MoFQA: An Approach for Automatic TDD Test Case Generation from MDD Models

Linda Riquelme
Departamento de Electrónica e Informática
Universidad Católica "Nuestra Señora de la Asunción"
Asunción, Paraguay
linda.riquelme@uc.edu.py

Magalí González
Departamento de Electrónica e Informática
Universidad Católica "Nuestra Señora de la Asunción"
Asunción, Paraguay
mgonzalez@uc.edu.py

Nathalie Aquino
Departamento de Electrónica e Informática
Universidad Católica "Nuestra Señora de la Asunción"
Asunción, Paraguay
nathalie.aquino@uc.edu.py

Luca Cernuzzi
Departamento de Electrónica e Informática
Universidad Católica "Nuestra Señora de la Asunción"
Asunción, Paraguay
lcernuzz@uc.edu.py

***Abstract*—Due to the complexity of software systems and the high probabilities of new errors appearing in any stage of the software life cycle, techniques for quality verification are needed. Software testing is a widely used approach but, due to the costs involved in this process, development teams often debate its applicability in their projects. In the endeavor to reduce the complexity of this process, this document presents an approach for software development based in Test-Driven Development (TDD) supported by Model-Based Testing (MBT) tools to allow automatic test-case generation. Besides, a toolkit for the generation of unit and acceptance tests for Web applications is proposed.**

***Keywords*—*Software Testing*, MBT, TDD, ATDD, BDD, *Acceptance Tests*, *Web Testing*, Selenium, TestNG**

I. Introducción

Una de las técnicas tradicionales para asegurar la calidad del *software* son las técnicas de ***testing***. Las limitaciones y dificultades que se presentan en el *testing* tienen implicancias en puntos como [1]: los costos y la valorización del proceso de *testing*, la búsqueda de métodos de diseño que permitan obtener *tests* más óptimos así como algunas abstracciones que se deberán realizar sobre el producto a verificar. En este trabajo buscamos promover el proceso de *testing*, aplicándolo de forma más natural al proceso de desarrollo de *software* (tratando de reducir los tiempos y esfuerzos necesarios para su implementación). Con este objetivo, nos centramos en dos enfoques principales: *Model-Based Testing* (MBT) [2] y *Test-Driven Development* (TDD) [3].

MBT aparece como una propuesta MDD (*Model-Driven Development*)[1] para facilitar el *testing* automatizado, permitiendo la generación automática de *tests* a partir de modelos que representan una abstracción del sistema a verificar. Esta técnica de *testing* ha sido bastante adoptada y posee soporte de varias herramientas en diversas áreas de aplicación[2]. La utilización de MBT puede reducir el tiempo requerido para la generación, ejecución y validación de *test cases*, lo cual permitiría a la vez, el incremento de tareas de *testing* durante el desarrollo de *software* y la definición de relaciones entre *tests* y requisitos. En [4], por ejemplo, se exponen los resultados de un estudio comparativo sobre la efectividad (cantidad de errores o problemas detectados) y eficiencia (esfuerzo requerido) del proceso de *testing* utilizando *testing* manual y MBT[3]. Los resultados indican que usando MBT fue posible detectar una mayor cantidad de errores críticos con menor esfuerzo de análisis y *testing*, respecto al *testing* completamente manual.

TDD, por su parte, es un proceso de desarrollo de *software* basado en *tests*: para cada funcionalidad a implementar, se define inicialmente un conjunto de *tests* que dirigirá el proceso del desarrollo. Consiste en una práctica definida en algunos métodos de desarrollo ágil y ofrece, entre otras ventajas, la posibilidad de obtener un *software* (generado de forma incremental en cada iteración) con mayor cantidad de *tests* asociados.

Sin embargo, hemos encontrados algunas limitaciones tanto en MBT como en TDD. En lo que respecta a herramientas MBT, hemos verificado principalmente dos limitaciones[4]. Pri-

Este trabajo fue desarrollado en el marco del proyecto "Mejorando el proceso del desarrollo de software: Una propuesta basada en MDD" (14-INV-056), financiado por el Consejo Nacional de Ciencias y Tecnología (CONACYT).

[1]MDD (*Model-Driven Development*) consiste en un paradigma para el desarrollo de *software* en el que se utilizan modelos para la representación abstracta del sistema, a partir de los cuales se generará posteriormente el sistema final.

[2]La encuesta "2014 Model-based Testing User Survey" realizada por Robert V. Binder, Anne Krammer, Bruno Legeard en el año 2014, refleja un importante grado de aceptación de MBT (para la población encuestada) a nivel de la industria. La encuesta se encuentra disponible en: http://model-based-testing.info/wordpress/wp-content/uploads/2014_MBT_User_Survey_Results.pdf

[3]Utilizando notación de máquinas de estado finito (FSM) para describir el comportamiento esperado de un sistema con plataforma *Web*.

[4]Ambas limitaciones pueden verse reflejadas en los resultados obtenidos en la encuesta disponible en: http://model-based-testing.info/wordpress/wp-content/uploads/2014_MBT_User_Survey_Results.pdf

meramente, aún existe una brecha importante en la integración de herramientas que permitan realizar todos los pasos para la automatización de *tests* (definición, generación y ejecución de *tests*). En segundo lugar, si bien los modelos representan abstracciones del sistema a implementar, muchas notaciones utilizadas aún resultan complicadas y tediosas, inclusive para los mismos *testers*.[5] Por otro lado, la definición de tests en TDD es un paso que se realiza, generalmente, de forma manual: se elaboran *scripts* utilizando *frameworks* y herramientas específicas de *testing* para definir los *tests*. Los *tests* definidos resultan tan largos que finalmente es posible tener tantas líneas de código de *testing* como líneas de código de implementación del *software* a verificar[6]. Si la generación de *tests* es apoyada por técnicas y herramientas que acorten (y automaticen) los pasos de definición y ejecución de *tests*, podríamos facilitar la integración de prácticas TDD al proceso de desarrollo de *software*.

En particular, en este trabajo, velando por la participación activa del usuario final en el proceso de definición y verificación del *software* según sus criterios de aceptación, partimos de la siguiente conjetura: "El proceso de desarrollo de *software* basado en TDD, apoyado por MBT para la generación de *test cases*, puede mejorar la productividad del equipo de desarrollo en cuanto al menor tiempo invertido para la elaboración de *tests* y la mayor cantidad de *tests* obtenidos para cada funcionalidad a implementar; con la posibilidad de mejorar, consecuentemente, la calidad del producto final.".

Algunos elementos ampliamente aceptados en la literatura especializada han orientado nuestra propuesta. Si seguimos de forma rigurosa el proceso TDD, cada funcionalidad a implementar estará guiada por el *test* que deberá pasar. de esta forma es posible anticipar la detectección de fallas en el sistema y será menos costoso solucionarlas [5]. Además, los modelos MDD, apoyados por herramientas eficientes de generación de código, prometen reducir la complejidad del proceso del desarrollo de *software*. Otra de sus ventajas es que los modelos independientes de la plataforma (PIM) permiten la generación de *tests* en diferentes plataformas de destino [6]. También, la combinación de TDD con MDD (para definir las unidades de *testing*) puede otorgar al equipo de desarrollo los criterios de aceptación que requiere para el desarrollo de la solución, consecuentemente reduciendo las dudas sobre los resultados esperados [7].

