para validar los requisitos del sistema lo más extendido es la utilización de técnicas de prototipación integradas a enfoques semiformales. Generalmente los prototipos son bosquejos de la interfaz del producto, concentrándose principalmente en aspectos de entrada y salida del sistema. Un tipo particular de prototipación está orientada a la simulación o animación detallada de la

*Este trabajo ha sido financiado por el proyecto MENHIR de la Comisión Interministerial de Ciencia y Tecnología, con referencia TIC97-0593-C05-01

funcionalidad sistema. Esta forma de prototipación resulta atractiva para cubrir de forma exploratoria la validación del modelo conceptual, analizando su comportamiento ante situaciones de funcionamiento más complejas. El principal obstáculo para la animación del modelo conceptual es conseguir su ejecutabilidad, el esfuerzo de programación puede ser comparable al esfuerzo del desarrollo del producto final. En este sentido la utilización de un método formal para la especificación del modelo conceptual ofrece como ventaja el proporcionar una semántica precisa con la cual establecer un procedimiento automático para la animación de la especificación. Por lo anterior, resulta interesante combinar métodos formales y técnicas de prototipación. En los últimos años se ha registrado gran interés en torno a la validación de especificaciones formales mediante animación [11].

$\mathcal{OASIS}$ 3.0 (**O**pen and **A**ctive **S**pecification of **I**nformation **S**ystems) [5] es un enfoque formal para la construcción de modelos conceptuales siguiendo el paradigma orientado a objeto. Otras propuestas formales con una motivación similar a la de $\mathcal{OASIS}$ son: TROLL [4], ALBERT [1] y OBLOG [9]. En este trabajo se expone resumidamente cómo hemos abordado la animación de especificaciones $\mathcal{OASIS}$ orientada a la exploración interactiva del comportamiento del modelo conceptual. Algunas propuestas relacionadas con validación mediante animación en otros modelos cercanos a $\mathcal{OASIS}$ son la presentada en [3] que utiliza TROLL y la desarrollada en [2], basada en ALBERT. Para OBLOG, se está construyendo una herramienta de animación pero aún no está disponible. Respecto de la animación establecida, las diferencias entre otras propuestas y este trabajo radican principalmente en las características formales del enfoque utilizado y la expresividad ofrecida. Además, en los trabajos relacionados señalados no se presentan detalles del modelo de ejecución utilizado. Finalmente, considerando los resultados publicados, el estado del arte es similar y se limita a versiones preliminares de entornos para animación.

A continuación se indica la organización del resto de este trabajo. En la sección 2 se exponen brevemente los aspectos básicos de $\mathcal{OASIS}$. La sección 3 presenta un modelo abstracto para la ejecución de especificaciones $\mathcal{OASIS}$. En la sección 4 se muestra un ejemplo de aplicación del modelo de ejecución. Finalmente, en la sección 5 se presentan las conclusiones y trabajo futuro.

2 $\mathcal{OASIS}$

Una especificación $\mathcal{OASIS}$ es una presentación de una teoría en el sistema formal usado, expresada como un conjunto estructurado de definiciones de clase. Las clases pueden ser simples o complejas. Una clase compleja es aquella que en su definición utiliza la definición de otras clases (simples o complejas). Las clases complejas se definen estableciendo relaciones entre clases. Éstas son agregación y especialización. Una clase se compone de un nombre de clase, uno o más mecanismos de identificación para instancias de la clase (objetos) y un tipo o plantilla que comparten todas las instancias. A continuación se presentan los conceptos básicos de $\mathcal{OASIS}$.

Definición 1 ***Plantilla o tipo***. *Una plantilla de clase está representada por la siguiente tupla:* $\langle Atributos, Eventos, Fórmulas \rangle$. *Atributos es el alfabeto de atributos. Eventos es el alfabeto de eventos. Fórmulas es un conjunto de fórmulas la variante de Lógica Dinámica [8] utilizada como formalismo subyacente.*

Para una clase simple todas las propiedades quedan establecidas en su plantilla. Para una clase compleja, además de las propiedades establecidas por su plantilla, existirán propiedades adicionales determinadas por los operadores de clase que la definen.

Definición 2 ***Servicio***. *Un servicio es un evento o una operación. En el primer caso se trata de un servicio atómico e instantáneo. Una operación es un servicio no atómico y que, en general, tiene duración.*

Definición 3 ***Acción***. *Una acción es una tupla ⟨Cliente,Servidor,Servicio⟩ que representa tanto la acción asociada a requerir el servicio en el objeto cliente como la acción asociada a proveer el servicio en el objeto servidor.*

Para cada clase, asumiremos la existencia implícita de un conjunto A de acciones obtenido a partir de los servicios que los objetos de la clase pueden requerir (cuando son clientes) o proveer (cuando son servidores). Al igual que para los eventos, $\forall a \in A$ podemos obtener $\underline{a} = \theta a$ siendo θ una substitución básica de los posibles cliente, servidor y servicio.

