Casos de Prueba del Sistema Generados en el Contexto MDD/MDT

Natalia Correa[1], Roxana Giandini[1]
[1] LIFIA- Laboratorio de Investigación y Formación en Informática Avanzada,
Universidad Nacional de La Plata,
1900 La Plata, Argentina
{natalia.correa, roxana.giandini}@lifia.info.unlp.edu.ar

Abstract. *Model Driven Testing* (MDT) es una propuesta incipiente basada en la generación de modelos y artefactos de *testing* a partir de modelos que describen la funcionalidad del sistema bajo testeo. Este trabajo brinda un aporte en tal sentido, presentando un proceso general para la obtención de casos de prueba -del sistema en general y de cada funcionalidad en particular-, automatizando su derivación a partir del modelo de casos de uso, obteniendo a través de transformaciones en *Model Driven Development* (MDD) diagramas de actividades de *testing* intermedios y casos de prueba finales –del sistema y de cada funcionalidad-.

Keywords: Ingeniería de software, MDD, MDT, Transformaciones de Modelos, Casos de Uso, Casos de prueba, Lenguajes de Modelado

1 Introducción

Durante el desarrollo de software, las actividades de testeo del sistema generalmente comienzan a realizarse en la etapa de diseño del sistema, con los detalles de implementación ya definidos. Sin embargo, es en etapas iniciales del desarrollo de software, cuando se define claramente la funcionalidad del sistema, indicando el "qué" sin mencionar el "cómo". Usualmente esta funcionalidad se especifica a través de casos de uso de UML [1], con su respectiva documentación, sus pre y post condiciones, describiendo los pasos a ejecutar e indicando el resultado esperado. Por esta razón nos basamos en los casos de uso como punto inicial de trabajo en la definición de casos de prueba a nivel análisis, teniendo en cuenta además que al testeo de sistemas le concierne la prueba general de todo el sistema en base a sus especificaciones.

Un modelo UML de casos de uso representa la funcionalidad global de un sistema dado, indicando además qué roles inician a esos casos de uso, si hay funcionalidades que se ejecutan como parte de otras o cuáles casos de uso son extensiones de alguna funcionalidad dada. Cada caso de uso aporta información sobre el estado del sistema antes y después de su ejecución, por lo que resulta un artefacto muy útil para el *testing* de los sistemas.

Model Driven Testing (MDT) [2], [3], dentro del contexto de *Model Driven Development* (MDD) [4], [5], [6] es una propuesta reciente que desafía la

problemática de generar los modelos y artefactos necesarios para el testeo de software ya que los modelos de *testing* son derivados total o parcialmente desde modelos que describen la funcionalidad del sistema en desarrollo. MDT define una forma de prueba de caja negra que utiliza modelos -estructurales y de comportamiento- para automatizar la generación de casos de prueba.

MDT se centra en la modelización de los sistemas, donde un modelo es la descripción del comportamiento de un sistema que ayuda a entender el funcionamiento del mismo. Si a partir de este mismo modelo se generan los *tests* para verificar su comportamiento, cuando el comportamiento del sistema cambie, también lo harán los tests. Esta visión general de MDT se presenta en la figura 1.

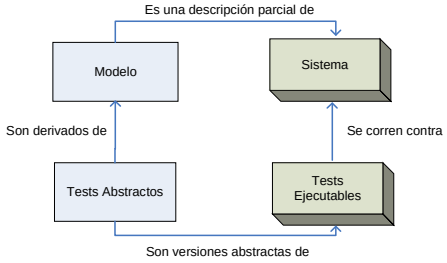

Fig. 1. Esquema general de *Testing* Basado en Modelos

Con el objetivo de brindar soporte al *testing* de sistemas, este trabajo presenta un proceso de generación de casos de prueba del sistema -a los que llamaremos *tests de integración*- que testeen conjuntos de posibles ejecuciones de los usuarios-, automatizando la derivación de los casos de prueba mencionados a partir del modelo de casos de uso. Para obtener los posibles caminos de ejecución mencionados, se propone la generación de un diagrama de actividades que especifique secuencias de ejecuciones funcionales que diferentes roles o usuarios pueden ejecutar sobre un sistema dado. A este diagrama lo denominaremos “Diagrama de Actividades de *Test* del Sistema” (DATS*)*. Un *DATS* puede verse, a su vez, como un grafo donde las actividades son nodos y las transiciones definidas entre las mismas son las aristas. Esta visión será de utilidad al momento de encontrar todos los caminos de ejecución que un usuario puede realizar sobre un sistema bajo testeo.

Este proceso, más general y abarcativo, se completa con otro proceso (definido en un trabajo previo [7]) para especificar casos de prueba a partir de la documentación de cada caso de uso, generando un diagrama de actividades de *testing* intermedio que permite especificar detalladamente qué tarea realiza la actividad de *testing* generada a partir del caso de uso. A esos casos de prueba los llamamos *tests de unidad* y son estos los que primero necesitan ser ejecutados y probados para luego poder probar los *tests* de integración.

La propuesta actual es entonces, una extensión del trabajo previo, permitiendo completar la definición de un proceso para la generación de casos de prueba del sistema en general y de cada funcionalidad en particular, a partir del modelo de casos de uso.

La organización de este artículo es la siguiente: en la sección 2 se introducen los conceptos necesarios sobre modelos de casos de uso y diagramas de actividades. En la sección 3, presentamos el proceso desarrollado para la generación de casos de prueba

del sistema conjuntamente con su implementación. Además, se muestra la integración del presente trabajo con el trabajo anterior. En la sección 4, aplicamos el perfil y la transformación definida en la sección 3 a un ejemplo concreto. La sección 5 analiza trabajos relacionados con nuestra propuesta. Para finalizar, la sección 6 expone conclusiones y presenta líneas de trabajo futuro.

