Revista
Española de

Innovación, Calidad e Ingeniería del Software

REICIS

Volumen 4, No. 4, diciembre, 2008

Web de la editorial: www.ati.es
E-mail: reicis@ati.es
ISSN: 1885-4486

Publicado por la Asociación de Técnicos en Informática

Revista Española de Innovación, Calidad e Ingeniería del Software (REICIS)

Contenidos

REICIS

Propuesta para pruebas dirigidas por modelos usando el perfil de pruebas de UML 2.0

Beatriz Pérez Lamancha
Centro de Ensayos de Software (CES), Instituto de Computación,
Universidad de la República de Uruguay
bperez@fing.edu.uy

Pedro Reales Mateo, Ignacio García-Rodriguez de Guzmán, Macario Polo Usaola
Grupo de Investigación ALARCOS, Departamento de Tecnologías y Sistemas de Información, Universidad de Castilla-La Mancha, España
{pedro.reales,ignacio.grodriguez,macario.polo}@uclm.es

Abstract

This work presents a proposal for testing in Model Driven Engineering environment. The system design models represented in UML are automatically transformed in testing models conforms with the UML Testing Profile. For automatically generate the test cases, an extension for sequence diagram in UML is defined where pre and pos condition are attached to the models. This information is used later for the test oracle generation. The extension uses OCL to define the pre and pos conditions to the test cases.

Key words: model based testing, UML 2.0 testing profile, oracle generation

Resumen

Se presenta una propuesta para pruebas en el contexto de la Ingeniería dirigida por modelos. A partir de los modelos de diseño del sistema en UML, se propone realizar transformaciones a modelos de prueba basados en el perfil de pruebas de UML. Para que la generación de los casos de prueba sea automática, se define una extensión del metamodelo de UML de forma que se puedan anotar los diagramas de secuencia con información que, luego, pueda ser utilizada para generar el oráculo de pruebas. Esta información es anotada en OCL como pre y postcondiciones en el diagrama.

1. Introducción

El perfil de pruebas para UML (*UML Testing Profile*) define un lenguaje para diseñar, visualizar, especificar, analizar, construir y documentar los artefactos de un sistema de pruebas. Extiende UML con los conceptos específicos de pruebas, se basa en el metamodelo de UML y reutiliza su sintaxis definiendo conceptos para: observación del comportamiento de las pruebas y las actividades durante las pruebas, arquitectura de las pruebas, datos de pruebas y tiempo [5].

En este artículo se presenta una propuesta para la generación automática de casos de prueba en el contexto de Ingeniería dirigida por modelos (*Model Driven Engineering*). La metodología se basa en el metamodelo de UML y el perfil de pruebas de UML, realizando transformaciones desde los modelos UML al modelo de pruebas, utilizando como modelo de descripción de comportamiento del sistema el diagrama de secuencias de UML. Para la realización de transformaciones se utiliza el estándar de OMG QVT (Query/View/Transformation).

Dado que para las pruebas dirigidas por modelos es esencial la automatización de todo el proceso, uno de los problemas más importantes al que hay que enfrentarse es la generación automática de los oráculos de las pruebas ya que éstos son dependientes del dominio de la aplicación. Este hecho hace necesario añadir información adicional a los modelos que definen el diseño del sistema para poder generar los oráculos. Se presenta una extensión al metamodelo del diagrama de secuencia de UML para representar la información dependiente del dominio como pre y postcondiciones anotadas usando OCL que servirá posteriormente para generar los oráculos de las pruebas. Los diagramas de secuencia extendidos con pre y postcondiciones son transformados luego en modelos de prueba que son instancias del perfil de pruebas de UML.

El artículo se estructura de la siguiente manera: sección 2 presenta los principales trabajos relacionados en los que se basa esta investigación, en la sección 3 se describe la propuesta para pruebas dirigidas por modelos usando el perfil de pruebas de UML, en la sección 4 se muestra un ejemplo práctico de dicha metodología y por último en la sección 5 se presentan las conclusiones y trabajo a futuro.

2. Trabajos relacionados

En esta sección se describen los principales trabajos relacionados con nuestra investigación que se centra en la definición automática de un oráculo para las pruebas, derivando pruebas en el contexto de la ingeniería dirigida por modelos y generando un modelo de pruebas basado en el perfil de pruebas de UML.

2.1. Oráculo para las pruebas

El oráculo es el mecanismo de que se dota a cada caso de prueba para determinar, tras su ejecución, si el sistema que se está probando supera o no el caso. No existe un mecanismo

que permita describirlo de manera genérica y, en la práctica, se implementa siempre manualmente [6]. Una de las principales dificultades con las que se encuentra la investigación en el área del *testing*, y que, de acuerdo con Bertolino [7], representa un gran obstáculo para avanzar en su automatización, es la descripción del oráculo. Bertolino hace referencia al oráculo ideal, que describe como "un método mágico [sic] que proporciona las salidas para cada caso de prueba, aunque en la práctica se implemente como un motor o heurística que emite un veredicto de paso o fallo sobre las salidas observadas".

El trabajo de Baresi y Young, del año 2001 [6], presenta el estado del arte en cuanto al problema del oráculo. La mayor parte de las propuesta analizadas se refieren a la inserción de instrucciones en los programas que realizan algún de comprobación, como las assert de Java, macros de C o extensiones para lenguajes que permiten, mediante algún tipo de preprocesamiento, embeber en el código la comprobación de restricciones.

