Instituto de Computación – Facultad de Ingeniería – Universidad de la República

Test Case Tool

Proyecto de grado

Autores: Federico García y Julio Stirling

Tutor: Beatriz Pérez
Cotutor: Mónica Wodzislawski

Resumen

La técnica de Casos de Uso es una de las más utilizadas en la especificación de requerimientos en los proyectos de software. En general los casos de uso se describen usando lenguaje natural, por lo tanto desde el punto de vista de *testing* es muy difícil su automatización.

El objetivo de este proyecto es formalizar la descripción de los casos de uso para generar casos de prueba en forma automática.

El resultado es una herramienta implementada como un *plug-in* de Eclipse que permite modelar los casos de uso en forma estructurada y generar a partir de éstos, modelos de prueba y casos de prueba ejecutables.

Durante la primera etapa del proyecto se investigaron y estudiaron meta-modelos de casos de uso propuestos por varios autores. El resultado de esta etapa fue la creación de un meta-modelo de casos de uso donde se incorporaron conceptos relativos al *testing*.

La segunda etapa consistió en la generación de casos de prueba a partir del meta-modelo de casos de uso construido en la etapa previa. Se investigaron técnicas de generación de casos de prueba a partir de casos de uso y se crearon algoritmos para hacer la transformación. Se utilizó el meta-modelo UML *Testing Profile* para modelar los casos de prueba resultantes.

En la tercera etapa del proyecto se investigó la plataforma de Eclipse y se construyó un *plug-in* que permite ingresar los casos de uso de forma estructurada y obtener como resultado casos de prueba ejecutables.

El trabajo se enmarca dentro de la necesidad de contar con una herramienta para el *testing* funcional basado en casos de uso, de tal forma que sea posible automatizar, tanto la creación de pruebas como los valores de prueba a ser usados en las mismas.

Tabla de Contenidos

Capítulo 1 Introducción....8
1.1 Motivación....8
1.2 Objetivos del proyecto....8
1.3 Resultados esperados....8
1.4 Organización del documento....9
Capítulo 2 Estado del arte....11
2.1 Casos de uso....12
2.2 Meta-modelo UML para casos de uso....14
2.2.1 Evaluación de la propuesta....16
2.3 Meta-modelo de casos de uso según Siqueira y Muniz....17
2.3.1 Evaluación de la propuesta....20
2.4 Meta-modelo de casos de uso según Somé....21
2.4.1 Evaluación de la propuesta....25
2.5 Meta-modelo de casos de uso según Williams....26
2.5.1 Evaluación de la propuesta....30
2.6 Diagramas de actividad a partir de casos de uso....31
2.7 Casos de prueba a partir de diagramas de actividad....33
2.7.1 Caso de prueba....33
2.7.2 Creación de casos de prueba....33
2.8 UML Testing Profile (UTP)....35
2.9 Generación de valores de prueba....37
2.9.1 Black-box testing....37
2.9.2 Clases de equivalencia....37
2.9.3 Valores límite....37
Capítulo 3 Contribuciones del proyecto....38
3.1 Meta-modelo de casos de uso para TestCaseTool....38
3.1.1 Descripción del meta-modelo....39
3.1.2 Conceptos vinculados al testing....44
3.2 Generación de diagramas de actividad a partir de casos de uso....48
3.2.1 Nuestra propuesta....48
3.3 Generación de casos de prueba a partir de diagramas de actividad....52
3.3.1 Algoritmo propuesto....52
3.3.2 Solución propuesta y UTP....55
3.3.3 Generación de valores de prueba....56

3.3.3.1 Técnica utilizada....56
Tipo de dato numérico....56
Tipo de dato string....57
Tipo de dato enumerado....58
Capítulo 4 Caso de estudio....59
4.1 Caso de Uso: Retiro de dinero....61
4.1.1 Diagrama de actividad....62
4.1.2 Casos de prueba....63
4.2 Caso de Uso: Validar usuario....66
4.2.1 Diagrama de Actividad....67
4.2.2 Casos de prueba....68
4.3 Caso de Uso: Cambio de PIN....70
4.3.1 Diagrama de Actividad....71
4.3.2 Casos de Prueba....72
4.4 Caso de Uso: Consulta de saldo....74
4.4.1 Diagrama de Actividad....75
4.4.2 Casos de Pruebas....75
Capítulo 5 Implementación....77
5.1 Eclipse....77
5.1.1 Arquitectura de plug-ins....79
5.1.2 La plataforma UI....79
5.1.3 Tecnologías utilizadas en la implementación....80
5.1.3.1 EMF....80
5.1.3.2 EEF....81
5.1.3.3 Papyrus....81
5.2 Desarrollo de la herramienta TestCaseTool....81
5.3 Interfaz gráfica de la herramienta....86
Capítulo 6 Conclusiones....91
6.1 Resultados obtenidos....91
6.2 Conclusiones....92
6.3 Trabajos a futuro....93
Referencias....94
Glosario....95
Anexos....97
Anexo 1 Visualización de casos de uso a través de diagramas de actividad....97
1.1 Introducción....97
1.2 Meta-modelos de Requerimientos Funcionales y de Diagramas de Actividad....98

1.3 El meta-modelo de Diagramas de Actividad de UML....99
1.4 Transformaciones de Requerimientos Funcionales textuales a Diagramas de Actividad. 100
Anexo 2 Un enfoque para generación de pruebas en MDT....103
2.1 Estado del arte....103
2.2 El trabajo propuesto....103
2.3 Solución concreta....105
Anexo 3 Un enfoque nuevo para generar casos de prueba a partir de diagramas de actividad....109
Anexo 4 Manual de Usuario....114

Índice de Figuras

Figura 1: Meta-modelo de casos de uso en UML....14
Figura 2: Meta-modelo de casos de uso propuesto por Siquiera y Muniz....18
Figura 3: Extensión al meta-modelo de casos de uso....20
Figura 4: Variantes para la descripción del caso de uso....22
Figura 5: Descripción de la meta-clase NormalDescription....23
Figura 6: Tipos de pasos en el caso de uso....24
Figura 7: Relación entre ActionStep y Alternative....25
Figura 8: Meta-modelo de caso de uso de Williams....28
Figura 9: Flujos de casos de uso y actores....28
Figura 10: Parámetros, particiones y valores....29
Figura 11: Meta-modelo UML Testing Profile....36
Figura 12: Concepto de UseCaseModel....39
Figura 13: Concepto de UseCase....40
Figura 14: Flujos de ejecución del caso de uso....41
Figura 15: Diferentes tipos de pasos....42
Figura 16: Parámetros, restricciones y valores de prueba....44
Figura 17: Meta-modelo de Casos de Uso de TestCaseTool....47
Figura 18: Meta-modelo de Diagrama de Actividad....49
Figura 19: Ejemplos de finalización de flujos alternativos....51
Figura 20: Diagrama de clases de la salida del algoritmo que genera los escenarios de prueba....54
Figura 21: Meta-modelo de Casos de Prueba y su relación con UTP....55
Figura 22: Diagrama de casos de uso UML....59
Figura 23: Diagrama de actividad para Retiro de dinero....62
Figura 24: Diagrama de actividad para Validar usuario....67
Figura 25: Diagrama de actividad generado para Cambio de PIN....71
Figura 26: Diagrama de actividad generado para Consulta de saldo....75

Figura 27: Plataforma de Eclipse....78
Figura 28: Meta-modelo de casos de uso para implementación....83
Figura 29: Diagrama de componentes de la herramienta....85
Figura 30: Componentes de la interfaz de usuario....86
Figura 31: Vista Project Explorer....87
Figura 32: Vista Outline....87
Figura 33: Sección "Use Cases" del editor....88
Figura 34: Sección "Flows" del editor....89
Figura 35: Casos de prueba para Retiro de dinero....90
Figura 36: Meta-modelo de requerimientos funcionales....98
Figura 37: Meta-modelo de Diagramas de Actividad....100
Figura 38: Generación de casos de prueba siguiendo el paradigma de MDE....104
Figura 39: Meta-modelo de requerimientos funcionales....105
Figura 40: Requerimiento funcional estructurado....106
Figura 41: Diagrama de Actividad generado....107
Figura 42: Escenarios de ejecución generados....107
Figura 43: Representación tabular del Caso de Prueba....108
Figura 44: Diagrama de Actividad de ejemplo....109
Figura 45: Reglas de mapeo de conceptos....110
Figura 46: Grafo de Actividad generado....111
Figura 47: Caso de Prueba en forma de Tabla....113

Índice de Tablas

Tabla 1: Pseudo-código del algoritmo para generar diagramas de actividad....50
Tabla 2: Pseudo-código para la generación de casos de pruebas....53
Tabla 3: Generación de valores límite para enteros....57
Tabla 4: Generación de valores para cadenas de caracteres....57
Tabla 5: Generación de Valores para enumerados....58
Tabla 6: Descripción del caso de uso "Retiro de dinero"....61
Tabla 7: Caso de prueba generado para Retiro de dinero....65
Tabla 8: Descripción del caso de uso “Validar usuario”....66
Tabla 9: Caso de prueba generado para Validar usuario....69
Tabla 10: Descripción del caso de uso “Cambio de PIN”....70
Tabla 11: Caso de prueba generado para Cambio de PIN....73
Tabla 12: Descripción del caso de uso “Consulta de saldo”....74
Tabla 13: Caso de prueba generado para Consulta de saldo....76

Capítulo 1 Introducción

1.1 Motivación

La técnica de Casos de Uso es una de las más utilizadas en la especificación de requerimientos funcionales en los sistemas de software.

La principal motivación del proyecto es la construcción de una herramienta que permita automatizar parte de las tareas en el proceso de *testing* de un sistema de software especificado mediante casos de uso.

1.2 Objetivos del proyecto

Uno de los objetivos del proyecto es la búsqueda de una representación estructurada de casos de uso a partir de la cual sea posible generar casos de prueba en forma automática.

El problema que se intenta abordar tiene como dificultad que no existe un consenso sobre como deben ser escritos los casos de uso y menos aún el nivel de detalle empleado en su escritura.

En la búsqueda de una estructura para los casos de uso es necesario crear un meta-modelo que represente los conceptos y relaciones que forman parte de ellos. El meta-modelo debe incluir información de *testing* ya que será usado como base para aplicar transformaciones que deriven en la creación automática de pruebas.

Una vez obtenido el meta-modelo, el siguiente objetivo es desarrollar una técnica para crear casos de prueba a partir de la representación de los casos de uso. Para modelar los casos de prueba se usará el meta-modelo UML *Testing Profile*.

Por último se construirá un *plug-in* de Eclipse donde se implementan los objetivos anteriores y será posible ingresar casos de uso en forma estructurada y obtener casos de prueba ejecutables a partir de ellos.

1.3 Resultados esperados

Se espera contar con una herramienta que permita generar casos de prueba desde las primeras etapas del desarrollo de software, en particular se espera que la herramienta permita automatizar las siguientes tareas:

- Descripción de casos de uso, estructurando la información representada en los mismos.

- Obtención de un modelo de prueba a partir de los casos de uso, el cuál utilizará como meta-modelo UML *Testing Profile*.
- Obtención de los datos de prueba a partir de la información almacenada en los casos de uso utilizando las técnicas de *testing* de partición de equivalencia y valor límite.
- Obtención de casos de prueba ejecutables a partir de la información de los puntos anteriores.

1.4 Organización del documento

El informe consta de seis capítulos cuyo contenido se esboza a continuación.

El capítulo 1 es una introducción inicial al problema donde se plantean los objetivos y resultados esperados del proyecto.

El capítulo 2 es un estudio del estado del arte donde se presentan los casos de uso y un conjunto de meta-modelos para representarlos. Se estudian técnicas para generar casos de prueba a partir de casos de uso, se introduce *UML Testing Profile* como meta-modelo de pruebas y se exhiben técnicas de testing para la generación de valores de prueba.

El capítulo 3 contiene las contribuciones del proyecto donde se construye un meta-modelo de casos de uso con información de testing y se desarrollan algoritmos para generar casos de prueba a partir de este meta-modelo. Se desarrolla una técnica para generar valores de prueba en forma automática a partir de una descripción textual.

El capítulo 4 consiste en un caso de estudio en el que se plantean casos de uso y se muestran las transformaciones necesarias hasta obtener casos de prueba a partir de ellos.

En el capítulo 5 se introduce Eclipse como plataforma de desarrollo y se describe la implementación de la herramienta, se detallan los principales componentes construidos y como estos se relacionan.

El capítulo 6 contiene las conclusiones del trabajo, los resultados obtenidos y las posibilidades de trabajos futuros a partir de los resultados alcanzados en este proyecto.

El documento también cuenta con 4 anexos, los cuales se describen a continuación.

El anexo 1 es un resumen sobre visualización de casos de uso a través de diagramas de actividad donde se estudia un algoritmo para generar diagramas de actividad a partir de los requerimientos funcionales de un sistema.

El anexo 2 es un resumen sobre generación de casos de prueba a partir de requerimientos funcionales del sistema aplicando la metodología *Model Based Testing*. En el estudio se considera la generación de pruebas a partir de transformaciones automáticas entre modelos.

El anexo 3 es un resumen de una metodología para generar casos de prueba a partir de diagramas de actividad de UML.

Los anexos anteriores extienden el capítulo 2 de estado del arte con información sobre la generación de diagramas de actividad y casos de prueba a partir de casos de uso de un sistema.

El anexo 4 es un resumen de los contenidos del manual de usuario creado para la herramienta y se incluye una referencia al repositorio del proyecto donde se encuentra.

Capítulo 2 Estado del arte

El estado del arte comienza con una introducción a los casos de uso desde la perspectiva de Alistair Cockburn, uno de los principales impulsores de la aplicación de casos de uso como herramienta para documentar procesos de negocios y requerimientos de software [Coc00].

Luego se estudian los últimos trabajos sobre meta-modelos de casos de uso que existen en la literatura y que son relevantes para nuestro proyecto. En ellos se verán los meta-modelos propuestos y las principales dificultades que se encuentran en su construcción.

Posteriormente se presentan trabajos que utilizan diagramas de actividad de UML para generar casos de prueba a partir del meta-modelo de casos de uso.

Finalmente se estudia el meta-modelo de casos de prueba UML Testing Profile y se describen algunos conceptos de testing que son claves para nuestro proyecto.

2.1 Casos de uso

El contenido de esta sección es una síntesis extraída del libro “Writing Effective Use Cases” escrito por Alistair Cockburn [Coc00].

Un caso de uso puede ser considerado simplemente como una forma de escribir y que puede ser usado en diferentes situaciones. Por ejemplo, para describir un proceso de negocio, para servir como base en la discusión sobre los requerimientos que debería tener un sistema de software, para describir los requerimientos funcionales de un sistema, o para documentar un diseño.

En cada una de estas situaciones, se podrían utilizar diferentes estilos y formatos, donde el hilo común es describir actores en la búsqueda de algún objetivo. Los casos de uso son una descripción de secuencias de interacciones entre un sistema y actores externos que persiguen un objetivo en particular.

Esta primera definición plantea la necesidad de profundizar sobre los significados de “descripción”, “secuencias de interacciones”, “sistema”, “actor” y “objetivo”. La definición no dice nada acerca de cuando son escritos los casos de uso, cual es el formato utilizado o cual es el tamaño del sistema.

Los casos de uso, a través de la técnica de escritura empleada, son útiles para describir el comportamiento de cualquier sistema que interactúa con un ambiente externo.

Algunas aplicaciones de la técnica de casos de uso donde se han obtenido buenos resultados son:

- Para capturar el diseño de procesos de negocio en compañías y organizaciones. En esos casos dichos procesos han sido documentados o rediseñados.
- Para documentar el comportamiento de sistemas de software.
- Para especificar requerimientos de sistemas de software en situaciones donde los casos de uso han sido creados antes de la etapa de diseño del sistema.
- Para documentar del comportamiento de subsistemas internos.

En todas estas aplicaciones se muestra como el sistema responde al mundo exterior, las responsabilidades y el comportamiento del sistema, sin revelar como las partes internas son construidas. En todas estas situaciones, se desea escribir lo menos posible, lo más detallado posible, mostrando las diversas formas en que el sistema reacciona.

La dificultad en la elaboración de un caso de uso radica en que no existen reglas sobre la estructura que debe tener para cada situación o sistema particular. El nivel de detalle se deberá considerar en cada situación de manera que se adecue a lo que se desea describir.

Los casos de uso se expresan fundamentalmente a través de texto, aunque también pueden ser escritos mediante redes de Petri, diagramas de flujo, diagramas de secuencia o pseudo-código.

El principal propósito es la comunicación entre una persona y otra. A menudo ambas personas no tienen un entrenamiento especial en informática y entonces, la especificación textual es la mejor opción para describirlos.

Los actores que participan en un caso de uso pueden ser tanto personas como sistemas de software. Entre ellos existe uno en particular que es llamado principal o primario. El caso de uso representa los pasos que debe seguir dicho actor para alcanzar un objetivo.

Los casos de uso describen la interacción entre actores y el sistema así como las responsabilidades del sistema que se está diseñando, sin considerar técnicas de implementación.

Las secuencias de interacciones son llamadas escenarios o flujos. El caso de uso agrupa todos los flujos relacionados a la meta del actor principal, incluyendo aquellos donde el objetivo es alcanzado y aquellos donde no es posible hacerlo.

Se define precisión como la cantidad de detalle a expresar relacionado con el momento en que es expresado. En ciertas circunstancias un nivel de detalle bajo puede ser suficiente y en otras no.

La precisión empleada para describir los casos de uso es importante y dependerá de la finalidad que se persigue. Como se vio anteriormente, los casos caso de uso pueden ser utilizados para expresar un comportamiento y ser un medio de comunicación entre personas. En tal caso, un nivel de detalle bajo puede ser suficiente.

2.2 Meta-modelo UML para casos de uso

La especificación de UML[1] describe los casos de uso como una forma de detallar los usos de un sistema. Típicamente, son usados para capturar los requerimientos de un sistema, esto es, aquello que el sistema debe realizar.

Los conceptos clave asociados con los casos de uso son actores y sistema. El sistema que se está modelando es el sujeto sobre el cual el caso de uso aplica. Los actores representan a los usuarios y a cualquier otro sistema que pueda interactuar con el sujeto, éstos siempre representan entidades que están fuera de los límites del sistema. Los casos de uso especifican las funcionalidades o comportamiento que brinda el sistema y son definidos de acuerdo a las necesidades de los actores.

En la Figura 1 se presentan los conceptos del meta-modelo de casos de uso propuesto por UML.

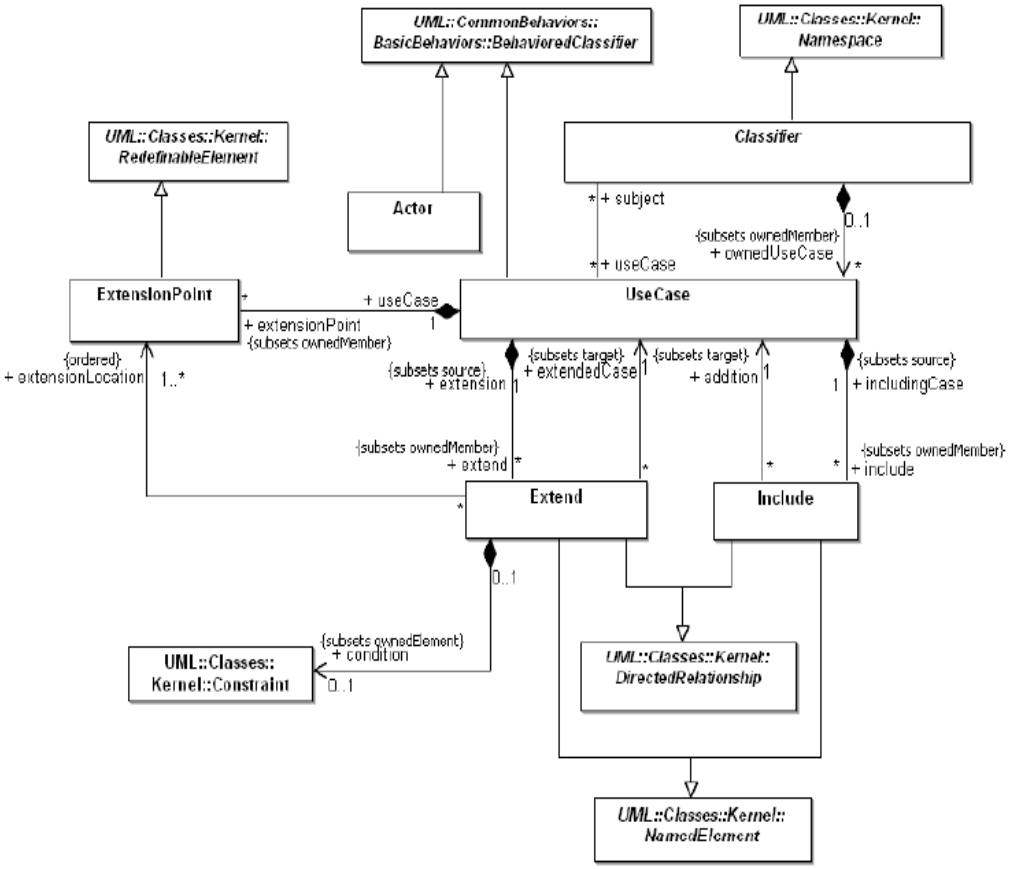

Figura 1: Meta-modelo de casos de uso en UML

1 Unified Modeling Language UML 2.4.1 Specification http://www.omg.org/spec/UML/2.4.1/

El concepto *Actor* especifica el rol interpretado por un usuario o cualquier otro sistema que interactúa con el sujeto. El término rol se utiliza de manera informal para indicar el papel que juega un usuario u otro sistema. En el meta-modelo de casos de uso de UML un actor no tiene ninguna asociación con otros conceptos del meta-modelo.

El concepto *Classifier* tiene la propiedad de ser dueño de casos de uso. Un mismo caso de uso puede ser usado en diferentes situaciones para distintos sujetos, estos son identificados con el rol de *subject* en la asociación con el caso de uso. Por ejemplo el caso de uso de "Login" puede ser utilizado tanto para un sistema Web, como para una aplicación de escritorio.

El concepto *Extend* representa una relación que especifica que parte del comportamiento de un caso de uso puede ser extendido por el comportamiento de otro. La extensión se produce en uno o más puntos definidos en el caso de uso.

Los puntos de extensión *extensionPoint* son una marca dentro del comportamiento de un caso de uso que señala donde dicho comportamiento puede ser extendido. El caso de uso que declara puntos de extensión, es definido independientemente de los casos de uso que lo extienden. Por otro lado el caso de uso que extiende, puede no ser de utilidad por si mismo fuera de la relación de extensión.

La asociación *condition* hace referencia a una condición que se debe cumplir para que la extensión se produzca. En el caso de no existir una restricción asociada, la extensión es incondicional.

El lugar donde el comportamiento del caso de uso es extendido se representa por una lista ordenada de puntos de extensión. La relación *extensionLocation* fue sub-especificada intencionalmente debido a que los casos de uso son descritos con diversos formatos, entre ellos en lenguaje natural y por lo tanto no es fácil capturar su estructura en un modelo formal.

El concepto de *Include* es una relación que permite incluir un caso de uso dentro de otro. El caso de uso que incluye a otro puede depender solo del resultado del caso de uso incluido. Este resultado puede ser un valor obtenido en la ejecución del caso de uso incluido. Se debe notar que el caso de uso que es incluido, no es opcional y siempre es requerido para una correcta ejecución del caso de uso que lo incluye.

El concepto *Use Case* es uno de los más importantes del meta-modelo. Representa la idea de un conjunto de acciones llevadas a cabo por un sistema, las cuales derivan en un resultado observable y de valor para uno o más actores. Este concepto define el comportamiento ofrecido por un sujeto sin tener conocimiento de su estructura interna. El comportamiento consiste en interacciones entre el actor y el sujeto, las cuales pueden resultar en cambios en el estado del sistema o comunicaciones con el ambiente.

El caso de uso puede incluir variantes al comportamiento básico como son situaciones de excepciones o manejo de errores.

El sujeto de un caso de uso puede ser un sistema o cualquier otro elemento que tenga comportamiento, por ejemplo, un componente, subsistema o clase.

Cada caso de uso especifica una funcionalidad que el sujeto provee a sus usuarios y describe un modo específico de interactuar con el sujeto.

Los actores inician una funcionalidad solicitándola y ésta siempre debe ser completada para que el caso de uso esté completo. Se considera que el caso de uso está completo, si luego de su ejecución, el sujeto queda en un estado en el cual no hay más entradas o acciones esperadas y el caso de uso puede ser iniciado nuevamente.

Los casos de uso pueden ser usados para especificar los requerimientos sobre el sujeto así como la funcionalidad ofrecida por éste.

El comportamiento presente en los casos de usos puede ser descrito mediante diversas especificaciones como son, interacciones, actividades, máquinas de estado, pre-condiciones y pos-condiciones, así como por texto en lenguaje natural cuando es apropiado. La técnica elegida dependerá de la naturaleza del comportamiento del caso de uso, así como de la audiencia a quien esté dirigido.

Los sujetos para los cuales el caso de uso aplica se representan con la asociación *subject*. Los casos de uso que son incluidos en otros se expresan a través de la asociación *include*.

La asociación *extend* referencia los casos de uso cuyo comportamiento es extendido, mientras que, la asociación *extensionPoint* define puntos de extensión que presenta el caso de uso y que pueden ser usados para extender su comportamiento.

