UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Titulo: Proceso de Generación de Casos de Prueba en el Contexto MDD/MBT
Autores: Natalia Correa
Director: Dr. Roxana Giandini
Codirector: -
Asesor profesional: -
Carrera: Licenciatura en Sistemas

Resumen

Desde la aparición de la metodología MDD, mucho se ha propuesto y definido en cuanto a lenguajes y herramientas que sirven de soporte y automatizan sus diferentes aspectos. Uno de estos aspectos donde se ha puesto más énfasis es en la definición de lenguajes que permiten traducir un modelo en otro. Luego, con la aparición de MBT, se han propuesto nuevos aportes e investigaciones que sirven de soporte para la generación de casos de prueba y para el *testing* en general.
En este sentido, se ha analizado que las actividades de testeo a nivel modelado, muchas veces se realizan en la etapa de diseño del sistema, con los detalles de implementación ya definidos. Sin embargo, es en etapas iniciales del desarrollo de software, cuando se define claramente la funcionalidad del sistema, indicando el "qué" sin mencionar el "cómo".
Esta tesina presenta un proceso, dentro de los contextos mencionados, que permite generar casos de prueba tempranamente en el ciclo de vida del desarrollo, de manera automática por medio de transformaciones y con generación de trazabilidad. Se brinda, de esta manera, soporte al *testing* de sistemas y a las técnicas de trazabilidad dentro de los contextos MDD y MBT

Palabras Claves

Desarrollo de Software Dirigido por Modelos (MDD),
Testing Basado en Modelos (MBT),
Trazabilidad,
Lenguajes de Transformaciones de Modelo
UML –Lenguaje Unificado de Modelado-
Casos de Uso

Conclusiones

Se considera que los objetivos planteados en esta tesina han sido cumplidos y que el aporte que realiza es valioso. A partir del estudio y análisis de necesidades, paradigmas y tecnologías, se ha definido un proceso que permite generar casos de prueba de manera automática, por medio de transformaciones de modelo, a partir de Casos de Uso.
Se generan, además, modelos de trazas que vinculan a los elementos del proceso definido.

Trabajos Realizados

Un estudio sobre los paradigmas MDD y MBT, Lenguajes de Transformación en MDD, el estado del arte en *Traceability* (en MDD y en MBT). La definición de perfiles UML para testing, con su metamodelo y reglas de buena formación en OCL. La definición e implementación de las transformaciones (M2M YM2T) que permiten automatizar la generación de los modelos de *testing*. La definición de un metamodelo para trazabilidad. La definición e implementación de transformaciones (M2M YM2T) que permiten automatizar la generación de trazas.-

Trabajos Futuros

Definir una herramienta de soporte que permita automatizar completamente el proceso permitiendo aplicar al Diagrama de Casos de Uso (MCU) las transformaciones que generen los diagramas de actividades intermedios con conceptos de *testing* y sus consecuentes transformaciones a casos de pruebas.Incorporar opciones de métricas al proceso definido. Entre ellas: de estimación por medio de UCP (Use Case Points), de cobertura de casos de prueba, de análisis de impacto por cambios introducidos.

Fecha de la presentación: NOVIEMBRE 2015

Proceso de Generación de Casos de Prueba en el Contexto MDD/MBT

Facultad de Informática- UNLP

Natalia Correa. Noviembre 2015

Agradecimientos

A Dios, en primer lugar y ante todo.

A mi familia, a mi hermana Eugenia, a mis amigos.

A 'La Casita', los chicos y mis hermanos del alma. Mi vida es muy diferente gracias a todos ellos.

A mi directora y compañera Roxana, por su empuje y su empeño en que me reciba. De corazón, gracias!

A Raúl y Adriana, porque mi vida cambió a partir de conocerlos. Mi felicidad tiene mucho que ver con ustedes. Gracias!

A Claudio, mi compañero de vida, por su amor profundo.

A Paz y Esperanza, por elegirme y hacerme una feliz mamá.

Proceso de Generación de Casos de Prueba en el Contexto MDD/MBT

Índice

Índice de Figuras

Índice de Tablas

Capítulo 1- Introducción

En este capítulo presentamos la motivación y el contexto en el que se encuadra esta tesina.
A continuación, los objetivos de la misma, adicionando luego más información de interés como las publicaciones relacionadas al presente trabajo de investigación y finalmente, la organización general de la tesina.

1.1 Motivación

La Ingeniería de Software ha sido definida por la IEEE [1] como "la aplicación de un enfoque sistemático, disciplinado y cuantificable para el desarrollo, operación y mantenimiento de software". En otras palabras, podríamos decir que trata de la aplicación de la disciplina ingenieril a la construcción de un producto software.
Para el desarrollo de software se han definido -y se utilizan- distintos paradigmas o metodologías que precisan fases, etapas, roles y responsabilidades, métricas y circuitos bien definidos que permiten ordenar el proceso de generación del software. Estas propuestas intentan mejorar la producción del software atacando los problemas que el desarrollo tradicional del mismo ha ocasionado. Estos problemas, conocidos y estudiados por la Ingeniería de Software, son los de productividad, de portabilidad, de interoperabilidad y del mantenimiento y de la documentación.

En los últimos años, una iniciativa denominada "Desarrollo de Software Dirigido por Modelos" (MDD- *Model Driven software Development*) [2], [3], [4] ha trascendido dentro de la Ingeniería de Software y se ha convertido en un nuevo paradigma de desarrollo software.
El enfoque MDD presenta una nueva forma de enfrentar el desarrollo de los sistemas de software. En primera instancia, otorga al uso de los modelos una real significación. Como mencionan las autoras de [3] "... *Los modelos 'dirigen' el proceso de desarrollo - diseño, construcción, pruebas, despliegue, operación, administración, mantenimiento y modificación de los sistemas - convirtiéndose en entidades activas y centrales del paradigma*".
Esta es una verdadera ventaja de MDD ya que en la actualidad, la mayoría de las metodologías de desarrollo de software utilizan modelos. La diferencia entre los variados métodos reside básicamente en la clase de modelos que deben construirse, en qué etapas y cómo estos van a representarse y manipularse.
Por otra parte, los beneficios más destacados del Desarrollo Dirigido por Modelos se relacionan directamente con los problemas de la construcción de software mencionados anteriormente y tienen que ver con mejorar la productividad, permitir portabilidad, soportar interoperabilidad y facilitar el mantenimiento de la aplicación.

Dentro del contexto mencionado (MDD), el "Testing Basado en Modelos" (MBT - *Model Based Testing*) también conocido como "Testing dirigido por modelos" (MDT - *Model Driven Testing*) [5], [6] es una propuesta reciente que desafía la problemática de generar los modelos y artefactos necesarios para el testeo de software. Su relación con MDD reside, por ejemplo, en que los modelos de *testing* como son los casos de prueba, son derivados total o parcialmente desde modelos origen que describen la funcionalidad del sistema en desarrollo.

Los *tests* producidos desde el modelo serán abstractos, debiendo transformarse más luego en ejecutables. Por cierto que este paso requiere intervención de *testers* o ingenieros de *testing* pero la mayoría de las herramientas de testeo (también las basadas en modelos) proveen de cierta asistencia durante el proceso.

Las actividades de testeo a nivel modelado, generalmente se realizan en la etapa de diseño del sistema, con los detalles de implementación ya definidos[1]. Sin embargo, es en etapas iniciales del desarrollo de software, cuando se define claramente la funcionalidad del sistema, indicando el "qué" sin mencionar el "cómo".
En este sentido, cobran vital importancia las prometedoras visiones de MDD y MBT para la automatización del *testing* de sistemas, reduciendo esfuerzos y asistiendo al equipo de *testing* con la generación de modelos -por medio de transformaciones de modelos- por ejemplo de casos de prueba.

Entre los lenguajes de modelado que pueden ser utilizados para modelar modelos de requerimientos y automatizar la generación de modelos de *testing* se encuentra UML [7], estándar oficial de la OMG (*Object Management Group*), maduro, adoptado y utilizado por la comunidad de las ciencias de la computación, además de contar con numerosas herramientas CASE que son utilizadas por los equipos de desarrollo.
Un modelo UML de Casos de uso representa la funcionalidad global de un sistema dado. Esta funcionalidad puede representarse además con sus pre y post condiciones, indicando el resultado esperado, aportando información sobre el estado del sistema antes y después de su ejecución, por lo que resulta un artefacto muy útil para el *testing* de los sistemas.

Dada la complejidad per se del proceso de *testing* dentro del proceso de desarrollo, es deseable y necesario que cuente con un proceso definido, en una etapa temprana dentro del proceso de desarrollo del software y que pueda ser automatizado, parcial o totalmente.
En particular y para este trabajo, es importante la relación entre los modelos de requerimientos de software y su relación con la etapa de *testing*.

Nos interesa además adicionar trazabilidad al proceso de generación de casos de prueba ya que reporta varios beneficios al proceso de desarrollo de sistemas.

En la Ingeniería de Software, se ha definido trazabilidad como la capacidad de establecer relaciones entre dos o más artefactos de un sistema dentro de un proceso de desarrollo, especialmente aquellos artefactos con vínculos 'predecesor-sucesor' o 'maestro-subordinado' entre sí, por ejemplo, entre los requerimientos y los elementos de diseño que los modelen [1].
Además, contar con técnicas que soporten trazabilidad dentro del ciclo de desarrollo del software es considerado como una medida de madurez del proceso de desarrollo por varios estándares.
En la fase de *testing* en particular brinda la posibilidad de identificar los casos de prueba que son necesarios ejecutar para testear una determinada funcionalidad y permite contar con una medida de la cobertura de los casos de prueba definidos.

Para finalizar, podemos mencionar además que es deseable que puedan utilizarse los beneficios aportados por MDD en cuanto a transformaciones de modelo y las facilidades de los lenguajes –y herramientas- de transformación de modelos en relación a trazabilidad. Observar además qué características tienen y qué facilidades brindan en tal sentido. Con el agregado de las trazas entre elementos de los diferentes modelos, se completa el proceso generador de casos de prueba.

[1] Trabajos que abordan esta temática son descriptos en la sección de "Trabajos Relacionados" del capítulo "Conclusiones finales"

1.2 Objetivos

El objetivo general de esta tesina es brindar soporte al *testing* de sistemas en particular (y a la ingeniería de software en general) con un proceso definido para tal fin y realizar un aporte a las técnicas de trazabilidad dentro de los contextos MDD y MBT.

Para ello, se ha desarrollado un proceso que permita generar de casos de prueba del sistema -a los que llamaremos *tests* de integración- que testeen conjuntos de posibles ejecuciones de los usuarios-, automatizando la derivación de los casos de prueba mencionados a partir del modelo de casos de uso, adicionando trazabilidad entre los elementos.
Este proceso, más general y abarcativo, se completa con otro (sub)proceso para especificar casos de prueba a partir de la documentación de cada caso de uso. A esos casos de prueba los llamamos *tests* de unidad y son estos los que primero necesitan ser ejecutados y probados para luego poder probar los *tests* de integración. Esta derivación también pretende ser automatizada y contar con la definición -también automática- de trazas entre los elementos.

Los aportes que brinda esta tesina son:

- En general, la definición de un proceso de generación de modelos de *testing* a partir de modelos de Casos de Uso.

- En particular, la definición de este proceso incluye:

  - La determinación de elementos de modelado específicos para diagramas de actividades relacionados con Casos de uso
  - La definición de perfiles UML para *testing*, con su metamodelo y reglas de buena formación en OCL
  - La definición de las transformaciones, modelo a modelo y modelo a texto que permiten automatizar la generación de los modelos de *testing*
  - La definición de un metamodelo para trazabilidad
  - La definición de transformaciones modelo a modelo y modelo a texto que permiten automatizar la generación de trazas

Además, se presentan:

- Una recopilación y resumen sobre Lenguajes de Transformación en MDD, junto con un análisis y evaluación comparativa con un esquema propuesto para tal fin
- Una revisión del estado del arte en *Traceability*, enfocado a los contextos de interés del presente trabajo.

1.3 Publicaciones

Parte de la investigación y realización del trabajo relacionado con esta tesina, ha sido publicado en distintos congresos como artículos científicos que a continuación se enuncian:

- ***"Integrando Trazabilidad a la Generación de Casos de Prueba del Sistema: una propuesta MDD/MBT".***
  Natalia Correa, Roxana Giandini. XVI Conferencia Iberoamericana de "Software Engineering" (**CIbSE 2013**) (2013). ISBN: 978-9974-8379-1-1.

- ***"Casos de Prueba del Sistema Generados en el Contexto MDD/MBT".***
  Natalia Correa, Roxana Giandini. Simposio Argentino de Ingeniería de software ASSE 2012. Anales de Jornadas Argentinas de Informática **41 JAIIO** (2012). ISSN: 1850-2792.

- "***Generación Automática de Casos de Prueba a partir de Casos de Uso: Una Propuesta Basada en MDD/MBT".***
  Natalia Correa, Roxana Giandini. Simposio Argentino de Ingeniería de software ASSE 2011. Anales de Jornadas Argentinas de Informática **40 JAIIO** (2011). ISSN 1850-2792

- ***"Lenguajes de Transformación de Modelos. Un análisis comparativo".***
  Natalia Correa, Roxana Giandini. Congreso Argentino de Ciencias de la Computación **CACIC**. (2007). ISBN 950-656-109-3

El Anexo I incluye el texto completo de estas publicaciones.

1.4 Organización de la tesina

El presente trabajo se estructura de la siguiente manera:

- En el capítulo 2 se presentan los conceptos básicos necesarios para la comprensión del contexto en el que se realiza nuestra propuesta: el paradigma MDA/MDD, el enfoque específico al área de *testing* y de interés para este trabajo: MBT, además de las nociones de metamodelado, arquitectura de 4 capas y el lenguaje de modelado UML.

- En el capítulo 3 se recopilan y estudian los diferentes lenguajes para definir transformaciones de modelos en MDD. Se describen con mayor detalle dos de estos lenguajes y herramientas: ATL [9] y MofScript [10].

- El capítulo 4 presenta el concepto de trazabilidad en Ingeniería de Software y en particular dentro de los contextos MDD y MBT. Se mencionan las distintas propuestas existentes en estos contextos y se analizan las diferencias con el nuestro trabajo.

- En el capítulo 5 se presenta nuestra propuesta: el proceso de generación de los casos de prueba, en dos partes.
  En primera instancia se presentan los pasos necesarios para generar casos de prueba del sistema. En segundo lugar, se completa el proceso con la definición de un (sub)proceso para especificar casos de prueba a partir de la documentación de cada caso de uso.
  En ambos casos, se menciona cómo se incluye la generación de modelos de trazas.

- La implementación de este proceso se detalla en el capítulo 6. Allí se mencionan la definición de perfiles UML, la definición de transformaciones de modelo y la generación de los modelos intermedios necesarios para generar los casos de prueba.

- En el capítulo 7 se presenta un caso de estudio al que se aplica el proceso definido en esta tesina. Se generan todos los modelos que participan de la definición del proceso siguiendo los pasos definidos en el mismo y utilizando las herramientas estudiadas en los capítulos previos

- Por último, el capítulo 8 expone las conclusiones de la tesina, incluyendo los trabajos relacionados con nuestro trabajo, los aportes de la tesina y el trabajo a futuro.

Capítulo 2- Conceptos Básicos

En este capítulo presentamos los conceptos básicos necesarios para la comprensión del contexto en el que se realiza nuestra propuesta. Así, la primera sección está dedicada a presentar el Desarrollo de Software Dirigido por Modelos incluyendo -en diferentes subsecciones- detalles de su implementación más conocida: MDA (*Model Driven Architecture*), la técnica de metamodelado, la arquitectura de modelado de 4 capas y el concepto de transformación y su definición.
La segunda sección introduce conceptos de *testing* y presenta el Testing Basado en Modelos, analizando esta nueva metodología y el beneficio de aplicarla en el testeo de sistemas de software.
La sección 3 cierra el capítulo con nociones del lenguaje de modelado UML, necesarios para la definición de metamodelos- y sus mecanismos de extensión.

2.1 El Desarrollo de Software Dirigido por Modelos

En el capítulo introductorio mencionamos que MDD se convirtió en un nuevo paradigma de la Ingeniería de Software y que entre sus objetivos generales se encuentra la mejora del proceso de construcción de software.

MDD se basa en un proceso guiado por modelos; es decir que el desarrollo de un sistema de software se produce a partir de modelos y herramientas adecuadas que permitan generar una aplicación. Esto es así porque la idea más fuerte del paradigma es considerar que "todo es un modelo" –de la misma forma que en el paradigma de la Programación Orientada a Objetos "todo es un objeto"-.
MDD permite el uso y el aprovechamiento de los modelos para simular, estimar, comprender, comunicar y producir código [4].
Estos modelos se van generando como parte del desarrollo de los sistemas desde los más abstractos a los más concretos. Por medio de transformaciones modelo a modelo o bien modelo a texto, los modelos se van "refinando" sucesivamente hasta llegar al código final de la aplicación a partir de la última transformación.

Como objetivos del Desarrollo Dirigido por Modelos podemos mencionar los siguientes, identificados por diferentes autores:

- Automatizar muchas de las tareas de programación complejas (y rutinarias) que aún hoy se realizan manualmente, aumentando los niveles de abstracción en que los desarrolladores escriben programas. Se menciona particularmente la derivación automática de programas [11]
- Utilizar un mayor nivel de abstracción en la especificación del problema a resolver y de su solución [3]
- Enfatizar el uso de la automatización asistida por computadora para soportar el análisis, el diseño y la ejecución de sistemas software [3]
- Adherir al uso de estándares industriales como medio para facilitar las comunicaciones, la interacción entre diferentes aplicaciones y productos y la especialización tecnológica [3]
- En [4] lo explicitan como "modelar una vez, utilizar en cualquier lado", en el sentido de aprovechar los esfuerzos de análisis y diseño en la creación de

modelos, para que luego las aplicaciones puedan generarse para distintas plataformas.

Los beneficios que reporta el uso de MDD según diferentes autores son presentados a continuación:

- *Productividad*: el aumento de la productividad es una consecuencia del uso de altos niveles de abstracción ya que le permite a los desarrolladores concentrarse en los ítems centrales del dominio de la aplicación trabajando independientemente de los detalles de las plataformas.
- *Portabilidad*: mantener modelos independientes de las tecnologías, plataformas o bien dispositivos donde se ejecutan las aplicaciones es lo que logra el objetivo de la portabilidad en MDD.
- *Reusabilidad*: estos modelos independientes de las tecnologías, pueden reutilizarse para obtener o derivar a partir de ellos otros modelos o aplicaciones para diferentes tecnologías.
- *Interoperabilidad*: las transformaciones modelo a modelo o modelo a código identifican claramente cómo se relacionan el origen y el destino de una transformación determinada. Esto permite que haya interoperabilidad tanto a nivel de modelos como de código.
- *Mantenimiento*: al realizar un cambio en un modelo deben ejecutarse nuevamente las transformaciones para reflejar los cambios en los modelos generados a partir de este y así siguiendo hasta el código final. De esta forma y por medio de las transformaciones, las modificaciones se propagan desde su origen hasta el código final, manteniendo la consistencia y facilitando la evolución del sistema en construcción.
- *Documentación*: esta se encuentra en los mismos modelos ya que son la representación, en diferentes niveles de abstracción, de la aplicación.

2.1.1 MDA: La Arquitectura Dirigida por Modelos

Dentro del ámbito MDD, existen diferentes estándares que implementan de manera particular sus principios. Entre estos cabe mencionar a la iniciativa más conocida y extendida: *Model Driven Architecture* (MDA) [12], un framework para desarrollo de software, patrocinado por la OMG [13] y presentada en el año 2000 con el objetivo de afrontar los desafíos de integración de las aplicaciones y los continuos cambios tecnológicos. MDA propone además el uso de un conjunto de estándares como MOF [10], UML[7] y XMI [14].

A continuación se detallan los tipos de modelo que son el *core* de MDA:

- **Modelo Independiente de la Computación (CIM -*Computation Independent Model*-)**

Un CIM representa una vista del sistema desde un punto de vista independiente de la computación y que, por lo tanto, no muestra detalles de la estructura del sistema. Se lo denomina comúnmente "modelo del dominio".

- **Modelo Independiente de la Plataforma (PIM - *Platform Independent Model*-)**

Un PIM es un modelo con un alto nivel de abstracción independiente de cualquier tecnología de implementación.

- **Modelo Específico de la Plataforma (PSM - *Platform Specific Model*-)**

Un PSM es un modelo adaptado para especificar el sistema en construcción pero ahora en términos de plataformas o tecnologías determinadas.

Un modelo PIM puede transformarse en uno o más PSMs.

- **Modelo de la Implementación -o simplemente- Código**

Es una descripción del sistema en código fuente. Cada PSM se transforma en código.

Es notable en MDA, a partir de los modelos presentados, la clara separación que existe entre las especificaciones de los sistemas de sus posibles implementaciones en tecnologías concretas. En términos de MDA, se distinguen PIMs de PSMs de código.

El ciclo de desarrollo de MDA, que tiene en cuenta a los tipos de modelos mencionados arriba, se explica a continuación y se visualiza en la Figura 2.1.
El código de una aplicación se puede generar a partir de un modelo PIM, mediante sucesivas transformaciones de modelos hasta llegar al código fuente.
A partir de un modelo PIM que representa un sistema, pueden generarse varios PSMs y a partir de ello, el código. Esta idea permitiría que la aplicación posea las características de portabilidad, interoperabilidad, mantenimiento y reusabilidad necesarias para un sistema sólido.
Además, puede suceder que existan más de tres niveles de abstracción y pueden existir transformaciones intermedias entre modelos PIM/PSM, donde el modelo de salida de una transformación es la entrada para la siguiente, generando así una cadena hasta llegar a la generación de código [15].

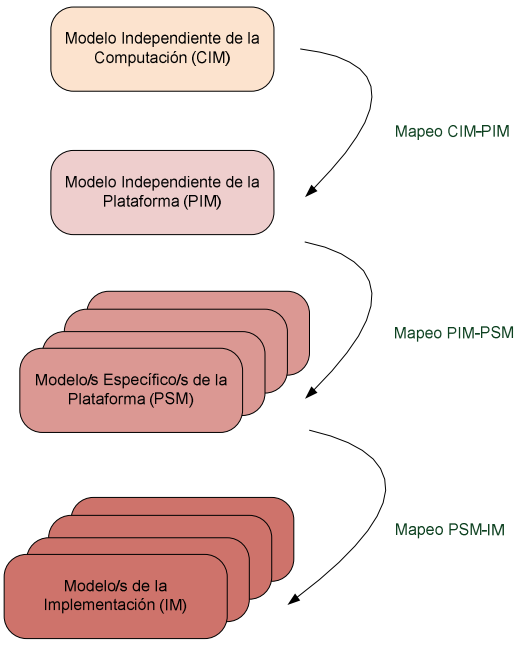


Figura 2.1. Ciclo de Desarrollo en MDA

Las transformaciones entre modelos juegan un papel muy importante en MDD. Introduciremos en la sección 2.1.3 su concepto, definición y especificación.

2.1.2 Metamodelos y la Arquitectura de 4 capas

Años atrás, la técnica usual para definir un lenguaje era utilizar BNF (*Backus–Naur Form*) [16].
BNF es un metalenguaje usado para expresar gramáticas libres de contexto, vale decir, una manera formal de describir cuáles son los elementos básicos y cómo deben combinarse para formar expresiones correctas dentro de un lenguaje. BNF ha sido muy utilizado para definir las gramáticas de los lenguajes de programación.
Con la aparición de los lenguajes gráficos, fue necesario buscar una técnica diferente para definirlos -BNF se ajusta muy bien a los lenguajes textuales pero no a los gráficos, donde básicamente hay una diferencia entre la sintaxis concreta y la sintaxis abstracta de los elementos que conforman el lenguaje-.
Es así que aparece el 'metamodelado' como técnica para definir lenguajes gráficos de modelado. Un metamodelo permite definir un lenguaje por medio de un modelo, indicando qué elementos pertenecen al lenguaje y de qué forma pueden ser combinados[2].
A modo de ejemplo, UML -lenguaje gráfico estándar- describe que dentro de un modelo pueden existir los elementos casos de uso, actores, asociaciones, etc. y que para indicar que un rol determinado ejecuta una función, un actor se relaciona mediante una asociación con un caso de uso.

La 'Arquitectura de 4 capas de Modelado' es la propuesta de OMG orientada a estandarizar conceptos relacionados al modelado, desde los más abstractos a los más concretos. A continuación se mencionan las capas y se describe cada una, desde la más concreta a la más abstracta:

- **M0: Instancias**

El nivel M0 representa todas las instancias reales del sistema, es decir, los objetos de la aplicación.
Por ejemplo, los usuarios/ clientes de un sistema de venta de libros, cada copia de libro, cada compra del sistema pertenecen a esta capa.

- **M1: Modelo del Dominio o del Sistema**

Esta capa representa a los conceptos del sistema; son categorías de las instancias del nivel M0.
Por ejemplo, las clases de diseño del sistema de venta de libros (Cliente, Copia, Libro, Compra) que modelan a las instancias nombradas en el nivel M0, pertenecen a esta capa.

- **M2: MetaModelo**

Esta capa representa a los conceptos del lenguaje utilizado para definir los modelos del nivel M1. De la misma forma en que los elementos de M0 son instancias de M1, los elementos de M1 son instancias de M2.
Por ejemplo, las metaclases Class y Association que al instanciarlas crean una clase o una asociación entre clases. Los representantes aquí son metamodelos como UML, OCL, Java.

- **M3: Meta-MetaModelo**

Un meta-metamodelo es un modelo que define el lenguaje para representar un metamodelo. La relación entre un meta-metamodelo y un metamodelo es análoga a la relación entre un metamodelo y un modelo [3].
El nivel M3 es el nivel más abstracto, que permite definir metamodelos concretos. A

[2] Como un metamodelo es también un modelo, el metamodelo en sí mismo debe estar escrito en un lenguaje bien definido. Este lenguaje se llama metalenguaje [3]

modo de comentario, la OMG ha definido a MOF (*Meta-Object Facility*) [17] como el lenguaje estándar de la capa M3 por lo cual todos los metamodelos de la capa M2, son instancias de MOF. No existe otro nivel por encima de MOF y básicamente MOF se define a sí mismo.
Por ejemplo, las meta-metaclases Class y Association y el conjunto de meta-meta elementos que permite crear un metamodelo.

La Figura 2.2 muestra una vista general de la arquitectura de 4 capas indicando las relaciones existentes entre los elementos de diferentes niveles.

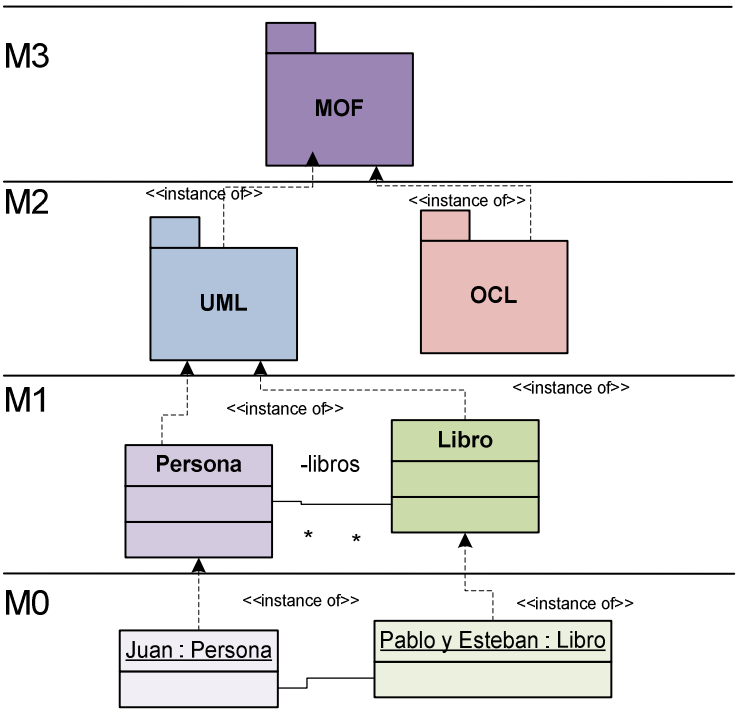


Figura 2.2. Arquitectura de 4 capas

Puede verse que en la capa M3 se encuentra el meta-metamodelo MOF, a partir del cual se pueden definir distintos metamodelos que pertenecen al nivel M2. Ejemplos de ello son UML y OCL. Las instancias de estos metamodelos serán, por ejemplo, los modelos UML. Estos pertenecen al nivel M1. Correlativamente, los objetos creados en tiempo de ejecución correspondientes a estos modelos son los elementos del nivel M0.

2.1.3 Transformaciones de Modelos

En secciones anteriores mencionábamos que las transformaciones de modelos juegan un rol muy importante en MDD. De hecho, son trascendentes.
Esta idea puede verse asociada a la representación del ciclo de desarrollo de MDD (Figura 2.1), donde los distintos tipos de modelos de MDA se 'mapean' unos con otros. Una herramienta que de soporte al framework MDA tomará un modelo PIM como entrada y lo transformará en un PSM. Así siguiendo, tomará ese modelo PSM y lo transformará en código.

La MDA Guide [12] define la transformación de modelos como "el proceso de convertir un modelo en otro modelo del mismo sistema".

Estas ideas y la definición mencionada, presentan a la transformación como una 'caja negra' que a partir de un modelo fuente genera un modelo destino.
Para poner un poco de luz sobre esta 'caja negra', podemos decir que hay dos conceptos para diferenciar. Uno es la transformación en sí -el proceso de generar un modelo a partir de otro- y el otro concepto es su especificación. Este último concepto es la 'definición de la transformación', es decir, la definición que describe cómo se debe transformar el modelo fuente para producir el modelo destino.

Aquí podemos comprender que para escribir las definiciones de transformación serán necesarios lenguajes de transformación especiales para establecer las correspondencias entre dos lenguajes de modelado (o entre un lenguaje de modelado y un lenguaje de programación) [3].

La Figura 2.3 muestra la relación entre modelos, definiciones de transformación y herramientas de transformación.

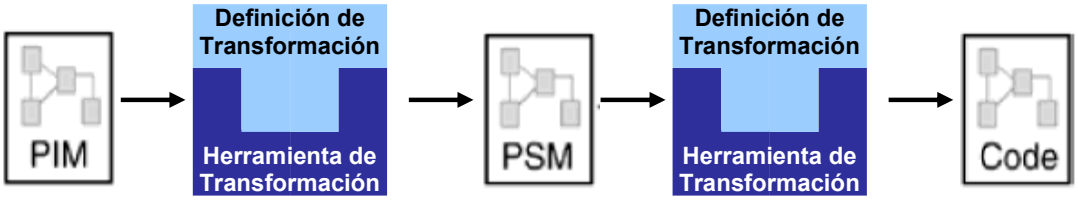


Figura 2.3 – Las definiciones de transformaciones dentro de las herramientas de transformación

A manera de ejemplo, se podría definir una transformación que relacione los lenguajes UML y Java. O sea, tomando como entrada un modelo de clases UML se especifica cómo se genera como salida un modelo en lenguaje Java. Las reglas de transformación -de la transformación- describirían cómo una clase UML se transformaría en una clase Java en el modelo destino.