En este trabajo definimos una forma de anotar la información requerida para poder construir el oráculo de las pruebas desde los modelos de diseño del sistema, en particular, se extiende el metamodelo de UML para definir dicha información en los diagramas de secuencia del sistema.

2.2. Perfil de Pruebas de UML (UML Testing Profile)

El perfil de pruebas de UML (UML 2.0 Testing Profile, U2TP) define un lenguaje para diseñar, visualizar, especificar, analizar, construir y documentar los artefactos de prueba del sistema. Extiende UML 2.0 con conceptos específicos para las pruebas, dichos conceptos son agrupados en: arquitectura de pruebas, datos de prueba, comportamiento de las pruebas y tiempo de prueba. Al ser un perfil, se integra perfectamente con UML y se basa en el metamodelo de UML, reutilizando su sintaxis [8].

La arquitectura de las pruebas es la definición de todos los conceptos necesarios para realizar las pruebas, se define en el paquete de la arquitectura el contexto de las pruebas y los conceptos necesarios para definir las pruebas. El contexto de prueba (*Test Context*) permite agrupar casos de prueba para describir una configuración de pruebas y definir el control de las pruebas, o sea, el orden requerido para la ejecución de los casos de prueba. La configuración de las pruebas (*Test configuration*) muestra la estructura de comunicación entre los componentes de prueba y el sistema bajo prueba (*system under test ó SUT*). Los componentes de prueba (*Test components*), que son definidos como objetos

dentro del sistema de prueba y pueden comunicarse con el SUT o los otros componentes para llevar a cabo el comportamiento de la prueba.

El comportamiento de la prueba especifica las acciones y evaluaciones necesarias para probar el objetivo de prueba (*test objetive*) que describe qué debe ser probado. El caso de prueba (*test case)* es una operación de un contexto de prueba que especifica cómo un conjunto de componentes cooperan con el SUT para realizar el objetivo de prueba. Un veredicto (*verdict*) es una enumeración predefinida que especifica los posibles resultados de las pruebas (pasa, falla, error, inconclusa), las acciones de validación (*validation action*) son realizadas por el componente de prueba para indicar que el arbitro (*arbiter*) esta informado de los resultados de las pruebas. El árbitro evalúa los resultados de las pruebas realizadas por los distintos objetos dentro del sistema de prueba para determinar un veredicto global para el caso de prueba o contexto. Los tiempos (*timers*) son usados para manipular y controlar el comportamiento de las pruebas y asegurarse que los casos de prueba terminan [8]. Otro aspecto importante de las especificación es la definición y codificación de los datos de prueba, para esto U2TP soporta *wildcards*, conjuntos de datos (*data pools*), particiones de datos (*data partitions*), selectores (*data selectors*) y reglas de codificación [8].

2.3. Ingeniería dirigida por modelos (Model Driven Engineering)

La ingeniería dirigida por modelos (Model-Driven Engineering ó MDE) combina: (i) Lenguajes de modelado específicos del dominio (Domain-specific modeling languages ó DSML), que son descritos usando metamodelos que definen las relaciones entre los conceptos en un dominio y especifican en forma precisa la semántica y las restricciones asociadas con esos conceptos del dominio; (ii) motores de transformación y generadores que analizan ciertos aspectos de los modelos y que sintetizan varios tipos de artefactos, tales como código fuente, simulación de entradas, descripciones de distribución XML o representaciones alternativas de modelado [1]. Estándares tales como *Query/Views/Transformations (QVT)* y *Meta Object Facility(MOF)* han sido definidos como parte del estándar de OMG de Arquitectura dirigida por modelos *(Model-Driven Architecture ó MDA)*, los cuales pueden ser utilizados como la base para herramientas MDE específicas del dominio[1].

El enfoque MDA utiliza la idea de separar la especificación del sistema de los detalles en que el sistema utiliza las capacidades de su plataforma. MDA provee un enfoque para especificar un sistema independiente de la plataforma que lo soporta, especificar plataformas, elegir una plataforma particular para el sistema y transformar la especificación del sistema en una de las plataformas particulares [2]. MDA especifica tres vistas del sistema: 1) Vista independiente de la computación (CIM): focaliza en los requisitos del sistema, los detalles de estructura y procesamiento del sistema no son tenidos en cuenta. 2) Vista independiente de la plataforma (PIM): focaliza en la operación del sistema sin tener en cuenta detalles de una plataforma particular. 3) Vista específica de la plataforma (PSM): Combina la vista independiente de la plataforma con el objetivo en los detalles de su uso en un plataforma específica [2].

La transformación de modelos es el proceso de convertir un modelo en otro para el mismo sistema. Se utiliza un lenguaje para describir dicha transformación [2]. Una transformación establece un conjunto de reglas que describen cómo un modelo expresado en un lenguaje origen puede ser transformado en un modelo en un lenguaje destino. Las categorías principales de transformaciones en MDA son los refinamientos (mappings) verticales y horizontales. Los refinamientos verticales relacionan modelos del sistema situados en distintos niveles de abstracción, como transformaciones de PIM a PSM ó de PSM a PIM. Los refinamientos horizontales, relacionan o integran modelos que cubren distintos aspectos en un mismo nivel de abstracción, mantienen la consistencia entre distintos niveles y garantizan que la información modelada sobre una entidad del sistema es consistente con lo que se dice sobre ella en cualquier otra especificación situada en el mismo nivel de abstracción [3].

Para realizar las transformaciones, OMG ha definido un estándar para la definición de transformaciones llamado *QVT (Query/View/Transformation)* [4].

