VALIDACIÓN DE MODELOS USANDO ESCENARIOS Y PROTOTIPADO AUTOMÁTICO*

Ángel Roche[1], Patricio Letelier[1], Elena Navarro[2] y Manuel Llavador[1]

1: Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia, Camino de Vera s/n 46022 Valencia
e-mail: {aroche, letelier, mllavador}@dsic.upv.es web: http://www.issi.dsic.es
2: Elena Navarro
Departamento de Sistemas Informáticos
Universidad de Castilla-La Mancha, Avenida de España s/n 02071 Albacete
e-mail: enavarro@dsi.uclm.es

Palabras clave: Validación de requisitos, Escenarios, Prototipado

Resumen. *La ingeniería de software dirigida por modelos está generando grandes expectativas debido a la mejora que podría suponer en productividad y calidad para el desarrollo de software. Así, con el énfasis puesto en los modelos del software, se hace patente la necesidad de ofrecer mecanismos que apoyen su elaboración. La generación automática de código a partir de modelos comprime el ciclo de vida del software y permite obtener versiones ejecutables del sistema en poco tiempo. Sin embargo, el producto final no es el más apropiado para hacer una validación exhaustiva de los modelos de origen, especialmente cuando la lógica del modelo queda accesible a través de una "gruesa" capa de presentación. En este trabajo presentamos un enfoque para especificación y validación de modelos, ajustada al proceso de descubrimiento de requisitos que, de forma incremental, afrontan el analista y el cliente. Nuestra propuesta utiliza un marco formal para la especificación y generación automática del prototipo asociado al modelo. El modelo es validado interactuando con el prototipo usando técnicas de escenarios basados en trazas para establecer tanto el comportamiento esperado como el ejecutado por el modelo. Mediante un ejemplo se ilustra la aplicación de nuestro enfoque y se presenta un entorno software de apoyo que ha sido desarrollado.*

1. INTRODUCCIÓN

Un modelo conceptual permite establecer los requisitos funcionales de un sistema de información y constituye el vínculo entre el espacio del problema y el espacio de la solución. La construcción de dicho modelo conceptual es un proceso de descubrimiento en el cual la interacción entre el cliente y el desarrollador debe ser estrecha. La construcción iterativa e incremental del modelo conceptual es la estrategia más adecuada. El proceso incluye básicamente cuatro actividades: captura de requisitos, modelado o especificación, verificación de criterios de calidad y consistencia, y finalmente validación. El grado de validación con el cliente es determinante para asegurar que el producto se ajuste a lo esperado. En un contexto

* Este trabajo ha sido parcialmente financiado por el proyecto CICYT DYNAMICA TIC2003-07776-C02-01

de generación de código a partir de modelos, el problema de validación no queda resuelto simplemente por la facilidad de obtener rápida y automáticamente una versión ejecutable del producto. Desafortunadamente, el producto final presenta importantes inconvenientes para realizar una validación profunda de los requisitos funcionales contenidos en el modelo de partida. Además, aunque una modificación del modelo no implica gran esfuerzo en cuanto a su generación (automática) la necesaria validación de la nueva versión del modelo no es automática. Lo usual y recomendado es que el modelo conceptual quede implementado en una capa lógica independiente, accesible a través de una capa de presentación que puede ser bastante compleja. Las pruebas funcionales apoyadas por herramientas *capture and replay* se centran en probar la funcionalidad desde una perspectiva de uso del sistema, en términos de interacción mediante las interfaces provistas, siendo difícil asegurar una validación exhaustiva de todo el comportamiento incluido en el modelo conceptual.

Las técnicas de escenarios [1][6] son consideradas como las más eficaces para la captura de requisitos del cliente. El punto clave a favor de los escenarios es que éstos son fáciles de construir y de ser verificados por el cliente. Además, como constituyen una especificación parcial del sistema, son en sí mismos una unidad de incremento que el modelo conceptual debe satisfacer. Así, desde una perspectiva metodológica, en cualquier estado de la construcción (iterativa e incremental) del modelo conceptual sería posible validar el comportamiento del prototipo correspondiente, respecto del comportamiento esperado, establecido por el conjunto de escenarios recolectados hasta el momento.

En este trabajo presentamos un marco de trabajo para validación de modelos basado en animación de especificaciones formales [8]. En nuestra propuesta utilizamos escenarios para la especificación del comportamiento esperado del sistema, el cual puede ser comparado con el comportamiento mostrado por la ejecución del prototipo asociado al modelo [3][7]. Dicho prototipo se genera automáticamente a partir del modelo, aprovechando el formalismo subyacente utilizado para la especificación del modelo.

