Modelado y Derivación Automática de Código para Pruebas de Software

Fernando Palacios, Claudia Pons

LIFIA, Facultad de Informática, Universidad Nacional de La Plata
Buenos Aires, Argentina
{fernando.palacios, cpons}@lifia.sol.edu.ar

Resumen. El Perfil de Pruebas UML (en inglés UML 2.0 Testing Profile, U2TP) permite especificar y modelar dominios de pruebas posibilitando además derivar dichos modelos a código ejecutable en forma automática. En este artículo presentamos una propuesta para automatizar las transformaciones de modelos de pruebas estructurales y de comportamiento a código JUnit dentro del contexto de Pruebas de software Dirigidas por Modelos (en inglés Model-Driven Testing, MDT). Para lograrlo, extendemos el metamodelo UML 2 para definir un lenguaje de modelado para dominios de prueba. Este lenguaje permite agregar anotaciones sobre los diagramas UML con información adicional que luego es utilizada para generar los casos de pruebas correspondiente. Además, presentamos las reglas formales de transformación de modelos U2TP a código de pruebas JUnit. Estas reglas están implementadas a través de un componente de software en forma de plug-in de Eclipse utilizando MOFScript.

Palabras claves: Perfil de Pruebas UML 2.0, Testing, Diseño de Pruebas, Prueba de Caja Negra, Transformación de Modelos, MOF, MDA, MDT, JUnit, Diagrama de Clase, Diagrama de Secuencia.

1 Introducción

A medida que el software se torna más complejo, se torna necesario el desarrollo de nuevos paradigmas para su construcción. Uno de estos nuevos paradigmas es el Desarrollo de software Dirigido por Modelos (en inglés Model Driven software Development, MDD), el cual ya ha demostrado impacto en reducir el tiempo y esfuerzo insumidos en el proceso de desarrollo del software, mientras que mejora la calidad del producto final.

Este paradigma propone un proceso guiado por modelos que van desde los más abstractos (en inglés Platform Independent Model, PIM) a los más concretos (en inglés Platform Specific Model, PSM) realizando transformaciones y/o refinamientos sucesivos que permitan llegar al código aplicando una última transformación.

Los modelos en MDD se construyen utilizando diversos lenguajes de modelado, especialmente UML, que ha sido declarado estándar por el OMG [1] y que además ha logrado buena aceptación en la industria del software.

Sin embargo, el desarrollo de sistemas de alta calidad requiere no solo de un proceso de desarrollo riguroso, sino también de un proceso sistemático de pruebas para evaluar el producto final. Este proceso consiste en ejercitar un producto para verificar que satisface los requerimientos e identificar diferencias entre el comportamiento real y el comportamiento esperado (IEEE Standard for Software Test Documentation, 1983). De esta forma el proceso de prueba permite la detección de errores y fallas en la implementación garantizando así la calidad del producto final.

Dentro del contexto de MDD, las Pruebas de software Dirigidas por Modelos [2] [3] (en inglés Model-Driven Testing, MDT) son una forma de prueba de caja negra [4] que utiliza modelos estructurales y de comportamiento para automatizar el proceso de generación de prueba.

Dado que UML no provee medios adecuados para el diseño y desarrollo de las pruebas, el OMG impulsó recientemente una iniciativa para la creación de un lenguaje específico para este fin. Dicho lenguaje fue definido con mecanismos de perfiles basado en UML y se denomina Perfil de Pruebas UML [5] (en inglés UML 2.0 Testing Profile, U2TP). Permite diseñar los artefactos de los sistemas de pruebas e identificar los conceptos esenciales del dominio en cuestión adaptados a plataformas tecnológicas y a dominios específicos. La especificación del U2TP proporciona además un marco formal para la definición de un modelo de prueba bajo la propuesta de caja negra que incluye las reglas que se deben aplicar para transformar dicho modelo a código ejecutable.

U2TP combina e integra el desarrollo del sistema y el desarrollo de las pruebas vía UML, brindando un lenguaje de comunicación común tanto para los desarrolladores como para los clientes. Permite además razonar acerca de la calidad y la cobertura de las pruebas.

En este artículo se presenta una propuesta para realizar la generación automática de casos de prueba a partir de modelos de software expresados en el lenguaje UML 2 y su perfil U2TP. Las transformaciones de los modelos UML de pruebas consideran los diagramas estructurales y de comportamiento como entrada para la generación a código JUnit [6] dentro del contexto de MDT. Para lograr que la generación de los casos de prueba sea automática, definimos el lenguaje para modelar dominios de prueba basada en U2TP. Esta extensión al metamodelo UML 2 se utiliza para agregar anotaciones sobre los diagramas de clases y secuencia con información que pueda ser utilizada para generar el oráculo de pruebas. Dentro de la propuesta, presentamos las reglas formales de transformación de modelos U2TP a código de testing JUnit. Además, presentamos la implementación de un componente de software en forma de plug-in de Eclipse [7] que traduce a texto los modelos de pruebas generando así el código JUnit. Para lograr una mayor precisión en la traducción del esqueleto del código luego de transformar los diagramas de pruebas de clases y secuencia, agregamos una nueva facilidad para incorporar comportamiento dentro de los métodos traducidos que representamos con el uso del framework EasyMock [8].