2.4. Pruebas dirigidas por modelos (Model Driven Testing)

Puede aplicarse el perfil de pruebas de UML a un modelo diseñado con UML para derivar un modelo de pruebas. Los niveles de abstracciones de MDA pueden aplicarse al modelado de las pruebas, los modelos de pruebas pueden ser independientes de la plataforma o específicos de la plataforma en forma previa a generar el código de pruebas ejecutable [9]. En la Figura 1(a) se muestra el enfoque propuesto por Dai siguiendo el

enfoque MDA para los modelos de pruebas. En la Figura 1(b) se muestran las transformaciones para las pruebas basadas en los metamodelos. El metamodelo fuente es el de UML y el metamodelo destino es el metamodelo de UML Testing Profile (U2TP). La transformación entre modelos es especificada por reglas de acuerdo a la especificación QVT [9]. Esta propuesta es teórica y no ha sido desarrollada.

En el libro de Baker [8] se definen modelos de prueba utilizando el perfil de pruebas de UML, pero las transformaciones se realizan manualmente, sin usar lenguaje específico.

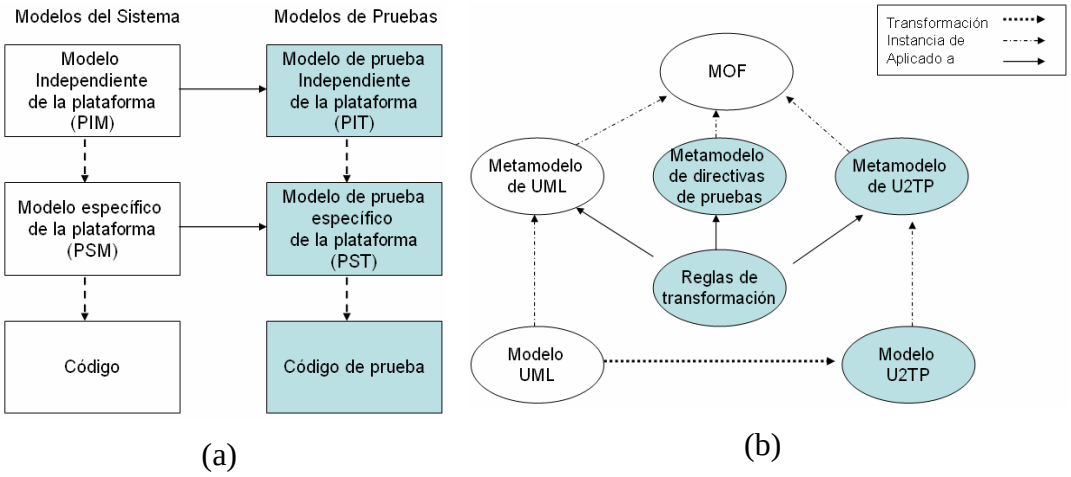

Figura 1. Transformación de modelos de pruebas[9]

3. Propuesta para pruebas dirigidas por modelos usando el perfil de pruebas de UML

Para definir nuestro método de derivación de pruebas dirigido por modelos, nos basamos en la propuesta de Dai [9] (ver sección 2.3).

En las pruebas dirigidas por modelos resulta imprescindible poder generar casos de prueba completos en forma automática. Esto implica representar la información dependiente del dominio para las pruebas en los modelos UML y poder transformarla luego en el oráculo de pruebas en los distintos modelos de prueba.

La propuesta presentada en este artículo añade una pequeña extensión al metamodelo de UML (ver Figura 2) para agregar la información requerida para el oráculo de las pruebas en los diagramas de secuencia. Dado que un diagrama de secuencia se corresponde con un escenario relevante que debe ser probado, el diagrama se anota de forma que incluya información sobre el resultado esperado como resultado de la ejecución de dicho escenario.

En la Figura 2 se muestra la extensión del metamodelo de UML definida, donde en la clase *InteractionFragment* se le agrega una *Constraint* de OCL, y dos relaciones, una que representa la precondición del diagrama de secuencia y otra que representa sus postcondiciones. Se agrega también una restricción que indica que dichas pre y postcondiciones no se aplican al *InteractionFragment* de tipo *StateInvariant*, ya que éste representa invariantes donde las condiciones no cambian.

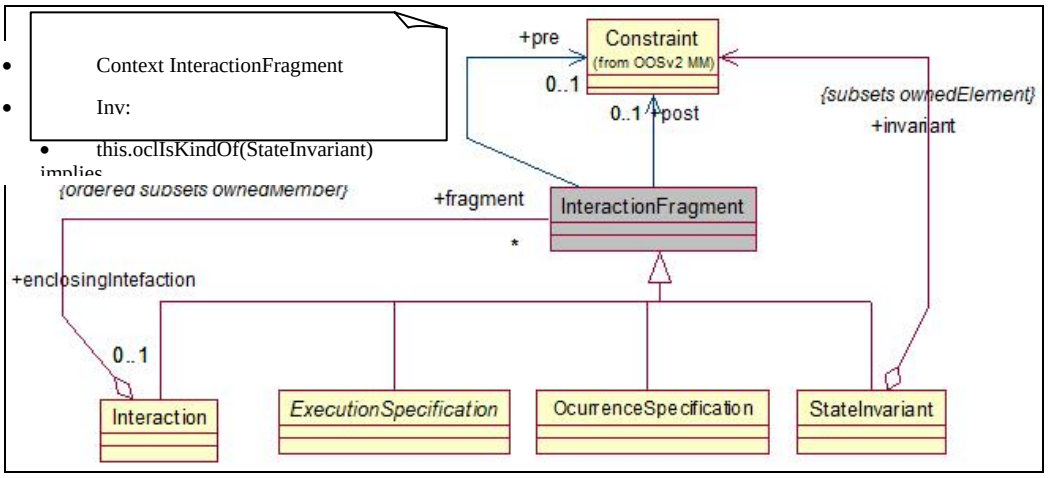

Figura 2. Extensión al metamodelo de UML para representar pre y postcondiciones para las pruebas