De esta forma, es posible aprovechar el nivel de abstracción que otorgan los modelos (en este caso, para representar comportamientos del sistema y generar *test cases*) reduciendo además la cantidad de detalles específicos de implementación. Asimismo, el enfoque *test-first* de TDD permitiría obtener códigos de implementación con una menor cantidad de errores.

Así, el objetivo general de nuestra propuesta es la definición de un método de desarrollo de *software* basado en la creación de *tests* a partir de modelos (que representen los requisitos de cada funcionalidad a implementar) definidos por el usuario y el desarrollador, siguiendo los pasos y prácticas descritas en TDD. Se presenta la propuesta para metodologías de desarrollo ágiles, guiando el desarrollo incremental de funcionalidades a partir de *tests-cases* que satisfagan las historias de usuario. Se pretende brindar herramientas que den soporte a las etapas iniciales del método propuesto para generar y ejecutar *tests* automáticamente y de forma integrada con el ciclo de desarrollo de *software*.

En cuanto a las herramientas desarrolladas como parte de la propuesta, nos enfocamos en la definición y generación de *tests* unitarios y de aceptación para sistemas de plataformas *Web* 2.0, debido al gran auge de este tipo de aplicaciones [8].

En este documento, presentamos los elementos constituyentes de nuestra propuesta: la problemática y motivación para su definición, el método y las herramientas propuestas. Finalmente presentamos dos experiencias de validación que nos permitieron realizar un análisis preliminar sobre las herramientas presentadas y la necesidad de seguir trabajando en el área.

II. Trabajos Relacionados

Hemos realizado una revisión de la literatura que nos permitió identificar, de acuerdo a algunos criterios definidos, desafíos abiertos en el área. A continuación, destacamos los que resultan de interés para este trabajo:

- La integración de técnicas MBT al proceso de desarrollo de *software* basado en TDD.
- Mayor cercanía de *tests* a los requerimientos del *software*. En particular, para el proceso de *testing*, mantener este vínculo ayuda a encontrar posibles errores de forma anticipada [9].
- Limitación de herramientas MBT integradas para el proceso completo de *testing*: diseño, generación, ejecución y validación de *tests*.
- Necesidad de reducir los conocimientos requeridos para la utilización de herramientas MBT. Destacamos la participación de usuarios finales para la definición de *tests* de aceptación por lo que la complejidad en la definición de sus requerimientos o criterios de aceptación debe ser disminuida.

Durante las búsquedas efectuadas, no se halló un método que proponga la integración de los enfoques TDD y MBT. En el análisis de las herramientas, se encontró un solo caso [10] que utiliza una herramienta MBT siguiendo los pasos definidos por TDD.

En [9] se presenta un mapeo sistemático de trabajos con el objetivo de verificar qué tan vinculados se encuentran el *testing* y los requisitos del *software*. En varios trabajos se lleva a cabo la generación de tests para programas ya implementados. Ante esta situación, [10] menciona que el *testing* de código ya existente (o de modelos derivados de código ya existente) puede crear desconexiones entre los requisitos y los *tests*.

Por otro lado, se deben utilizar varias herramientas para llevar a cabo el proceso de desarrollo de *software* de la mano de MBT. Se necesita: herramienta de modelado para definición

[5] *Tester* es la persona encargada de diseñar y ejecutar los *tests* durante el proceso de desarrollo de *software*.

[6] https://ianhammondcooper.wordpress.com/2007/03/20/ratio-of-test-code-to-production-code/

de *tests*, generador de *tests*, entorno de desarrollo, herramienta para la ejecución y validación de *tests*. En la mayoría de los casos observados, se tienen herramientas diferentes para cada tarea y la integración entre dichas herramientas no es un trabajo sencillo. En [11] se destaca la falta de integración de técnicas MBT al proceso de desarrollo de *software*, mencionando que puede deberse en gran parte a la falta de integración de herramientas que soporten todos los pasos envueltos en el desarrollo de *software* (incluyendo herramientas MBT para definición de *tests*).

En cuanto a la dificultad que conlleva la aplicación de MBT, [11] destaca que existe aún la necesidad de reducir la complejidad: el *tester* hoy en día debe tener conocimientos sobre los lenguajes de modelado, la definición de los criterios de cobertura, formatos de salida generados, entre otros. Además, [12] recalca que el modelado de *tests* sigue siendo un desafío:

- Para sistemas complejos, los modelos necesitan abstraer una gran cantidad de detalles, de otra manera los modelos resultarían inmanejables.
- Las habilidades necesarias para el modelado de *tests* son mucho mayores que las requeridas para la escritura de procedimientos de *tests*.

Finalmente, es necesario independizar los modelos utilizados para la generación de *tests* de los modelos o código de implementación para evitar replicar los errores existentes en la implementación [13]. Por ello se recomienda no reutilizar los modelos de desarrollo para llevar a cabo MBT. Pese a esto, algunos trabajos revisados utilizan el mismo modelo para generación de *tests* y del sistema final con la finalidad de reducir los costos en la definición de *test cases*. Consideramos que los trabajos que realizan ingeniería inversa presentan la misma problemática: si los *tests* se definen a partir del código de implementación, no sería posible detectar ciertos tipos de errores (por ejemplo, falta de adecuación a los requisitos del *software*, incompletitud en la solución, etc.). Otros autores agregan que en algunos casos, sin embargo, la reutilización de modelos puede resultar beneficiosa para verificar la correctitud de la herramienta de generación de código [14].

III. MoFQA: Una Propuesta para el Desarrollo de *Software*

La propuesta de este trabajo, a la cual denominamos **MoFQA** o *Model-First Quality Assurance*, se compone de dos elementos principales:

1. Un método (serie de pasos y prácticas definidas) para el desarrollo de *software* teniendo en cuenta el proceso de *testing*.
2. Un conjunto de herramientas que permite la adopción del método propuesto: (i) para el modelado de requisitos por parte de usuarios finales; (ii) perfiles UML para la definición de *tests* de aceptación y unitarios abstractos; (iii) reglas de transformación para convertir los *tests* abstractos a ejecutables. Las herramientas definidas se limitan al dominio específico de aplicaciones *Web* 2.0 donde la utilización de metodologías ágiles para el desarrollo resulta factible.

Es importante aclarar que ambos componentes de MoFQA son independientes: no se requiere la utilización de las herramientas que presentaremos para seguir el método de desarrollo propuesto, y viceversa. El método propuesto puede ser aplicado a otros dominios (no limitándose solo a aquellos orientados al desarrollo *Web*), en los cuales sea aplicable el desarrollo basado en metodologías ágiles. En las siguientes secciones, presentamos los componentes de la propuesta con mayor detalle.

III-A. Proceso de Desarrollo de Software Propuesto

La Fig. 1 describe el flujo de pasos a seguir, para cada historia de usuario o funcionalidad a desarrollar, de acuerdo al modelo propuesto por MoFQA. Es posible verificar que el método exige seguir el enfoque *test-first* de TDD, a partir de la utilización de modelos definidos tanto por el usuario final como por el desarrollador del *software*.