Kleppe, menciona en [2] que:

- "una definición de *transformación* es un conjunto de reglas de transformación que juntas describen cómo un modelo en el lenguaje fuente puede ser transformado en un modelo en el lenguaje destino", y que

- "una *regla de transformación* es una descripción de cómo una o más construcciones en el lenguaje fuente pueden ser transformadas en una o más construcciones en el lenguaje destino".

En el momento de definir las transformaciones, se debe elegir el lenguaje en el que se escribirá la transformación así como también las reglas que especificarán cómo los elementos de un modelo generarán elementos en otro modelo.
Existen una serie de lenguajes de transformaciones de modelos que estudiaremos en próximos capítulos (puntualmente en el capítulo 3). En este punto cabe mencionar al lenguaje QVT (*Query/View/Transformation*) [18] el cual es el lenguaje estándar de OMG para transformaciones.

Para completar esta idea, introducimos la Figura 2.4.

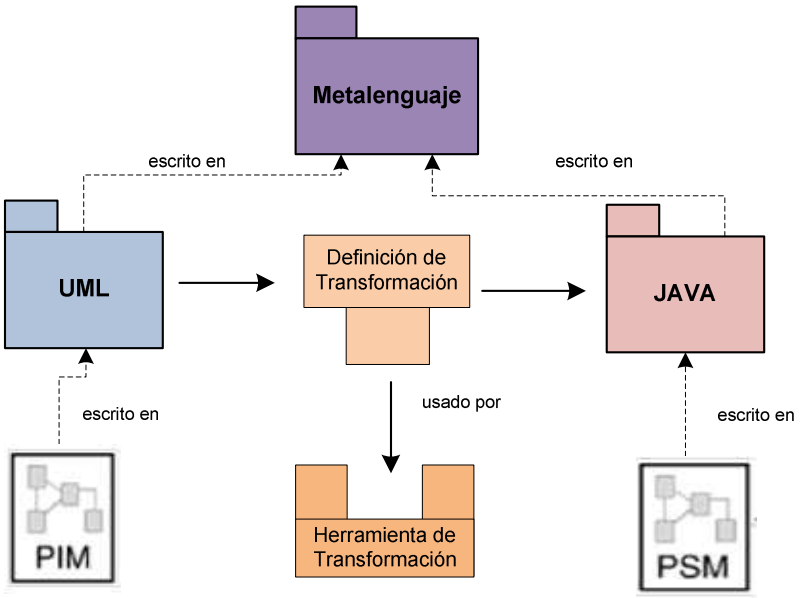


Figura 2.4 – Visión de MDA: modelos, transformaciones, lenguajes y metalenguajes

La definición de la transformación necesitará nombrar y tratar a los elementos de los modelos de origen y destino uniformemente. Eso requerirá que los lenguajes de los modelos sean instancias de un metalenguaje o metamodelo -por supuesto más abstracto- que nos permita establecer relaciones entre elementos dentro de la transformación. Es por esto que se ha introducido en la sección anterior las nociones de metamodelado y la arquitectura de 4 capas.

2.2 Testing Basado en Modelos

"*El éxito del testing depende fuertemente de nuestra capacidad para crear pruebas útiles, para descubrir y eliminar problemas en todas las etapas del desarrollo*" [19]. Esta capacidad mencionada por Beizer requiere que los casos de prueba tengan el alcance y el tipo necesario para cada caso y que además, se diseñen y definan desde una etapa temprana del proceso de desarrollo.
El acto de diseñar *tests* es uno de los mecanismos conocidos más efectivos para prevenir errores (en distintos contextos en general y en particular, se puede aplicar al área de sistemas). El proceso mental que debe desarrollarse para crear tests útiles puede descubrir y eliminar problemas en todas las etapas del desarrollo de software.
Existen distintos tipos de *testing*. Tomando la taxonomía descripta en [20], se describen a continuación:

- **Test de Unidad** (*Unit Testing*), que requiere testear una única unidad a la vez;
- **Test de Composición** (*Component Testing*), que testea cada componente/subsistema de forma separada;
- **Test de Integridad** (*Integration Testing*), que busca asegurar que los distintos componentes del sistema funcionan correctamente juntos;
- **Test del Sistema** (*System Testing*) se basa en testear el sistema como un todo.

La Figura 2.5 muestra la clasificación mencionada sobre un eje que representa al

Sistema bajo testeo (SUT- *System Under Testing*), agregando 2 'ejes' más y ubicando MBT en el contexto de los procesos y tipos de *testing*.

El eje horizontal refleja a los *tests* de caja negra y de caja blanca. En el primer caso, se generan las pruebas a partir de los requerimientos del sistema que describen su comportamiento; no se 've' la estructura interna del mismo: es una caja negra.
En el segundo caso, los *tests* de caja blanca se basan en el examen de los detalles procedimentales.

El eje diagonal, especifica distintas características deseables de ser probadas. Allí se mencionan al *Testing Funcional* -que busca encontrar errores en la funcionalidad del sistema-; *Testing de Robustez* –que busca encontrar errores bajo determinadas condiciones (entradas inesperadas, aplicaciones dependientes o no disponibles, etc.); *Test de Usabilidad* –que busca encontrar problemas de interfaz de usuario-.

En la figura mencionada, se establece el alcance de MBT dentro del amplio contexto del *Testing*. Se ve, por ejemplo, que MBT –descripto a continuación- puede aplicarse en cualquiera de los niveles de *tests* del eje vertical.

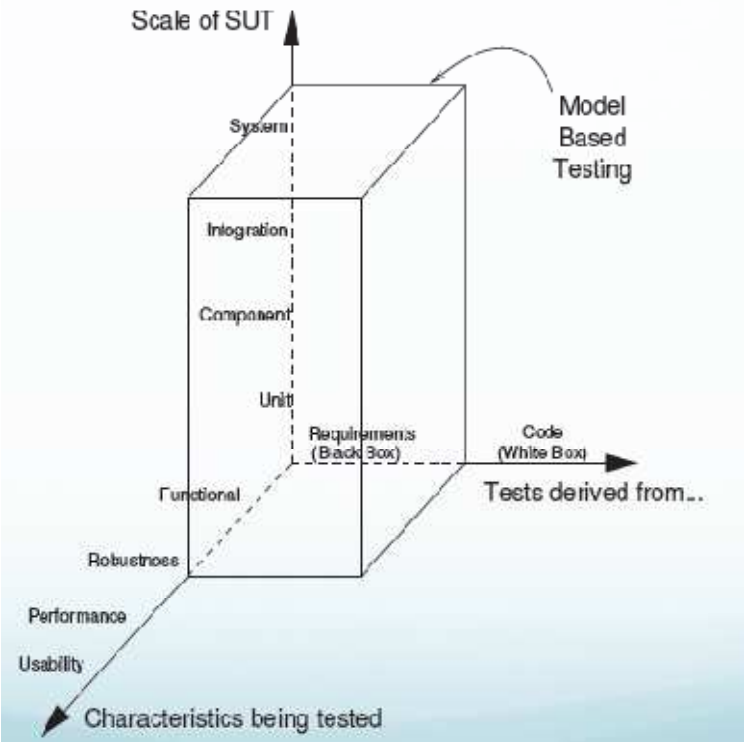


Figura 2.5 – *Testing*: Tipos y visión de MBT en ese contexto

Dada la complejidad per se del proceso de testing dentro del proceso de desarrollo, es deseable y necesario que cuente con un proceso definido y que pueda ser automatizado, parcial o totalmente.

Model Based Testing (MBT) [20], [21], [22], [23], que sigue los lineamientos y que podríamos incluir por sus características en el paradigma MDD, es una propuesta reciente que desafía la problemática de generar los modelos y artefactos necesarios para el testeo de software ya que los modelos de *testing* son derivados total o parcialmente desde modelos que describen la funcionalidad del sistema en desarrollo.

Este último, visto desde la perspectiva del *testing* se conoce como SUT (*System Under Test*- Sistema bajo Testeo).

El SUT puede ser un artefacto como una clase, puede ser también un método o puede ser tan complejo como un sistema software completo. Para el *testing*, los modelos del sistema en desarrollo proveen una descripción de la funcionalidad del mismo.
A partir de estas descripciones/comportamiento especificado en los modelos del SUT, es posible generar un conjunto de casos de prueba que se pueden utilizar para determinar si el sistema se ajusta a ciertos requerimientos o propiedades deseables para el sistema -representado en los modelos-.

MBT define una forma de prueba de caja negra –o testing funcional- que utiliza modelos estructurales y de comportamiento- para automatizar la generación de casos de prueba.

MBT se centra en la modelización de los sistemas, donde un modelo es la descripción del comportamiento de un sistema que ayuda a entender el funcionamiento del mismo.
La idea general de este paradigma presenta lo siguiente: "Si a partir de este mismo modelo se generan los *tests* para verificar su comportamiento, cuando el comportamiento del sistema cambie, también lo harán los tests" [21]. Esta visión general de MBT se presenta en la figura 2.6.

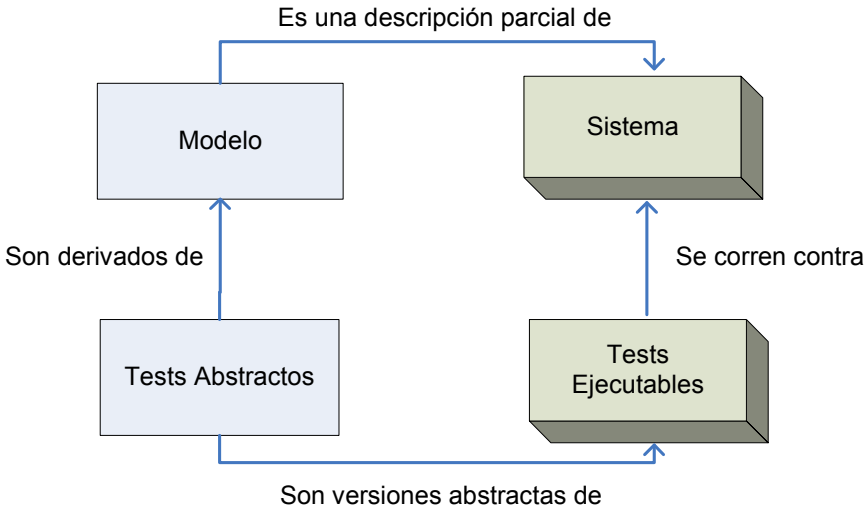


Figura 2.6 – Esquema general de *Testing* Basado en Modelos

El proceso de MBT se presenta en cinco pasos generales, como se detalla en [20] y en [23]:

1. **Modelar el SUT y/o su entorno**: un modelo abstracto del sistema a testear (un modelo más simple o no totalmente completo del SUT)
2. **Generar casos de prueba abstractos a partir del modelo anterior:** se debe seleccionar un criterio de selección de *tests* para filtrar los que queramos generar desde el modelo. Al surgir de un modelo que utiliza una vista simplificada del SUT, estos *tests* carecen de la totalidad de los detalles necesarios por el SUT y no son directamente ejecutables.

3. **Concretizar los casos de prueba abstractos para hacerlos ejecutables**: puede ser realizado por una herramienta de transformación (de caso de prueba abstracto a script ejecutable) o escribiendo un script adaptador que capture al SUT e implemente cada operación abstracta en términos de las facilidades de más bajo nivel del SUT.
   El objetivo de este paso es eliminar la brecha entre los *tests* abstractos y el SUT concreto, adicionando detalles de bajo nivel del SUT que no hayan sido mencionados en el modelo abstracto.
4. **Ejecutar los casos de prueba en el SUT y asignar veredictos**: se pueden ejecutar con distintas herramientas (MBT u otra)
5. **Analizar los resultados de los casos de prueba y realizar cambios adecuados**: cuando un test falla, se debe a una falla del SUT o una falla del caso de prueba mismo. Como se utiliza MBT, una falla en el caso de prueba debe deberse a una falla en el código adaptador o en el modelo (e incluso quizás en el documento de requerimientos); por lo que aquí se obtiene *feedback* sobre la correctitud del modelo.

Es de notar que los pasos 4 y 5 son parte de cualquier proceso de *testing*, incluído el *testing* manual. El paso 3 es similar a la fase adaptativa del testeo basado en palabras clave, donde el significado de cada palabra clave es definido.
Los primeros dos pasos distinguen al MBT de cualquier otro tipo de *testing*.

MBT presenta asimismo y como noción dentro del paradigma, la generación automática de *tests*. En [21] los autores mencionan a las Máquinas Finitas de Estado (FSM) como los generadores de casos de prueba más utilizados ya que proveen propiedades, como estados y transiciones que las aplicaciones -o bien el sistema- poseen. Estas propiedades son las que pueden ser comparadas en los casos de prueba.
Se mencionan además otras herramientas utilizadas para el proceso de generación de casos de prueba, entre ellos las redes de Petri, destacando que el proceso en sí es muy similar para todos los formalismos.

En definitiva, una especificación es 'traducida' a un modelo formal del cual son generados los casos de prueba y la elección del modelo formal, del mecanismo por el cual se genera este y luego a su vez los casos de prueba son aún propuestas que se están estudiando y probando dentro de la Ingeniería de Software.

2.3 El Lenguaje Unificado de Modelado

Entre los conceptos necesarios para presentar nuestra propuesta se incluye a UML, el Lenguaje Unificado de Modelado (*Unified Modeling Language*) [7]. UML es un lenguaje de modelado de sistemas de software, de propósito general y que se encuentra entre los más conocidos y utilizados en la actualidad. Es un estándar aprobado por la ISO como ISO/IEC19501 desde el año 2005 y respaldado por la OMG.
UML es un lenguaje gráfico que permite visualizar, especificar, construir y documentar los artefactos de un sistema.
Como características del lenguaje podemos decir que UML es útil en las diferentes etapas del ciclo de vida del desarrollo de sistemas; es independiente del proceso de desarrollo de software ya que puede utilizarse con diferentes paradigmas y es

independiente del lenguaje de implementación que finalmente se utilice para codificar los modelos.

Entre las facilidades provistas podemos decir que UML permite:

- Definir los límites del sistema y sus principales funciones mediante Casos de Uso y actores.
- Ilustrar el funcionamiento de un caso de uso mediante Diagramas de Actividad o de Interacción
- Representar la estructura de un sistema mediante Diagramas de Clases
- Modelar el comportamiento de los objetos mediante Diagramas de Máquinas de Estados
- Describir la arquitectura de la implementación física con Diagramas de Componentes y Despliegue
- Extender la funcionalidad de elementos estándar mediante estereotipos
- Proveer base formal para los diagramas (metamodelo, OCL)

UML cuenta con varios tipos de diagramas, los cuales muestran diferentes aspectos del sistema representado. Estos diagramas se presentan a continuación en la Figura 2.7.

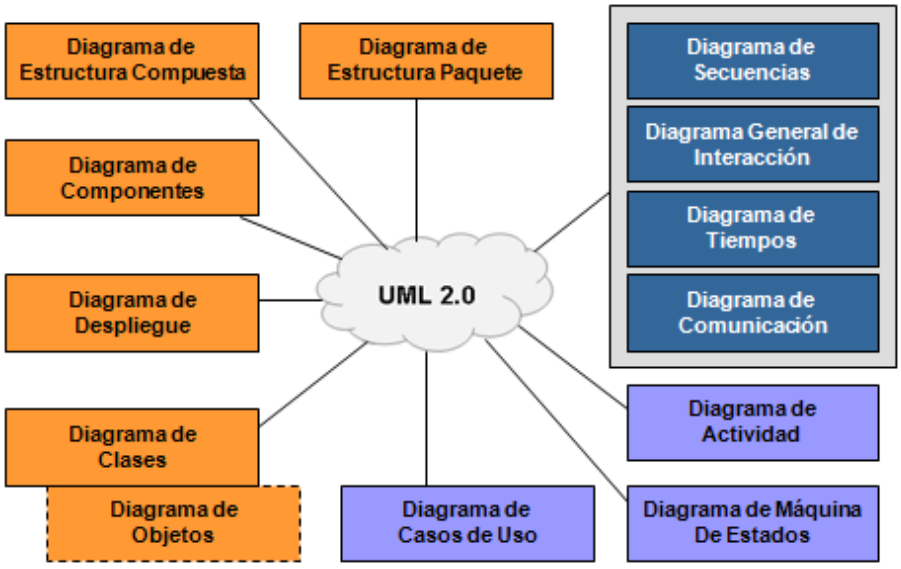


Figura 2.7 – Conjunto de diagramas de UML 2

Para nuestra propuesta, utilizaremos básicamente los diagramas de Casos de Uso y los diagramas de Actividades que a continuación describimos brevemente.

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

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

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

Es un hecho que ningún lenguaje puede ser suficiente para expresar todos los matices de todos los modelos en todos los dominios y en todo momento. Y es por esta razón que UML, mediante sus *mecanismos de extensión*, permite agregar a un modelo conceptos importantes de su dominio que el lenguaje no provee.
De esta manera podemos decir que UML ha sido diseñado para ser extensible y rígido al mismo tiempo, haciendo posible extender el lenguaje de forma controlada.

UML incluye tres construcciones principales de extensión que a continuación se describen brevemente:

- **Restricciones**: es una declaración textual de una relación semántica expresada en un cierto lenguaje formal (OCL) ó en lenguaje natural.
- **Estereotipos**: una nueva clase de elemento, ideada por el modelador y basada en un tipo existente de elemento del lenguaje.
- **Valores etiquetados**: reúne información -con nombre- unida a cualquier elemento de un modelo.

Al extender UML mediante estereotipos y perfiles se pueden agregar nuevos conceptos específicos al contexto que estamos modelando.
Los estereotipos permiten extender el vocabulario de UML para crear nuevos elementos del modelo, derivados de otros existentes, pero que tengan propiedades específicas del dominio.
Un perfil de UML es un conjunto de extensiones que especializan a UML para su uso en un dominio o contexto particular. Los perfiles están basados en estereotipos adicionales y valores etiquetados que son aplicados a los elementos, atributos y métodos entre otros.

Para nuestra propuesta vamos a valernos de la extensión del lenguaje por medio de estereotipos, definiendo dos perfiles UML. Presentaremos en próximos capítulos su definición y sus restricciones escritas formalmente en OCL.

2.4 Resumen del Capítulo

En este capítulo hemos presentado los conceptos fundamentales del Desarrollo de Software Dirigido por Modelos (MDD) y el Testing Basado en Modelos (MBT).
En nuestra propuesta, establecemos la relación entre ambos paradigmas, ya que para la generación de los modelos formales de *testing* que realizamos nos valemos de transformaciones entre modelos. Además, focalizamos nuestro trabajo en los 2 primeros pasos del proceso MBT, ya que son éstos lo que lo distinguen de cualquier otro tipo de *testing*.
Esta generación de modelos/ casos de prueba se realiza de forma automática, partiendo de los modelos de requerimientos, como son los de Casos de Uso de UML. Para agregar semántica y elementos necesarios para definir las transformaciones que nos permitan generar casos de prueba, fue necesario definir además una extensión del lenguaje UML con perfiles para testing.

Al elegir como contextos de esta tesina a las propuestas MDD y MBT, se aprovechan los

beneficios aportados por ambos paradigmas. A los aportes ya conocidos de MDD, se suman los del enfoque MBT en cuanto a la reducción del costo de mantenimiento de las pruebas, ya que automatizar la generación de casos de prueba, se regeneran los *tests* con los cambios en los requerimientos -y el SUT-. Lo contrario sería mantener el grupo de tests *(Test suite) más* manualmente.

Al elegir a los Casos de Uso como modelo inicial en el proceso de generación de casos de prueba del sistema, es posible obtener ciertos elementos de importancia para los *tests* y la etapa de Pruebas, entre ellas:

- Cuáles son los objetivos de prueba
- Qué acciones debe realizar una prueba (pero no cómo realizarlas)
- Cuál es el resultado esperado (pero no cómo están definidos esos resultados)

A nivel más general, se relacionan con otros aspectos como son:

- La cobertura de los casos de prueba (una medida de ella)
- La determinación del orden de los casos de prueba

Capítulo 3- Lenguajes de Transformación de Modelos

Este capítulo presenta a los lenguajes de transformación de modelos así como un análisis y comparación entre ellos, según una clasificación particular –presentada en un trabajo de investigación anterior-. Dado el contexto de interés (MDD/ MBT) y el proceso que se define para el mismo, resulta de mucho interés estudiar y conocer estos lenguajes. Se advierte además que resulta sumamente útil para toda la comunidad informática que desee comenzar a involucrarse y conocer más sobre estas propuestas.

3.1 Introducción

Como se mencionó en el capítulo anterior, la propuesta MDA (*Model Driven Architecture*- Arquitectura Dirigida por Modelos) [12] de la OMG (*Object Management Group*) [13] presenta un proceso de desarrollo de software concebido para dar soporte al desarrollo de sistemas, donde los conceptos más importantes son los modelos y las transformaciones entre ellos que generan a su vez, otros modelos.
Describir transformaciones de modelos requiere de lenguajes específicos para la definición de las mismas. Actualmente, existen varias propuestas de lenguajes, muchas de ellas basadas en el estándar de la OMG QVT (*Query/View/Transformation*) [18].

Entre los lenguajes definidos existen gran variedad de "tipos", con diferentes características: icónicos y textuales; declarativos, operacionales y declarativos-operacionales; algunos basados en QVT -y otros no-, están aquellos que son compatibles con MOF; aquellos que soportan OCL, los que proveen trazabilidad entre los elementos de los modelos; los que proveen composición de transformaciones y hasta aquellos que proveen una herramienta CASE o un *plugin* para Eclipse.

Estudiar y conocer los lenguajes de transformación que sirven de soporte a la propuesta MDA es útil a toda la comunidad del área de sistemas interesada en la temática abordada. En nuestro caso en particular, al momento de elegir un lenguaje/ herramienta que nos permitiera definir las transformaciones de modelo que son parte del proceso definido en este trabajo fue necesario ahondar en las diferentes propuestas así como las características que cada uno presentaba.
De las varias propuestas existentes, se destacan en el presente capítulo algunas de ellas.

Como trabajo adicional, se definió una extensión a una clasificación de lenguajes existente. La misma fue parte de un trabajo de investigación previo y se presenta en el Anexo III. Tomando a esta clasificación como referencia, se analizaron algunos de los lenguajes que más crecimiento han tenido y que son los más utilizados y nombrados en la comunidad MDA.

3.2 Lenguajes de Transformación de Modelos. Propuestas Existentes

Nuestro trabajo comenzó con la investigación de los lenguajes existentes y la publicación de estos resultados en [24]. En esta tesina, presentamos una actualización de dicha recopilación dado el tiempo transcurrido desde la publicación del trabajo mencionado.

Como se ve en la Tabla 1, las propuestas actuales son muchas y muy diversas. La lista de lenguajes incluye una breve referencia de cada uno de ellos.

Tabla 1.Lenguajes de transformación de modelos. Una lista preliminar

Lenguaje	**Características**
ATL (Atlas Transformation Language) [26], [27]	Lenguaje de transformación de modelos y herramienta en Eclipse desarrollada por el Atlas Group (INRIA).
Epsilon [28]	Se trata de una familia: lenguajes y herramientas para transformación de modelos, validación de modelos, comparación de modelos, etc.
GreAT (Graph Rewriting and Transformations) [29]	Basado en la transformación de grafos. Es la propuesta de una organización independiente: ESCHER Research Institute (*The Embedded Systems Consortium for Hybrid and Embedded Research*).
JTL (Janus Transformation Language) [30], [31]	Lenguaje de transformación de modelos bidireccional y con propagación de cambios. Es un framework para Eclipse presentado como 'Prototipo'.
Kent o KMTL (Kent Model Transformation Language) [32]	Propuesta realizada por la Universidad de Kent.
MTRANS [33]	Proyecto de la Universidad de Nantes. Es un framework que permite expresar transformaciones de modelos.
MOFScript [10], [34]	Lenguaje de transformación modelo a texto (cuya propuesta pertenece a la OMG) y herramienta como *plugin* para Eclipse.
MOLA (MOdel transformation LAnguage) [35], [36]	Lenguaje gráfico para describir transformaciones propuesto por la Universidad de Letonia.
MT model transformation language [37]	Basado en QVT y desarrollado como DSL (Domain Specific Language) por L. Tratt del King's College de Londres.
MTL (Model Transformation Language) [38]	Lenguaje de transformación de modelos y herramienta en Eclipse desarrollada por el Triskell team (INRIA)
QVT (Query/View/ Transformation) [18]	Especificación estándar de OMG. Está basado en MOF (Meta Object Facility) para lenguajes de transformación en MDA.
SiTra [39], [40]	Librería Java para escribir transformaciones de modelos.
Stratego [41]	Lenguaje de descripción de transformaciones de programas. La herramienta desarrollada como soporte del lenguaje es Stratego/XT
Tefkat [42]	Lenguaje declarativo basado en MOF y QVT. Es el aporte de la Universidad de Queensland.
UMLX [43]	Lenguaje gráfico que extiende a UML y también a QVT.

3.3 Clasificación para la Evaluación de Lenguajes de Transformación

En la sección anterior se ha podido ver que las propuestas de lenguajes de transformación de modelos son muchas y muy diversas. Es claro que a la hora de describir una transformación aparece un abanico de opciones de las cuales elegir y surge, como tarea adicional, la elección de un lenguaje. Resulta necesario entonces conocer las características de estos lenguajes.
Esta necesidad fue vislumbrada por Czarnecki y Helsen en [25], quienes propusieron una clasificación para lenguajes de transformación de modelos.
En nuestra propuesta, tomamos como base el trabajo realizado por ellos, reordenando algunas características y adicionando otras.

En la Figura 3.1 puede observarse en color rosa la propuesta original y en color celeste la extensión sugerida. Este trabajo de investigación fue presentado y publicado en [24].

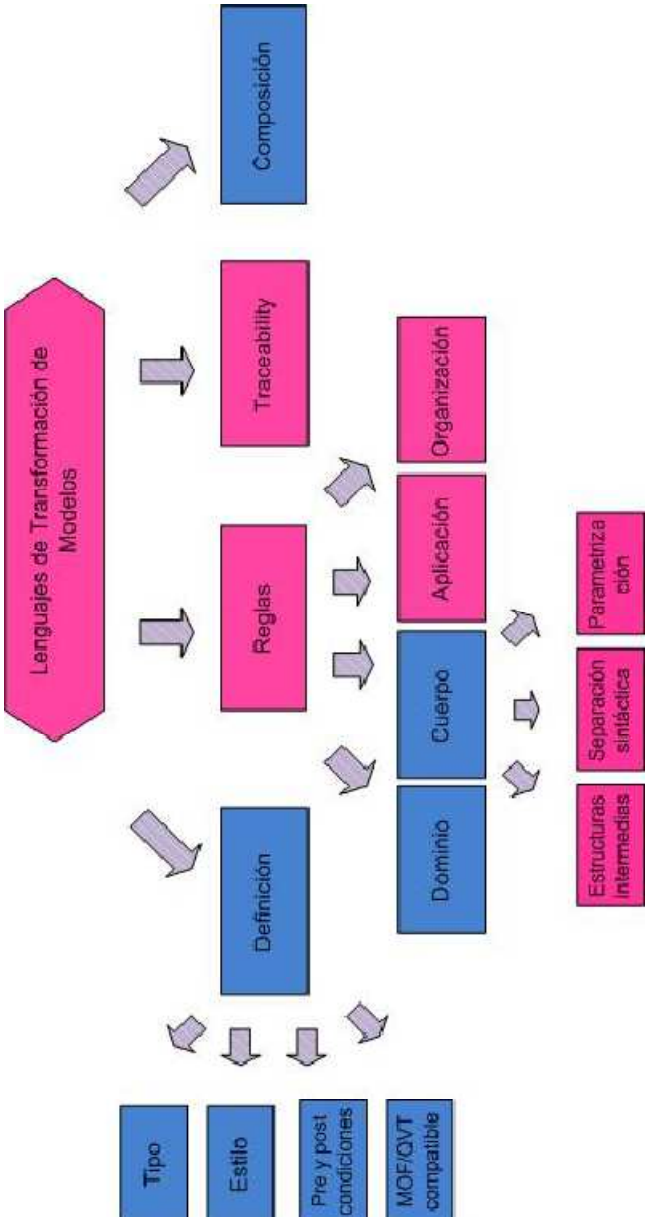


Figura 3.1. Esquema de Clasificación de Lenguajes de Modelado propuesto

En el Anexo III "Esquema de clasificación para los lenguajes de transformación" se presentan cada uno de los aspectos representados en la Figura 3.1, indicando su significado y a qué conceptos hacen referencia.

3.4 Comparación de los Lenguajes de Transformación

En base a los criterios presentados en la sección 3.3, analizamos algunos de los lenguajes de transformaciones de modelos presentados en la sección 3.2. Para tal análisis, se estudiaron y utilizaron, en la medida de lo posible, los lenguajes presentados con un ejemplo simple compuesto por dos metamodelos y al menos una transformación entre los mismos (en general, compuesta por varias reglas).

La elección de estos lenguajes se hizo en base al crecimiento y aceptación que han tenido en la comunidad MDA. Los seleccionados para el análisis presentado fueron los siguientes: **ATL**, **MOFScript**, **MOLA**, **Tefkat** y **ULMX**. Cabe aclarar que todos los lenguajes elegidos cuentan con la implementación de una herramienta que da soporte a la definición del lenguaje. Excepto MOLA, los desarrollos de tales herramientas se han hecho bajo la plataforma y como *plugins* para Eclipse.

3.4.1 ATL (Atlas Transformation Language)

ATL [26], [27] es un lenguaje de transformación de modelos híbrido que permite, en su definición de transformaciones, especificar construcciones declarativas y operativas. La propuesta es del ATLAS Group del INRIA & LINA, de la Universidad de Nantes y fue desarrollado como parte de la plataforma AMMA (*ATLAS Model Management Architecture*).

Otros detalles de la Definición del lenguaje:

- Es compatible con los estándares de la OMG: es posible describir transformaciones modelo a modelo (y ambos deben ser instancias de MOF).
- Permite definir pre y post condiciones en un lenguaje ya conocido: OCL

ATL proporciona un editor que se ejecuta sobre Java para definir transformaciones y utiliza el lenguaje propietario ATLAS, del mismo grupo de desarrollo.

En cuanto a las reglas de transformación, ATL define un dominio (metamodelo) para el source y otro dominio para el target, siendo ambos instancias de MOF. Source y target pueden tener iguales o diferentes dominios (aunque sean iguales, ambos deben ser claramente identificados). Si bien las transformaciones son unidirecciones, ATL permite la definición de transformaciones bidireccionales como la implementación de dos transformaciones, una para cada dirección.

