Generación Automática de Casos de Prueba a partir de Casos de Uso: Una Propuesta Basada en 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 nueva propuesta que se basa en la definición de tests basados en modelos y en la definición de procesos sistemáticos para la obtención de casos de prueba. Este trabajo presenta un proceso para la generación automática de casos de prueba a partir del modelo de casos de uso de UML, contando con la ventaja de definir tempranamente casos de prueba en la etapa de análisis incentivando así el comienzo del proceso de testing en el momento en que un requerimiento funcional se encuentre identificado y documentado; evitando de esta manera posponer la actividad de testeo a etapas de construcción, donde corregir errores implica importantes pérdidas de tiempo y esfuerzo.

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

1 Introducción

"El éxito del testing depende fuertemente de nuestra capacidad para crear pruebas útiles, para descubrir y eliminar problemas en todas las etapas del desarrollo" [1]. Esta capacidad mencionada por Beizer requiere que los casos de prueba tengan el alcance y el tipo necesario para cada caso y que además, se diseñen y definan desde una etapa temprana del proceso de desarrollo. Dada la complejidad per se del proceso de testing dentro del proceso de desarrollo, es deseable y necesario que cuente con un proceso definido y que pueda ser automatizado, parcial o totalmente.

Model Driven Testing (MDT) [2], dentro del contexto de *Model Driven Development* (MDD) [3], [4], [5] es una nueva propuesta que desafía esta problemática. MDT define una forma de prueba de caja negra que utiliza modelos (estructurales y de comportamiento) para automatizar la generación de casos de prueba.

Las actividades de testeo a nivel modelado, generalmente se realizan en la etapa de diseño del sistema, con los detalles de implementación ya definidos[1]. Sin embargo, es en etapas iniciales del desarrollo de software, cuando se define claramente la

[1] Trabajos que abordan esta temática serán descriptos en la sección de "Trabajos Relacionados"

funcionalidad del sistema, indicando el "qué" sin mencionar el "cómo". Usualmente esta funcionalidad se especifica a través de casos de uso, con sus pre y post condiciones, partiendo desde la funcionalidad a definir e indicando el resultado esperado. Es por esto que los casos de uso pueden brindan el punto inicial de trabajo en la definición de casos de prueba a nivel análisis.

Entre los lenguajes de modelado que pueden ser utilizados para automatizar la generación de modelos de testing se encuentra UML [6], estándar oficial de la OMG (Object Management Group) [7], maduro, adoptado y utilizado por la comunidad de las ciencias de la computación, además de contar con numerosas herramientas CASE que son utilizadas por los equipos de desarrollo.

Nuestra propuesta se basa en la generación automática de casos de prueba a partir del modelo de casos de uso de UML contando con la ventaja de la definición temprana de casos de prueba en la etapa de análisis. La generación de estos casos de prueba se hará con diferente granularidad: casos de prueba de cada funcionalidad del sistema –a partir de cada caso de uso- y casos de pruebas de datos –a partir de los datos necesarios para la ejecución de un caso de uso-. En adición, nos serviremos de diagramas de actividades UML, derivados del modelo de casos de uso, que serán útiles para especificar el comportamiento de los casos de prueba; y de una transformación modelo a texto para obtener como resultado final de este proceso, las pruebas.

La organización de este artículo es la siguiente: en la sección 2 comentamos trabajos relacionados, realizando una comparación entre ellos y nuestra propuesta. En la sección 3, presentamos el proceso desarrollado para la generación de casos de prueba. Se detalla el trabajo realizado, a saber: transformación de la documentación del caso de uso a un diagrama de actividades, definición de un perfil UML para estereotipar actividades y de una segunda transformación –basadas ambas en MDD- que genera el caso de prueba a partir del diagrama de actividades enriquecido con el perfil definido. En la sección 4, aplicamos el perfil y la transformación definida en la sección 3 a un ejemplo concreto. Para finalizar, la sección 5 expone las conclusiones del presente artículo y presenta líneas de trabajo futuro.