El resto del trabajo está organizado como se describe a continuación. La sección 2 presenta brevemente la estrategia de generación automática del prototipo a partir del modelo y el marco formal utilizado. La sección 3 muestra un ejemplo que ilustra la aplicación de nuestro enfoque. En la sección 4 se presenta una herramienta software de apoyo para nuestro enfoque. Finalmente, se destacan las conclusiones del trabajo y sus posibles líneas de continuación.

2. PROTOTIPADO AUTOMÁTICO DE MODELOS

La notación más utilizada en la actualidad para representar modelos conceptuales es UML, aunque también están ganando aceptación enfoques de modelado para dominios específicos (Domain Specific Modelling), que permiten definir una notación particular a un dominio. En cualquiera de estos casos, lo esencial del modelo conceptual (en cuanto a la información necesaria para generar código) suele quedar representada básicamente en diagramas de clases, diagramas de estados y/o diagramas de actividad, o alguna variación de ellos.

En nuestro enfoque utilizamos la expresividad de los diagramas de clases y diagramas de actividad de UML, soportada ampliamente por editores gráficos de herramientas CASE, las cuales suelen ofrecer algún mecanismo de exportación de los modelos (usando XML, XMI,

etc.). Estos modelos UML son interpretados como especificaciones OASIS[2], un lenguaje formal para modelado conceptual. En trabajos previos [4][7] hemos establecido las correspondencias básicas entre una especificación OASIS y su representación gráfica utilizando UML. Así, un modelo en UML es traducido en un conjunto de fórmulas en lógica dinámica mediante dichas correspondencias. Éstas formulas son ejecutadas usando el modelo de ejecución de OASIS, constituyendo así un prototipo del modelo original. Para expresar el comportamiento de los objetos, OASIS utiliza una variante de la Lógica Dinámica que incluye operadores deónticos [5]. Dichos operadores se reflejan en las siguientes formulas:

$\psi \rightarrow [a]\,\phi$	En estados que satisfacen ψ, después de la ocurrencia de la acción a debe satisfacerse ϕ
$\psi \rightarrow [a]\ false$	La ocurrencia de la acción a está prohibida en los estados donde se satisface ψ
$\psi \rightarrow [\overline{a}]\ false$	La ocurrencia de la acción a es obligatoria en los estados donde se satisface ψ

Donde ψ es una fórmula bien formada que caracteriza el estado de un objeto. Una acción a representa el evento instantáneo que se produce en un objeto cliente por requerir un servicio a un objeto servidor (o a él mismo), ocurriendo el evento recíproco en el objeto servidor, el cual representa el hecho de proveer un servicio solicitado por determinado cliente. Así, una acción se representa como una tupla <*Cliente, Servidor, Servicio*>. El símbolo $\overline{a}$ representa la no ocurrencia de la acción a (otras acciones podrían ocurrir). Además, *false* representa un estado de violación del sistema que no debería alcanzarse. Así, una acción está prohibida si su ejecución conduce al sistema a dicho estado de violación, y una acción está obligada si su no ocurrencia lleva al sistema al estado de violación.

La vida de un objeto se describe mediante una secuencia de pasos (conjuntos de acciones) ordenada en el tiempo. Cada objeto tiene su hilo de ejecución y periódicamente analiza su *Inbox* procesando las acciones que se han acumulado. Llamamos t_1, t_2, etc. al tiempo de cada activación o "tick" que es isomorfo con los números naturales. Es decir, dados i y j números naturales existe una relación $i < j \Leftrightarrow t_i < t_j$.

Definición 1. *Un Inbox es una lista de Actions que esperan ser ejecutadas por un objeto en un tick. Un Inbox en el instante t_i es denotado por $Inbox_i$.*

Definición 2. *El estado de un objeto en un tick es denotado por $State_i$ y corresponde al estado constante del objeto en el intervalo* $[t_i, t_{i+1})$ *(t_i incluido).*

Presentamos a continuación todas las posibles acciones.

- **Acciones obligadas**: Son acciones que **tienen que** ocurrir. Se denota como $O\,Exec_i$ y son acciones determinadas por fórmulas de obligación de la forma $\psi \rightarrow [\overline{a}]\ false$. Una acción obligada debe ejecutarse antes del siguiente tick.
- **Acciones no obligadas**: Son acciones correspondientes a servicios requeridos por otros objetos (o por sí mismo) que podrían ser ejecutadas o no, dependiendo de las prohibiciones establecidas sobre ellas y verificadas en $State_i$. Son denotadas por $\overline{O}\ Exec_i$. Se distinguen:
  - **Acciones rechazadas**: Acciones no obligadas cuya ocurrencia está prohibida en el estado $State_i$. Son denotadas por $Reject_i$ y son acciones determinadas por fórmulas

de prohibición de la forma $\psi \rightarrow [a]\ false$.