Como características particulares del lenguaje, podemos mencionar las estructuras que define este lenguaje. En primer lugar, la definición de transformaciones forman módulos (modules) que contienen las declaraciones iniciales y un número de *helpers* y reglas de transformación. Los *helpers*, son una estructura intermedia dentro de las transformaciones que facilitan la navegación, la modularización y el reuso. Permiten definir operaciones y tuplas OCL Existe también una construcción llamada *called rule*, y que representa a un *procedure*. Estas pueden contener argumentos y pueden ser

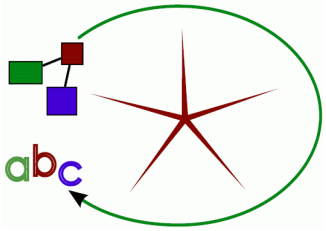
invocadas por su nombre. Resulta sumamente expresivo y de fácil escritura para quienes conocen OCL (no es necesario aprender un nuevo lenguaje).

La aplicación de las reglas se realiza de forma no determinística, por "macheo" de reglas y no se ha provisto ninguna construcción o claúsula que permita aplicar en forma condicional las reglas. Cabe mencionar que la invocación de *called rules* es determinística. Esta invocación, junto con la utilización de parámetros permiten soportar recursión.

ATL provee modularización por sus procedimientos o *called rules*, además de que sus módulos pueden incluir a otros; y mecanismos de reuso ya que las reglas se pueden heredar.
Presenta sentencias para trazabilidad entre elementos por medio de *traceability links,* conceptos del lenguaje.
Para ejecutar transformaciones compuestas es necesario ejecutar cada una de las transformaciones participantes una a una.
Aunque la sintaxis de ATL es muy similar a la de QVT, no es interoperable con este último.
No proporciona mecanismos para la validación de las transformaciones.

Para finalizar, vale la pena mencionar que ATL se ha convertido en un lenguaje tan utilizado que ya se ha definido un repositorio de transformaciones entre diversos lenguajes, documentadas y con los archivos fuente disponibles.

3.4.2 MOFScript



MOFScript [10], [34] es un lenguaje de transformación de modelo a texto presentado por la OMG. Este lenguaje presta particular atención a la manipulación de strings y de texto y al control e impresión de salida de archivos.
La herramienta está desarrollada como un *plugin* para Eclipse, que soporta el parseo, chequeo y ejecución de scripts escritos en MOFScript.

No sólo es un estándar, sino que además se basa en otros estándares de la OMG: es QVT compatible y MOF compatible con el modelo de entrada (el target siempre es texto).

Para las reglas de transformación, MOFScript define un metamodelo de entrada para el source y sobre el cual operarán las reglas. El target es generalmente un archivo de texto (o salida de texto por pantalla).

Las transformaciones son siempre unidireccionales y no es posible definir pre y post condiciones para ellas. La separación sintáctica resulta clara por la misma definición de reglas, lo que hace a la legibilidad del lenguaje. No provee estructuras intermedias, pero sí parametrización necesaria para la invocación de reglas.

En cuanto a la aplicación de reglas, podemos decir que se aplican en forma determinística y en orden secuencial. Se provee aplicación condicional e iteración de reglas. Las condiciones de aplicación se expresan en cláusulas *when*, como definición de guardas. La iteración se realiza mediante los iteradores *for each* y *while*.

MOFScript no organiza las reglas en módulos propiamente dichos. Ahora bien, en la definición de una regla se puede invocar a otras reglas, utilizando incluso parámetros,

con lo cual se asemeja bastante a un módulo. Y es posible definir jerarquías de transformaciones.

Finalmente, para *traceability*, MOFScript define un conjunto de conceptos para relacionar elementos del source con sus ubicaciones en los archivos de texto generados en el target.
En cuanto a la composición de transformaciones, no es algo que se haya pensado en este lenguaje aún.

3.4.3 MOLA (MOdel transformation LAnguage)

MOLA [35], [36] es un lenguaje gráfico de transformación de modelos, propuesto por la Universidad de Letonia. La intensión de este lenguaje es combinar la programación estructurada tradicional con reglas basadas en patrones de transformación.

La definición del lenguaje no se basó en QVT y los metamodelos que participan de la transformación pueden o no ser instancias de MOF; lo único restrictivo es que ambos deben ser definidos como dos modelos diferentes (*source* y *target*) aunque sean el mismo. El *source* es el modelo de entrada in/out (sus elementos pueden ser modificados) y el target de salida (sólo out). Las transformaciones son unidireccionales.
Si bien no se mencionan pre y post condiciones, el lenguaje provee notas donde pueden escribirse cláusulas OCL para la aplicación condicional de reglas. Estas podrían usarse entonces para pre y post condiciones.
La parte ejecutable es similar a los diagramas de actividad de UML, que comienzan en un nodo inicio y contiene elementos gráficos que representan:

- loops (while, foreach)
- reglas
- sentencias
- llamadas a procedimientos
- nodo final

En cuanto al cuerpo de la transformación, se ve que no hay separación sintáctica de los elementos que se transforman y de los que se crean. Tampoco provee estructuras intermedias.
El elemento principal del lenguaje es un concepto gráfico del *loop*, que se utiliza mucho para iterar sobre los elementos de los modelos que participan de la transformación. También para ello, permite definir variables locales a cada regla de transformación.

Sobre la aplicación de reglas, podemos decir que se aplican en forma determinística y en orden secuencial según son definidas. Existe la aplicación condicional mediante la adición de notas con cláusulas OCL. La iteración se realiza mediante dos tipos de loop: un loop de "tipo uno" que se ejecuta una vez para cada instancia válida del source (for each); un loop de "tipo dos" que continúa la ejecución mientras haya al menos una instancia válida en el source (while). El lenguaje utiliza patrones definidos para ser aplicados en las transformaciones.
Si bien gráficamente resulta intuitivo, las iteraciones pueden volverse en poco confusas, sobre todo si hay varias reglas anidadas.

En cuanto a *traceability*, MOLA permite definir "*mapping associations*" para tracear

instancias entre modelos. Estas se definen en las reglas como notas anexadas entre los elementos del dominio y codominio que participan de la transformación.

La composición de transformaciones no ha sido tenida en cuenta en la definición del lenguaje.

3.4.4 SiTra

SiTra [39], [40] es un lenguaje de transformación de modelos y un motor para la transformación de modelos. El lenguaje está basado en Java y el motor es una librería Java. La propuesta es de un grupo de investigación de la Universidad de Birmingham.

El objetivo del lenguaje es dar soporte a un enfoque de programación en Java de la codificación de las transformaciones, por un lado y en segundo lugar, proporcionar un *framework* mínimo para la ejecución de las transformaciones.
Sitra traduce esta simpleza proveyendo al usuario de dos interfaces (*Rule*, *Transformer*) como se muestra a continuación.

```
interface Rule<S,T> {
  boolean check(S source);
  T build(S source, Transformer t);
  void setProperties(T target, S source, Transformer t);
}

interface Transformer {
  Object transform(Object source);
  List<Object> transformAll(List<Object> sourceObjects);
  <S,T> T transform(Class<Rule<S,T>> ruleType, S source);
  <S,T> List<T> transformAll(Class<Rule<S,T>> ruleType, List<S> source);
}
```

Para crear los modelos *source* y *target*, se implementa, simplemente, una clase -o clases- .java. Por ejemplo, Libro, Copia, etc. Para codificar la transformación, se debe extender la interface *Transformer* codificando las reglas en java. Cabe destacar cuán rápidamente pueden codificarse transformaciones con SiTra.
Por las características de Sitra, se provee reuso, composición y separación sintáctica, todo producto del uso de Java.

No se presenta una sintaxis específica para traceability, pero puede ser incorporada de la misma forma en que se codifican las reglas, codificando explícitamente, creando instancias de Trazas.
No se basa en ningún estándar, ni conformidad o interoperabilidad con QVT

3.4.5 Tefkat

Tefkat [42] es la definición e implementación de un lenguaje para transformación de los modelos. La propuesta fue realizada por de la Universidad de Queensland, Australia.
El lenguaje transforma modelos (M2M) en forma textual y ha adoptado un paradigma declarativo. Tefkat es totalmente

compatible con los estándares de la OMG. Entiende que los modelos *source* y *target* definen dominios diferentes para cada uno de ellos, siendo ambos instancias de MOF.
Como desventaja, podemos mencionar que no hay declaración de pre ni de post condiciones.
Introduciéndonos en el cuerpo de la transformación, vemos que es posible declarar variables y metavariables, y que la separación sintáctica es provista por el lenguaje. De hecho, le aporta mucha legibilidad y claridad a la escritura de las reglas con sus estructuras FORALL MAKE y FROM TO:

- FORALL (elementos/s del source) MAKE (elemento/s del target) y
- FROM (algo de source) TO (algo del target) más la identación de las sentencias.

Tefkat también provee patrones (para el *source*) y templates (para el *target*) que se utilizan para nombrar restricciones que se pueden utilizar en más de una regla. Estas construcciones pueden a su vez parametrizarse, permitiendo la invocación a patrones y la recursión.

Las reglas se aplican en forma no determinística. Existe la aplicación condicional soportada por la cláusula IF-THEN-ELSE que permite la ejecución de reglas cuando la guarda es evaluada como verdadera. Las reglas pueden iterar y ejecutarse con recursión.

Para *traceability*, Tefkat incluye una cláusula LINKING que representa un *mapping* (asociación) entre los elementos del *source* y del *target* que participan de la transformación y que son almacenados una vez realizada la misma.

En este lenguaje es posible componer transformaciones mediante la definición de reglas, tomando transformaciones existentes y creando una nueva que preserva las relaciones.

3.4.6 UMLX

UMLX [43] es un lenguaje gráfico de transformaciones entre modelos (M2M) y que se basó en extensiones mínimas a UML. Es una propuesta de E. Willink, del GMT Consortium. En un último avance, se ha anunciado que la transformación textual que la herramienta UMLX traduce desde el gráfico, puede también editarse. Como se traduce a lenguaje OCL, sería muy conveniente para aquellos que ya conocen este lenguaje y no se cae en la necesidad de conocer otro lenguaje.

La definición de este lenguaje se basó en QVT. Asimismo, los modelos participantes de las transformaciones, deben ser instancias de MOF. Cada uno de estos es tomado como un dominio diferente (uno de entrada o *in* y otro de salida o *out*), aunque se trate de los mismos modelos. Para estos no pueden especificarse pre o post condiciones.

En cuanto a las reglas de transformación, UMLX utiliza diferentes íconos gráficos para las transformaciones, para las reglas y para las relaciones de creación, preservación y eliminación de elementos. Cabe destacar que la semántica de los íconos del lenguaje no se ha definido, y aunque es gráficamente intuido, para algunas relaciones, no queda claro cuál es su significado.

Las estructuras que define este lenguaje son bastante simples e intuitivas, pero carecen

de una semántica bien definida. No hay módulos, estructuras intermedias ni cláusulas de iteración o condición. Sólo existen variables locales, para indicar que se hace referencia a una instancia en particular. Por ejemplo, si se define una regla para todas las clases, y se define una regla también para cada atributo, al hacer referencia a una instancia de clase particular (cl, por ejemplo), se pueden mencionar a todos los atributos de cl, notándolo con @cl

La aplicación de las reglas se realiza de forma no determinística, por *pattern matching* de reglas y no se ha provisto ninguna construcción o claúsula que permita aplicar en forma condicional las reglas. Tampoco pueden invocarse otras reglas, ni se utilizan parámetros.

Ni la trazabilidad ni la composición han sido agregadas al lenguaje.

En las tablas 2 y 3 se presenta un resumen de las características enunciadas anteriormente y a modo de comparación directa entre los distintos lenguajes seleccionados.

Tabla 2. Comparación de lenguajes.

Característica/ lenguaje			ATL	MOFScript	MOLA
DEFINICIÓN	Tipo		Textual/ M2M	Textual/ M2Text	Gráfica/ M2M
	Estilo		Declarativo y operacional	Declarativo y operacional	Declarativo y operacional
	Pre y post condiciones		Si (OCL)	No	No
	MOF/QVT compatible		MOF: sí QVT: sí	MOF: sí QVT: sí	
REGLAS DE TRANSFORMACIÓN	DOMINIO	Lenguajes de dominio	Sí	Sí	Sí
		Dirección	Uni y bidireccional	Unidireccional	Unidireccional
		Relación entre origen y destino	Mismos o diferentes modelos	Diferentes modelos	Mismos o diferentes modelos
	CUERPO	Declaración de (meta)variables	Sí	Sí	Sí
		Patrones de transformaciones	No	No	Sí
		Separación sintáctica	Sí	Sí	No
		Estructuras intermedias	Sí	No	No
		Parametrización	Sí	Sí	Sí
	APLICACIÓN	Orden	No determinístic o	Determinístico	Determinístico
		Aplicación condicional	No	Sí	Sí
		Iteración de reglas	Sí	Sí	Sí
	ORGANI ZACIÓN	Modularización	Sí	No	Sí
		Mecanismos de reuso	Sí	Sí	No
TRACEABILITY			Sí	Sí	Sí
COMPOSICIÓN			Sí	No	No

Tabla 3. Comparación de lenguajes (continuación).

Característica/ lenguaje			Si Tra	Tefkat	UMLX
DEFINICIÓN	Tipo		M2M	Textual/ M2M	Gráfico/ M2M
	Estilo		Declarativo y operacional	Declarativo	Declarativo y operacional
	Pre y post condiciones		No	No	No
	MOF/QVT compatible		MOF: no QVT: no	MOF: sí QVT: sí	MOF: sí QVT: sí
REGLAS DE TRANSFORMACIÓN	DOMINIO	Lenguajes de dominio	Sí	Sí	Sí
		Dirección	Bidireccional	Unidireccional	Unidireccional
		Relación entre origen y destino	Mismos o diferentes modelos	Mismos o diferentes modelos	Mismos o diferentes modelos
	CUERPO	Declaración de (meta)variables	Si	Sí	Sí
		Patrones de transformaciones	No	Sí	No
		Separación sintáctica	Sí	Sí	No
		Estructuras intermedias	No	No	No
		Parametrización	Sí	Sí	No
	APLICACIÓN	Orden	No determinístico	No determinístico	No determinístico
		Aplicación condicional	Sí	Sí	No
		Iteración de reglas	Sí	Sí	No
	ORGANIZACION	Modularización	Sí	Sí	No
		Mecanismos de reuso	Sí	Sí	No
TRACEABILITY			Sí	Sí	No
COMPOSICIÓN			Sí	Sí	No

3.5 Resumen del Capítulo

Nuestro trabajo ha presentado una extensión a un esquema de clasificación existente que nos permite evaluar características de los lenguajes de transformación de modelos. A su vez, realizamos una investigación sobre las diversas propuestas y seleccionamos algunas para evaluarlas en base a la clasificación antes mencionada.
De este análisis se desprende que tanto ATL como Tefkat son lenguajes muy completos y en avanzado desarrollo, siendo ATL más reconocido y utilizado (Las investigaciones nos permitieron identificar asimismo que es un lenguaje de referencia en el contexto MDD). Ambos adhieren a los estándares de la OMG, utilizan un lenguaje formal como es OCL y proveen mecanismos para *traceability* y composición de las reglas de transformación. ATL hace uso además de estructuras intermedias que mejoran la legibilidad y el reuso.
Todas estas características lo hacen un lenguaje elegible a la hora de escribir las transformaciones pensadas para la presente propuesta.
Características similares, con respecto además a la compatibilidad con estándares y a la

posibilidad de generaciones modelo a texto –otro requerimiento de nuestra propuesta- hacen elegible a MOFscript como lenguaje de transformación de modelos.

Capítulo 4- Trazabilidad en MDD/ MBT

Este capítulo presenta el concepto de trazabilidad, destacando su importancia dentro del proceso de desarrollo de software. Luego se lo trata dentro de los contextos de interés del presente trabajo de investigación. Así, en la segunda sección analizamos diferentes propuestas de trazabilidad dentro de MDD; y en la tercera sección realizamos un análisis similar en el contexto MBT sobre propuestas actuales de *Traceability*.

4.1 Introducción

En la Ingeniería de Software, se ha definido trazabilidad como la capacidad de establecer relaciones entre dos o más artefactos de un sistema software dentro de un proceso de desarrollo, especialmente aquellos artefactos con vínculos 'predecesor-sucesor' o 'maestro-subordinado' entre sí; por ejemplo, entre los requerimientos y los elementos de diseño que los modelen [1].

Utilizar técnicas de trazabilidad en la producción del software es considerado como una medida de madurez del proceso de desarrollo por varios estándares, algunos de ellos derivados de la metodología 'en cascada' donde era necesario "mostrar" que el producto final se correspondía con los requerimientos. Fue necesario entonces establecer algún mecanismo que 'ligara' distintos artefactos de los sistemas. Ejemplo de ello es la norma para especificación de requerimientos de software de la IEEE [44].
Luego, CMMI también reconoce esta práctica como ítem de calidad al indicarlo como requisito necesario para alcanzar su nivel 2 [45].
Por lo tanto, podríamos decir que adherir a las técnicas de trazabilidad en los proyectos de desarrollo de sistemas informáticos es indicativo de una mejor calidad del proceso de desarrollo y del software producido.

Nuestro enfoque propone adicionar trazabilidad al proceso de generación de casos de prueba, tema central de esta tesina, comprendiendo que reporta varios beneficios al proceso de desarrollo de sistemas. A modo de ejemplo, podemos mencionar que en la fase de *testing* en particular, brinda la posibilidad de identificar los casos de prueba que son necesarios ejecutar para testear una determinada funcionalidad y permite contar con una medida de la cobertura de los casos de prueba definidos.

4.2 Trazabilidad en MDD

La propuesta MDD –*Model Driven Development*- [2], [3], en cuanto al aporte metodológico y de proceso para la trazabilidad de un sistema, provee la posibilidad de automatizar la creación y la administración de links -o trazas- mediante la definición y ejecución de transformaciones de modelos.
Es deseable, en este contexto, que puedan utilizarse los beneficios aportados por este paradigma en cuanto a las facilidades de los lenguajes de transformación que adhieren nociones de trazabilidad a su sintaxis.

Realizando un análisis de diferentes propuestas, vemos que existen en la literatura un conjunto de enfoques que definen modelos, metamodelos y frameworks para

trazabilidad.

Como introducción general podemos mencionar que existen varios tipos de propuestas para la generación de links (trazas) entre elementos de modelado. En general, existen dos grandes grupos de propuestas:

- Trazabilidad automática o trazas impuestas
- Inferencia de trazas o trazas inferidas

En primer término, adicionar a la transformación entre modelos otros elementos –el código necesario- para que se generen además del modelo destino, las trazas entre los elementos involucrados en la transformación. Lo que realmente se genera es un modelo adicional al modelo destino: un modelo de trazas. Podríamos decir que el modelo de trazas se genera de manera conjunta con el modelo destino.

Otra posibilidad –segundo grupo- es la de generar las relaciones (el modelo de trazas) a partir de elementos ya existentes. La transformación, en este caso, genera como nuevos elementos a los links a partir de la verificación de la existencia de un artefacto en el modelo origen que se corresponde con un artefacto –o más- en el modelo destino. Podríamos decir que el modelo de trazas se genera post generación del modelo destino.

A continuación hacemos una breve referencia de algunos de los trabajos más relevantes o representativos de diferentes tipos de aporte.

El lenguaje de modelado UML [7], por medio de la definición de estereotipos –uno de sus mecanismos de extensión como lo especificamos en el capítulo 2, sección 2.3-, permite representar relaciones como trazas entre elementos de modelado. En este caso, se utiliza el estereotipo «trace» sobre una relación de dependencia entre los elementos de modelado.

Entre las propuestas que definen metamodelos, podemos mencionar los siguientes trabajos. Amar, Leblanc y Coulette en [46] presentan un metamodelo que permite definir diferentes tipos de links entre los elementos de modelado que relacionan. Lo particular de esta propuesta es que a nivel de metamodelo se contempla la posibilidad de que los links puedan anidarse –muchos no lo contemplan-.
La utilidad que le encontramos es la de poder definir un tipo de traza de mayor nivel o jerarquía que contiene a trazas más 'simples': una traza compuesta de trazas simples o bien de trazas que contienen a otras.
Esta propuesta es ideal para la representación de trazas en la composición de transformaciones. Este metamodelo se presenta en la figura a continuación.

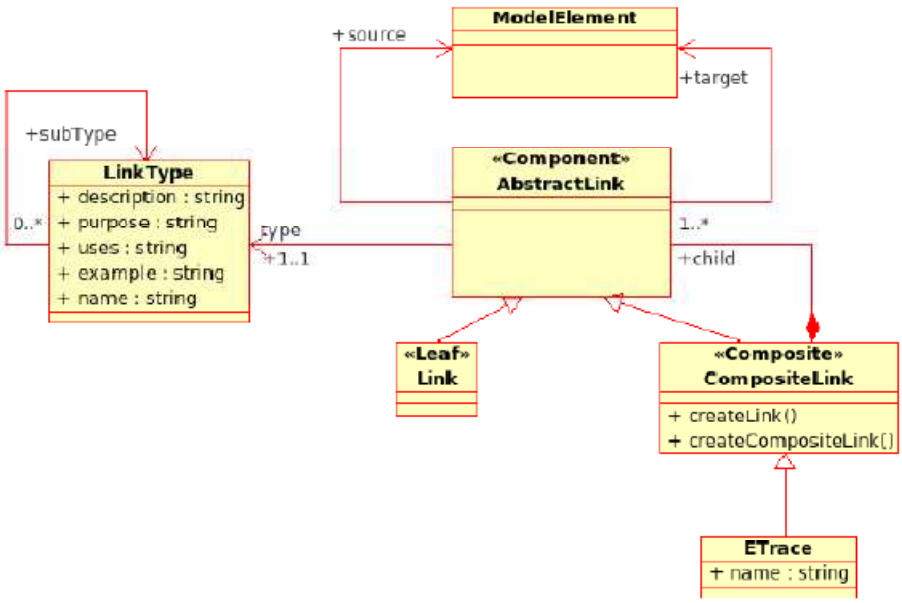


Figura 4.1. Metamodelo para trazabilidad propuesto por Amar, Leblanc y Coulette

La propuesta de [47] resulta ser amplia y abarcativa. Al igual que el trabajo anterior, contempla la creación de diferentes tipos de trazas, pero en este caso, son tipos definidos para todo el ciclo de vida del software (desde los requerimientos al producto final) y no sólo para las primeras etapas (análisis, diseño).

Otro trabajo en este sentido es [48], donde los autores presentan links tipados y semánticamente ricos. Esto se debe a que las trazas se definen para múltiples metamodelos, cada uno para relacionar tipos determinados de elementos, y que permiten adicionar restricciones en el lenguaje EVL (*Epsilon Validation Language*).

En [49] en cambio, se presenta un trabajo que se encuadra dentro del grupo de las inferencias de las trazas. Los autores presentan una técnica que infiere de forma automática la información de trazas a partir del código de una transformación en QVT [18]. La técnica se basa en el análisis de la especificación de la transformación, en particular de las variables auxiliares, que le permiten inferir relaciones entre elementos origen y destino.

En cuanto a los lenguajes de transformaciones de modelos propiamente dichos, analizamos el soporte que éstos brindan para generar trazas. Entre ellos, podemos destacar a dos lenguajes de transformaciones: ATL (ATLAS Transformation Language) [9], [26] y Kermeta [50].

El trabajo presentado por Jouault [26] sobre trazabilidad en ATL es ya un trabajo 'de referencia' en la literatura de MDD (y trazabilidad en MDD por las características definidas para tal fin). En el mismo, se muestra cómo agregar información adicional a las transformaciones codificados en ATL, de manera débilmente acoplada, que permite generar el modelo de trazas sin alterar la definición de las transformaciones de modelos.
Fue pensado como una primera extensión al lenguaje. Provee separación sintáctica dentro de la transformación, lo cual lo hace fácilmente identificable.

En cuanto al lenguaje Kermeta, en [50] los autores toman la propuesta presentada en [26] para ATL y desarrollan un framework para Kermeta con características similares en

cuanto a la generación de trazas con el código de la transformación. Agregan además nuevos conceptos o tipos de traza que permiten almacenar la traza (*trace*) y la traza con los artefactos que relaciona (*static trace*).

No podemos dejar de mencionar, aunque sea brevemente, al lenguaje estándar (CORE) QVT –*Query/View/Transformation*- [18], el estándar de transformación de modelos adoptado por la OMG. Este define la metaclase *Trace*, una clase MOF con propiedades que referencian a los artefactos de los modelos que están relacionados por una transformación.

Nuestra propuesta, a diferencia de algunos de estos trabajos que resultan más genéricos por el contexto MDD, propone un metamodelo de trazas con tipos de links definidos para artefactos de *testing*. Además, utiliza transformaciones de modelo y genera a partir de ellas el modelo de trazas. Este es un modelo separado de los modelos de origen y destino que forman parte de las transformaciones entre modelos. Todas estas nociones sobre nuestra propuesta se explicarán en los siguientes capítulos.

4.3 Trazabilidad en MBT

Model Based Testing (MBT) [19], [20], [21], [22] es una propuesta reciente que desafía la problemática de generar los modelos y artefactos necesarios para el testeo de software ya que los modelos de *testing* son derivados total o parcialmente desde modelos que describen la funcionalidad del sistema en desarrollo. Este último, visto desde la perspectiva del *testing* se conoce como SUT (*System Under Test*- Sistema bajo Testeo).
Según este paradigma, el SUT puede ser un único artefacto -como una clase por ejemplo-, puede ser también un método o puede ser tan complejo como un sistema software completo. Para el *testing*, los modelos del sistema en desarrollo proveen una descripción de la funcionalidad del mismo. A partir de estas descripciones/ comportamiento especificado en los modelos del SUT, es posible generar un conjunto de casos de prueba que se pueden utilizar para determinar si el sistema se ajusta a ciertos requerimientos o propiedades deseables para el sistema -representado en los modelos-.
MBT define una forma de prueba de caja negra –o *testing* funcional- que utiliza modelos estructurales y de comportamiento- para automatizar la generación de casos de prueba.

MBT se centra en la modelización de los sistemas, donde un modelo es la descripción del comportamiento de un sistema que ayuda a entender el funcionamiento del mismo.
La idea general de este paradigma presenta lo siguiente: "Si a partir de un mismo modelo se generan los *tests* para verificar su comportamiento, cuando el comportamiento del sistema cambie, también lo harán los tests" [19].

Distintos trabajos se han estudiado en el contexto MBT, mencionando a continuación aquellas que resultaron más representativas de las soluciones propuestas.

MATERA [51] (y su trabajo relacionado [52]) es un framework que integra el modelado del sistema con la definición de trazas dentro de un proceso MBT.
Los requerimientos, modelados en SysML [53], pueden ser relacionados con otros artefactos que modelan el sistema; con modelos o bien con elementos de los modelos. Cuando los requerimientos se usan para la generación de *tests* –abstractos, no ejecutables– se asocian con un *tag* al caso de prueba generado. Para 'tracear' los

requerimientos a partir de los casos de prueba, se utiliza un script *Phyton* que analiza el *log* del *test* y genera un *query* en OCL que identifica al requerimiento involucrado en el test.

En segundo lugar, en [54], la técnica de trazabilidad definida por Pasupulati propone un tipo de modelo para los requerimientos y la definición de trazas como anotaciones que identifican los elementos que esta relaciona, siendo el objetivo del autor simplificar el proceso de gestión de trazas.
Los requerimientos se especifican en un diagrama de clases semi-formal que incluye dos tipos de elementos: uno es el requerimiento representado como un identificador y el otro, un grupo de clases que representa a los actores del sistema. Estas clases contienen toda la funcionalidad del sistema especificadas como operaciones del diagrama de clases. Las trazas se definen como anotaciones con *tags* de identificación, técnica elegida de entre las muchas que la trazabilidad en requerimientos propone y como hemos mencionado al inicio del presente capítulo.

La forma en que se gestiona la trazabilidad es manteniendo una relación (que contiene el *tag* identificatorio del requisito) entre el modelo de requerimientos mencionado y el diagrama de estados que se modela como representación del comportamiento del sistema.

Por último, un trabajo más actual en el tiempo [55], propone el uso de tecnologías XML para la definición de trazabilidad en MBT. En este caso, la traza es una relación elemento a elemento definida con una estructura XML formal: RDML (*Relation Definition Markup Language*). Para cada tipo de modelo utilizado, una definición RDML -basado en un esquema XML- especifica la relación con otro modelo y su tipo, siendo el tipo "derivado" o bien, "referenciado".
Si bien esta idea permite una rápida definición de la traza y cuenta con la facilidad de estar basada en un esquema XML, carece de especificación del tipo de traza.

4.4 Resumen del Capítulo

A modo de cierre de este capítulo podemos hacer las siguientes menciones:

Nuestra propuesta tiene similitudes y diferencias con los trabajos citados en varios aspectos.
Para el modelado de requerimientos, elegimos Casos de uso ya que tienen la ventaja de pertenecer a un estándar (UML), tener buen soporte de herramientas y amplia difusión entre los analistas.

Por otra parte, nuestro enfoque propone un metamodelo de trazas no genérico, sino específico para artefactos de *testing*. Como ventaja, podemos decir que los tipos de trazas que definimos son específicos del contexto que interesa dentro del proceso MBT. Además, contar con un modelo de trazas favorece la realización del análisis y validaciones sintácticas automáticas que pueden realizar distintas herramientas.

Otra diferencia es que nuestra propuesta genera el modelo de trazas a partir de las transformaciones de modelo; separado de los modelos de origen y destino, otorgándole a la traza identidad como elemento de modelado y evitando de esta forma la 'polución' de los modelos origen y/o destino. Asegura además, la creación de cada link relacionando elementos de los modelos origen y destino.

Con esta revisión sobre el estado del arte de Trazabilidad en los contextos que nos

interesan, MDD/MBT y los lenguajes de transformaciones de modelos estamos en condiciones de plantear nuestra propuesta.

Capítulo 5- Propuesta: Proceso de Generación de Casos de Prueba

En este capítulo se presenta el proceso que permite generar casos de prueba de forma automática mediante transformaciones de modelo, partiendo de la especificación de Casos de Uso y adicionando la generación de trazas entre los elementos de los distintos modelos.
Para su mejor comprensión, se describe el esquema general o especificación del proceso en primera instancia, para hacer luego una descripción detallada del mismo.

5.1 Esquema General del Proceso

Con el objetivo de brindar soporte al *testing* de sistemas, este trabajo presenta un proceso de generación de casos de prueba.
Para mayor claridad a la hora de especificar el proceso en sí, definimos su esquema general que se compone de dos grandes partes (proceso y subproceso).

El proceso define las siguientes etapas:

1. Definir el Modelo de Casos de Uso (MCU).
2. Aplicar una transformación a partir del MCU para obtener un DATS (Diagrama de Actividades de *Test* del Sistema)
1. Aplicar otra transformación desde el DATS obteniendo todas las posibles secuencias de ejecución como integración de las distintas funcionalidades.

Por otro lado, el proceso incluye este sub-proceso:

1.1. Por cada CU y a partir de su documentación, se aplica una transformación para obtener un DAT (Diagrama de Actividades de *Testing*).
1.2. Por cada DAT, se ejecuta una nueva transformación modelo a texto: desde el DAT al Caso de Prueba.

Cada transformación genera además las trazas entre los elementos de modelado participantes en la misma (origen y destino).