Este artículo está organizado de la siguiente manera; en la sección 2 se introducen los conceptos de U2TP, los frameworks y tecnologías utilizadas en el desarrollo del trabajo. La sección 3 describe nuestra propuesta para transformar modelos de pruebas U2TP a código ejecutable; la sección 4 muestra un ejemplo aplicado utilizando el lenguaje creado para especificar los casos de pruebas y la generación de código en forma automática realizada por la herramienta desarrollada. La sección 5 describe los

trabajos relacionados, y finalmente la sección 6 expone las conclusiones finales donde también se citan futuros trabajos.

2 Desarrollo de Pruebas Dirigidas por Modelos

En esta sección presentamos el Perfil de Pruebas UML 2.0, luego describimos los frameworks JUnit y EasyMock que serán utilizados para la elaboración de pruebas y por último introducimos el lenguaje de transformación de modelos a texto denominado MOFScript.

2.1 El Perfil de Pruebas UML 2

El Perfil de Pruebas UML 2 [5] es un lenguaje de modelado que se ha implementado utilizando el mecanismo de perfiles de UML 2 para diseñar, visualizar, especificar, analizar, construir y documentar los artefactos de los sistemas de pruebas. Este perfil proporciona un marco formal para la definición de un modelo de prueba bajo el acercamiento de caja negra [4]. Aplicando este enfoque las pruebas se derivan de la especificación del artefacto que se somete a la verificación, donde éste es una *caja negra* cuyo comportamiento sólo se puede determinar estudiando sus entradas y salidas.

U2TP se organiza en cuatro grupos lógicos que cubren los aspectos de pruebas de:

1. Arquitectura (en inglés *test architecture*), que define un conjunto de conceptos agregados a los estructurales de UML 2 permitiendo especificar los aspectos estructurales de un contexto de pruebas.
2. Comportamiento (en inglés *test behavior*), agrega nuevos conceptos adicionales a los descriptos en UML 2 para especificar el comportamiento de las pruebas, sus objetivos y la evaluación de los Sistema Bajo Pruebas (en inglés *System Under Test*, *SUT*).
3. Datos (en inglés *test data*), estas pruebas utilizan símbolos especiales (en inglés *wildcards*) para manejar eventos inesperados, o eventos con distintos valores.
4. Tiempo (en inglés *test time*), U2TP agrega un conjunto adicional de estos conceptos para proveer una descripción precisa y completa al especificar casos de pruebas que ayudan a describir restricciones y observaciones de tiempo, además de cronómetros para cuantificar la ejecución de las pruebas.

2.2 El Framework de Testing JUnit

JUnit es un framework de código abierto creado por Erich Gamma y Kent Beck que se utiliza para escribir y ejecutar los tests unitarios de aplicaciones escritas en Java [9]. En JUnit las pruebas se representan como clases Java y pueden definir cualquier cantidad de métodos públicos de prueba. JUnit provee una librería (*Assert*) para poder comprobar el resultado esperado con el real. La prueba falla si la condición de la aserción no se cumple, generando un informe del fallo.

2.2.1 EasyMock

EasyMock [8] es una librería de código abierto que extiende al framework JUnit permitiendo crear Objetos Fantasmas (en inglés *Mock Objects*) para simular parte del comportamiento del código de dominio. Los objetos fantasmas los crea dinámicamente, permitiendo además que devuelvan un resultado concreto para una entrada concreta. Las principales características de EasyMock es que nos evita escribir manualmente los objetos que simulan determinados comportamientos para cada caso de prueba, soportando el retorno de valores y excepciones además de la ejecución de métodos en un orden dado para uno o más objetos mocks. EasyMock soporta únicamente la creación por defecto de objetos fantasmas a partir de sus interfaces. De todas maneras, EasyMock Class Extension derivado de EasyMock genera objetos mocks para interfaces y clases.

2.3 MOFScript: Transformando Modelos a Texto

MOFScript [10] (www.eclipse.org/gmt/mofscript) permite la transformación de cualquier modelo MOF [11] a texto, generando por ejemplo código Java. MOFScript provee un lenguaje de metamodelo independiente que permite el uso de cualquier clase de metamodelo y sus instancias para la generación de texto.