Finalmente, los casos de uso pueden tener actores asociados y esta asociación describe como el actor participa en la interacción con el sujeto.

2.2.1 Evaluación de la propuesta

El meta-modelo de casos de uso de UML tiene como objetivo fundamental representar las relaciones que existen entre casos de uso y no considera la interacción de los actores y el sistema.

Desde el punto de vista del testing, la falta de una representación del comportamiento tanto de actores como del sistema, hace que el meta-modelo propuesto por UML no sea de gran utilidad. Para representar el comportamiento en los casos de uso, UML propone utilizar otros modelos como son diagramas de secuencia o diagramas de actividad.

2.3 Meta-modelo de casos de uso según Siqueira y Muniz

En esta sección se describe el trabajo de Fábio Levy Siqueira y Paulo Sérgio Muniz Silva donde se propone un meta-modelo que representa la descripción textual de casos de uso [SM11].
El meta-modelo contiene conceptos encontrados en el análisis de veinte estudios.

El objetivo que se persigue es una representación genérica y extensible de casos de uso que permita utilizarlos en actividades de "*Model Driven Engineering*" (MDE).

Los casos de uso son descriptos en la especificación de UML, por lo tanto, una posibilidad sería utilizar el meta-modelo propuesto por ese estándar. El principal problema es que UML permite varias opciones en cuanto a la forma de representar el comportamiento en los casos de uso, entre ellas, diagramas de actividad y diagramas de secuencia. A su vez, en UML no se especifica una representación textual para casos de uso, la cual, según varios autores es la más adecuada.

Los autores plantean que no existe un meta-modelo estándar para la representación textual de casos de uso, aunque sí existen varias propuestas en ese sentido. En general, los meta-modelos están basados en uno o dos formatos de casos de uso y por lo tanto, no contemplan el universo de conceptos que podrían formar parte.

Los autores se basan en una revisión de la literatura para encontrar términos o conceptos que se consideran relevantes, estos son:

- *name*: la identificación del caso de uso.
- *description*: una pequeña descripción de lo que representa el objetivo del caso de uso.
- *actor*: un rol que toma una entidad externa y que interactúa con el sistema.
- *pre-condition*: una condición aplicada al sujeto y que se debe cumplir antes de que el caso de uso comience.
- *post-condition*: una condición aplicada al sujeto y que debe ser cumplida luego de la ejecución del flujo básico (*basic flow*).
- *basic flow*: el flujo de eventos más común o también llamado flujo exitoso.
- *alternative flow*: un flujo de eventos que describe la desviación del caso de uso por otro flujo de eventos diferente al flujo básico.

El concepto de *step* es la unidad de comportamiento que representa la interacción entre un actor y el sistema. El flujo de eventos es un conjunto de pasos (*steps*) ejecutados en forma ordenada por actores o el sistema, que tiene el objetivo de cumplir la meta del caso de uso.

Los flujos alternativos (*alternative flow*) tienen asociados los conceptos de punto de extensión (*extension points*) y condiciones (*conditions*). La idea de *condition* es una expresión que puede ser evaluada como verdadero o falso. El concepto de *extension point* representa el lugar donde existe una desviación del flujo principal tal que dicha desviación depende de cierta condición.

Los autores analizaron los trabajos seleccionados con el fin de obtener que consideraciones se hacían sobre conceptos como *conditions* (condiciones), *loops* (iteraciones), *parallelism (*pasos ejecutados en paralelo) y *ad-hoc blocks* (conjuntos de pasos donde no existe un orden requerido para su ejecución). En el resultado se observó que solo los conceptos de *conditions* y *loops* eran definidos frecuentemente.

Los conceptos identificados en el análisis de las distintas propuestas son representados en la Figura 2 mediante un diagrama de clases de *Meta-Object Facility* (MOF).

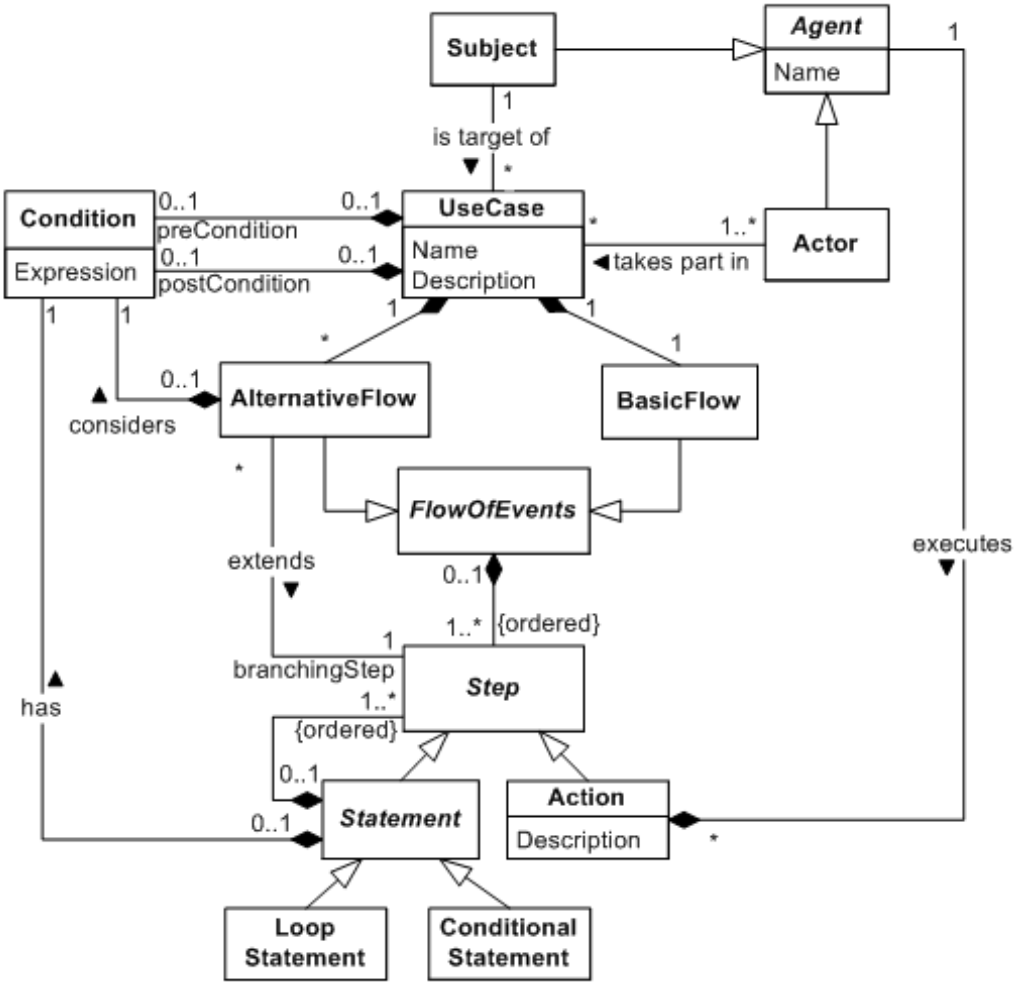

Figura 2: Meta-modelo de casos de uso propuesto por Siquiera y Muniz

El concepto de *conditional statement* especifica una condición que debe cumplirse para ejecutar una secuencia ordenada de pasos.

El concepto de *loop statement* especifica una condición para ejecutar una secuencia ordenada de pasos y que mientras dicha condición se cumpla, se debe ejecutar repetidamente. La idea de *statement* se define como un tipo de paso que tiene asociado una condición y un conjunto ordenado de pasos.

Además de los conceptos identificados, los autores añaden cuatro conceptos nuevos:

- *UseCase*: representa un caso de uso. Los conceptos de *name* y *description* son considerados atributos de esta meta-clase.
- *FlowOfEvents*: sirve para simplificar el meta-modelo y representa a los conceptos de *BasicFlow* y *AlternativeFlow* que tienen una semántica similar y poseen un conjunto ordenado de pasos.
- *Statement*: representa la generalización de *loop statements* y *conditional statements*. Este concepto fue creado con fines extensivos.
- *Action*: creado para representar un paso descrito en forma textual. La descripción textual es incluida con un atributo de la meta-clase.

En un trabajo posterior, los autores proponen una extensión al meta-modelo, donde se agregan tres conceptos nuevos.[2]

Las meta-clases incluidas son: *UseCaseModel*, *Agent* y *Subject*. La relación entre estos conceptos se pueden apreciar en la Figura 3. Los conceptos de *Agent* y *Subject* representan quien ejecuta una acción dentro de la secuencia de pasos del caso de uso. La noción de agente es una generalización de sujeto y actor. La meta-clase *UseCaseModel* permite la existencia de varios casos de uso en el modelo.

2 Essential Use Case Meta-model http://www.levysiqueira.com.br/2011/11/essential-use-case-mm/

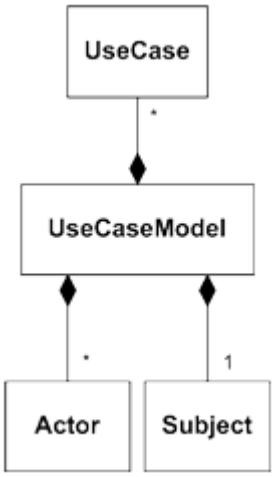

Figura 3: Extensión al meta-modelo de casos de uso

2.3.1 Evaluación de la propuesta

Los autores debieron decidir si el meta-modelo cumpliría con la especificación de casos de uso que existe en UML. Una opción natural sería extender el meta-modelo de casos de uso propuesto en UML, sin embargo, el único mecanismo de extensión consiste en el uso de *profiles* (perfiles). El uso de perfiles hace que sea necesario agregar restricciones al meta-modelo para evitar contradicciones con la semántica estándar de UML. Además, sería necesario agregar los conceptos de inclusión y extensión, así como definir la semántica de las acciones dentro de un flujo de eventos.

El meta-modelo intenta reflejar los principales conceptos y relaciones que existen en los casos de uso y por lo tanto, se considera que no es necesario agregar la complejidad que supone el uso de perfiles de UML. En consecuencia, el meta-modelo propuesto no sigue el estándar de UML.

Los autores plantean la posibilidad de agregar información al meta-modelo ya que este solo considera los conceptos más frecuentes que aparecen en la descripción de casos de uso. De ser necesario, puede ser extendido con nuevas meta-clases y atributos, no existiendo restricciones para que la extensión pueda ser efectuada.

En la selección de los trabajos que fueron analizados se tomaron algunas decisiones arbitrarias como fueron la base de datos de búsqueda, las palabras clave utilizadas y el número de libros considerados. Como resultado, algunas propuestas no fueron incluidas. Entre ellas se encuentra el trabajo de Somé[Som08]. Los autores utilizan el trabajo de Somé para presentar la complejidad que supone la creación de un meta-modelo que busca cumplir con el estándar UML.

2.4 Meta-modelo de casos de uso según Somé

En esta sección se describe el trabajo de Stéphane S. Somé donde hace un análisis de varias propuestas para la representación textual de casos de uso y propone una extensión del meta-modelo de casos de uso de UML [Som08]. El meta-modelo resultante está basado en elementos comúnmente encontrados en plantillas usadas para describir casos de uso.
El autor escoge dos representaciones que a su criterio, son aquellas que han tenido más aceptación. Por un lado la plantilla propuesta por *Rationale Unified Process*[3] y por otro, la plantilla propuesta por Alistair Cockburn conocida como "*fully-dressed*" [Coc00].

El enfoque seguido por el autor para la construcción del meta-modelo consiste en extender el meta-modelo de casos de uso propuesto por UML incluyendo conceptos que surgen al describir los casos de uso en forma textual. Se entiende por descripción textual aquella que se expresa mediante texto y utiliza lenguaje natural.

El autor plantea que los beneficios que surgen al contar con un meta-modelo son, por un lado, el potencial para automatizar la edición de casos de uso y por otro, la posibilidad de generar otros modelos de comportamiento tales como diagramas de actividad o diagramas de estado.

Las dos representaciones de casos de uso consideradas tienen en común que es posible distinguir dos partes fundamentales. Una parte estática y una parte dinámica.

La parte estática incluye elementos relativos al estado del sistema así como elementos con propiedades descriptivas. Algunos de los componentes que representan el estado del sistema son las pre-condiciones y pos-condiciones. Entre los elementos descriptivos se encuentran los actores, la descripción del caso de uso y la prioridad, entre otras.

La parte dinámica captura el comportamiento del caso de uso. Esta consiste en un disparador, una secuencia principal de pasos y posiblemente una o varias secuencias alternativas. El disparador es un evento que determina el comienzo de una secuencia de pasos. La ejecución de los pasos que conforman una secuencia es gobernada por diferentes tipos de estructuras de control, entre las que se encuentran:

- Secuencia: cuando un paso sigue a otro sin necesidad de una condición. El orden de la secuencia está implícita en el orden de los pasos que la componen.
- Alternativa: cuando la ejecución de un paso es condicional. Existe una condición que se debe cumplir.
- Iteración: cuando una misma secuencia de pasos se repite en base a una condición.

3 Rational Unified Process http://www-01.ibm.com/software/rational/rup/

Los pasos incluyen acciones que son ejecutadas por actores o el sistema, así como directivas para la la inclusión de casos de uso o ramificaciones.

En la Figura 4, la clase *UseCaseDescription* es utilizada para capturar la descripción textual del caso de uso.

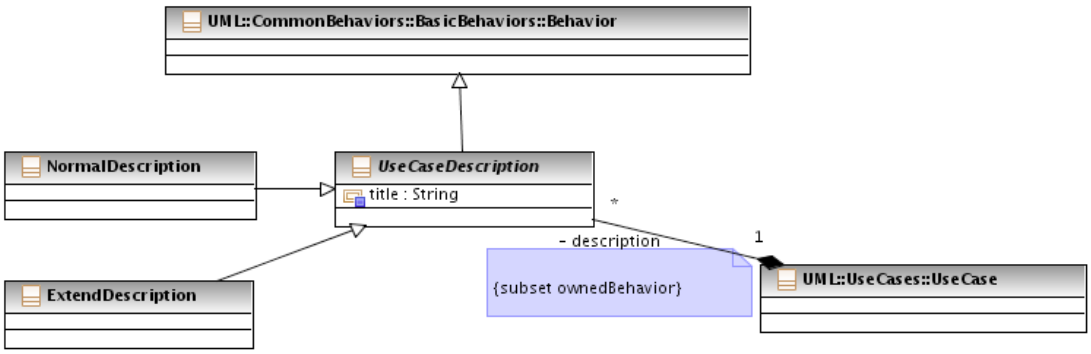

Figura 4: Variantes para la descripción del caso de uso

Se distinguen dos subclases de *UseCaseDescription*: *NormalDescription* y *ExtendDescription*.

El concepto de *NormalDescription* especifica la descripción del caso de uso, mientras que el concepto de *ExtendDescription* es usado para representar la relación de extensión entre casos de uso. La distinción anterior es necesaria debido a que en UML, la extensión de un caso de uso define un conjunto de cambios en el comportamiento bajo ciertas condiciones.

En la Figura 5 se muestra la meta-clase *NormalDescription*, donde se incluyen una parte estática y una dinámica.

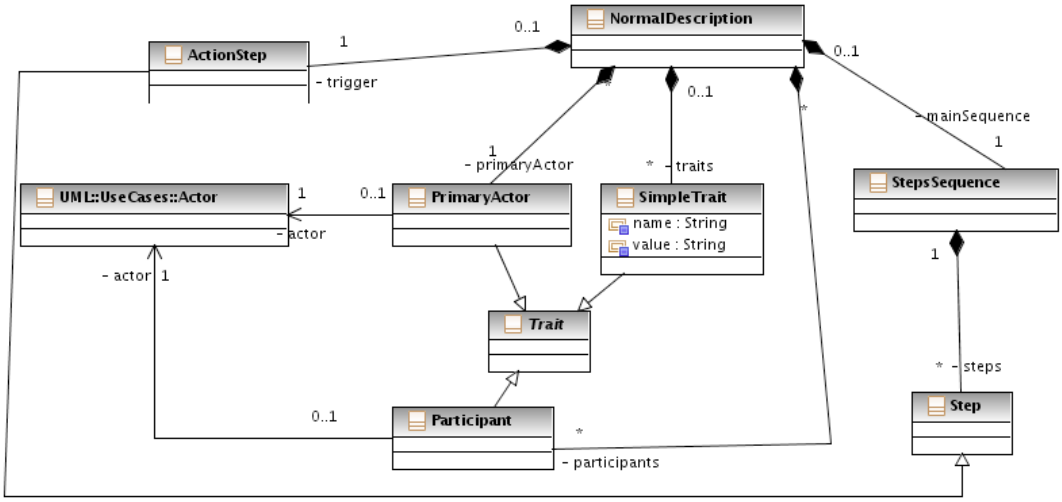

Figura 5: Descripción de la meta-clase NormalDescription

La parte estática contiene información descriptiva del caso de uso y los actores participantes.

La meta-clase *NormalDescription* extiende la meta-clase *Behavior* de UML y por lo tanto hereda las asociaciones *pre-condition* y *post-condition*. Estas dos asociaciones son utilizadas describir el estado del sistema. La asociación *pre-condition* describe el estado requerido del sistema antes de la ejecución del caso de uso. La asociación *post-condition* describe el estado del sistema luego de la ejecución exitosa del caso de uso.

El autor incluye el concepto de *SimpleTrait* (característica) para poder describir un caso de uso con características particulares.

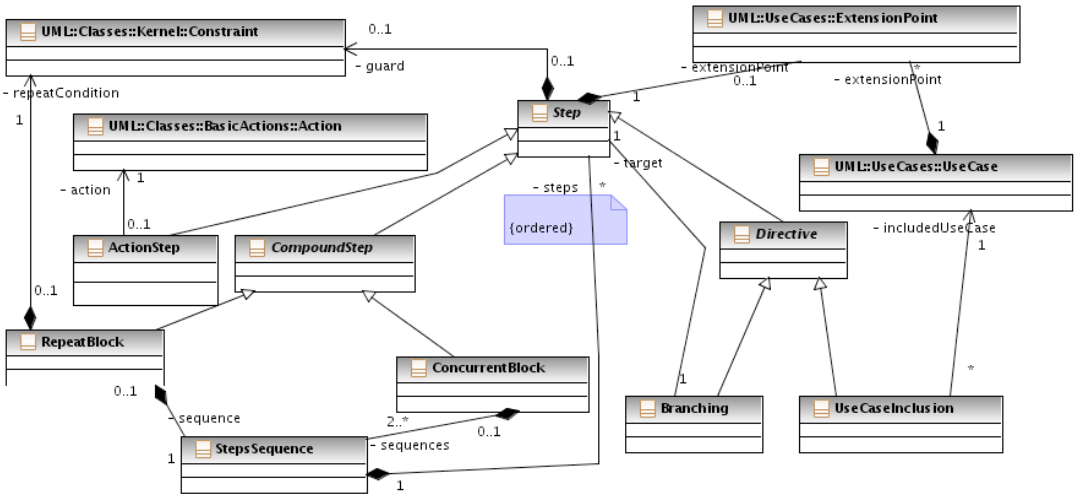

Figura 6: Tipos de pasos en el caso de uso

La parte dinámica incluye un disparador, instancia de la clase *ActionStep,* y una secuencia de pasos *StepsSequence* que representa la secuencia principal del caso de uso.

En la Figura 6 se distinguen diferentes tipos de pasos (*steps*) en el caso de uso.

Los pasos pueden ser condicionados mediante la declaración de una guarda y servir como punto de extensión del caso de uso.

Un paso de tipo *ActionStep* hereda de la clase *Action* de UML. El contexto asociado a la clase *Action* posee el comportamiento del cual la acción forma parte.

Junto a los pasos de tipo acción *ActionStep* se pueden especificar pasos compuestos *CompoundStep* o pasos que son directivas *Directives*.

Las directivas incluyen saltos *Branching*, que son usados para redirigir la secuencia a otros pasos del flujo, e inclusión *UseCaseInclusion*, que expresa la realización de la relación de inclusión entre casos de uso.

Los pasos compuestos *CompoundStep* son una agrupación de pasos, entre ellos se distinguen los conceptos de *RepeatBlock* y *ConcurrentBlock*. Un bloque de repetición *RepeatBlock* es usado para bloques iterativos formados por pasos que deben repetir su ejecución. El bloque de repetición tiene una condición que controla la permanencia en la iteración. El bloque de concurrencia *ConcurrentBlock* es utilizado en pasos que deben ejecutarse en paralelo.

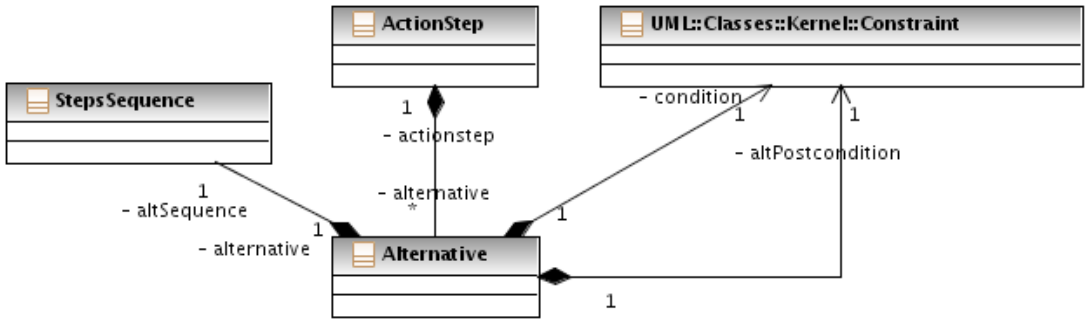

Figura 7: Relación entre ActionStep y Alternative

En la Figura 7 se aprecia el concepto de *Alternative* que especifica una variación del curso de ejecución normal del caso de uso y puede ser asociado a un paso de tipo *ActionStep*.

Las alternativas corresponden a situaciones excepcionales o de error u otras formas de lograr el objetivo propuesto por el caso de uso.

Una alternativa está determinada por una condición. El cumplimiento de esa condición implica la ejecución de una secuencia de pasos *StepsSequence* asociada. El orden de los pasos dentro de la secuencia determina el orden de ejecución de las acciones. Los pasos compuestos y pasos que indican directivas plantean algunas alteraciones a este orden.

2.4.1 Evaluación de la propuesta

La falta de semánticas bien definidas es una de las limitantes de una mayor adopción de los casos de uso en la industria.

El meta-modelo para la descripción textual de casos de uso busca servir como definición de la semántica estática de los casos de uso.

Las restricciones asociadas con el meta-modelo permiten asegurar la consistencia entre casos de uso textuales y el modelo de casos de uso. En el meta-modelo se definen restricciones mediante *Object Constraint Language* (OCL) lo que hace que este cumpla con las restricciones de consistencia definidas en UML.

El hecho de que el meta-modelo construido sea compatible con UML lo hace más complejo y por lo tanto más difícil de entender y utilizar.

2.5 Meta-modelo de casos de uso según Williams

En esta sección se describe el trabajo de Clay E. Williams donde se discuten cuestiones que surgen al utilizar los casos de uso como base para el *testing* basado en modelos [Wil01]. El resultado es un meta-modelo de casos de uso donde se incorporan los conceptos necesarios para crear pruebas del sistema.
En el estudio se mencionan tres propósitos para los cuales los casos de uso son de utilidad:

- Los casos de uso proveen un mecanismo para describir la funcionalidad del sistema de un modo que es entendible por expertos del dominio.
- Los casos de uso proveen un punto de partida a los desarrolladores para entender e implementar la funcionalidad requerida del sistema.
- Los casos de uso sirven como base para probar el sistema a medida que evoluciona.

En el primer caso, los expertos del dominio utilizan los casos de uso para comprobar que el sistema en construcción provee la funcionalidad requerida en forma completa y correcta.

Así como las clases y sus conceptos asociados requieren un meta-modelo bien definido para producir código con ciertas cualidades, los casos de uso necesitan un meta-modelo preciso para producir casos de prueba válidos y robustos.

El autor considera que en el proceso de desarrollo de software, el *testing* es una actividad importante y costosa que típicamente requiere entre el 30% y 50% del presupuesto total del proyecto. Por lo tanto mejoras en la actividad de *testing* pueden conducir a una disminución importante de los costos en la organización.

En la construcción del meta-modelo, es importante identificar que información debe estar presente en los casos de uso, para ser usados en conjunto con técnicas de *testing* basadas en modelos.

Los casos de uso definen o describen la funcionalidad de un sistema, por lo tanto, el primer requerimiento para el meta-modelo es el concepto de sistema. El sistema es el sujeto sobre el cual se quieren ejecutar las pruebas extraídas de los casos de uso.

Se considera un caso de uso como una entidad que describe un conjunto de secuencias de acciones, cada una de las cuales involucra un actor o el sistema. Los casos de uso no mantienen estado, sin embargo, es necesario mantener un registro que indique el lugar actual dentro de la ejecución del caso de uso.

El autor considera un caso de uso como un compuesto de acciones que conforman una secuencia sin atributos persistentes ni estado, aunque sí considera la existencia de un contador que lleva registro de la acción actual.

Las acciones describen la interacción entre actores y el sistema y se distinguen los siguientes tipos:

- Las acciones relativas a actores entre las que se encuentran:
  - Acciones que implican el ingreso de datos al sistema por parte del actor.
  - Acciones que implican el retorno de datos al actor.
  - Acciones de computación en el actor. El actor recibe una respuesta del sistema y calcula nuevas entradas que dependen de la respuesta previa.
- Las acciones relativas al sistema que reflejan cambios en el estado interno como resultado de la interacción con un actor.