2 Trabajos Relacionados

La generación de casos de prueba 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 [8], [9], [10]. Estos trabajos definen casos de prueba utilizando diagramas de clase para definir la estructura de los tests, transformando cada clase de diseño en una clase de prueba y cada operación de la clase en un mensaje a ser testeado. De esta forma, se generan las estructuras de las clases de prueba sin que se le agregue o defina un comportamiento al test; sólo se obtiene la estructura –en algún lenguaje XUnit-

Otras propuestas especifican el comportamiento de los tests a través de diagramas de secuencia de testing para las operaciones a validar. Para ello es necesario contar con la estructura del caso de prueba (por medio de un diagrama de clases) y de los

objetos del sistema. Mediante una derivación o transformación del diagrama de secuencia se completa la funcionalidad del test a la estructura ya definida. [11], [12].
Casos similares se han estudiado en el dominio de aplicaciones web, donde diagramas de interacción del usuario (UIDs) definen la funcionalidad a verificar. En estos casos estudiados, es en la etapa del diseño o bien de implementación en que se generan casos de prueba. En particular, hemos estudiado herramientas como FlexMonkey [13] o Selenium [14] que permiten grabar, generar y correr casos de prueba a partir de la interacción del usuario con la interfaz del sistema. FlexMonkey resulta muy claro y fácil de utilizar, cuenta con muy buena y actualizada documentación, pero tiene algunas limitaciones. Una de ellas es que no pueden elegirse los casos de prueba a correr dentro de un set de pruebas (o deshabilitar uno en particular); corren todos o ninguno. Otra desventaja es que sólo pueden testearse componentes Flex. Tener alguna funcionalidad que, por ejemplo, adjunte un archivo y para ello abra la ventana del explorador del sistema operativo, es un requerimiento para el que no se podrá generar un testeo. Observamos que la definición de los casos de prueba se encuentra acotada a una implementación en particular. De la misma manera, distintos ensayos de uso han sido ejecutados con Selenium, llegando a conclusiones similares a las arribadas con FlexMonkey.

En definitiva, nos encontramos con que los casos de prueba son definidos en etapas de diseño (los trabajos que generan tests a partir de diagramas de clases, por ejemplo) o aún más tardíamente: una vez implementado el sistema.

Algo similar ocurre con los diagramas de comportamiento elegidos para completar la funcionalidad del caso de prueba. Por lo estudiado en otras propuestas, observamos que generalmente se utilizan diagramas de secuencia. En nuestra propuesta, hacemos uso de los diagramas de actividades. La primera razón de esta elección es la fase temprana del proceso de desarrollo en la que centramos nuestro proceso. Otra razón, es que permite visualizar en forma clara y precisa, todos los posibles caminos de ejecución (flujos normal y alternativos del caso de uso) adicionando como actividades las pre y post condiciones y las llamadas a otros casos de uso. Este modelo obtenido es utilizado como entrada para la segunda transformación definida, que obtiene como modelo de salida un caso de prueba XUnit.

A diferencia de los trabajos estudiados y 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.

3 Proceso de Generación de Tests a partir de Casos de Uso

Es nuestro propósito definir un proceso que genere casos de prueba a partir de casos de uso (nuestro proceso se centra más en el proceso de desarrollo que en los tests en sí mismos). Este proceso se basa en una serie de pasos que se detallan a continuación (ver Figura 1):

1. En primer lugar, se debe definir el Modelo de casos de uso -Modelo Independiente de la Plataforma (PIM) en MDD- y la documentación textual de cada uno.

2. A continuación y a partir de cada caso de uso, se aplica una transformación del mismo a un diagrama de actividades particular para testing -Modelo Específico para Testing (PSM) en MDD-, construido a través de un Perfil UML definido para modelar actividades de Testing.
3. Se ejecuta luego una transformación modelo a texto –PSM a PSM- que tiene como entrada al diagrama de actividades construido con el perfil de testing mencionado y como salida al caso de prueba. Es decir, ejecutar la transformación genera como resultado, el caso de prueba.