El esquema general del proceso se muestra en la Figura 5.1.

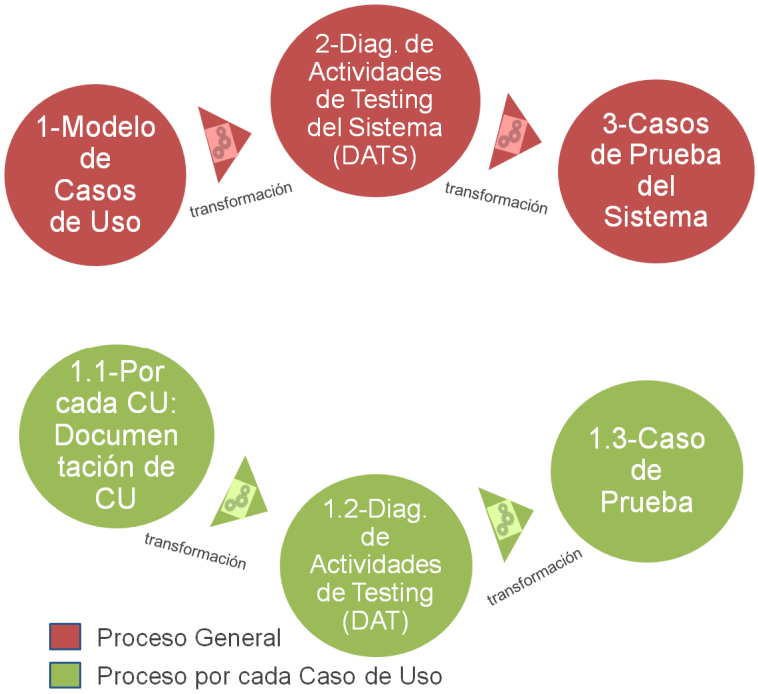


Figura 5.1. Esquema: Proceso general y subproceso

En una explicación más detallada, podemos decir que como parte del "proceso general" (color bordó de la Figura 5.1) se derivan "casos de prueba del sistema" -a los que llamaremos también *tests* de integración-. Estos testean conjuntos de funciones las cuales no son más que (posibles) ejecuciones de los usuarios-. Se generan automatizando la derivación de los mismos a partir de un modelo inicial de Casos de uso, por medio de transformaciones de modelo. Este proceso fue presentado como trabajo de investigación en [56]

Este proceso, más general y abarcativo, se completa con otro proceso para especificar casos de prueba a partir de la documentación de cada caso de uso. A esos casos de prueba los llamamos *tests* de unidad y son estos los que necesitan ser ejecutados primero y probados para luego poder probar los *tests* de integración.
Este subproceso se identifica en la Figura 5.1 como parte del "proceso por cada caso de uso" (color verde). Este subproceso fue presentado como trabajo de investigación en [57].

En este punto, vale mencionar que la propuesta actual es la unión de al menos tres (3) trabajos de investigación -publicaciones- anteriores, ya que al proceso (definido y completado en [56] y [57]) se le agregó la posibilidad de generar las trazas entre los elementos [58], permitiendo completar la definición de un proceso general para la generación de casos de prueba del sistema en general y de cada funcionalidad en particular, a partir del modelo de casos de uso incluyendo trazabilidad.

A continuación se describe el proceso según las etapas presentadas al inicio de esta sección.

5.2 Descripción del Proceso en etapas

En esta sección, se completa y describe con mayor detalle el proceso presentado en esta tesina.

5.2.1 PROCESO GENERAL

Como se mencionó en la sección anterior, el proceso general abarca una serie de etapas que se mencionan y describen a continuación.

ETAPA 1: Definición del Modelo de Casos de Uso -Modelo Independiente de la Plataforma (PIM) en MDD-

Entre los lenguajes de modelado que pueden ser utilizados para automatizar la generación de modelos de *testing* se encuentra UML [7], estándar oficial de la OMG (Object Management Group) [13], maduro, adoptado y utilizado por la comunidad de las ciencias de la computación, además de contar con numerosas herramientas CASE que son utilizadas por los equipos de desarrollo.

UML provee un tipo de diagrama que permite modelar los requerimientos funcionales del sistema: se trata del diagrama de Casos de Uso (Ver Anexo I).
Un modelo UML de Casos (MCU) de uso representa la funcionalidad global de un sistema dado, indicando además: qué roles inician a esos casos de uso, si hay funcionalidades que se ejecutan como parte de otras o cuáles casos de uso son extensiones de alguna funcionalidad dada (Ver Figura 5.2). Cada caso de uso aporta información sobre el estado del sistema antes y después de su ejecución, por lo que resulta un artefacto muy útil para el *testing* de los sistemas.

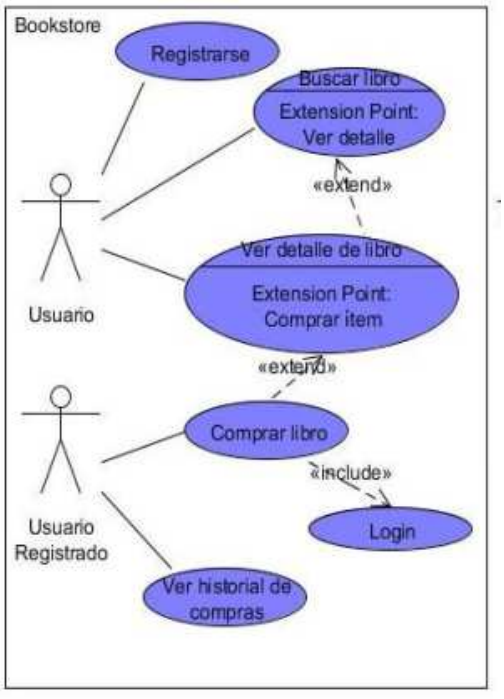


Figura 5.2. Etapa 1: Definición del Modelo de Casos de Uso (DCU)

Nuestra propuesta se basa en la generación automática de casos de prueba a partir del modelo de casos de uso de UML contando con la ventaja de la definición temprana de casos de prueba en la etapa de análisis e incentivando el comienzo del proceso de *testing* en el momento en que un requerimiento funcional se encuentre identificado y documentado; evitando de esta manera posponer la actividad de testeo para etapas más adelantadas en el desarrollo del sistema cuando corregir errores –o descubrirlos- implica importantes pérdidas de tiempo y esfuerzo.

ETAPA 2: Definir y aplicar una transformación a partir del MCU para obtener un DATS

Se define y aplica en esta etapa una transformación del MCU para obtener un diagrama de actividades del sistema particular para *testing* (DATS)- otro modelo (PIM) en MDD-, construido utilizando un Perfil UML definido para modelar actividades de *testing* del sistema. Ver Figura 5.3.
Como hemos mencionado, un modelo de casos de uso representa la funcionalidad global del sistema y tiene un alto nivel de abstracción. La funcionalidad modelada del sistema no presenta un orden secuencial o cronológico sino que es meramente descriptivo.

Para la definición de *tests* de integración, interesa saber cuáles funcionalidades se integran unas con otras. En otras palabras, se trata de ver qué acciones realizará en forma conjunta un usuario del sistema (llamémoslos 'caminos de ejecución'). Esa información no se encuentra definida en el MCU - si bien hay funcionalidades que se integran en forma explícita en algunos casos e implícita en otros (relaciones <<include>> y <<extend>> de Casos de Uso). (Ver Anexo II)

Para obtener los posibles caminos de ejecución mencionados, se propone la generación de un 'Diagrama de Actividades del Sistema' que especifique distintas ejecuciones funcionales. De esta manera, se pueden 'ordenar' las funcionalidades del sistema, secuenciar, alternar, etc.
En un 'Diagrama de Actividades del Sistema', una actividad representa una funcionalidad del sistema y se condice con un Caso de Uso del Diagrama de Casos de Uso del Sistema.
Al representar la ejecución de las funcionalidades en un diagrama de actividades, tenemos muchos elementos de modelado provistos por este tipo de diagrama y que pueden destacarse como ventajas dado que enriquecen el modelo y le otorgan mayor expresividad. Por ejemplo, de las actividades podemos utilizar sus parámetros de entrada, sus guardas y acciones, los estereotipos ya definidos <<precondition>> y <<postcondition>>, además de la representación de objetos como ingreso a una actividad o como salida-resultado producida por un comportamiento determinado.

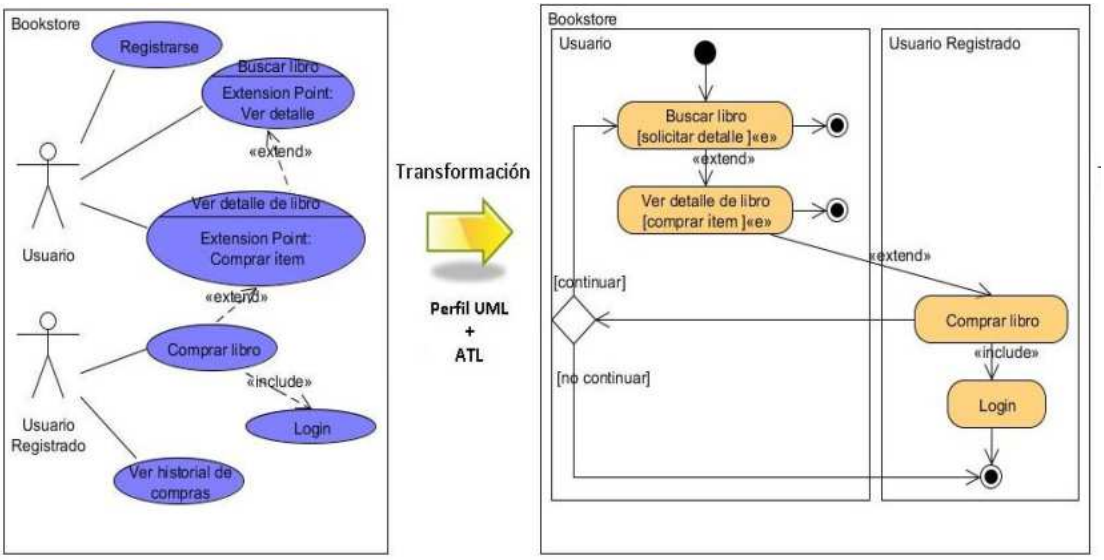


Figura 5.3. Etapa 2: Transformación DCU a DATS

Por medio de una transformación, se toma como modelo origen a un MCU y se genera un Diagrama de Actividades del Sistema.

Ahora bien, como el Diagrama de Actividades del Sistema no refleja en forma precisa las relaciones entre Casos de Uso, se ve la necesidad de definir un perfil UML que agregue esta información al modelo destino. A este diagrama generado lo denominaremos "Diagrama de Actividades de *Test* del Sistema" o simplemente DATS.
En el siguiente capítulo se define el perfil UML mencionado con sus reglas de buena formación en OCL.Una vez obtenido el DATS –al que el analista en este punto puede modificar-, el paso siguiente es generar los *tests* de integración de las funcionalidades descriptas en el Modelo de Casos de Uso.

ETAPA 3: Definir y aplicar una transformación desde el DATS, para obtener todas las posibles secuencias de ejecución como integración de las distintas funcionalidades

Finalmente y a partir del *DATS* generado en la etapa 2, se aplica otra transformación que produce un archivo XML con todas las posibles secuencias de ejecución que pueden realizarse en el sistema y que integran las distintas funcionalidades, como muestra la Figura 5.4.
El DATS generado le permite al ingeniero de software tener una visión global de la secuencia de acciones que un usuario puede realizar en el sistema.

Si este no estuviese completo, en el sentido en que algunas actividades no transiten hacia otras, se podría en este punto relacionarlos a través de transiciones en el *DATS*. Estas son las "relaciones implícitas" entre las diferentes funcionalidades. Es decir, hay transiciones entre actividades no definidas explícitamente en los casos de uso, pero que por lógica de ejecución pueden ser requeridas como dependencias entre funcionalidades (por ejemplo, un usuario para poder comprar debe loguearse, para poder loguearse debe registrarse en el sistema). Eso será útil también para la conformación de los casos de prueba por la completitud de información otorgada al modelo DATS.

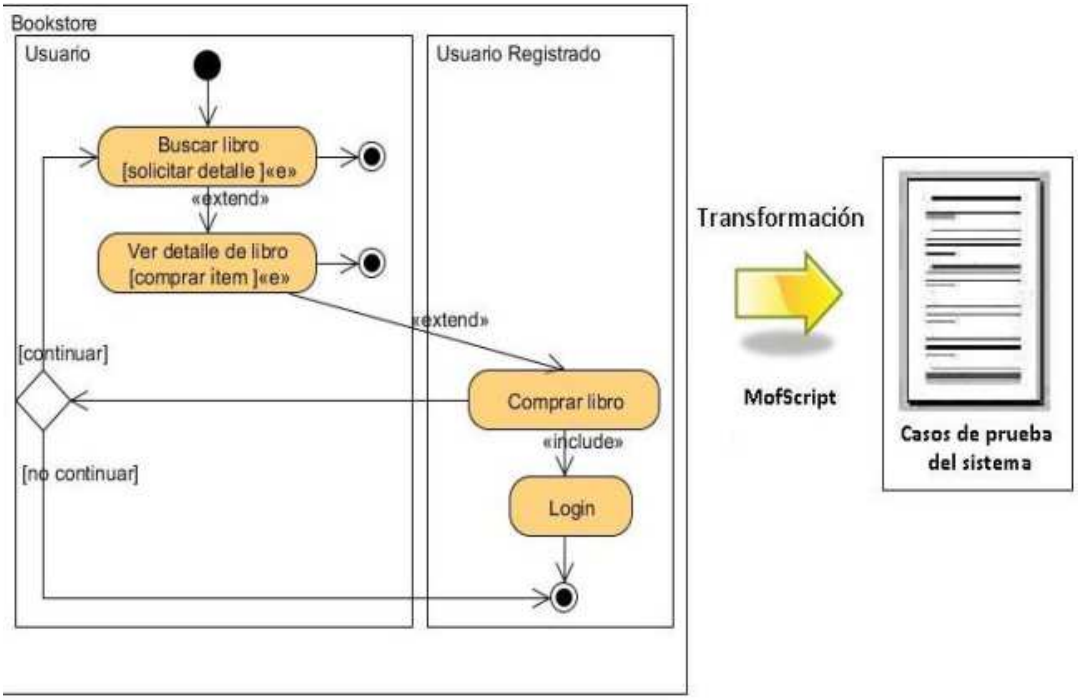


Figura 5.4. Etapa 3: Transformación DATS a CPs

Para encontrar todos los caminos de ejecución entre las funcionalidades, se propone la visión del Diagrama de Actividades como un grafo dirigido. La Figura 5.5, muestra al *DATS* como un grafo dirigido conexo del que pueden extraerse con un algoritmo DFS (*Deep First Search*), los caminos de ejecución desde una actividad raíz –inicial- hasta las actividades hojas. Permite encontrar todos los caminos de ejecución entre las funcionalidades, o los caminos de ejecución de una longitud determinada, pudiendo contemplar asimismo la aparición de ciclos. Esta idea se menciona en el trabajo de Briand y Labiche [59] y de allí la adaptamos.

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

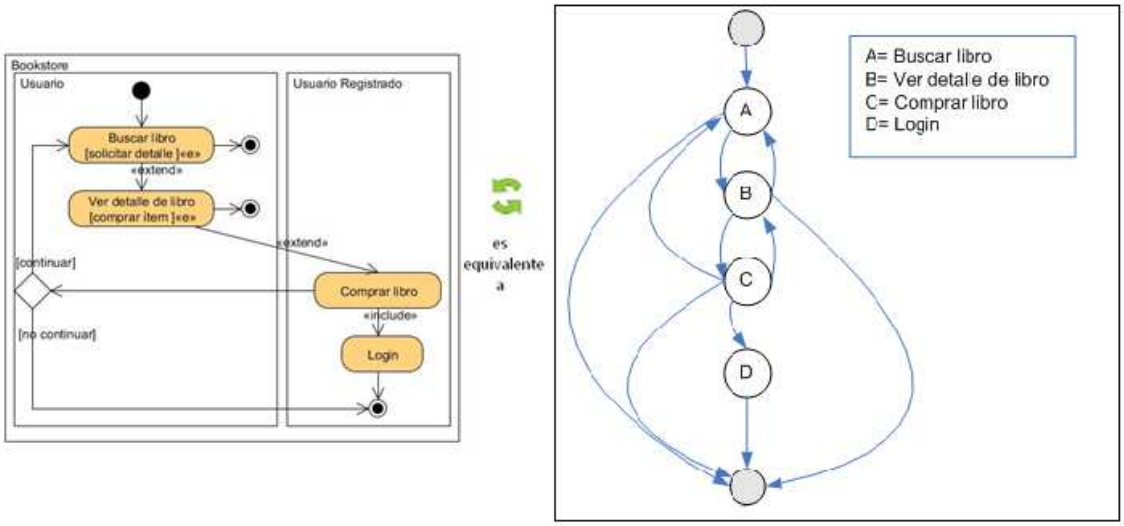


Figura 5.5. DATS visto como un grafo dirigido

Para la definición de la transformación del modelo DATS al texto de los casos de prueba se eligió MOFscript [10]. MOFscript es un lenguaje de transformaciones que cuenta con la implementación de una herramienta -del mismo nombre-, que permite definir y ejecutar transformaciones modelo a texto basadas en el estándar QVT (Query/ View/ Transformation) [18]. Utilizamos este lenguaje por adecuarse a nuestro requerimiento de obtener un texto como salida y por estar actualmente adoptado masivamente por la comunidad MDD.

La Figura 5.6 muestra en forma integrada esta primera parte del proceso con las etapas mencionadas anteriormente.

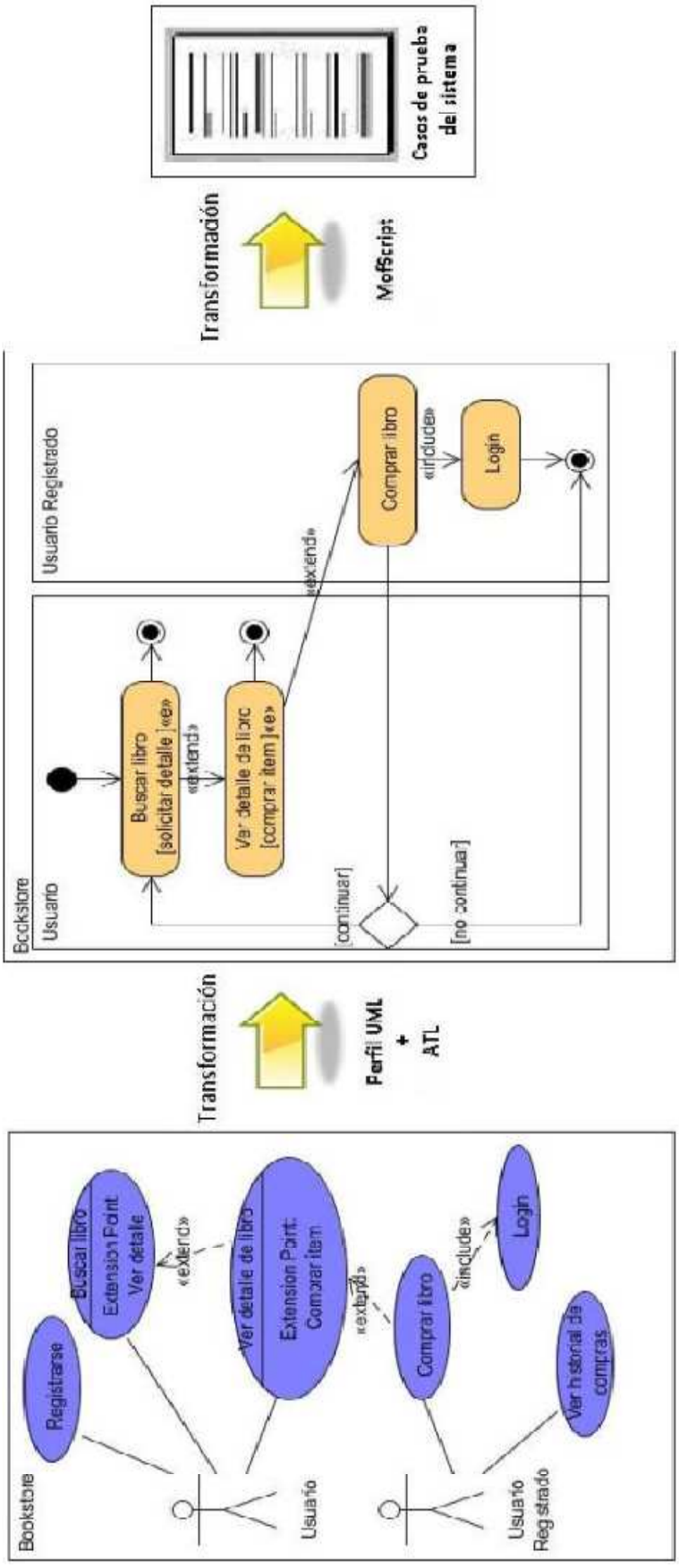


Figura 5.6. Vista unificada del proceso general

Para completar la explicación del proceso en sí, podemos indicar que la "etapa 1" es la más artesanal. Se trata de un modelo o Diagrama de Casos de Uso (DCU) que contiene la información sobre la funcionalidad relevante del sistema. Es por ello que se constituye en nuestro punto de partida, siendo además uno de los modelos que se especifica en las tempranas etapas del análisis dentro del ciclo de vida de un sistema.

Por medio de la definición de una transformación que toma como modelo de entrada al DCU, se genera un *DATS* enriquecido con el perfil UML al que se hace mención en el "etapa 2".
En el *DATS*, los casos de uso fueron transformados a actividades manteniendo semánticamente las relaciones existentes entre los mismos (inclusión, extensión y generalización).
Se obtiene de esta manera y como resultado de la transformación definida, un modelo de actividades de *testing* del sistema como puede observarse en la parte media de la Figura 5.6.

El DATS le permite al ingeniero de software tener una visión de ejecución de funcionalidad global del sistema. También le permite agregar o modificar estas secuencias de ejecución antes de generar los *tests* de integración o pruebas del sistema.

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

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

Finalmente, para cumplimentar la etapa 3, se define una **transformación** modelo a texto (M2T) a partir del *DATS* obtenido en el paso anterior que genere un archivo XML con los posibles caminos de ejecución que un usuario pueda realizar sobre el sistema y que integran, en distintos casos de prueba, las distintas funcionalidades del sistema.
Estos elementos se presentan en el capítulo 6: "Implementación del proceso".

5.2.2 SUBPROCESO

A continuación se describe, ahora con mayor detalle, el subproceso. Recordando que nuestro propósito es definir un proceso que genere casos de prueba a partir de casos de uso, Es decir, el proceso general se centra más en el proceso de desarrollo que en los *tests* en sí mismos.

Este subproceso se basa en una serie de pasos que se detallan a continuación:

ETAPA 1 Y 1.1: Definición del Modelo de Casos de Uso -Modelo Independiente de la Plataforma (PIM) en MDD-

En primer lugar, se debe definir el Modelo de casos de uso -Modelo Independiente de la Plataforma (PIM) en MDD- y la documentación textual de cada uno. (Ver Figura 5.7).

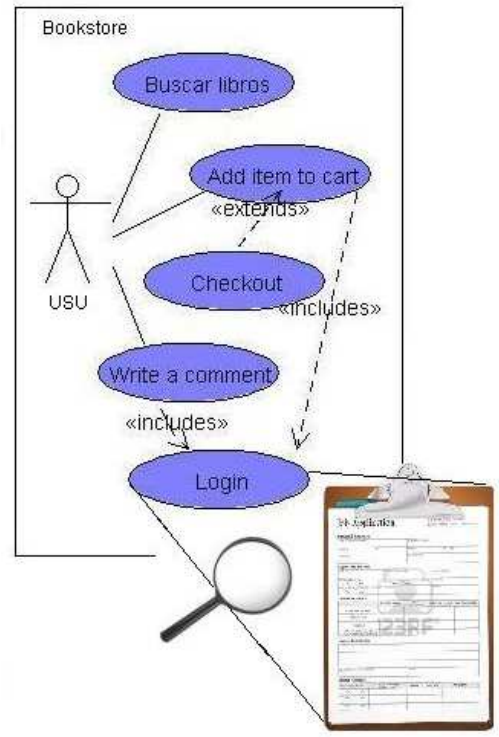


Figura 5.7. Etapas 1 y 1.1: Definición del Modelo de Casos de Uso con su documentación (MCU)

Las razones para la elección del modelo de Casos de Uso fueron explicadas en la 'Etapa 1' del proceso general, con lo cual mencionarlo aquí sería redundante.

ETAPA 1.2: Definir y aplicar una transformación a partir del MCU para obtener un DAT

A partir de los casos de uso se pueden elaborar casos de prueba para el *testing* funcional de un sistema.
Cada caso de uso describe varios escenarios de uso entre los que se distinguen:

- Los escenarios correspondientes al curso normal del caso de uso.
- Los escenarios que incluyen al menos un curso opcional del caso de uso, ya sean alternativas o extensiones.
- Los escenarios que producen al menos una excepción.

A partir de cada Caso de Uso, se aplica una transformación del mismo a un Diagrama de Actividades particular para *testing* (DAT)-Modelo Específico para *Testing* (PIM) en MDD-, construido a través de un Perfil UML definido para modelar actividades de *Testing*. (Ver Figura 5.8).
A partir de la documentación de cada caso de uso, se genera un diagrama de actividades de *testing* intermedio que permite especificar detalladamente qué tarea realiza cada 'actividad de *testing*' generada a partir del caso de uso.

En este paso se realiza una transformación de la documentación de los casos de uso a diagramas de actividades de *testing*. Este diagrama especifica las acciones a realizar para verificar la funcionalidad definida. Se utiliza la facilidad de definición de gramáticas de JavaCC [60] y su consecuente generación de código Java. JavaCC tomará como origen un texto a parsear –la documentación del caso de uso- y tendrá como salida, un archivo .java cuya ejecución crea el diagrama de actividades de *testing* al que se hace mención.
Se usan para ello pre y post-condiciones y los pasos de los flujos del caso de uso. Estas condiciones son necesarias para establecer el estado del sistema antes y después de la ejecución de un caso de uso. Los flujos establecen comportamiento que completa la funcionalidad del diagrama de actividades de *testing*.

Si bien esta transformación que se menciona se explica en el capítulo siguiente (Capítulo 6: "Implementación de la Propuesta"), podemos mencionar que de la documentación de casos de uso, en particular del flujo normal, se conformará el conjunto de las actividades a ejecutar: una actividad por cada paso del caso de uso.

El tipo de la actividad quedará determinado por el paso en sí. Puede tratarse de:

- la ejecución de una operación (actividades <<operation>> y <<query>>). Por ejemplo, 'comprar un libro' y 'consultar saldo' respectivamente.
- la verificación de condiciones y estados (actividad <<verify>>). Por ejemplo, 'verificar tipo de socio igual a Vitalicio'.
- la creación de objetos (actividad <<creation>>) como 'crear una compra'.
- la validación de datos ingresados (actividad <<inputData>>). Por ejemplo, 'usuario no es igual a blanco'.
- la llamada de ejecución a otros casos de uso (actividad <<compound>>). Representa al <<include>> entre casos de uso.

Las pre y post condiciones determinan actividades <<verify>>.

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

Una vez obtenido el Diagrama de Actividades intermedio, el paso siguiente es generar los casos de prueba de cada funcionalidad descripta en el DCU.
A esos casos de prueba los llamamos (en general) *tests* de unidad y son estos los que primero necesitan ser ejecutados y probados para luego poder probar los *tests* de integración.

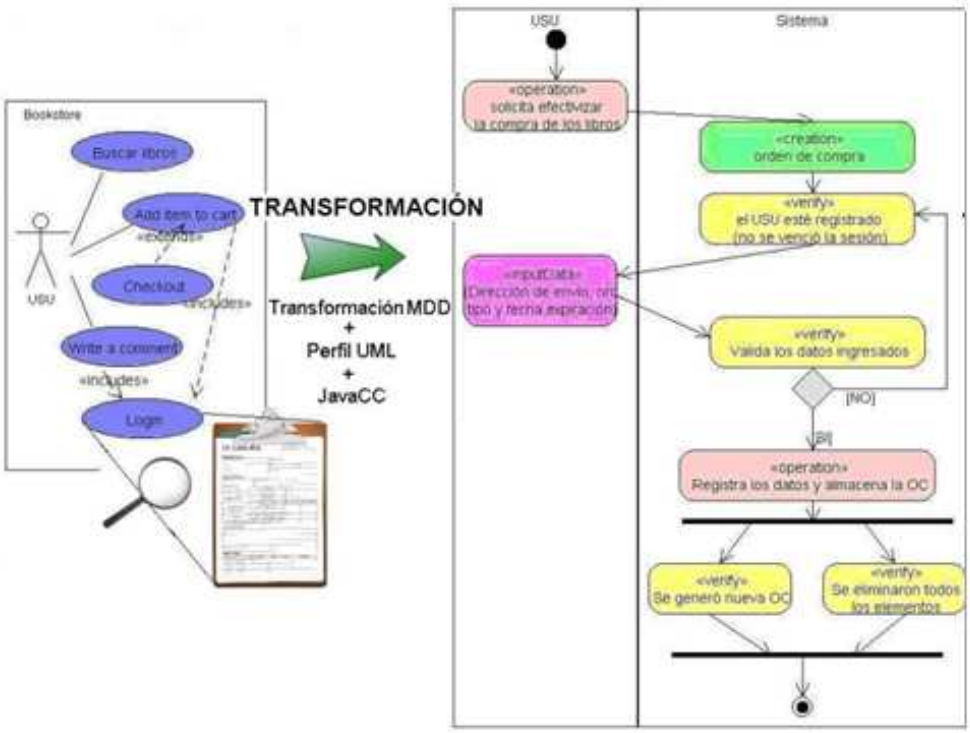


Figura 5.8. Paso 1.2: Transformación de DCU a DAT

ETAPA 1.3: Definir y aplicar una transformación desde el DAT, para obtener casos de prueba de la funcionalidad y datos de prueba para la misma

Se ejecuta luego una transformación modelo a texto –PSM a PSM- que tiene como entrada al Diagrama de Actividades construido con el perfil de *testing* mencionado y como salida al caso de prueba (CP) y también las pruebas de datos (DP). Es decir, ejecutar la transformación genera como resultado (esquemas de) casos de prueba. (Ver Figura 5.9)

A esta altura del proceso contamos con un Diagrama de Actividades de *testing* que especifica una serie de actividades -de *test*- a realizarse el comportamiento del *test*. El último paso es el que nos permite obtener de este diagrama, la definición del caso de prueba. Para ello, hemos definido una transformación en MOFscript [10] que toma como modelo de entrada el diagrama mencionado y genera un modelo en texto: el caso de prueba. MOFscript es un lenguaje implementado con una herramienta del mismo nombre, que permite definir y ejecutar transformaciones modelo a texto basadas en el estándar QVT (*Query/View/Transformation*) [18]. Utilizamos este lenguaje por adecuarse a nuestro requerimiento de obtener un texto como salida y por estar actualmente adoptado por la comunidad MDD como lenguaje de transformaciones modelo a texto.