Un *InteractionFragment* es un trozo de una interacción, por lo que la semántica de nuestra extensión es que cualquier segmento de una interacción puede tener una pre y poscondición a ser utilizada como oráculo para las pruebas. Esto nos permite poder derivar casos de prueba a distintos niveles: unitario, de integración ó funcionales.

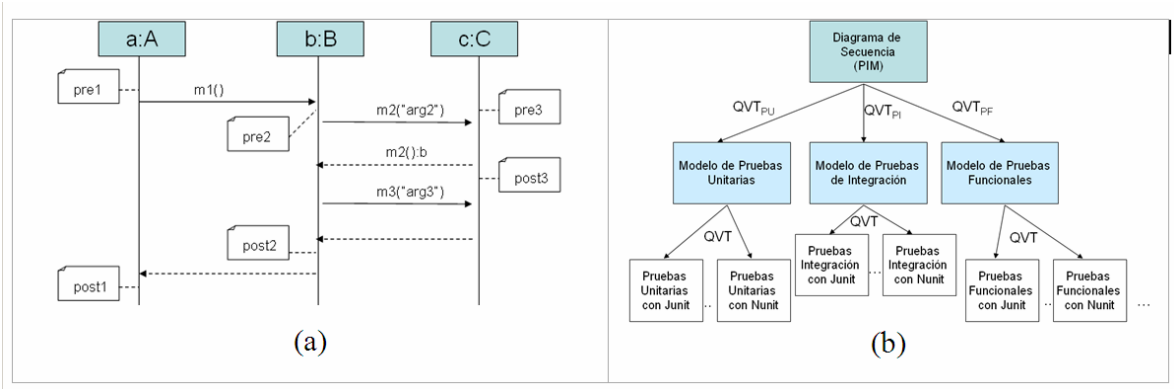

Figura 3. Información para el oráculo según el nivel de prueba

En la Figura 3 (a) se pueden observar los distintos puntos donde se pueden poner pre y postcondiciones. Por ejemplo, la pareja pre1 y pos 1 tienen la información necesaria para construir el oráculo para todo el diagrama, lo que se conoce como una prueba funcional. La

pareja pre2 y pos2 dan la información necesaria para construir el oráculo para probar la interacción entre B y C, mientras que la pareja pre3 y pos3 nos permiten realizar pruebas unitarias de C. La Figura 3 (b) muestra las transformaciones posibles desde el diagrama de secuencia extendido a pruebas unitarias, de integración y funcionales. Siguiendo un enfoque MDA, primero se obtiene el modelo de pruebas en el nivel deseado y luego dicho modelo es refinado según la plataforma, en caso de que el lenguaje sea Java, podríamos derivar casos de prueba en JUnit. Las transformaciones serán realizadas utilizando QVT.

Para poder anotar las pre y postcondiciones en OCL es necesario conocer la información sobre las clases del sistema. A partir del diagrama de secuencia y el diagrama de clases se pueden realizar las transformaciones al modelo de pruebas, como se muestra en la Figura 4.

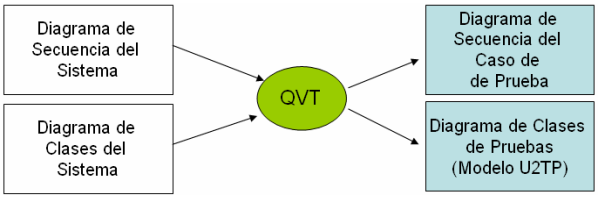

Figura 4. Transformaciones desde los diagramas de diseño del sistema al modelo de pruebas

3.1. Transformación del Diagrama de Secuencia extendido al Modelo de Pruebas

A partir del diagrama de clases, que representa la arquitectura del sistema, y de los diagramas de secuencia extendidos con pre y postcondiciones, que representan el comportamiento, se realizan las transformaciones para llegar al modelo de pruebas. Si bien como se explicó anteriormente podrían generarse distintos niveles de prueba, en esta sección nos enfocaremos en generar las transformaciones desde el diagrama de secuencia al modelo de pruebas funcionales.

El primer paso es construir la arquitectura del modelo de pruebas a partir de la arquitectura del sistema y de los modelos que representan el comportamiento (en nuestro caso, diagramas de secuencia). Los elementos de la arquitectura se exponen en la Tabla 1.