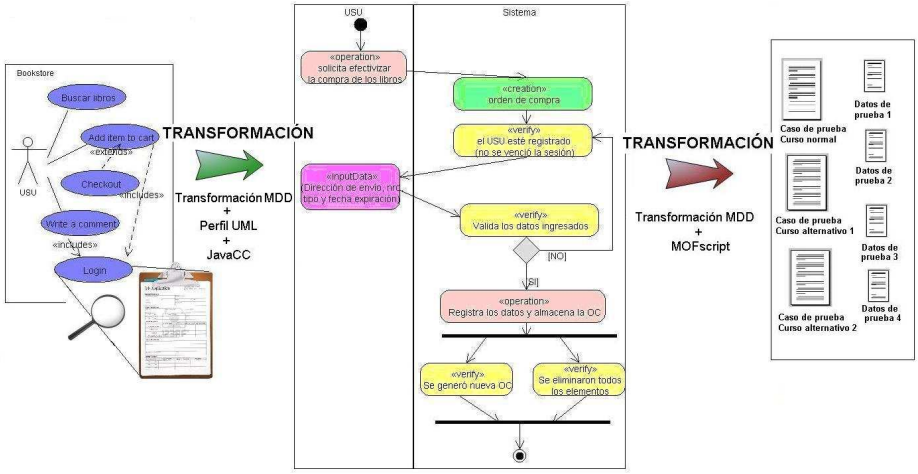


Fig. 1. Pasos del proceso de nuestra propuesta

Nuestra propuesta plantea la generación de casos de prueba en una etapa temprana del ciclo de vida del desarrollo del sistema. Los artefactos generados en los pasos 1 a 3 aquí mencionados, se utilizan una vez definidos los casos de uso con sus respectivas documentaciones, antes de comenzar con el diseño.

3.1 Implementación del Proceso

La propuesta de generación automática de casos de prueba a partir de casos de uso consta de la serie de pasos mencionados al comienzo de esta sección. Para implementar ese proceso, hemos desarrollado algunas herramientas auxiliares que permiten generar las transformaciones mencionadas en los pasos 2 y 3 al inicio de esta sección. A continuación, se detalla cómo se implementó cada uno de esos pasos.

3.1.1 Perfil UML para Modelar Actividades de Testing

Un Diagrama de Actividades en UML no provee el comportamiento necesario para especificar tareas de testing, por lo tanto, se observa la necesidad de enriquecer la metaclase Activity de UML a través de estereotipos, formando así un perfil que defina en forma más explícita, qué tarea realiza cada actividad de testing dentro del caso de

prueba. Es decir, una Actividad de Testing puede representar alguna de estas tareas: la ejecución de una operación; la verificación de condiciones y estados; la creación de objetos; la validación de datos ingresados y la llamada de ejecución a otros casos de uso.
La figura 2 muestra la definición del perfil. Los estereotipos definidos tienen base en la metaclase Activity de UML permitiendo extender su definición. Acompaña a este diagrama de metaclases, una especificación formal de los nuevos estereotipos: restricciones en OCL [15] para expresar reglas de buena formación. La especificación completa de las mismas no se incluye en el presente trabajo por razones de espacio pero pueden ser consultadas en [16].

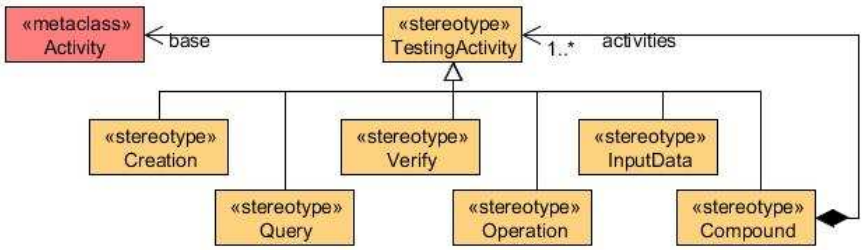


Fig. 2. Perfil UML para actividades de testing con base en Activity

Algunas de las reglas de buena formación para el perfil definido:

STEREOTYPE Query
Base: Activity
Constraints:

1- Una actividad <<query>> contiene tantas variables como operaciones no nulas

```
self.operacionesNoNulas(self.operations)-> size() =
  self.atributosNoNulos(self.attributes)-> size()
```

operacionesNoNulas es una operación que retorna todas las operaciones no nulas de la actividad

```
operacionesNoNulas: Set(BehavioralFeature) ->
                                        Set(BehavioralFeature)
operacionesNoNulas(ops) = ops-> select(each| each notNil)
```

atributosNoNulos es una operación que retorna todos los atributos no nulos de la actividad

```
atributosNoNulos: Set(StructuralFeature) ->
                                    Set(StructuralFeature)
atributosNoNulos(atts) = atts-> select(each| each notNil)
```

STEREOTYPE Compound
Base: Activity
Constraints:

1- Una actividad <<compound>> contiene 1 ó más actividades con estereotipos de la jerarquía <<TestingActivity>>

```
self.activities->size() > 0 and
          self.activities-> includes(<<TestingActivity>>)
```

A continuación se describe brevemente la semántica de cada actividad extendida por el perfil:

- **Actividades <<creation>>.** Especifican la creación de un objeto necesario para la conformación del test. Utilizan un atributo "concept" donde se almacena el objeto a crear.
- **Actividades <<query>>.** Definen una operación de consulta necesaria, en general, para luego realizar validaciones. Una actividad con estereotipo <<query>>debe especificar la operación de consulta. Usa atributos para almacenar las consultas.
- **Actividades <<verify>>.** Describen la validación de condiciones o estados que se deben cumplir. Una actividad <<verify>> debe especificar una premisa de la que se pueda dar un valor de verdad.
- **Actividades <<operation>>.** Especifican la ejecución de una operación determinada. Una actividad <<operation>> debe precisar: la operación (con su contexto, si es necesario) y los parámetros de la operación.
- **Actividades <<inputData>>.** Denotan el ingreso de datos necesarios para realizar alguna funcionalidad y que son necesarios validar. Almacena los parámetros ingresados en atributos propios.
- **Actividades <<compound>>.** Definen **l**a composición de otras actividades (ejecución de otra funcionalidad completa como lo es un caso de uso). Denotan las relaciones de inclusión (<<incluye>>), extensión (<<extend>>) y generalización entre casos de uso.

3.1.2 Transformación del Modelo de Casos de Uso al Diagrama de Actividades de Testing

En este paso se realiza una transformación de la documentación de los casos de uso a diagramas de actividades de testing que especifican las acciones a realizar para verificar la funcionalidad definida, utilizando la facilidad de definición de gramáticas de JavaCC [17] y su consecuente generación de código Java. JavaCC tomará como origen un texto a parsear –la documentación del caso de uso- y tendrá como salida, un archivo .java cuya ejecución crea el diagrama de actividades de testing al que se hace mención. Se usan para ello pre y post-condiciones y pasos de los flujos del caso de uso. Estas condiciones son necesarias para establecer el estado del sistema antes y después de la ejecución de un caso de uso, respectivamente. Los flujos establecen comportamiento que completa la funcionalidad del diagrama de actividades de testing.

3.1.2.1 Reglas de la Transformación

Definimos las siguientes reglas de transformación:

1. El nombre del diagrama de actividad destino, deviene del nombre del caso de uso y la descripción del mismo se agrega en un elemento de anotación.
2. Se definen calles para el diagrama: uno por cada actor y uno por el sistema.
3. Por cada precondición, que especifica un estado del sistema antes de comenzar la ejecución del caso de uso documentado, se crea una actividad de testing <<verify>>.
4. Todas estas verificaciones –precondiciones- se unen en un "join" en el diagrama de actividades, indicando que deben ser verificadas todas las precondiciones antes de continuar con la ejecución de otras actividades.

El flujo normal del caso de uso conformará el conjunto de las siguientes actividades a ejecutar: una actividad por cada paso de ejecución. El tipo de la actividad quedará determinado por el paso en sí. Puede tratarse de la ejecución de una operación (actividades <<operation>> y <<query>>); la verificación de condiciones y estados (actividad <<verify>>); la creación de objetos (actividad <<creation>>); la validación de datos ingresados (actividad <<inputData>>) o la llamada de ejecución a otros casos de uso (actividad <<compound>>).