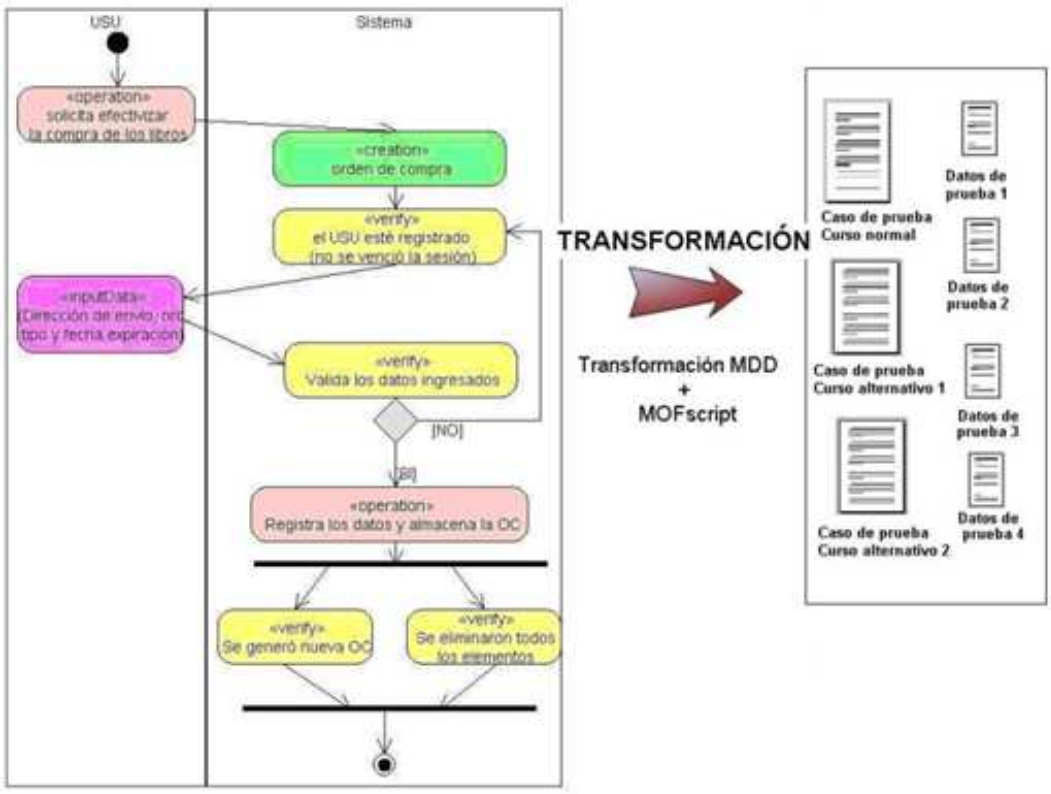


Figura 5.9. Etapa 1.3: Transformación DAT a CP y DP

La Figura 5.10 a continuación muestra en forma integrada esta segunda parte del proceso (el subproceso) con las etapas mencionadas anteriormente.

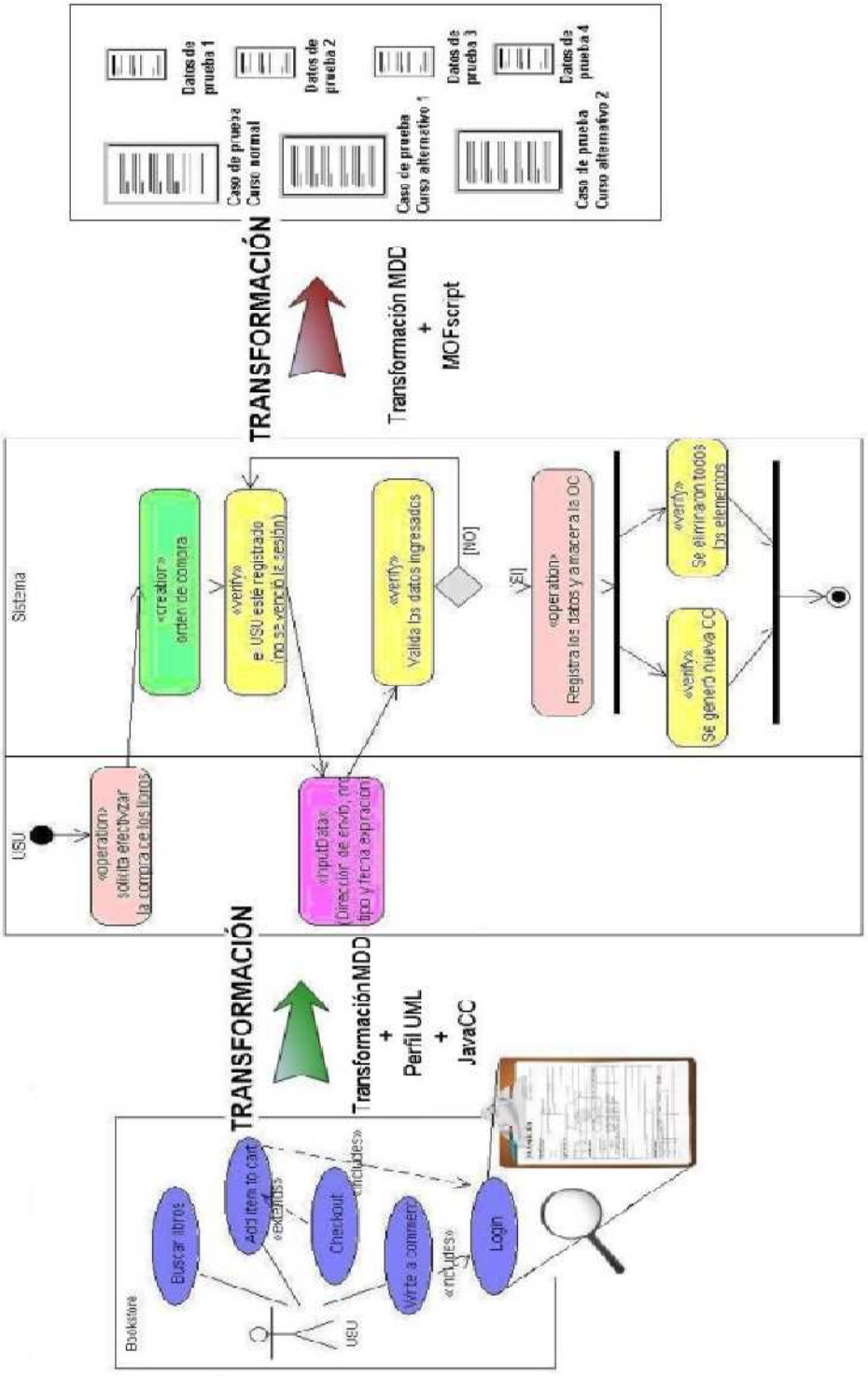


Figura 5.10. Subproceso de generación de Casos de Prueba completo

Para implementar el subproceso de generación automática de casos de prueba a partir de cada caso de uso es necesario contar con elementos de modelado específicos que permitan concretar la primera transformación mencionada en esta sección en la etapa 1.2. La construcción de estos elementos de modelado consiste en:

- la definición de un **perfil** UML para modelar actividades de *testing* del sistema
- la definición de una **transformación** entre modelos (M2M) que genere el *DAT* a partir de un caso de uso con su documentación
- la aplicación del perfil al modelo de actividades del sistema que se genera por medio de la **transformación**

Finalmente, para cumplimentar la etapa 1.3, se define una **transformación** modelo a texto (M2T) a partir del *DAT* obtenido en el paso anterior que genere los casos de prueba y las pruebas de datos (de los datos necesarios para la ejecución de un caso de uso).
Estos elementos se presentan en el capítulo 6: "Implementación del proceso".

Los casos de prueba que se obtendrán serán:

- Uno por cada escenario (camino del diagrama de actividades)
- Uno por cada dato ingresado (identificados en la sección "datos ingresados" de la especificación del Caso de uso y también en las actividades "inputData")

Si bien no se especifica en la definición de este proceso qué tipo de criterio de aceptación se utilizará con las pruebas, pensamos en una generación de dato de prueba tendiente a "P**articiones equivalentes".**
En este sentido, la idea es intentar particionar los dominios de las entradas de la funcionalidad a probar, en un número de clases equivalentes (asumiendo que el test de un valor representativo es equivalente al test de cualquier otro valor de esa clase).
Así, un dato de prueba siempre se genera con el valor completado y se podrán generar otros dependiendo del tipo de dato ingresado.
En nuestro caso, sólo hicimos la definición para los *strings*, agregando al valor ingresado, la cadena vacía y una cadena 'aleatoria'.

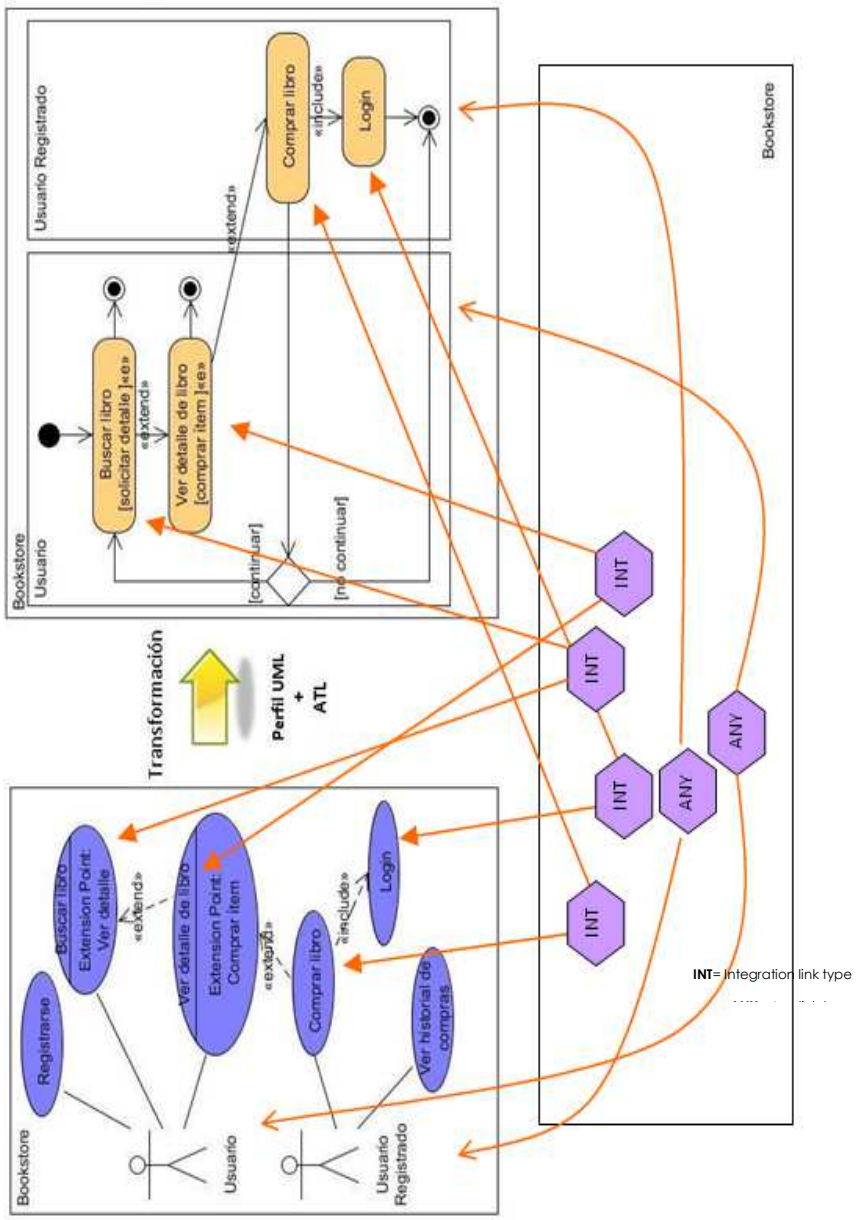
5.2.3 ADICIÓN DE TRAZABILIDAD

Del análisis realizado en el capítulo 3 de este trabajo "Trazabilidad en MDD/ MBT", concluimos que nuestra propuesta, al modelar trazas, debe contar con un metamodelo que lo soporte.
Éste debe contener los tipos de links predefinidos que son necesarios para notar los tipos de trazas. En nuestro caso, basta especificar que se trata de una traza a test de integración, test de unidad o caso de prueba de datos.
Además y como se propone en varios de los trabajos mencionados, es muy útil brindar la posibilidad de definir nuevos tipos que el ingeniero de software pueda requerir. En nuestro caso se trata de un tipo genérico.

De esta forma, se unifica criteriosamente a los tipos de trazas identificados y se permite la configuración de otros diferentes a los establecidos.
En tal sentido, el trabajo presentado por Amar, Leblanc y Coulette [24] se acerca más a los objetivos de nuestra propuesta. Hemos tomado ese trabajo y realizado las adaptaciones necesarias para modelar trazas que relacionen artefactos de *testing*.

El metamodelo definido así como los detalles de implementación para agregar las sentencias de código a las transformaciones que permiten generar los modelos de trazas, se presentan en el capítulo 6: "Implementación del proceso".

La Figura 5.11 muestra gráficamente parte del proceso general presentado al inicio del capítulo ahora con la adición de trazas. Allí se ve el modelo de Casos de Uso a partir del cual se generan el modelo intermedio DATS -Diagrama de Actividades de *Testing* del Sistema- y el modelo de trazas, obtenidos ambos de la primera transformación definida en nuestro proceso.



ansformación

A manera de cierre, se presenta el último gráfico del capítulo (Figura 5.12), intentado reflejar el proceso completo.
Al proceso ya definido se le agregan las nociones de trazabilidad mencionadas en esta última sección del capítulo. La generación de los modelos de trazas, obtenidos con cada una de las transformaciones mencionadas, permite establecer links entre los distintos artefactos de los modelos involucrados, por ejemplo, entre un caso de uso, la actividad de test y el caso de prueba que lo contiene.

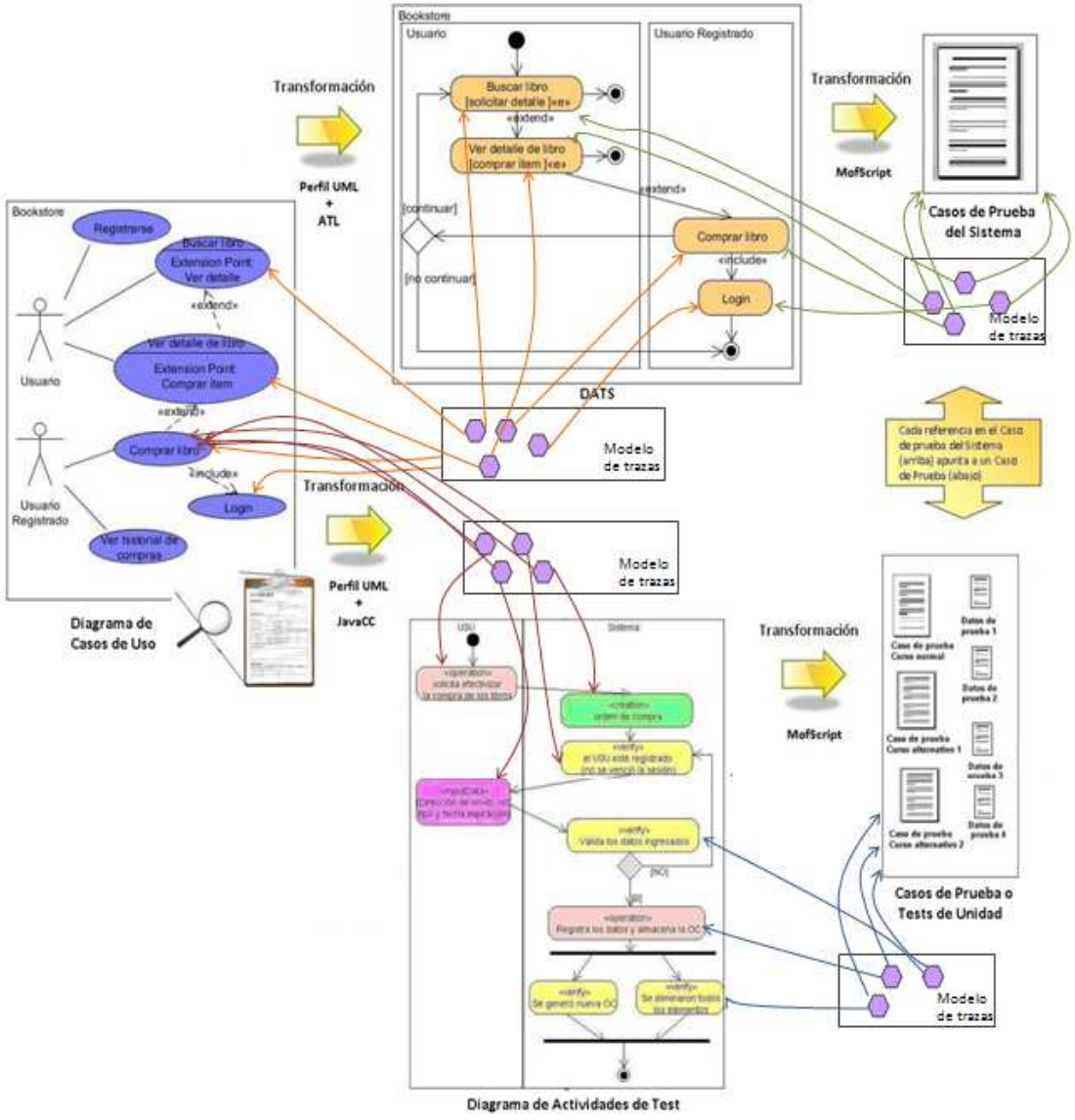


Figura 5.12. Proceso completo de generación de casos de prueba a partir de casos de uso

5.3 Resumen del Capítulo

En este capítulo hemos presentado el proceso que permite generar casos de prueba. Para su mejor comprensión y detalle, se lo ha dividido en dos partes.

Por un lado, se planteó la generación de un diagrama intermedio (DATS) para:

- Definir 'caminos de ejecución' del usuario dentro del sistema
- Brindar la posibilidad al ingeniero de requerimientos de completar o modificar esta visión
- Generar a partir de allí, los esquemas de test de integración necesarios para probar la funcionalidad del sistema.

Al contemplar todo el modelo de requerimientos desde una primera instancia, se asegura de contar con la totalidad de la funcionalidad y con una amplia cobertura de casos de prueba.

Por otra parte, se planteó la generación de *tests* de unidad y datos de prueba, en la segunda parte o subproceso.
Por un lado, se planteó la generación de un diagrama de actividades intermedio (DAT) para:

- Contemplar las ejecuciones posibles dentro de un Caso de uso, flujos normales y alternativos y ciclos
- Brindar la posibilidad al ingeniero de requerimientos de completar o modificar esta visión
- Generar a partir de allí, los esquemas de test de unidad necesarios para probar la funcionalidad del sistema, generando además pruebas de datos.

El Caso de Uso, en su documentación cuenta con la secuencia de pasos a realizarse para completar la funcionalidad. Se presenta, como en el MCU una visión de 'caminos de ejecución' podríamos decir ahora 'a nivel Caso de uso'.
De la misma forma que la visión planteada para el sistema, se asegura de contar con la totalidad de la funcionalidad y con una amplia cobertura de casos de prueba, sin omitir ninguno de los caminos que pueda tomar la ejecución (flujos normales y alternativos).
Los datos de prueba se generan a partir de los datos necesarios para la ejecución del Caso de uso (nombre de usuario, dirección, total, etc.) que en los diseños de pruebas de datos toman valores para testear la funcionalidad. Nuevamente, se cuenta con amplia cobertura, en este caso para los datos de prueba.

Finalizando el capítulo, se completa el proceso con la adición de trazabilidad entre los modelos.

Capítulo 6- Implementación de la Propuesta

En este capítulo se presenta la implementación (y todos los elementos auxiliares necesarios para su definición) del proceso que permite generar casos de prueba de forma automática mediante transformaciones de modelo, partiendo de la especificación de Casos de Uso y adicionando la generación de trazas entre los elementos de los distintos modelos.
Para su mejor comprensión, se describe el esquema general o especificación del proceso en primera instancia, para hacer luego una descripción detallada del mismo.

6.1 Introducción

Como parte de la implementación e introducción a este capítulo, queremos mencionar dónde –dentro de un proceso de desarrollo- puede aplicarse el proceso que este trabajo define de forma tal de contextualizarlo dentro del ámbito ingenieril de sistemas.

Se presenta entonces la Figura 6.1 la cual muestra al Proceso Unificado *–Unified Process UP-* (proceso de desarrollo de software) [61] con las sus fases y etapas, adicionando información gráfica sobre dónde puede aplicarse nuestra propuesta: en las actividades de Requisitos y de Pruebas, de las fases Inicio y Elaboración.

Si bien puede utilizarse en diversos procesos de desarrollo, consideramos UP por ser éste dirigido por casos de uso, incrementando iterativamente la definición de los mismos.

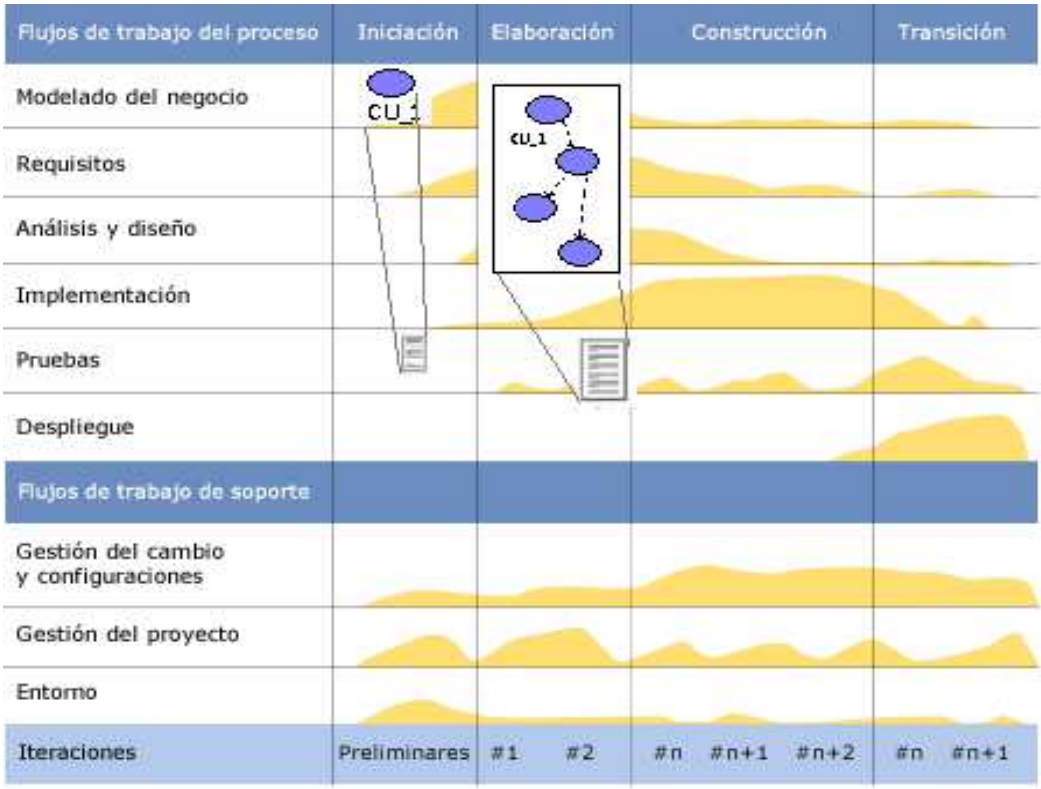


Figura 6.1. Generación de casos de prueba del sistema dentro de las actividades de RUP

6.2 Implementación del Proceso en etapas

En esta sección, se completan y describen los elementos auxiliares definidos para implementar el proceso definido en esta tesina.

PROCESO GENERAL

Como se mencionó en la sección anterior, el proceso general abarca una serie de etapas que se mencionan y describen a continuación.

ETAPA 1: Definición del Modelo de Casos de Uso -Modelo Independiente de la Plataforma (PIM) en MDD-

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

ETAPA 2: Definir y aplicar una transformación a partir del MCU para obtener un DATS

Se define y aplica en esta etapa una transformación del MCU para obtener un diagrama de actividades del sistema particular para *testing* (DATS)- otro modelo (PIM)

en MDD-, construido utilizando un Perfil UML definido para modelar actividades de *testing* del sistema.

Para obtener los posibles caminos de ejecución mencionados, se propone la generación de un 'Diagrama de Actividades del Sistema' que especifique distintas ejecuciones funcionales.

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

Veamos por casos:

- **Transformación de la relación <<include>> en secuencia de actividades**

  Si se define una secuencia en el diagrama de actividades por medio de una transición entre Act_A y Act_B para representar un <<include>>, no se respetaría su semántica. Una relación <<include>>, comienza su comportamiento en CU_A, hace un llamado a CU_B, ejecuta el comportamiento de CU_B y debe retornar a CU_A, mientras que la transición entre dos actividades Act_A y Act_B define que Act_B sólo se ejecutará una vez finalizada Act_A.

- **Transformación de la relación <<extend>> en secuencia de actividades**

  En este otro caso, tampoco es válido utilizar una secuencia entre actividades para denotar un <<extend>>, por dos motivos: en primer lugar, no existe semánticamente en casos de uso tal secuencialitas y, segundo, porque la opcionalidad de ejecución dada por la condición del punto de extensión se omitiría.

Ahora bien, como el Diagrama de Actividades del Sistema no refleja en forma precisa las relaciones entre Casos de Uso, se ve la necesidad de definir un perfil UML que agregue esta información al modelo destino, manteniendo la semántica de las relaciones entre casos de uso y actividades. A este diagrama generado lo denominaremos "Diagrama de Actividades de *Test* del Sistema" o simplemente DATS.

6.2.1 Perfil UML para Modelar Diagramas de Actividades de Testing

Los estereotipos definidos tienen base en la metaclase ActivityEdge del metamodelo UML v2.4. En esta versión, el elemento de modelado mencionado, reemplaza el uso de Transition en el modelado de actividades en UML 1.5. Acompaña a este diagrama de metaclases, una especificación formal de los nuevos estereotipos con sus restricciones definidas en OCL [8]. La figura 6.2 muestra la definición gráfica del perfil.

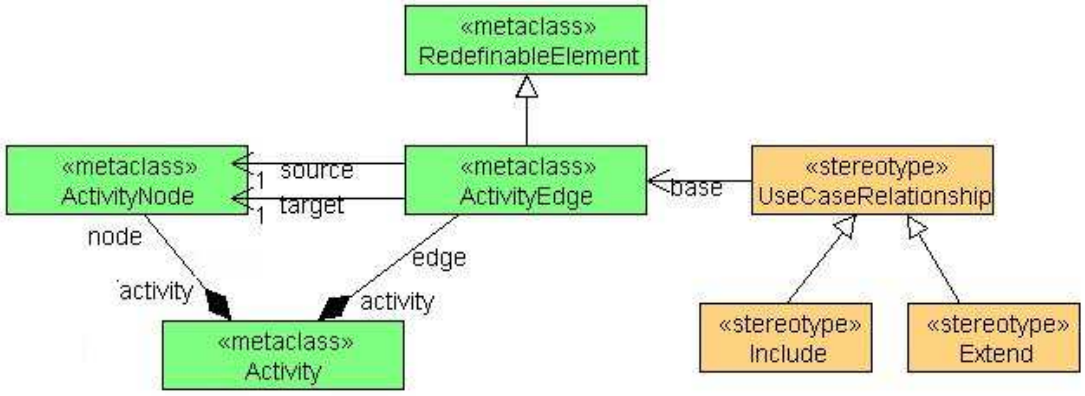


Figura 6.2. Perfil UML para actividades de *testing* del sistema con base en ActivityEdge

Para la definición de las reglas de buena formación descriptas a continuación, se utilizó la versión 2.2 de OCL que incorpora operadores de ejecución de operaciones y envío de señales. En este trabajo en particular, se hace uso del operador hasSent() -^- notado por '^.operation' que indica que la operación *operation* fue ejecutada.

STEREOTYPE <<UseCaseRelationship>>
Base: ActivityEdge
Constraints:

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

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

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

```
let performedActions: Sequence(OclMessage)=
   self.activity^action in
   performedActions -> notEmpty() and
   performedActions.hasReturned()

where

   action:: Sequence(ActivityNode) -> Sequence(Action)
   action= self.activity.edge-> collect:
   (aedge:ActivityEdge| aedge(OCLType)= Action)
```

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

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

STEREOTYPE << Extend >>
Base: ActivityEdge
Constraints:

Atributo *condition*: representa la condición que debe cumplirse cuando la ejecución llega al punto de extensión. Si la misma es verdadera, se ejecutará la actividad que extiende a la actividad actual (donde es especificado el punto de extensión). Si la condición es falsa, no se ejecuta la extensión. Si no hubiese una condición asociada, el punto de extensión es incondicional.

1- El punto de extensión referenciado por la actividad que extiende el comportamiento, debe pertenecer a la actividad que es extendida (de forma similar a la mecánica de extensión entre casos de uso).

```
let act: Sequence(OclType=Action) = self.activity.node
      in act-> includes (act.name->notEmpty())
```

Con estas reglas de buena formación, finaliza la definición del Perfil UML para Diagramas de Actividades.

6.2.2 Reglas de la Transformación de MCU a DATS

Con respecto a la transformación, definimos las siguientes reglas de transformación:

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

Relaciones explícitas

4. Cada relación entre dos casos de uso, se corresponde con una transición entre dos actividades.
   a. Si la relación es <<include>>, la transición será estereotipada con <<include>> indicando que se incluyen los pasos de la actividad de destino y que luego se retorna el flujo de ejecución a la actividad fuente.
   b. Si la relación es <<extend>>, la transición será estereotipada con <<extend>> y la actividad agregará una guarda <<E>> y una acción asociada que denotan el punto de extensión definido en el caso de uso.

   c. Si es una generalización, el caso de uso abstracto no se tiene en cuenta en la transformación. Los casos de uso que se transforman son los concretos o hijos.
5. Existen varios inicios y varios fines de ejecución en el *DATS*. Inicialmente, cada actividad tiene su actividad de inicio y de fin, indicando que puede realizarse sólo esa funcionalidad (por ejemplo, el usuario sólo quiere realizar búsquedas). Para aquellas funcionalidades relacionadas con otras, se agrega además la transición correspondiente de una hacia la otra, manteniendo el inicio y el fin correspondientes.

Esto último es indicativo de que el camino de ejecución puede terminar allí o bien puede continuar con otra funcionalidad. Por ejemplo, el usuario, luego de buscar un libro, se interesa en ver el detalle del mismo.

Para la definición de esta transformación se eligió el lenguaje de transformaciones ATL (ATLAS Transformation Language) [9]. ATL es un lenguaje de transformaciones, que cuenta con una herramienta de soporte que permite editar, definir y ejecutar transformaciones modelo a modelo (M2M). Dado que se ajusta a nuestros requerimientos, además de contar con buena documentación y ejemplificación, fue el lenguaje elegido para la definición de las transformaciones M2M.