El sistema, al cual el caso de uso refiere, es una entidad que está siempre envuelta implícitamente en la acción. En general los actores interactúan con el sistema ejecutando secuencias de casos de uso, estas secuencias pueden considerarse como transacciones realizadas por los actores usando el sistema.

El concepto de *flow* se incorpora al meta-modelo para especificar estas secuencias. Un flujo entre casos de uso define una relación en la que un caso de uso sigue inmediatamente a otro. El autor no considera la existencia de escenarios que agrupen acciones de usuarios o del sistema. De la misma manera, no se maneja la noción de escenarios alternativos como parte de un caso de uso.

El último conjunto de requerimientos refiere a la capacidad de un caso de uso de especificar las entradas que los actores ingresan al sistema así como las salidas que el sistema devuelve. El concepto de parámetro representa los valores de entrada requeridos por un caso de uso. Estos parámetros consisten en particiones lógicas que categorizan los datos de prueba. Por ejemplo, dado un parámetro *password*, las dos particiones lógicas que el parámetro puede tener son válido e inválido. Estas particiones permiten asociar datos concretos a ellas.

El autor sostiene que un meta-modelo que permita realizar actividades de *testing* a partir de casos de uso debe incluir la noción de parámetros, particiones lógicas y valores de prueba.

En la Figura 8 se muestra una porción del meta-modelo donde cada caso de uso consiste en una secuencia ordenada de acciones.

El concepto *Classifier* es el sujeto o sistema, el cual provee un contexto para las acciones del caso de uso. El caso de uso está asociado con *Classifier* para enfatizar que expresa funcionalidad para un sujeto.

Las pre-condiciones y pos-condiciones son incluidas para representar restricciones sobre el contexto del caso de uso.

Los puntos de extensión son definidos como una subclase de *Action* para mantener consistencia con la noción de que un caso de uso es una secuencia de acciones.

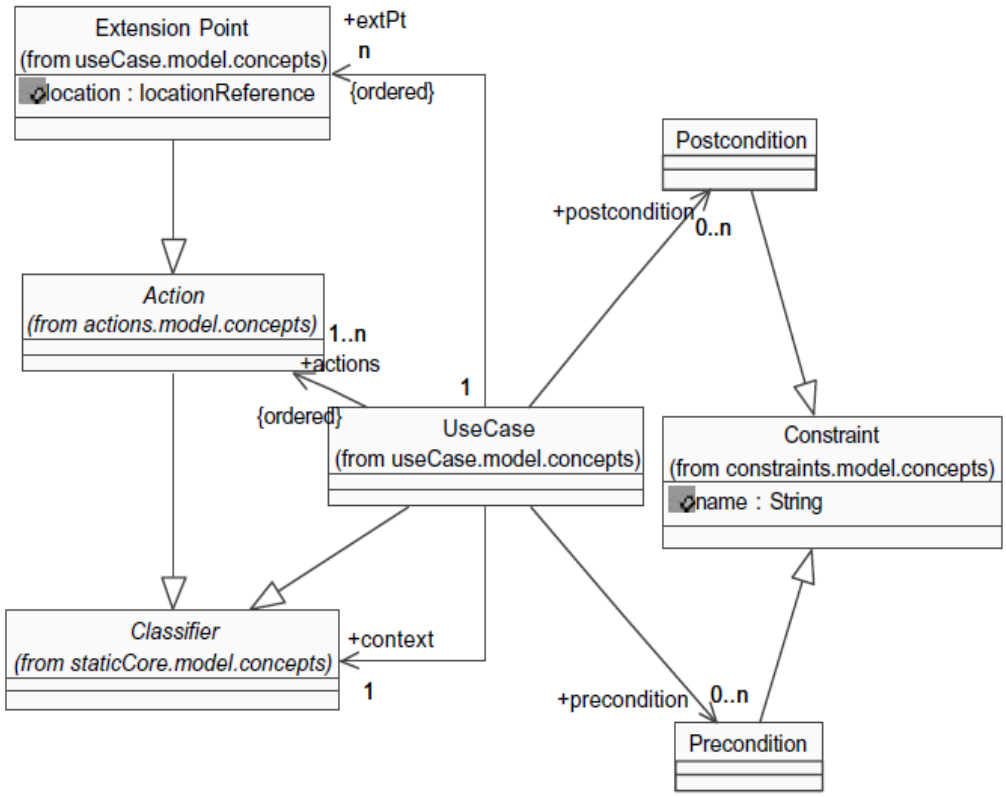

Figura 8: Meta-modelo de caso de uso de Williams

La Figura 9 muestra flujos de casos de uso y los actores asociados.

La secuencias de casos de uso que representan estos flujos son interesantes desde el punto de vista de *testing*. Los actores se comunican con el sistema a través de mensajes *MessageAction*.

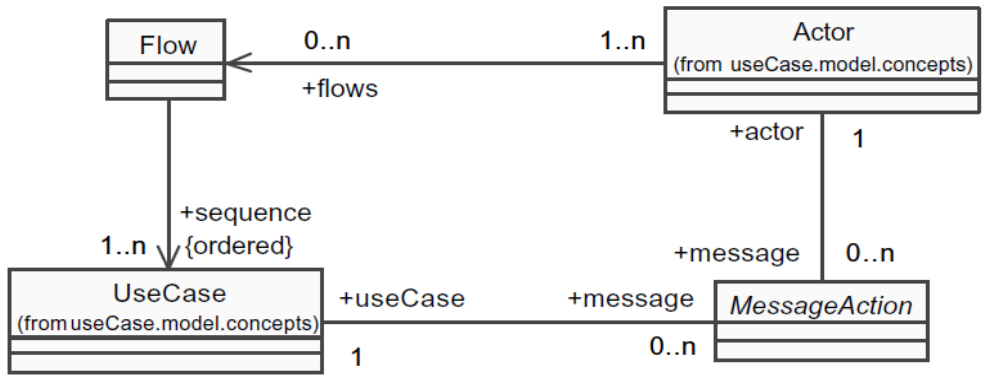

Figura 9: Flujos de casos de uso y actores

A continuación, en la Figura 10 se formalizan los conceptos de parámetros, particiones y valores asociados.

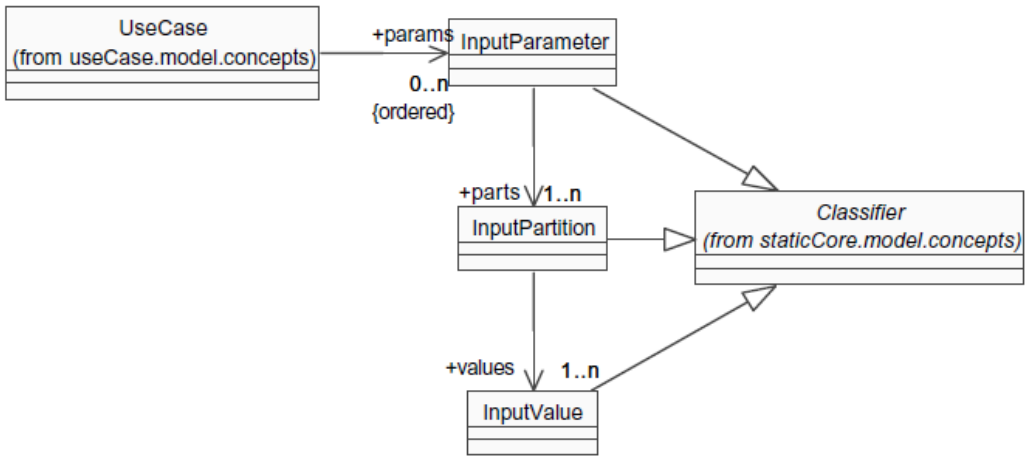

Figura 10: Parámetros, particiones y valores

El conjunto de parámetros asociados con un caso de uso fluyen desde el exterior del sistema hacia el interior a través de los actores. Estos parámetros están ordenados y cada uno puede tomar un valor lógico representado por el concepto *InputPartition*. Los valores lógicos pueden ser asociados con valores concretos a través del concepto *InputValue*.

2.5.1 Evaluación de la propuesta

El autor considera que el meta-modelo de casos de uso de UML puede servir como base para el testing basado en modelos pero requiere información adicional para ser de utilidad. En el trabajo se identificaron un conjunto de requerimientos que el meta-modelo debe satisfacer para facilitar la captura de dicha información adicional.

Las relaciones de inclusión y extensión requieren trabajo adicional para formalizar la noción de punto de referencia en el caso de uso base. También debe considerarse con cuidado como estas relaciones afectan a uno u otro caso de uso.

Otros elementos de UML podrían ser usados para modelar el comportamiento detrás de los casos de uso, como por ejemplo diagramas de secuencia. La información así representada podría servir para mejorar los algoritmos de generación de pruebas, sin embargo, necesita ser relacionada con el meta-modelo de casos de uso en forma rigurosa.

2.6 Diagramas de actividad a partir de casos de uso

Entre los trabajos que existen sobre el tema seleccionamos dos que tienen un enfoque similar al que nos proponemos. Por un lado analizamos una propuesta donde los autores presentan un sistema para generar diagramas de actividad UML en forma automática a partir de requerimientos funcionales de un sistema [GCEMR08]. Para ello, crean un meta-modelo de requerimientos funcionales y definen un conjunto de transformaciones entre ese meta-modelo y el meta-modelo de diagramas de actividad de UML.

La contribución principal de su trabajo, es la generación automática de diagramas de actividad a partir de la descripción textual de los escenarios del caso de uso. Esta generación es realizada a partir de transformaciones entre modelos que representan el caso de uso textual y los diagramas de actividad de UML. Los autores construyen un meta-modelo de requerimientos funcionales y aplican transformaciones implementadas en lenguaje QVT para generar diagramas de actividad.

Al disponer de una representación gráfica de los casos de uso (como Diagramas de Secuencia o de Actividad) es posible mostrar fácilmente cuestiones dinámicas como pueden ser bucles y escenarios alternativos así como diferentes opciones que puedan existir. Además, estos diagramas pueden ser fácilmente analizados por herramientas automatizadas y ser usados para generar casos de prueba.

Por otro lado, estudiamos un trabajo donde los autores presentan una serie de pasos para generar casos de prueba en forma automática a partir de los requerimientos funcionales del sistema [GEMT09].

La metodología seguida está basada en el paradigma de *Model Driven Engineering* (MDE) para la generación automática de casos de prueba y utiliza diagramas de actividad como herramienta en la automatización. La solución planteada comienza con la definición de los requerimientos funcionales de un sistema y se proponen una serie de transformaciones automáticas basadas en QVT para derivar casos de prueba.

Los escenarios de prueba son el resultado de dichas transformaciones, un escenario de prueba es una secuencia concreta de ejecución del sistema y que es derivado de los requerimientos funcionales.

Ademas, como resultado de las transformaciones, se obtiene un conjunto de variables operacionales que son elementos del dominio cuyo valor puede cambiar en los distintos escenarios de ejecución de un requerimiento funcional.

De acuerdo al paradigma *MDE*, el trabajo propuesto tiene dos partes, por un lado se definen meta-modelos y por otro transformaciones entre ellos. Las transformaciones entre modelos son la forma de producir resultados en este paradigma.

El meta-modelo de requerimientos funcionales utilizado es el propuesto en el trabajo [GCEMR08]. El meta-modelo con conceptos sobre el testing funcional está compuesto por un conjunto de meta-clases que abarcan la definición abstracta de los escenarios de prueba, de las variables operacionales y de los casos de prueba del sistema, entre otros.

Luego de aplicar las transformaciones entre estos modelos se obtienen los escenarios de ejecución y finalmente, a partir de los escenarios de ejecución se construyen los casos de prueba del sistema.

Los trabajos presentados anteriormente son estudiados con mayor detalle en los anexos 1 y 2 de este informe.

2.7 Casos de prueba a partir de diagramas de actividad

2.7.1 Caso de prueba

Un *Caso de Prueba* es un conjunto formado por valores de prueba, condiciones de ejecución y resultados esperados, que se combinan para lograr un determinado objetivo. Dicho objetivo puede ser ejecutar un camino específico dentro de un programa o verificar que el mismo cumpla con cierto requerimiento funcional.

El propósito de un *Caso de Prueba* es identificar y comunicar las condiciones que se deben cumplir y los pasos que se deben seguir para realizar la prueba. También incluye los resultados que se deben esperar durante y luego de la ejecución de la prueba.

Los casos de prueba sirven para verificar la correcta implementación y funcionamiento del sistema a partir de los requerimientos funcionales que sean definidos. Estos requerimientos pueden ser especificados a través de los casos de uso.

2.7.2 Creación de casos de prueba

En el trabajo de Naresh Ahlowalia, se utiliza la técnica de *Path Analysis*, para generar casos de pruebas a partir de los flujos de actividad obtenidos de la descripción de casos de uso [Ahl02].

La técnica consiste en cuatro pasos fundamentales. El primer paso consiste en crear un diagrama de flujo a partir de la descripción del caso de uso. A partir de este diagrama se determinan todos los caminos posibles, luego se analizan y priorizan los caminos encontrados, finalmente se decide que caminos se utilizarán en las pruebas.

El autor describe la técnica a grandes rasgos y no describe un algoritmo para la generación de los caminos y apela al criterio de la persona encargada del testing para limitar la generación de caminos cuando ante la presencia de *loops*.

Para analizar y priorizar los caminos se utilizan los parámetros frecuencia y criticidad. La frecuencia representa la probabilidad de que el usuario ejecute cierto camino mientras que la criticidad representa la importancia de la ocurrencia de una falla en dicho camino. Ambos valores se combinan para determinar un factor de prioridad para los caminos generados.

Para poder definir estos valores, se requiere también de la experiencia del tester o en su defecto de la ayuda de un experto del dominio que conozca de la lógica de negocio en cuestión.

En la selección de caminos, se toma en cuenta que se incluya el flujo básico del caso de uso ya que estos caminos son de gran importancia y no pueden ser ignorados desde el punto de vista funcional. Por último se utiliza el factor de prioridad para seleccionar otros caminos relevantes que no se cubren con los criterios anteriores.

En el trabajo se introduce el concepto de *Test Scenario* o escenario de prueba que representa un camino de ejecución al que se adicionan valores de prueba. Finalmente se genera un caso de prueba por cada camino seleccionado y múltiples escenarios de prueba según los distintos valores de pruebas.

En forma similar, en el trabajo de Jim Heumann se propone analizar los flujos del caso de uso para generar los distintos escenarios de prueba.[Heu01]

Utilizando estos escenarios, se analizan las condiciones que son necesarias para que cada uno sea ejecutado y se agrupan en una matriz llamada "Matriz de Casos de Prueba". Finalmente se agregan los datos de prueba necesarios para para ejecutar cada uno de los casos. La matriz expandida con datos de prueba es llamada "Matriz de Casos de prueba con Valores".

En el trabajo se menciona que existen varias técnicas para la generación de valores de prueba pero quedan fuera de su alcance.

En el trabajo de Gutierrez et al, se presenta una serie de pasos para generar casos de pruebas a partir de los requerimientos funcionales del sistema [GEMT09]. La metodología está basada en el paradigma de *Model Driven Engineering* (MDE) para la generación automática de casos de prueba. La solución comienza con la definición de los requerimientos funcionales, y propone un conjunto automatizable de transformaciones basadas en el standard *Query/View/Transformation* (QVT)[4] para derivar los casos de prueba del sistema.

Finalmente, en un trabajo reciente, Kundu y Samanta proponen una metodología para generar casos de prueba a partir de los diagramas de actividad de UML [KS09].

Esta metodología consiste en tres pasos:

1) Agregar la información necesaria para el testing en los diagramas de actividad.

2) Convertir el diagrama de Actividad en un grafo de actividad.

3) Generar los casos de prueba a partir del grafo de actividad.

El trabajo cubre diagramas de actividad con nodos de *Fork* y *Join* para el manejo de *threads* que pueden ser ejecutados en paralelo.

El tipo de fallas que los autores pretenden identificar con los casos de pruebas son de tres tipos diferentes, fallas en los nodos de decisión, fallas en los *loops* y fallas de sincronización. A este conjunto de fallas, los autores lo denominan "modelo de fallas".

4 Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) http://www.omg.org/spec/QVT/

2.8 UML Testing Profile (UTP)

Un perfil de UML (*UML profile*) puede ser visto como una especialización de UML que extiende el lenguaje original. Concretamente, UTP [UTP13] a través de sus conceptos provee extensiones a UML que permiten el diseño, visualización, especificación, análisis, construcción y documentación de los artefactos involucrados en las pruebas.

El perfil es independiente de los lenguajes de implementación y tecnologías utilizadas para desarrollar el sistema y además puede ser utilizado en diferentes dominios de aplicaciones.

En particular, el perfil introduce conceptos que se agrupan según la arquitectura de las pruebas (*test architecture*), el comportamiento de las pruebas (*test behavior*), los datos a ser usados en las pruebas (*test data*) y la representación del tiempo en las pruebas (*test time*).

La arquitectura de pruebas (*test architecture*) es un conjunto de conceptos (en adición a los conceptos estructurales de UML) que permiten especificar el aspecto de las pruebas. Dentro de dichos conceptos se encuentran:

- El contexto de prueba (*test context*) permite agrupar casos de prueba, configurar las pruebas y definir un control en la ejecución de las pruebas. Esto es el orden de ejecución requerido por las pruebas.
- La configuración de las pruebas (*test configuration*) es la estructura formada por el contexto de pruebas mostrando la comunicación entre componentes de pruebas y el sistema bajo prueba.
- El sistema bajo prueba (*SUT*) es uno o más objetos dentro de una especificación de prueba que pueden ser identificados con el sistema que se quiere probar.
- Los componentes de pruebas (*test component)* definen objetos dentro del sistema de prueba que se comunican con el *SUT* u otros componentes para realizar el comportamiento de la prueba.

Otro aspecto importante que describe UTP, es el comportamiento de las pruebas (*test behavior*), este especifica las acciones y criterios necesarios para evaluar el objetivo de la prueba (*test objective*). Este último describe que es lo que debe ser probado.

Los principales conceptos relativos al comportamiento de las pruebas que aparecen en UTP se describen a continuación:

- El objetivo de prueba (*test objective*) permite al diseñador expresar la intención de la prueba.
- El caso de prueba (*test case*) es una operación del contexto de prueba que especifica como un conjunto de componentes interactúan con el *SUT* para realizar el objetivo de prueba

- El veredicto (*verdict*) es un enumerado predefinido que especifica los posibles resultados que puede tener una prueba.
- La acción de validación (*validation action*) es realizada por un componente de prueba para indicar que el árbitro sea informado del resultado del componente en la prueba.

Por otro lado, encontramos la definición de datos de prueba. Para este propósito, los principales conceptos que brinda UTP son:

- Los selectores de datos (*data selectors*) permiten la ejecución repetida de los casos de prueba con valores en sus datos para estimular el sistema bajo prueba de diferentes maneras.
- Los "*data pools*" están asociados con el contexto de pruebas y pueden incluir particiones de datos (*data partitions*) que representan clases de equivalencia, así como valores de datos concretos.

La finalidad que persigue UTP es proveer la capacidad para construir especificaciones precisas de pruebas.

En la Figura 11 se muestran los conceptos antes mencionados y sus relaciones.

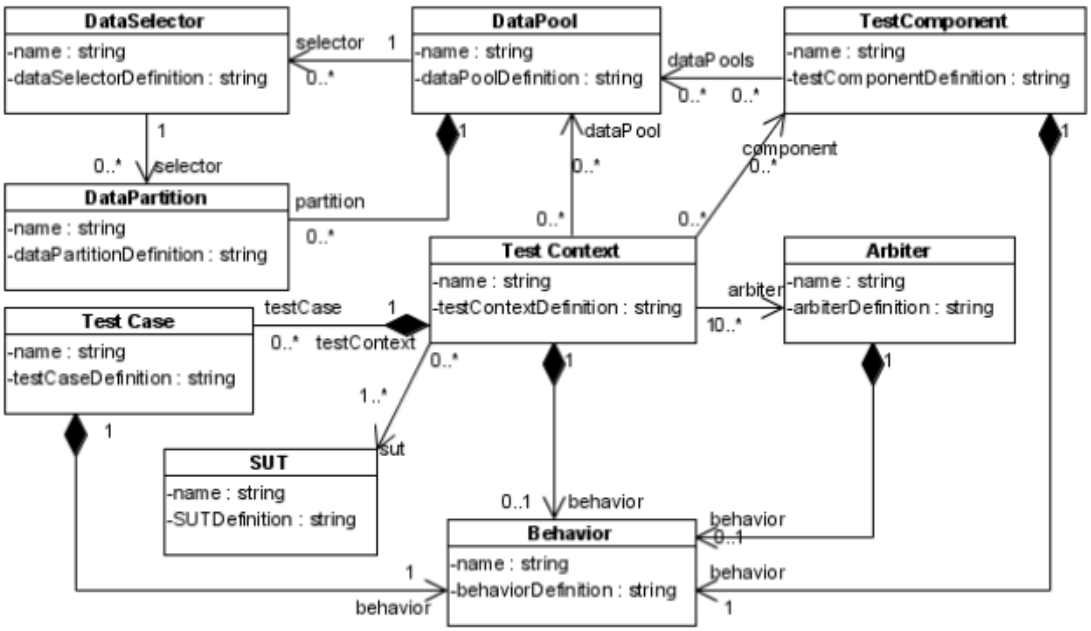

Figura 11: Meta-modelo UML Testing Profile

Más información sobre UTP puede ser encontrada en el sitio oficial.[5]

5 http://utp.omg.org/

2.9 Generación de valores de prueba

2.9.1 Black-box testing

Las pruebas de caja negra, también llamadas pruebas funcionales, tratan al *SUT* desde el punto de vista de su interacción con el medio. Las pruebas son creadas considerando la funcionalidad que el sistema brinda y no como esta funcionalidad es implementada. El comportamiento del *SUT* es evaluado en base a las entradas proporcionadas y las salidas producidas. Típicamente, un conjunto de casos de prueba es definido para cada funcionalidad del *SUT*, luego la salida del sistema es comparada con los resultados esperados para cada funcionalidad.

Dos técnicas para el desarrollo sistemático de pruebas de caja negra son partición en clases de equivalencia y el análisis de valores límite.

2.9.2 Clases de equivalencia

La partición en clases de equivalencia consiste en clasificar el dominio de cada parámetro de entrada según una propiedad que determina el mismo resultado en una funcionalidad del sistema. Los valores que pertenecen a una clase de equivalencia son tratados de igual manera por el sistema, por lo tanto es posible elegir un representante de cada clase de equivalencia para probar una funcionalidad. Esto permite reducir la cantidad de pruebas construidas para una funcionalidad ya que el resultado esperado es el mismo para cualquier representante de la clase.

2.9.3 Valores límite

La técnica de valores límite es usado frecuentemente en conjunto con partición en clases de equivalencia. En esta técnica, las pruebas creadas con partición en clases de equivalencia son complementadas por pruebas que utilizan valores límites de cada clase.

En muchos casos el sistema tiene casos especiales o extremos que son propensos a errores y los valores que causan estos errores suelen encontrarse en los bordes de las clases de equivalencia.

Capítulo 3 Contribuciones del proyecto

3.1 Meta-modelo de casos de uso para TestCaseTool

En este capítulo se presenta el meta-modelo de casos de uso que proponemos en nuestro trabajo y que tiene como base los trabajos estudiados en el capítulo de estado del arte.

El meta-modelo contiene conceptos e información relevantes para actividades de *testing* y en particular para la generación automática de pruebas. En él se representan conceptos que están presentes en la descripción textual de los casos de uso y que son independientes de la forma en que estos sean escritos.

Una de las ventajas del meta-modelo es que permite resolver en una misma etapa la creación de los casos de uso y la incorporación de conceptos de *testing*.

Es posible entonces, obtener casos de prueba a partir del modelo en que se expresan los casos de uso. La principal ventaja de esto es disponer de casos de prueba desde una etapa temprana en el desarrollo del sistema.

Los conceptos del meta-modelo vinculados al *testing* son un aporte novedoso de este proyecto. Estos fueron diseñados con el objetivo de que sea posible generar casos de prueba en forma automática a partir de los casos de uso. Además, se incluyen conceptos que permiten generar valores de prueba para verificar el comportamiento del sistema ante esos valores usados en las pruebas.

En el estado del arte, si bien existen varios meta-modelos sobre casos de uso, solo en uno de ellos se mencionan conceptos vinculados a pruebas del sistema, aunque no se considera la generación automática de esas pruebas.

Entonces, el principal aporte del meta-modelo que construimos es que permite crear, en forma automática, casos de pruebas junto a valores de prueba a partir de la descripción textual de los casos de uso.

Al igual que en el trabajo de Siqueira y Muniz, se construirá un meta-modelo con la finalidad de representar los conceptos y relaciones que están presentes en los casos de uso, sin agregar la complejidad que supone cumplir con el estándar de UML.

En el contexto de nuestro proyecto, contar con un meta-modelo que cumpla con todas las restricciones de UML no es imprescindible, ya que es una herramienta intermedia para generar pruebas a partir de los casos de uso. Sin embargo, se deja abierta la posibilidad en el futuro de agregar restricciones al meta-modelo que permitan cumplir con la especificación de UML.