2 Modelos de Casos de Uso vs. Diagramas de Actividades

En esta sección presentamos conceptos introductorios relacionados con casos de uso y diagramas de actividades que serán utilizados en adelante, de forma tal que se pueda comprender la necesidad de definir un perfil para el *DATS* a ser generado.

Un modelo de casos de uso representa la funcionalidad global del sistema. En cuanto a las relaciones entre elementos de modelado, podemos mencionar que:

- una relación <<include>> entre casos de uso especifica que un caso de uso incluye comportamiento definido en otro caso de uso. Esta relación determina que durante la ejecución de una funcionalidad, tiene lugar la ejecución de otra función del sistema. Para ser más explícitos, si un caso de uso CU_A incluye a un caso de uso CU_B, CU_B se ejecutará al ser invocado explícitamente durante la ejecución de CU_A. Finalizado CU_B, el flujo de ejecución vuelve a CU_A. La figura 2 presenta la noción gráfica de la ejecución de la relación <<include>>.

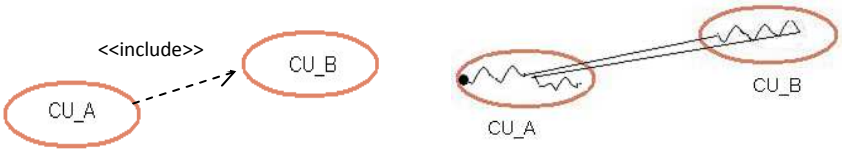

Fig 2. Relación <<include>> (izquierda) y gráfica de su ejecución (derecha)

- una relación <<extend>> entre casos de uso especifica que el comportamiento definido en un caso de uso puede ser extendido por el comportamiento de otro caso de uso. Esta extensión se lleva a cabo en uno o más puntos específicos (puntos de extensión) definidos en el caso de uso extendido, cuando la condición asociada al punto de extensión es verdadera. En otras palabras, que un caso de uso CU_A está relacionado mediante un <<extend>> con CU_B, indica que CU_B será ejecutado durante la ejecución de CU_A sólo si se cumple cierta condición definida en el punto de extensión de la relación. La figura 3 presenta la noción gráfica de la ejecución de la relación <<extend>>.

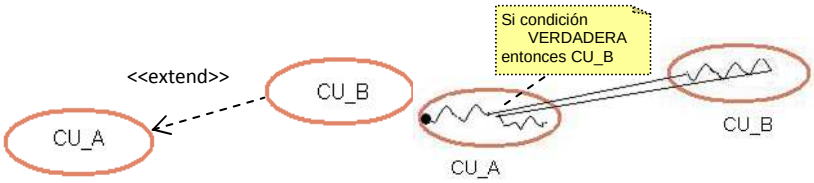

Fig 3. Relación <<extend>> (izquierda) y gráfica de su ejecución (derecha)

Por otro lado, los diagramas de actividades permiten representar un conjunto de acciones o tareas que un objeto (que puede ser el sistema) realiza para completar una funcionalidad dada. Estas acciones, se encuentran relacionadas por medio de transiciones entre ellas.

Una transición es una relación dirigida (origen-destino) entre actividades. Especifica que una vez finalizada la tarea origen, se pasa a la tarea destino directamente, sin que sea necesaria la ocurrencia de eventos.

Las actividades resultan similares a los casos de uso en el sentido en que ambos se componen, básicamente, de una secuencia de acciones. Sin embargo, los casos de uso mantienen, en general, un mayor nivel de abstracción que los diagramas de actividades. Además, éstos últimos no permiten especificar la diferencia semántica entre transiciones y relaciones <<include>> y <<extend>>. Específicamente, la secuencia y el paralelismo del flujo de control entre actividades no son suficientemente apropiadas para expresar la semántica de las relaciones de inclusión y extensión entre casos de uso. Veamos por casos:

- en la relación <<include>>, si se define una secuencia en el diagrama de actividades por medio de una transición entre Act_A y Act_B para representarla, no se respetaría su semántica. Una relación <<include>>, como lo expresamos anteriormente, comienza su comportamiento en CU_A, hace un llamado a CU_B, ejecuta el comportamiento de CU_B y debe retornar a CU_A, mientras que la transición entre dos actividades Act_A y Act_B define que Act_B sólo se ejecutará una vez finalizada Act_A.
- en la relación <<extend>>, tampoco es válido utilizar una secuencia entre actividades para denotarla por dos motivos: en primer lugar, no existe semánticamente en casos de uso tal secuencialidad y, segundo, porque la opcionalidad de ejecución dada por la condición del punto de extensión se omitiría.

Por las razones mencionadas, se evidencia la necesidad de contar con elementos específicos de modelado necesarios para definir en forma correcta los *tests* de integración del sistema, manteniendo la semántica de las relaciones entre casos de uso al transformarlos al *DATS*.

3 Proceso de Generación de *Tests* del Sistema a partir del Modelo Casos de Uso

En esta sección se define el proceso que genera casos de prueba del sistema a partir del modelo de casos de uso. Asimismo y como se ve en la figura 4, se presentan las etapas del Proceso Unificado (proceso de desarrollo de software) [8] donde puede aplicarse nuestra propuesta: en las actividades de Requisitos y de Pruebas. Si bien puede utilizarse en diversos procesos de desarrollo, consideramos RUP por ser éste dirigido por casos de uso, incrementando iterativamente la definición de los mismos.