- **Acciones candidatas**: Acciones no obligadas cuya ocurrencia está permitida en $State_i$. Son denotadas por $Cand_i$.

– **Acciones ejecutadas**: Son las acciones que forman el paso ejecutado en t_i, y denotadas por $Exec_i$. Así, un paso estará compuesto por $O\,Exec_i$ unido con un subconjunto de $Cand_i$.
– **Acciones en conflicto**: Son un subconjunto de $Cand_i$ formadas por acciones en conflicto (dos acciones están en conflicto si afectan el valor de conjuntos no disjuntos de atributos) con algunas acciones obligadas o candidatas seleccionadas. Es denotado por $Conf_i$.

3. EJEMPLO DE VALIDACIÓN INCREMENTAL

En esta sección ilustraremos cómo trabaja nuestro enfoque. Usaremos como ejemplo el modelo de una cuenta bancaria sencilla. Siguiendo el proceso planteado en la Figura 1, estableceremos una primera versión del modelo (versión V.1) basándonos en una lista de requisitos (Ver Figura 2), el correspondiente modelo de clases OASIS (Ver Figura 3) y un conjunto de escenarios que validan el modelo mediante la ejecución de su prototipo. Cada escenario se expresa como un conjunto de acciones. Además para cada acción se indica que objeto la ha ejecutado y el estado alcanzado por un objeto tras la ejecución de cada paso (Ver Figura 4). Cada iteración en el proceso está dirigida a satisfacer los escenarios que constituyen el comportamiento esperado del modelo. Los requisitos informales se han capturado de forma textual en una lista a partir de la cual se han definido 2 clases (*cuenta* y *customer*) que darían soporte a dichos requisitos. Finalmente, utilizando 5 escenarios (SC001 al SC005) se ha establecido el comportamiento esperado del modelo. Cada escenario es una traza de acciones acontecidas a ciertos objetos, indicando, cuando corresponda, los nuevos estados alcanzados. Este comportamiento esperado debe ser contrastado con el prototipo ejecutable del modelo, realizando las modificaciones pertinentes en el modelo (y obteniendo un nuevo prototipo) hasta que el comportamiento esperado coincide con el ejecutado.

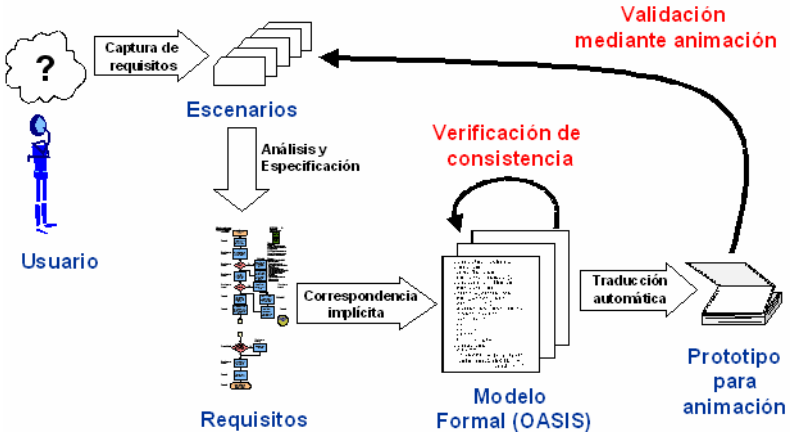

Figura 1: Proceso de especificación y validación incremental de modelos basado en animación

1. Atributos de una cuenta: número (mecanismo de identificación), nombre del titular de la cuenta**, número secreto (PIN), buen saldo (verdadero si el saldo el mayor que 1000 EUROS), categoría y número acumulado de operaciones**.
2. Se permite cerrar una cuenta sólo si su saldo es cero.
3. Se permite efectuar un reintegro sólo si el saldo es mayor o igual a la cantidad a retirar.
4. Los clientes del sistema pueden efectuar depósitos y reintegros y cambiar el número secreto.
5. **Se permite un reintegro sólo si el PIN introducido coincide con el PIN de la cuenta.**
6. **Se permite cambiar el número secreto sólo si el PIN introducido coincide con el PIN de la cuenta.**
7. **Cuando una cuenta acumula 5 operaciones (depósitos o reintegros) se aplica una comisión de 1 EURO, salvo si la cuenta es de categoría 1 o 2, o que se tenga un buen saldo (saldo mayor a 1000 EUROS).**
8. **Las cuentas por defecto son de categoría 0.**
9. **Sólo en cuentas de categoría 2 se permite retirar una cantidad mayor al saldo disponible.**