Para realizar esta segunda etapa del proceso, fue necesario:

1. modificar el metamodelo UML con la definición del perfil UML presentado, de forma tal que el Diagrama de Actividades que se genere por medio de la transformación, exprese la semántica de las relaciones <<include>> y <<extend>> entre casos de uso
2. definir la transformación en ATL de Casos de Uso a Actividades (siguiendo las reglas de transformación ya definidas)

A continuación –en la Figura 6.3- se presenta parte del segundo paso, el de la transformación en ATL.

```
  ...
helper def: getId() : String =
      thisModule.refSetValue('currentId',
thisModule.currentId + 1).currentId.toString();

rule Main {
      from
            uc : UML!UseCase
      to
            act : UML!StateMachine (
                  acts <- UML!UseCases.allInstances()
            )
}

rule UseCases {
      from
            uc : UML!UseCases (uc.oclIsKindOf (UML!UseCase)
            )
      to
            act : UML!StateMachine (
                  actId <- thisModule.getId(),
                  name <- uc.name,
                  predecessors <- uc.getPredecessors()
            )
}
```

Figura 6.3. Parte de la transformación UC2DATS.atl

ETAPA 3: Definir y aplicar una transformación desde el DATS, para obtener todas las posibles secuencias de ejecución como integración de las distintas funcionalidades

Como mencionamos en el capítulo anterior, a partir del *DATS* generado en la etapa 2, se aplica otra transformación que produce un archivo XML con todas las posibles secuencias de ejecución que pueden realizarse en el sistema y que integran las distintas funcionalidades.
El DATS generado le permite al ingeniero de software tener una visión global de la secuencia de acciones que un usuario puede realizar en el sistema.

A continuación se enuncian las reglas de transformación.

6.2.3 Reglas de la Transformación de DATS a Casos de Prueba del Sistema

Definimos las siguientes reglas para esta transformación:

1. El nombre del archivo de casos de prueba del sistema o modelo destino, coincide con el nombre del *DATS* con el agregado de la extensión XML.
2. Por cada subgrafo conexo en el DATS:
   2.1. Se crea un número de secuencia de camino principal en el archivo destino

Mediante un algoritmo DFD se obtienen todos los caminos desde un punto inicial (actividad de inicio) hasta el punto final (actividad de fin), pudiéndose indicar el número de ciclos a especificar, siendo 2 el valor por defecto. Entonces:

2.2. Por cada camino de ejecución obtenido se le asigna un número de secuencia de camino en el archivo o modelo destino

2.3. Por cada actividad dentro de cada camino de ejecución, se genera una llamada a una funcionalidad; o sea, se nombra la actividad en el modelo destino.

Cabe aclarar que:

- los puntos de inicio y final son ahora únicos por cada subgrafo dirigido conexo.
- El número de secuencia, permitirá identificar con mejor precisión a la secuencia de ejecuciones que se realizan en un camino dado.

De esta manera y teniendo en cuenta la Figura 6.4 presentada asimismo en el capítulo anterior y con un ejemplo particular, todas las secuencias comienzan con 1 dado que el grafo presentado es el único grafo del diagrama (no hay subgrafos) y cada camino tendrá su número de secuencia indicativo. Por ejemplo, 1.1 denotará la secuencia de ejecución <A, fin>, 1.2 <A, B, fin>, 1.3 <A, B, A, fin>.

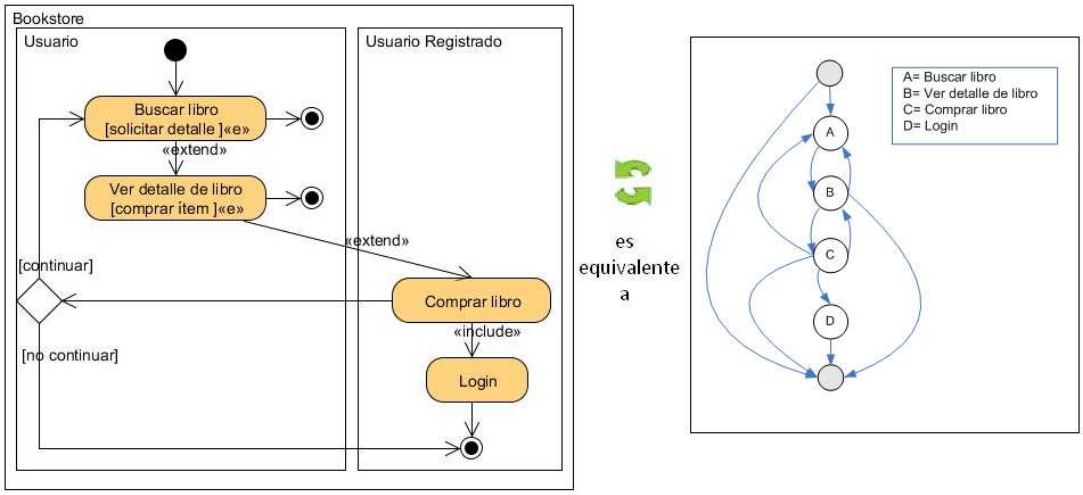


Figura 6.4. DATS visto como grafo dirigido

Finalmente, para generación de los caminos de ejecución, se utilizó una plantilla XML definida según otras especificaciones analizadas como [62], [63].
Los *tags* que se utilizaron para el archivo generado, entre otros, son:

- <use case (id)>
- <mainSequence>
- <step (id)>
- <alternative>
- <step (id, mainSeq, type)>
- <path>

En la Figura 6.5, se muestra con una parte del código de la transformación, cómo es la forma en que el archivo final es generado. Esta generación parte de las reglas de transformación mencionadas anteriormente.

```
texttransformation DATSToTest (in
                    model:"http://www.eclipse.org/uml2/2.1.0/UML")
{
model.Model::main () {
file ("test_" + self.name.firstToUpper() + ".txt");

self.ownedMember->
                    forEach(a: model.Activity)
                                {a.activityToTestLine();}
println ("}");}
//end of main

model.Activity::activityToTestLine(){

`
```

Figura 6.5. Extracto de código de la transformación en MOFscript

SUBPROCESO

A continuación se describe, ahora con mayor detalle, la implementación del subproceso. Recordando que nuestro propósito es definir un proceso que genere casos de prueba a partir de casos de uso, Es decir, el proceso general se centra más en el proceso de desarrollo que en los *tests* en sí mismos.

De la misma forma que el proceso general, el subproceso se basa en una serie de pasos que se detallan a continuación:

ETAPA 1 Y 1.1: Definición del Modelo de Casos de Uso -Modelo Independiente de la Plataforma (PIM) en MDD-

De la misma forma que la etapa 1 del proceso, este paso es el más artesanal. Se trata de completar la documentación de los Casos de Uso del Diagrama de casos de uso (MCU).

Los Casos de Uso, si bien pertenecen a un lenguaje estandarizado (UML), no han estandarizado su documentación (sólo se indica que posee un texto).
Para especificar la documentación de Casos de Uso, se han estudiado diversos autores, que se podrían dividir en dos grandes grupos:

- Propuestas para modelado de Casos de Uso
- Propuestas para modelado de Pruebas –o basadas en pruebas-

Se podría decir que 'el representante' del primer grupo es A. Cockburn con su

conocido trabajo de referencia en el tema [66].
En cuanto a diversas propuestas del segundo grupo ([67], [68], [7]0, [71]) encontramos en general menciones a los algunos pocos elementos a ser especificados en la documentación de los Casos de Uso. Es de notar que no se mencionan plantillas de documentación ni referencian a otros trabajos que los definan.

La Tabla 6.1 muestra los elementos que contienen las plantillas para especificar Casos de Uso (columna de la izquierda) o los elementos mencionados en los trabajos citados (columna de la derecha).

Tabla 6.1. Elementos considerados en la documentación de Casos de Uso

Propuestas para modelado de Casos de Uso	**Propuestas para modelado de Pruebas –basadas en pruebas-**
• Nombre • Actor primario • Actores secundarios • Objetivo • Nivel (del caso de uso) • Prioridad • Release/s • Condiciones de ejecución • Secuencia de acciones • Extensiones • Condiciones de extensión • Precondiciones • Postcondiciones • ... (entre otras)	• Nombre • Actor principal • Escenario principal • Cursos alternativos Alternativamente, se encuentran: • Precondiciones • Postcondiciones • Descripción

Para la generación del DAT que nos interesa, podemos indicar que la plantilla que es deseable que se utilice contenga los siguientes elementos especificados:

- *Identificador*
- *Nombre*
- *Descripción*
- *Incluye (Caso de Uso)*
- *Extiende (Caso de Uso)*
- *Precondiciones*
- *Actor principal*
- *Curso normal*
- *Cursos alternativos (con su nombre)*
- *Postcondiciones*
- *Datos de entrada (con su tipo de ser posible)*

Estos elementos son de interés tanto para la especificación de los Casos de uso, como para la posterior especificación de Casos de prueba.

Parte de esta investigación en Casos de Uso y en Casos de Prueba se utilizó para la definición de buenas prácticas y documentación interna del SEPG (Software Engineering Process Group) de Lifia (Laboratorio de Investigación y Formación en Informática Avanzada) [69].

Como será necesario 'parsear' la documentación del Caso de uso, se propone asimismo el siguiente formato para la descripción de los pasos de los cursos (normales y alternativos).

[Número]- [actor / sistema] [acción]

Donde Acción puede ser:

'solicita' operación
'verifica' objeto/ estado
'crea' objeto
'consulta' dato/ atributo
'ejecuta' cu

Habiendo presentado los requerimientos en cuanto a la escritura de la documentación de los Casos de Uso, introducimos a continuación la continuación del subproceso con la etapa 1.2.

ETAPA 1.2: Definir y aplicar una transformación a partir del MCU para obtener un DAT

A partir de cada Caso de Uso, se aplica una transformación del mismo a un Diagrama de Actividades particular para *testing* (DAT)-Modelo Específico para *Testing* (PIM) en MDD-, construido a través de un Perfil UML definido para modelar actividades de *Testing*. (Ver Figura 5.8).
A partir de la documentación de cada caso de uso, se genera un diagrama de actividades de *testing* intermedio que permite especificar detalladamente qué tarea realiza cada 'actividad de *testing*' generada a partir del caso de uso.

6.2.4 Perfil UML para Modelar Actividades de Testing

Un Diagrama de Actividades en UML no provee el comportamiento necesario para especificar tareas de *testing*, por lo tanto, se observa la necesidad de enriquecer la metaclase *Activity* de UML a través de estereotipos, formando así un perfil que defina en forma más explícita, qué tarea realiza cada actividad de *testing* dentro del caso de prueba. Es decir, una Actividad de *Testing* puede representar alguna de estas tareas:

- la ejecución de una operación
- la verificación de condiciones y estados
- la creación de objetos
- la validación de datos ingresados, y
- la llamada de ejecución a otros casos de uso.

La Figura 6.5 muestra la definición del perfil. Los estereotipos definidos tienen base en la metaclase *Activity* de UML permitiendo extender su definición. Acompaña a este diagrama de metaclases, una especificación formal de los nuevos estereotipos: restricciones en OCL [8] para expresar reglas de buena formación. La especificación completa de las mismas no se incluye en el presente trabajo por razones de espacio pero pueden ser consultadas en [16].

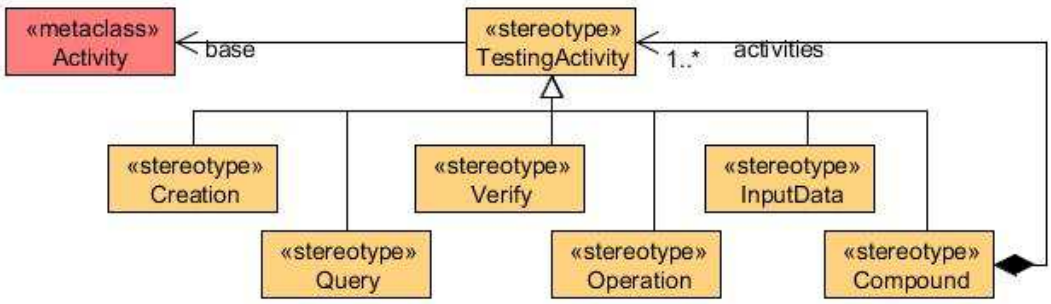


Figura 6.5. Perfil UML para actividades de *testing* con base en Activity

Las reglas de buena formación para el perfil definido se presentan a continuación:

STEREOTYPE Creation
Base: Activity
Constraints:

1- Una actividad de creación de conceptos (<< creation >>) se conforma de un atributo, concept.

```
self.attributes-> exists (att: StructuralFeature|
                                    att.name = 'concept')
```

2- El atributo concept de una actividad << creation >> no debe ser nulo.

```
self.concept notNil
```

STEREOTYPE Query
Base: Activity
Constraints:

1- Una actividad de consulta (<< query >>) contiene al menos una operación no nula

```
self.operations-> size() > 0 implies self.operations -> exists
op: BehavioralFeature| op notNil)
```

2- Una actividad << query >> contiene tantas variables (temporales) como operaciones no nulas

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

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

```
operacionesNoNulas: Set (BehavioralFeature) -> Set
(BehavioralFeature)
operacionesNoNulas (ops) = ops-> select(each| each notNil)
atributosNoNulos es una operación que retorna todos los
```

```
atributos no nulos de la actividad
atributosNoNulos: Set (StructuralFeature) -> Set
(StructuralFeature)
atributosNoNulos (atts) = atts-> select(each| each notNil)
```

STEREOTYPE Verify
Base: Activity
Constraints:

1- Una actividad << verify >> contiene un atributo "assertion" que almacena la premisa de la puede darse un valor de verdad.

```
self.attributes-> exists (att: StructuralFeature| att.name =
'assertion')
```

2- El atributo "assertion" de una actividad << verify >> no debe ser nulo.

```
self.assertion notNil
```

STEREOTYPE Operation
Base: Activity
Constraints:

1- Una actividad de operación contiene un atributo "operation" que representa la operación a ejecutar junto con sus parámetros.

```
self.attributes-> exists (att: StructuralFeature|
                                    att.name = 'operation')
```

2- El atributo "operation" de una actividad << operation >> no debe ser nulo.

```
self.operation notNil
```

STEREOTYPE InputData
Base: Activity
Constraints:

1- Una actividad de ingreso de datos contiene atributos "data1", "data2", .. , "datan" que almacenan los datos ingresados, al menos uno, no nulo.

```
self.attributes-> exists (att: StructuralFeature|
                                    att.name = 'data1') and
(self.attributes -> size() -> 0 implies self.attributes ->
                                    exists att:
StructuralFeature| att notNil))
```

2- Una actividad << inputData >> contiene tantas variables (temporales) como operaciones no nulas.

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

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

```
operacionesNoNulas: Set (BehavioralFeature) -> Set
(BehavioralFeature)
operacionesNoNulas (ops) = ops-> select(each| each notNil)
atributosNoNulos es una operación que retorna todos los
atributos no nulos de la actividad.
atributosNoNulos: Set (StructuralFeature) -> Set
(StructuralFeature)
atributosNoNulos (atts) = atts-> select(each| each notNil)
```

STEREOTYPE Compound
Base: Activity
Constraints:

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

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

2- Una actividad compuesta puede generar cambios no locales a su contexto. Esto se debe a que representa un << include >>, << extend >> o generalización entre casos de uso.

```
self.isReadOnly=false
```

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

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

6.2.5 Reglas de la Transformación de MCU a DATS

Definimos las siguientes reglas de transformación:

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

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

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

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

Para realizar esta segunda etapa del subproceso, fue necesario:

1. modificar el metamodelo UML con la definición del perfil UML presentado, de forma tal que el Diagrama de Actividades que se genere por medio de la transformación, contenga los tipos de actividades definidos
2. definir la transformación en ATL de (especificación de) Casos de Uso a Actividades (siguiendo las reglas de transformación ya definidas)

ETAPA 1.3: Definir y aplicar una transformación desde el DAT, para obtener casos de prueba de la funcionalidad y datos de prueba para la misma

Se ejecuta luego una transformación modelo a texto –PSM a PSM- que tiene como entrada al Diagrama de Actividades construido con el perfil de *testing* mencionado y como salida al caso de prueba (CP) y también las pruebas de datos (DP). Para ello, hemos definido una transformación en MOFscript.

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

1. Por cada camino de ejecución posible, lo que es equivalente a cada escenario especificado, se genera un caso de prueba distinto (bifurcaciones del diagrama de actividades).
2. El nombre del test proviene del nombre del diagrama de actividades acompañado de la partícula "test_" delante.
3. La nota del diagrama será el comentario de los casos de prueba.
4. Por cada actividad, y de acuerdo a su "tipo", se tiene su transformación dentro del test de la siguiente manera:
   - **Actividades <<creation>>**: crea un concepto. Es decir, instancia el concepto guardado en su atributo "concept".
   - **Actividades <<query>>**: realiza el llamado a la operación de consulta y almacena su respuesta.
   - **Actividades <<verify>>**: define aserciones para las premisas de sus atributos
   - **Actividades <<operation>>**: realiza el llamado a la operación del sistema
   - **Actividades <<inputData>>**: verifica que los atributos sean válidos (no nulos, no fuera de rango, etc.)
   - **Actividades <<compound>>**: indica la ejecución de un caso de uso.
5. Para las actividades **<<inputData>>** se generan pruebas de datos, por cada atributo de la actividad, de forma tal de tener una prueba también para los datos ingresados y de acuerdo a su tipo.

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

La transformación se detalla en el informe técnico en [64] y que se presenta sólo una parte en la Figura 6.6.

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

// Activity2TestLine
model.Activity::activityToTestLine(){

if (self.hasStereotype("creation")){
self.ownedMember -> forEach (att:model.StructuralFeature){
att.creationToTestLines();
                        }
                        }
else if (self.hasStereotype("query"))
self.ownedMember-> forEach(att: model.StructuralFeature){
att.queryToTestLines();
      }
else if (self.hasStereotype("verify"))
self.ownedAttribute-> forEach (att:model.StructuralFeature){
att.verifyToTestLines();
      }
      ...
}

// Creation2TestLines
model.StructuralFeature::creationToTestLines(){
println(self.name.first() + ":=" + self.name.firstToUpper() +
"new;");
}
```

Figura 6.6. Transformación en MOFscript: DAT2Test

Con la definición de esta transformación, finaliza la implementación del proceso de generación de Casos de prueba.

6.2.6 Metamodelo para Trazabilidad

Como mencionamos en los capítulo 4 y 5, el trabajo presentado por Amar, Leblanc y Coulette [46] en cuanto a la definición de un metamodelo -dentro del contexto de

interés de este trabajo- se acerca más a los objetivos de nuestra propuesta. Hemos tomado ese trabajo y realizado las adaptaciones necesarias para modelar trazas que relacionen artefactos de *testing*.

Esta propuesta ha seguido también las conclusiones realizadas por trabajos de investigación [47], [65] .En ambos casos se menciona, luego de realizar un estudio sobre el estado del arte de traceability, que una solución óptima debería proveer un metamodelo no totalmente genérico, sino con tipos de trazas predefinidas para el contexto de interés donde se vaya a aplicar, con algún tipo de traza no predefinida que sirva para propósitos generales.

La Figura 6.7 presenta el metamodelo para trazabilidad mencionado con la extensión para *testing* en recuadro punteado sobre el lado izquierdo.

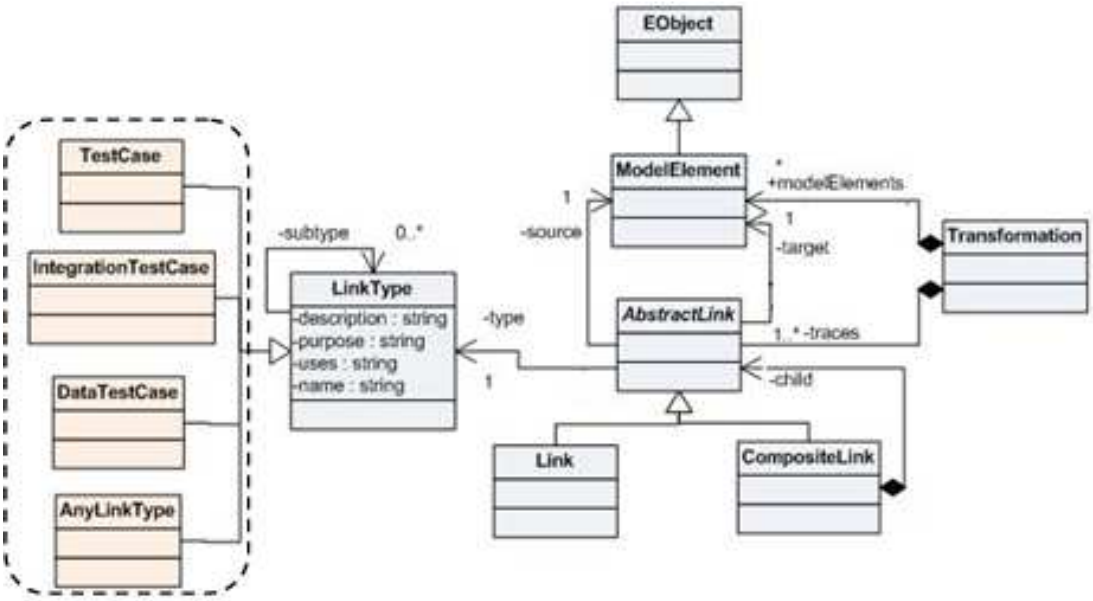


Figura 6.7. Metamodelo para trazabilidad con artefactos de *testing* (recuadro punteado)

El metamodelo define que una transformación se compone de un conjunto de trazas las cuales relacionan elementos del modelo origen y elementos del modelo destino. La noción de traza puede ser simple o compuesta, lo cual permite representar cadenas de transformaciones o trazas anidadas. Este hecho se verifica al definir transformaciones que llaman a otras. En nuestro caso, un caso de prueba puede requerir casos de pruebas de datos de los parámetros ingresados.
La adición de tipos de links tiene el objetivo de especificar los tipos de *tests* que pueden relacionar a los requerimientos con los artefactos de *tests*. En nuestro caso, los casos de prueba del sistema presentan *tests* de integridad entre las diferentes funcionalidades del sistema, cada caso de uso tiene sus correspondientes casos de prueba y los datos de ingreso, cuentan con sus casos de prueba de datos. Para cualquier especificación adicional de traza que requiera ser modelada, se cuenta con el elemento genérico *AnyLinkType*.

Nuestra propuesta pertenece al grupo de trazas impuestas ya que genera el modelo de trazas a partir de la ejecución de una transformación entre modelos origen y

destino en lugar utilizar técnicas de inferencia que, según el caso, deberían analizar los artefactos o el código para encontrar situaciones como que una operación llame a otra y por ello genere una relación de dependencia –traza- entre estos elementos.

A continuación, en la Figura 6.8 presentamos parte de la transformación que genera los elementos del modelo destino junto con las trazas. Se destaca aquí, en color rojo, sólo la adición de la generación de trazas. La misma se presenta en pseudocódigo "like ATL" [9], [27] para aportar mayor legibilidad al ejemplo presentado.

```
Transformation uc2Activity (UML uml, TraceModel traceM)
Input uml:uc
Output uml:activity, traceM:trace

{activity ←UML::Activity.new
 activity.name ← "ad" + uc.name
 initialNode:= UML::InitialNode.new
 activity.node.add (initialNode)
 …
 for each uc.actor in uc{
       partition ←UML.Partition.new
       partition.name ← uc.actor.name
       partitions.add (partition)
       invokeRule createTraces (uc.actor, partition,
                          AnyLinkType.new("actor2Partition"));
 }
 …
for each uc.useCase in uc{
       activity ←UML.Activity.new
       activity.name ← uc.name
       activity.node.add (activity)
       invokeRule createTraces (uc.useCase, activity,
       IntegrityTestCase.new ());
  }
 …
for each uc.relationship in uc{
       if (uc.relationship.isTypeOf ("include"))
              edge ← UML::ActivityEdge.new
              edge.stereotype ← "include"
              activity.edge.add (edge)
       else if (uc.relationship.isTypeOf ("extend"))
       …}
…
finalNode:= UML::FinalNode.new
activity.node.add (finalNode)
}

Transformation Rule createTraces (trace_source, trace_target, trace_type) {
traceLink ←CompositeLink.new
traceLink.source ← trace_source
```

Figura 6.8. Trasformación Caso de Uso a Actividad con la adición de trazas

6.3 Resumen del Capítulo

En este capítulo hemos presentado la aplicación de los conceptos teóricos presentados en los capítulos anteriores y que dan un sustento formal al trabajo de investigación que esta tesina presenta.
Las tareas desarrolladas han abarcado el estudio y aplicación de diversos aspectos de la Ingeniería de Software, desde la definición de modelos en UML hasta la definición de transformaciones; desde el estudio de paradigmas hasta el estudio de lenguajes y herramientas.

El proceso definido pretende asistir al ingeniero/ desarrollador/ tester para que desde un modelo de casos de uso inicial, pueda 'correr' transformaciones que le permitan generar diagramas intermedios. Estos diagramas le permiten visualizar y comprender así las funcionalidades del sistema, sus interrelaciones y los alcances definidos.

Luego, por medio de otras transformaciones, el proceso le permite generar casos de prueba -de integración, de unidad, datos de prueba- que lo asistan en la tarea de la definición de pruebas del sistema.

El objetivo asimismo es proporcionar una alta cobertura de generación de casos y datos de prueba y de integración de las funcionalidades, así como una determinación del orden de las pruebas o priorización de los mismos.

Finalmente, la generación de los modelos de traza le permitirá relacionar los elementos de modelado de los requerimientos y las pruebas, estableciendo asimismo un temprano análisis de impacto de los cambios que puedan surgir en el sistema.

Capítulo 7- Caso de Estudio

En este capítulo se presenta un ejemplo completo con la generación de los casos de prueba.
Con el objetivo de mostrar con mayor detalle el proceso definido –y las etapas intermedias-, el ejemplo que se elige no es trivial ni imaginado, sino un ejemplo tomado de la vida real.
Si bien el ejemplo puede considerarse como complejo, solamente nos enfocaremos en una parte reducida para apreciar los detalles del proceso.

7.1 Enunciado de Ejemplo

El ejemplo se basa en un sistema de venta de libros por Internet, al estilo Amazon, que permite vender y comprar libros. A continuación se enumeran los requisitos generales del sistema:

1. El *bookstore* será un sistema basado en la web para la venta de libros.
2. El *bookstore* tiene libros de los que se conoce el nombre, descripción, los autores, el precio. Los libros están organizados por categorías (novela, cuento, historia).
3. Un mismo libro puede tener varias versiones, algunas impresas y otras digitales. De los impresos se conoce el tamaño físico, la presentación, el stock y el peso para el envío. De los digitales su tamaño en bytes y el formato del archivo que lo contiene.
4. La presentación impresa puede ser de tapa dura, edición de lujo, etc.
5. El usuario debe poder agregar libros en un carrito de compras online (cart), previo a realizar la compra. Al momento de finalizar la compra, se generará una orden (order) que incluye los ítems (order item) agregados al carrito de compras. Una vez confirmada la orden, se le enviará un mail al usuario informándole los datos de la misma. Similarmente, el usuario debe poder sacar ítems de su carrito o actualizar las cantidades de un ítem pedido.
6. El usuario debe ser capaz de mantener una lista (wish lists) con los libros que desea comprar más tarde.
7. El usuario debe poder cancelar órdenes antes de que hayan sido enviadas.
8. El usuario debe poder pagar con tarjeta de crédito (credit card) o transferencia bancaria (bank transfer).
9. El usuario podría devolver libros que ha comprado.
10. El usuario debe poder crear una cuenta (client account), de forma tal que el sistema recuerde sus datos (nombre de usuario, nombre y apellido, dirección, email, detalles de su tarjeta de crédito) en el momento de registrarse (login).
    a. El sistema debe mantener una lista con las cuentas de los usuarios en su base de datos central.

b. Cuando un usuario se registra, su nombre de usuario y la palabra clave (password) debe coincidir con el nombre de usuario y la palabra clave que está almacenada en la base de datos.
c. Se debe permitir que el usuario modifique su password cuando lo requiera. De la misma manera, se debe permitir que modifique cualquier otro dato de su cuenta.

11. El usuario debe ser capaz de buscar libros usando distintos métodos de búsqueda – título, autor, palabra clave o categoría – y luego podrá ver los detalles (book Spec) de los resultados de su búsqueda.
12. El usuario debe poder escribir una opinión acerca de sus libros favoritos. Las opiniones (review comments) aparecerán junto con los detalles del libro.

El enunciado completo, fue tomado de [3]. En las siguientes subsecciones se muestran los pasos del proceso definido en esta tesina, aplicados a este ejemplo.

7.2 Caso de Estudio paso a paso

A continuación se aplica el ejemplo siguiendo la presentación que se hizo en los capítulos anteriores sobre las etapas que el proceso comprende.

PROCESO

ETAPA 1: Definición del Modelo de Casos de Uso -Modelo Independiente de la Plataforma (PIM) en MDD-

La Figura 7.1 presenta el Diagrama de Casos de uso del sistema. No es el Diagrama de casos de uso completo. Se redujo el modelado a unos pocos Casos de uso conteniendo una relación <<include>> y un <<extend>> a efectos de contar con todos los elementos de modelado pero sin agregar una complejidad innecesaria.

Nuestro sistema tiene dos actores, Usuario y Usuario Registrado que podrán:

- registrarse
- buscar libro
- ver los detalles de un libro en particular
- ingresar al sistema (*Login*)
- agregar un libro al carrito de compras
- ver el historial de compras, entre otras cosas.

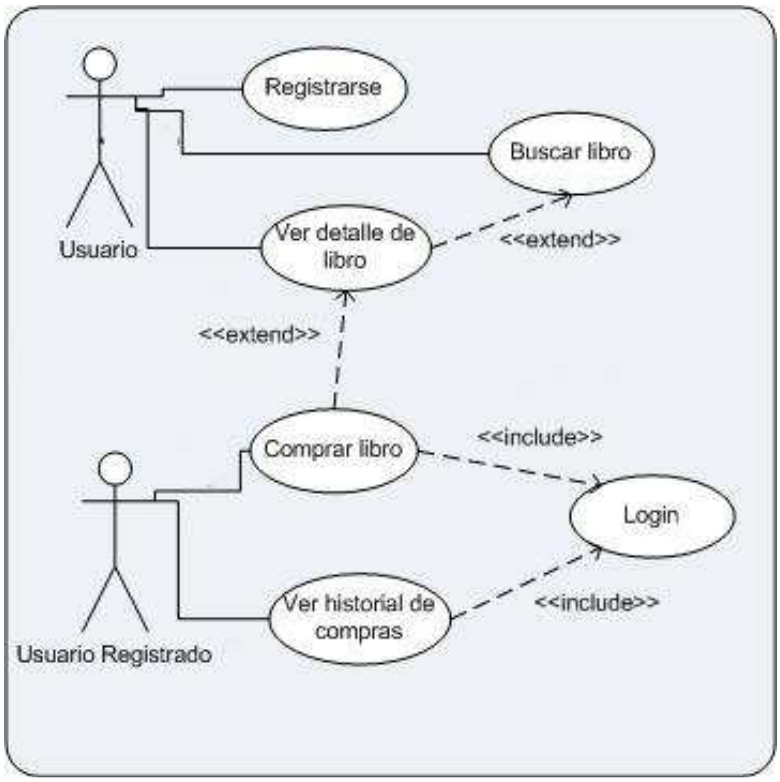


Figura 7.1. Modelo de Casos de Uso de Bookstore

ETAPA 2: Definir y aplicar una transformación a partir del MCU para obtener un DATS

Al correr la transformación, se genera como resultado un archivo XMI con los elementos generados por la transformación.

Este archivo puede ser 'leído' por herramientas de modelado que permiten visualizar gráficamente el modelo generado.

A manera de ejemplo, se presenta la Figura 7.2 con el Diagrama de Actividades del Sistema generado.

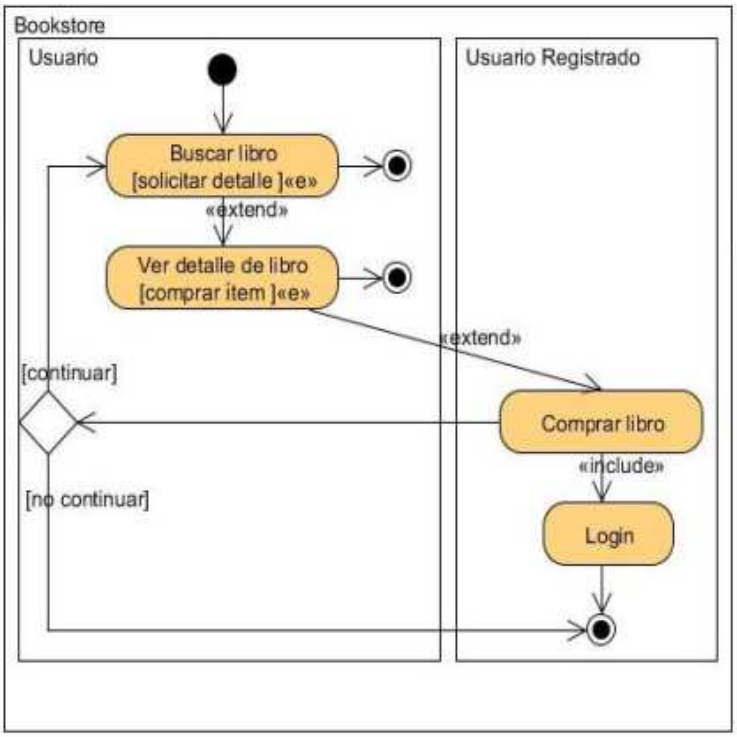


Figura 7.2. DATS generado a partir de la transformación

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

En este punto, el ingeniero de software podría querer incluir otras relaciones que no se explicitaron en el diagrama de casos de uso, pero que serían útiles para la posterior generación de los casos de prueba. Por lo tanto es útil ver a nuestro DATS como un grafo dirigido ya que pueden inferirse los posibles caminos que un actor puede ejecutar en el sistema.

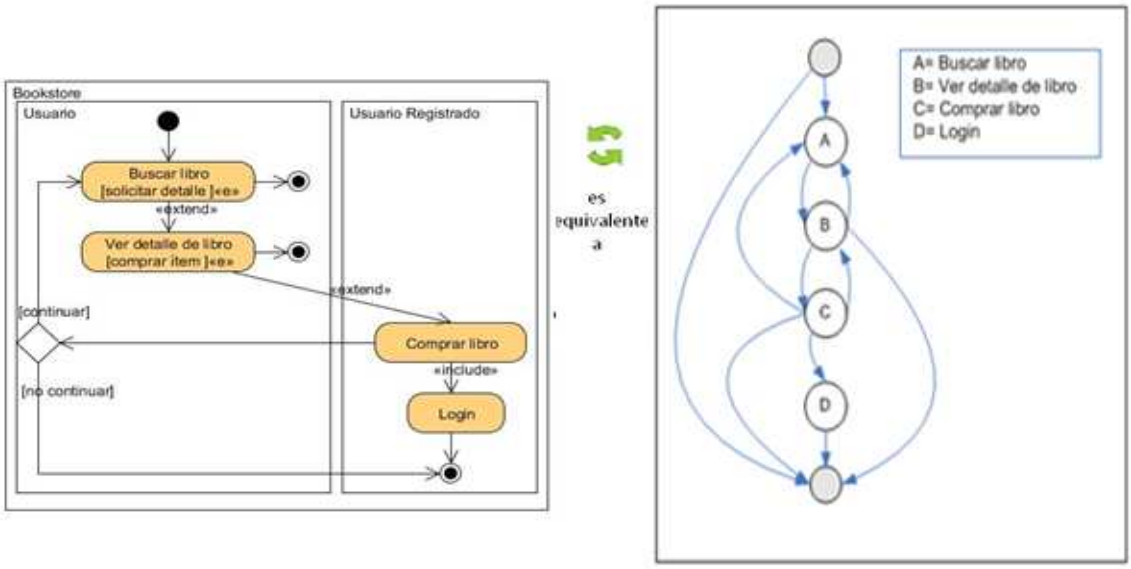


Figura 7.3. DATS visto como grafo.

Testear esos caminos de ejecución brinda la ventaja de conocer si una secuencia de acciones puede ser realizada o no en el sistema; o bien, si ese camino es exitoso, falla, o genera un error.

ETAPA 3: Definir y aplicar una transformación desde el DATS, para obtener todas las posibles secuencias de ejecución como integración de las distintas funcionalidades

Luego de la ejecución de la transformación de DATS a casos de prueba, se obtiene un archivo con todos los caminos posibles que un usuario puede realizar sobre el sistema, contemplando los ciclos de ejecución.

La figura 7.4 muestra parte del archivo generado con los posibles caminos de ejecución para el Modelo de Casos de uso del ejemplo. Estos son los casos de prueba del sistema o *tests* de integración.