El lenguaje de transformación MOFScript fue enviado al pedido de propuestas de lenguajes de transformación de modelo a texto lanzado por el OMG, pero no resultó seleccionado. A pesar de eso, lo hemos seleccionado ya que es muy similar al estándar elegido por OMG, además de contar con un plug-in de Eclipse basado en EMF [12] y Ecore. La herramienta soporta el parseo, chequeo y ejecución de scripts escritos en MOFScript y reconoce cualquier archivo con extensión "*.m2t*".

3 Propuesta para Derivar Modelos de Pruebas

En esta sección presentamos la propuesta sobre cómo transformar los modelos de pruebas a código JUnit. Inicialmente introducimos los pasos para especificar nuestro perfil de pruebas basado en U2TP. Luego mostramos el mapeo para convertir los estereotipos U2TP utilizados en la especificación del lenguaje a código JUnit [13]. Explicamos qué tecnologías adicionales podemos utilizar para traducir los componentes no soportados por el framework JUnit y el comportamiento dinámico de los métodos. Finalmente, describimos las reglas para transformar los modelos de prueba estructurales y de comportamiento a código ejecutable JUnit.

3.1 Especificandor un Perfil de Pruebas UML

Para especificar un Perfil de Pruebas UML hay que brindar la terminología para un entendimiento básico de los conceptos de U2TP; definir el metamodelo base UML 2 y una representación MOF para el Perfil de Pruebas UML. Finalmente, debemos establecer un mapeo del Perfil de Pruebas UML al ambiente de ejecución de pruebas

que se quiere implementar, en nuestro caso JUnit. Dicho perfil podemos aplicarlo a los diagramas estructurales y de comportamiento que representan el sistema.

La figura 1 muestra la definición del modelo del lenguaje de pruebas basado en la especificación del Perfil de Prueba de UML a partir del cual se va a generar el código JUnit. Para el desarrollo de nuestro perfil de pruebas basados en U2TP, utilizamos el editor gráfico para metamodelos Ecore contenido dentro de un proyecto Eclipse.

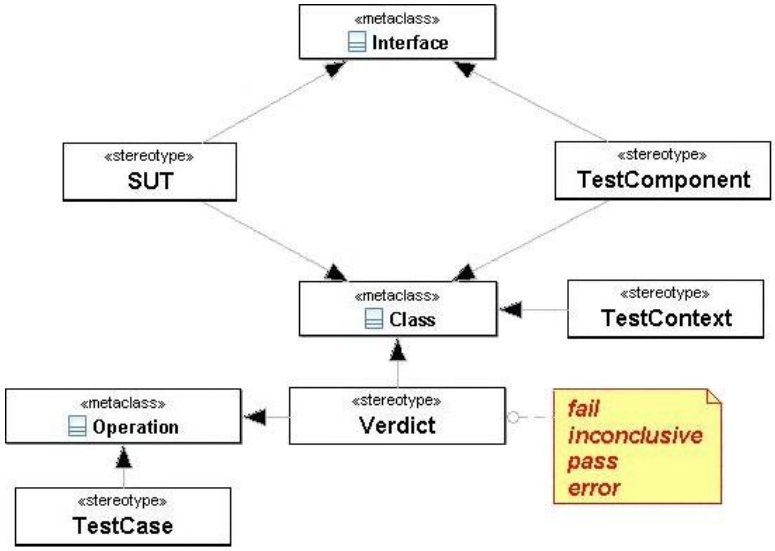

Fig. 1. Definición del Perfil de Pruebas UML

3.2 Mapeo del Perfil de Pruebas UML a JUnit

En esta sección describimos el mapeo de U2TP a JUnit [13]. Este mapeo se muestra en la tabla 1 y considera primariamente al framework JUnit. De todos modos, en los casos donde no exista una asociación hacia JUnit, proponemos otras alternativas para soportar los conceptos incluidos en U2TP. El framework EasyMock fue seleccionado e incluido en la propuesta para extender y traducir el estereotipo de prueba *TestComponent* no soportado por JUnit.

Tabla 1. U2TP vs. JUnit.