En conjunto, el usuario y el desarrollador definen los elementos del dominio del *software* a implementar, agregando cada elemento en la medida que sea necesario. Esta definición se realiza utilizando la/s herramienta/s MBT de elección. Llamamos dominio a los elementos principales (entidades y relaciones) para la definición de la lógica de negocios del *software*. Los elementos del dominio definen la estructura de los datos que serán procesados por el *software* en cuestión.

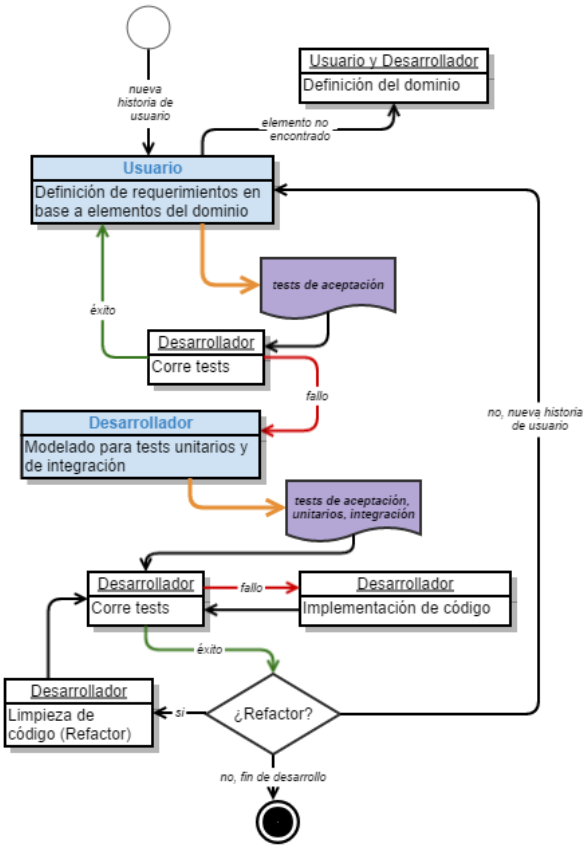

Figura 1. Proceso de desarrollo de software definido en la propuesta.

Ante una nueva funcionalidad, que puede ser descrita a partir de una historia de usuario, el usuario primeramente define los criterios de aceptación que servirían para aprobar o rechazar la funcionalidad, una vez implementada. La definición de dichos criterios se lleva a cabo utilizando elementos del dominio, previamente modelado. El modelado de elementos del dominio y de criterios de aceptación debe ser realizado utilizando herramientas MBT, a elección. A partir de ahí, es posible generar *tests* de aceptación de forma automática, los cuales son ejecutados y validados por el desarrollador.

El éxito en la ejecución de los *tests* generados puede significar que la funcionalidad ya haya sido implementada o que hubo algún error en la definición de los *tests*.

Ante un fallo, el desarrollador pasa a extender el modelo, definiendo *tests* unitarios para la funcionalidad actual, y posiblemente algunos *tests* de integración para verificar la interacción entre los módulos implementados y a implementar. A partir de ahí, se derivan los *tests* ejecutables de forma automática y se sigue el proceso TDD convencional hasta pasar todos los *tests*.

MoFQA propone llevar a cabo el desarrollo de *software* siguiendo puramente el proceso TDD para la generación incremental de sus funcionalidades, cada una de ellas asociadas al menos a un *test* que la verifica. Para facilitar la generación y mantenimiento de *tests*, se propone la utilización de técnicas y herramientas MBT, que permitan la generación automática de *tests* ejecutables a partir de modelos abstractos sobre el comportamiento del *software* a verificar. Los modelos de *test*, además, sirven como documentación sobre el comportamiento de las diversas porciones del *software*. Ante cambios en los requisitos, solo se requeriría modificar dichos modelos. Su actualización permitiría la generación automática de *tests* ejecutables actualizados.

Este proceso incluye la participación del cliente de forma activa y constante la definición del dominio del sistema a implementar, el modelado de criterios de aceptación para las historias de usuario y la validación de las funcionalidades implementadas mediante los *tests* de aceptación generados.

La definición de todos los tipos de *tests* se realiza a partir del mismo conjunto de elementos del dominio. Además, la implementación misma del sistema a verificar debería estar basada en dicho dominio (caso contrario no podrá pasar algunos tipos de *test*). Intentamos así poner en práctica los principios de DDD (Domain Driven Design)[7] buscando mejorar la comunicación entre el usuario y el desarrollador en la transmisión de conocimientos del dominio del *software* a implementar.

III-B. Herramientas Propuestas

Acompañando al método previamente presentado, hemos desarrollado un conjunto de herramientas MBT orientadas a la definición, generación y ejecución de *tests* unitarios y de aceptación para sistemas de plataforma *Web*. Estas herramientas pueden ser utilizadas para: definir el dominio, modelar y generar los *tests* de aceptación, modelar y generar *tests* unitarios, y ejecutar los *tests*.

Con el análisis del estado del arte vimos que uno de los problemas en la aplicación de las prácticas MBT en la industria es la falta de integración entre las herramientas involucradas en el proceso de *testing*. Esta motivación nos llevó a desarrollar una serie de herramientas integradas orientadas al *testing* de sistemas basados en plataformas *Web*. Inicialmente, nos hemos enfocado en la definición y generación de *tests* unitarios y de aceptación.

La definición y generación de *tests* utilizando nuestras herramientas conlleva los pasos ilustrados en la Fig. 2.

Los principales actores involucrados en este proceso son el desarrollador y el usuario final:

- El usuario final dispone de una herramienta a la cual denominamos **MoFQA Modeler** que le permitiría definir los requisitos que deberán ser satisfechos al correr los *tests* de aceptación. Los requisitos se representan a partir de: (i) *mockups* (relacionados a la interfaz de usuario) de las páginas que componen la aplicación *Web* a verificar; (ii) datos de ejemplo a utilizar en las pruebas. A partir de las definiciones del usuario, MoFQA Modeler genera modelos que representan los elementos del dominio y *tests* abstractos de aceptación. Estos modelos son representados utilizando un perfil UML definido en MoFQA y están expresados en formato EMF[8].
- El desarrollador puede enriquecer los modelos definidos por el usuario, para agregar *tests* unitarios especificando pre y post condiciones para la ejecución de cada método presente en el código del sistema a verificar, utilizando el perfil UML definido como parte de esta propuesta.

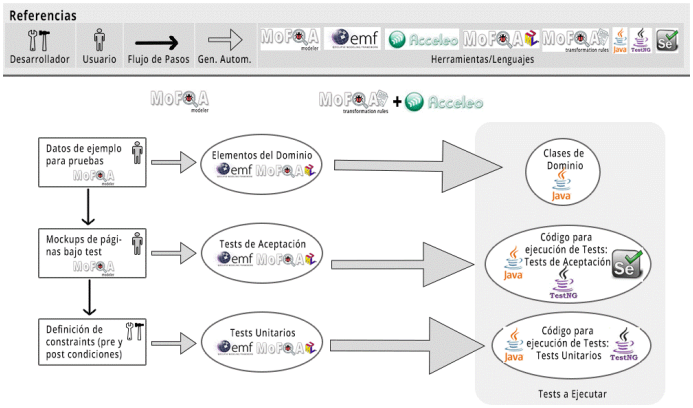

Figura 2. Flujo de pasos para la definición de *tests* con MoFQA.