Definición 4 ***Atributo***. *Un atributo es un par ⟨nombre, sort de evaluación⟩, siendo nombre el identificador del atributo y sort de evaluación el nombre del conjunto soporte (carrier) con los valores que puede tomar dicho atributo.*

Definición 5 ***Estado del objeto***. *Llamaremos estado del objeto al conjunto de sus atributos evaluados. En $\mathcal{OASIS}$ el estado se expresa mediante Fórmulas bien formadas (Fbfs) de lógica de primer orden.*

El estado permite caracterizar a un objeto en un instante determinado. Llamaremos Σ al conjunto de estados alcanzables por un objeto y $\sigma \in \Sigma$ un estado en particular. El estado de un objeto cambia por la ocurrencia de acciones. Los cambios de estado son modelados por la siguiente relación:

$$efecto : \underline{A} \rightarrow (\Sigma \rightarrow \Sigma)$$

Donde $\underline{A}$ es el conjunto de acciones instanciadas ($\underline{a} \in \underline{A}$) que le pueden acontecer al objeto y $efecto$ es la función que asocia a cada acción instanciada una función de cambio de estado, siendo $(\Sigma \rightarrow \Sigma)$ el conjunto de todas las funciones de cambio de estado.

Definición 6 ***Conjunto consistente de acciones***. *Un conjunto de acciones $M \subseteq \underline{A}$ es consistente si a partir de un estado σ dado, el estado σ' alcanzado por la ocurrencia de las acciones en M es siempre el mismo, independiente del orden en el cual se ejecuten*[1] *las acciones.*

Definición 7 ***Paso***. *Llamaremos paso a un conjunto consistente de acciones que ocurren en un mismo instante de la vida del objeto.*

Definición 8 ***Vida o traza de un objeto***. *Llamaremos vida o traza de un objeto a un prefijo finito de pasos que le acontecen al objeto. De esta forma, la traza es una caracterización alternativa del estado del objeto en un instante determinado.*

2.1 $\mathcal{OASIS}$ expresado en Lógica Dinámica

En $\mathcal{OASIS}$ el comportamiento de los objetos de una clase está determinado por un conjunto de fórmulas en la variante de Lógica Dinámica utilizada. Dichas fórmulas establecen prohibiciones, obligaciones y cambios de estado para los objetos, ellas se muestran a continuación.

$\psi \rightarrow [a]false$	"La ocurrencia de a está prohibida en los estados que satisfacen ψ".
$\psi \rightarrow [\neg a]false$	"La ocurrencia de a es obligatoria en los estados que satisfacen ψ"
$\psi \rightarrow [a]\phi$	"En los estados que satisfacen ψ, inmediatamente después de la ocurrencia de la acción a, ϕ debe satisfacerse".

[1] Se utilizará "ejecución de acción" como sinónimo de "ocurrencia de acción" para hacer más intuitiva la presentación. Sin embargo, desde el punto de vista declarativo y formal, "ocurrencia" es el término más adecuado.

Donde a representa la ocurrencia de una acción. Por otra parte, ψ y ϕ son Fbfs en lógica de primer orden que caracterizan el estado del objeto en un instante determinado. Además $\neg a$ representa la no-ocurrencia de la acción a (es decir, que no ocurre nada o que ocurre otra acción distinta de a). Finalmente, $false$ es un átomo tal que no hay un estado que pueda hacerlo verdadero. Esta situación representa la no-existencia de un estado válido alcanzable.

El siguiente ejemplo modela una cuenta bancaria utilizando la sintaxis del lenguaje $\mathcal{OASIS}$.

```
class cuenta
identification
    numero:(numero);
constant attributes
    numero:nat;
    titular:string;
variable attributes
    saldo:nat(0);
    veces:nat(0);
    pin:nat(0);
    categoria:nat(0);
derived attributes
    buen_saldo:bool;
derivations
    good_balance:={saldo>=100};
events
    abrir new;
    cerrar destroy;
    ingreso(Cantidad:nat);
    reintegro(Pin:nat,Cantidad:nat);
    pagar_comision;
    cambiar_pin(Pin:nat,NuevoPin:nat);
    asignar_categoria(Categoria:nat);
valuations
    [ingreso(Cantidad)] saldo:=saldo+Cantidad, veces:=veces+1;
    [reintegro(Pin,Cantidad)] saldo:=saldo-Cantidad, veces:=veces+1;
    [pagar_comision[2]] saldo:=saldo-1;
    [::pagar_comision] veces:=0;
    [cambiar_pin(Pin,NuevoPin)] pin:=NuevoPin;
    [asignar_categoria(Categoria)] categoria:=Categoria;
preconditions
    reintegro(Pin,Cantidad) if {(pin=Pin and saldo>=Cantidad) or
            (pin=Pin and saldo<Cantidad and categoria=2)};
    cambiar_pin(Pin,NuevoPin) if {pin=Pin};
    cerrar if {saldo=0};
triggers
    ::pagar_comision when {veces>=3 and buen_saldo=false and categoria=0};
end class
```

[2] En OASIS, para evitar detallar cada elemento de la tupla $< Cliente, Servidor, Servicio >$ que representa a una acción, se adoptan las siguientes simplicaciones: a) Si el servidor es el propio objeto, se escribe *Cliente*:*Servicio* y si el cliente no es relevante sólo se escribe *Servicio*. b) Si el cliente es el propio objeto, se escribe *Servidor*::*Servicio*. *c)* En el caso particular de comunicación *self*, la acción de requerir se escribe *::Servicio* y la acción de proveer *self:Servicio* (o simplemente *Servicio*).