Figura 2: Requisitos en su versión V.1 (en negrita el incremento de la versión V.2)

```
class account
identification
    number :(number);
constant attributtes
    number :nat;
    name :string;
variable attributes
    balance :int(0);
    pin :nat;
    times :nat(0);
    rank :nat(0);
derived attributes
    good_balance :bool;
derivations
    good_balance={balance>=1000};
events
    open new;
    close destroy;
    deposit(Amount :nat);
    withdraw(Pin :nat, Amount :nat);
    pay_commission;
    change_pin(Pin :nat, NewPin :nat);
valuation
    [deposit(Amount)] balance=balance+Amount
                      and times=times+1;
    [withdraw(Pin, Amount)]
                      balance=balance-Amount
                      and times=times+1;
    [pay_commission] balance=balance-1;
    [pay_commission] times=0;
    [change_pin(Pin,NewPin)] pin=Pinpreconditions
preconditions
    withdraw(Pin,Amount) if
        {(pin=Pin and balance>=Amount)
        or (pin=Pin and balance<Amount and
                rank=2) };
    change_pin(Pin,NewPin) if {pin=Pin};
    close if {balance=0};
triggers
    pay_commission if
        {times>=5 and
        good_balance=false and rank=0};
end class
```

```
class customer
identification
    name :(name);
constant attributtes
    name :string;
events
    add new;
    remove destroy;
end class
```

Figura 3: Clases que forman el modelo en su versión V.1 (mostrando en negrita el incremento para V.2)

SC001: Abrir cuenta		
1	User(root)	<user(root), account, open(number:10, name:Juan, balance:0**, pin:123, good_balance:false**)>
2	Account	<user(root), account, open(number:10, name:Juan, balance:0**, pin:123, good_balance:false**)>
3	account(10)	<user(root), account(10), open(number:10, name:Juan, balance:0**, pin:123, good_balance:false**)>
SC002 (extiende SC001): Abrir cuenta con un número de cuenta ya existente		
4	User(root)	<user(root), account, open(number:10, name:Pedro, balance:0)>
5	Account	<account, user(root), reject(open(number:10, name:Pedro, balance:0**, pin:123, good_balance:false**), duplicate_key)>
6	User(root)	<account, user(root), reject(open(number:10, name:Pedro, balance:0**, pin:123, good_balance:false**), duplicate_key)>

SC003 (extiende SC001): Se abre una cuenta y se crea un cliente. El cliente realiza depósitos consecutivos de 10, 20 y 30 € sobre la cuenta.			
4	User(root)	<user(root), customer, add(name:Pedro)>	
5	Customer	<user(root), customer, add(name:Pedro)>	
6	customer(Pedro)	<user(root), customer(Pedro), add(name:Pedro)>	
	STATE :account(10)	[number:10, name:Juan, balance:0**, pin:123, good_balance:false**]	
7	customer(Pedro)	<customer(Pedro), account(10), deposit(10)>	
8	account(10)	<customer(Pedro), account(10), deposit(10)>	
	STATE:account(10)	[number:10, name:Juan, balance:10**, pin:123, good_balance:false**]	
9	customer(Pedro)	<customer(Pedro), account(10), deposit(20)>	
10	account(10)	<customer(Pedro), account(10), deposit(20)>	
	STATE:account(10)	[number:10, name:Juan, balance:30**, pin:123, good_balance:false**]	
11	customer(Pedro)	<customer(Pedro), account(10), deposit(30)>	
12	account(10)	<customer(Pedro), account(10), deposit(30)>	
	STATE: account(10)	[number:10, name:Juan, balance:60**, pin:123, good_balance:false**]	
SC004 (extiende SC003): Se abre una cuenta y se crea un cliente. El cliente realiza depósitos consecutivos de 10, 20 y 30 € sobre la cuenta. Se intentan realizar reintegros consecutivos de 40 y 50 €			
13	customer(Pedro)	<customer(Pedro), account(10), withdraw(40)>	
14	account(10)	<customer(Pedro), account(10), withdraw(40)>	
	STATE:account(10)	[number:10, name:Juan, balance:20**, pin:123, good_balance:false**]	
15	customer(Pedro)	<customer(Pedro), account(10), withdraw(50)>	
16	account(10)	<customer(Pedro), account(10), withdraw(50)>	
17	account(10)	<account(10), customer(Pedro), reject(withdraw(50)), not(balance>=Amount))>	
SC005 (extiende SC003): Se abre una cuenta y se crea un cliente. El cliente realiza depósitos consecutivos de 10, 20 y 30 € sobre la cuenta. Se intenta cerrar la cuenta.			
13	User(root)	<user(root), account(10), close>	
14	account(10)	<account(10), user(root), reject(close, not(balance=0))>	
SC006 (extiende SC003): Se abre una cuenta y se crea un cliente. El cliente realiza depósitos consecutivos de 10, 20 y 30 € sobre la cuenta. Se intenta realizar reintegro con un PIN erróneo.			**Escenarios exclusivos V.2**
13	customer(Pedro)	<customer(Pedro), account(10), withdrawt(456,40)>	
14	account(10)	<account(10), customer(Pedro), reject(withdraw(456,40), not(pin=Pin and balance>=Amount))>	
SC007 (extiende SC001): Abrir cuenta e intentar cambiar el PIN introduciendo un pin erróneo			
4	customer(Pedro)	<customer(Pedro), account(10), change_pin(456,789)>	
5	account(10)	<account(10), customer(Pedro), reject(change_pin(456,789), not(pin=Pin))>	
SC008 (extiende SC001): Abrir cuenta e intentar cambiar el PIN introduciendo un correcto.			
4	customer(Pedro)	<customer(Pedro), account(10), change_pin(123,789)>	
5	account(10)	<account(10), customer(Pedro), change_pin(123,789)>	