U2TP	JUnit
Test Behavior	
Test Control	Sobrecargar el método "*runTest*" de *JUnit TestCase*. Con esto se especifica la secuencia de ejecución de los Test Cases.
Test Case	Se representa con una operación en JUnit. Este método público pertenece a la clase del Contexto de Prueba (*Test Context*), por convención comienza con el prefijo "*test*" y no tiene argumentos para no utilizar el Control de Prueba (*Test*

	Control).
Test Invocation	Es una operación que se puede llamar desde otra operación *Test Case* o desde el Control de Prueba (*Test Control*).
Test Objective	Se puede usar haciendo una llamada al método "*setName*" del framework de pruebas.
Stimulus Observation	No existe un concepto similar aplicable en JUnit, estos conceptos son parte directa de la implementación de los métodos de pruebas (*Test Cases*).
Coordination	Puede implementarse a partir de cualquier mecanismo de sincronización disponible para los Componentes de Pruebas. Ejemplo utilizando semáforos.
Default	No existe un concepto similar aplicable en JUnit. El mecanismo de excepciones en Java se podría utilizar para implementar la jerarquía default del Perfil de Testing.
Verdict	En JUnit, los veredictos puede ser: *pass* (pasa), *fail* (falla), *error*. No existe un veredicto *inconclusive* en JUnit, por lo que se considera como un veredicto *fail*. Este último valor ya no existe en JUnit versión 4 o superior.
Validation Action	Se consideran las llamadas a la librería *Assert* de JUnit.
Log Action Test Log	No existe un mecanismo de log en JUnit.
Test Architecture	
Test Context	El Contexto de Prueba se representa a través de una clase que hereda de la clase *TestCase* de JUnit versión 3.8 (o bien utiliza la anotación *@Test* en versiones superiores).
Test Configuration	No existe un concepto similar aplicable en JUnit.
Test Component	No existe un Componente de Pruebas en JUnit. Para simularlo se utilizan extensiones a JUnit como Mock Objects (objetos fantasmas) como los mencionados anteriormente. En nuestra propuesta, seleccionamos EasyMock framework para complementar esta característica.
System Under Test (SUT)	El Sistema Bajo Prueba no necesita traducirse específicamente a JUnit ya que cualquier clase del sistema puede considerarse como una clase utilitaria o una clase *SUT*.
Arbiter	Se puede considerar como una propiedad del *Test Context* de un tipo de resultado de la prueba (*TestResult*). El algoritmo por defecto genera los valores Pass, Fail y Error similar a un Veredicto, aunque esos valores se pueden redefinir por el usuario.
Scheduler	Se puede representar como una propiedad del *Test Context*.
Utility Part	Cualquier clase disponible en el classpath de Java se puede considerar como una clase utilitaria o parte del *SUT*.
Test Data	
Data pool	Agrupa todas las operaciones de acceso al pool de datos en una clase.
Data partition	Clase que hereda del *Data Pool* con métodos para acceder a la partición de datos.
Data Selector	Es un método del *Data Pool* o *Data Partition*.
Wildcards Coding Rules	No existe un concepto similar aplicable en JUnit.

Time Concepts	
Time zone, Timer	JUnit no soporta conceptos de tiempo, aunque se podrían implementar a través de las APIs estándares disponibles para manipular el tiempo.
Test Deployment	
Test Artifact Test Node	El despliegue está fuera del alcance de JUnit.

3.3 Reglas para Transformar Modelos de Prueba a Código JUnit

Aquí se describen las reglas de transformación para generar el código JUnit a partir del modelo de prueba escrito con el Perfil de Testing UML [14]. Estas reglas son bien conocidas [5] y ya se detallaron anteriormente en la Tabla 1.

3.3.1 Generación del Esqueleto del Código de Pruebas

Traduciendo los conceptos estructurales:

1. Las clases raíces con el estereotipo *TestContext* (Contexto de Prueba) son traducidas como clases que heredan de TestCase en JUnit 3.8, para versiones superiores se utiliza la anotación @Test. Opcionalmente podemos crear un constructor con un parámetro de tipo String e invocar al constructor de la clase padre pasándole dicho String.
2. Las clases no raíces con el estereotipo *TestContext* son traducidas como clases respetando la jerarquía establecida en el modelo de prueba.
3. Las operaciones con el estereotipo *TestCase* son traducidas como operaciones de la clase que representa al contexto de prueba. Por convención, los nombres de estas operaciones deben comenzar con “test” y no deben tener parámetros; los posibles resultados son:
   Pasa (pass): el sistema bajo la prueba cumple las expectativas.
   Falla (Fail): diferencia entre los resultados esperados y reales.
   Error: un error (excepción) en el ambiente de prueba.
   En JUnit no hay un veredicto incierto (inconclusive), por lo general éste es traducido como una falla (fail). En cada clase de prueba se pueden sobrescribir opcionalmente los métodos setUp() y tearDown() para correr código con el objetivo de inicializar o liberar recursos antes o después de cada caso de prueba respectivamente.
4. Las operaciones sin estereotipos son traducidas como operaciones Java de la clase correspondiente.
5. Las clases con el estereotipo *TestComponent* no pueden ser traducidas directamente a JUnit ya que éste no maneja este concepto. Sin embargo, utilizamos el framework EasyMock como extensión a JUnit para crear los componentes de prueba.