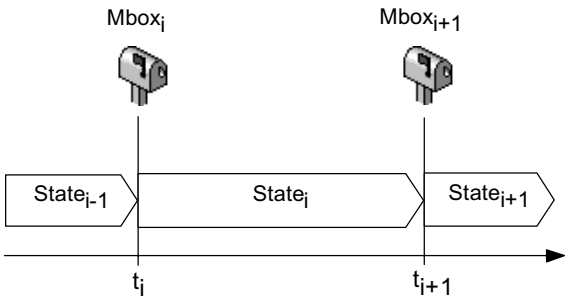

Figura 1: Ciclo de vida de un objeto

3 Un modelo abstracto para animación en $\mathcal{OASIS}$

El modelo abstracto de ejecución establece el vínculo entre la semántica de una especificación $\mathcal{OASIS}$ y una implementación concreta en un entorno de programación. Así, en nuestro caso, el modelo de ejecución no es más que un modelo para animar fórmulas de obligación, permiso y evaluación, asociadas a un objeto. En esta perspectiva, la información utilizada por el modelo es la signatura de atributos y acciones del objeto junto al conjunto de fórmulas en Lógica Dinámica que describen su comportamiento. Cuando se trata con objetos de clases simples esta información es precisamente la plantilla de la clase del objeto. Sin embargo, si la clase del objeto es compleja (por especialización o agregación) no toda la información está directamente en la plantilla de la clase del objeto. A pesar de esto, podemos suponer externo al modelo de ejecución el trabajo que determina la estructura y el comportamiento del objeto en el caso de objetos pertenecientes a clase complejas y así reducir el planteamiento del modelo de ejecución al caso de un objeto instancia de una clase simple.

3.1 Aspectos básicos

La secuencia de pasos que le acontecen a un objeto está ordenada respecto del tiempo. Podemos suponer que existe un objeto "reloj" que envía acciones especiales –llamadas *ticks*- hacia cada objeto del sistema. Los *ticks* recibidos por un objeto son correlativos con los números naturales, t_1, t_2, etc. Siendo i y j números naturales entonces se cumplirá que $i < j \Longleftrightarrow t_i < t_j$.

Definición 9 ***Buzón****. Llamaremos buzón al conjunto de acciones que pueden formar parte del paso que un objeto ejecuta en un determinado tick. El buzón en el instante t_i se denota por $Mbox_i$. Consideraremos el buzón como un buffer ilimitado, en el cual las acciones se ordenan por instante de llegada.*

Definición 10 ***Estado i-ésimo del objeto****. Denotado por $State_i$, representa el estado del objeto en el intervalo $(t_i, t_{i+1}]$. Es decir, inmediatamente después de t_i y hasta t_{i+1} inclusive, el estado se mantiene constante.*

La Figura 1 ilustra la vida de un objeto según las definiciones dadas. El conjunto de acciones en cada paso es obtenido desde el buzón de acciones del objeto y la ejecución del paso en un instante produce un cambio de estado, el cual no será observable hasta el siguiente *tick*. Así, en la estructura de tiempo utilizada aunque la ocurrencia de acciones es instantánea, los estados tienen una cierta duración.

3.2 Concurrencia Intra-Objetual

La concurrencia intra-objetual es imprescindible para una animación fiel a la semántica de $\mathcal{OASIS}$. Básicamente dos situaciones justifican la incorporación de concurrencia intra-objetual en el modelo de ejecución:

- Cuando en un instante más de una acción está obligada a ejecutarse, deben todas ellas realizarse en el mismo instante de lo contrario se contradice la semántica de las fórmulas de obligación correspondientes.
- En el caso de objetos agregados, la ejecución concurrente de los objetos componentes en forma natural puede producir acciones concurrentes vistas desde el objeto agregado.

3.3 Conflicto entre acciones

Si se ejecuta más de una acción en un paso deberían garantizarse dos propiedades: la consistencia y la terminación, o al menos indicar que dichas propiedades no se pueden determinar con certeza en la especificación. La propia definición del concepto de paso establece dicha consistencia, sin embargo, para el establecimiento del modelo de ejecución a continuación se explicará cómo se aseguran dichas propiedades.

Definición 11 ***Relación afecta****. Llamaremos Att al conjunto de atributos del objeto. Se define la relación afecta como:*

$$Afecta_{State_i} \subseteq Mbox_i \times Att$$

Siendo $a_1 \in Mbox_i$ y $att_1 \in Att$, diremos que $a_1 Afecta_{State_i} att_1$ si y sólo si la ejecución de la acción a_1 modifica[3] el valor del atributo att_1 en t_i.