De los trabajos vistos, el meta-modelo propuesto por UML no considera la descripción en forma textual de los caso de uso, ni la secuencia de acciones que forman parte de ellos. Por lo tanto, para describir las acciones en los casos de uso se usarán conceptos que aparecen en los trabajos de Siqueira-Muniz y Somé.

El trabajo de Williams es el único donde se incorporan conceptos que están relacionados con la generación de pruebas a partir de los casos uso. Algunos de estos conceptos son parámetros, particiones y valores. Estos conceptos sirven de base para el meta-modelo que proponemos.

3.1.1 Descripción del meta-modelo

El meta-modelo cuenta con una gran cantidad de conceptos por lo tanto es apropiado dividir la presentación en secciones para facilitar su comprensión. Por último, para dar una visión global, se muestra el meta-modelo completo.

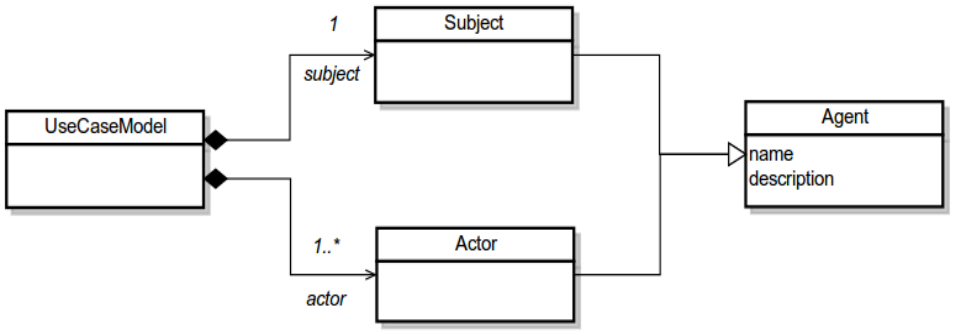

Figura 12: Concepto de UseCaseModel

Se comienza la descripción introduciendo el concepto de *UseCaseModel* que se observa en la Figura 12. Este concepto fue extraído del meta-modelo propuesto Siqueira-Muniz y permite la existencia de varios casos de uso en el modelo.

A diferencia del trabajo de Siqueira y Muniz , donde los conceptos de *Subject*, *Actor* y *UseCase* están contenidos en *UseCaseModel*, en nuestro modelo solamente *Subject* y *Actor* se relacionan con *UseCaseModel*. Además proponemos representar un único sujeto, que es sobre el cual los casos de uso aplican.

El concepto de *Agent* también aparece en el trabajo de Siqueira-Muniz y es una generalización de los conceptos *Subject* y *Actor*. En el meta-modelo agregamos la propiedad *description* al concepto de *Agent* para incluir información que ayude a describir la semántica de los conceptos *Subject* y *Actor*.

En la Figura 13 se muestra el concepto más importante del meta-modelo que es el concepto de *UseCase.*

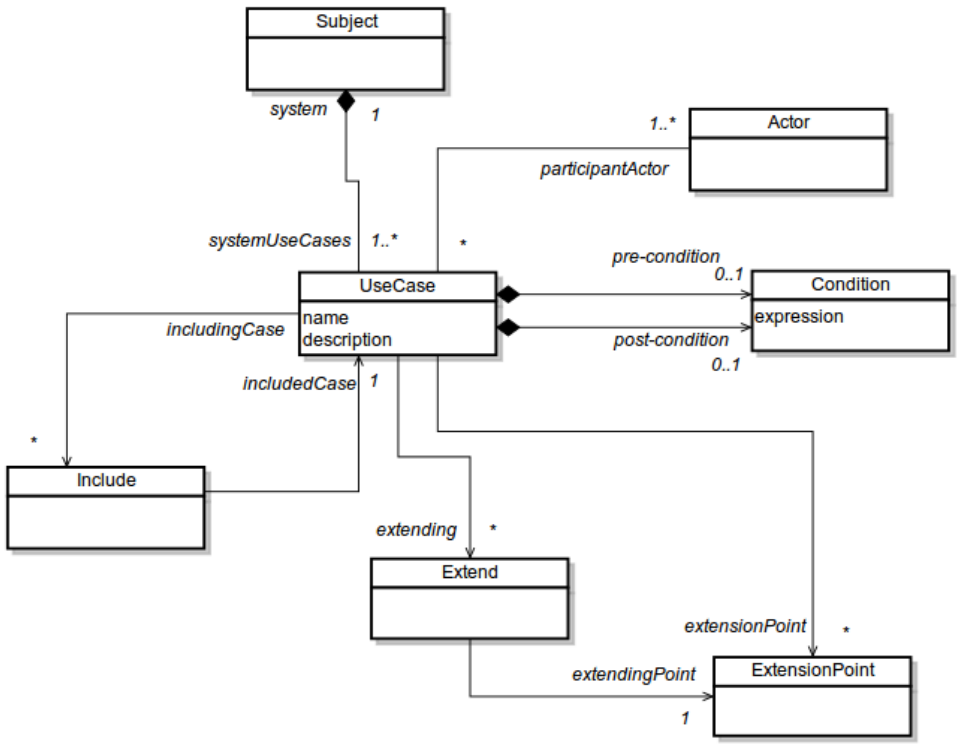

Figura 13: Concepto de UseCase

En el trabajo de Siqueira-Muniz los casos de uso son incluidos en la meta clase *UseCaseModel.* Se observa una carencia en cuanto a como los conceptos de *Subject* y *UseCase* se relacionan entre sí ya que no existe una vinculación directa entre ellos.

Para resolver esto, creemos que es conveniente que los casos de uso sean contenidos por la meta-clase *Subject*. Las ventajas de esta representación son que queda clara la idea de caso de uso como la interacción entre actores y un sistema, y además representa la relación entre los casos de uso y el sujeto.

En la Figura 13 también se observa la meta-clase *Actor* y su participación en los casos de uso.

Los casos de uso tienen condiciones asociadas. Por un lado, condiciones llamadas *pre-conditions*, que son necesarias para que el caso de uso pueda ser ejecutado. Por otro lado, condiciones llamadas *post-conditions*, que describen el estado del sistema luego de la ejecución del caso de uso. Para representar dichas condiciones agregamos el concepto *Condition* que aparece en el trabajo de Siqueira-Muniz. La propiedad *expression* permite representar en forma textual la condición requerida.

Los conceptos de *Include* y *Extend* son utilizados para representar las relaciones de inclusión y extensión entre casos de uso. Estos fueron tomados del meta-modelo de casos de uso de UML y tienen el mismo significado que se describe allí.

La Figura 14 muestra otra perspectiva del concepto *UseCase* desde el punto de vista de los flujos de ejecución.

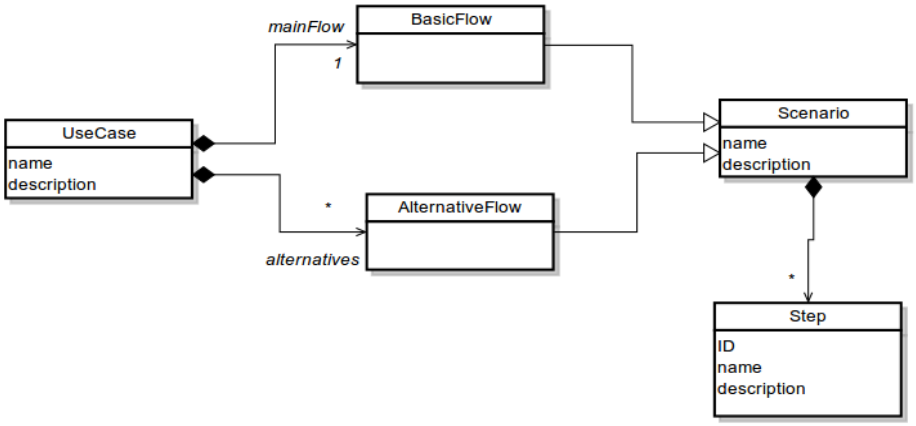

Figura 14: Flujos de ejecución del caso de uso

Un flujo de ejecución es una secuencia de pasos ordenados que son llevados a cabo por el sistema o los actores dentro del caso de uso. La idea de flujo está presente en el trabajo de Siqueira-Muniz con el nombre *FlowOfEvents*. También, el trabajo de Somé hace referencia al concepto de flujo pero es presentado como una secuencia de pasos sin hacer mención explícita del término flujo.

Los flujos en un caso de uso son *BasicFlow* (flujo básico) y *AlternativeFlow* (flujos alternativos). Todo caso de uso tiene un flujo básico que representa la ejecución exitosa del caso de uso.

Los flujos alternativos están asociados a una condición y representan un conjunto de pasos diferentes al flujo básico. En general corresponden a una ejecución del caso de uso fallida, donde no se alcanzan las metas de los actores involucrados.

A diferencia de la propuesta de Siqueira-Muniz, utilizamos el concepto de *Scenario* como generalización de los distintos tipos de flujo.

Todo escenario está compuesto por una secuencia ordenada de pasos *Steps*, donde cada uno de estos representa una acción o evento que realiza el sistema o los actores.

El término *Scenario* es usado en la definición de caso de uso que considera Cockburn y creemos conveniente utilizarlo ya que representa mejor la idea de interacción entre actores y sistema. Extendimos el concepto de *Scenario* con las propiedades *name* y *description* para facilitar la comprensión del modelo que se representa.

La idea de *Step* también se menciona en los trabajos de Siqueira-Muniz y Somé. En el meta-modelo extendemos este concepto con las propiedades *id*, *name* y *description*. Utilizamos la propiedad *id* para identificar un paso dentro de la secuencia.

La Figura 15 presenta con más detalle el concepto de *Step* y los subtipos que identificamos.

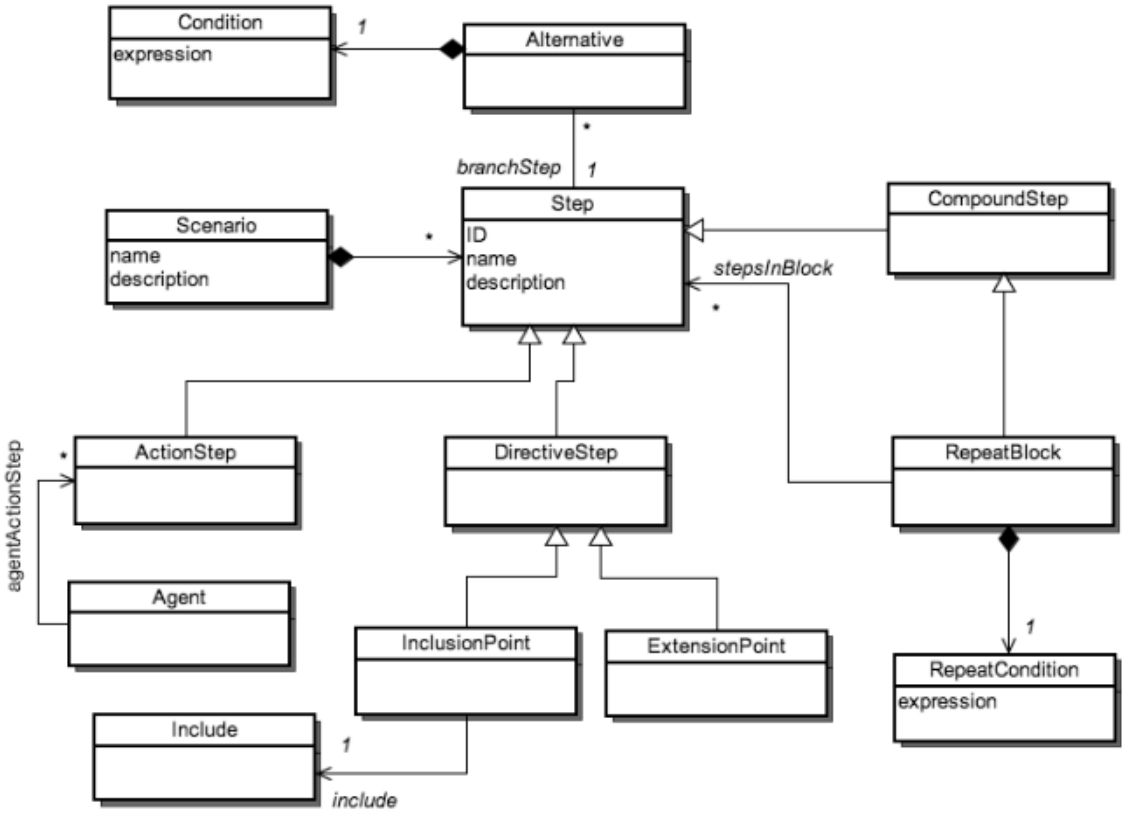

Figura 15: Diferentes tipos de pasos

El concepto de *Step* permite especificar las relaciones de inclusión y extensión entre casos de uso. Para esto utilizamos el concepto *DirectiveStep* que generaliza los subtipos *InclusionPoint* y *ExtensionPoint*.

El subtipo *InclusionPoint* representa la inclusión de un caso de uso a través del concepto *Include.*

La semántica detrás del concepto *InclusionPoint* es que la inclusión de un caso de uso se realiza a través de un paso especial en la secuencia.

El subtipo *ExtensionPoint* es un paso que sirve como punto de extensión para otro caso de uso. El caso de uso que extiende puede indicar cual es el caso de uso que está extendiendo en que puntos se produce la extensión

El concepto *CompoundStep* representa un compuesto que permite diferenciar un conjunto de pasos dentro de una secuencia. Esto es útil para introducir el concepto de *RepeatBlock* el cual permite la iteración de un conjunto de pasos mientras se cumpla cierta condición.

Los conceptos *DirectiveStep* y *CompoundStep* aparecen solo en la propuesta de Somé y creemos relevante incluirlos en el meta-modelo que proponemos.

En la propuesta de Siqueira-Muniz se hace referencia a los conceptos *Statement*, *LoopStatement* y *ConditionalStatement*. Dado que estos conceptos pueden ser representados como compuestos por medio de *CompoundStep*, optamos no incluirlos en nuestro meta-modelo.

La noción de *ActionStep* se considera en los trabajos de Somé y Siqueira-Muniz. De igual manera, Williams menciona acciones que son llevadas a cabo por los actores y el sistema. Ya que dicho concepto tiene consenso entre todos los trabajos, creemos imprescindible incluirlo en el meta-modelo.

En nuestro trabajo, un paso de tipo *ActionStep* representa una acción, que puede ser, tanto una invocación que hace un actor al sistema, como una acción que realiza el sistema en respuesta a un evento. La relación entre *Agent* y *ActionStep* sirve para conocer quien ejecuta una acción, que pueden ser o bien el sistema, o bien un actor participante.

La asociación *branchStep* permite conocer cual es el paso en el flujo básico, donde un flujo alternativo puede comenzar si se cumple cierta condición. La condición para que un flujo alternativo se ejecute se expresa a través de la asociación con el concepto *Condition*.

3.1.2 Conceptos vinculados al testing

Para poder construir casos de prueba a partir de los casos de uso, es necesario que el meta-modelo incluya conceptos que representen el ingreso de datos al sistema así como las respuestas que este genera ante un evento.

Consideramos que estos conceptos se pueden representar a través de la noción de parámetro. El concepto de parámetro aparece en el trabajo de Williams pero no se hace referencia en que momento éstos son asociados al caso de uso ni de que manera son usados en las pruebas.

En este sentido, el aporte de nuestro meta-modelo es la vinculación de parámetros con un paso de tipo *ActionStep*. Esto se corresponde con la interpretación de *ActionStep* como una invocación al sistema o una respuesta del mismo, y es allí donde los parámetros se vuelven significativos.

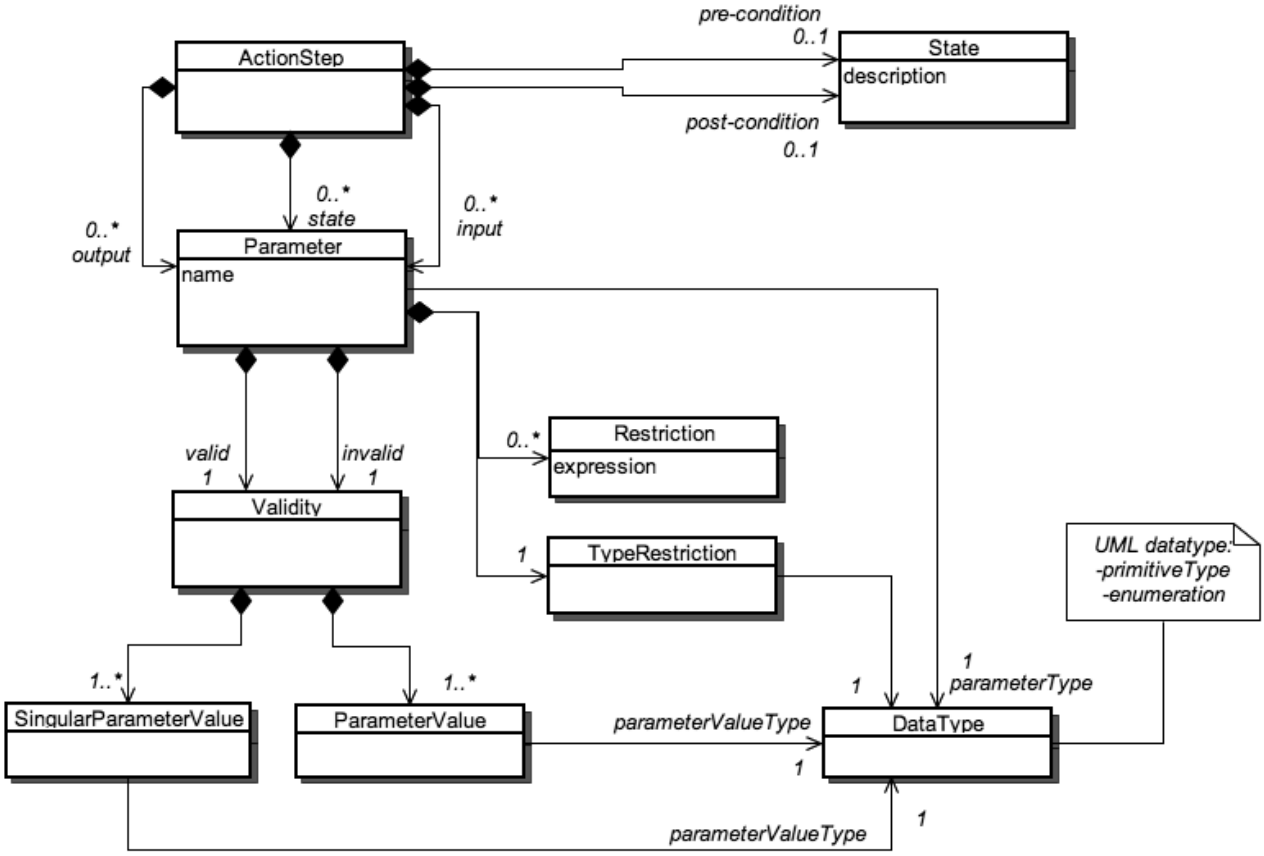

Figura 16: Parámetros, restricciones y valores de prueba

Las acciones representadas por *ActionStep* pueden estar asociadas con un conjunto de parámetros.

Definimos tres tipos de parámetros que son de entrada, de salida y de estado. En nuestro meta-modelo, el tipo de parámetro es representado por los roles *input*, *output* y *state* en las asociaciones entre *ActionStep* y *Parameter*. Los parámetros de entrada se usan en una invocación al sistema, representada en un paso de tipo *ActionStep*. Los parámetros de salida representan el resultado del sistema en respuesta a una invocación definida en un paso de tipo *ActionStep*. Los parámetros de estado, asociados con una acción, representan el estado interno del sistema y pueden ser relevantes en la ejecución de dicha acción.

El concepto de parámetro de estado es un aporte de nuestro trabajo ya que Williams no hace referencia al mismo. El concepto *State* permite describir condiciones sobre el estado del sistema antes y después de la ejecución de una acción. La diferencia entre el concepto *State* y un parámetro de estado, es que el primero solo describe, mientras que los parámetros pueden ser usados para generar valores de prueba.

El objetivo de automatizar actividades de *testing* a partir del meta-modelo, hace que sea necesario representar conceptos vinculados a valores de prueba y la validación de las acciones que realiza el sistema.

Con este fin introducimos el concepto *Validity* que es una clasificación de los valores concretos que puede tomar un parámetro. Esta clasificación agrupa los valores según una condición que determina si son válidos o inválidos.

La validez (*Validity*) se entiende como una cualidad que tiene un parámetro de ser aceptable para una acción, lo que permite definir y conocer el comportamiento esperado del sistema. El concepto de validez es útil para verificar el funcionamiento del sistema así como la ejecución de los pasos que componen los casos de uso.

La validez de un parámetro permite determinar cual es el siguiente paso a seguir en la ejecución de un flujo. Por ejemplo, dado un parámetro definido en una acción, si el parámetro es válido, el siguiente paso a ejecutar es el próximo en el flujo, mientras que, si el parámetro es inválido, el próximo paso a ejecutar corresponde al primer paso en un flujo alternativo.

El concepto *Restriction* representa una restricción que se impone sobre el dominio de un parámetro. La propiedad *expression* permite expresar la restricción en forma de texto y es usada para generar los valores de prueba que satisfacen la restricción.

Los parámetros tienen asociado un tipo de dato que se representa con el concepto *DataType*. El tipo de dato puede ser usado también para imponer una restricción sobre los posibles valores que puede tomar un parámetro. Las restricciones vinculadas el tipo de dato son representadas por el concepto *TypeRestriction*. A modo de ejemplo, los valores que puede tomar un parámetro de tipo entero positivo son aquellos mayores a cero.

El valor de prueba que puede tomar un parámetro es representado por el concepto *ParameterValue*. El tipo de dato asociado a este valor es el mismo que el tipo de dato asociado al parámetro.

Los valores que puede tomar a un parámetro están determinados por la restricción sobre el parámetro y la validez del mismo. El conjunto de valores que cumplen las restricciones impuestas sobre el parámetro son considerados valores válidos. Es decir, valores asociados a la clasificación de parámetro válido.

De igual forma, el conjunto de valores que no satisfacen las restricciones sobre el parámetro son considerados valores inválidos. Es decir, valores asociados a la clasificación de parámetro inválido.

Los valores concretos de un parámetro podrían ser generados en forma automática mediante el procesamiento de las restricciones.

El concepto de *SingularParameterValue* permite incluir valores que no puedan generarse en forma automática y que resulten de interés para el tester al momento de probar el sistema. El tipo de dato asociado a *SingularParameterValue* no está restringido al tipo de dato del parámetro. Esto permite definir valores de prueba que no cumplen con el tipo de dato, por ejemplo asignar un valor de tipo *string* a un parámetro de tipo entero para verificar el comportamiento del sistema.

Nuestro meta-modelo busca representar los parámetros, restricciones y valores en forma abstracta para ser independiente de la técnica usada en la generación de valores de prueba concretos.

A continuación, en la Figura 17 se muestra el meta-modelo completo.

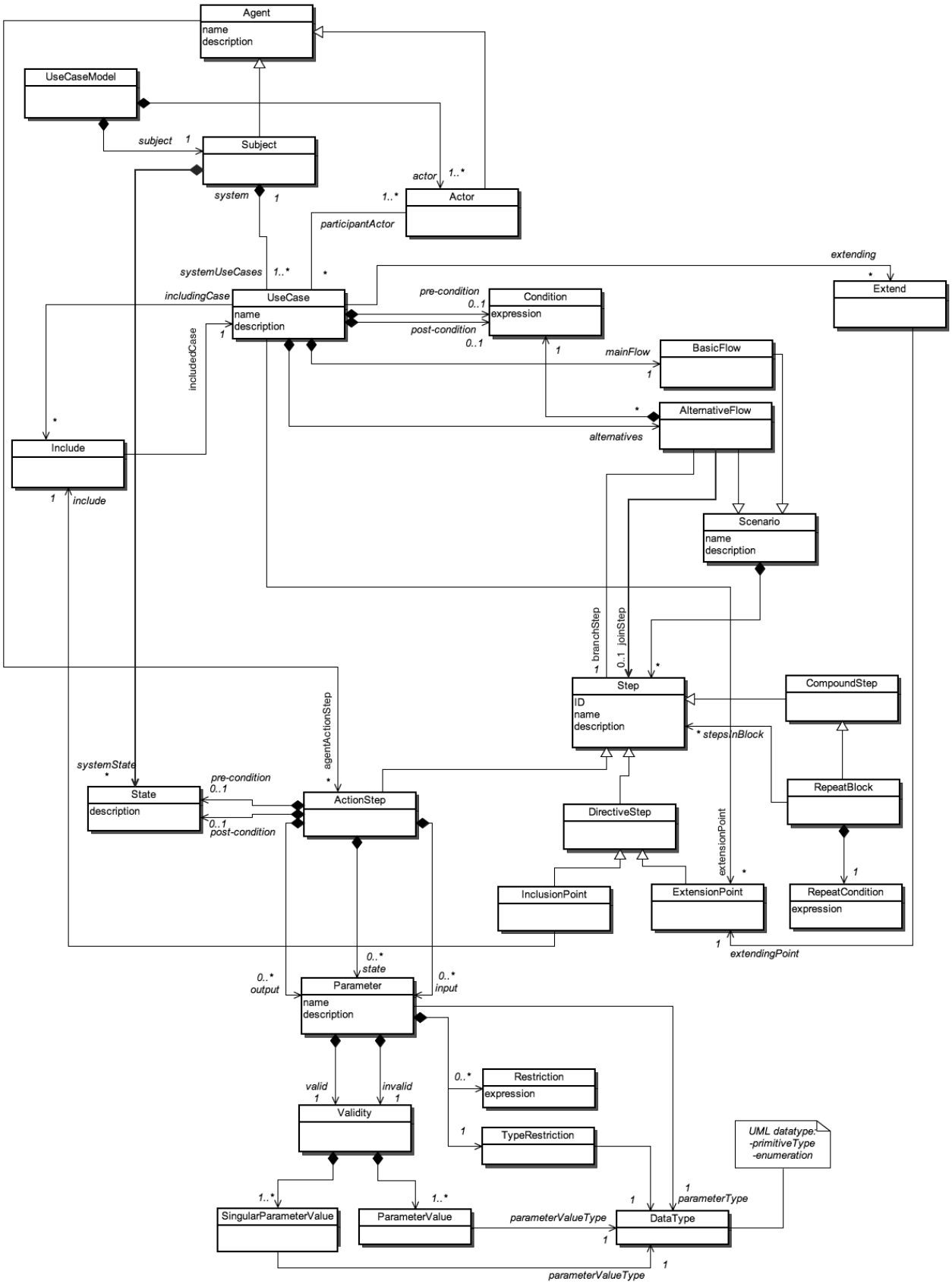

Figura 17: Meta-modelo de Casos de Uso de TestCaseTool

3.2 Generación de diagramas de actividad a partir de casos de uso

En la búsqueda de una metodología para la generación automática de pruebas a partir de casos de uso consideramos transformar el meta-modelo de casos de uso en un meta-modelo de pruebas. La transformación entre modelos sigue la propuesta planteada en *Model Based Testing*.[BDGSW07]

Existen dos alternativas para realizar la transformación, por un lado es posible transformar directamente el meta-modelo de casos de uso en un meta-modelo de pruebas, por otro lado, es posible transformar el meta-modelo de casos de uso en un meta-modelo de diagrama de actividad y luego, a partir de este crear un meta-modelo de pruebas.