Figura 4: Escenarios ejecutados sobre el modelo V.1 (y V.2 indicando el incremento en negrita)

Posteriormente, en la medida que se captura más detalle en los requisitos, éstos se refinan (incluyendo todo tipo de cambios, pero principalmente incrementos de funcionalidad), lo cual debe reflejarse en el modelo y en los escenarios. El proceso de la Figura 1 se vuelve a repetir hasta conseguir otra vez que el comportamiento esperado coincida con el comportamiento exhibido por el prototipo en ejecución. Por limitaciones de espacio, sólo una versión adicional del sistema (versión V.2) se ha descrito (aparece en negrita en la Figura 2, Figura 3 y Figura 4 el incremento correspondiente a V.2). Como se puede observar se ha extendido la funcionalidad de la clase *cuenta*, lo cual ha incrementado su modelo, modificando y extendiendo a la vez los escenarios asociados al comportamiento esperado.

4. ENTORNO DE ANIMACIÓN

A continuación describiremos la herramienta que hemos desarrollando para apoyar nuestra propuesta. En la Figura 5 se describe tanto la arquitectura del entorno de validación y parte del diseño del prototipo que se genera automáticamente a partir del modelo. La implementación del algoritmo de clasificación y procesamiento de acciones se encuentra definida en la clase *Item*, en el método *Run*. Las clases *Router*, *Clase* y *Objeto* heredan este comportamiento redefiniendo el procesamiento de las acciones recibidas. Un *Router* es creado por la clase *Principal* existiendo siempre una única instancia del mismo. En el momento de su creación ha de establecer las referencias a todas las clases del modelo, como instancias de *Clase*, utilizando para ello reflexión. Todas las acciones recibidas por el *Router* son redirigidas a sus clases respectivas. Cada clase se encarga a su vez de redirigir la acción al objeto correspondiente o de crear objetos si la acción recibida es una acción de creación de objetos. Por último los objetos toman de su *Inbox* aquellas acciones obligadas y las no obligadas que no entren en conflicto para formar el conjunto $Exec_i$ (ver sección 2) y ejecutarse.

La clase *GestionHilos* es la encargada de gestionar los hilos, en los que se ha de ejecutar cada elemento de la animación, iniciándolos, suspendiéndolos, reanimándolos o finalizándolos según se indique desde la interfaz gráfica de usuario (IGU). Además, cuando un ítem recibe, envía, ejecuta o rechaza una acción en el sistema este evento es capturado por *GestionHilos*, el cuál informa a la IGU. Ésta debe implementar la interfaz *IInterfaz* a fin de conseguir independizar el modelo de objetos de la IGU. La IGU almacena la acción en una instancia de *listaAcciones* que contiene todas las acciones que han sido lanzadas en el sistema.

El código de las clases, en nuestro ejemplo, *Account* y *Customer* se genera a partir del modelo que se desea animar, en este caso *simple_bank*, y se compila en una "*Dynamic Linking Library*" (DLL). De esta DLL el animador obtiene la información del modelo original mediante reflexión. Además, ésta es la encargada de ejecutar los servicios de los objetos.