Definición 12 ***Conflicto entre acciones****. Sea $Acc \subseteq Mbox_i$ y Att el conjunto de atributos del objeto. Diremos que Acc tiene conflicto entre acciones si NO se satisface la siguiente condición:*

$$\forall a_1, a_2 \in Acc \text{ y } \forall att \in Att : [a_1 Afecta_{State_i} att \wedge a_1 Afecta_{State_i} att] \Longrightarrow a_1 = a_2$$

Definición 13 ***Conjunto consistente de acciones (versión modelo de ejecución)****. Sea $Acc \subseteq Mbox_i$, diremos que Acc es consistente si no presenta conflicto entre acciones.*

Así, para garantizar la consistencia de un conjunto de acciones en el buzón bastará con asegurar que dichas acciones no están en conflicto. Esto puede detectarse automáticamente, basta con inspeccionar los conjuntos de atributos que pueden sufrir una asignación de valor para cada acción, revisando las evaluaciones asociadas a dicha acción. Los conjuntos de atributos obtenidos deben ser disjuntos para que el conjunto de acciones correspondiente sea consistente. Nótese que cuando existe conflicto entre acciones puede aún existir consistencia (según su definición original). Por ejemplo, consideremos las siguientes evaluaciones:

$$att = V[a_1]att = V + 1$$
$$att = V[a_2]att = V + 2$$

[3]La modificación puede ser indirecta en el caso de atributos derivados. Así, si att_2 es un atributo derivado definido en función de att_1, entonces: $a_1 Afecta_{State_i} att_1 \Longrightarrow a_1 Afecta_{State_i} att_2$.

Las acciones a_1 y a_2 están en conflicto pero al aplicar a_1 y luego a_2 el efecto es el mismo que al hacerlo en el otro orden, con lo cual a_1 y a_2 constituyen un conjunto consistente (según la definición de consistencia original). Sin embargo, la exclusión de estos casos no contradice la semántica de $\mathcal{OASIS}$ y facilita la determinación de los conjuntos de acciones que pueden ejecutarse en un paso.

Una vez seleccionado el conjunto consistente de acciones que forma el paso, se aplican todas las acciones a la vez, produciendo un sólo cambio de estado correspondiente al efecto de todas las acciones. Debido a esto último, la propiedad de terminación en el procesamiento de las acciones contenidas en un paso está siempre asegurada.

3.4 Clasificación de acciones en el buzón

El procesamiento de las acciones del buzón implica clasificar las acciones contenidas en él. A continuación se presentan las categorías de acciones identificables en el buzón, caracterizadas como subconjuntos de $Mbox_i$ (es decir, en el instante t_i).

Definición 14 ***Acciones obligadas de requerir****. Acciones asociadas a obligaciones por requerir un servicio (en las cuales el cliente es el propio objeto) y que* ***deben*** *acontecer. El conjunto de acciones obligadas de requerir se denota por* $OReq_i$.

Las acciones de requerir están determinadas por fórmulas de obligación, es decir, de la forma: $\phi[\neg a]false$, en las cuales a es la tupla $< self, Servidor, Servicio >$, es decir, el cliente de la acción es el propio objeto, indicado usando la palabra $self$.

Definición 15 ***Acciones obligadas de proveer****. Acciones correspondientes a servicios que el objeto* ***debe*** *proveer. Estas acciones están asociadas a obligaciones, en las cuales el cliente no es el propio objeto. El conjunto de acciones obligadas de proveer en se denota por* $OProv_i$.

Las acciones obligadas de proveer están determinadas por fórmulas de obligación, es decir, de la forma: $\phi[\neg a]false$, en las cuales a es la tupla $< Cliente, self, Servicio >$, es decir, el servidor de la acción es el propio objeto.

Definición 16 ***Acciones obligadas de ejecutar****. Este conjunto se denotado por* $OExec_i$ *y se define como:* $OExec_i = OReq_i \cup OProv_i$.

Supondremos que el garantizar la no-existencia de conflicto entre acciones obligadas es parte de la verificación estática de la especificación. Es decir, en tiempo de ejecución no puede presentarse conflicto entre obligaciones. Del mismo modo, supondremos que no puede darse una situación en la cual una acción obligada no está permitida.

Definición 17 ***Acciones no-obligadas de ejecutar****. Acciones correspondientes a servicios solicitados por otros objetos (o por el propio objeto) y que el objeto* ***puede*** *proveer, dependiendo de los permisos establecidos sobre dichas acciones y verificados sobre* $State_i$. *El conjunto de acciones no-obligadas se denota por* $\overline{O}Exec_i$.

Definición 18 ***Acciones rechazadas****. Es un subconjunto de* $\overline{O}Exec_i$ *constituido por acciones no-obligadas de ejecutar cuya ocurrencia no está permitida. El conjunto de acciones rechazadas se denota por* $Reject_i$.