El primer enfoque consiste en una transformación directa entre casos de uso y pruebas del sistema, éste tiene la desventaja que no ha sido abordado en profundidad en la literatura y por lo tanto requiere un estudio formal que escapa al alcance de este proyecto.

Por lo tanto, el enfoque que seguiremos, es aquel que utiliza un modelo intermedio entre casos de uso y pruebas del sistema. Este enfoque ha sido estudiado y existen trabajos donde se ha aplicado.

La técnica para generar pruebas a partir de los casos de uso consiste en crear diagramas de actividad que representan la interacción en los casos de uso, y luego, aplicar un algoritmo que recorre cada uno de los diagramas de actividad y genera los casos de prueba del sistema.

3.2.1 Nuestra propuesta

Los pasos para crear diagramas de actividad que representan las interacciones en los casos de uso están inspirados en los trabajos estudiados en el estado del arte.

El meta-modelo de diagrama de actividad que usaremos (Figura 18) es una simplificación del propuesto por UML y fue tomado del trabajo de Gutiérrez et al [GCEMR08]. Este permite representar todos los conceptos vinculados a las pruebas que fueron definidos en el meta-modelo de casos de uso construido.

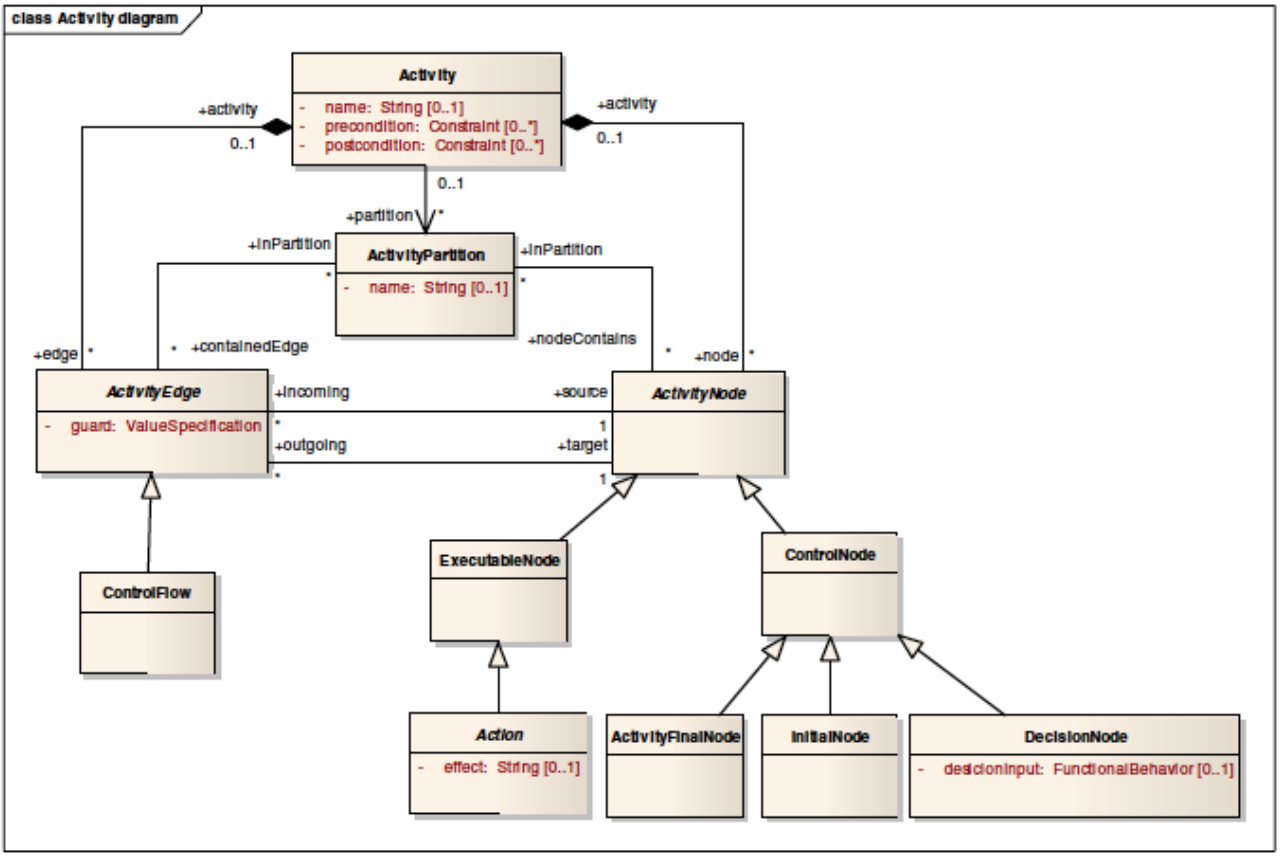

Figura 18: Meta-modelo de Diagrama de Actividad

En el trabajo de Gutiérrez et al, se describe a grandes rasgos una serie de pasos que definen la transformación entre casos de uso y diagramas de actividad.

Inspirados en dicho trabajo, pero haciendo un análisis de los conceptos presentes en nuestro meta-modelo de casos de uso, construimos un algoritmo que transforma las interacciones en los casos de uso en diagramas de actividad que las representan.

A continuación se presenta el pseudo-código del algoritmo construido.

```
Algoritmo UC-AD

1  Crear un nodo de acción (Action) del diagrama de actividad por cada paso del
   flujo principal (BasicFlow) del caso de uso.
2  Se define flujoActual = flujo principal
3  Para cada paso (Step) de flujoActual y el nodo (Action) correspondiente en
   el Diagrama de Actividad:
4    Asignar acción del nodo = acción correspondiente al paso.
5    Si no existen flujos alternativos en el paso:
6      Si el paso es el último del flujo
7        Si el flujo actual = flujo principal
8          Crear el nodo final y unir la acción actual al nodo final.
9        Sino // estamos en un flujo alternativo
10         Si el flujo tiene un punto de retorno (joinStep)
11           Unir la acción actual a la acción que referencia joinStep
12         Sino
13           Crear nodo final y unir la acción actual al nodo final.
14     Sino // no es el último paso del flujo actual
15       Unir la acción actual al nodo acción del siguiente paso del flujo
16   Sino // existe un branch en el paso (hay flujo alternativo)
17     Crear un nodo decisión.
18     Unir la acción actual al nodo decisión.
19     Para cada flujo alternativo en el paso actual
20       Crear una acción por cada paso del flujo alternativo
21       Crear una rama del nodo decisión hacia la primer acción del
         flujo alternativo.
22       Poner la condición de branch a la rama creada.
23       Marcar flujo actual = flujo alternativo
24       Ir al paso 3)
```

Tabla 1: Pseudo-código del algoritmo para generar diagramas de actividad

En el primer paso del algoritmo se generan las acciones *Action* que forman parte del flujo de actividades del diagrama y que corresponden a los pasos del flujo principal del caso de uso.

A partir de allí, estas acciones se unen mediante enlaces *links* que determinan los caminos que existen en el diagrama de actividad (pasos 11, 15, 18 y 21).

El tercer paso del algoritmo tiene características recursivas ya que la presencia de flujos alternativos en el caso de uso determina una nueva ejecución del algoritmo para cada uno de esos flujos (paso 24).

La existencia de flujos alternativos en el caso de uso, determina la existencia de nodos de decisión que permiten representar las distintas alternativas en el recorrido de los flujos de actividad (paso 17).

En la Figura 19 se representa un diagrama de actividad que contiene las distintas formas en que los flujos alternativos pueden finalizar (pasos 11 y 13).

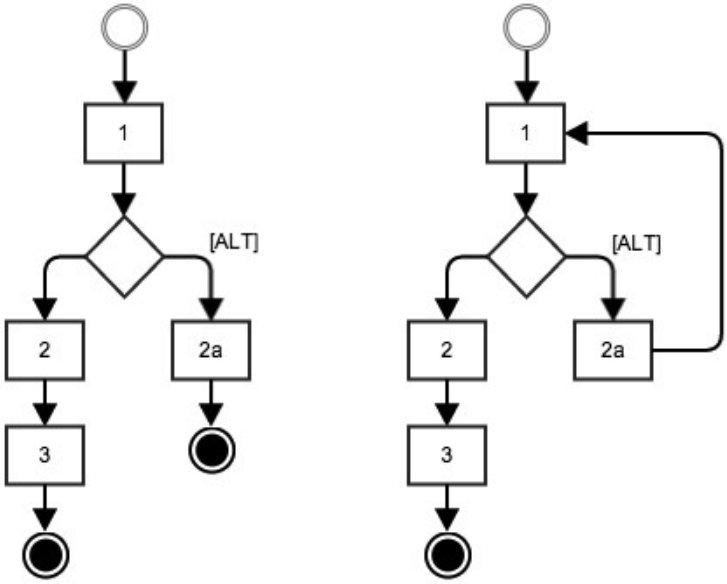

Figura 19: Ejemplos de finalización de flujos alternativos

El flujo principal siempre termina con un nodo de fin (paso 8), en cambio, para los flujos alternativos existen dos posibilidades. Por un lado, al igual que el flujo principal, un flujo alternativo puede terminar el caso de uso. En esta situación es necesario crear un nodo de fin y vincularlo a la ultima acción en el flujo (paso 13).

Por otro lado un flujo alternativo podría volver a un nodo *Action* del flujo principal. Esta situación produce un vínculo en el diagrama donde el nodo destino es la actividad correspondiente al paso del caso de uso identificado como *joinStep* (paso 11).

El resultado de este algoritmo es un diagrama de actividad que representa los posibles flujos de ejecución de un caso de uso, y mediante la utilización del mismo, es posible implementar recorridas que sirven para construir pruebas basadas en la secuencia de acciones que existen en el diagrama de actividad.

3.3 Generación de casos de prueba a partir de diagramas de actividad

En las etapas anteriores del proyecto se ha trabajado sobre la especificación formal de un caso de uso y la elaboración de un meta-modelo para su representación. También se definieron una serie de pasos para generar diagramas de actividad a partir de los casos de uso.

En esta etapa, nos enfocaremos en generar casos de prueba a partir de los diagramas de actividad construidos. La propuesta que planteamos se fundamenta en los trabajos estudiados en el estado del arte.

Definiremos un algoritmo que permite obtener escenarios de ejecución de los casos de uso a partir de recorridas realizadas sobre el diagrama de actividad. Estos escenarios serán la base para generar los casos de prueba.

3.3.1 Algoritmo propuesto

Los casos de prueba serán generados en base a las ideas propuestas en la literatura y que fueron estudiadas en el estado del arte. En nuestro trabajo, las transformaciones entre modelos son definidas en lenguaje natural e implementadas utilizando Java, a diferencia de los trabajos estudiados donde se emplea el lenguaje QVT. Esta decisión fue tomada considerando el alcance del proyecto.

Una vez obtenido el diagrama de actividad a partir del modelo de casos de uso, nuestro objetivo es recorrer dicho diagrama para así construir los distintos caminos a los que llamaremos escenarios de prueba. Un camino es una secuencia de nodos que tiene su origen en el nodo inicial y termina en algún nodo final.

Para generar estos caminos se definirá un algoritmo que emplea la técnica de *Backtracking* y se ejecuta sobre el diagrama de actividad, haciendo una recorrida en profundidad *Deep-First-Search* (DFS). El algoritmo que proponemos es similar al propuesto por Kundu y Samanta [KS09], pero a diferencia de esos, utilizaremos directamente el diagrama de actividad en vez de construir un grafo de actividad intermedio. Además no se contemplará el soporte para threads ya que en nuestro trabajo estas estructuras de control no fueron tomadas en cuenta con el fin de reducir la complejidad del problema.

Las variables operacionales que se utilizan en las decisiones entre flujos alternativos son incorporadas en el algoritmo. La información asociada a estas variables será agregada como último paso en la construcción de los casos de prueba. Esto será de utilidad para la persona encargada de ejecutar las pruebas.

A continuación vemos el pseudo-código del algoritmo construido para generar los escenarios de pruebas.

```
Algoritmo: GenerarEscenariosDelDiagramaDeActividad
Entrada: Diagrama de Actividad UML
Salida: Conjunto de Escenarios de prueba (TestScenario)

Inicializar:
LoopFlag = 0;
Para cada nodo: CantidadDeVisitas = 0;
Stack S vacío;
Stack P vacío; // Stack de variables operacionales
Set salida vacío;

Recorrer el Diagrama de Actividad en Profundidad con un algoritmo DFS (Depth-
First-Search).

Para cada nodo visitado en el algoritmo:
1  Incrementar la cantidad de visitas en 1;
2  Hacer push del nodo en el Stack S
3  Si el nodo define una variable operacional
4    Hacer push en P de [nodo, variable, #visitas_al_nodo, validez]
5  FinSi
6  Si la cantidad de visitas del nodo = 2 entonces:
7    LoopFlag = 1
8  FinSi
9  Si LoopFlag = 1 y cantidad de visitas del nodo = 3 entonces:
10   Hacer Backtracking hasta el último nodo D de decisión que tiene
     al menos un hijo con cantidad de visitas menor a 2.
11   Quitar del Stack los nodos hasta que el nodo D quede en el tope.
12 FinSi
13 Si el nodo es de tipo Final entonces:
14   Copiar el actual contenido de S a un array, donde el tope del stack
     representa a el último paso del escenario encontrado. La secuencia de
     elementos (nodos) en el array representa el escenario de ejecución.
15   Para cada elemento copiado,
16     Si existe una entrada en el stack de parámetros para dicho nodo,
17       Asociar la información de la variable operacional en el stack con
         el nodo en cuestión.
18     FinSi
19   FinPara
20   Agregar el array al conjunto de salida.
21   Hacer Backtracking hasta el último nodo D de decisión que tiene
     al menos un hijo sin ser visitado.
22   Quitar del Stack los nodos hasta que el nodo D quede en el tope.
23   Para cada nodo quitado del stack S
24     Si el tope del Stack P corresponde al nodo
25       Hacer POP de P
26     FinSi
27   FinPara
28   Si el Stack S está vacío entonces:
29     Terminar el Algoritmo
30     Retornar salida
31   FinSi
32   Si hay elementos en el Stack P
33     Cambiar la validez al elemento en el tope del stack
34   FinSi
35 FinSi
FinPara
```

Tabla 2: Pseudo-código para la generación de casos de pruebas

El resultado de este algoritmo es un conjunto de escenarios de prueba *TestScenario_TestContext*. Estos escenarios contienen una secuencia formada por los nodos del diagrama de actividad que participan en el escenario y además guardan la secuencia de flujos que incluyen a los nodos participantes (paso 14).

En los nodos que involucran variables operacionales, se incluyen los valores de prueba de dichas variables así como la restricción de validez que aplica sobre los valores (pasos 4 y 17).

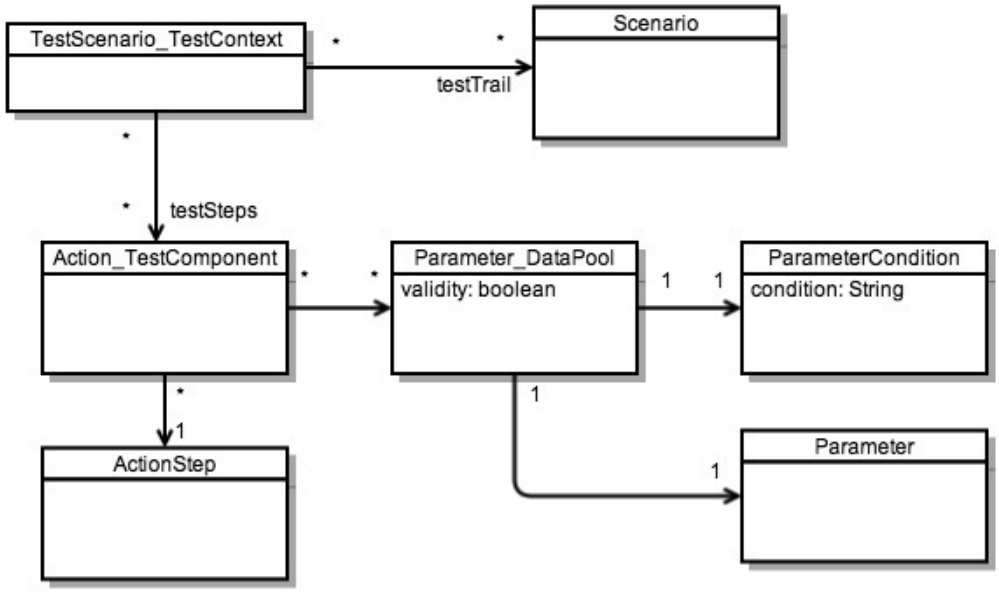

Figura 20: Diagrama de clases de la salida del algoritmo que genera los escenarios de prueba

El nodo *Action_TestComponent* del diagrama de actividad posee una referencia a un elemento *ActionStep* del meta-modelo del casos de uso. Este elemento es quien dio origen a la acción al transformar el caso de uso en el diagrama de actividad.

La validez *Validity* que forma parte de la salida del algoritmo y que está asociada a *Parameter_DataPool*, se usará al momento de generar los valores de prueba, que serán tomados de la restricción impuesta para el parámetro.

Para evitar que el algoritmo quede en loop infinito con los ciclos generados en el diagrama de actividad se utiliza una flag auxiliar llamada *loopFlag* (pasos 6, 7, 9, 10 y 11).

3.3.2 Solución propuesta y UTP

Los escenarios de prueba que se obtienen con el algoritmo visto en la sección anterior son usados para generar casos de prueba (*TestCase*). Los casos de prueba se identifican con un nombre que está formado por el nombre del caso de uso junto a un identificador "TC_<id>", donde "<id>" es un entero que se incrementa con cada caso de prueba generado.

Por ejemplo, si el caso de uso tiene el nombre "Validar usuario" entonces los posibles casos de prueba pueden ser:

- Validar_Usuario_TC_1
- Validar_Usuario_TC_2
- Validar_Usuario_TC_3

Luego, se agregan los pasos que se deben ejecutar en el caso de prueba junto a la información sobre variables operacionales que pueda estar vinculada a estos pasos.

Entre los objetivos del proyecto se propuso utilizar conceptos de *UTP* para representar los casos de prueba. *UTP* presenta la ventaja de ser un lenguaje estandarizado basado en *UML* para la representación de pruebas y que permite a los usuarios familiarizados con estas tecnologías una rápida comprensión del modelo de pruebas propuesto.

Con el fin de cumplir con la especificación de *UTP*, a continuación se muestra el rol que cumplen las partes de nuestro modelo de pruebas desde la perspectiva de *UML Testing profile*.

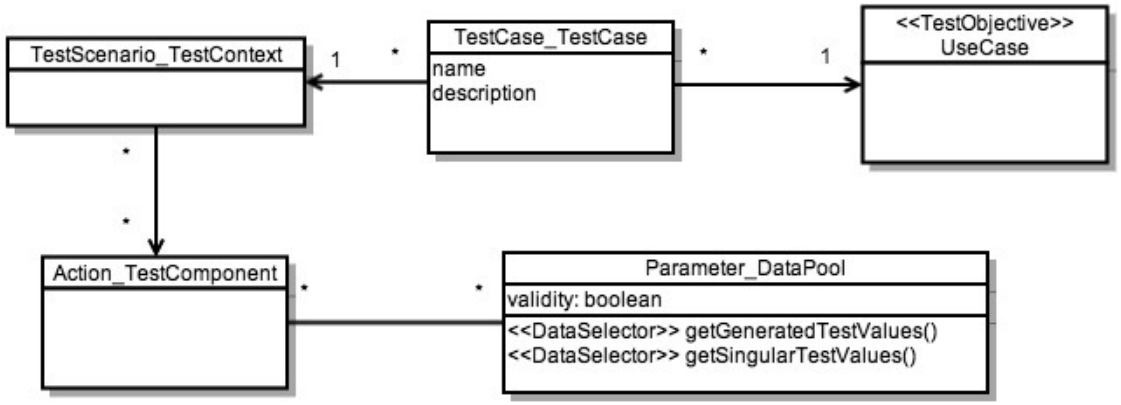

Figura 21: Meta-modelo de Casos de Prueba y su relación con UTP

3.3.3 Generación de valores de prueba

En esta sección se describe como los valores de prueba son generados a partir de las restricción impuestas sobre una variable operacional.

Las dos técnicas básicas en pruebas de software desde el punto de vista de la accesibilidad al código son las pruebas de caja negra y pruebas de caja blanca. También, ambas se pueden combinar en lo que se llama Gray Box Testing. [BDGSW07]

En el marco de nuestro trabajo utilizaremos la técnica de caja negra ya que en la etapa de creación de los casos de uso el sistema aún no ha sido implementado. Por lo tanto las pruebas consisten en valores de entrada y resultados esperados del sistema. La técnica de caja negra fue tratada en el capítulo de estado del arte.

Las variables operacionales se especifican al momento de crear los pasos en el caso de uso. La información asociada a estas variables es la que está representada en el meta-modelo de casos de uso a través del concepto *Parameter*.

Luego, al construir los casos de prueba, éstas variables estarán asociadas a los pasos del caso de prueba junto a valores que pueden tomar.

3.3.3.1 Técnica utilizada

La información almacenada en el caso de uso para una variable operacional, es utilizada para generar valores de prueba que serán agregados al caso de prueba.

A continuación describimos la técnica utilizada para generar valores dependiendo del tipo de dato utilizado y la condición aplicada sobre el mismo.

Tipo de dato numérico

Las restricciones soportadas sobre un parámetro de este tipo son de la forma “>” , “>=”, “<”, “<=”, “=” e “in(...)”.

En la tabla siguiente se muestra los valores generados automáticamente para las categorías de válido e inválido que puede tomar una variable operacional. Los valores son generados a partir de la condición impuesta a la variable.

Condición	**Valores válidos**	**Valores inválidos**
ParamID > Value	Value+1	Value, Value-1
ParamID < Value	Value-1	Value, Value+1
ParamID >= Value	Value, Value+1	Value-1
ParamID <= Value	Value, Value-1	Value+1
ParamID = Value	Value	Value-1
ParamID in(Value1, Value2...)	Value1, Value2, ...	Value notIn (Value1, value2...)

Tabla 3: Generación de valores límite para enteros

En el caso de que el parámetro sea definido como entero positivo, también se pueden inferir como valores de prueba inválidos a el "0" y algún valor negativo, por ejemplo el "-1".

Además de los valores generados automáticamente para las categorías "Válida" e "Invalida", se agregan a dichas categorías todos los valores singulares que el usuario haya ingresados cuando describió el caso de uso. Este mismo criterio de inclusión de valores singulares, se aplica en los tipos de datos que veremos a continuación.

Tipo de dato string

Las restricciones soportadas son la condición de "=" (igualdad) entre strings y la función "*length(string)*" que retorna el largo del string. El resultado de esta función será utilizada con las condiciones existentes sobre valores numéricos.

A continuación vemos como son generados los valores de pruebas para este tipo de dato y las distintas condiciones soportadas.

Condición	**Valores válidos**	**Valores inválidos**
ParamID = StringValue	StringValue	Not StringValue
length(ParamID) > Value	*String(Value)+char	*String(Value), String(Value-1)
Idem resto de expresiones de comparación numéricas con length(ParamID) en el lado izquierdo.		