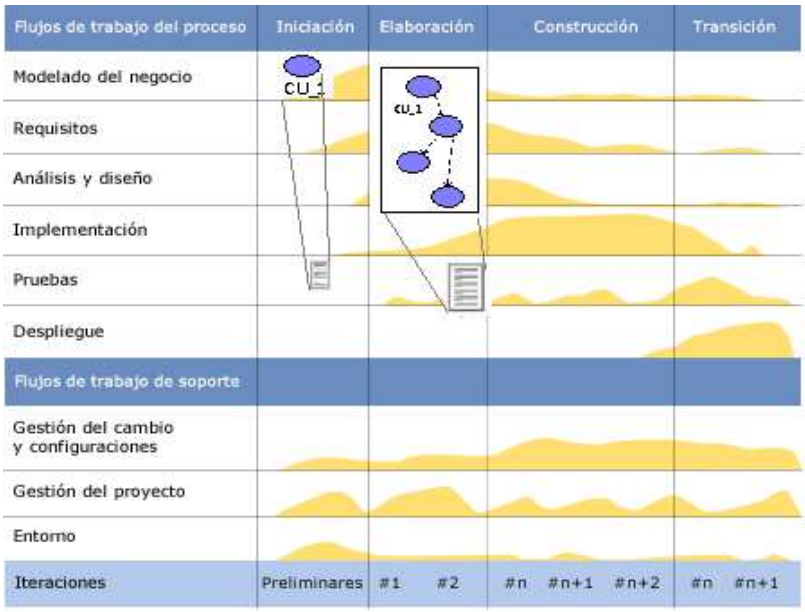

Fig. 4. Generación de casos de prueba del sistema dentro de las actividades de RUP

El proceso presentado en este trabajo se basa en los siguientes pasos:

1. En primera instancia, es necesario definir el Modelo de Casos de Uso -Modelo Independiente de la Plataforma (PIM) en MDD-.
2. A continuación, se aplica una transformación del mismo para obtener un diagrama de actividades del sistema particular para *testing* (DATS)- otro modelo (PIM) en MDD-, construido utilizando un Perfil UML definido para modelar actividades de *testing* del sistema.
3. Finalmente y a partir del *DATS* obtenido, se aplica otra transformación que genera un archivo XML con todas las posibles secuencias de ejecución que pueden realizarse en el sistema y que integran las distintas funcionalidades.

El paso 1 es el más artesanal: un modelo o Diagrama de casos de uso (DCU) contiene la información sobre la funcionalidad relevante del sistema y por ello es nuestro punto de partida, siendo además uno de los modelos que se especifica en las tempranas etapas del análisis dentro del ciclo de vida de un sistema.

A través de la definición de una transformación que toma como modelo de entrada al DCU, se genera un *DATS* enriquecido con el perfil UML al que se hace mención en el paso 2. En el *DATS*, los casos de uso fueron transformados a actividades manteniendo semánticamente las relaciones existentes entre los mismos (inclusión, extensión y generalización).

Se obtiene de esta manera y como resultado de la transformación definida, un modelo de actividades de *testing* del sistema como puede observarse en la parte media de la figura 5. El DATS le permite al ingeniero de software tener una visión de ejecución de funcionalidad global del sistema. También le permite agregar o

modificar estas secuencias de ejecución antes de generar los *tests* de integración o pruebas del sistema.

Cabe destacar que los *test* de unidad generados a partir de la documentación de cada caso de uso y que constituyen los tests de integración mencionados, fueron presentados en un trabajo anterior [7] como una primera etapa dentro del proceso presentado actualmente. Es por ello que no se dan detalles en la presente propuesta sobre la obtención de los mismos.

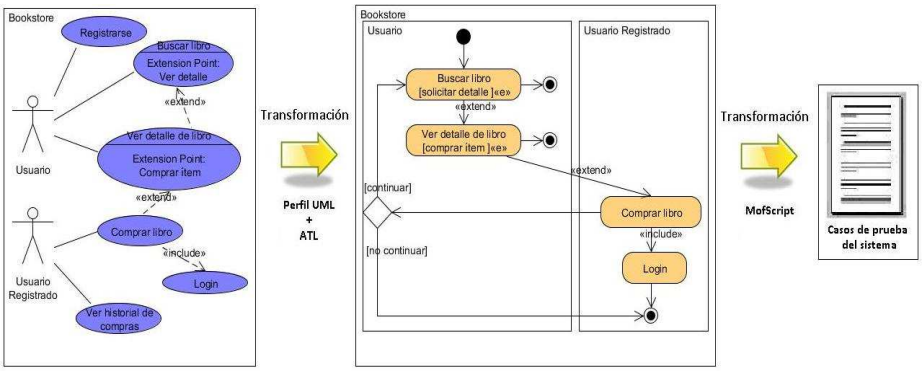

Fig. 5. Proceso de generación de casos de prueba del sistema a partir de casos de uso

Para implementar el proceso de generación automática de casos de prueba del sistema es necesario contar con elementos de modelado específicos que permitan concretar la primera transformación mencionada en esta sección en el paso 2. La construcción de estos elementos de modelado consiste en:

- la definición de un **perfil** UML para actividades de *testing* del sistema
- la definición de una **transformación** entre modelos (M2M) que genere el *DATS* a partir del modelo de casos de uso, manteniendo las relaciones entre los casos de uso y su semántica
- la aplicación del perfil al modelo de actividades del sistema que se genera por medio de la **transformación**

Finalmente, para cumplimentar el paso 3, se define una **transformación** modelo a texto (M2T) a partir del *DATS* obtenido en el paso anterior que genere un archivo XML con los posibles caminos de ejecución que un usuario pueda realizar sobre el sistema y que integran, en distintos casos de prueba, las distintas funcionalidades del sistema.