Las acciones rechazadas están determinadas por fórmulas de prohibición, es decir, de la forma: $\phi[a]false$.

Definición 19 ***Acciones candidatas****. Es un subconjunto de $\overline{O}Exec_i$ constituido por acciones no-obligadas de ejecutar cuya ocurrencia está permitida. El conjunto de acciones candidatas se denota por $Cand_i$.*

Además, se cumple que $\overline{O}Exec_i = Reject_i \cup Cand_i$.

Definición 20 ***Acciones ejecutadas****. Es el conjunto de acciones correspondiente al paso que se ejecuta en t_i. El paso se denota por $Exec_i$ y estará constituido por $OExec_i$ más un subconjunto de $Cand_i$, es decir, acciones no-obligadas de ejecutar que están permitidas.*

La selección del subconjunto de acciones de $Cand_i$ que serán incluidas en $Exec_i$ puede responder a distintos criterios y la única restricción es que no se produzcan conflictos entre las acciones del paso.

Definición 21 ***Acciones en conflicto****. Es un subconjunto de $Cand_i$ constituido por acciones no-obligadas de ejecutar cuya ocurrencia está permitida, pero que están en conflicto con alguna de las acciones obligadas de ejecutar o con alguna de las acciones no-obligadas seleccionadas. El conjunto de acciones en conflicto se denota por $Conf_i$.*

Se utiliza un criterio sencillo para la selección de acciones desde $Cand_i$: ante dos acciones que no pueden ser seleccionadas por estar en conflicto se seleccionará aquella llegada antes al buzón. Este criterio evita postergación indefinida de acciones en conflicto. Las acciones en conflicto $Conf_i$ serán "copiadas" al buzón correspondiente al siguiente instante ($Mbox_{i+1}$).

3.5 Comunicación self

La comunicación del objeto consigo mismo se produce cuando una acción obligada de requerir es dirigida hacia el mismo objeto (el servidor es $self$). Esta comunicación es tratada de forma homogénea al resto de comunicaciones, es decir, la acción obligada de requerir será posteriormente recibida en el buzón como una acción no-obligada (una solicitud de servicio). Según esto, en dicha situación se distinguen dos acciones: la acción de requerir el servicio a sí mismo (obligación) y la acción no-obligada de proveer dicho servicio. En general la segunda acción podría sufrir algún retraso y ser procesada en un intervalo de tiempo posterior. La primera por tratarse de una obligación no puede ser rechazada, la segunda sí.

3.6 Actividad en el objeto

El comportamiento de un objeto estará regido por un bucle de actividad interna. Esta actividad realiza la transición del objeto desde un estado inicial $State_i$ a un estado final $State_{i+1}$. Utilizaremos una acción implícita para representar los *ticks* e incluirlos homogéneamente dentro del buzón. Un *tick* corresponderá a la acción $< reloj, Object, tick(t, i) >$. Donde $Object$ es una referencia de objeto servidor que coincide con el propio objeto, es decir, se trata como una acción de servicio solicitada por "reloj" y t es el tiempo real correspondiente al i-ésimo *tick*. La separación real de tiempo entre *tick* y *tick* podría no ser constante.

La Figura 2 ilustra los principales aspectos considerados en el modelo de ejecución. Un objeto transita desde el estado $State_i$ hacia el estado $State_{i+1}$ por la ocurrencia del paso representado por $Exec_i$ correspondiente a un subconjunto de acciones de $Mbox_i$. La comunicación desde otros objetos (o desde el propio objeto) se representa por las flechas etiquetadas como $\overline{O}Exec_i$ y $\overline{O}Exec_{i+1}$. La comunicación hacia otros objetos se representa en la parte inferior de la Figura 2