6. Las clases con el estereotipo *SUT* (Sistema Bajo Prueba) no necesitan traducirse; cualquier clase en el classpath puede considerarse como una clase de utilidad o una clase de *SUT*. Con la herramienta también vamos a generar el código adicional que acompaña al modelado de pruebas.

3.3.2 Agregado de Comportamiento a los Métodos de Prueba

Los conceptos dinámicos de los modelos de pruebas son diseñados a través de diagramas de secuencia. De todos modos, la generación del código de pruebas completo desde estos modelos dinamicos generalmente no es posible. Esta incompletitud se debe a que los modelos suelen ser demasiado abstractos, careciendo de la información necesaria. Por lo tanto, los desarrolladores están obligados a completar parte del código manualmente. Adicionalmente, los estereotipos *TestComponent* no se pueden traducir a JUnit en forma directa, por esta razón vamos a utilizar EasyMock como una extensión de JUnit para obtener el comportamiento de la prueba.

3.4 Definiendo las Transformaciones de Modelos de Prueba a Código JUnit

En esta sección presentamos parte del código de definición de la transformación para convertir los modelos de pruebas (PIM) basados en UML 2 y U2TP a código JUnit (PSM).

3.4.1 Transformando Modelos Estructurales

La primera regla define la transformación de un modelo UML. Esta regla se complementa con la definición de los estereotipos del Perfil de Pruebas UML, con las reglas que recorren los paquetes para transformar los elementos del modelo en clases JUnit, agregar el comportamiento en los métodos de pruebas definidos en los diagramas de comportamiento, y generar las clases Java de los objetos relacionados entre otras.

```
//Define la transformación de un modelo UML2
texttransformation umlmodelgenerator
  (in uml:"http://www.eclipse.org/uml2/2.1.0/UML") {
    ...
  //Propiedades de estereotipos U2TP.
  property U2TP_VERDICT      : String = "Verdict"
  property U2TP_TEST_CONTEXT : String = "TestContext"
  property U2TP_TEST_CASE    : String = "TestCase"

  uml.Model::main() {
  self.ownedMember->forEach(p:uml.Package) {
    p.mapPackage()
```

```
  }
}

//Mapea una instancia UML a una clase Java o JUnit.
uml.Class::mapClass(packageName : String) {
    ...
  //Si la clase tiene estereotipo <<TestContext>> y no
  //es super clase entonces extiende de TestCase, sino
  //extiende de su super clase.
  if (!self.superClass.isEmpty()) {
    <% extends %>self.superClass.first().name <%%>
  } else if (self.superClass.isEmpty() &&
             self.hasStereotype(U2TP_TEST_CONTEXT)) {
    <% extends TestCase%>
  }

  ...
  if (self.hasStereotype(U2TP_TEST_CONTEXT)) {
    //Agrega el constructor de una clase de Test JUnit
    //y los métodos setUp() y tearDown().
  }
  ...
  //Si la operación esta definida con el estereotipo
  //<<TestCase>> se traduce como método de Prueba JUnit
  //public void test...(), de lo contrario se traduce
  //como una operación de la clase Java.
  if (self.hasStereotype(U2TP_TEST_CASE)) {
    ...
  } else {
    ...
  }

//Verifica si el método tiene comportamiento
//implementado en algún diagrama de secuencia para
//agregarlo en la operación que está procesando.
  if (operationHasBody) {
    ...
  } else {
    ...
  }
 ...
}
```

3.4.2 Transformando Modelos Dinámicos

Para generar el código primero buscamos los métodos con comportamiento dentro de los diagramas de secuencia del modelo de prueba recorriendo todos los paquetes en los que existen clases de tests para luego traducirlos. Los métodos con comportamiento están indicados con el estereotipo *TestComponent* de U2TP. Para detectarlos con MOFScript se utiliza el soporte de UML 2 de la siguiente manera.

```
//Procesa todos los estereotipos <<TestComponent>>
//y retorna las Clases de Tests con sus casos de
//prueba.
uml.Interaction::getTestComponents() : Hashtable {
  ...
  self.ownedMember->forEach(lifeLine:uml.Lifeline) {
    ...
    //Si tiene estereotipo <<TestComponent>>
    if (classReceiver.hasStereotype(
        U2TP_TEST_COMPONENT)) {
      //Procesa los componentes de tests y sus
      //operaciones mocks.
      ...
    }
  }
  ...
}
```

3.5 La Herramienta de Transformación Propuesta

En la figura 2 mostramos la arquitectura de implementación propuesta y describimos la interacción entre sus distintos componentes. Esta herramienta permite generar el código de pruebas JUnit a partir de modelos UML 2 especificados con el Perfil de Pruebas UML.