En las siguientes subsecciones, se detallará la construcción de cada uno de estos elementos específicos.

3.1 Perfil UML para Modelar Diagramas de Actividades de *Testing*

Como mencionamos previamente, para poder especificar las relaciones entre casos de uso en un diagrama de actividades es necesario enriquecer las transiciones entre actividades. Para ello se hace uso de uno de los mecanismos de extensión provistos por UML: la definición de un perfil.

Los estereotipos definidos tienen base en la metaclase ActivityEdge del metamodelo UML v2.4. En esta versión, el elemento de modelado mencionado, reemplaza el uso de Transition en el modelado de actividades en UML 1.5. Acompaña a este diagrama de metaclases, una especificación formal de los nuevos estereotipos con sus restricciones definidas en OCL [9]. La figura 6 muestra la definición gráfica del perfil.

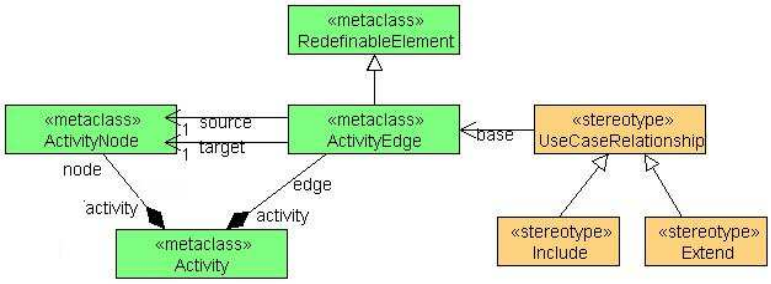

Fig. 6. Perfil UML para actividades de *testing* del sistema con base en ActivityEdge

Para la definición de las reglas de buena formación descriptas a continuación, se utilizó la versión 2.2 de OCL que incorpora operadores de ejecución de operaciones y envío de señales. En este trabajo en particular, se hace uso del operador hasSent() -^- notado por '^.operation' que indica que la operación *operation* fue ejecutada. Sólo se presentan las restricciones del estereotipo UseCaseRelationship por razones de espacio. La definición completa puede consultarse en el Informe Técnico [10].

STEREOTYPE <<UseCaseRelationship>>
Base: ActivityEdge
Constraints:

1- Una transición <<include>> o una transición <<extend>> entre 2 actividades -source y target- implica la existencia de una transición (activityEdge) del target al source que permite la continuidad de la ejecución de la actividad source inicial.

```
self.activity.node->exists (an:ActivityNode|
                  an.source.activity = self.activity) and
an.target.activity.node -> exists (an: ActivityNode|
                        an.source.activity = self.activity)
```

2- Una actividad termina de ejecutarse cuando sus transiciones <<include>> y <<extend>> fueron ejecutadas o bien evaluadas sus condiciones de ejecución

```
let performedActions: Sequence(OclMessage)=
     self.activity^action in
     performedActions -> notEmpty() and
     performedActions.hasReturned()
where
     action:: Sequence(ActivityNode) -> Sequence(Action)
     action= self.activity.edge-> collect:
            (aedge:ActivityEdge| aedge(OCLType)= Action)
```

3- No existen autotransiciones (transición a la misma actividad) cuando la transición es una transición <<include>> o una transición <<extend>>

```
self.activity.edge -> forAll:( aedge:ActivityEdge|
                               aedge.source<> aedge.target)
```

3.2 Transformación para Obtener un Modelo de Actividades de *Test* a partir de Casos de Uso

A partir del modelo de casos de uso y utilizando el perfil presentado en el punto anterior, definimos una transformación que nos permita obtener un *DATS*. Esto es, un diagrama de actividades reflejando la funcionalidad global del sistema modelada con casos de uso y en el que puedan especificarse gráficamente los posibles caminos de ejecución que pueden realizar los actores.
Una idea similar a esta ha sido presentada en [11]. Las diferencias con ese enfoque se presentan en la sección "Trabajos Relacionados".
Al representar la ejecución de las funcionalidades en un diagrama de actividades, tenemos muchos elementos de modelado ya provistos y que pueden destacarse como ventajas dado que enriquecen el modelo y le otorgan mayor expresividad. Por ejemplo, de las actividades podemos utilizar sus parámetros de entrada, sus guardas y acciones, los estereotipos ya definidos <<precondition>> y <<postcondition>>, además de la representación de objetos como ingreso a una actividad o como artefacto de salida producido por un comportamiento determinado.

A continuación se expresan las reglas que componen la transformación mencionada.

3.2.1 Reglas de la Transformación de Modelo de Casos de Uso a *DATS*

Definimos las siguientes reglas de transformación:

1. El nombre del diagrama de actividad o modelo destino, proviene del nombre del diagrama de casos de uso adicionándole el prefijo "ad" –por Activity Diagram- .
2. Por cada actor en el modelo de casos de uso se define una calle en el modelo destino
3. Por cada caso de uso se define una actividad en el modelo destino ubicada dentro de la calle que corresponda según el actor que inicie el caso de uso

Relaciones explícitas

4. Cada relación entre dos casos de uso, se corresponde con una transición entre dos actividades.
   a. Si la relación es <<include>>, la transición será estereotipada con <<include>> indicando que se incluyen los pasos de la actividad de destino y que luego se retorna el flujo de ejecución a la actividad fuente.
   b. Si la relación es <<extend>>, la transición será estereotipada con <<extend>> y la actividad agregará una guarda <<E>> y una acción asociada que denotan el punto de extensión definido en el caso de uso.
   c. Si es una generalización, el caso de uso abstracto no se tiene en cuenta en la transformación. Los casos de uso que se transforman son los concretos o hijos.