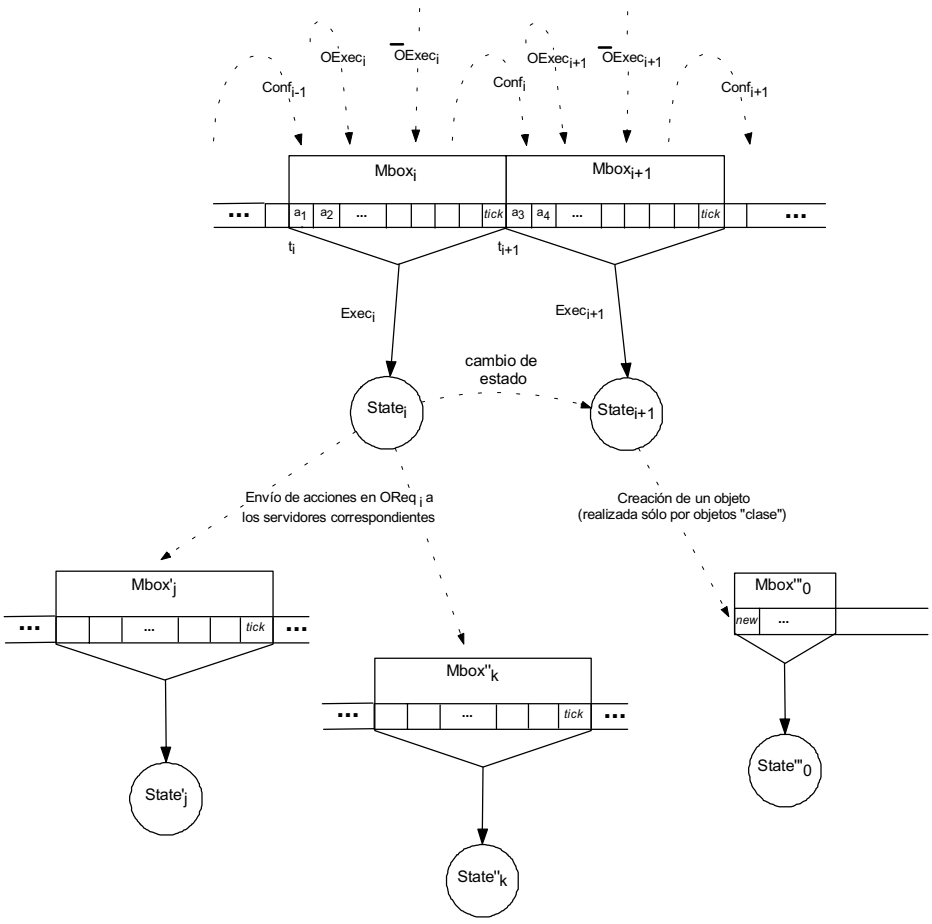

Figura 2: Vida de un objeto en OASIS

por las flechas saliendo hacia buzones de otros objetos. La flecha etiquetada con $Conf_i$ representa las acciones en conflicto en t_i, las cuales son copiadas al siguiente buzón. La flecha etiquetada con $OExec_{i+1}$ representa las obligaciones de requerir que el objeto adquirirá en el instante t_{i+1} producto de la ocurrencia de las acciones $Exec_i$.

Las clases en ejecución serán implementadas homogéneamente como objetos. Estos *metaobjetos* proveen el servicio de creación de objetos de la clase correspondiente. Por lo tanto, como ilustra la Figura 2 en su parte inferior derecha, si un *metaobjeto* ejecuta una acción de creación de instancias (acción *new*) dentro de su proceso de cambio de estado se incorporará la creación de un nuevo objeto, cuya primera acción ejecutada será la acción *new*.

Un objeto examinará su *Mbox* sólo cuando ha llegado al menos una acción *tick*. Si el objeto termina el cambio de estado y aún no se recibe la acción *tick*, esperará hasta que ésta llegue. Por el contrario, si el objeto termina el cambio de estado y ha llegado más de una acción *tick*, sólo se considerará la último, las otras son ignoradas. Nótese que la frecuencia de *tick* determina el tamaño del *Mbox* que se procesará, por lo tanto si se aumenta la frecuencia de *ticks (*sin variar la frecuencia de acciones que llegan al *Mbox*) se puede llegar al extremo en el cual el buzón contiene sólo una acción además de la acción *tick* o incluso sólo la acción *tick*. Por otra parte, reduciendo la frecuencia de *ticks* el buzón tiene la posibilidad de contener mayor cantidad de acciones lo cual podría producir más acciones en conflicto. Cualquiera sea el caso, la frecuencia de *ticks* y el ordenamiento de acciones del buzón representan el indeterminismo propio de los sistemas concurrentes. Sin embargo, la fidelidad respecto de la semántica de $\mathcal{OASIS}$ siempre se mantiene.

La semántica asociada a las acciones obligadas de proveer exige un tratamiento especial. Aunque ya esté presente un *tick* en el $Mbox$ el objeto no puede alcanzar el nuevo estado si no están presentes en él las acciones obligadas de proveer. Esto ocasionará una posible espera, en la cual podrán llegar nuevos *ticks* que serán ignorados. Sólo cuando todas las acciones obligadas de proveer están contenidas en el buzón y ha llegado un *tick* (en un instante posterior) el objeto procesará el buzón y alcanzará el estado siguiente.

Finalmente, el siguiente algoritmo expresa la actividad interna de cada objeto durante su existencia:

$i = 0$
$OProv_i = \{\}$
$State_0 \leftarrow \{\}$
REPETIR
    ESPERAR HASTA QUE $(tick \in Mbox_i \wedge OProv_i \subseteq Mbox_i)$
    $OReq_i \leftarrow obtener\_acciones\_de\_requerir(Mbox_i)$
    $enviar\_acciones\_de\_requerir(OReq_i)$
    $\overline{O}Exec_i \leftarrow obtener\_acciones\_no\_obligadas(Mbox_i)$
    $Reject_i \leftarrow calcular\_acciones\_rechazadas(\overline{O}Exec_i, State_i)$
    $Cand_i \leftarrow \overline{O}Exec_i - Reject_i$
    $Conf_i \leftarrow calcular\_acciones\_en\_conflicto(Cand_i, State_i)$
    $Exec_i \leftarrow OReq_i \cup OProv_i \cup (Cand_i - Conf_i)$
    $State_{i+1} \leftarrow determinar\_nuevo\_estado(Exec_i, State_i)$
    $OProv_{i+1} \leftarrow calcular\_próx\_accs\_obligadas\_de\_proveer(State_{i+1})$
    $OReq_{i+1} \leftarrow calcular\_próx\_accs\_obligadas\_de\_requerir(State_{i+1})$
    $Mbox_{i+1} \leftarrow Mbox_{i+1} \cup OReq_{i+1} \cup Conf_i$
    $i = i + 1$
FIN_REPETIR