5. Los flujos alternativos serán notados por medio de bifurcaciones de los diagramas de actividad.
6. Terminada la secuencia de pasos de la documentación del caso de uso, se debe verificar que el estado del sistema sea el esperado. Para ello, se agrega un elemento "fork" del cual parten actividades <<verify>> derivadas de las post-condiciones; es decir en el diagrama destino habrá una verificación por cada postcondición registrada.
7. Finalmente, todas las verificaciones convergen en la actividad de fin de ejecución.

Una aclaración debe hacerse en este punto con respecto a los flujos alternativos. Un paso dentro de un caso de uso puede tener una condición que determina un flujo alternativo. Esa condición, que lleva la ejecución hacia el curso alternativo, puede repetirse un número infinito de veces, pero el caso de prueba se genera para una única vez en que eso pase. Por ejemplo, al ingresar al sistema y ver un formulario de registro, un usuario puede no ingresar su clave y querer ingresar. La definición del caso de uso determina que se verifiquen los datos ingresados por el usuario. Si no son válidos (este es el curso alternativo), se informa que deben ingresarse correctamente los datos y se presenta la página de registro nuevamente. El usuario, podría repetir la acción de no ingresar clave n-1 veces y en la vez n, ingresar una clave válida. El caso de prueba que se generará contemplará sólo el caso: ingreso no válido- ingreso válido.

3.1.3 Definición de la Transformación de Diagrama de Actividades de Testing a Caso de Prueba

A esta altura del proceso contamos con un diagrama de actividades de testing que especifica el comportamiento del test. El último paso es el que nos permite obtener de este diagrama, la definición del caso de prueba. Para ello, hemos definido una transformación en MOFscript [18] que toma como modelo de entrada el diagrama mencionado y genera un modelo en texto: el caso de prueba. MOFscript es un lenguaje implementado con una herramienta del mismo nombre, que permite definir y ejecutar transformaciones modelo a texto basadas en el estándar QVT (Query, View

and Transformation) [19]. Utilizamos este lenguaje por adecuarse a nuestro requerimiento de obtener un texto como salida y por estar actualmente adoptado por la comunidad MDD como lenguaje de transformaciones modelo a texto.

3.1.3.1 Reglas de la Transformación

Para definir la transformación se han pensado las siguientes reglas que permiten obtener el caso de prueba:

1. Por cada camino de ejecución posible, lo que es equivalente a cada escenario especificado, se genera un caso de prueba distinto (bifurcaciones del diagrama de actividades).
2. El nombre del test proviene del nombre del diagrama de actividades acompañado de la partícula "test_" delante.
3. La nota del diagrama será el comentario de los casos de prueba.
4. Por cada actividad, y de acuerdo a su "tipo", se tiene su transformación dentro del test de la siguiente manera:

- **Actividades <<creation>>**: crea un concepto. Es decir, instancia el concepto guardado en su atributo "concept".
- **Actividades <<query>>**: realiza el llamado a la operación de consulta y almacena su respuesta.
- **Actividades <<verify>>**: define aserciones para las premisas de sus atributos
- **Actividades <<operation>>**: realiza el llamado a la operación del sistema
- **Actividades <<inputData>>**: verifica que los atributos sean válidos (no nulos, no fuera de rango, etc.)
- **Actividades <<compound>>**: indica la ejecución de un caso de uso.

5. Para las actividades <<inputData>> se generan pruebas de datos, por cada atributo de la actividad, de forma tal de tener una prueba también para los datos ingresados y de acuerdo a su tipo.

Sólo a modo de ejemplo breve, si se tratase de una cadena (el nombre de usuario), las pruebas podrían verificar si se trata de una cadena vacía, valores cortos y largos, uso de caracteres especiales, cadenas que comiencen o terminen con un espacio vacío. Teniendo previamente definidas estas verificaciones para tipos de datos simples, es que puede automatizarse la generación de datos de prueba.