5. Existen varios inicios y varios fines de ejecución en el *DATS*. Inicialmente, cada actividad tiene su actividad de inicio y de fin, indicando que puede realizarse sólo esa funcionalidad (por ejemplo, el usuario sólo quiere realizar búsquedas). Para aquellas funcionalidades relacionadas con otras, se agrega además la transición correspondiente de una hacia la otra, manteniendo el inicio y el fin correspondientes.

Esto último es indicativo de que el camino de ejecución puede terminar allí o bien puede continuar con otra funcionalidad. Por ejemplo, el usuario, luego de buscar un libro, se interesa en ver el detalle del mismo. Para la definición de esta transformación se eligió el lenguaje de transformaciones ATL (ATLAS Transformation Language) [12]. ATL es un lenguaje de transformaciones, que cuenta con una herramienta de soporte que permite editar, definir y ejecutar transformaciones modelo a modelo (M2M). Dado que se ajusta a nuestros requerimientos, además de contar con buena documentación y ejemplificación, fue el lenguaje elegido para la definición de las transformaciones M2M.

3.3 Transformación del *DATS* a Casos de Prueba del Sistema

A partir del DATS, puede derivarse un archivo XML que contenga los caminos de ejecución, expresados como secuencias de ejecuciones. Ante todo, el DATS generado le permite al ingeniero de software tener una visión global de la secuencia de acciones que un usuario puede realizar en el sistema. Si este no estuviese completo, en el sentido en que algunas actividades no transiten hacia otras, se podría en este punto relacionarlos a través de transiciones en el *DATS*. Estas son las "relaciones implícitas" entre las diferentes funcionalidades. Es decir, hay transiciones entre actividades no definidas explícitamente en los casos de uso, pero que por lógica de ejecución pueden ser requeridas como dependencias entre funcionalidades (por ejemplo, un usuario para poder comprar debe loguearse, para poder loguearse debe registrarse en el sistema). Eso será útil también para la conformación de los casos de prueba por la completitud de información otorgada al modelo.

La visión del diagrama de actividades como un grafo dirigido, permite también encontrar todos los caminos de ejecución entre las funcionalidades, o los caminos de ejecución de una longitud determinada, pudiendo contemplar la aparición de ciclos en la ejecución. La figura 7, muestra el *DATS* como un grafo dirigido conexo del que pueden extraerse con un algoritmo DFS (Deep First Search) los caminos de ejecución desde una actividad raíz inicial hasta las actividades hojas. Esta idea se menciona en el trabajo de Briand y Labiche [11].

Es necesario tener en cuenta los ciclos en las ejecuciones; por ejemplo, un usuario puede buscar un libro, ver su detalle, volver a buscar, ver el detalle y seguir de esta manera, o finalizar, o bien comprar el libro. Estos son los caminos de ejecución que se enumerarán en la transformación final en el archivo XML. En este sentido se utiliza una estrategia similar a la usada para probar ciclos en código: el ciclo se completa al menos una vez (y como máximo un número arbitrario de veces).

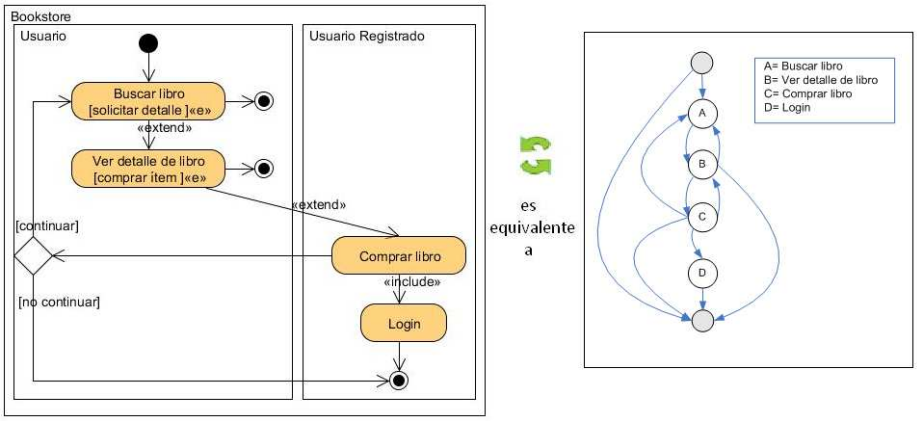

Fig. 7. DATS visto como grafo dirigido

Para la definición de la transformación del modelo DATS al texto de los casos de prueba se eligió MOFscript [13]. MOFscript es un lenguaje de transformaciones que cuenta con la implementación de una herramienta -del mismo nombre-, que permite definir y ejecutar transformaciones modelo a texto basadas en el estándar QVT (Query, View and Transformation) [14]. Utilizamos este lenguaje por adecuarse a nuestro requerimiento de obtener un texto como salida y por estar actualmente adoptado masivamente por la comunidad MDD

3.3.1 Reglas de la Transformación de *DATS* a Casos de Prueba del Sistema

Definimos las siguientes reglas de transformación:

1. El nombre del archivo de casos de prueba del sistema o modelo destino, coincide con el nombre del *DATS* con el agregado de la extensión XML.
2. Por cada subgrafo conexo en el DATS:
   2.1 Se crea un número de secuencia de camino principal en el archivo destino
   Mediante un algoritmo DFD se obtienen todos los caminos desde un punto inicial (actividad de inicio) hasta el punto final (actividad de fin), pudiéndose indicar el número de ciclos a especificar, siendo 2 el valor por defecto. Entonces:
   2.2 Por cada camino de ejecución obtenido se le asigna un número de secuencia de camino en el archivo o modelo destino
   2.3 Por cada actividad dentro de cada camino de ejecución, se genera una llamada a una funcionalidad; o sea, se nombra la actividad en el modelo destino.