Observaciones

- Las funciones "*obtener_...*" seleccionan subconjuntos de $Mbox$ de acuerdo con las definiciones dadas.
- El procedimiento *enviar_acciones_de_requerir* se encarga de hacer llegar las acciones de requerir a los buzones de los respectivos servidores.
- Las funciones "*calcular_...*" utilizan además las fórmulas que definen el comportamiento del objeto (fórmulas de obligaciones, permisos y cambios de estado).
- La función *determinar_nuevo_estado* obtiene el estado siguiente utilizando además las fórmulas de cambio de estado.
- Mediante las definiciones y comentarios respecto de cada uno de los conjuntos involucrados todas las funciones pueden ser implementadas sin ambigüedad en el entorno de programación concurrente elegido.
- El bucle de actividad del objeto puede terminar si el objeto ejecuta una acción asociada al evento *destroy*. Esto podría ocurrir sólo si dicho evento está definido en la clase del objeto.

4 El modelo de ejecución en funcionamiento

Usando el ejemplo de la cuenta bancaria, realizaremos una animación basada en el modelo de ejecución propuesto. La Figura 3 muestra parte del ciclo de vida de un objeto de la clase `cuenta` (entre t_1 y t_5). Para esta instancia de la clase cuenta consideraremos que su estado actual (hasta t_1) es: $State_1 = \{< num, 1010 >, < nombre, juan >, < saldo, 10 >, < veces, 1 >, < categoria, 1 >, < buen\_saldo, false >\}$

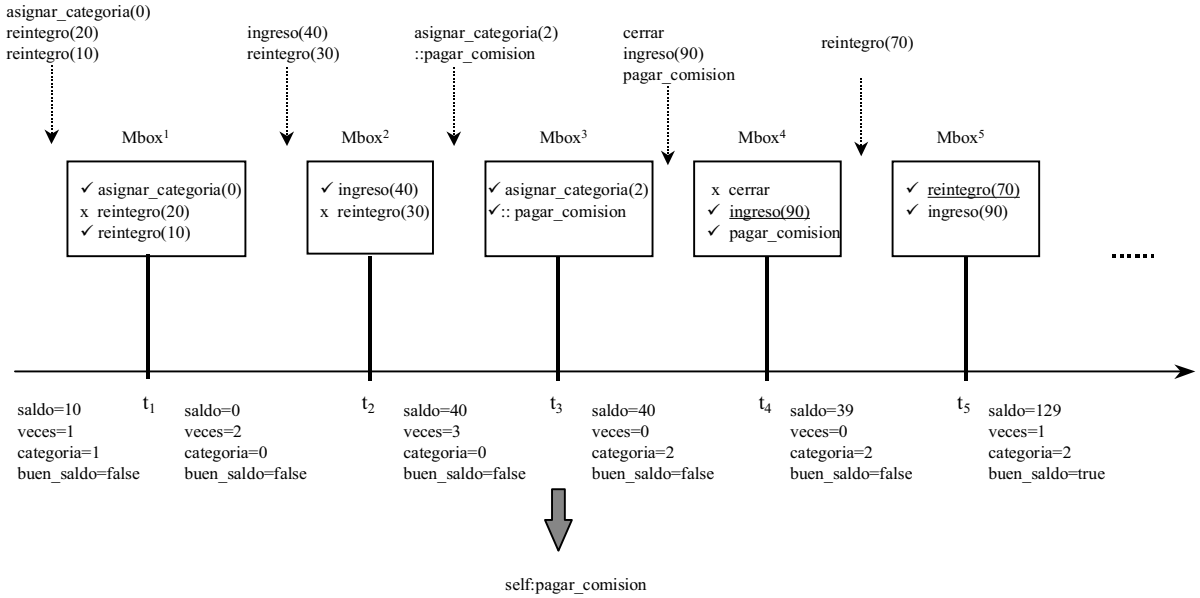

Figura 3: Ciclo de vida de una instancia de cuenta entre t_1 y t_5

El eje horizontal representa el tiempo de izquierda a derecha. Bajo el eje del tiempo se indican los valores de atributos variables del objeto. Entre $(t_i - t_{i+1}]$ el estado del objeto no cambia. Sobre el eje del tiempo y bajo los rectángulos se indica los cambios que sufrirán los atributos variables y que se reflejarán en el siguiente t_i. Los rectángulos representan el buzón ($Mbox_i$) y su contenido en cada. El símbolo "✓"delante de una acción en el $Mbox_i$ indica que está permitida en dicho instante, el símbolo "`x`" delante de una acción indica lo contrario. Las acciones en conflicto y no seleccionadas han sido subrayadas. Las acciones no-obligadas recibidas entre $(t_i - t_{i+1}]$ están representadas en las flechas descendientes entre $Mbox_i$ y $Mbox_{i+1}$. La flecha ancha inferior representa acciones obligadas de requerir.