```
<useCaseModel id="Bookstore">
  <mainPath id="1">
            <path id="1">
                      <useCase id="01-Buscar libro"/>
                      <useCase id="10-FIN"/>
             </path>
            <path id="2">
                      <useCase id="01-Buscar libro"/>
                       <useCase id="02-ver detalle de libro"/>
                       <useCase id="10-FIN"/>
              </path>
  ...
</mainPath>
</useCaseModel>
```

Figura 7.4. Parte de los casos de prueba generados, caminos de ejecución del usuario

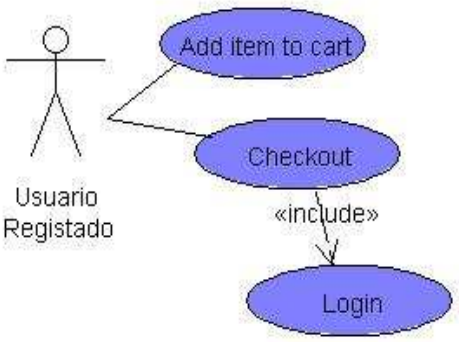
Con esto, finalizan las etapas del proceso. A continuación, el subproceso.

SUBPROCESO

De la misma forma que lo hicimos para el proceso general, el subproceso se basa en una serie de pasos que se detallan a continuación:

ETAPA 1 Y 1.1: Definición del Modelo de Casos de Uso -Modelo Independiente de la Plataforma (PIM) en MDD-

Aquí es necesario especificar la documentación del Caso de Uso. A modo de ejemplo, se documenta un caso de uso. Para esta documentación se utilizan las secciones propuestas en el capítulo 5, etapas 1 y 1.1 del subproceso y se visualiza en la Figura 7.5.



ID_CU: 06
Nombre: Checkout
Actor: usuario (USU)
Descripción: El usuario solicita efectivizar la compra de los libros agregados a su carrito de compras.
Incluye: Login
Precondición: El carrito tiene 1 ó más ítems.
El usuario está registrado en el sistema".
Postcondición: Se generó una nueva orden de compra.
Se eliminaron todos los elementos del carrito de compras.
Curso Normal
1- El USU solicita efectivizar la compra de los libros de su carrito
2- El sistema consulta el carrito de compras de la sesión de USU
3- El sistema crea una orden de compra
4- El sistema verifica que el USU esté registrado (no se venció la sesión)
5- El sistema solicita agregar los ítems del carrito a la orden de compra (OC)
6- El sistema consulta la OC y el precio total a pagar
7- El sistema solicita ingresar datos: dirección de envío y los datos de la tarjeta de crédito
8- El usuario solicita enviar los datos: dirección de envío, los datos de su tarjeta de crédito
9- El usuario solicita realizar la compra
10- El sistema valida los datos ingresados
11- El sistema registra los datos y almacena la OC
Curso Alternativo- Usuario no logueado
3.1-El usuario no está logueado o se venció su sesión de usuario -> ejecutar CU_Login
Curso Alternativo- Datos inválidos

Figura 7.5. Documentación del CU_Checkout

ETAPA 1.2: Definir y aplicar una transformación a partir del MCU para obtener un DAT

Al correr la transformación definida para este paso, se genera como resultado un diagrama de Actividades que especifica la secuencia de acciones del Caso de Uso.

De forma similar a la etapa 2 del proceso, este archivo puede ser 'leído' por herramientas de modelado que permiten visualizar gráficamente el modelo generado.

A manera de ejemplo, se presenta la Figura 7.6 con el Diagrama de Actividades generado para la documentación del Caso de uso de ejemplo.

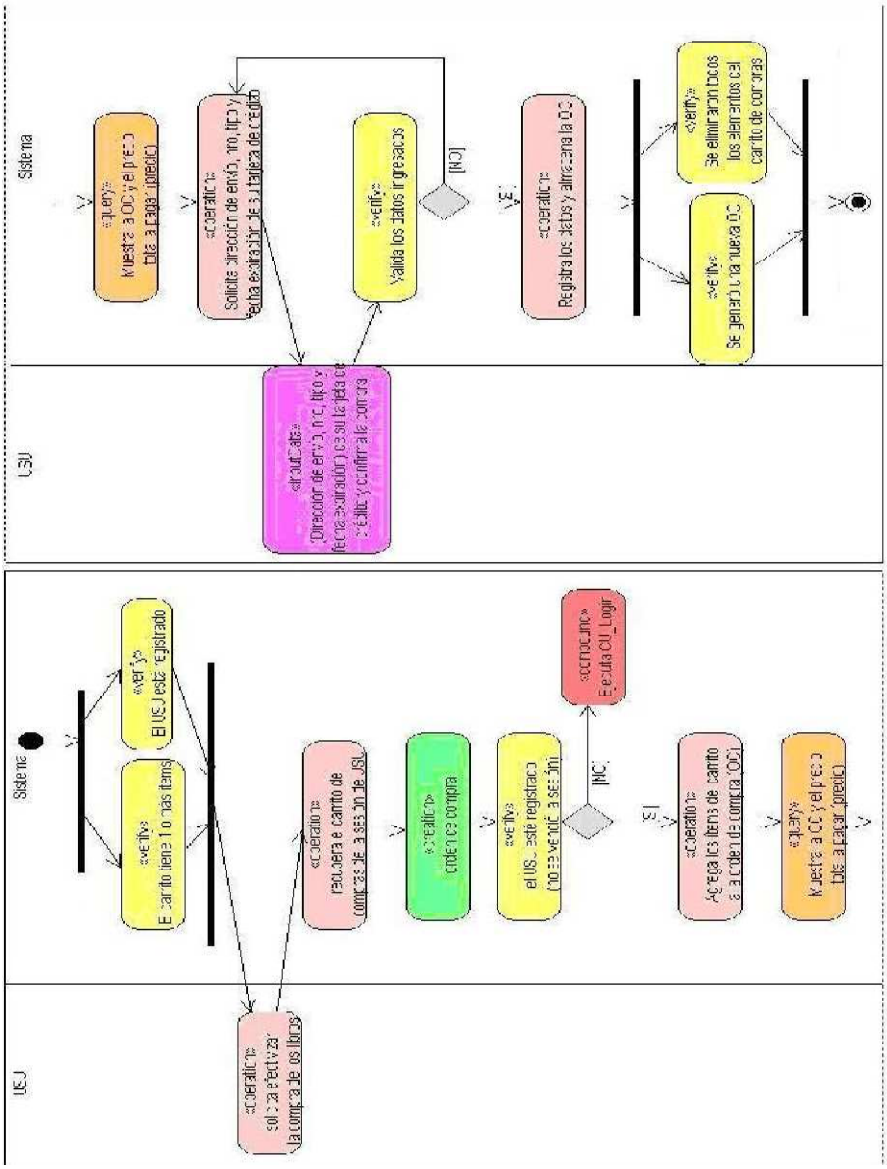


Figura 7.6. Diagrama de actividades de *testing* generado a partir del caso de uso

A continuación se presenta la última etapa del subproceso.

ETAPA 1.3: Definir y aplicar una transformación desde el DAT, para obtener casos de prueba de la funcionalidad y datos de prueba para la misma

Luego de la ejecución de la transformación de DAT a casos de prueba, se obtienen una serie de archivos, a saber:

- Uno por cada camino en el DAT a caso de prueba (prueba de unidad)
- Uno por cada dato de entrada definido (pruebas de datos)

La figura 7.7 muestra parte del archivo generado con los posibles caminos de ejecución para el Modelo de Casos de uso del ejemplo. Estos son los casos de prueba del sistema o *tests* de integración.

```
test_Checkout (){
/**
El usuario solicita efectivizar la compra de los libros agregados a su carrito de
compras.
*/

self assert: “El carrito tiene 1 ó más ítems”.
self assert:”El usuario está registrado en el sistema”.
USU.efectivizarCompraDeLibros;
```

```
test_dato_de_prueba
CU:  Checkout ()

ESC: Checkout
CAMPO: dirección de envío (str)
VALOR:' 55 nro. 5555'
```

```
test_dato_de_prueba
CU:  Checkout ()