Tabla 4: Generación de valores para cadenas de caracteres

La función String(Value) crea un string con cantidad de caracteres igual a Value. o sea, la cadena de caracteres generada cumple que length(String(Value)) = Value

Tipo de dato enumerado

Las condiciones sobre un parámetro de este tipo serán la de igualdad “=” y la de inclusión “in(...)”.

En cada una de estas expresiones, el identificador de parámetro es incluido del lado izquierdo de la expresión y los valores concretos serán indicados del lado derecho de la expresión.

Condición	**Valores válidos**	**Valores inválidos**
ParamID = EnumValue	EnumValue	OtherEnumValue
ParamID in(EnumValue1,EnumValue2,...)	EnumValue1	EnumValue notIn (EnumValue1, EnumValue2,...)

Tabla 5: Generación de Valores para enumerados

Capítulo 4 Caso de estudio

En este capítulo se presenta un caso de estudio donde se aplica el trabajo realizado en la sección anterior. El objetivo es construir casos de prueba ejecutables a partir de casos de uso de un sistema concreto.

La realidad planteada consiste en un sistema ATM (Automated Teller Machine) o cajero automático que es un sub-sistema bancario que provee acceso a transacciones financieras a los clientes del banco en un ambiente público sin restricciones de horario ni la necesidad de personal de operaciones.

Los clientes utilizan el cajero automático para controlar el balance de cuentas bancarias, depositar fondos, retirar dinero, consultar información de cuentas o transferir fondos entre otras acciones.

En los ATM el cliente se autentica insertando una tarjeta plástica e ingresado un numero de identificación personal PIN.

Los casos de uso considerados en este estudio son Validar usuario, Retiro de dinero, Consulta de saldo y Cambio de PIN.

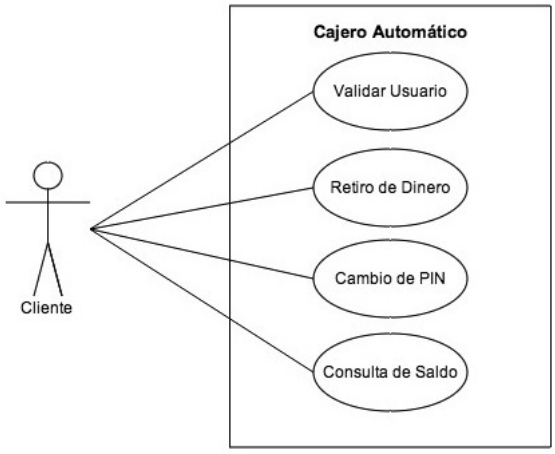

Figura 22: Diagrama de casos de uso UML

La información descrita en los casos de uso puede ser modelada utilizando el meta-modelo propuesto en el Capítulo 3.

El caso de uso Validar usuario describe las acciones que realiza el sistema para permitir el acceso a las operaciones financieras a un cliente determinado.

El caso de uso Retiro de dinero describe la interacción de un cliente con el cajero para retirar dinero de una cuenta bancaria.

El caso de uso Consulta de saldo consiste en las acciones que realiza un cliente para consultar el saldo de una cuenta bancaria.

El caso de uso Cambio de PIN permite a un cliente cambiar su numero de identificación personal.

Los casos de uso se presentan en forma textual, luego se muestra el diagrama de actividad que se genera a partir de ellos y finalmente se describen los casos de prueba construidos detallando su formato y la información que contienen.

4.1 Caso de Uso: Retiro de dinero

En la siguiente tabla se presenta el caso de uso Retiro de dinero.

Nombre	Retiro de dinero
Descripción	El caso de uso describe el retiro de dinero del cajero por parte de un cliente
Sistema bajo prueba	Cajero Automático
Actores	Usuario
Pre-condiciones	El usuario ingresa una tarjeta valida
Post-condiciones	El usuario retira el dinero y el monto es debitado de su cuenta. Se registra la extracción en el sistema
Secuencia principal de pasos	
Retiro exitoso *Describe la secuencia de pasos necesarias que un usuario deber realizar para retirar dinero en forma exitosa.* 1) El cajero despliega las distintas opciones disponibles y el cliente selecciona "Retirar Dinero" 2) El cajero solicita que cuenta utilizar para realizar la extracción 3) El cliente selecciona una cuenta para realizar el retiro de dinero. 4) El cajero pregunta el monto a retirar 5) El cliente ingresa el monto a retirar. El monto debe ser menor o igual que $500 **{(input: monto, restriction: monto <= 500, valid)}** 6) El cajero verifica el monto ingresado. Verifica que el monto disponible es mayor que el monto ingresado. **{(state: disponibleATM, restriction: disponibleATM >= &monto, valid)}** 7) El cajero envía los datos de monto y numero de cuenta al banco y este confirma si el usuario excede o no el límite diario de extracción para su cuenta. **{(output: limiteDiario, restriction: limiteDiario >= &monto, valid)}** 8) El cajero dispensa el monto de dinero solicitado al cliente 9) El cajero devuelve la tarjeta al cliente 10) El cajero imprime un recibo como comprobante de la extracción.	
Secuencias alternativas	
Monto incorrecto (branchCondition: monto > 500) *El cliente ingresa un monto mayor que el máximo permitido para la extracción* Pasos: 6a) El cajero despliega un mensaje de error diciendo que el monto ingresado no es correcto. El caso de uso vuelve al paso 4 del flujo básico **Dinero en ATM insuficiente** (branchCondition: disponibleATM < monto) *El dinero disponible en el ATM no es suficiente para entregar el monto de dinero solicitado por el usuario* Pasos: 7a) El cajero despliega un mensaje de error indicando que el cajero no tiene suficiente dinero disponible para realizar la operación. El caso de uso vuelve al paso 4 del flujo básico **Monto excede límite diario** (branchCondition: limiteDiario < monto) *El monto solicitado excede el límite diario disponible para retiro en la cuenta del usuario* Pasos: 8a) El cajero despliega un mensaje de error diciendo que el monto excede el límite diario a retirar. El caso de uso vuelve al paso 4 del flujo básico	

Tabla 6: Descripción del caso de uso "Retiro de dinero"

4.1.1 Diagrama de actividad

En la siguiente imagen (Figura 23) se muestra el diagrama de actividad generado a partir del caso de uso mediante la aplicación de los algoritmos construidos en el proyecto.

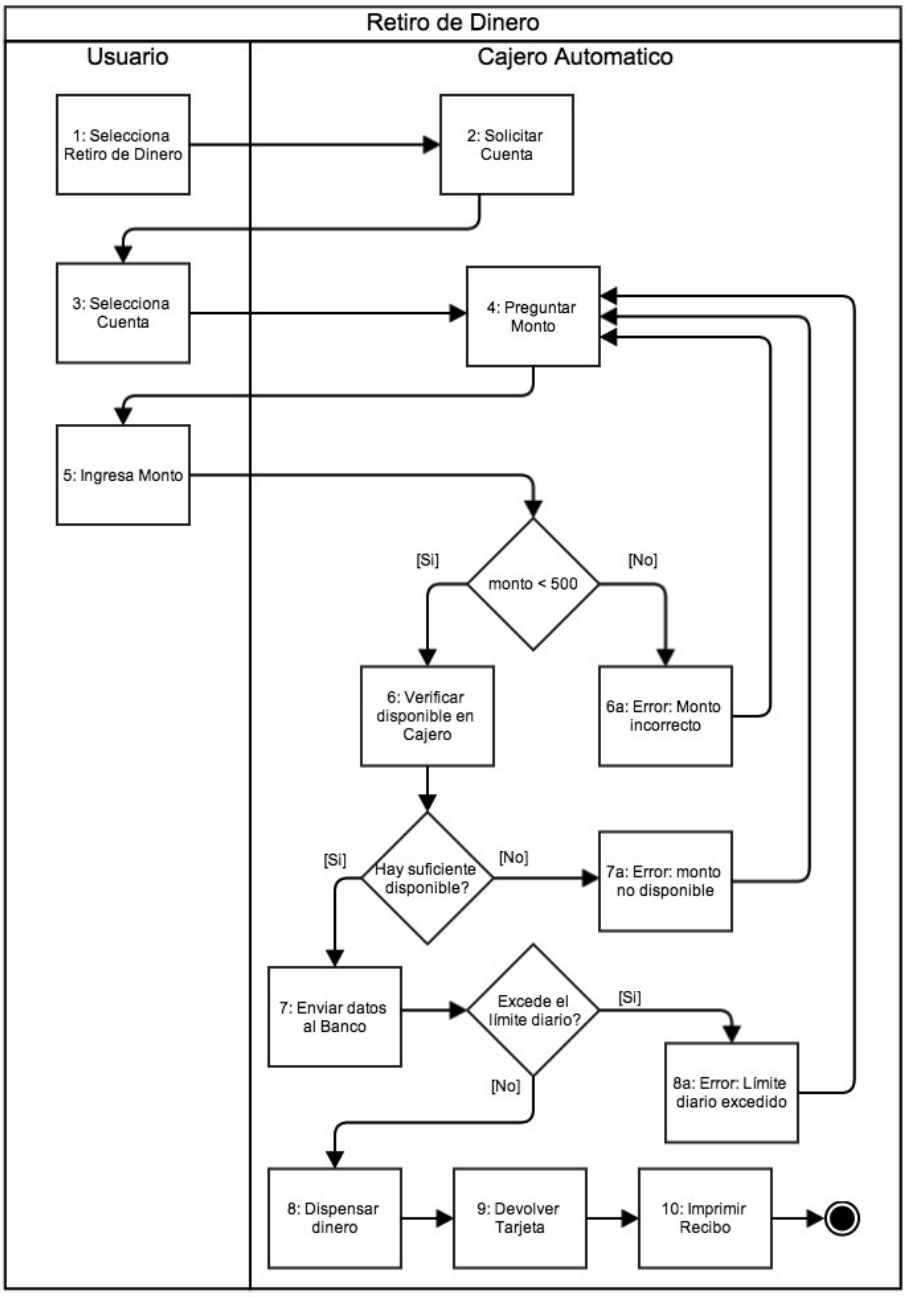

Figura 23: Diagrama de actividad para Retiro de dinero

Cada uno de los pasos que forman los escenarios del caso de uso son transformados en instancias de la clase *Action* del diagrama de actividad. Estás instancias son representadas en la figura a través de cajas. El sistema y los actores que llevan a cabo acciones en el caso de uso se representan dentro de una partición *ActivityPartition.*

A modo de ejemplo, el paso 5 en el caso de uso, es transformado en una acción del diagrama de actividad llamada “Ingresa Monto” y se ubica en la partición de acciones correspondientes al Usuario.

De forma similar, el paso 10 se transforma en una acción llamada “Imprimir recibo” que se ubica en la partición de acciones llevadas a cabo por el Cajero Automático o sistema.

El paso 10, al ser el último de la secuencia, se conecta con un nodo de fin *ActivityFinalNode* en el diagrama de actividad

En los pasos donde se producen escenarios alternativos, como por ejemplo en el paso 6 donde el usuario ingresa un monto incorrecto, se genera un nodo de decisión *DecisionNode* en el diagrama de actividad.

Este nodo tiene como entrada la acción correspondiente al paso anterior (paso 5) y sus salidas se conectan con los nodos correspondientes con el paso 6 y su alternativo 6a. Las salidas posibles en una decisión son, o bien, el siguiente paso de la secuencia, o bien el primer paso de un escenario alternativo.

El último paso de un escenario alternativo puede volver a un paso anterior de la secuencia básica, esto se observa en el paso 6a donde la salida se conecta con el paso 4.

4.1.2 Casos de prueba

A partir del diagrama de actividad anterior se generan en forma automática los siguientes casos de prueba:

- **Retiro_de_dinero_TC_1**: Este caso cubre la secuencia de retiro exitoso de dinero.
- **Retiro_de_dinero_TC_2**: Este caso cubre la secuencia que comienza con retiro de dinero exitoso pero luego el monto ingresado excede el límite diario (se cumple la condición: limiteDiario < Monto). Finalmente el usuario corrige el monto y puede hacer el retiro de dinero.
- **Retiro_de_dinero_TC_3**: Este caso cubre la secuencia que comienza con retiro de dinero exitoso, pero luego monto ingresado excede el límite diario (se cumple la condición: limiteDiario < Monto), a continuación el usuario corrige el monto pero el dinero disponible en el cajero es insuficiente para el monto elegido (se cumple la condición: disponibleATM < Monto), finalmente el usuario corrige el monto y puede hacer el retiro de dinero.

- **Retiro_de_dinero_TC_4**: Este caso comienza con retiro de dinero exitoso, pero luego el monto excede el límite diario (se cumple la condición: limiteDiario < Monto), a continuación el usuario corrige el monto pero el dinero disponible en el cajero es insuficiente para el monto elegido (se cumple la condición: disponibleATM < Monto), luego el usuario corrige el monto pero ingresa un monto incorrecto (se cumple que: Monto > 500), finalmente, el usuario corrige el monto y realiza el retiro de dinero.

Los casos de prueba cubren todos los caminos posibles en el diagrama, se impuso la condición que se visite al menos una vez cada nodo del diagrama de actividad para asegurar que cada nodo de decisión se evalúe al menos una vez verdadero y una vez falso.

Además, los casos de prueba incluyen las variables operacionales que sirven de referencia para la persona que ejecuta las pruebas.

Los valores de prueba para las variables operacionales son generados utilizando la técnica descrita en el Capítulo 3 sección 3.3.

Los valores singulares, aquellos que no son generados automáticamente pero que fueron descritos en el caso de uso, también se incorporan en los casos de prueba.

En la Tabla 7 se muestra la representación del caso de prueba Retiro_de_dinero_TC_1.

Name	Retiro_de_dinero_TC_1									
Test Trail	Retiro Exitoso									
Preconditions	El usuario ingresa una tarjeta valida									
Steps										
				Operational Variables						
ID	Name	Description	Type	Data Type	Name	Condition	Test values		Expected Result	
							Generated	Singular		
1	Mostrar opciones	El cajero despliega las distintas opciones disponibles y el cliente selecciona "Retirar Dinero"								
2	Solicitar cuenta	El cajero solicita que cuenta utilizar para realizar la extracción								
3	Cliente selecciona cuenta	El cliente selecciona una cuenta para realizar el retiro de dinero.								
4	Preguntar monto	El cajero pregunta el monto a retirar								
5	Ingresar monto	El cliente ingresa el monto a retirar. El monto debe ser menor o igual que $500	input	unsignedint	monto	<= 500	0, 500, 499	100		
6	Verificar monto	El cajero verifica el monto ingresado. Verifica que el monto disponible es mayor que el monto ingresado.	state	integer	disponibleATM	>= &monto	&monto, &monto+1			
7	Enviar datos al banco	El cajero envía los datos de monto y numero de cuenta al banco y este confirma si el usuario excede o no el límite diario de extracción para su cuenta.	output	integer	limiteDiario	>= &monto	&monto, &monto+1			
8	Dispensar dinero	El cajero dispensa el monto de dinero solicitado al cliente							El disponible del cajero pasa a ser: disponibleATM = disponibleATM - monto Además actualiza el limiteDiario que le queda para extraer al cliente haciendo: limiteDiario = limiteDiario - monto	
9	Devolver tarjeta	El cajero devuelve la tarjeta al cliente								
10	Imprimir recibo	El cajero imprime un recibo como comprobante de la extracción.								
11	Fin del caso de uso	El caso de uso termina								
Expected Results		El usuario retira el dinero y el monto es debitado de su cuenta. Se registra la extracción en el sistema								

Tabla 7: Caso de prueba generado para Retiro de dinero

4.2 Caso de Uso: Validar usuario

En la siguiente tabla se describe el caso de uso Validar usuario.

Nombre	Validar usuario
Descripción	El caso de uso describe el comportamiento del ATM para validar un usuario para poder operar
Sistema a probar	Cajero Automático
Actores	Usuario
Pre-condiciones	El cajero ATM está conectado al banco
Post-condiciones	El cliente es validado en el sistema. El usuario queda logueado en el cajero automático y puede continuar con las acciones que se le ofrezcan
Secuencia principal de pasos	
Validación Exitosa *Este escenario describe la secuencia de pasos que se deben cumplir para que el usuario sea validado en el cajero y quede logueado para poder operar en el mismo.* 1) El caso de uso comienza cuando el cliente del banco inserta su tarjeta en el cajero 2) El cajero lee el código de la tarjeta y envía los datos al banco. El banco verifica que el código sea válido**{(input: codigo, restriction: codigo length = 12, valid)}** 3) El cajero solicita al cliente que ingrese su PIN 4) El cajero valida el pin y termina el caso de uso **{(input: pin, restriction: pin range [1000,9999], valid)}** 5) El caso de uso termina cuando el cajero despliega las distintas opciones que están disponibles en la máquina	
Secuencias alternativas	
PIN invalido (branchCondition: PIN invalido) *El PIN ingresado por el usuario no es correcto y el sistema ofrece reintentar el ingreso del mismo.* Pasos: 5a) El cajero despliega el mensaje “Pin invalido” y vuelve al paso 3 para solicitar nuevamente el PIN al usuario **Código de tarjeta inválido** (branchCondition: lenght(codigo) <> 12 OR system.tarjetas.exist(codigo) == false) *Este escenario se ejecuta cuando la tarjeta ingresada al cajero no es valida en el sistema, ya sea porque el código no cumple con las restricciones de la definición del mismo o porque el código no es encontrado en el sistema.* Pasos: 3a) El cajero despliega el mensaje “Tarjeta inválida” , devuelve la tarjeta al usuario y el proceso termina sin que el usuario sea validado.	

Tabla 8: Descripción del caso de uso “Validar usuario”

4.2.1 Diagrama de Actividad

El diagrama de actividad generado a partir del caso de uso anterior se muestra en la siguiente imagen (Figura 24).

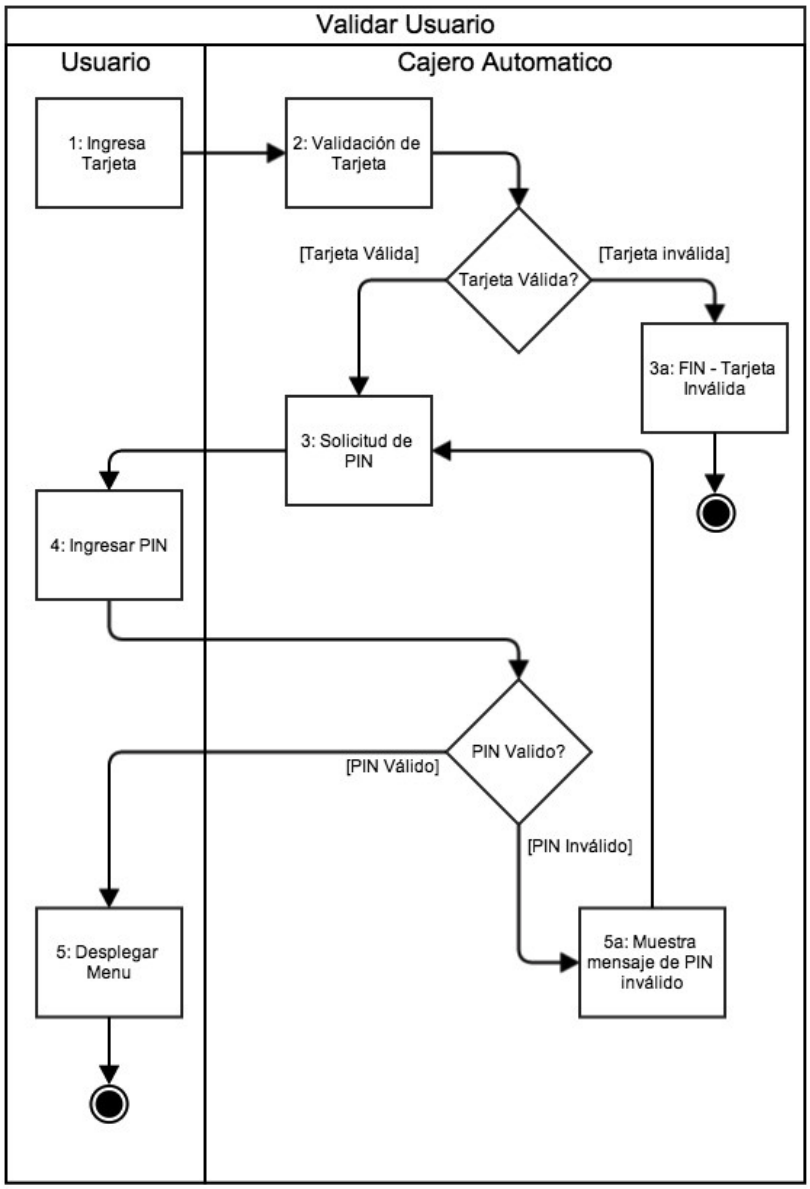

Figura 24: Diagrama de actividad para Validar usuario

Los nodos del diagrama de actividad son generados en forma similar al caso de uso anterior.

El escenario alternativo cuando la tarjeta es inválida termina sin retornar a un nodo del flujo principal. La última acción de este escenario está conectada a un nodo de finalización, correspondiente al paso 3a.

4.2.2 Casos de prueba

Los casos de prueba generados para el caso de uso anterior cubren todos los caminos del diagrama de actividad y evalúan todas las condiciones en los nodos de decisión:

- **Validar_usuario_TC_1**: Este caso cubre la secuencia de validación exitosa, cuando el usuario ingresa una tarjeta válida y su identificador PIN correcto en el sistema.
- **Validar_usuario_TC_2**: Este caso comienza en la secuencia de validación exitosa, luego el usuario ingresa un número PIN inválido y el sistema solicita nuevamente el número de PIN. Finalmente, el usuario ingresa el número correcto y el caso termina con la secuencia de validación exitosa.
- **Validar_usuario_TC_3**: Este caso comienza en la secuencia de validación exitosa pero continúa por el escenario de código de tarjeta inválido (se cumple la condición de: lenght(código) <> 12 OR system.tarjetas.exist(código) == false) y el caso de uso termina en forma anormal y el usuario no es validado.

En la Tabla 9 se muestra la representación del caso de prueba Validar_usuario_TC_3.

Name	Validar_usuario_TC_3
Test Trail	Validación Exitosa -> Código de tarjeta inválido (condition: lenght(codigo) <> 12 OR system.tarjetas.exist(codigo) == false)
Preconditions	El cajero ATM está conectado al banco

Steps

ID	**Name**	**Description**	**Operational Variables**						**Expected Result**
			Type	**Data Type**	**Name**	**Condition**	**Test values**		
							Generated	**Singular**	
1	Insertar Tarjeta	El caso de uso comienza cuando el cliente del banco inserta su tarjeta en el cajero							
2	Validación Tarjeta	El cajero lee el código de la tarjeta y envía los datos al banco. El banco verifica que el código sea válido	input	string	codigo	LENGTH() <> 12	5LOG7I4Z9QK, CJQY6VNCUK5EV	AAAASSSSDDDD	
3a	Tarjeta Inválida	El cajero despliega el mensaje “Tarjeta inválida” , devuelve la tarjeta al usuario y el proceso termina sin que el usuario sea validado.							

Expected Results	El caso de uso termina anormalmente. La tarjeta es devuelta al usuario. El usuario no es logueado al sistema. El cajero vuelve a su estado inicial

Tabla 9: Caso de prueba generado para Validar usuario

4.3 Caso de Uso: Cambio de PIN

Nombre	Cambio de PIN
Descripción	El caso de uso describe las interacciones entre el Cajero Automático y el Usuario para poder cambiar el PIN que utiliza para poder operar con su tarjeta.
Sistema a probar	Cajero Automático
Actores	Usuario
Pre-condiciones	El cajero ATM está conectado al banco. El usuario ingresó una tarjeta válida
Post-condiciones	El PIN del cliente es cambiado en el sistema. El PIN nuevo es distinto al PIN anterior
Secuencia principal de pasos	
Cambio de PIN exitoso *Describe los pasos que el usuario realiza para cambiar exitosamente el PIN de su tarjeta de cajero.* 1) El caso de uso comienza cuando el cliente del banco selecciona la opción de cambiar PIN 2) El cajero solicita al cliente que ingrese su PIN actual. 3) El cliente ingresa su PIN antes de poder cambiar el PIN **{(input: pinActual, restriction: pinActual range [1000,9999], valid)}** 4) El cajero solicita al cliente que ingrese su nuevo PIN. 5) El cliente ingresa su nuevo PIN. **{(input: nuevoPin, restriction: nuevoPin range [1000,9999], valid)}** 6) El cajero compara y valida que el nuevo PIN ingresado por el usuario sea distinto al PIN anterior **{(state: nuevoPin, restriction: nuevoPin = &pinActual, invalid)}** 7) El cajero solicita al cliente que confirme su nuevo PIN. 8) El cliente debe ingresar su nuevo PIN para confirmar el cambio **{(input: confirmaNuevoPin, restriction: confirmaNuevoPin = &nuevoPin, valid)}** 9) El cajero muestra un mensaje de PIN cambiado exitosamente y el caso de uso termina	
Secuencias alternativas	
PIN Inválido (branchCondition: PIN Inválido) *Este escenario se ejecuta cuando el PIN ingresado por el usuario no corresponde con el PIN de la tarjeta* Pasos: 4a) El cajero despliega el mensaje “Pin invalido”. El caso de uso vuelve al paso 2 **PIN nuevo inválido** (branchCondition: nuevoPin inválido) *El escenario cubre el caso en que el formato del nuevo PIN ingresado no cumple con las restricciones de la variable operacional* Pasos: 6a) El cajero despliega un mensaje al usuario diciendo "Formato de nuevo Pin inválido. Por favor intente nuevamente". El caso continúa en el paso 4 **PIN nuevo igual al anterior** (branchCondition: nuevoPin = pinActual) *Es el escenario que se ejecuta cuando el usuario ingresa un nuevo PIN que es igual al PIN anterior.* Pasos: 7a) El cajero despliega el mensaje “El nuevo PIN debe ser diferente al anterior”. El caso de uso vuelve al paso 4 **PIN nuevo no coincide** (branchCondition: nuevoPin <> confirmaNuevoPin) *El cajero despliega el mensaje “Pin invalido”. El caso de uso vuelve al paso 4* Pasos: 9a) El cajero despliega el mensaje “El nuevo PIN y su confirmación no coinciden. Por favor intente nuevamente”. El caso de uso vuelve al paso 4	

Tabla 10: Descripción del caso de uso “Cambio de PIN”

En este caso de estudio, podemos ver la particularidad de que en el paso 6 del caso de uso se define nuevamente la variable operacional *nuevoPin* como un parámetro de tipo *state* con el objetivo de poder utilizar el valor ingresado por el usuario para hacer una comparación en un paso siguiente del caso de uso.