Elemento U2TP	Transformación realizada desde los modelos de clases y secuencia del sistema
TestContext	Para cada funcionalidad perteneciente al sistema aparece un TestContext con los casos de prueba de esa funcionalidad.
TestCase	Para cada diagrama de secuencia que representa un escenario de ejecución de la funcionalidad aparece un testCase. Se representa como una operación del TestContext.
TestObjective	Cada test case tendrá asociado el diagrama de secuencia que prueba.
Deployment	En cada TestContext aparecerá un Deployment, es la operación encargada de inicializar el sistema para las pruebas.
TestComponent	Por cada actor aparecerá un TestComponent que se encargará de interactuar con el SUT para realizar el test case.
Arbiter	Cada TestContext estará relacionado con un Arbiter. El Arbiter tendrá dos métodos, uno para almacenar resultados de casos de prueba y otro para leerlos.
DataPool	Cada TestContext estará relacionado con un DataPool.
Data Partition	Almacena datos relacionados con un *test case* y se relaciona con el *DataPool*
DataSelector	Aparecerá como una operación en un DataPool y habrá uno por cada testCase. Al ejecutarse deberá tener en cuenta la precondición asociada al diagrama de secuencia del sistema para generar los datos de prueba correctos para el caso de prueba al que esté ligado el DataSelector.

Tabla 1. Arquitectura del modelo de pruebas

Elemento U2TP	Transformación realizada desde los modelos de clases y secuencia del sistema
Diagrama de secuencia	Aparece un diagrama de secuencia nuevo por cada TestCase contenido en la arquitectura de las pruebas.
Línea de vida del TestContext	Cada diagrama de secuencias aparece una línea de vida que referencia a la clase TestContext que contiene el caso de prueba cuyo comportamiento está definiendo el diagrama.
Línea de vida del Arbiter	Cada diagrama de secuencias aparece una línea de vida que referencia a la clase Arbiter que dará el veredicto del test case.
Línea de vida del DataPool	Cada diagrama de secuencias aparece una línea de vida que referencia a la clase DataPool con los datos a usar para el test case.
Líneas de vida de los TestComponent	Cada diagrama de secuencias aparecen líneas de vida que se corresponden con los TestComponent que referencia los actores que intervienen en el test case.
Líneas de vida de los SUTs	Cada diagrama de secuencias aparecen líneas de vida que se corresponden con los SUTs, clases del sistema, que interactúan directamente con los actores.

Tabla 2. Comportamiento del modelo de pruebas (diagrama de secuencia de pruebas)

Una vez construida la arquitectura de sistema de pruebas, hay que generar el comportamiento asociado a cada uno de los casos de prueba. Para ello se generan una serie de diagramas de secuencia, de manera que cada uno modela el comportamiento de un caso de prueba. En la Tabla 2 se muestran los elementos que aparecen en cada uno de los diagramas de secuencia.

Una vez definidos los elementos que del diagrama de secuencia, se deben realizar las transformaciones para obtener los mensajes. Estos mensajes tienen un orden y, por tanto, las transformaciones que se muestran en la Tabla 3 han de realizarse en el orden especificado.

Elementos U2TP	Transformación realizada desde los modelos de clases y secuencia del sistema
Mensajes de inicio TestComponents	El TestContext le envía un mensaje a los TestComponents encargados de realizar el test case.
Mensaje para obtener los datos de prueba	Aparece un mensaje desde el TestContext hasta el DataPool o Data Partition invocando al DataSelector.
Mensajes de interacción con el SUT	Los mensajes que partían o llegaban a un actor, ahora parten o llegan al TestComponent que representa a ese actor hacia el SUT.
Mensajes de actualización del veredicto	Son los mensajes que van desde el TestComponent correspondiente hasta el Arbiter, para actualizar el Verdit del caso de prueba. Justo antes de ejecutar esta llamada cada TestComponent ha de calcular el Verdict correcto en función de la postcondición asociada al diagrama de secuencia del sistema.

Tabla 3. Mensajes del diagrama de secuencia de pruebas

3.1.1 Transformaciones en QVT

En QVT una transformación genera un modelo a partir de otro haciendo uso de relaciones. Por lo cual, la transformación en QVT que obtendrá el modelo de pruebas hará uso de relaciones, las cuales implementarán las diferentes transformaciones expuestas en las tablas anteriores.

A continuación se muestran las dos primeras transformaciones de la Tabla 1 diseñadas con QVT. Dada la complejidad del código QVT, se ha empleado la sintaxis gráfica propia del lenguaje para representar las transformaciones.

La relación QVT de Figura 5 genera los "*testContext*" necesarios para la arquitectura de pruebas, uno por cada funcionalidad. Mediante esta representación podemos

ver que la relación esta compuesta por dos dominios, uno de entrada (*useCase*), el cual tiene un conjunto de comportamientos asociados (*ownedBehavior*) y otro de salida (*classContext*). Así, para cada caso de uso del sistema se va a generar una clase estereotipada como <<TestContext>> que se encargará de probar la funcionalidad modelada. Además, en el campo "where" de la transformación se buscan todos los "*behaviors*" asociados al caso de uso que sean del subtipo "*Interaction*", es decir, que sean un diagrama de secuencias, y se ejecuta la transformación de generación de casos de prueba para generar, por cada diagrama de secuencias, un caso de prueba.