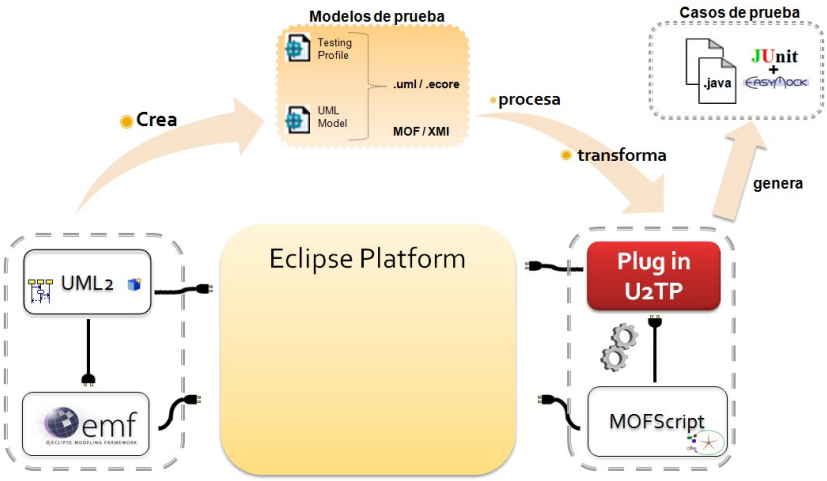

Fig. 2. Arquitectura de implementación.

EMF es el framework de modelado y generación de código base que nos permite construir herramientas y otras aplicaciones basadas en su modelo de estructuras de datos. UML 2 es una implementación basada en EMF del metamodelo OMG del Lenguaje Unificado de Modelado (UML) 2.x para la plataforma Eclipse. A partir de estos dos componentes podemos construir metamodelos UML reusable a través de un esquema XMI [15] común que facilita su intercambio entre distintas herramientas de modelado. UML 2 provee un editor de modelos UML 2 que nos permite diseñar los diagramas. En nuestro trabajo, lo hemos utilizado para diseñar el perfil de pruebas, los diagramas de clases y secuencias y aplicar el perfil sobre el modelo generado. El editor de UML 2 permite generar las instancias de metamodelos en archivos con extensiones .uml y/o .ecore.

MOFScript es la herramienta de transformación de modelos a texto utilizada para generar el código de implementación JUnit. Los templates están creados con el lenguaje independiente del metamodelo que permite la generación de texto desde cualquier clase de metamodelo.

4 U2TP en Acción

En esta sección mostramos cómo utilizar la propuesta a través de un caso práctico sobre la validación y autenticación de un usuario. Este ejemplo demuestra cómo funcionan los estereotipos que provee nuestro perfil en la creación de un diagrama estructural y cómo colaboran en la descripción de los aspectos dinámicos de la prueba definidos en un diagrama de comportamiento. La figura 3 muestra el diagrama de clases con los objetos que participan de la prueba y las relaciones que existen entre ellos. Podemos ver el uso de los estereotipos del perfil de testing sobre operaciones, clases e interfaces UML que facilitan la comunicación y ayudan a comprender el

diagrama de manera más clara. La figura 3 muestra también el diagrama de clases que representa el modelo completo donde aparecen los siguientes nuevos estereotipos aplicados a clases, interfaces, y métodos: *TestContext*, *SUT*, *TestCase*, *Verdict*.

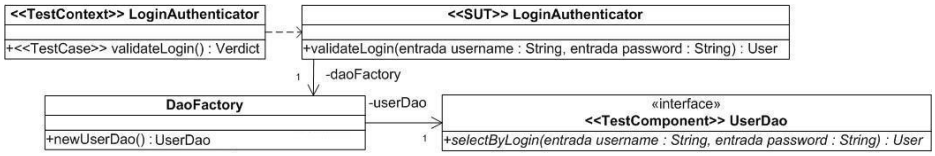

Fig. 3. Modelo de clases del ejemplo especificado con U2TP.

Para traducir y determinar el comportamiento de los conceptos dinámicos para el caso de prueba *testValidateLogin* vamos a escribir el diagrama de secuencia del modelo de prueba que tomaremos como entrada en el proceso de transformación.

La figura 4 muestra la interacción completa de todos los objetos. La instancia UserDao muestra el estereotipo *TestComponent* que JUnit no puede traducir por sí mismo. Por lo tanto, vamos a utilizar EasyMock para completar el comportamiento de la prueba.