Por último para definir los escenarios de ejecución se utiliza la persistencia de los objetos mediante su serialización en ficheros XML. De este modo los escenarios se pueden volver a cargar y continuar, ejecutar o importar desde una versión del modelo a otra posterior. Junto a

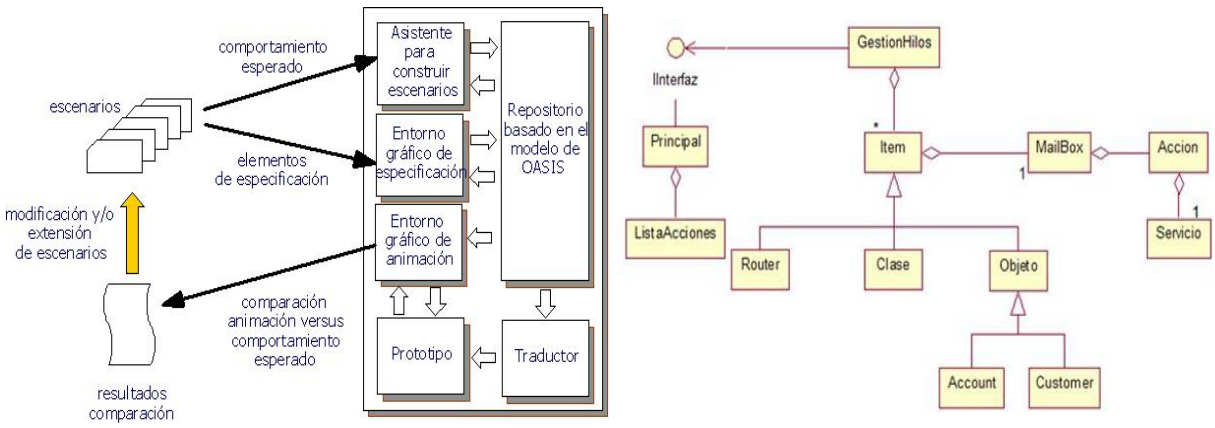

Figura 5: Diseño del entorno para validación de modelos

la DLL generada a partir del modelo se empaquetan todos los escenarios que hemos definido.
El primer paso consiste en generar el prototipo de acuerdo con la especificación. Esto es llevado a cabo por un traductor de la especificación a C#. Dicho prototipo generado es cargado posteriormente pudiendo empezar a interactuar, de forma inmediata, con el prototipo (validando de forma exploratoria). Se puede definir un escenario de forma exploratoria o editando el escenario con las utilidades que proporciona el animador. De esta forma el escenario producido puede ser almacenado para continuarlo más tarde o ejecutarlo. Cuando se carga un escenario almacenado, el entorno restaura el estado alcanzado por los objetos y así se puede continuar su ejecución o editarlo. La ejecución automática de un escenario intenta reproducir la ejecución de cada acción del escenario, comparando la ejecución con el comportamiento esperado definido en el propio escenario. La ejecución de un escenario puede fallar por una o varias de las siguientes razones: porque no se alcance el estado esperado, porque se produzca la ejecución de una acción no esperada (por ejemplo por el disparo de un *trigger*) o porque un servicio que debería haber sido ejecutado sea rechazado o viceversa.
Para una validación exploratoria se pueden crear acciones desde el entorno de animación haciendo doble clic en un ítem del *TreeView*. La acción creada se envía al *Inbox* del ítem. Para editar un escenario el analista debe seleccionar la opción del menú principal "Herramientas >> Generar escenario…" entonces se muestra un formulario igual al de la Figura 6. En dicho formulario el analista pude introducir las acciones tanto ejecutadas como rechazadas y editar los estados alcanzados tras las acciones ejecutadas en un paso concreto.
En la Figura 7 se muestra una sesión de animación. La sociedad de objetos aparece a modo de TreeView a la izquierda. En el lado derecho aparecen las acciones procesadas y los estados alcanzados por los objetos. Por defecto se crea un objeto por cada clase del modelo. Este