Después de t_2, y sabiendo que en t_3 será satisfecha la obligación que produce una acción por requerir el servicio `pagar_comision`, es generada automáticamente la correspondiente acción de requerir en $Mbox_3$. En t_3 se tiene en el buzón una acción de requerir asociada a la obligación de solicitar a sí mismo el servicio `pagar_comision`. Posteriormente, se recibe como la correspondiente acción de solicitud de servicio y es ejecutada en t_4. En t_4 y t_5 el buzón contiene acciones que están en conflicto pues el atributo `saldo` puede ser modificado por más de una acción. En t_4 la acción cuyo servicio es `ingreso(90)` está en conflicto con la acción cuyo servicio es `pagar_comision`. Así, la acción cuyo servicio es `ingreso(90)` es postergada hasta el siguiente *tick*, siendo copiada a $Mbox_5$. En t_5 la acción cuyo servicio es `ingreso(90)` está en conflicto con la acción cuyo servicio es `reintegro(20)`. En este caso, esta última acción se ha copiado en el siguiente buzón.

5 Conclusiones y Trabajo Futuro

La animación de especificaciones formales resulta de gran interés para ayudar en la validación temprana de requisitos de un sistema. El principal obstáculo es obtener un prototipo ejecutable de la especificación. En este trabajo se ha presentado un modelo abstracto para la ejecución de especificaciones $\mathcal{OASIS}$. Utilizando este modelo de ejecución se han construido traductores de $\mathcal{OASIS}$ a distintos entornos de programación [7, 10]. Además, se ha construido una versión preliminar de un entorno para especificación incremental y validación de requisitos incluyendo animación automática de modelos $\mathcal{OASIS}$ [6]. En un trabajo posterior, el modelo de ejecución debería ser extendido para incluir aspectos más complejos del comportamiento tales como: restricciones de integridad, operaciones etiquetadas como transacción y otros mecanismos de comunicación más complejos incluidos en $\mathcal{OASIS}$.

Referencias

[1] Dubois,E., Du Bois P., Petit M. *O-O Requirements Analysis: An Agent Perspective*. In Proc. of the 7th European Conf. on OO Programming ECOOP'93, pp.458-481. July 1993.

[2] Heymans P. *The Albert II Specification Animator*. Technical Report CREWS 97-13, Cooperative Requirements Engineering with Scenarios, http://sunsite.informatik.rwth-aachen.de/CREWS/reports97.htm.

[3] Herzig R., Gogolla M. *An Animator for the Object Specification Language TROLL Light*. Proc. Colloq. on Object-Orientation in Databases and Software Engineering, 1994.

[4] Jungclaus R., Saake G., Hartmann T., Sernadas C. *TROLL - A Language for Object-Oriented Specification of Information Systems*. ACM Transactions on Information Systems, Volume 14, Number 2, pp.175-211, April 1995.

[5] Letelier P., Ramos I., Sánchez P., Pastor O. *OASIS 3.0: Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto*. Servicio de Publicaciones Universidad Politécnica de Valencia, SPUPV-98.4011, 1998.

[6] Letelier P., Sánchez P., Ramos I.. *Un Ambiente para Especificacion Incremental y Validacion de Modelos Conceptuales*. Actas del 2^{o}Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software IDEAS'99, pp.216-228, Costa Rica, 1999.

[7] Letelier P., Sánchez P., Ramos I.. *Prototyping a Requirements Specification through an Automatically Generated Concurrent Logic Program*. In Proc. of First Int.Workshop on Practical Aspects of Declarative Languages, LCNS 1551, pp.31-45, Springer-Verlag, 1998.

[8] Meyer J.-J.Ch. *A Different Approach to Deontic Logic: Deontic Logic viewed as a Variant of Dynamic Logic*. In Notre Dame Journal of Formal Logic, vol.29, pp.109-136, 1988.

[9] OBLOG Software S.A. *The OBLOG Software Development Approach (White Paper)*. 1999, `http://www.oblog.pt/Download/Documentation.exe`

[10] Sánchez P., Letelier P., Ramos I. *Una Aproximación a la Representación de OASIS Gráfico en Redes de Petri Orientadas a Objeto*. Actas de las III Jornadas de Ingeniería de Software, JIS'98, pp.267-280, Universidad de Murcia, Murcia, 1998.

[11] Siddiqi J., Morrey I.C., Roast C.R., Ozcan M.B. *Towards Quality Requirements via Animated Formal Specifications*. Annals of Software Engineering, Vol.3, pp.131-155, 1997.