ESC: Checkout
CAMPO: dirección de envío (str)
VALOR:' '
```

Figura 7.7. Caso de prueba y datos de prueba del dato "dirección de envío" con un valor ingresado y con la cadena vacía

7.3 Resumen del Capítulo

En este capítulo hemos presentado un caso de estudio en el que se visualizan claramente los pasos del proceso, las etapas y los resultados obtenidos.
Se destaca, la posibilidad de editar el diagrama de actividades intermedio, adaptándolo a lo que el ingeniero de requerimientos/ tester quiera precisar.
Esto se debe a que el formato final de la transformación en ATL es un formato estándar que puede ser 'leido' por las herramientas de modelado.

En cuanto al proceso, cabe destacar que tanto la cantidad de pasos intermedios así como las especificaciones necesarias para la generación de los casos de prueba no son demasiadas, haciendo de éste, un proceso sencillo y ágil.

Capítulo 8 - Conclusiones

En este capítulo se presentan en primera instancia, distintos trabajos relacionados con esta tesina. Luego, se destacan los aportes de este trabajo a la Ingeniería de Software y en particular, al paradigma MDD. Para cerrar el capítulo, se exponen las conclusiones finales y las líneas de trabajo futuro.

8.1 Trabajos Relacionados

En esta sección se destacan trabajos relacionados con nuestra propuesta, divididos por temas en tres subsecciones, haciendo una comparación en cada caso.
En primer lugar, se mencionan trabajos relacionados con nuestra propuesta, en los contextos de interés.
En la segunda subsección se detallan trabajos relacionados con el nuestro por elegir al MCU como modelo inicial para generar otros modelos.
En la subsección final se señalan trabajos vinculados desde el punto de vista de la trazabilidad.

8.1.1 Trabajos Relacionados con la Generación de Casos de Prueba – en MDD y MBT-

La generación de casos de prueba en los contextos de interés, en particular en MDD, ha sido estudiada en varias propuestas y publicaciones.

Muchos de ellos se centran y comienzan sus planteos a partir de Diagramas de clase UML.
A manera de ejemplo, las propuestas [70], [72], [73] generan la estructura de los *tests* a partir de los Diagramas de clase. Con variantes, se definen Casos de prueba utilizando Diagramas de clase para definir la estructura de los *tests*, transformando:

- cada clase de diseño en una clase de prueba
- cada operación de la clase en un mensaje a ser testeado.

De esta forma, se generan las estructuras de los Casos de prueba –en algún lenguaje XUnit-.

Otras propuestas completan lo mencionado, especificando el comportamiento de los *tests* a través de diagramas de secuencia de *testing* para las operaciones a validar.
Para ello es necesario contar con la estructura del caso de prueba (por medio de un diagrama de clases) y de los objetos del sistema. Mediante una derivación o transformación del Diagrama de secuencia se completa la funcionalidad del *test* a la estructura ya definida. Ejemplo de esto son los trabajos de investigación y tesis [74], [75].

Por lo estudiado en otras propuestas, observamos que generalmente se utilizan Diagramas de clases. En nuestra propuesta, hacemos uso de los Diagramas de actividades.

La primera razón de esta elección es la fase temprana del proceso de desarrollo en la que centramos nuestro proceso. Otra razón, es que permite visualizar en forma clara y precisa, todos los posibles caminos de ejecución (flujos normal y alternativos del caso de uso) adicionando como actividades también a las pre y post condiciones y las llamadas a otros casos de uso.
La coincidencia se da en la elección de la automatización de los pasos/ etapas por medio de transformaciones de modelo.

8.1.2 Trabajos Relacionados con la Generación de Modelos a partir de Casos de Uso –en MDD-

Uno de los trabajos estudiados fue el de Heumann [68]. El objetivo del trabajo tiene que ver con iniciar tempranamente la actividad de *testing* dentro del ciclo de vida del desarrollo. Para ello, este autor se vale del Modelo de Casos de Uso como modelo iniciador del método y describe una serie de pasos a seguir para generar los casos de prueba.
Estos pasos son:

1. Por cada Caso de Uso en el MCU, generar un set de escenarios del mismo.
2. Para cada uno de esos escenarios, identificar al menos un caso de prueba.
3. Para cada caso de prueba, identificar los datos del casos de uso y datos para realizar pruebas.

De forma manual en cada uno de los pasos, se sugiere tomar cada Caso de uso y listar los escenarios que se describen en el mismo.
Luego crear una matriz donde las columnas son el id del Caso de prueba, el número del escenario, el resultado esperado además de tener una por cada dato que se ingrese y cada requisito de ejecución.
El último paso completa las filas de la matriz mencionada con los datos requeridos en el encabezado de cada columna.
Podemos decir que finalmente se obtiene una lista de casos de prueba, con los valores que se deben probar y los resultados esperados.

Nuestra propuesta comparte dos aspectos esenciales con este planteo: el objetivo y la simplicidad del método.
Básicamente se diferencian en la automaticidad que nuestro trabajo le imprime a la generación de casos de prueba, además de contemplar otros tipos de prueba además de la adición de trazabilidad.

En [71] Riebisch et al. definen una propuesta para realizar pruebas estadísticas automáticas se centra inicialmente, en la transformación automática de un MCU a un "modelo de uso".
El método comienza con una extensión de los Casos de uso completándolos con:

- Precondiciones
- Postcondiciones
- Curso alternativa al camino de ejecución principal.
- Referencia a otros casos de uso relacionados.

Luego, ese modelo ya completo se traduce a Diagramas de estado –similares a los de Máquinas de Estado de UML- y se elabora el "modelo de uso". Este modelo tiene una utilidad auxiliar, porque es donde se indica la probabilidad de que ocurra una transición y se identifican los caminos de ejecución más frecuentes. Es decir no representa al proceso en sí mismo.

Por último, se extraen los modelos de prueba a partir de los modelos de uso y se generan recorridos aleatorios sobre cada modelo de uso. Cada recorrido aleatorio será un caso de prueba. Este método está parcialmente automatizado.

Nuestra propuesta comparte algunos puntos con este trabajo y se diferencia en otros. En común, se encuentra la necesidad de unificar/ especificar la documentación de los Casos de uso con miras a las 'pruebas' y la identificación de 'caminos' de ejecución. Básicamente, se diferencian en la cantidad de pasos, modelos intermedios y adhesión a estándares, que nuestra propuesta tiene en cuenta.

8.1.3 Trabajos Relacionados con Trazabilidad en MDD y MBT

Entre las propuestas de trazabilidad en MDD/ MBT, podemos mencionar a los siguientes trabajos:
Los autores Amar, Leblanc y Coulette en [46] presentan un metamodelo que permite definir diferentes tipos de links entre los elementos de modelado que se relacionan.
Entre sus puntos positivos, vemos que a nivel metamodelo, se contempla la posibilidad de que los links puedan anidarse. La utilidad que le encontramos es la de poder definir un tipo de traza de mayor nivel o jerarquía que contiene a trazas más 'simples': una traza compuesta de trazas simples o bien de trazas que contienen a otras.
Esta propuesta es ideal para la representación de trazas en la composición de transformaciones. Además de utilizar las meta-metaclases de MOF.

Nuestra propuesta, a diferencia de algunas de estas propuestas que resultan más genéricas por el contexto MDD, propone un metamodelo de trazas con tipos de links definidos para artefactos de *testing*.

Por último, otro trabajo [54], propone el uso de tecnologías XML para la definición de trazabilidad en MBT. En este caso, la traza es una relación elemento a elemento definida con una estructura XML formal: RDML (*Relation Definition Markup Language*). Para cada tipo de modelo utilizado, una definición RDML -basado en un esquema XML- especifica la relación con otro modelo y su tipo, siendo el tipo "derivado" o bien, "referenciado".

Si bien esta idea permite una rápida definición de la traza y cuenta con la facilidad de estar basada en un esquema XML, carece de especificación del tipo de traza, a diferencia del nuestro.

8.2 Aportes

Los aportes que brinda esta tesina a la Ingeniería de software en general y al paradigma MDD/MBT, son:

- En general, la definición de un proceso de generación de modelos de *testing* a partir de modelos de Casos de Uso dentro de los paradigmas MDD y MBT.

- En particular, la definición de este proceso incluye:

    - La determinación de elementos de modelado específicos para diagramas de actividades relacionados con Casos de uso.

- La definición de perfiles UML para *testing*, con su metamodelo y reglas de buena formación en OCL.
- La definición de las transformaciones, modelo a modelo y modelo a texto que permiten automatizar la generación de los modelos de *testing*.
- La definición de un metamodelo para trazabilidad.
- La definición de transformaciones modelo a modelo y modelo a texto que permiten automatizar la generación de trazas.

El aporte de este trabajo es valioso en el área de MDD/ MBT, ya que sobre la base de un análisis cuidadoso de las necesidades existentes y de los métodos, lenguajes y herramientas que le dan soporte, hemos generado un proceso de generación de Casos de prueba automatizado con transformaciones de modelo que enriquece al proceso MDD y al proceso de testing.

Además, podemos detallar el trabajo realizado durante las etapas de investigación y desarrollo de esta propuesta:

- Un estudio sobre los paradigmas MDD y MBT.
- Una recopilación y resumen sobre Lenguajes de Transformación en MDD.
- La definición de un esquema para la evaluación de lenguajes de transformación, junto con un análisis y evaluación comparativa de los mismos con el esquema mencionado.
- Una revisión del estado del arte en *Traceability*, en general en MDD y en particular en MBT, los contextos de interés.
- Un revisión sobre las especificaciones y templates de documentación para Casos de Uso
- Cuatro publicaciones en congresos.

8.3 Conclusiones finales

En este trabajo, se ha analizado que las actividades de testeo a nivel modelado, muchas veces se realizan en la etapa de diseño del sistema, con los detalles de implementación ya definidos. Sin embargo, es en etapas iniciales del desarrollo de software, cuando se define claramente la funcionalidad del sistema, indicando el "qué" sin mencionar el "cómo".

Esta tesina presenta un proceso, dentro de los contextos MDD/MBT, que permite generar casos de prueba tempranamente en el ciclo de vida del desarrollo, de manera automática por medio de transformaciones y con generación de trazabilidad. Se brinda, de esta manera, soporte al testing de sistemas y a las técnicas de trazabilidad dentro de los contextos MDD y MBT.

Específicamente, se propuso una extensión con conceptos de testing a un metamodelo existente para trazabilidad. En esta extensión, se definen los tipos de links necesarios para modelar las trazas permitiendo además definir otros tipos según la necesidad del usuario. Estos tipos de trazas son específicos al contexto de testing en MBT. Asimismo, se extendieron las transformaciones de modelos que permiten generar modelos de tests (definidas en trabajos anteriores) con la generación de las trazas.

Como ventajas, podemos indicar que nuestra propuesta permite que se cuente con un modelo separado de los modelos de origen y destino para las trazas, evitando

sobrecargar los modelos con información referente a la trazabilidad.

Podemos considerar que el aporte de este trabajo es valioso en el área de MDD/ MBT, ya que sobre la base de un análisis cuidadoso de las necesidades existentes y de los métodos, lenguajes y herramientas que le dan soporte, hemos generado un proceso de generación de Casos de prueba con transformaciones de modelo y trazabilidad entre modelos que enriquece al proceso MDD y al proceso de testing en particular.

Además, nuestra propuesta se basa completamente en la aplicación de estándares, tanto de lenguajes y herramientas, ampliamente aceptados en la comunidad de MDD.

Si bien resta aún realizar experimentos de aplicación en casos industriales reales, a partir de los casos de estudio desarrollados se vislumbran su aplicabilidad y sus ventajas.

8.4 Trabajo futuro

En base a las conclusiones abordadas y al objetivo alcanzado, se mencionan las siguientes líneas de trabajo futuro:

- Desarrollar experimentos de aplicación en casos industriales reales, para validar más fehacientemente la propuesta y refinarla de ser necesario.

- Definir una herramienta de soporte que permita automatizar completamente el proceso permitiendo aplicar al Diagrama de Casos de Uso (MCU) las transformaciones que generen los diagramas de actividades intermedios con conceptos de *testing* y sus consecuentes transformaciones a casos de pruebas.

- Incorporar opciones de métricas al proceso definido y a la herramienta. Entre ellas: de estimación por medio de UCP (Use Case Points); de cobertura para casos de prueba; de análisis de impacto por cambios introducidos.

Referencias

1. Geraci, A.: IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries. Institute of Electrical and Electronics Engineers Inc. (1991)
2. Kleppe, A., Warmer J., Bast, W.: "MDA Explained: The Model Driven Architecture: Practice and Promise." Addison-Wesley (2003).
3. Pons, C., Giandini, R., Pérez, G.: "Desarrollo de Software Dirigido por Modelos. Conceptos teóricos y su aplicación práctica". EDULP & McGraw-Hill Educación. (2010).
4. Gherbi, T., Meslati, D. and Borne, I.: "MDE between Promises and Challenges". Proceedings of UKSim 2009: 11th International Conference on Computer Modelling and Simulation, 2009, pp.152-155.
5. Baker, P. et all: "Model-Driven Testing Using the UML Testing Profile". Springer-Verlag
6. Blackburn, M., Busser, R., Nauman, A.: "Why model-based test automation is different and what you should know to get started". International Conference of Practical Software Quality & Testing (2004).
7. UML 2.4.1. The Unified Modeling Language Superstructure version 2.4.1. OMG Final Adopted Specification. http://www.omg.org/spec/UML/2.4.1 (2011).
8. OCL 2.2. The Object Constraint Language Specification-OMG. http://www.omg.org (2010).
9. ATL (ATLAS Transformation Language) http://www.eclipse.org/m2m/atl/
10. MOFScript Home page - http://www.eclipse.org/gmt/mofscript/ (actualizado 2015)
11. Atkinson, C. and Kuhne, T.: "Model-driven development: a metamodeling foundation". Journal of IEEE Software. Vol. 20, Nº 5, pp. 36-41. (2003)
12. MDA Guide, v 2.0 http://www.omg.org/cgi-bin/doc?ormsc/14-06-01. (2014).
13. OMG (Object Management Group) http://www.omg.org
14. XML Metadata Interchange (XMI), v2.1, http://www.omg.org/cgibin/doc?formal/2005-09-01, (full specification)
15. Giandini, R. :"Un Marco Formal para Transformaciones en la Ingeniería de Software Conducida por Modelos". Tesis de Doctorado en Cs. Informáticas. UNLP (2007)
16. Estier, T. What is BNF notation?
http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html#Naur60
17. Meta Object Facility (MOF) 2.0 Core Specification. OMG (2005).
18. MOF QVT final adopted specification v1.2. Technical Report http://www.omg.org/spec/QVT/1.2, OMG (2015).
19. Beizer, B.: Software Testing Techniques. Van Nostrand Reinhold (1990)
20. Utting, M., Legeard, B.: "Practical Model-Based Testing: A Tools Approach". Morgan-Kaufmann (2010)
21. Baker, P. et all: Model-Driven Testing Using the UML Testing Profile. Springer-Verlag
22. Blackburn, M., Busser, R., Nauman, A.: "Why model-based test automation is different and
what you should know to get started". Int.Conf. Practical Software Quality & Testing (2004).
23. Dai, Z. et all: "Model-Driven Testing with UML 2.0"
24.Correa, N., Giandini, R.: "Lenguajes de Transformación de Modelos. Un análisis comparativo". CACIC: Congreso Argentino de Ciencias de la Computación. (2007).
25. Czarnecki, Helsen. "Feature-based survey of model transformation approaches" .IBM System Journal. Vol. 45, Nº 3 (2006)
26. Jouault, F., Kurtev, I.: "Transforming Models with ATL". Workshop in Model Transformation in Practice. MoDELS 2005 (2005)
27. ATL /User Guide - The ATL Language. Comunidad Eclipse.
https://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language (actualizado /2015)
28. Epsilon Language & tools: http://www.eclipse.org/epsilon/ (actualizado 2014)

29. Balasubramanian, D. et all: "The Graph Rewriting and Transformation Language: GReAT". Proceedings of the Third International Workshop on Graph Based Tools (GraBaTs) (2006)
30. A. Cicchetti, D. Di Ruscio, R. Eramo, A. Pierantonio, JTL: a bidirectional and change propagating transformation language, 3rd International Conference on Software Language Engineering (2010)
31. Janus Homepage. http://jtl.di.univaq.it/index.php (actualizado 2010)
32. Akehurst, D., Howells, W., McDonald-Maier K.: Kent Model Transformation Language. Workshop in Model Transformation in Practice. MoDELS 2005 (2005)
33. Peltier, M., Bézivin, J., Guillaume, G.: "MTRANS: A general framework based on XSLT for model transformations". Proceedings of the Workshop on Transformations in UML (2001)
34. Oldevik, J.: "MOFScript User Guide". MOFScript v 1.4.0 (2011)
35. Kalnins A., Barzdins J., Celms E.: "Model Transformation Language MOLA". Proceedings of MDAFA 2004, pp.14-28 (2004)
36. Kalnina, E. et all: "Generation mechanisms in graphical template language". Proceedings of 2nd International Workshop on Model-Driven Architecture and Modeling Theory-Driven Development MDA&MTDD 2010, pp. 43-52. (2010)
37. Tratt, L.: "The MT model transformation language". Proceedings of the 2006 ACM symposium on Applied Computing, pp 1296 - 1303 (2006)
38. Akehurst, D., Howells, W.,McDonald-Maier, K.: "Model Transformation Language". Workshop in Model Transformation in Practice. MoDELS 2005 (2005)
39. Akehurst, D. et all: SiTra: Simple Transformations in Java. 9TH International Conference on Model Driven Engineering Languages and Systems, LNCS, Vol. 4199, pp. 351-364 (2006)
40. Sitra tutorials & documentation. http://www.cs.bham.ac.uk/~bxb/Sitra/index.html (actualizado 2011)
41. Program-Transformation.Org. Stratego: Strategies for Program Transformation. Program-Transformation (2004). http://strategoxt.org/ (actualizado 2013)
42. Lawley, M.,Steel, J.: "Practical Declarative Model Transformation with TefKat". Workshop in
Model Transformation in Practice. MoDELS 2005 (2005)
43. Willink, E. "UMLX - A graphical transformation language for MDA". OOPSLA 2003 Conference (2003)
44. IEEE. IEEE Recommended Practice for Software Requirements Specications (1998)
45. Galvão, I., Goknil, A.: "Survey of Traceability Approaches in Model-Driven Engineering". IEEE International EDOC Enterprise Computing Conference (2007).
46. Amar, B., Leblanc, H., Coulette, B.: "A Traceability Engine Dedicated to Model Transformation for Software Engineering". Traceability Workshop of the European Conference on MDA (2008).
47. Letelier, P.: "A framework for requirements traceability in UML-based projects". Intl. Workshop on Traceability in Emerging Forms of Software Engineering (2012).
48. Drivalos, N., Paige, R., Fernandes, K., Kolovos, D.: "Towards rigorously defined model-to-model traceability". Traceability Workshop of the European Conference on MDA (2008).
49. Martínez Grassi, O., Pons, C.: "Variable-Based Analysis for Traceability in Models Transformation". SADIO Electronic Journal of Informatics and Operations Research. ISSN 1514-6774 vol. 12, no. 1 (2013)
50. Triskel project (IRISA). The Metamodeling Language Kermeta. http://www.kermeta.org
51. Abbors, F., Bäcklund, A., Truscan, D.: "MATERA - An Integrated Framework for Model Based Testing". IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (2010)
52. Abbors, F., Bäcklund, A., Truscan, D.: "Tracing Requeriments in a Model-Based Testing Approach". First International Conference on Advances in System Testing and Validation Lifecycle (2009)
53. OMG SysML final adopted specification. http://www.omg.org/spec/SysML/1.4/.

(2014).
54. Pasupulati, B. "Traceability in Model Based Testing". Tesis de Master of Software Engineering. School of Innovation, Design and Engineering. Sweden (2009).
55. George, M. et all: "Traceability in Model-Based Testing". Article of Future Internet.An Open Access Journal from MDPI (2012). www.mdpi.com/journal/futureinternet
56. Correa, N., Giandini, R.: "Casos de Prueba del Sistema Generados en el Contexto MDD/MDT". ASSE 2012. Argentine Symp on 41 JAIIO (2012)
57. Correa, N., Giandini, R.: "Generación Automática de Casos de Prueba a partir de Casos de Uso: Una Propuesta Basada en MDD/MDT". ASSE 2011. Argentine Symp on 40 JAIIO (2011)
58. Correa, N., Giandini, R.: "Integrando Trazabilidad a la Generación de Casos de Prueba del Sistema: una propuesta MDD/MBT". Memorias de la XVI Conferencia Iberoamericana de "Software Engineering" (CIbSE) (2013). ISBN: 978-9974-8379-1-1.
59. Briand, L., Labiche, Y.: A UML-Based approach to system testing. 4th International Conference on the Unified Modeling Language, Modeling Language and Tools, London (2001).
60. Java Compiler Compiler. http://javacc.java.net/
61. Kruchten, P.: The Rational Unified Process. Addison Wesley (2000)
62. Gutiérrez, J. "Definición de casos de uso para generación automática de pruebas del sistema". Documento interno. Universidad de Sevilla (2006)
63. Gutiérrez, J. et all: "Using use case scenarios and operational variables for generating test objectives". Workshop on Systems Testing and Validation (STV) (2007)
64. Informe técnico del Proceso de Generación de Casos de Prueba del Sistema: https://sol.lifia.info.unlp.edu.ar/~nataliac/
65. Aizenbud-Reshef, N et all.:"Model Traceability". IBM Systems Journal, Vol. 45 Nº 3, pp. 515–526 (2006).
66. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley. (2001)
67. Briand, L: " An UML-Based Approach to System Testing using TOTEM" (2001)
68. Heumann,J.: "Generating Test Cases From Use Cases," *The rational edge* http://download.boulder.ibm.com/ibmdl/pub/software/dw/rationaledge/jun01/GeneratingTestCasesFromUseCasesJune01.pdf (2001)
69. Wiki de SEPG (Software Engineering Process Group) de Lifia (Laboratorio de Investigación y Formación en Informática Avanzada).
http://www.tecnologia.lifia.info.unlp.edu.ar/mediawiki/index.php/Portada (2009)
70. Hartman, A., Nagin, K.: The AGEDIS Tools for Model Based Testing. UML Satellite Activities (2004)
71. Riebisch, M. et all: "UML-Based Statistical Test Case Generation". International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World, pp. 394-411 (2003)
72. Abadía, A., Barisich, J.: Testing Basado en Modelos. Especificación Gráfica y Derivación Automática de Código. Tesis de Grado. Facultad de Informática, Universidad Nacional de La Plata (2009)
73.Boyapati,C., Khurshid,S., Marinov,D.: Korat: Automated Testing Based on Java Predicates, MIT Laboratory for Computing Science, MA, USA (2002)
74. Palacios, L.: Perfiles de Testing Aplicados a Modelos de Software. Tesis de Magister. Facultad de Informática, Universidad Nacional de La Plata (2009)
75. Javed, A., Strooper, P., Watson, G.:Automated Generation of Test Cases Using Model-Driven Architecture. 2nd. International Workshop on Automation of Software Test (AST '07), IEEE (2007)

Anexo I- Publicaciones Relacionadas a esta Tesina. Artículos Completos

Anexo II- Casos de Uso y Diagramas de Actividad

En este anexo se presentan los Diagramas de Casos de Uso y los Diagramas de Actividades de UML. Dado que han sido elegidos como diagramas –inicial e intermedio- del proceso desarrollado en esta tesina.
En cada una de las dos subsecciones de este anexo, se detallan ambos diagramas: su objetivo, elementos de modelado y nociones más importantes.
Se incluye una breve introducción sobre UML, el Lenguaje Unificado de Modelado.

Introducción a UML

UML es un lenguaje gráfico aceptado como estándar (por la OMG) para modelar sistemas de software, resultante de la combinación de las notaciones más conocidas en la década del '90, tales como OMT y Booch y los casos de uso de Jacobson.
El Lenguaje Unificado de Modelado se ha convertido en una notación estándar para el modelado de sistemas con gran cantidad de software.

UML es un lenguaje gráfico estándar por OMG para:

- visualizar
- especificar
- construir
- documentar

los artefactos de un sistema.

Donde un artefacto es una pieza de información usada o producida por un proceso de desarrollo de software.

Antes de introducir los Diagramas de Casos de Uso, mencionamos los beneficios de modelar antes de construir. Los modelos:

- Ayudan a la comprensión de sistemas complejos
- Indican QUÉ hará el sistema pero NO CÓMO lo hará
- Ayuda a la corrección de errores
- Ayuda a la evolución y reuso
- Esencial para la comunicación entre miembros de un equipo:
  - Expertos del dominio
  - Usuarios
  - Analistas Funcionales
  - Arquitectos de Software
  - Programadores

Diagramas de Casos de Uso

El modelado de Casos de Uso describe qué cosas debe hacer el sistema para satisfacer las necesidades de los distintos usuarios.

Los propósitos principales del modelado de Casos de Uso son:

- Decidir y describir los requerimientos funcionales del sistema, logrando un acuerdo entre los futuros usuarios del sistema, y los desarrolladores del mismo.
- Dar una clara y consistente descripción de qué debe hacer el sistema, para que este modelado pueda ser utilizado a lo largo de todo el desarrollo del sistema por los diferentes equipos de desarrollo que estén involucrados.
- Proveer las bases para realizar pruebas y verificar que el sistema trabaje apropiadamente. Por ejemplo, preguntando si el sistema actual realiza las funcionalidades que inicialmente fueron solicitadas.

Un Caso de Uso se representa como un "conjunto de acciones realizadas por el sistema, que retornan un resultado observable, que típicamente es un valor para uno o más actores del mismo". El caso de uso es una estructura que ayuda a los analistas a trabajar con los usuarios para determinar la forma en que se usará un sistema.

Un Caso de Uso es, en esencia, una interacción típica entre un usuario y un sistema. Podemos decir también que:

- Representa una unidad funcional completa del sistema.
- El Caso de Uso capta una función visible para el usuario.
- El Caso de Uso logra un objetivo concreto para el usuario

Características generales:

- Siempre se inicia por un actor.
- Provee un valor para un actor.
- Es completo.

Los elementos de modelado de un Diagrama de Casos de Uso son básicamente tres, como se representa en la Figura II.1:

- Actores
- Casos de Uso
- Relaciones

Los casos de uso se mencionan al inicio de esta sección y las relaciones se detallan a continuación.
En cuanto a los actores, podemos decir que un actor modela un tipo de rol que juega una entidad que interacciona con el sistema pero que es externa a él.

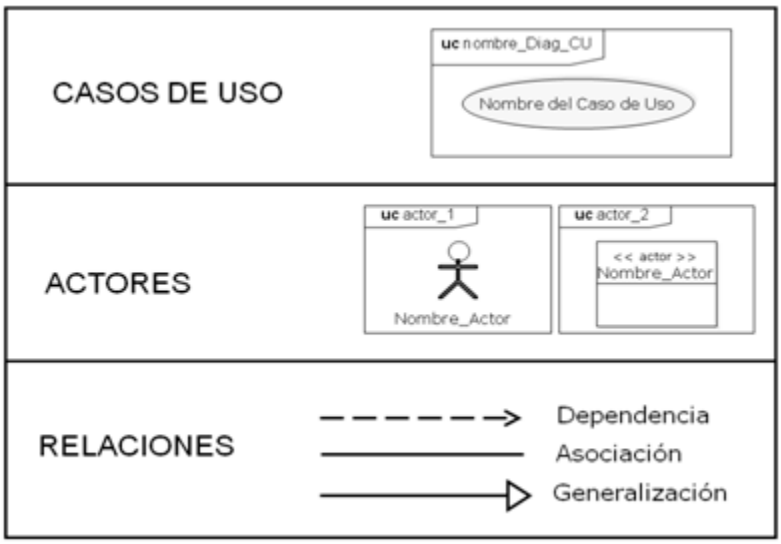


Figura AII.1. Elementos de modelado de Diagrama de Casos de Uso

Las relaciones a utilizar en los Diagramas de Casos de Uso se muestran en la Figura AII.2.

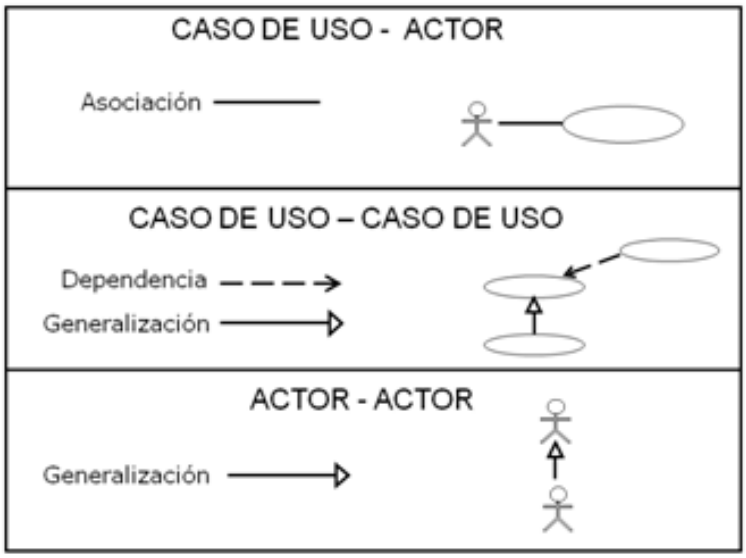


Figura AII.2. Relaciones entre elementos de modelado de Diagrama de Casos de Uso

Se describen a continuación:

- **Relaciones de asociación:** un Caso de Uso se conecta a los actores a través de relaciones de asociación, las cuales a veces son referenciadas con el nombre de "asociaciones de comunicación". Las asociaciones muestran los actores que se comunican con un Caso de Uso, incluyendo aquél actor que inicia la ejecución del caso de uso. Las asociaciones siempre deben ser binarias. Implican un diálogo entre el actor y el sistema. Adicionalmente, UML permite la utilización de multiplicidades al final de las asociaciones (en este caso nos referimos a que un actor se comunica con una o más instancias de ese caso de uso).

- **Relaciones de generalización**: son relaciones de "herencia" que se utilizan tanto entre Casos de Uso, como entre actores.
- **Relaciones de dependencia:** son relaciones de uso, utilizadas entre Casos de Uso. Como veremos, esta relaciones se van a utilizar con estereotipos (<<extend>>, <<include>>)

En cuanto a las relaciones entre elementos de modelado, podemos mencionar que:

- **una relación <<include>>** entre casos de uso especifica que un caso de uso incluye comportamiento definido en otro caso de uso. Esta relación determina que durante la ejecución de una funcionalidad, tiene lugar la ejecución de otra función del sistema. Para ser más explícitos, si un caso de uso CU_A incluye a un caso de uso CU_B, CU_B se ejecutará al ser invocado explícitamente durante la ejecución de CU_A. Finalizado CU_B, el flujo de ejecución vuelve a CU_A. La figura AII.3 presenta la noción gráfica de la ejecución de la relación <<include>>.

Figura AII.3. Relación <<include>> (izquierda) y gráfica de su ejecución (derecha)

- **una relación <<extend>>** entre casos de uso especifica que el comportamiento definido en un caso de uso puede ser extendido por el comportamiento de otro caso de uso. Esta extensión se lleva a cabo en uno o más puntos específicos (puntos de extensión) definidos en el caso de uso extendido, cuando la condición asociada al punto de extensión es verdadera. En otras palabras, que un caso de uso CU_A está relacionado mediante un <<extend>> con CU_B, indica que CU_B será ejecutado durante la ejecución de CU_A sólo si se cumple cierta condición definida en el punto de extensión de la relación. La figura AII.4 presenta la noción gráfica de la ejecución de la relación <<extend>>.

Figura AII.4. Relación <<extend>> (izquierda) y gráfica de su ejecución (derecha)

Documentación de los Casos de Uso

Para cada caso de uso se crea un documento que describe lo que el sistema le da al actor cuando el caso de uso es ejecutado.
Contenidos típicos:

- Flujo normal de eventos
- Flujo alternativo de eventos

Estas son las secciones para el modelo de documentación que proponemos en función a las distintas plantillas que actualmente existen en el estudio de análisis de requerimientos:

- Identificador
- Nombre
- Descripción
- Incluye (Caso de Uso)
- Extiende (Caso de Uso)
- Precondiciones
- Actor principal
- Curso normal
- Cursos alternativos (con su nombre)
- Postcondiciones
- Datos de entrada (con su tipo de ser posible)

Vista de un Diagrama simplificado de Casos de Uso con los elementos mencionados en esta sección.

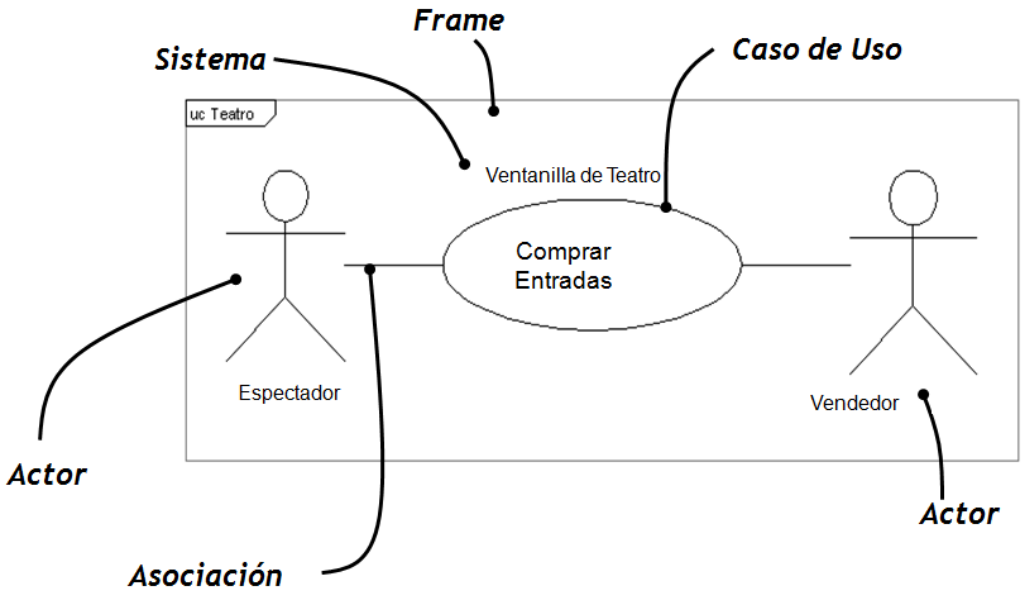


Figura AII.5. Ejemplo de un Diagrama de Casos de Uso

Diagramas de Actividades

Un Diagrama de Actividades es un tipo especial de Diagrama de Máquina de Estados. Tiene cierta ssimilitud con los Diagramas de flujos en programación (pasos – puntos de decisión – bifurcaciones) y son muy útiles para mostrar los procesos de negocio/operaciones.

Fueron diseñados para mostrar una visión simplificada de lo que ocurre durante una operación / proceso.
Un diagrama de actividades muestra básicamente el flujo de actividades en un dominio específico intentando modelar el comportamiento dinámico. Representa el comportamiento mediante un modelo de flujo de control entre las distintas actividades, cumpliendo una finalidad. Destaca la actividad a lo largo del tiempo.

Se lo puede pensar como un Diagrama de Interacción, con algunas diferencias:

- Un Diagrama de Interacción muestra objetos que se envían **mensajes**.
- Un Diagrama de Actividades muestra las **operaciones** que se dan entre los objetos.

Los elementos de modelado de un Diagrama de Actividades son básicamente tres, como se representa en la Figura II.6.

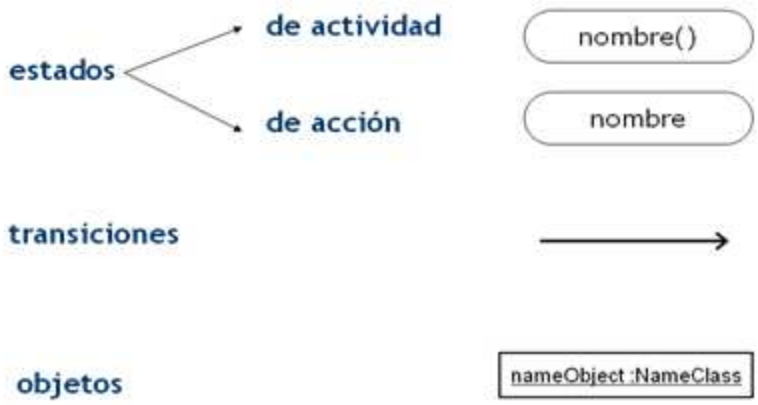


Figura II.6. Elementos de modelado de un Diagrama de Actividades

Una actividad (o un estado de actividad) es una especificación de un comportamiento parametrizado que se expresa como un flujo de ejecución por medio de una secuencia de unidades subordinadas.

Una acción (o un estado de acción) es una especificación de una unidad fundamental de comportamiento que representa una transformación o procesamiento. Las acciones están compuestas por cálculos atómicos ejecutables que producen un cambio de estado o la devolución de un valor. Ejemplos de acciones son los siguientes:

- Una acción es un estado atómico.
- Llamadas a otras operaciones.
- Creación de objetos.
- Destrucción de objetos.
- Obtener o setear un valor.

Una transición denota el paso de una actividad a otra. Puede estar acompañada de

guardas booleanas que indican una condición a cumplirse para que la actividad a la que llega la transición sea ejecutada.

Entre los elementos, hay nodos que se destacan. Tienen una función particular dentro del diagrama. Estos se muestran en la figura a continuación.

Figura II.7. Tipos de nodos de un Diagrama de Actividades

Otros de los elementos notacionales de los Diagramas de actividades, son los elementos de interconexión, que se describen a continuación:

- Bifurcaciones: indican una división o bifurcación en el flujo de control
- Uniones: indican una reunión del flujo de control
- Swimlanes: notan una división de las actividades según el rol que las ejecuta

Para finalizar se cierra este Anexo con una vista de un Diagrama simplificado de Actividades con los elementos mencionados en esta sección.

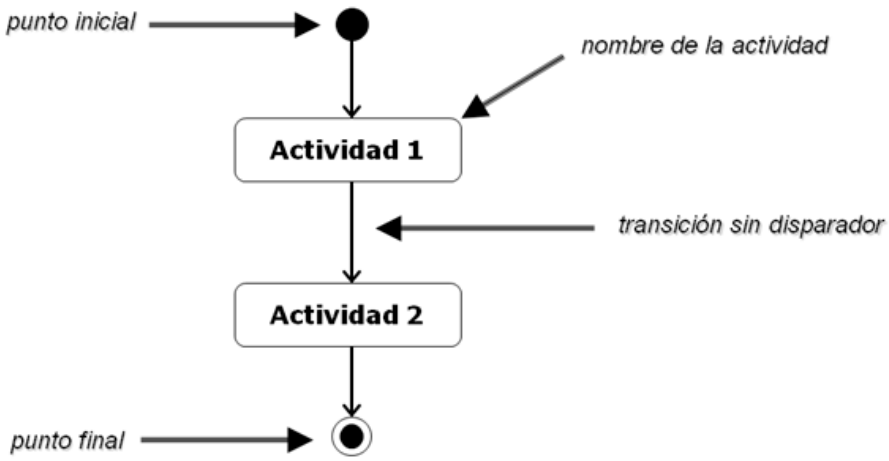


Figura AII.8. Ejemplo de un Diagrama de Actividades

Anexo III- Esquema de Clasificación para los Lenguajes de Transformación

Este anexo presenta en detalle el esquema de clasificación definido para evaluar lenguajes de transformación de modelos, de manera tal que ante el abanico de posibilidades y según los requerimientos de los usuarios, se puede tomar una decisión de uso bajo los criterios definidos.

El anexo en general se divide en dos partes: la definición del esquema por un lado y la especificación de los elementos incluidos en la misma por otro.

Esquema de clasificación

Czarnecki y Helsen en [25], reconociendo la necesidad de conocer sus características, propusieron una clasificación para lenguajes de transformación de modelos.

En nuestra propuesta, tomamos como base el trabajo realizado por estos autores, reordenando algunas características y adicionando otras.

En la Figura AII.1 puede observarse en color rosa la propuesta original y en color celeste la extensión sugerida por nosotros.

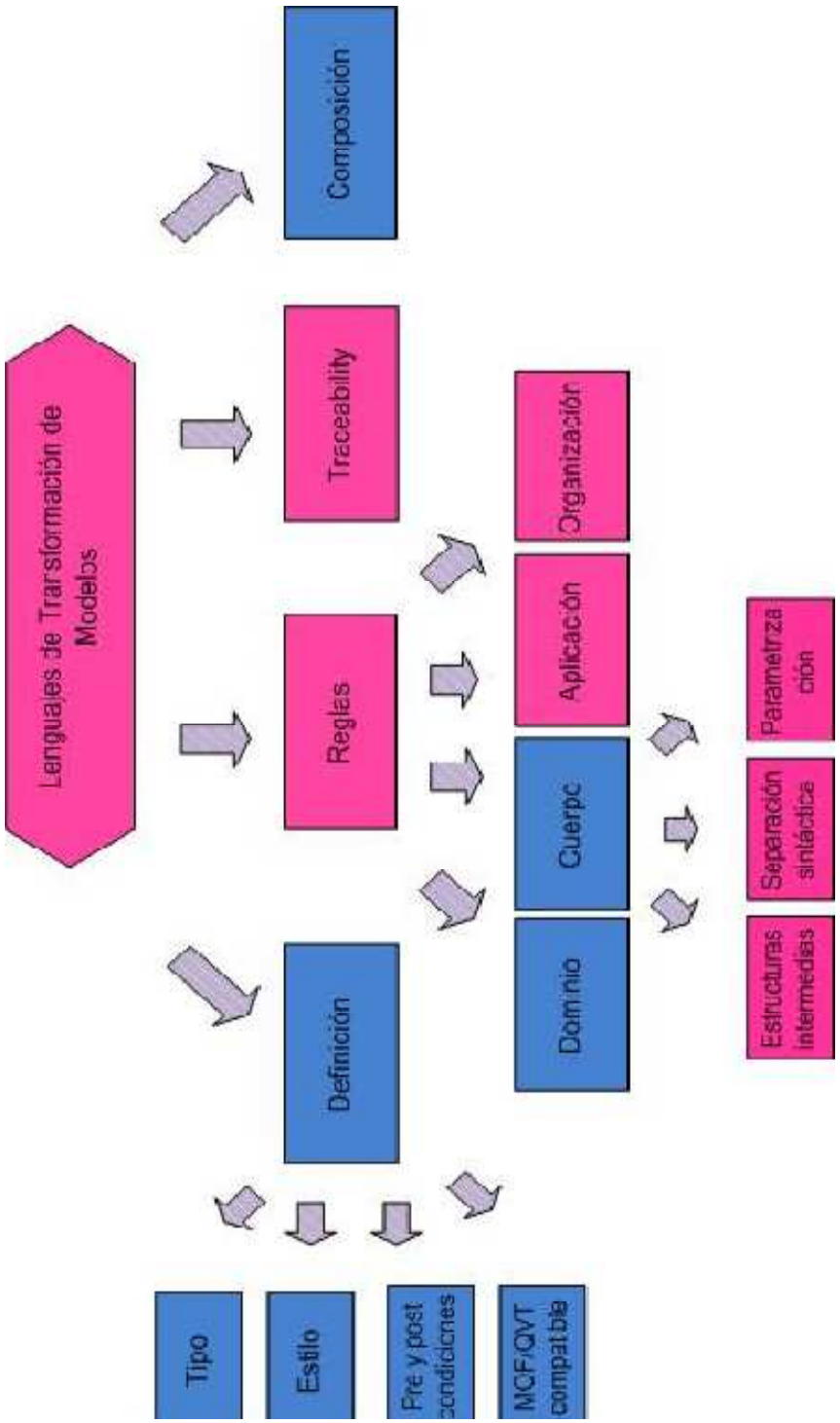


Figura AIII.1. Esquema de Clasificación de Lenguajes de Modelado según nuestra propuesta

Detalle de la Clasificación

A continuación se presenta cada una de los aspectos representados en la figura 1, indicando a qué hace referencia cada uno.

Definición

La definición de una transformación determina características generales de la transformación. Se enuncian a continuación:

- **Tipo**: indica si la sintaxis del lenguaje es icónica (gráfica) o textual y si es modelo a modelo (M2M) o modelo a texto (M2Text).
- **Estilo:** un lenguaje puede ser declarativo, operacional o ambos
- **Pre y post condiciones:** si es posible especificarlos en la transformación y de qué forma puede hacerse (coloquial, con OCL, otro lenguaje). Lo deseable es que se puedan especificar y en lo posible, en OCL ya que es un lenguaje formal y estándar.
- **MOF/QVT compatible:** indica si el lenguaje se basa en los estándares definidos por OMG: si los metamodelos (lenguajes de dominio y codominio) que participan en la transformación son instancias de MOF y si el lenguaje de transformación se basa en QVT. Lo esperado es que los lenguajes sean compatibles con ambas propuestas.

Reglas de Transformación

Las reglas de transformación son unidades más pequeñas que componen una transformación. Como fue mencionado anteriormente, estas describen cómo un elemento del modelo de entrada puede ser transformado en uno o más elementos del modelo de salida.

Los aspectos a analizar son:

- **Dominio**: define uno o más modelos "de entrada" u origen o *source* y uno o más modelos "de salida" o destino o *target* sobre los cuales operarán las reglas de transformación
    - **Lenguajes del dominio**: cada dominio tiene un lenguaje asociado. Se espera que los lenguajes sean instancias de MOF.
    - **Dirección**: indica si los metamodelos source y target son *in*, *out*, o *in/out* (en el sentido en que se usan los parámetros en programación). Y si es unidireccional o bidireccional el sentido de la transformación
    - **Relación entre origen y destino de la transformación**: Si el source y el destino tienen el mismo o diferente modelo.

- **Cuerpo de la transformación**: en cuanto al cuerpo de la transformación, podemos decir que consta de:
    - Declaración de (meta)variables y sus tipos
    - Patrones de transformaciones
    - **Separación sintáctica**: algunos lenguajes separan claramente las partes de una regla de transformación que operan sobre el modelo source

(LHS) de las partes que operan sobre el modelo target (RHS). La separación sintáctica hace a la legibilidad del lenguaje.
    - **Estructuras intermedias:** si el lenguaje recurre a alguna estructura adicional para describir la transformación y que no es parte del modelo a transformar.
    - **Parametrización**: el tipo más simple de parametrización es el uso de los parámetros del control que permiten el paso de valores.

- **Aplicación de las reglas**: como hemos mencionado, una regla se aplica a algún elemento del modelo de entrada. Como puede haber más de una regla que "machee" con un elemento particular, se debe definir alguna estrategia para determinar el orden de aplicación de las reglas.
    - **Orden:** la aplicación de las reglas puede ser determinístico o no determinístico.
    - **Aplicación condicional**: en algunas reglas de transformación se puede tener aplicación condicional de las mismas. En estos casos, existe una condición que debe ser verdadera para que la regla se ejecute.
    - **Iteración de reglas**: si el lenguaje provee estructuras de iteración, o mecanismos de recursión.

- **Organización de las reglas**: se refiere a la composición y estructuración de múltiples reglas. Se puede dividir en varios aspectos:
    - **Modularización:** en el caso en que se provea modularización de las reglas (agrupar un conjunto de reglas en un módulo, que puede ser llamado desde otros módulos).
    - **Mecanismos de reuso:** la definición de regla/s en base a otra/s.

Traceability

Hace referencia a aquellos mecanismos provistos por los lenguajes que permiten guardar ciertos aspectos de la ejecución de la transformación. Se pueden crear y mantener "conexiones" o *links* entre elementos del dominio y del codominio que mapean los dominios source y target, cada vez que una regla de transformación es ejecutada.

Composición

La composición indica cómo pueden relacionarse un número de transformaciones para obtener una nueva. Pueden encadenarse dos o más transformaciones consecutivamente; pueden componerse dos o más transformaciones existentes en una nueva transformación con sus nuevas relaciones o pueden combinarse dos transformaciones de forma tal que se obtienen codominios más amplios.