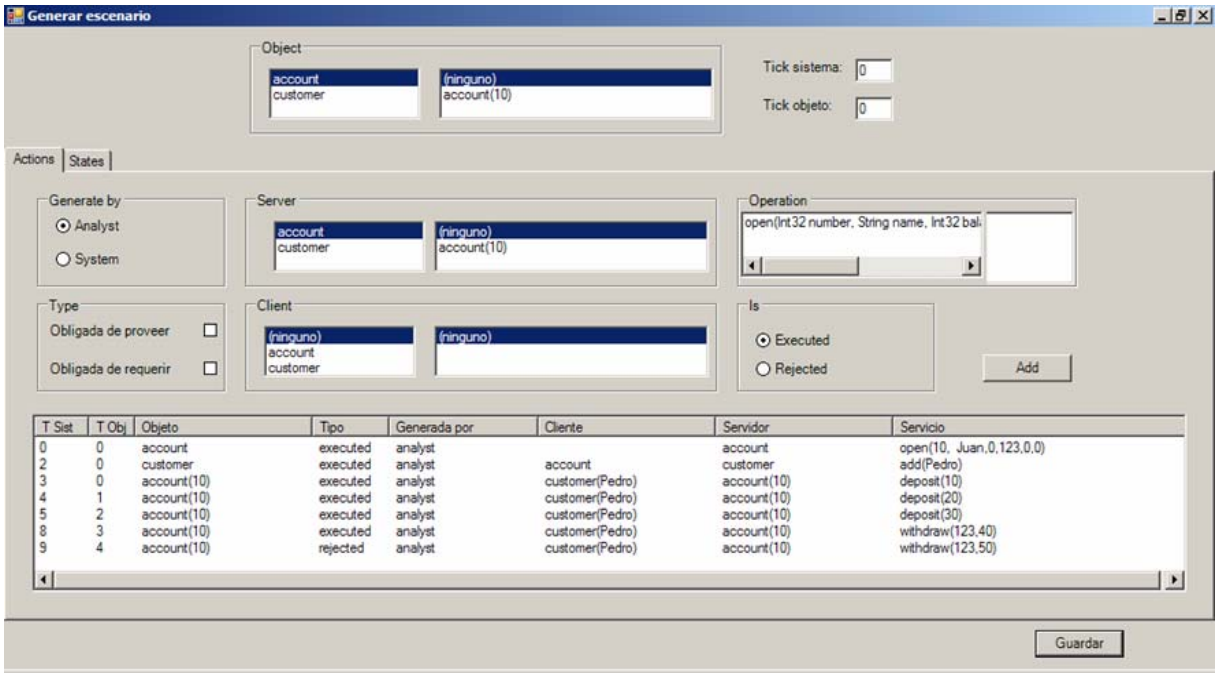

Figura 6: Descripción de un escenario (acciones y estados)

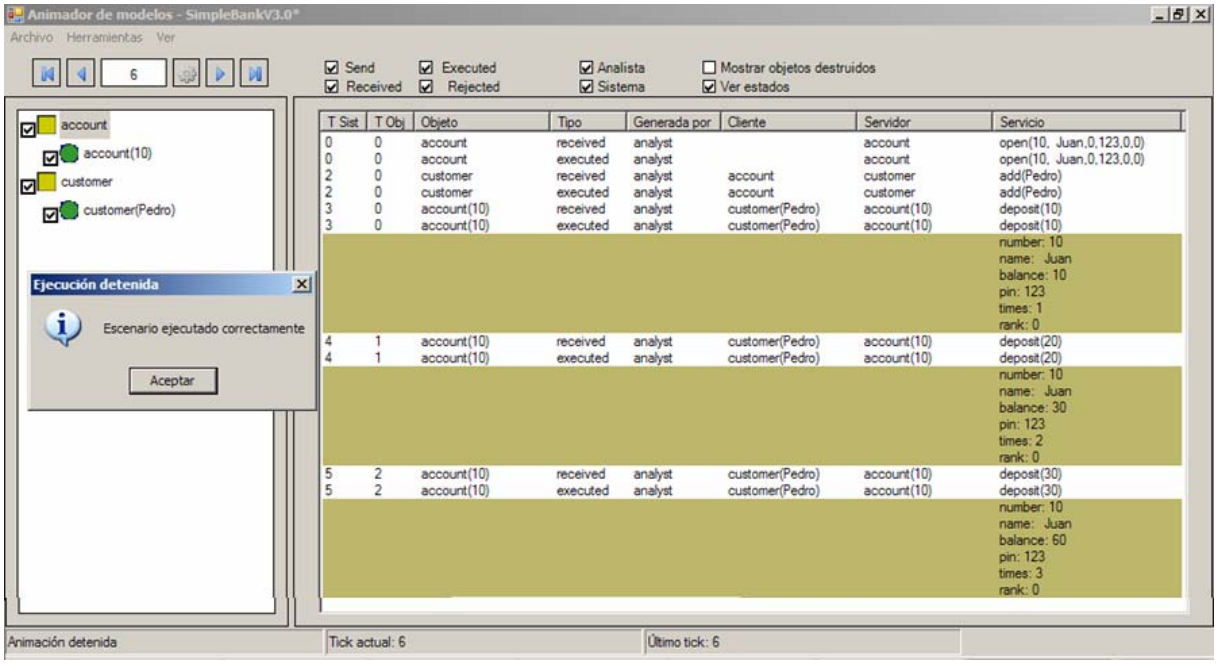

Figura 7: Sesión de animación mostrando un escenario ejecutado

objeto (clase) se encarga de crear nuevos objetos, de registrar el conjunto de objetos en su población y de dirigir las acciones desde objetos clientes hacia objetos servidores de su población. Las acciones pueden ser filtradas según los tipos de acciones (Sent, Received, Executed y Rejected), el objeto concreto seleccionado, o también por acciones generadas por el analista o el sistema utilizando los checkboxes de la interfaz. Los botones Play y Pause proporcionan el control de la sesión de animación. Además, si la animación se encuentra detenida la herramienta proporciona la posibilidad de retroceder o avanzar a un tick determinado.