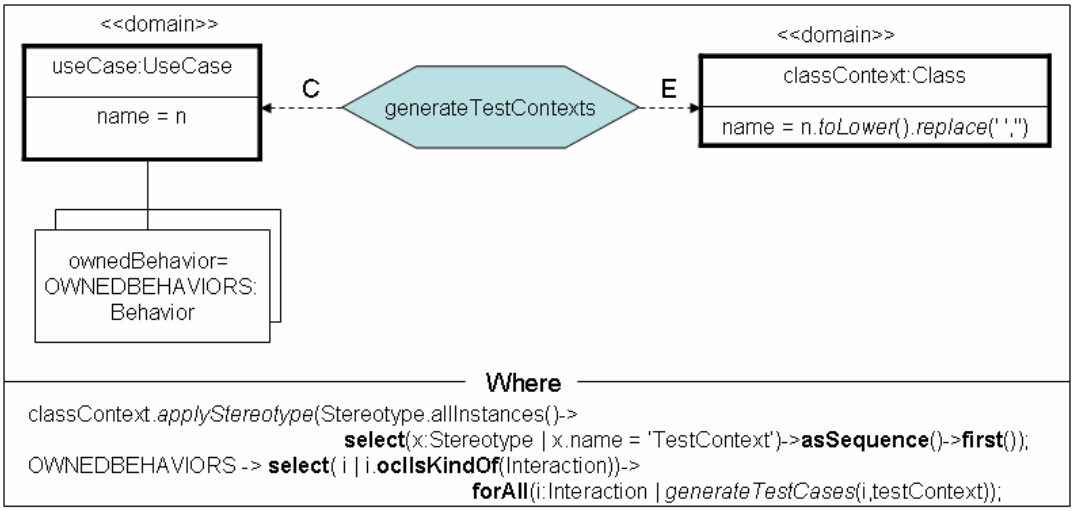

Figura 5 - Transformación para generar los diferentes "TestContext"

La Figura 6 muestra la representación visual de la relación QVT que implementa la segunda transformación de la Tabla 1, la cual se encarga de generar los test cases. Se puede ver que existe un dominio de entrada, un diagrama de secuencias, y dos dominios de salida, el "*testContext*" que contendrá el "*testCase*", el segundo dominio de salida, que ejecutará la prueba referente al diagrama de secuencias de entrada.

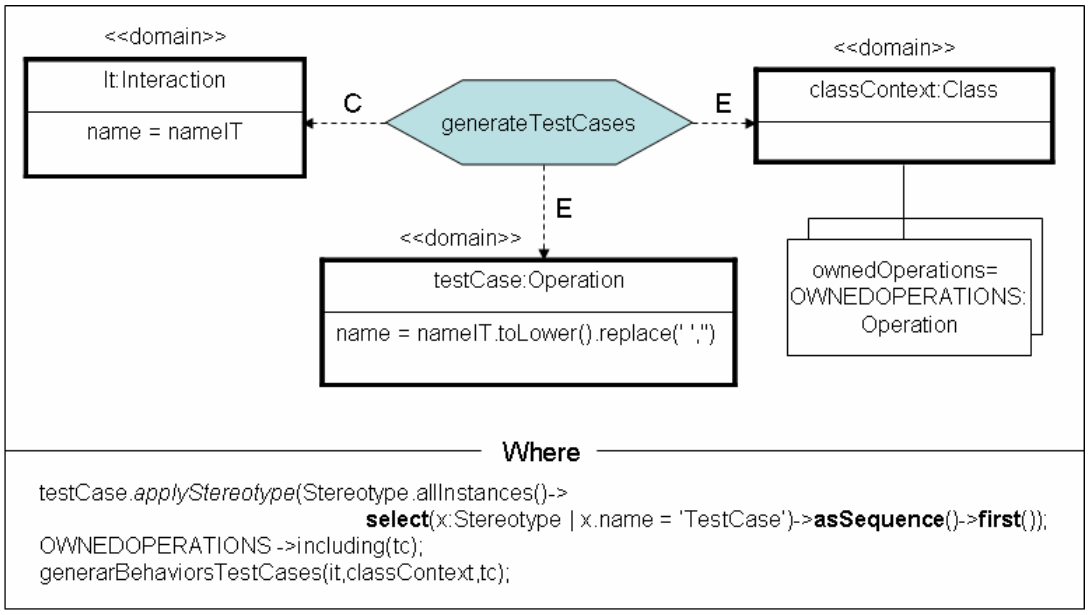

Figura 6 - Transformación para generar los diferentes "TestCase"

En el campo "where" de esta relación se hace uso de otra relación, *generarBehaviorsTestCases*, que se encarga, haciendo uso a su vez de otras relaciones QVT, de generar los diferentes elementos que modelan el comportamiento de caso de prueba (Tabla 2 y Tabla 3).

4. Ejemplo de aplicación de pruebas dirigidas por modelos

En esta sección se presenta un ejemplo de la metodología definida en la sección 3 de este artículo. El ejemplo modela el escenario de ejecución principal de la funcionalidad de autenticación de un usuario en el sistema. En un entorno real aparecerían múltiples funcionalidades con múltiples escenario, pero para este ejemplo nos centraremos en un escenario concreto.

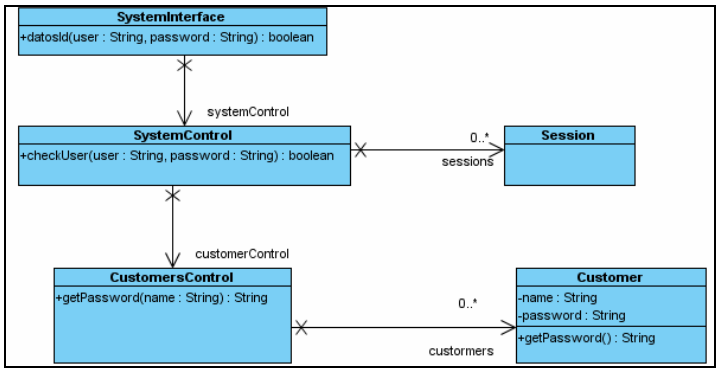

Figura 7. Diagrama de clases del sistema

La Figura 7 muestra el diagrama de clases con la arquitectura que implementa esta funcionalidad y la Figura 8 representa el diagrama de secuencia del comportamiento del sistema durante el escenario de ejecución principal. Las pre y postcondiciones que extienden el diagrama para este ejemplo están definidas en OCL y añadidas al diagrama de manera visual mediante una nota.