Los modelos generados pueden ser importados a Eclipse[9] donde, en conjunto con la herramienta Acceleo[10] y las reglas

[7]DDD (*Domain-Driven Design*) [15] consiste en una serie de patrones para la construcción de *software* prestando especial atención a su dominio y a la lógica de negocios, a partir de la definición de un modelado común definido en continua colaboración entre desarrolladores y expertos del dominio.

[8]EMF (*Eclipse Modeling Framework*): http://www.eclipse.org/modeling/emf/

[9]https://www.eclipse.org/

[10]https://www.eclipse.org/acceleo/

de transformación MoFQA (definidas como parte de esta propuesta), se generarán los *tests* ejecutables (unitarios y de aceptación). Los *tests* resultantes están definidos en el lenguaje Java[11], utilizando los *frameworks* de *testing* TestNG[12] y Selenium[13]. Las reglas de transformación MoFQA también definen transformaciones de los elementos del dominio (entidades y relaciones) del sistema a implementar, generando una clase Java para cada entidad modelada.

III-B1. Perfiles UML para la definición de Tests Abstractos: MoFQA propone un perfil UML para el modelado de *tests* abstractos por parte del usuario final y el desarrollador. Este perfil está orientado directamente a la definición de *tests* para aplicaciones *Web*.

El perfil UML *Acceptance Criteria*[14] definido está dividido en cuatro grupos principales:

1. *Data Provider* (Fig. 3): para modelar los datos de entrada para las pruebas.
2. *Domain Specification* (Fig. 4): permite describir elementos del dominio (entidades y relaciones) de la aplicación a probar.
3. *Content Specification* (Fig. 5): para la definición de componentes a ser desplegados en los navegadores *Web*.
4. *Constraint Specification* (Fig. 6): cada unidad funcional a ser codificada por el desarrollador puede estar asociada a una serie de pre y post condiciones para su ejecución.

Ilustraremos la aplicación del perfil *Acceptance Criteria* a partir del siguiente ejemplo[15]: supondremos que deseamos llevar a cabo el desarrollo del portal *Web* existente Amazon.es, para el cual definimos dos requisitos.

El primer requisito consiste en: Un *click* sobre la opción de inicio de sesión en la *Home*, re-direcciona a una página con el formulario que solicita los datos de inicio de sesión. La página debe contar con los elementos: (i) texto "Iniciar sesión"; (ii) formulario con campos e-mail y contraseña; (iii) botón con el texto "Iniciar sesión".

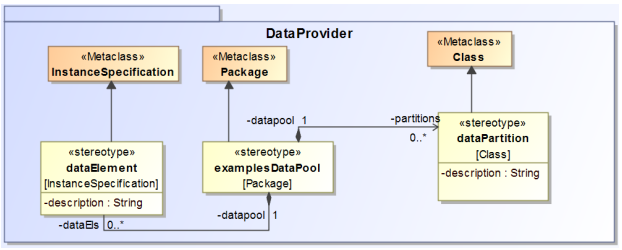

Figura 3. Elementos del grupo *Data Provider*.

[11]https://www.java.com/es/

[12]http://testng.org/doc/

[13]http://www.seleniumhq.org/

[14]Los modelos que pueden ser definidos utilizando este perfil están orientados a la descripción y generación de *tests* de aceptación en su gran mayoría. Se definió un pequeño subconjunto (grupo *Constraint Specification*) que puede ser utilizado para modelar *tests* unitarios abstractos (mediante la definición de pre y post condiciones para la ejecución de métodos).

[15]Descripción detallada de cada elemento del perfil y otros ejemplos de aplicación disponibles en [16]

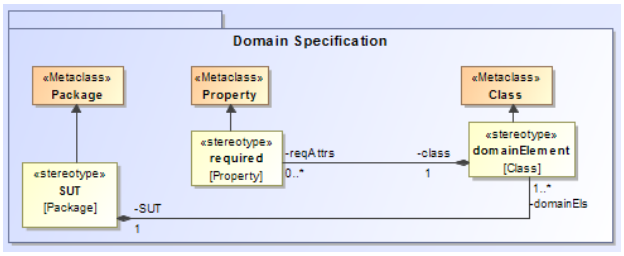

Figura 4. Elementos del grupo *Domain Specification*.

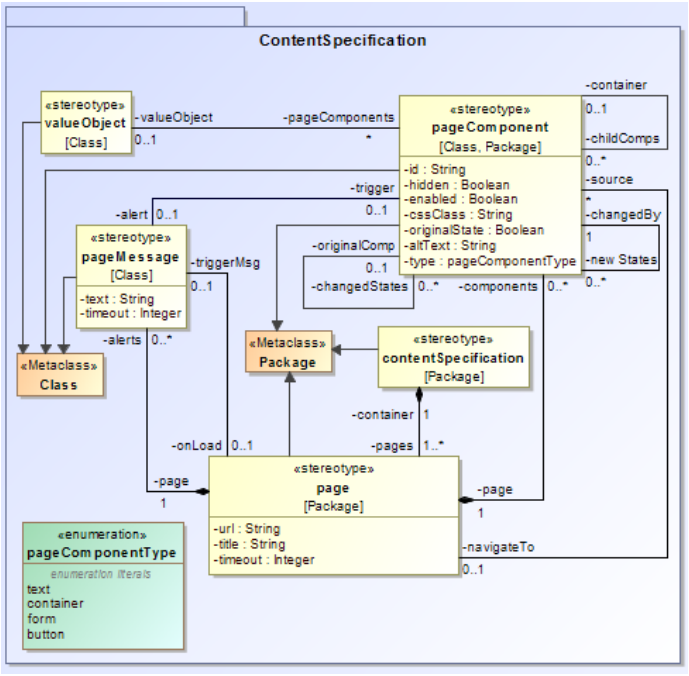

Figura 5. Elementos del grupo *Content Specification*.

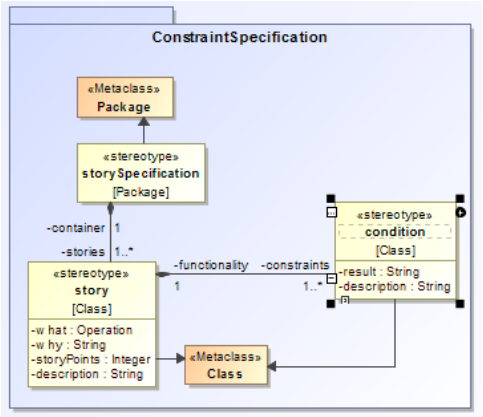

Figura 6. Elementos del grupo *Constraint Specification*.

El segundo requisito consiste en que, al iniciar sesión, un usuario válido debe ser dirigido a la *Home*, en la cual se visualizará el mensaje "Hola" seguido por su nombre.