4 Ejemplo

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

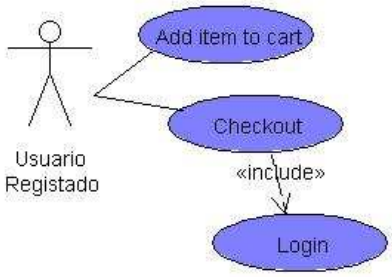
El ejemplo se basa en un sistema de venta de libros por Internet, al estilo Amazon, que permite vender y comprar libros. En forma sucinta, los usuarios que visiten la página del *bookstore* pueden buscar y ver detalles de los libros, novedades, comentarios de usuarios que han comprado un libro. Para comprar libros, los usuarios pueden agregar los ítems que deseen a su carrito de compras, previa identificación en el sistema (nombre de usuario y password). Datos como tarjeta para realizar el pago y dirección de envío serán solicitados por el sistema para efectivizar la compra. Una vez

que el usuario confirme la compra se generará una orden de compra, junto con los datos solicitados, que el sistema almacenará para su posterior envío.

El enunciado completo, que por razones de espacio no detallamos completamente, fue tomado de [4]. En las siguiente subsecciones se muestran los pasos del proceso definido previamente, aplicados a este ejemplo.

4.1 Definición de los Casos de Uso del Ejemplo

Se definen sólo tres (3) casos de uso para no agregar complejidad innecesaria al ejemplo. Asimismo, se acompaña con la documentación extendida al estilo Larman [21] del caso de uso "Checkout", como puede verse en la Figura 3.



Nombre: Checkout
Actor: usuario (USU)
Descripción: El usuario solicita efectivizar la compra de los libros agregados a su carrito de compras.
Precondición: El carrito tiene 1 ó más ítems. El usuario está registrado en el sistema.
Postcondición: Se generó una nueva orden de compra. Se eliminaron todos los elementos del carrito de compras.

Curso Normal
1- El USU solicita efectivizar la compra de los libros de su carrito
2- El sistema recupera el carrito de compras de la sesión de USU
3- El sistema crea una orden de compra
4- El sistema valida que el USU esté registrado (no se venció la sesión)
5- El sistema agrega los ítems del carrito a la orden de compra (OC)
6- El sistema muestra la OC y el precio total a pagar
7- El sistema solicita al cliente dirección de envío y los datos de la tarjeta de crédito
8- El usuario ingresa la dirección de envío, los datos de su tarjeta de crédito y confirma la compra
9- El sistema valida los datos ingresados
10- El sistema registra los datos y almacena la OC

Curso Alternativo- Usuario no logueado
3.1-El usuario no está logueado o se venció su sesión de usuario -> ejecutar CU Login

Curso Alternativo- Datos inválidos
9.1-Los datos ingresados no son válidos -> informar y volver al paso 7

Fig. 3. Casos de uso del ejemplo Bookstore y documentación de Checkout

4.2 Transformación de los Casos de Uso en Diagrama de Actividades de Testing

Este paso consiste en la generación de un diagrama de actividades enriquecido con el perfil para testing definido de forma tal de obtener un diagrama con la funcionalidad descripta en el caso de uso y los elementos necesarios para verificarla. Gráficamente, es claro distinguir las acciones que deben ser verificadas y cómo es la funcionalidad descripta. En este diagrama los analistas pueden intervenir y agregar

actividades para completar aquello que sea necesario, en caso de que se haya omitido por alguna razón, en la documentación del caso de uso.

De acuerdo al tipo de actividad de testing, algunas requieren: parámetros –si se trata de una operación-; especificación de objetos a crear –si se genera un nuevo objeto como parte de la ejecución del caso de uso-; el nombre de un caso de uso –si se trata de una relación <<include>>, <<extend>> o generalización o bien requieren una premisa -en el caso de una validación-, según lo especificado en la subsección 3.1.3.

La figura 4 muestra el diagrama de actividades con el perfil aplicado al ejemplo presentado en la figura 3.