Cabe aclarar que los puntos de inicio y final son ahora únicos por cada subgrafo dirigido conexo. El número de secuencia, permitirá identificar con mejor precisión a la secuencia de ejecuciones que se realizan en un camino dado. De esta manera y teniendo en cuenta la figura 6, todas las secuencias comenzarán con 1 dado que el grafo presentado es el único grafo del diagrama (no hay subgrafos) y cada camino tendrá su número de secuencia indicativo. Por ejemplo, 1.1 denotará la secuencia de ejecución <A, fin>, 1.2 <A, B, fin>, 1.3 <A, B, A, fin>.

3.4 Integración de las Propuestas Actual y Anterior para la Generación de Casos de Prueba a partir de Casos de Uso

Esta sección presenta el proceso completo de generación de casos de prueba a partir de casos de uso. Todo el proceso descripto en las secciones previas, hace mención a la generación de casos de prueba del sistema. El trabajo anterior presentó un proceso que permite obtener un caso de prueba (junto a casos de prueba de datos) de cada caso de uso. La integración de ambos trabajos se da en la "llamada" que se realiza desde los casos de pruebas del sistema a cada caso de prueba particular, de forma tal que se puedan testear todos los posibles caminos de ejecución con las pruebas funcionales correspondientes a cada componente. En este contexto, los casos de prueba del sistema funcionan como esquemas de prueba que realizan llamadas de ejecución a los casos de prueba funcionales. La figura 8 muestra la integración entre los dos trabajos desarrollados, completando el proceso de generación y automatización de casos de prueba para un sistema determinado.

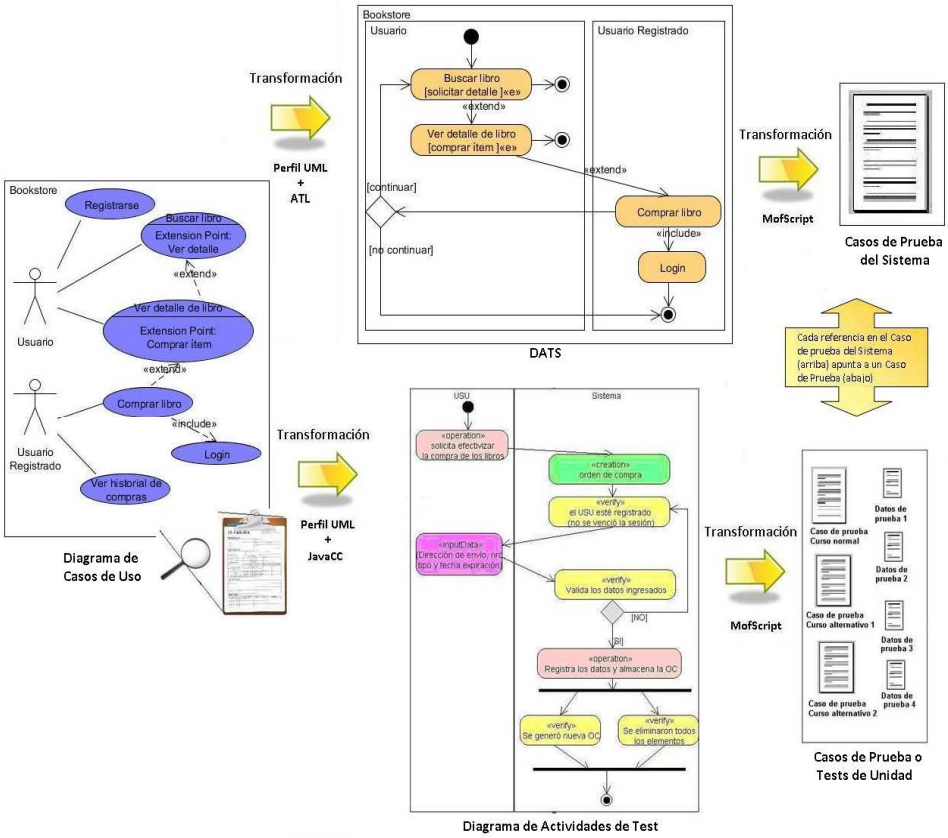

Fig. 8. Proceso completo de generación de casos de prueba a partir de casos de uso

En la figura se ve que a partir del modelo de casos de uso, la propuesta actual genera un DATS a nivel de sistema por medio de una transformación entre modelos, donde se explicitan las ejecuciones que los usuarios pueden realizar sobre el sistema. Este diagrama intermedio podría ser completado si fuese necesario con las relaciones implícitas entre funcionalidades. A partir del DATS se generan los casos de prueba del sistema, o sea, todos los caminos de ejecuciones, a través de una transformación M2T. Estos casos de prueba conforman un esquema general sin estar asociados a ningún lenguaje en particular; o sea, no son aún *test* ejecutables.
Por otra parte, y como fue presentado en el trabajo anterior [7] , del mismo modelo de casos de uso y a partir de una transformación desde cada caso de uso y su especificación, se genera un diagrama de actividades de test que permite identificar qué función cumple cada paso de la especificación. A partir de ese modelo, se generan por medio de otra transformación, los casos de prueba finales.

4 Ejemplo

El siguiente ejemplo ilustra el uso del proceso definido para la generación de casos de prueba, haciendo uso del perfil y de las transformaciones definidos para este proceso.