Procedemos así a modelar los requisitos definidos en el ejemplo usando el perfil *Acceptance Criteria*[16]:

1. Inicio de sesión: necesitamos definir una entidad en el dominio (*«domainElement»* Usuario) a partir de la cual definiremos datos de prueba (un usuario con datos válidos dado por el *«dataElement»* UsVal y otro con datos no válidos dado por el *«dataElement»* UsNoVal). Estos elementos aparecen ilustrados en la Fig. 7.

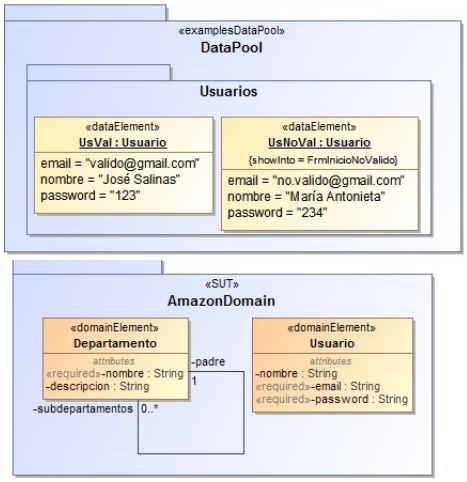

Figura 7. Entidad Usuario agregada al dominio Amazon.es y datos de prueba definidos.

La página de inicio de sesión (*«page»* InicioSesion) es modelada utilizando un elemento formulario (*«pageComponent»* FrmInicioSesion), un elemento de texto (*«pageComponent»* Titulo) y un botón (*«pageComponent»* BtnInicio). Los campos del formulario se definen mediante el *«valueObject»* UsuarioLoginVO. Se puede acceder a esta página haciendo click sobre el elemento *«pageComponent»* IniciarSesion, representado en la *Home*. La Fig. 8 ilustra los elementos mencionados.

2. Actualización de la Home para un usuario autenticado: Volvemos a modelar la página de inicio de sesión, esta vez, indicando que el usuario utilizado para las pruebas será el dado por el *«dataElement»* UsVal (especificado en el formulario *«pageComponent»* FrmInicioValido) pues cuenta con datos válidos para la autenticación. Un click sobre el botón *«pageComponent»* BtnInicioValido deberá re-direccionar a otra página (acción expresada mediante el valor navigateTo del *«pageComponent»* BtnInicioValido). El modelo se presenta en la Fig. 9
La *Home* para un usuario autenticado está representada en la Fig. 10. Se accede a ella luego de una autenticación exitosa y cuenta con un elemento de texto *«pageComponent»* Saludo que despliega el texto "Hola", seguido por el nombre del usuario autenticado (*«dataElement»* UsVal con atributos dados por el *«valueObject»* SaludosUsuarioVO).

[16]En los diagramas, resaltamos en azul los enlaces, en naranja los elementos de tipo texto y en verde los formularios.

III-B2. MoFQA Modeler para la definición de Tests de Aceptación: Para facilitar el modelado de requisitos (que derivarán en los *tests* de aceptación) por parte de los usuarios finales del sistema, hemos desarrollado **MoFQA Modeler**. Consiste en una herramienta de modelado simple que permite definir los elementos:

- Páginas con sus componentes (texto, campos de formularios, botones, mensajes de alerta).

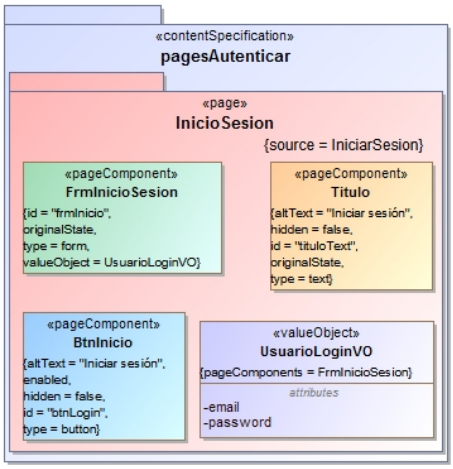

Figura 8. Modelado de página de Login Amazon.es.

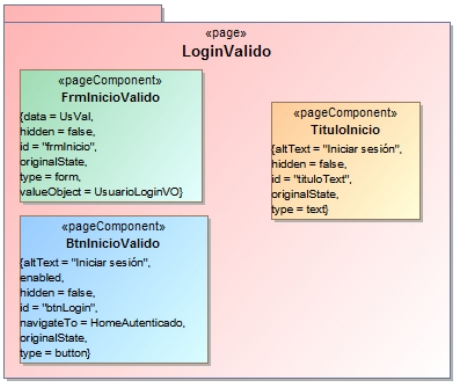

Figura 9. Modelado de "Autenticación Exitosa" en Amazon.es.

- Datos de ejemplo que deberán visualizarse en los componentes de las páginas durante la ejecución de los *tests* de aceptación.
- Resultados de las interacciones con los diferentes componentes de las páginas.

MoFQA Modeler toma los elementos definidos por el usuario y los transforma en modelos (EMF versión 5, enriquecido con el perfil UML definido especialmente para nuestra propuesta). Estos modelos representan los elementos del dominio del *software* bajo verificación y *tests* de aceptación abstractos.

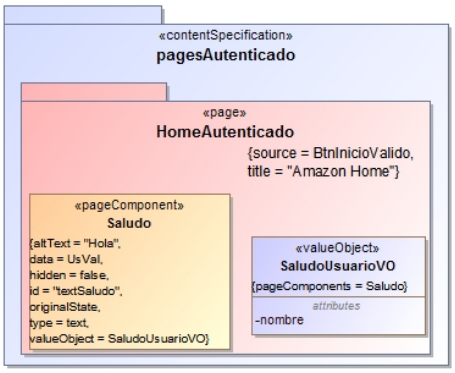

Figura 10. La autenticación exitosa re-direcciona a la Home, donde se visualiza un saludo de bienvenida al usuario.

Con esta herramienta se pretende otorgar un nivel de abstracción más para el usuario final del sistema para el cual generaremos los *tests*. De esta forma, el usuario no necesita tener conocimientos de modelado. Además, como la herramienta genera automáticamente varios elementos del modelo, el usuario debe invertir menos tiempo en el modelado de requisitos.

MoFQA Modeler es una herramienta de plataforma *Web* y fue desarrollada utilizando: lenguajes PHP[17] y Javascript[18] (acompañados de las librerías Bootstrap[19] y JQuery[20], principalmente) para operaciones del lado servidor y cliente, respectivamente. Utiliza una base de datos MySQL[21] para almacenamiento de las especificaciones de cada sitio, definidas por los usuarios. En [16] incluimos un manual de usuario donde se presentan las características de esta herramienta.

III-B3. Framework de Testing y Reglas de Transformación: Los modelos de usuario y de desarrollador representan *tests* de aceptación y unitarios abstractos. La transformación de *tests* abstractos a concretos se realiza utilizando la herramienta Acceleo, en conjunto con las reglas de transformación definidas por MoFQA (*MoFQA transformation rules*). Estas reglas permiten generar código en Java (*frameworks* TestNG y Selenium).