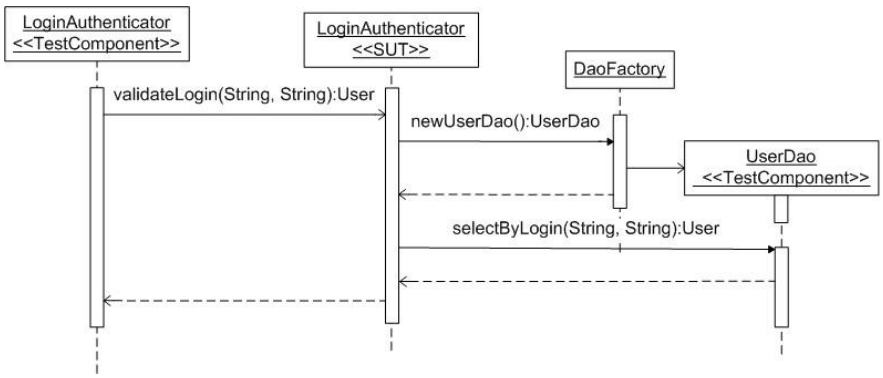

Fig. 4. Diagrama de secuencia del caso de prueba.

Finalmente, mostramos el código JUnit obtenido en forma automática generado por la herramienta. Este incluye además una mayor completitud del caso de prueba agregando el comportamiento modelado en el diagrama de secuencia.

```
public class LoginAuthenticatorTest extends TestCase {
  public LoginAuthenticatorTest(String name) {
    super(name);
  }

  /* (non-Javadoc)
   * @see junit.framework.TestCase#setUp()
```

```
     */
    @Override
    protected void setUp() throws Exception {
      super.setUp();
    }

    /* (non-Javadoc)
     * @see junit.framework.TestCase#tearDown()
     */
    @Override
    protected void tearDown() throws Exception {
      super.tearDown();
    }

    public void testValidateLogin() throws Exception {
      UserDaoImpl mockUserDaoImpl =
          EasyMock.createMock(UserDaoImpl.class);
      EasyMock.expect(mockUserDao.selectByLogin(
          EasyMock.anyObject(), EasyMock.anyObject())).
          andReturn(EasyMock.anyObject());
      String username = null;
      String password = null;
      assertEquals("Verify the expected value", null,
          loginAuthenticator.validateLogin(
              username,password));
      EasyMock.replay(mockUserDaoImpl);
      EasyMock.verify(mockUserDaoImpl);
    }
  }
```

5 Trabajos Relacionados

El trabajo presentado en [16] es una herramienta de generación de código que, en contraste con la propuesta de este artículo, no utiliza los estándares de MDT para diseñar modelos pruebas. [17] es un generador de código de pruebas basado en Java que además de crear el esqueleto estándar en JUnit de la clase a probar, permite especificar objetos no primitivos dentro de un archivo de configuración. [18] es una aplicación que también que documenta el comportamiento del codigo existente creando pruebas en JUnit. El framework [19] crea casos de pruebas en JUnit haciendo reflección y anotaciones dentro de los componentes a probar. Si bien [20] utiliza UML como lenguaje de modelación, la propuesta de reglas gráficas de transformación no tiene una herramienta de soporte que demuestre la factibilidad de su implementación, además de no seguir los estándares especificados por U2TP al igual que [17], [18] y [19]. En [21] encontramos la utilización de las notaciones UML 2 y U2TP estándares para diseñar los modelos de pruebas. Además, proporciona una herramienta de soporte

a través de un plug-in de Eclipse para generar el esqueleto del código de pruebas en forma automática en código TTCN-3 a partir de reglas de mapeo entre los conceptos de los metamodelos de ambas especificaciones. Finalmente, nuestra propuesta sigue la definición del proceso de los trabajos [13] y [22] este último derivado del primero. Aunque [13] introduce una metodología sobre cómo aplicar los conceptos de U2TP a un modelo de diseño UML para obtener un modelo de pruebas, las reglas de transformación no están completas ni se probaron a través de una herramienta que automatice el proceso. La motivación de la investigación de [22] se centra sobre las transformaciones de modelos de pruebas U2TP a código ejecutable por algún lenguaje a definir, para lo cual define una extensión del metamodelo UML en OCL, que luego utiliza para lograr las transformaciones. Este trabajo no logra aún probar dichas definiciones sobre un producto de software que ejecute el proceso automáticamente.

6 Conclusiones y Trabajos Futuros

Nuestra propuesta define la implementación de U2TP para diseñar los casos de pruebas utilizando diagramas de clases y secuencia UML. Además, nuestro trabajo proporciona las definiciones formales completas para transformar dichas anotaciones en código de prueba JUnit basadas en la versión 3.x. La generación de código de prueba brinda la posibilidad no sólo de generar el esqueleto del código, enriquecido con cierto comportamiento especificado en los diagramas de secuencia. La generación del comportamiento se logra a través del uso del framework EasyMock. Por lo tanto, demostramos la factibilidad de la propuesta a través del desarrollo de un plug-in de Eclipse que soporta la transformación de modelos U2TP a código JUnit. La herramienta fue aplicada sobre varios casos de estudio de mediana complejidad obteniendo resultados satisfactorios.