5. CONCLUSIONES Y TRABAJO FUTURO

La elaboración del modelo conceptual es un proceso de descubrimiento y es importante realizarlo iterativa e incrementalmente validando que se satisfacen los requisitos del cliente. Dicha validación es imprescindible, tanto en un proceso de desarrollo tradicional (usando transformaciones informales y/o poco automatizadas entre modelos) como en un proceso que incluye técnicas modernas de generación de código a partir de modelos. En este trabajo hemos presentado un enfoque para validación de modelos conceptuales basado en la animación de la especificación y una herramienta de soporte.

La mayoría de la investigación y tecnología asociada a generación de código en herramientas CASE está enfocada a conseguir el código del producto final a partir de modelos. Ésta es un área cercana a nuestro trabajo pero difiere respecto del propósito. Aunque el resultado de dicha generación automática de código puede ser utilizado como prototipo, éste no es el más apropiado para realizar validación puesto que por ejemplo, las interfaces del sistema dificultan la prueba exhaustiva de la funcionalidad de las clases. Además, dicha generación de

código requiere por lo general que el modelo esté suficientemente avanzado, lo cual limita su validación temprana. Por otra parte el contar con un marco formal para la animación nos ofrece ventajas en cuanto a la generación automática del prototipo y el asegurar que su ejecución es consistente respecto de la semántica del modelo.

La herramienta desarrollada, en su estado actual, incluye la siguiente funcionalidad: a) generación automática de prototipos, b) almacenamiento, carga, edición e importación (desde otros modelos) de escenarios c) validación exploratoria y validación automática (ejecución automática de escenarios). Nuestro objetivo a corto plazo es ofrecer nuestra herramienta a la comunidad. Estamos trabajando en otros casos de estudio más complejos y completos para hacer más robusta nuestra herramienta y validar en mayor medida nuestro enfoque.

Existe una iniciativa, orientada a proveer ejecutabilidad a los modelos UML, que ha sido incluida en la especificación UML con el nombre de *Action Semantics* (`www.umlactionsemantics.org`). Estamos estudiando la relación entre el modelo de ejecución de *OASIS* y la propuesta de *Action Semantics* para establecer las posibles correspondencias. Inicialmente, el modelo de ejecución de *OASIS* es bastante más abstracto y no está orientado a la ejecución en un lenguaje de implementación, lo cual es uno de los objetivos principales de *Action Semantics*.

REFERENCIAS

[1] Gibson P. *Formal object oriented requirements: simulation, validation and verification*. European Simulation Multi-conference, Warsaw, Poland, June 1999.

[2] Letelier P., Ramos I., Sánchez P. and Pastor O. *OASIS 3.0: Object oriented Conceptual Modeling using a Formal Approach*. Servicio Publicaciones UPV, SPUPV-98.4011, 1998.

[3] Letelier P., Sánchez P. and Ramos I. *Prototyping a requirements specification through an automatically generated concurrent logic program*. First Int. Workshop on Practical Aspects of Declarative Languages, New Mexico, USA, LNCS 1551, pp. 31-45, 1999.

[4] Letelier P., Sánchez P. *Validation of UML classes through animation*. Proceedings of the International Workshop on Conceptual Modeling Quality (IWCMQ'02), in conjunction with ER 2002, pp. 61-73, Tampere, Finland, Octubre 2002.

[5] 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.

[6] Rolland C., Ben Achour C., Cauvet C., Ralyté J., Sutcliffe A., Maiden N.A.M., Jarke M., Haumer P., Pohl K., Dubois E. and Heymans P. *A Proposal for a Scenario Classification Framework*. Requirements Engineering J., Vol. 3, No. 1, pp.23-47, Springer Verlag, 1998.

[7] Sánchez P., Letelier P. and Ramos I. *Validation of Conceptual Models by Animation in a Scenario-based Approach*. Proceedings of OOPSLA 2000 Workshop: Scenario-based round-trip engineering, Tarja Systä (Ed.), Report 20, pp. 32-37, Minneapolis, Minnesota USA. 2000. `www.cs.uta.fi/~cstasy/oopsla2000/schedu-le.html`.

[8] Siddiqi J., Morrey N.A.M., Roast C.R. and Ozcan M.B.*Towards quality requirements via animated formal specifications*. Annals of Software Engineering, n.3, 1997.