El código generado puede ser utilizado para ejecutar los *tests* de aceptación y *tests* unitarios sobre el *software* bajo verificación, en un entorno de desarrollo integrado como Eclipse. Es importante mencionar que, una vez configurado todo el entorno de desarrollo, será posible centralizar la ejecución de los pasos que el desarrollador debe llevar a cabo para el proceso de *testing*:

- Definición y modelado de *tests*: Importando los perfiles UML, en formato EMF, podrá llevar a cabo el modelado principalmente de *tests* unitarios. Los modelos generados por MoFQA Modeler también se generan en formato EMF por lo que pueden ser importados y adjuntados a los modelos definidos por el desarrollador.
- Generación automática de *tests*: La instalación de la herramienta Acceleo y la importación de las reglas de transformación MoFQA, le permitirá generar los tests ejecutables a partir del modelo que representa los *tests* abstractos del sistema a verificar.
- Ejecución de *tests*: Para ejecutar los *tests* generados y visualizar los resultados, es necesario instalar al proyecto las herramientas TestNG y Selenium. Una vez instalados, la ejecución del archivo XML generado ejecuta de forma automática todos los *tests* definidos por los modelos.
- Integración entre el desarrollo y *testing* del *software*: Además, si el código de implementación del sistema *Web* a verificar está escrito en Java, es posible integrar los ambientes de *testing* y desarrollo en un solo proyecto. Esto permitiría además la definición de *tests* que requieran acceder a métodos de las clases definidas en el código de implementación (este es el caso de los *tests* unitarios).

IV. Experiencias Preliminares de Validación

Para poder realizar un análisis inicial sobre la utilidad de las herramientas presentadas, llevamos a cabo dos experiencias en las que consideramos:

- Ventajas del *testing* utilizando las herramientas MoFQA en relación al *testing* completamente manual. En este sentido, se comparan de ambas técnicas: (i) las cantidades de tiempo necesarias para la definición y ejecución de *tests*; (ii) la cantidad de pasos de *test* llevados a cabo.
- Ventajas del modelado con MoFQA Modeler respecto al modelado utilizando la notación UML enriquecida con el perfil definido en esta propuesta. En este sentido, comparamos específicamente los tiempos necesarios para el modelado.
- Nivel de usabilidad de la herramienta MoFQA Modeler, dirigida a usuarios finales del sistema a verificar.

En las siguientes secciones se describen las dos experiencias realizadas y se presentan sus resultados.

IV-A. Experiencia 1: Taller con Alumnos

Como primera experiencia, hemos llevado a cabo un taller de laboratorio con alumnos de semestres iniciales de Ing. Informática e Ing. Electrónica de nuestra institución, para obtener una idea aproximada sobre las ventajas de utilizar la herramienta MoFQA Modeler para la generación de *tests* de aceptación a partir de las especificaciones de los usuarios. En esta experiencia se realizaron comparaciones entre las técnicas de *testing* completamente manual con respecto al *testing* ayudado por las herramientas ofrecidas por MoFQA.

Se buscó así obtener una medida de: (i) el tiempo que conlleva la definición y ejecución de *tests*; (ii) la cantidad de verificaciones realizadas en cada *test*. Además, se pudo obtener una apreciación sobre la usabilidad de la herramienta.

Como la herramienta está orientada a los usuarios finales del sistema a verificar, nos pareció adecuado el perfil de los

[17] http://php.net/
[18] https://www.javascript.com/
[19] http://getbootstrap.com/
[20] https://jquery.com/
[21] https://www.mysql.com/

alumnos: personas sin conocimiento de modelado y poco conocimiento de programación, pero con afinidad e interés en la definición de las funcionalidades de sistemas informáticos. En total, participaron de la experiencia 24 alumnos, de los cuales fueron considerados 23 resultados, debido a que en uno de los trabajos no se llegó a recopilar toda la información necesaria. En una sesión de 90 minutos, se presentó la herramienta MoFQA Modeler y se solicitó a los alumnos modelar un requisito para una plataforma en línea, utilizada y conocida por ellos: Aula Virtual[22]. Al iniciar la sesión se dio una breve introducción de motivación sobre el concepto de *tests* de aceptación, su importancia y los métodos que se utilizan para su definición (manual y herramientas de automatización para la ejecución de *tests*). Posteriormente, se llevó a cabo un trabajo que consistió en:

1. Seleccionar al menos un requisito, de una lista predefinida, para ser verificado en la plataforma Aula Virtual.
2. Definir *tests* a realizar de forma manual para la verificación de los requisitos. Ejecutar los *tests* manualmente.
3. Modelar los requisitos utilizando MoFQA Modeler y capturar los siguientes datos: tiempo de modelado con la herramienta, dudas/complicaciones con la utilización de la herramienta.
4. Para obtener una apreciación de la usabilidad de MoFQA Modeler para un grupo de usuarios con perfil no técnico, al finalizar la experiencia solicitamos a los alumnos que completen el cuestionario SUS[23] adaptado para los fines de nuestra evaluación.

De los 24 alumnos encuestados, 23 completaron las 10 preguntas del cuestionario (el valor SUS del alumno restante fue descartado por estar incompleto). Obtuvimos un promedio de SUS igual a 55,43 (valor menor a 68, es decir, por debajo del promedio según el análisis presentado en [17]). Clasificamos además los valores SUS individuales según la escala de adjetivos dada por [18]: un 47,83 % de los encuestados asignó un puntaje por debajo a "Aceptable", mientras que un 52,17 % calificó la herramienta como "Aceptable" o mejor.

Los participantes de esta experiencia no llevaron a cabo el proceso completo de *testing* utilizando las herramientas MoFQA debido, principalmente, a la limitación de tiempo disponible para la experiencia. En base a los comentarios realizados por los alumnos vimos, sin embargo, que la percepción de cada uno sobre la utilidad de la herramienta pudo haber mejorado si el alumno era capaz de cerrar el proceso de *testing*, mediante la generación y ejecución de los *tests* resultantes.

Se verificó que muy pocos alumnos lograron definir modelos significativos para los requerimientos que seleccionaron en el trabajo. Durante la experiencia, pudimos verificar que gran parte de los problemas se debieron a cuestiones de infraestructura respecto a fallas en la conexión a Internet. Por esta razón, y por la similitud que presentaban los modelos de los diferentes alumnos, seleccionamos solamente 3 de los más completos para generar los *tests* de aceptación. Con las muestras seleccionadas, realizamos una comparación entre los *tests* manuales[24] definidos por los alumnos y los *tests* generados a partir de los modelos resultantes. Para ello, dividimos los requisitos modelados por los alumnos en subconjuntos de requisitos más específicos y cuyas verificaciones nos interesaron. Cada sub-requisito fue asociado con un paso de *test* realizado tanto por los *tests* manuales como por los *tests* generados. Se deseó realizar así una comparación de la cobertura de requisitos por parte de ambos.

Vimos así que para cada requisito definido, las reglas de transformación permitieron generar varios pasos de *test*, con verificaciones intermedias. De esta forma, cada requisito está acompañado por al menos un *test* con varias comprobaciones. A fin de verificar el porcentaje de cobertura de requisitos por parte de los *tests* (manuales y generados) elaboramos una tabla por cada trabajo seleccionado, registrando los datos: sub-requisito, pasos generados por MoFQA para su comprobación, verificación de cobertura del sub-requisito por parte de los *tests* manuales definidos, pasos intermedios de *tests* manuales.