4.3.1 Diagrama de Actividad

A continuación vemos el diagrama de actividad generado para el caso de uso anterior (Figura 25).

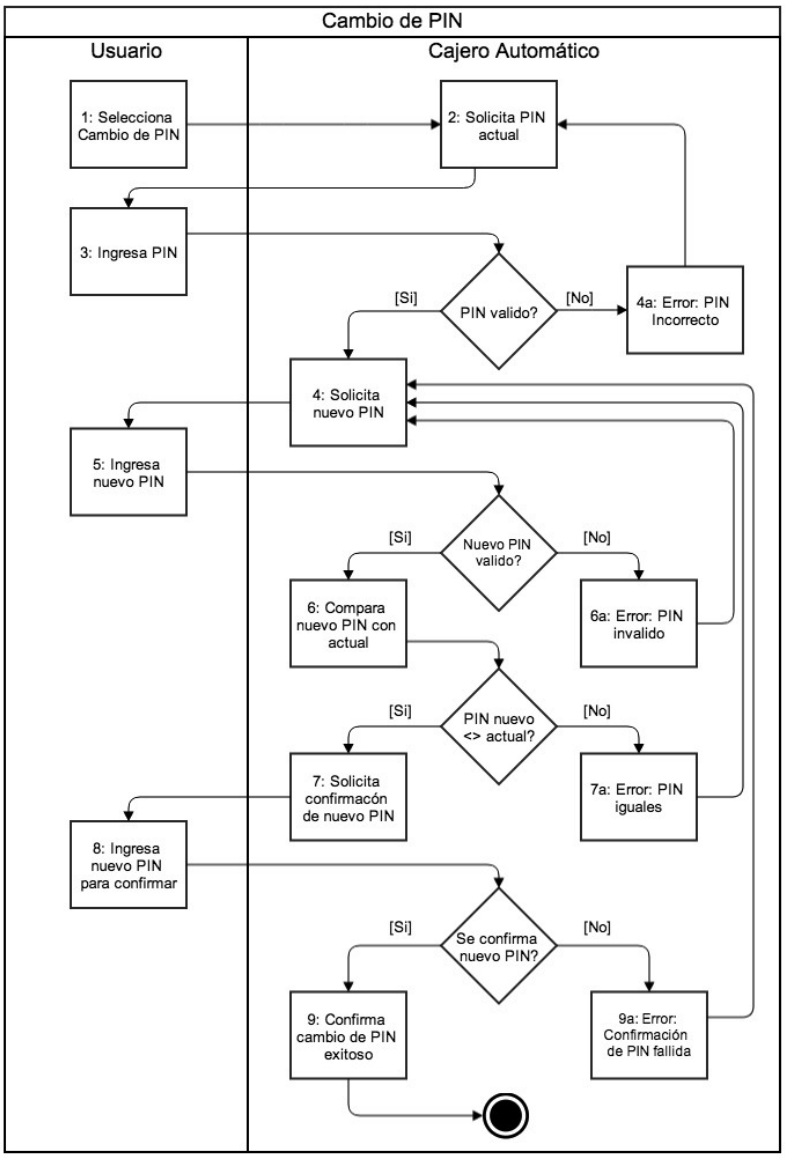

Figura 25: Diagrama de actividad generado para Cambio de PIN

4.3.2 Casos de Prueba

Los casos de prueba generados para el caso de uso se muestran a continuación:

- **Cambio_de_PIN_TC_1**: El caso cubre la secuencia de cambio de PIN exitoso.
- **Cambio_de_PIN_TC_2**: El caso comienza en la secuencia de cambio de PIN exitoso, luego al momento de validar el nuevo PIN no coincide (se cumple la condición: *nuevoPin* <> *confirmaNuevoPin*) y finalmente el usuario vuelve a ingresar un nuevo pin y lo hace de forma exitosa.
- **Cambio_de_PIN_TC_3**: Este caso de prueba cubre un cambio de PIN exitoso pero que previamente en el primer intento la validación del PIN nuevo no coincide (condición: *nuevoPin* <> *confirmaNuevoPin*), en el segundo intento ocurre que el nuevo PIN ingresado es igual al PIN anterior (condición: *nuevoPin* = *pinActual*), por lo que vuelve a intentar y finalmente realiza el cambio de PIN con éxito.
- **Cambio_de_PIN_TC_4**: Este caso de prueba, similar al anterior, cubre un Cambio de PIN exitoso pero que previamente en el primer intento de validación, el PIN nuevo no coincide (condición: *nuevoPin* <> *confirmaNuevoPin*), en el segundo intento ocurre que el nuevo PIN ingresado es igual al PIN anterior (condición: *nuevoPin* = *pinActual*), luego en un tercer intento el nuevo PIN ingresado tiene un formato inválido, por lo que el usuario vuelve a intentar y finalmente realiza el cambio de PIN con éxito.
- **Cambio_de_PIN_TC_5**: Este caso de prueba cubre la secuencia en que el usuario ingresa mal su PIN antes de hacer el cambio, por lo que el cajero solicita nuevamente el ingreso del PIN y, al hacerlo en forma correcta, continúa con el cambio de PIN en forma exitosa.

A continuación, en la Tabla 11 se muestra la representación del caso de prueba generado **Cambio_de_PIN_TC_1**, que cubre el cambio de PIN exitoso, y además en el paso 6 se puede apreciar el uso de la variable operacional *nuevoPin* para ser comparada con el *pinActual*.

Name	Cambio_de_PIN_TC_1
Test Trail	Cambio de PIN exitoso
Preconditions	El cajero ATM está conectado al banco. El usuario ingresó una tarjeta válida

Steps

ID	Name	Description	Operational Variables						Expected Result
			Type	Data Type	Name	Condition	Test values		
							Generated	Singular	
1	Selecciona Cambiar PIN	El caso de uso comienza cuando el cliente del banco selecciona la opción de cambiar PIN							
2	Solicita PIN Actual	El cajero solicita al cliente que ingrese su PIN actual.							
3	Ingresa PIN	El cliente ingresa su PIN antes de poder cambiar el PIN	input	unsignedint	pinActual	RANGE [1000,9999]	1001, 1000, 9998, 9999, 5499		
4	Solicita nuevo PIN	El cajero solicita al cliente que ingrese su nuevo PIN.							
5	Ingresa nuevo PIN	El cliente ingresa su nuevo PIN.	input	unsignedint	nuevoPin	RANGE [1000,9999]	1001, 1000, 9998, 9999, 5499	5555	
6	Compara PIN nuevo y actual	El cajero compara y valida que el nuevo PIN ingresado por el usuario sea distinto al PIN anterior	state	unsignedint	nuevoPin	<> &pinActual	&pinActual+1, &pinActual-1		
7	Solicita confirmación de nuevo PIN	El cajero solicita al cliente que confirme su nuevo PIN.							
8	Confirmar nuevo PIN	El cliente debe ingresar su nuevo PIN para confirmar el cambio	input	unsignedint	confirmaNuevoPin	= &nuevoPin	&nuevoPin		
9	Confirma cambio exitoso	El cajero muestra un mensaje de PIN cambiado exitosamente y el caso de uso termina							

Expected Results	El PIN del cliente es cambiado en el sistema. El PIN nuevo es distinto al PIN anterior

Tabla 11: Caso de prueba generado para Cambio de PIN

4.4 Caso de Uso: Consulta de saldo

Nombre	Consulta de saldo
Descripción	El caso de uso describe como el usuario interactúa con el cajero para solicitar el saldo de su cuenta en forma impresa
Sistema bajo prueba	Cajero Automático
Actores	Usuario
Pre-condiciones	El usuario ingresa una tarjeta valida
Post-condiciones	El usuario recibe un ticket con el saldo de su cuenta
Secuencia principal de pasos	
Consulta de saldo impresa exitosa *Describe la secuencia de pasos necesarias que un usuario pueda consultar el saldo de su cuenta y recibir un ticket impreso en forma exitosa.* 1) El cliente selecciona "Consulta de saldo" del menú principal 2) El cajero solicita que elija una cuenta para realizar la consulta 3) El cliente selecciona una cuenta 4) El cajero verifica si hay papel **{(state: hayPapel, restriction: hayPapel = 1, valid)}** 5) El cajero imprime un recibo con el saldo de la cuenta	
Secuencias Alternativas	
Sin Papel (branchCondition: hayPapel == 0) *El cajero no cuenta con papel para poder imprimir el estado de cuenta y consulta al usuario si desea ver en pantalla el saldo* Pasos: 5a) El cajero despliega un mensaje de error diciendo que no hay papel y consulta al usuario si desea ver el saldo de la cuenta en pantalla 6a) El usuario selecciona "SI" para ver el saldo en pantalla **{(input: mostrarSaldo, restriction: mostrarSaldo = SI, valid)}** 7a) El cajero despliega el Saldo de la cuenta en pantalla. El caso de uso termina **Usuario cancela la consulta** (branchCondition: mostrarSaldo == "NO") *El usuario opta por no ver el saldo en la pantalla del cajero y el caso de uso termina* Pasos: 7b) El cajero vuelve al menú principal y el caso de uso termina sin que el saldo haya sido mostrado en pantalla	

Tabla 12: Descripción del caso de uso "Consulta de saldo"

A diferencia de los casos de uso estudiados anteriormente, en este caso de uso se destaca la particularidad de que cuenta con escenarios alternativos anidados. Esto se puede apreciar en el escenario alternativo donde no hay papel en el cajero, en ese escenario existen dos posibles ejecuciones, en una de ellas se deriva en otro escenario alternativo cuando el usuario cancela la consulta.

Además el caso de uso contiene una finalización exitosa a través de un flujo alternativo al flujo principal, que es cuando el cajero está sin papel y el usuario opta por ver el saldo en pantalla. El objetivo de consultar el saldo se realiza, aunque no es posible imprimir el recibo. Esto hace que el caso de uso termine por una vía alternativa.

4.4.1 Diagrama de Actividad

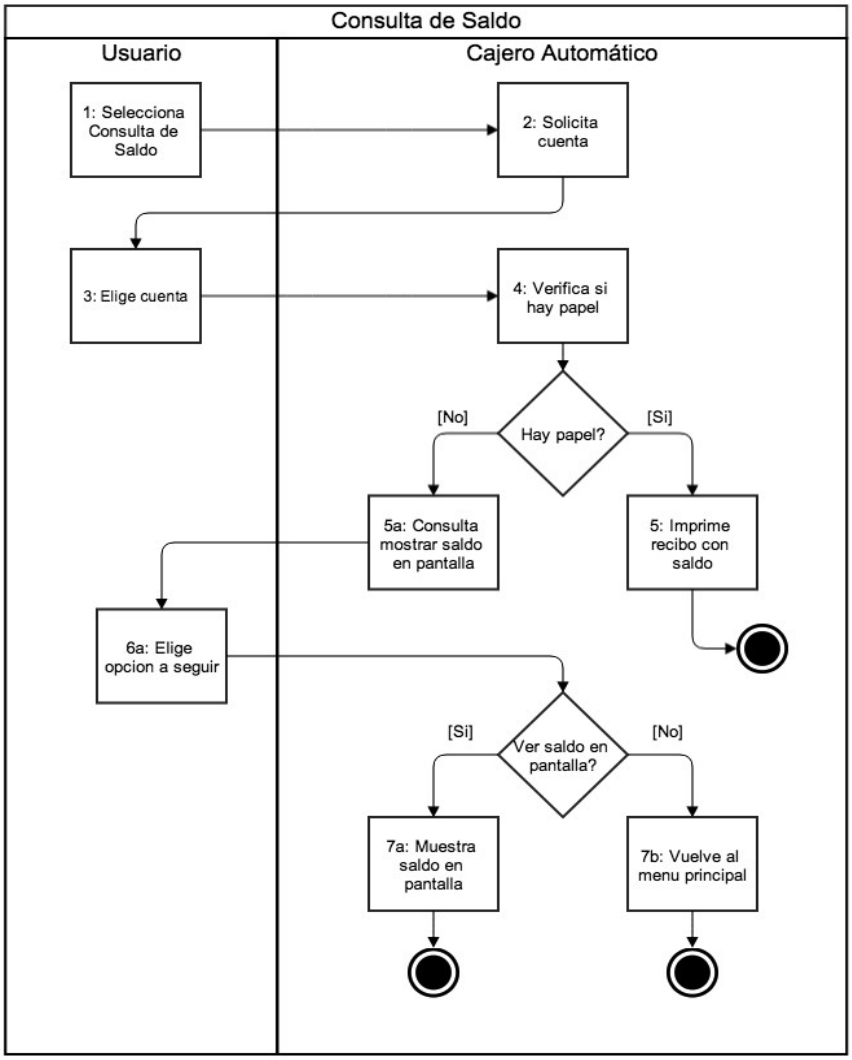

Figura 26: Diagrama de actividad generado para Consulta de saldo

En el diagrama de actividad se observan las características del caso de uso, la finalización por una vía alternativa y flujos alternativos anidados.

4.4.2 Casos de Pruebas

Los casos de prueba generados para el caso de uso se muestran a continuación:

- **Consulta_de_saldo_TC_1**: Cubre el caso básico cuando el usuario realiza la consulta de saldo y recibe la misma de forma impresa.

- **Consulta_de_saldo_TC_2**: Cubre el caso en que no hay papel en el cajero para imprimir el recibo pero el usuario selecciona imprimir el saldo en pantalla.
- **Consulta_de_saldo_TC_3**: Cubre el caso en que no hay papel en el cajero y el usuario cancela la consulta al decidir no mostrar el resultado en pantalla.

En la Tabla 13 se muestra la representación del caso de prueba Consulta_de_saldo_TC_3.

Name	Consulta_de_saldo_TC_3
Test Trail	Consulta de saldo impresa exitosa -> Sin Papel (condition: hayPapel == 0) -> Usuario cancela la consulta (condition: mostrarSaldo == "NO")
Preconditions	El usuario ingresa una tarjeta valida

Steps

ID	**Name**	**Description**	**Operational Variables**						**Expected Result**
			Type	**Data Type**	**Name**	**Condition**	**Test values**		
							Generated	**Singular**	
1	Selecciona Consulta de Saldo	El cliente selecciona "Consulta de saldo" del menú principal							
2	Solicita cuenta	El cajero solicita que elija una cuenta para realizar la consulta							
3	Elige cuenta	El cliente selecciona una cuenta							
4	Verifica papel	El cajero verifica si hay papel	state	unsignedint	hayPapel	<> 1	0, 2, "NotNumber", -1	0	
5a	Consulta mostrar saldo en pantalla	El cajero despliega un mensaje de error diciendo que no hay papel y consulta al usuario si desea ver el saldo de la cuenta en pantalla							
6a	Elije opción a seguir	El usuario selecciona "SI" para ver el saldo en pantalla	input	string	mostrarSaldo	<> SI	"S", "I", "not SI", "SI0"	NO	
7b	Vuelve al Menu	El cajero vuelve al menu principal y el caso de uso termina sin que el saldo haya sido mostrado en pantalla							

Expected Results	El saldo no fue consultado ni mostrado en pantalla

Tabla 13: Caso de prueba generado para Consulta de saldo

Capítulo 5 Implementación

El último objetivo del proyecto es la construcción de una herramienta que permita ingresar casos de uso y generar casos de prueba en forma automática. La herramienta se desarrolló como un plug-in de Eclipse y proporciona un ambiente visual de trabajo.

El estudio realizado en los capítulos anteriores son la base para la construcción de la herramienta. En ella se reflejan el meta-modelo de casos de uso, los algoritmos para generar pruebas y la representación visual de los casos de prueba.

Este capítulo comienza con una introducción a Eclipse y las posibilidades que brinda como plataforma de desarrollo. Luego se presentan las tecnologías usadas en la implementación y se describe el producto obtenido.

Las referencias del capítulo se encuentran en [SBPM08] y [CR09].

5.1 Eclipse

Eclipse es un programa compuesto por un pequeño núcleo y un cargador de extensiones. Estas extensiones son llamadas plug-ins. El núcleo es una implementación de la especificación OSGi R4 y provee un ambiente en donde los plug-ins pueden ejecutarse.[6]

Eclipse tiene un diseño modular que permite agregar partes de funcionalidad que pueden ser re-usadas para crear aplicaciones no consideradas por los creadores de Eclipse. La plataforma fue diseñada como una herramienta para el desarrollo de aplicaciones y no provee funcionalidad por si sola, en cambio, el valor se encuentra en los componentes que pueden ser integrados en base al modelo de plug-ins.

La arquitectura del núcleo de Eclipse permite descubrir en forma dinámica, cargar y ejecutar plug-ins. La plataforma maneja la logística de encontrar y ejecutar el código correcto mientras que la interfaz gráfica provee un modelo de navegación estándar.

La idea central del modelo de plug-ins es que cada uno se puede enfocar en hacer un número reducido de tareas y donde nuevas funcionalidades puedan ser agregadas sin impactar a otras existentes.

La plataforma en si está construida en capas de plug-ins, cada uno define extensiones para los puntos de extensión definidos en otros plug-ins. Y a su vez definen sus propios puntos de extensión para futuras implementaciones.

6 OSGi R4 http://www.osgi.org/Release4/HomePage

En la plataforma, cada subsistema es estructurado como un conjunto de plug-ins que implementan cierta función clave. El modelo de extensión permite que algunos plug-ins agreguen características visibles a la plataforma mientras que otros agregan librerías que pueden ser usadas para implementar extensiones al sistema.

El SDK (Software Development Kit) de Eclipse incluye, además de la plataforma básica, dos herramientas útiles para la construcción de plug-ins. Estas son JDT (Java development Tools) y PDE (Plug-in Developer Environment)

La primera provee un ambiente de desarrollo en Java y la segunda agrega herramientas específicas para el desarrollo de plug-ins y extensiones. Estas herramientas ademas de ser útiles, proveen un claro ejemplo de como nuevas herramientas pueden ser agregadas a la plataforma mediante la construcción de plug-ins que extienden el sistema.

En la Figura 27 se muestra la plataforma de Eclipse y el modelo de extensión donde nuevas herramientas pueden incorporarse al sistema.

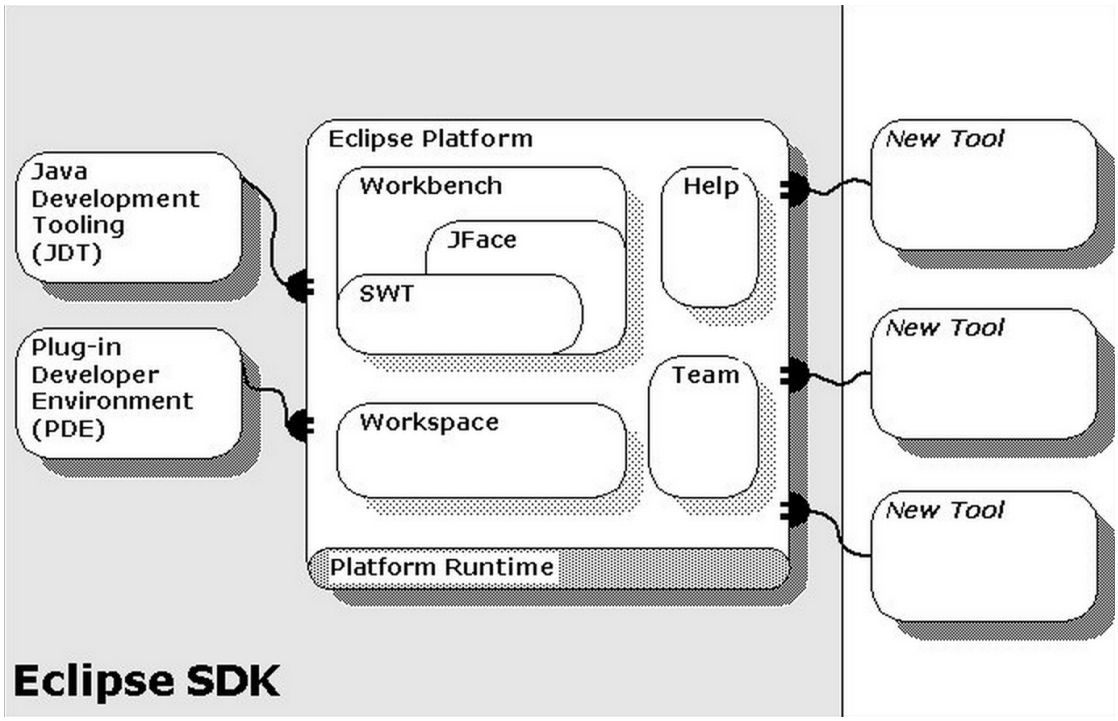

Figura 27: Plataforma de Eclipse

El framework solo define la estructura básica del IDE (Integrated Development Environment) y permite conectar herramientas específicas para definir un IDE particular.

5.1.1 Arquitectura de plug-ins

Una herramienta sencilla puede consistir de un solo plug-in mientras que otras más complejas usualmente están divididas en varios.

Cada plug-in contribuye con funcionalidad que puede ser invocada por el usuario o usada y extendida por otros plug-ins.

El ambiente de ejecución de la plataforma (runtime engine) es responsable de descubrir y ejecutar plug-ins. Está implementado sobre una plataforma de servicios OSGi que provee un mecanismo flexible y estándar que permite instalar y remover plug-ins sin reiniciar la plataforma.

Desde la perspectiva del empaquetado (packaging), un plug-in incluye todo lo que necesita para ejecutar como código Java, imágenes, textos y librerías. También incluye archivos de configuración llamados manifiestos que identifican el plug-in y proveen información para su ejecución.

Esta información incluye las dependencias de otros plug-ins, los paquetes que se exportan, los puntos de extensión que se proveen y las extensiones implementadas.

Los puntos de extensión son declaraciones de funcionalidad que están disponibles para otros plug-ins, las extensiones son las implementaciones que hace el plug-in de puntos de extensión declarados en otros.

5.1.2 La plataforma UI

El framework de interfaz de usuario de Eclipse se conoce como *Platform UI*. Esta plataforma consiste en dos conjuntos de herramientas de propósito general. Por un lado SWT y JFace y por otro un sistema de interfaz gráfica modificable (*workbench UI*).

SWT (*Standard Widget Toolkit*) es la librería de interfaz de usuario que usa Eclipse. Provee componentes como botones y campos de texto, así como un manager de layout. El manager de layout es usado para arreglar la disposición de los componentes en la pantalla de acuerdo a un conjunto de reglas.

El objetivo de diseño de SWT es estar lo más cerca posible del sistema operativo, esto significa que SWT utiliza componentes nativos siempre que sea posible y por lo tanto el aspecto de las aplicaciones se adaptan al sistema operativo donde ejecutan.

JFace es una herramienta de alto nivel implementada usando SWT. Provee clases que permiten programar tareas comunes en entornos UI como el registro de imágenes y fuentes, *dialog boxes*, *wizards*, *progress monitors* y más. La API de JFace no encapsula SWT sino que trabaja con el y lo expande.

Entre los componentes de JFace se destacan un conjunto de clases para trabajar en las vistas (*viewers*). Estas clases son útiles para trabajar con listas, árboles y tablas. En su implementación usan componentes de SWT pero proveen un nivel de abstracción más alto para la conexión con los datos que despliegan.

El workbench *UI*, también conocido como el escritorio de Eclipse, es la ventana principal en la que interactúan los usuarios. Este es implementado usando SWT y JFace.

Una ventana del *workbench* está compuesta por editores (*editors*) y vistas (*views*). Los editores están integrados en la ventada del *workbench* mientras que las vistas se lanzan en su propia ventana.

Las vistas son usadas para presentar información sobre los contenidos del editor activo o sobre un objeto seleccionado en un editor u otra vista.

Los editores y las vistas, cuando están activos, pueden contribuir con acciones en los menús y barras de herramientas del *workbench*.

El mecanismo principal para extender la plataforma de Eclipse consiste en implementar los puntos de extensión provistos por el *workbench*.

Los puntos de extensión permiten que herramientas agreguen nuevos editores y vistas en el *workbench.*

5.1.3 Tecnologías utilizadas en la implementación

5.1.3.1 EMF

El proyecto EMF (*Eclipse Modeling Framework*) es un framework de modelado y generación de código que facilita la construcción de herramientas y aplicaciones en base a un modelo de datos estructurados.[7]

A partir de un modelo especificado en formato XMI (*XML Metadata Interchange*), EMF provee herramientas y un ambiente de ejecución para producir un conjunto de clases Java para el modelo y un conjunto de clases *adapter* que habilitan la creación de vistas y edición del modelo a través de comandos.