El ejemplo se basa en un sistema de venta de libros por Internet, al estilo Amazon, que permite vender y comprar libros, tomado del trabajo [4]. En forma sucinta, los usuarios que visiten la página del *bookstore* pueden buscar y ver detalles de los libros, comprarlos, ver novedades y comentarios de usuarios, entre otras funcionalidades.

4.1 Definición de los Casos de Uso del Ejemplo

Los casos de uso del ejemplo se observan en la figura 5, donde se presentó parte del proceso enunciado. Por razones de espacio, no se muestra completamente el Modelo de Casos de Uso, pero pueden ser consultados en el Informe Técnico [10] donde se han descripto también sus especificaciones completas, incluyendo pre y post condiciones.

4.2 Transformación de los Casos de Uso en *DATS*

Los casos de uso presentados conforman el modelo de entrada de la transformación que permite generar el *DATS* del sistema, modelo que también se observa en la figura 5 y que se genera a partir de las reglas de transformación definidas en la sección 3.4. Por razones de espacio, no se muestra el código de la transformación, pero puede ser consultado en [10].

4.3 Transformación del *DATS* en Casos de Prueba del Sistema

El modelo obtenido es un diagrama de actividades (DATS) donde se mantienen las relaciones semánticas explicitadas en el modelo de casos de uso. En este punto, el

ingeniero de software podría querer incluir otras relaciones que no se explicitaron en el diagrama de casos de uso, pero que serían útiles para la posterior generación de los casos de prueba. Por lo tanto es útil ver a nuestro DATS como un grafo dirigido ya que pueden inferirse los posibles caminos que un actor puede ejecutar en el sistema. Testear esos caminos de ejecución brinda la ventaja de conocer si una secuencia de acciones puede ser realizada o no en el sistema; o bien, si ese camino es exitoso, falla, o genera un error.

Luego de la ejecución de la transformación, se obtiene un archivo con todos los caminos posibles que un usuario puede realizar sobre el sistema, contemplando los ciclos de ejecución. La figura 9 muestra parte del código de la transformación en MOFscript y parte del listado de los posibles caminos de ejecución para el modelo de casos de uso del ejemplo. El código de la transformación completo puede ser consultado en [10].

```
texttransformation DATSToTest (in
model:"http://www.eclipse.org/uml2/2.1.0/
UML") {
model.Model::main () {
   file ("test_" + self.name.firstToUpper()
+ ".txt");
  self.ownedMember->forEach(a:
model.Activity) {a.activityToTestLine();}
     println ("}");} //end of main
model.Activity::activityToTestLine(){
```

```
//test_Bookstore
<mainSequence> 1
</mainSequence>
  <subSeq>1</subSeq>
   <path>SearchBook(book).
    END </path>
  <subSeq>2</subSeq>
   <path>SearchBook
   (book).SeeDetailsOf
   Book(book).END </path>
```

Fig. 9. Código de la transformación y casos de prueba del sistema generados

5 Trabajos Relacionados

La generación de casos de prueba a partir de modelos ha sido estudiada en varios trabajos anteriores. Muchos de ellos, generan la estructura de los *tests* a partir de los diagramas de clase o de su implementación [15], [16], o de pre y post condiciones de métodos [17] proveyendo las estructuras de las clases de prueba sin que se le agregue o defina un comportamiento al test. Otras propuestas especifican el comportamiento de los tests a través de diagramas de secuencia de *testing* [18], [19], [20] contando con la estructura del caso de prueba (por medio de un diagrama de clases) y de los objetos del sistema. Casos similares se han estudiado en el dominio de aplicaciones web, donde diagramas de interacción del usuario (UIDs) definen la funcionalidad a verificar [21], [22] que permiten generar y correr casos de prueba a partir de la interacción del usuario con la interfaz del sistema.

Entre los trabajos investigados hemos encontrado una propuesta que se asemeja a la nuestra [11]. En este caso los autores proveen una metodología para obtener casos de prueba desde diagramas de casos de uso, diagramas de interacción y diagramas de clase de un sistema determinado y el punto de similitud se da en el diagrama de casos de uso ya que generan una derivación a un diagrama de actividades. En esa propuesta los casos de uso se representan directamente en un diagrama de actividades, notando la secuencialidad, bifurcación o unión de actividades. Para los casos de uso relacionados por medio de un <<include>>, se ejecuta primero el caso de uso incluido

y luego el que lo incluye. Este enfoque, a nuestro criterio, no es coincidente semánticamente con el concepto de la relación <<include>>. Además, no se presenta una mención sobre cómo se obtiene desde el diagrama de casos de uso al diagrama de actividades derivado. No se especifican las relaciones <<extend>> ni la generalización.

Como conclusión del estudio de trabajos relacionados podemos señalar que en general, los casos de prueba son definidos en etapas de diseño del desarrollo de software o aún más tardías y que se utilizan diagramas de secuencia para completar la funcionalidad de los casos de prueba. Otro aspecto observado, es que todas las propuestas tienden a trabajar (generalmente en forma implícita) sobre la idea de caminos de ejecución, como secuencias de acciones que los usuarios realizan con el sistema, por ejemplo diagramas de secuencia y de interacción de usuario. En nuestra propuesta, utilizamos los diagramas de actividades UML. Estos diagramas permiten modelar en forma más concreta los casos de prueba del sistema y admiten la visualización en forma clara y precisa de todos los posibles caminos de ejecución.