También interesó la cantidad de tiempo empleada para la definición y ejecución de *tests* manuales con respecto a la cantidad de tiempo empleada para el modelado, generación y ejecución de *tests* generados.

Así, en las tablas I y II incluimos los criterios de comparación para los *tests* manuales y *tests* generados, para los trabajos seleccionados. En la Tabla I se muestra: el requisito modelado en el trabajo, la cantidad de sub-requisitos definidos para el requisito seleccionado, la cantidad de tiempo que llevó la definición y ejecución de sus tests de forma manual, la cantidad de tiempo que llevó el modelado de requisitos con MoFQA Modeler. En la Tabla II, por su parte, se incluyen la cantidad de *tests* manuales realizados (TM), la cantidad de *tests* generados automáticamente (TG), la cantidad de líneas de código generadas para los *tests* (LOC) y el valor SUS asignado por el alumno a la herramienta.

Observamos que el código generado por MoFQA realiza una mayor cantidad de pasos con verificaciones intermedias que, además, quedan automatizados para posteriores ejecuciones. En promedio tenemos 8 pasos adicionales generados por las reglas de transformación, en comparación a los pasos definidos por los *tests* manuales. Tampoco existe mucha diferencia entre la cantidad de tiempo destinada al modelado de requisitos con MoFQA Modeler, en comparación a la definición y ejecución de tests de forma manual. Si bien la diferencia no es significativa, el modelado con MoFQA genera código de *test* de forma automática. Éstos pueden ser ejecutados sin costo adicional las veces que sea necesario, permitiendo *tests* de regresión automáticos. Los tres requisitos pudieron modelarse en un promedio de 4,3 minutos.

Finalmente, otro dato interesante en esta experiencia es el valor SUS asignado por los alumnos que realizaron los trabajos seleccionados. En promedio, obtenemos un valor SUS de 65,8.

[22]https://www.claroline.net/

[23]https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html

[24]Antes de utilizar la herramienta, los alumnos ejecutaron el *testing* del requisito seleccionado, de forma completamente manual: la definición de pasos de *test* así como la ejecución de dichos pasos se realizó manualmente.

Cuadro I
COMPARACIÓN 1: TIEMPO DE DEFINICIÓN MANUAL DE *tests* VS. TIEMPO DE MODELADO DE *tests*

Requerimiento	Cant. Sub-Req.	T. Manual	T. Modelado
1	7	4 min.	5 min.
2	2	5 seg.	3 min.
3	3	6 min.	5 min.

Cuadro II
COMPARACIÓN 2: CANTIDAD DE PASOS DE *test* OBTENIDOS Y VALOR SUS ASIGNADO

Requerimiento	TM	TG	LOC	SUS
1	9	22	109	67,5
2	2	6	73	82,5
3	9	15	341	47,5

Podemos ver un crecimiento importante en el valor obtenido anteriormente. Creemos que esto puede deberse a que estos alumnos no tuvieron problemas de infraestructura durante la experiencia. Pudieron así seguir los ejemplos presentados en clase y grabar con éxito sus modelos en la herramienta.

IV-B. Experiencia 2: Ilustración para un caso de ejemplo

Como segunda experiencia, definimos 5 requisitos para el caso de ejemplo presentado (Amazon.es) y generamos los *tests* correspondientes. En esta experiencia se comparó el tiempo de modelado de requisitos usando MoFQA Modeler respecto al tiempo de modelado mediante la utilización de un editor UML (MagicDraw[25]) y el perfil UML definido. Además, se verifica que es posible llevar a cabo el proceso completo de *testing* utilizando las herramientas MoFQA de forma integrada.

La ilustración permitió verificar el ahorro de tiempo que supone el modelado de *tests* de aceptación (según las definiciones de los perfiles MoFQA) utilizando MoFQA Modeler, en comparación a hacerlo utilizando directamente los perfiles UML y una herramienta de edición de UML. Creemos que MoFQA Modeler lleva ventaja porque abstrae varios detalles del modelo, que se presentan transparentes para el usuario. Su uso no solo permite que usuarios sin experiencia en modelado definan *tests* de aceptación sino que, además, permite que usuarios más técnicos puedan modelar dichos *tests* en menor cantidad de tiempo. En la Tabla III se comparan los tiempos transcurridos para el modelado de *tests* usando MagicDraw y los perfiles (caso 1) y MoFQA Modeler (caso 2). Por su parte la Tabla IV compara la cantidad de líneas de código generado[26] en total (para todos los requeisitos) a partir de cada modelo.

Si bien es evidente que el tiempo de modelado es mucho menor en todos los casos (un promedio de 12 minutos de ventaja para MoFQA Modeler en todos los requisitos), es importante mencionar que la herramienta tiene limitaciones y no es posible generar todos los elementos disponibles en el perfil UML de MoFQA. Sin embargo, para los casos presentados en esta experiencia, las funcionalidades ofrecidas fueron suficientes para generar los *tests* requeridos.

[25]https://www.nomagic.com/products/magicdraw

[26]Solo se consideran las líneas de código directamente vinculadas a los tests. El *framework* de *testing* de MoFQA no está incluido en el conteo.

Cuadro III
TIEMPO (EN MINUTOS) PARA EL MODELADO DE REQUERIMIENTOS USANDO: (I) PERFILES UML Y MAGICDRAW; (II) MOFQA MODELER

Req.	Perfiles UML + MagicDraw	MoFQA Modeler
1	18 min.	5 min.
2	19 min.	6 min.
3	20 min.	8 min.
4	14 min.	3 min.
5	14 min.	3 min.

Cuadro IV
CANTIDAD DE LÍNEAS DE CÓDIGO GENERADA PARA CADA REQUERIMIENTO USANDO LOS MODELOS GENERADOS POR: (I) PERFILES UML Y MAGICDRAW; (II) MOFQA MODELER

Código generado	Perfiles UML + MagicDraw	MoFQA Modeler
Configuración XML	29 líneas	27 líneas
Clases Dominio	2 (180 líneas)	2 (166 líneas)
Value Objects	3 (196 líneas)	7 (470 líneas)
Métodos de *Test*	3 (349 líneas)	5 (430 líneas)
Total	754 líneas	1.093 líneas

La diferencia en la cantidad de líneas de código generadas en cada caso se debe a que, mediante el modelado realizado directamente con los perfiles UML se logró una mayor optimización, reutilizando elementos del modelo ya existentes. MoFQA Modeler, sin embargo, crea una mayor cantidad de elementos de modelo ya que las relaciones que se pueden definir entre ellos son solo de tipo 1 a 1 (limitaciones de la herramienta en [16]).