Hasta el momento y de acuerdo a [23], no existe ningún componente de software que colabore en la generación de código JUnit en forma automática basado en modelos de pruebas diseñados con U2TP. Una propuesta en dicho sentido sería considerada de gran utilidad y aporte al estudio de MDT.

En base a los resultados obtenidos nos planteamos los siguientes trabajos futuros:

1. Extender la herramienta para soportar la transformación automática de diagramas estructurales y de comportamiento adicionales a los de clases y secuencia hasta ahora soportados.
2. Permitir la transformación de modelos a otros frameworks de testing unitario.
3. Agregar mayor precisión en la traducción de métodos de comportamiento evaluando la posibilidad de introducir otros modelos más expresivos aparte de los diagramas de secuencia.
4. Extender la herramienta y el lenguaje de modelado de pruebas para permitir la definición (y traducción) de pre y pos condiciones sobre los casos de pruebas. También permitir especificar la secuencia de ejecución de los casos de pruebas.
5. Permitir la re-generación del código a partir de modelos que cambiaron respetando el código escrito manualmente.

Referencias

1. Object Management Group (OMG). http://www.omg.org
2. Utting, M., Legeard, B., Practical model-based testing, a tools approach. Morgan Kaufmann Publishers, 2006
3. Mussa M.,Ouchani S.,Al Sammane W.,Hamou-Lhadj A., A Survey of Model-Driven Testing Techniques. In proceedings of 2009 Ninth International Conference on Quality Software. Korea. 2009
4. Beizer, B., Black-Box Testing: Techniques for functional testing of software and systems. John Wiley & Sons, Ltd., 1995
5. Consortium: UML 2.0 Testing Profile, Final Adopted Specification at OMG (ptc/04-04-02), 2004
6. JUnit testing framework, http://www.junit.org
7. The Eclipse Project. Home Page. Copyright IBM Corp, 2000. http://www.eclipse.org
8. http://www.easymock.org
9. http://java.sun.com
10. Oldevik, Jon. MOFScript User Guide, Version 0.6 (MOFScript v 1.1.11), 2006. Descargado de http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide.pdf en enero 2009
11. Meta Object Facility (MOF) 2.0 Core Specification. OMG, http://www.omg.org/docs
12. Eclipse Modeling Framework EMF. http://www.eclipse.org/modeling/emf/
13. Zhen Ru Dai, Model-Driven Testing with UML 2.0, Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany.

Descargado de http://www.cs.kent.ac.uk/projects/kmf/mdaworkshop/submissions/Dai.pdf en marzo 2009

14. Pons, C., Giandini R., Pérez G., Desarrollo de Software Dirigido por Modelos, Conceptos teóricos y su aplicación práctica, Mac Graw Hill Education and Edulp. (2010).
15. XML Metadata Interchange Specification,

http://www.omg.org/technology/documents/formal/xmi.htm

16. Boyapati, C., Khurshid, S., Marinov, D., Korat: Automated Testing Based on Java Predicates, MIT Laboratory for Computer Science. 200 Technology Square Cambridge, MA 02139 USA, 2002
17. Java Test Code Generator, http://jtcg.sourceforge.net
18. AgitarOne JUnit Generator,

http://www.agitar.com/solutions/products/automated_junit_generation.html

19. ClassMock - Test Tool for Metadata-Based and Reflection-Based Components

http://classmock.sourceforge.net

20. Heckel, R., Lohmann, M., Towards Model-Driven Testing, University of Paderborn, Paderborn, Germany, 2003. Descargado de http://www.cs.uni-paderborn.de/uploads/tx_sibibtex/TACos03-Heckel-Lohmann.pdf en enero 2009
21. Zander J., Dai Z., Schieferdecker I., Din G., From U2TP Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing, 2005
22. Lamancha, B., Mateo, P., Garcia-Rodriguez, I., Usaola, M., Propuesta para pruebas dirigidas por modelos usando el perfil de pruebas de UML 2.0, Universidad de la República de Uruguay, Uruguay / Universidad de Castilla-La Mancha, España, 2008. Descargado de http://www.sistedes.es/TJISBD/Vol-2/No-4/articles/pris-08-perez-perfilUML.pdf en marzo 2009
23. Baker P., Dai Z., Grabowski J., Haugen O., Schieferdecker I., Williams C., Model-Driven Testing. Using the UML Testing Profile. s. Springer-Verlag ISBN 978-3-540-72562-6 (2008)