Obsérvese que, el contexto de la restricción OCL que se ha añadido al diagrama está centrado en un elemento del propio diagrama, de manera que esta restricción solo es aplicable en este escenario de ejecución.

Aplicando las transformaciones definidas en la Tabla 1 a estos dos elementos obtenemos la arquitectura de pruebas modelada en la

Figura 9. Obsérvese que las diferentes clases están esteriotipadas según el metamodelo de pruebas de U2TP y que diversos elementos se nombran de manera que hacen referencia a lo que se está probando, tanto la funcionalidad, como su escenario de ejecución. También se pueden observar las diferentes relaciones entre el sistema de pruebas y el sistema.

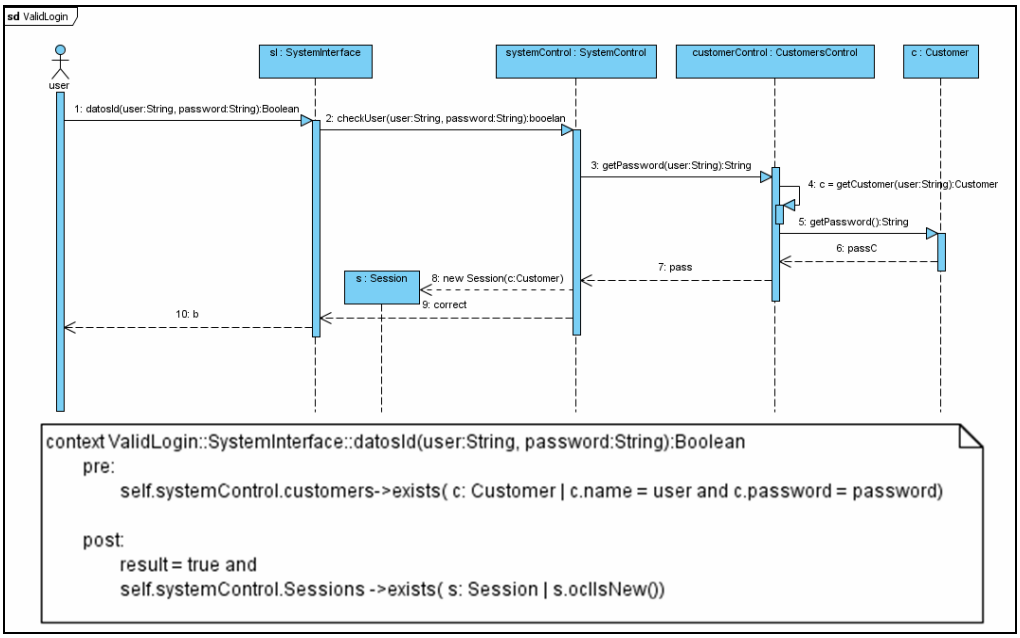

Figura 8. Diagrama de secuencia del sistema con pre y postcondiciones en OCL para las pruebas