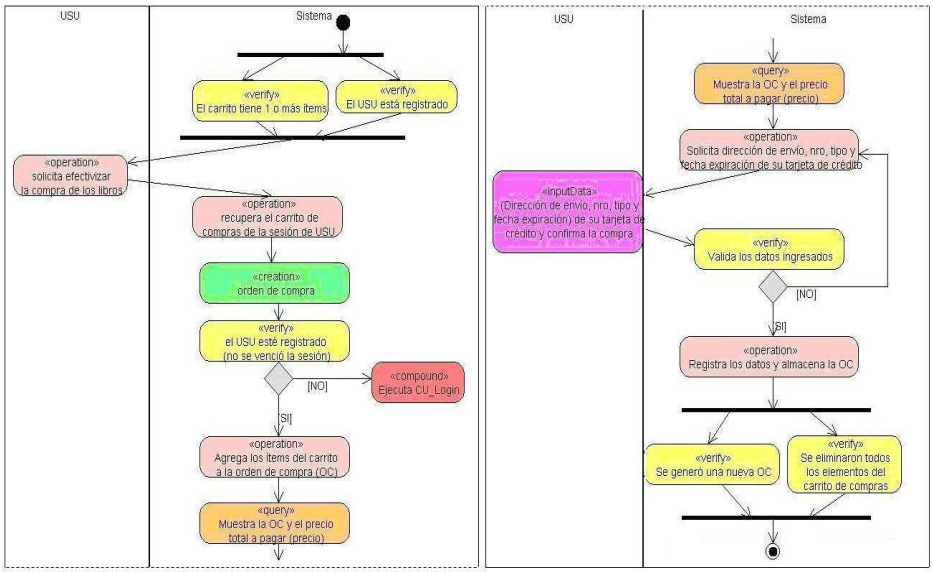


Fig. 4. Diagrama de actividades de testing generado a partir del caso de uso

4.3 Transformación del Diagrama de Actividades de Testing a Caso de Prueba

Una vez obtenido el diagrama de actividades de testing y a través de una transformación modelo a texto, se generan los casos de prueba: uno por el curso normal y uno por cada curso alternativo. La transformación define, para cada tipo de actividad, una sentencia determinada según lo especificado en la subsección 3.1.3.1.

La figura 5 se encuentra dividida en dos (2) partes y muestra: en la columna izquierda, parte de la transformación definida en MOFscript, y en la columna derecha, parte de uno de los casos de prueba generados a partir del curso normal detallado en la conversación del caso de uso. Por razones de espacio, la transformación mencionada sólo se presenta parcialmente en el presente trabajo. En [22], ha sido publicada en forma completa.

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

```
test_Checkout (){
/**
El usuario solicita efectivizar la
 compra de los libros agregados a
 su carrito de compras.
*/
self assert: “El carrito tiene 1 ó
 más ítems”.
self assert:”El usuario está
 registrado en el sistema”.
USU.efectivizarCompraDeLibros;
… }
```

Fig. 5. Transformación en MOFscript y el caso de prueba como modelo de salida

5 Conclusiones y trabajo futuro

En este trabajo hemos presentado un proceso para especificar casos de prueba a partir de casos de uso, incentivando el comienzo del proceso de testing en el momento en que un requerimiento funcional se encuentre identificado y documentado; evitando de esta manera posponer la actividad de testeo para etapas más adelantadas en el desarrollo del sistema cuando corregir errores –o descubrirlos- implican importantes pérdidas de tiempo y esfuerzo.

Específicamente, se definió un perfil UML orientado al testeo para formalizar la información tomada de la documentación del caso de uso. Se definió un proceso para generar diagramas de actividades que especifican el comportamiento del caso de prueba a partir del flujo normal, de los flujos alternativos del caso de uso y de sus pre y post condiciones. El perfil permite transformar estas actividades de testing en sentencias más específicas del caso de prueba. Asimismo, se definió una transformación modelo a texto que genera el caso de prueba y pruebas para los datos.

En lo inmediato, como primera etapa, se espera definir una herramienta de soporte que permita automatizar completamente el proceso, permitiendo aplicar al Modelo de casos de uso, la transformación que genere automáticamente los diagramas de actividades con conceptos de testing (los definidos en el perfil UML) y su consecuente –segunda- transformación a caso de prueba. El editor gráfico será creado como plugin para Eclipse [23] con EMF [24] y GEF [25]. JavaCC se utilizará como herramienta auxiliar para la transformación de la documentación del caso de uso al diagrama de actividades. Como los casos de prueba generados son textuales, otra mejora futura, es que se generen en XML para que puedan luego ser implementados como tests ejecutables en algún lenguaje definido.