Otro aspecto visto en estos trabajos es que se presenta la generación o derivación automática de los casos de prueba a partir de diagramas de interacción o bien desde diagramas de clase. A diferencia de los trabajos mencionados, el objetivo de este artículo se centra en definir un proceso de generación de casos de prueba en etapas tempranas del desarrollo de software, proveyendo además una forma de automatizarlo en el contexto de metodologías modernas –MDD/MDT- y lenguajes estándares de modelado y transformación de modelos, como son UML, QVT, ATL y MOFScript.

6 Conclusiones y trabajo futuro

En este trabajo hemos presentado un proceso para especificar casos de prueba del sistema a partir del modelo de casos de uso, incentivando el comienzo del proceso de *testing* en etapas tempranas del ciclo de desarrollo de software. En consecuencia, se evita posponer esta actividad para etapas posteriores, generando casos de prueba con transformaciones a partir de los modelos de especificaciones funcionales y permite mantener una consistencia entre los modelos funcionales y los de *testing*.

Específicamente, se ha definido un perfil UML que permita mantener la semántica relacional de casos de uso en los diagramas de actividades del sistema. Se definió una transformación para generar un DATS que especifica los flujos de ejecuciones que los usuarios pueden realizar en el sistema. Asimismo, se definió una transformación modelo a texto que genera los casos de prueba del sistema, con estas secuencias de ejecuciones mencionadas.

Respecto a líneas de trabajo futuro, como primera etapa, se extenderá el proceso de generación de casos de prueba, adicionándole la definición de trazas entre los elementos de cada modelo (original y generados) de forma tal que pueda mantenerse una relación entre artefactos del modelo de análisis y artefactos de *test* durante todo el proceso, pudiendo asegurar la consistencia entre los modelos.

Como segunda etapa, se espera definir una herramienta de soporte que permita automatizar completamente el proceso, permitiendo aplicar al Modelo de Casos de Uso, las transformaciones que generen el DATS, los diagramas de actividades con conceptos de *testing* (los definidos en el perfil UML) y sus consecuentes –segundas-

transformaciones a casos de prueba y casos de pruebas del sistema. El editor gráfico será creado como plugin para Eclipse [23] con EMF [24] y GEF [25]. JavaCC [26] y ATL se utilizarán como herramientas auxiliares para las transformaciones de la documentación del caso de uso al diagrama de actividades en un caso y, en el otro caso, para la transformación de DCU a DATS. Como los casos de prueba generados son textuales, otra mejora futura, es que puedan luego ser implementados como *tests* ejecutables en algún lenguaje particular.

Referencias

1. UML 2.3. The Unified Modeling Language Superstructure version 2.3 – OMG Final Adopted Specification. http://www.omg.org (2010).
2. Baker, P. et all: Model-Driven Testing Using the UML Testing Profile. Springer-Verlag
3. Blackburn, M., Busser, R., Nauman, A.: "Why model-based test automation is different and what you should know to get started". Int.Conf. Practical Software Quality & Testing (2004).
4. Stahl, T., Völter, M. Model-Driven Software Development. John Wiley & Sons, Ltd. (2006)
5. Pons, C., Giandini, R., Pérez, G.: "Desarrollo de Software Dirigido por Modelos. Conceptos teóricos y su aplicación práctica". EDULP & McGraw-Hill Educación. (2010).
6. Kleppe, A., Warmer J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison- Wesley Longman Publishing Co. (2003)
7. Correa, N., Giandini, R.: "Generación Automática de Casos de Prueba a partir de Casos de Uso: Una Propuesta Basada en MDD/MDT". ASSE 2011. Argentine Symp on 40 JAIIO (2011)
8. Kruchten, P.: The Rational Unified Process. Addison Wesley (2000)
9. OCL 2.2. The Object Constraint Language Specification-OMG. http://www.omg.org (2010).
10. Proceso de Generación de Casos de Prueba. Definición, implementación y ejemplo. https://sol.lifia.info.unlp.edu.ar/~nataliac/ (2012)
11. Briand, L., Labiche, Y.: A UML-Based approach to system testing. 4th International Conference on the Unified Modeling Language, Modeling Language and Tools, London (2001)
12. ATL (ATLAS Transformation Language) http://www.eclipse.org/m2m/atl/
13. MOFScript Home page - http://www.eclipse.org/gmt/mofscript/
14. MOF QVT final adopted specification. Technical Report ptc/05-11-01, OMG (2005).
15. Abadía, A., Barisich, J.: Testing Basado en Modelos. Especificación Gráfica y Derivación Automática de Código. Tesis de Grado. Facultad de Informática, UNLP. (2009)
16. Hartman, A., Nagin, K.: The AGEDIS Tools for Model Based Testing. UML Satellite Activities (2004)
17. Vallespir, D.: "Generación Automática de Casos de Prueba Unitarios para Objetos" Universidad de la República, Montevideo, Uruguay (2004)
18. Palacios, L.: Perfiles de Testing Aplicados a Modelos de Software. Tesis de Magister. Facultad de Informática, UNLP (2009)
19. Javed, A., Strooper, P., Watson, G.:Automated Generation of Test Cases Using Model-Driven Architecture. 2nd. International Workshop on Automation of Software Test (2007)
20. Apfelbaum, L., Doyle, J.: "Model Based Testing." Software Quality Week Conf. (1997).
21. FlexMonkey. Functional Testing Tool. www.gorillalogic.com/flexmonkey
22. Selenium Web Application Testing System. http://seleniumhq.org/
23. The Eclipse Project. Home Page. http://www.Eclipse.org/.
24. Eclipse Modeling Framework EMF. http://www.eclipse.org/modeling/emf/
25. Graphical Editing Framework GEF. http://www.eclipse.org/gef/
26. Java Compiler Compiler. http://javacc.java.net/