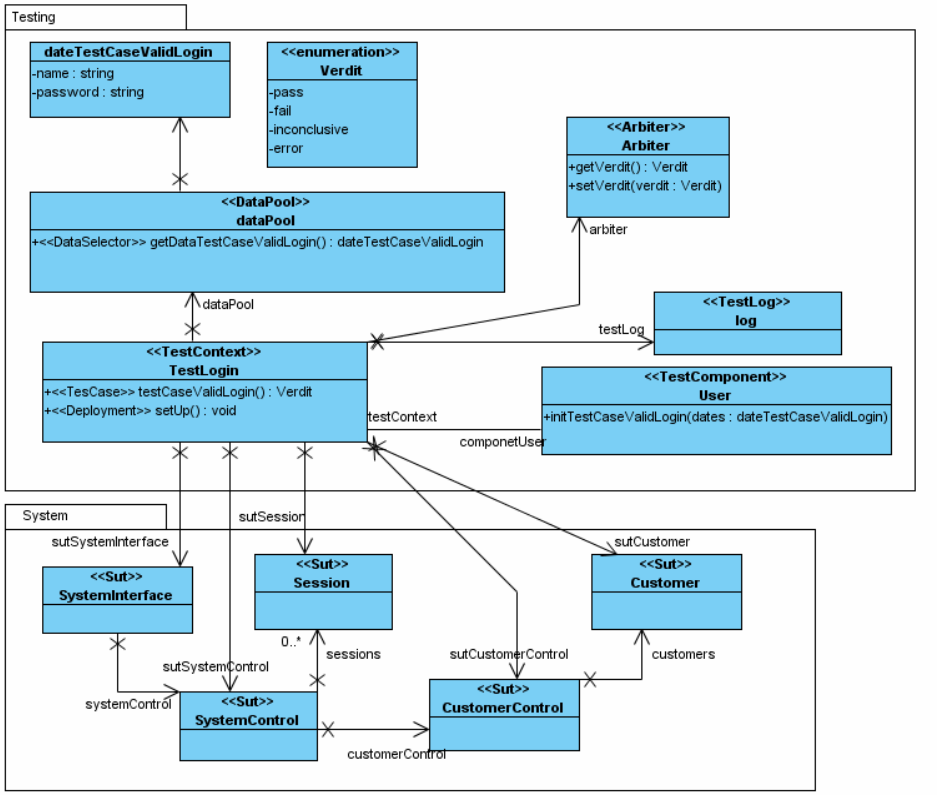

Figura 9. Modelo de clases del sistema de pruebas

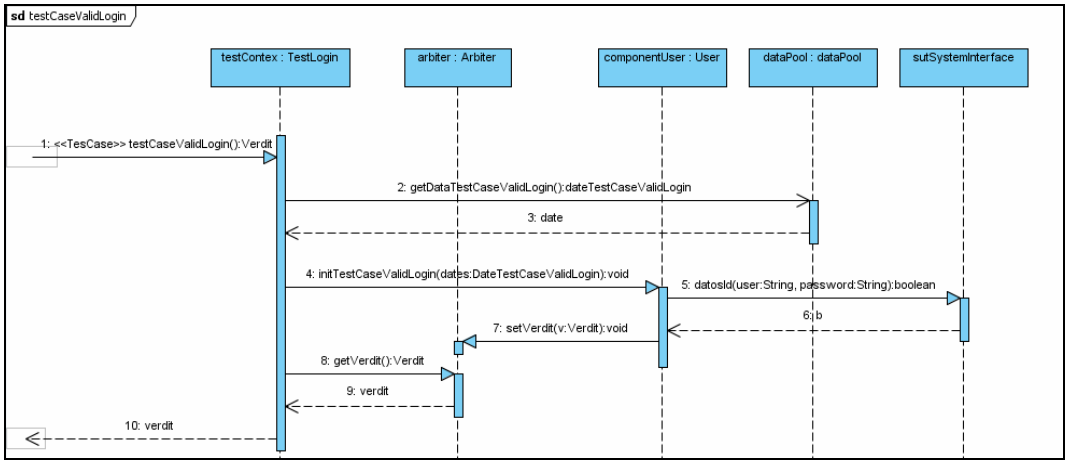

Figura 10. Diagrama de secuencia del caso de prueba

Una vez obtenida la arquitectura de pruebas, interesa modelar el comportamiento para el caso de prueba de la funcionalidad “Autenticación de un usuario al sistema”. Aplicando las transformaciones definidas en las Tabla 2 y de la

se obtienen los diferentes comportamientos de los casos de prueba, en este caso solamente para el caso de prueba “testCaseValidLogin()”. El diagrama mostrado en la

Figura 10 muestra este comportamiento. De esta forma se genera para el escenario principal de la funcionalidad de autenticación, el caso de prueba en forma automática.

5. Conclusiones y trabajo a futuro

El desarrollo de esta investigación se centra principalmente en la generación automática de pruebas siguiendo un enfoque MDA. La idea central es que los modelos de diseño del sistema como el diagrama de secuencia y el diagrama de clases, que corresponden al PIM se transforman mediante QVT en un modelo de pruebas. Este modelo de pruebas puede ser refinado luego según la plataforma del sistema, mediante transformaciones QVT, en un modelo *JUnit* por ejemplo.

Para lograr una automatización completa del proceso, se requiere contar con una especificación del oráculo de las pruebas, del cual se puedan derivar los datos de prueba en forma automática. Para esto se ha definido una extensión del metamodelo de UML donde se expresan las pre y pos condiciones de cada diagrama de secuencia mediante OCL. Actualmente se han realizado las transformaciones QVT desde el modelo del sistema al modelo de prueba. En una segunda etapa se realizarán las transformaciones desde el modelo de pruebas a un modelo dependiente de la plataforma (por ejemplo, JUnit).

Dentro del trabajo a futuro se encuentra llevar los resultados de esta investigación al ámbito de las pruebas en líneas de producto de software. La importancia de la trazabilidad entre los distintos modelos que definen la línea hace pensar que un enfoque combinado entre MDA y líneas de producto es factible. La metodología propuesta en este artículo puede ser extendida para tratar la variabilidad a nivel de los modelos de diseño del sistema y que esta variabilidad sea transformada a los modelos de prueba. De esta forma se obtendrán los casos de prueba de cada producto cuando la variabilidad sea resuelta.

Agradecimientos

Este trabajo ha sido parcialmente soportado por el proyecto PRALIN (Junta de Comunidades de Castilla-La Mancha, PAC08-0121-1374)

Referencias

[1] Schmidt, D., "Model-Driven Engineering", *IEEE Computer*, vol. 39 nº 2, pp. 25-31, 2006.

[2] Miller, J. and J. Mukerji, *MDA Guide Version 1.0. 1*, Object Management Group, 2003.

[3] Moreno N., R.J., Romero R., Vallecillo A., *Desarrollo de Software dirigido por modelos, in Fábricas de Software: experiencias, tecnologías y organización*, Ra-Ma, 2007.

[4] OMG, *Meta Object Facility (MOF) 2.0 Query/View/Transformation*, v1.0, OMG, 2008.

[5] OMG, *UML testing profile Version 1.0,* OMG, 2005.

[6] Baresi, L. and M. Young, *Test oracles*, Dept. of Computer and Information Science, Univ. of Oregon, 2001.

[7] Bertolino, A. "Software Testing Research: Achievements, Challenges, Dreams" in *International Conference on Software Engineering. 2007*.

[8] Baker, P., et al., *Model-Driven Testing: Using the UML Testing Profile*, Springer, 2007.

[9] Dai, Z., "Model-Driven Testing with UML 2.0" En *Second European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations. Canterbury, England*, 2004.