Como segunda etapa, se extenderá el proceso de generación de casos de prueba, incluyendo la posibilidad de poder especificar, mediante un diagrama de actividades, el orden de ejecución de los casos de uso del sistema y obtener, de esta manera, un caso de prueba general de todo el sistema.

Otro aporte que podrá proveer la herramienta es la posibilidad de definir trazas entre los artefactos de testing. Es notable que durante el proceso de desarrollo de un sistema, incluso en la temprana etapa del análisis, los artefactos cambian, mutan. Es

decir que puede suceder que algún caso de uso se divida; que dos o más se combinen para formar uno, que dos casos de uso se relacionen o que una relación sea eliminada. Consecuentemente, el diagrama de actividad asociado y el/los caso/s de prueba deberán ser modificados. La definición –y mantenimiento- de trazas permite saber a qué caso de uso corresponde un caso de prueba, permitiendo la generación de pruebas de regresión para asegurar que una modificación realizada (o bien la corrección de un bug), no introduzca nuevos errores. De esta forma, se espera asegurar consistencia entre los modelos de análisis y los de testing.

Referencias

1. Beizer, B.: Software Testing Techniques. Van Nostrand Reinhold (1990)
2. Baker, P. et all: Model-Driven Testing Using the UML Testing Profile. Springer-Verlag (2008)
3. Stahl, T., Völter, M. Model-Driven Software Development. John Wiley & Sons, Ltd. (2006)
4. Pons, C., Giandini, R., Pérez, G.: "Desarrollo de Software Dirigido por Modelos. Conceptos teóricos y su aplicación práctica". EDULP & McGraw-Hill Educación. (2010).
5. Kleppe, A., Warmer J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison- Wesley Longman Publishing Co. (2003)
6. UML 2.0. The Unified Modeling Language Superstructure version 2.0 – OMG Final Adopted Specification. http://www.omg.org (2003).
7. Object Management Group (OMG). http://www.omg.org/
8. Abadía, A., Barisich, J.: Testing Basado en Modelos. Especificación Gráfica y Derivación Automática de Código. Tesis de Grado. Facultad de Informática, Universidad Nacional de La Plata (2009)
9. Hartman, A., Nagin, K.: The AGEDIS Tools for Model Based Testing. UML Satellite Activities (2004)
10. Boyapati,C., Khurshid,S., Marinov,D.: Korat: Automated Testing Based on Java Predicates, MIT Laboratory for Computing Science, MA, USA (2002)
11. Palacios, L.: Perfiles de Testing Aplicados a Modelos de Software. Tesis de Magister. Facultad de Informática, Universidad Nacional de La Plata (2009)
12. Javed, A., Strooper, P., Watson, G.:Automated Generation of Test Cases Using Model-Driven Architecture. 2nd. International Workshop on Automation of Software Test (AST '07), IEEE (2007)
13. FlexMonkey Functional Testing Tool for Adobe Flex Applications. http://www.gorillalogic.com/flexmonkey
14. Selenium Web Application Testing System. http://seleniumhq.org/
15. OCL 2.0. The Object Constraint Language Specification – for UML 2.0, revised by the OMG. http://www.omg.org (2004).
16. https://sol.lifia.info.unlp.edu.ar/~nataliac/pages/profile.htm
17. Java Compiler Compiler. http://javacc.java.net/
18. MOFScript Home page - http://www.eclipse.org/gmt/mofscript/
19. MOF QVT final adopted specification. Technical Report ptc/05-11-01, OMG (2005).
20. https://sol.lifia.info.unlp.edu.ar/~nataliac/pages/example.htm
21. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall, 3era. ed. (2004).
22. https://sol.lifia.info.unlp.edu.ar/~nataliac/pages/transform.htm
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. http://www.eclipse.org/gef/