IV-C. Discusión Final

Los resultados arrojan una ventaja hacia la generación de código de *tests*, respecto a la definición y ejecución de *tests* manuales. Los tiempos requeridos para la definición de *tests* no representan una diferencia significativa y, además, los *tests* generados tienen la ventaja de que pueden ser guardados y reutilizados, siempre que sea necesario. Asimismo, la cantidad de pasos de *test* generados (y por ende, la cantidad de verificaciones realizadas) es mayor que la cantidad de pasos llevados a cabo de forma manual. La diferencia es significativa inclusive para *tests* pequeños. El modelado con MoFQA Modeler también mostró ser ventajoso respecto a la cantidad de tiempo que permite ahorrar en comparación al tiempo que llevaría modelar utilizando los perfiles y algún editor UML. En cuanto a la usabilidad de MoFQA Modeler, los alumnos participantes de una de las experiencias contestaron el cuestionario SUS adaptado. El promedio obtenido dio un valor por debajo del promedio proablemente debido a una serie de problemas que se presentaron en esa experiencia. Para un subconjunto de alumnos que pudo finalizar la experiencia, se observa un incremento importante en el valor SUS obtenido, llegando casi al valor promedio de la escala. La inclusión del usuario final en el proceso de definición y verificación

de requisitos de aplicaciones *Web* puede obtenerse gracias a las ventajas de la herramienta MoFQA Modeler: ahorro en el tiempo de modelado, abstracción (además de la disminución en la cantidad) de elementos de modelado y el nivel de usabilidad de la herramienta. El usuario así, podrá definir los criterios de aceptación para cada funcionalidad a verificar y generar *tests* de aceptación que los verifiquen. Como los *tests* se generan de forma automática, éstos serán una copia fiel de los requisitos del usuario, sin una intervención de los desarrolladores o testers en la definición de dichos *tests*. Otro punto interesante en este sentido es la aplicación de principios DDD en la definición de requisitos mediante la utilización de herramientas MoFQA: el usuario define los elementos del dominio (lógica de negocios) así como los nombres de los componentes de las diversas páginas, obligando así al desarrollador a utilizar los mismos conceptos durante su programación (caso contrario, no pasarían los *tests* generados). Finalmente, pudo verificarse la cobertura de requisitos por parte de los tests generados. Se verificó que para los requisitos modelables por las herramientas MoFQA, es posible obtener una cobertura del 100 % a través de las reglas de transformación generándose además, para cada *test*, varios pasos intermedios que aumentan la cantidad de verificaciones efectuadas.

V. Conclusiones y Trabajos Futuros

Este trabajo ha propuesto, por un lado, un modelo, denominado MoFQA, para el desarrollo de *software* basado en el proceso TDD integrado con técnicas MBT para reforzar (y automatizar) la generación de *tests*. En forma complementaria, hemos desarrollado herramientas MBT como propuestas para la definición y generación de *tests* unitarios y de aceptación para sistemas de plataforma *Web*. Dichas herramientas son generales y pueden o no ser utilizadas siguiendo el modelo de desarrollo propuesto por MoFQA.

Para poder llevar a cabo un análisis inicial sobre la utilidad de las herramientas desarrolladas, efectuamos dos experiencias: (i) un taller con alumnos, para obtener una percepción inicial sobre la usabilidad del sistema; (ii) una ilustración comparando los tiempos de definición de *tests* usando los perfiles UML y la herramienta de modelado propuesta. La validación que realizamos de la propuesta se ha enfocado en dos aspectos específicos: (1) cobertura de los *tests* generados, respecto a los requerimientos definidos para el sistema; (2) simplicidad en el uso de la herramienta dirigida a los usuarios finales para la generación de *tests* de aceptación. Ambas experiencias se enfocaron en las herramientas generadas como parte de esta propuesta, como trabajo futuro se propone ampliar las experiencias para validar el proceso TDD propuesto.

Con experiencias preliminares vimos que, con poca preparación, los usuarios finales (no técnicos) pueden automatizar la ejecución de *tests* de aceptación. A partir de modelos simples definidos por los usuarios, gracias a las reglas de transformación se generaron varios pasos intermedios de *test* de forma automática para cada requisito, facilitando además la integración del proceso de modelado y definición de *tests* con la generación de código de *tests* ejecutables. Esto ha permitido verificar no solo una disminución en el tiempo de modelado de requisitos sino además, la participación activa y de manera fluida del usuario final en la definición y verificación de requisitos basados en sus criterios de aceptación.

Como trabajo futuro, se considera la definición de nuevas experiencias, más formales, para obtener mayores evidencias tanto acerca de la utilidad del modelo de desarrollo de *software*, basado en TDD y prácticas de MBT, como sobre la usabilidad de la herramienta. También, se está planteando la generalización/extensión de la herramienta de modelado y de las reglas de transformación de forma a que los *tests* no estén únicamente orientados a la plataforma *Web*, sino puedan cubrir otras plataformas de despliegue.

Referencias

[1] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[2] M. Utting, "Position paper: Model-based testing," *Verified Software: Theories, Tools, Experiments. ETH Zürich, IFIP WG*, vol. 2, 2005.

[3] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[4] C. Schulze, D. Ganesan, M. Lindvall, R. Cleaveland, and D. Goldman, "Assessing model-based testing: an empirical study conducted in industry," in *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 135–144.

[5] C. Jones, P. O'Hearn, and J. Woodcock, "Verified software: A grand challenge," *Computer*, vol. 39, no. 4, pp. 93–95, 2006.

[6] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synthesis Lectures on Software Engineering*, vol. 1, no. 1, pp. 1–182, 2012.

[7] J. Hofstader, "Model-driven development applied to microsoft visual studio, 2006.[cited october 2009]," *Available by Internet:<http://msdn. microsoft. com/en-us/library/aa964145. aspx*.

[8] J. G. Guerra, "Web 2.0 presente y futuro de las aplicaciones," *Perspectiv@ s*, vol. 4, no. 4, pp. 60–64, 2017.

[9] Z. A. Barmi, A. H. Ebrahimi, and R. Feldt, "Alignment of requirements specification and testing: A systematic mapping study," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. IEEE, 2011, pp. 476–485.

[10] B. A. Shappee, "Test first model-driven development," Ph.D. dissertation, Worcester Polytechnic Institute, 2012.

[11] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: a systematic review," in *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*. ACM, 2007, pp. 31–36.

[12] J. Peleska, "Industrial-strength model-based testing-state of the art and current challenges," *arXiv preprint arXiv:1303.1006*, 2013.

[13] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.

[14] E. Escott, P. Strooper, J. Steel, and P. King, "Integrating model-based testing in model-driven web engineering," in *Software Engineering Conference (APSEC), 2011 18th Asia Pacific*. IEEE, 2011, pp. 187–194.

[15] F. Marinescu and A. Avram, *Domain-driven design Quickly*. Lulu. com, 2007.

[16] L. Riquelme, "Mofqa: Una propuesta para la generación automática de tests a partir de modelos siguiendo el proceso tdd," Universidad Católica "Nuestra Señora de la Asunción", Tech. Rep., 2017. [Online]. Available: http://www.dei.uc.edu.py/proyectos/mddplus/wp-content/uploads/2018/01/mofqa_libro.pdf

[17] J. Sauro, "Measuring usability with the system usability scale (sus)," 2011.

[18] A. Bangor, P. Kortum, and J. Miller, "Determining what individual sus scores mean: Adding an adjective rating scale," *Journal of usability studies*, vol. 4, no. 3, pp. 114–123, 2009.