EMF puede ser usado para modelar el modelo de dominio. Se distingue el meta-modelo del modelo concreto. El meta-modelo describe la estructura del modelo. EMF brinda un framework para guardar la información del modelo que por defecto utiliza XMI para persistir esta información.

EMF permite crear el meta-modelo a través de diferentes métodos como XMI, anotaciones en clases Java, UML o esquemas XML.

7 Eclipse Modeling Framework http://www.eclipse.org/modeling/emf/

Una vez que el meta-modelo EMF es especificado se pueden generar clases en código Java que corresponden a las clases del modelo.

EMF permite generar plug-ins que proveen ayuda para crear instancias del modelo así como un editor gráfico para ingresar la información del modelo.

5.1.3.2 EEF

El proyecto EEF (*Extended Editing Framework*) provee una mejora a EMF en cuanto a la generación del editor del modelo a través de servicios dedicados a facilitar el uso de este.[8]

El framework provee componentes para la editar de objetos y propiedades del modelo generado por EMF.

El principal objetivo de EEF es hacer más amigable e intuitivo el uso del editor del modelo que genera EMF.

5.1.3.3 Papyrus

El proyecto Papyrus es un componente de software libre que se integra en Eclipse a través del mecanismo de plug-ins y que provee un ambiente para editar diagramas UML.[9]

Papyrus busca proveer un ambiente integrado para editar toda clase de modelos EMF y particularmente UML. Provee un editor de diagramas para lenguajes de modelado basados en EMF, entre ellos UML2.

5.2 Desarrollo de la herramienta TestCaseTool

Luego de investigar las posibilidades que ofrece Eclipse para crear herramientas que se incorporan en el IDE, nos enfocamos en construir un plug-in que implementa una solución para los estudios realizados en el proyecto.

La herramienta construida es una extensión de un editor de Eclipse que permite ingresar la información de los casos de uso de acuerdo al meta-modelo construido y generar casos de prueba en forma automática.

El primer paso en el desarrollo fue utilizar *Papyrus* para crear el meta-modelo de casos de uso en forma visual a través de un editor gráfico.

8 Extended Editing Framework http://www.eclipse.org/modeling/emft/?project=eef
9 Proyecto Papyrus http://www.eclipse.org/papyrus/

El meta-modelo utilizado para la implementación es una adaptación del propuesto en el capítulo 3 en base a como los usuarios ingresarían la información de los casos de uso y la aplicación de los algoritmos para la generación de los casos de prueba.

Las adaptaciones al meta-modelo consisten en agregar las relaciones de la clase *Parameter* hacia *Validity* y *Datatype* como atributos de clase, así como implementar *ActionStep* como subtipo de la clase *Step*. También se omitió la clase *Condition* donde todas las relaciones con ésta fueron agregadas como atributos en las clases correspondientes.

En la Figura 28 se muestra el meta-modelo utilizado.

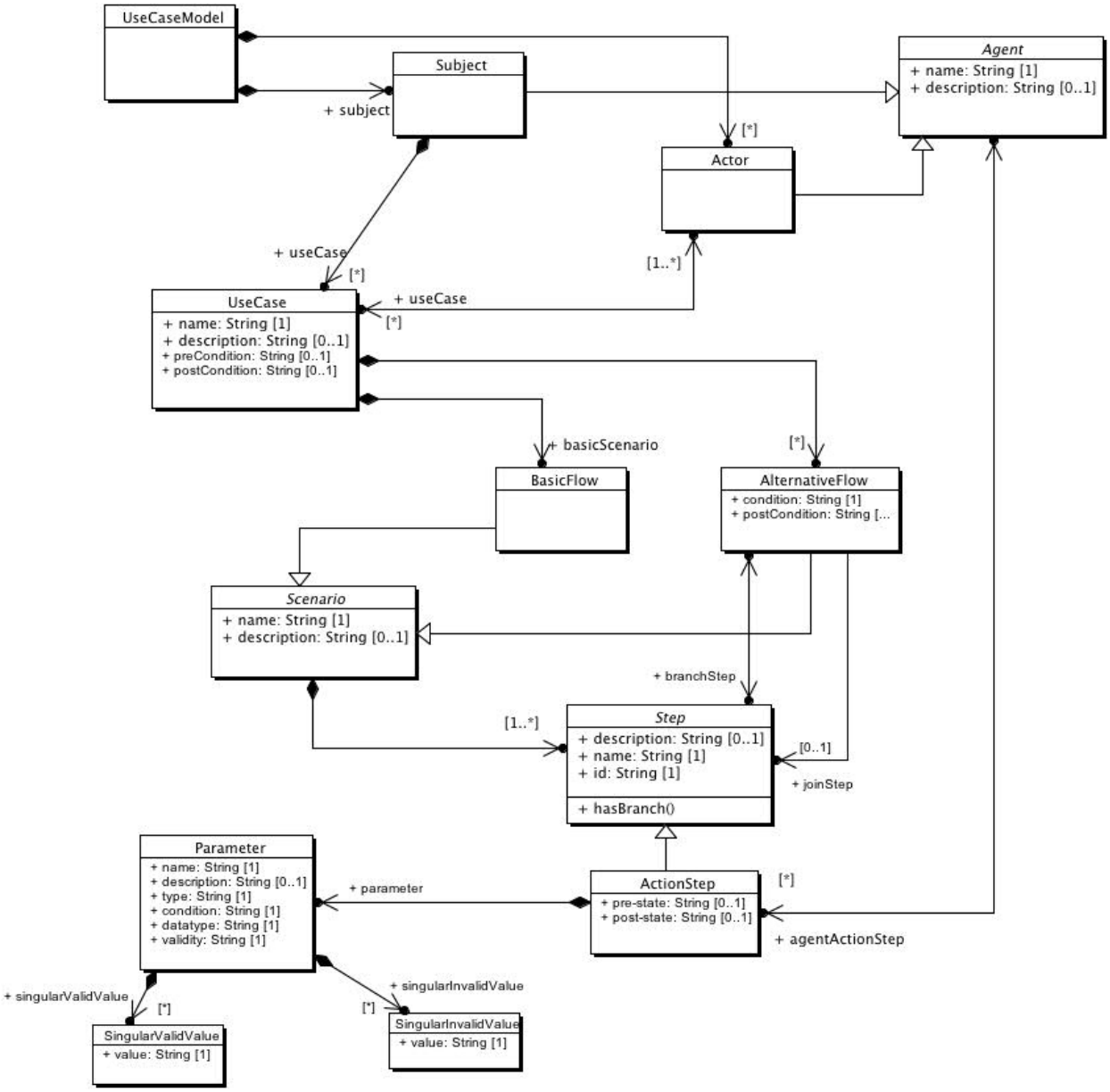

Figura 28: Meta-modelo de casos de uso para implementación

El resultado de este paso es un archivo con la representación en UML del meta-modelo de casos de uso. El archivo que contiene el meta-modelo se llama *testcasemodel.uml* y está disponible en el repositorio del proyecto.

El siguiente paso en el desarrollo consistió en crear un editor gráfico integrado en Eclipse para ingresar la información representada en el meta-modelo. La herramienta utilizada para esto fue EMF donde se aprovecharon las facilidades para generar código a partir del meta-modelo.

El archivo UML construido en el paso anterior fue la entrada a utilizada en EMF para crear las clases Java correspondientes al modelo. El resultado de esta generación es un plug-in que contiene las clases del dominio. El plug-in generado tiene el nombre *testcasetool.model.*

Luego, se utilizó EMF para generar clases *adapter* que permiten editar las clases del modelo y dan soporte para desplegar información en las vistas y comandos para edición. El plug-in generado tiene el nombre *testcasetool.edit.*

EMF también permite generar un editor básico para el modelo a través del cual es posible crear instancias del modelo. El editor también es un plug-in de Eclipse pero tiene funcionalidad reducida y despliega información de manera estándar. Por lo tanto decidimos mejorarlo para facilitar el uso y crear un entorno de trabajo más amigable. Para esto creamos la interfaz de usuario utilizando ideas del proyecto EEF que provee un editor gráfico para el modelo que está basado en formularios. Este editor fue modificado utilizando SWT y JFace para adaptarlo a las necesidades del proyecto.

El plug-in construido con el editor que permite crear instancias del meta-modelo de casos de uso tiene el nombre *testcasetool.editor*.

Los algoritmos presentados en el capítulo 3 para generar un modelo de pruebas a partir del modelo de casos de uso fueron implementados con clases que representan conceptos del diagrama de actividad y conceptos de UTP. Esta implementación se encuentran en el plug-in *testcasetool.transformation*. En este plug-in también se encuentra la implementación para generar valores de prueba a partir de las restricciones definidas para los parámetros en los casos de uso.

La herramienta construida, consiste en estos cuatro plug-ins descriptos. En la Figura 29 se muestran los componentes y sus dependencias. Estás dependencias indican como un plug-in utiliza a otros.

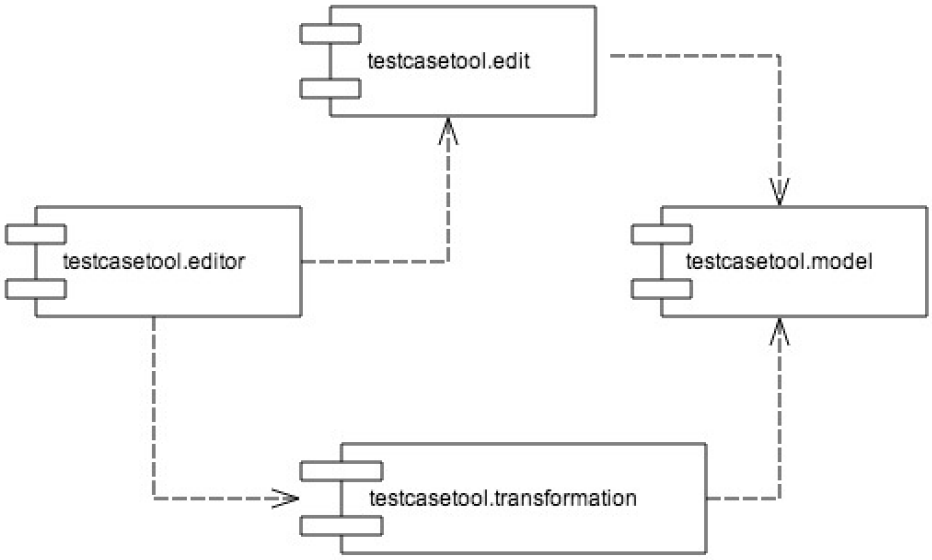

Figura 29: Diagrama de componentes de la herramienta

El componente *testcasetool.editor* contiene la interfaz de usuario que utiliza clases de *testcasetool.edit* para crear y desplegar objetos del modelo.

El componente *testcasetool.transformation* contiene la implementación que transforma los casos de uso en diagramas de actividad y posteriormente la generación de pruebas y valores de prueba. Este componente es utilizado desde el editor una vez que los casos de uso son ingresados.

El componente *testcasetool.model* contiene las clases que implementan los conceptos del meta-modelo de casos de uso. Este es utilizado tanto desde el editor como desde el componente que realiza la generación de pruebas.

El código fuente de la herramienta puede ser accedido en el repositorio SVN del proyecto en *Google Code*.[10]

10 SVN Test Case tool https://code.google.com/p/use-case-based-testing/

5.3 Interfaz gráfica de la herramienta

En esta sección se describen los principales componentes de la interfaz gráfica de la herramienta.

La interfaz es utilizada para crear instancias de conceptos representados en el meta-modelo de casos de uso y también para generar casos de prueba a partir de ellos.

La interfaz está compuesta por un editor y dos vistas. En la Figura 30 se muestran las vistas en los recuadros naranja y el editor en el recuadro azul.

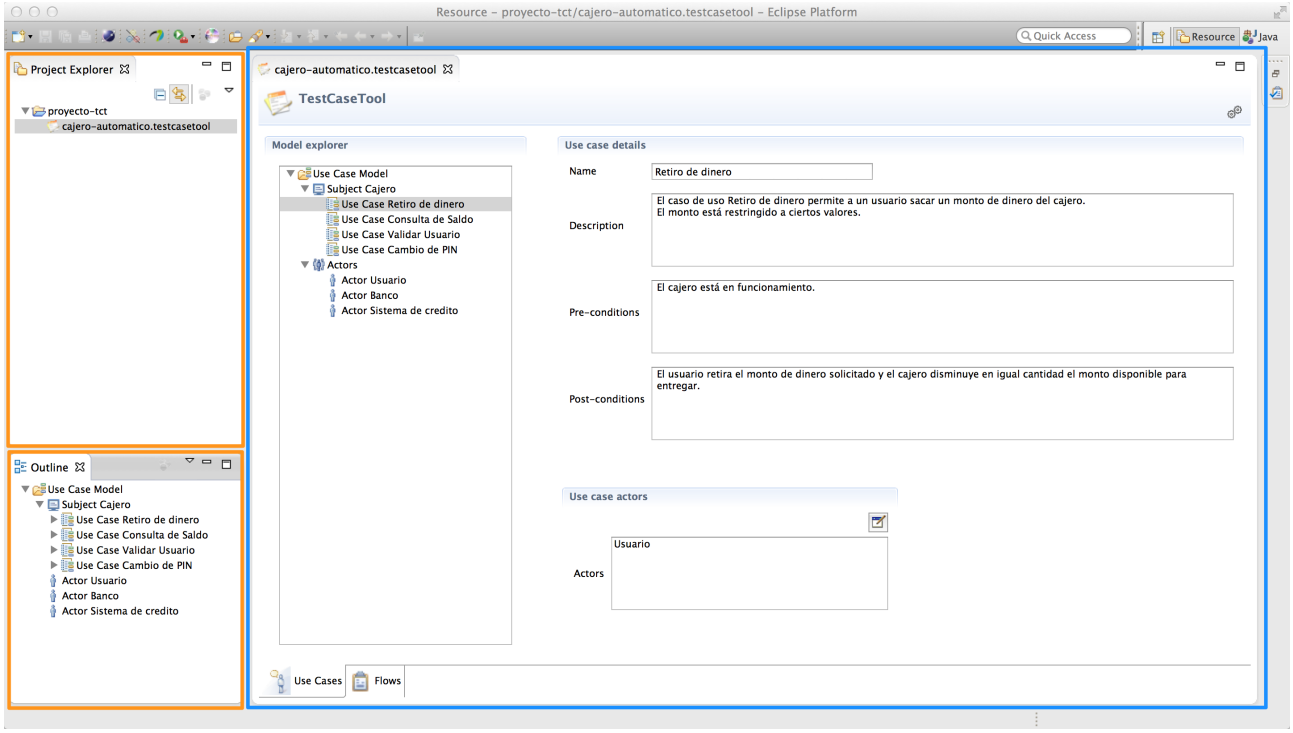

Figura 30: Componentes de la interfaz de usuario

La vista “Project Explorer” muestra el archivo que contiene el modelo de casos de uso. Este archivo tiene extensión “testcasetool” y puede ser incluido en cualquier tipo de proyecto de Eclipse. Cuando el archivo es seleccionado, se despliega el editor desarrollado en la herramienta. En la Figura 31 se muestra la vista “Project Explorer”.

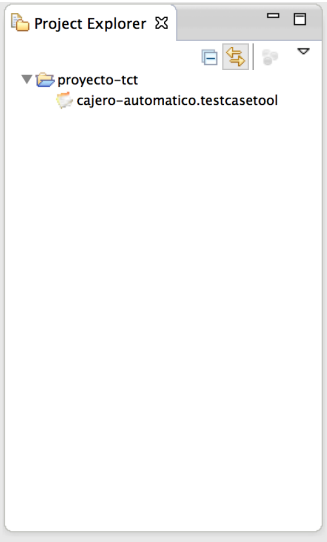

Figura 31: Vista Project Explorer

La vista “Outline” contiene todos los objetos del modelo que está siendo construido. Es útil para tener una visión global del modelo independiente de la acción que se esté ejecutando en el editor. En la Figura 32 se muestra la vista “Outline”.

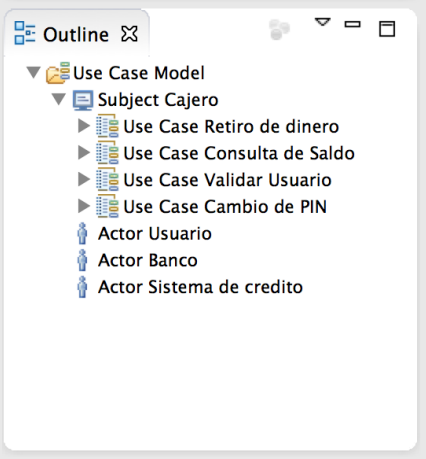

Figura 32: Vista Outline

El editor tiene una estructura que sigue el patrón master-detail, en el lado izquierdo se muestra un árbol con objetos del modelo mientras que en el lado derecho se muestran los detalles de un objeto determinado. Cuando se selecciona un elemento en el master se despliegan los detalles de elemento seleccionado.

El editor fue dividido en dos secciones que separan aspectos generales y particulares del modelo de casos de uso.

En la sección llamada “Use Cases” es posible ingresar información del sistema en construcción, la descripción de los casos de uso y de los actores participantes.

En la Figura 33 se muestra la sección “Use Cases” del editor donde se está creando el caso de uso Retiro de dinero.

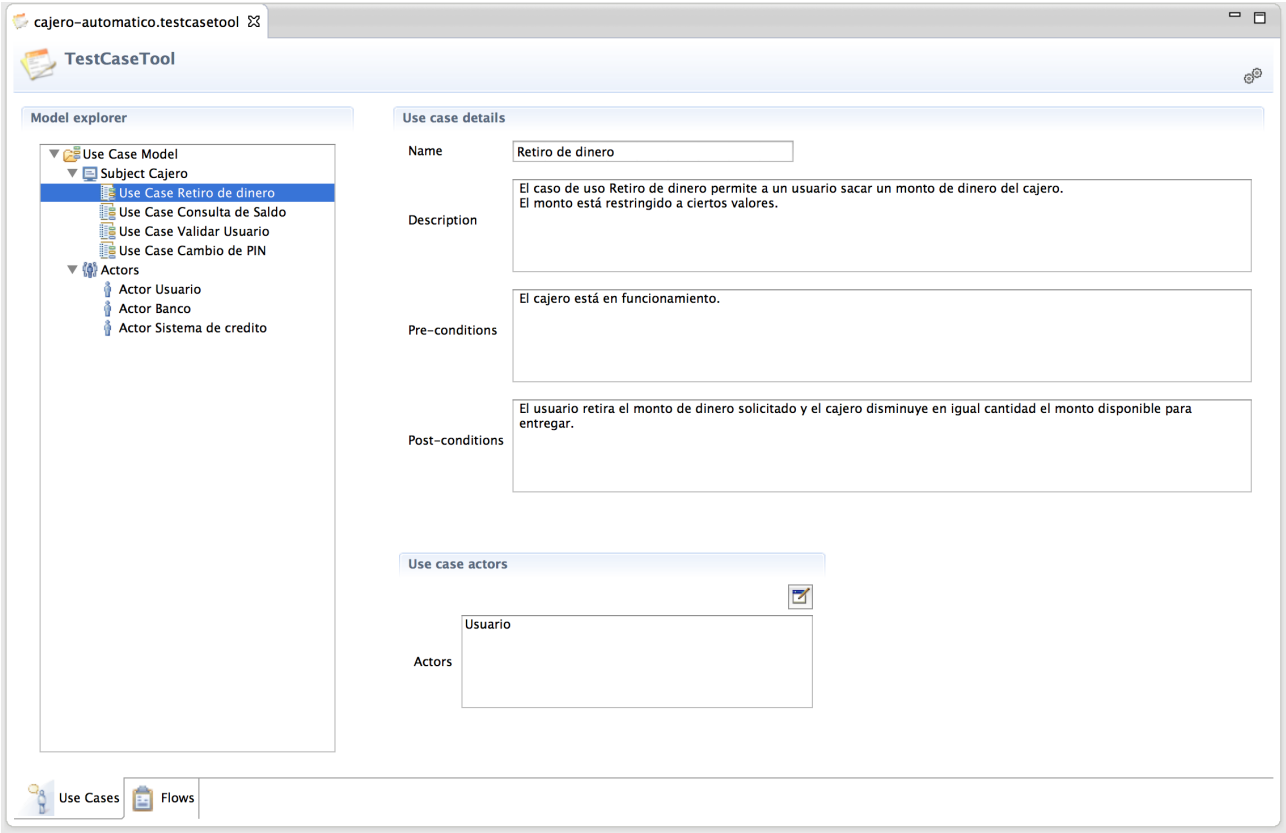

Figura 33: Sección "Use Cases" del editor

En la sección llamada “Flows” es posible crear el flujo básico y los flujos alternativos de un caso de uso. Estos flujos están formados por acciones que a su vez pueden contener parámetros. Esta sección del editor permite ingresar información del comportamiento del caso de uso a través de los diferentes flujos de ejecución.

En la Figura 34 se muestra la sección “Flows” donde se está editando un paso del flujo básico correspondiente al caso de uso Retiro de dinero.

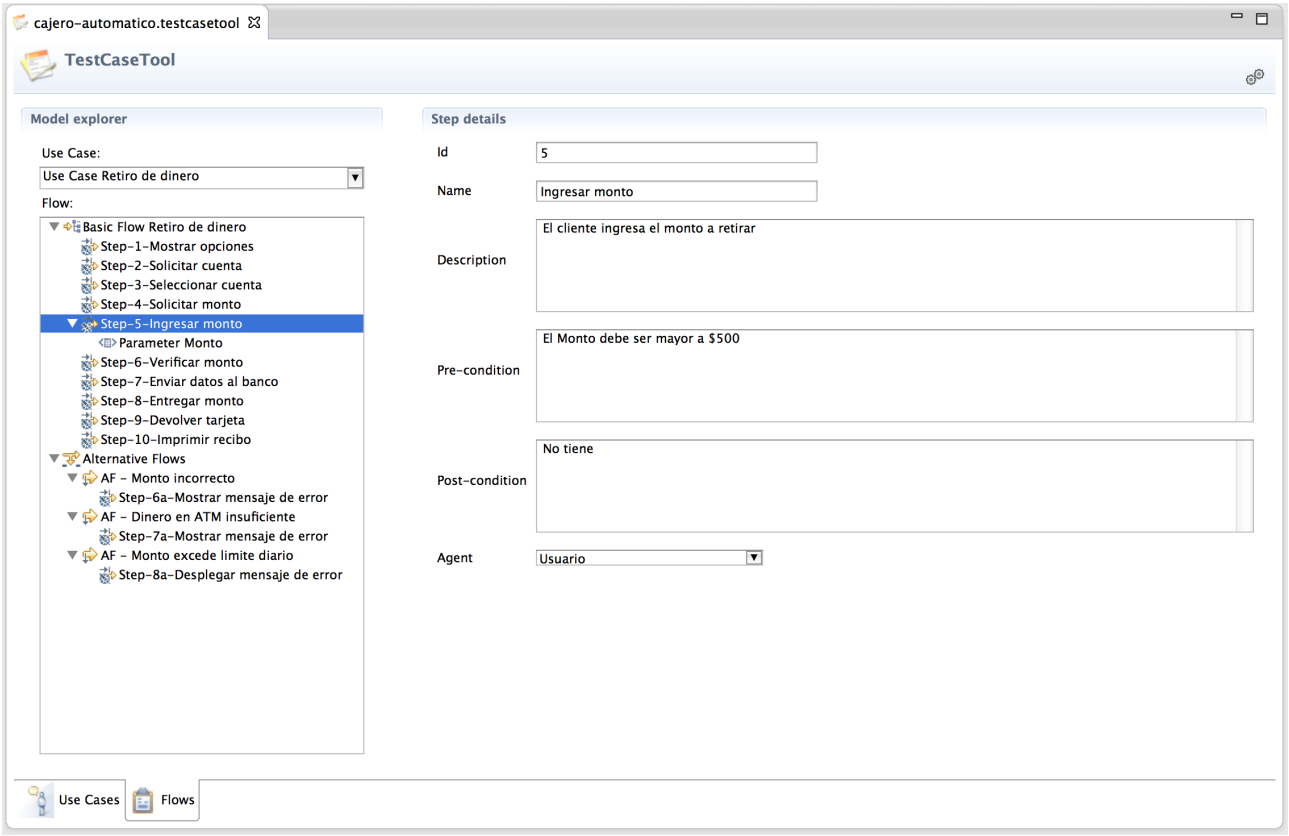

Figura 34: Sección "Flows" del editor

Las pestañas que se encuentran en la parte inferior del editor permiten seleccionar la sección en la que se desea trabajar.

Luego de crear los casos de uso, es posible generar casos de prueba a partir de la información ingresada. Para esto se debe seleccionar la imagen que se encuentra en la esquina superior derecha del editor.

Los casos de prueba son representados en tablas que están incluidas en archivos html. Estos archivos están agrupados en una carpeta llamada “testcases” dentro del proyecto que contiene el archivo con extensión “testcasetool”.

En la Figura 35 se muestra la vista “Project Explorer” con los casos de prueba generados. También se observan los casos de prueba en un editor de html que utiliza Eclipse por defecto.

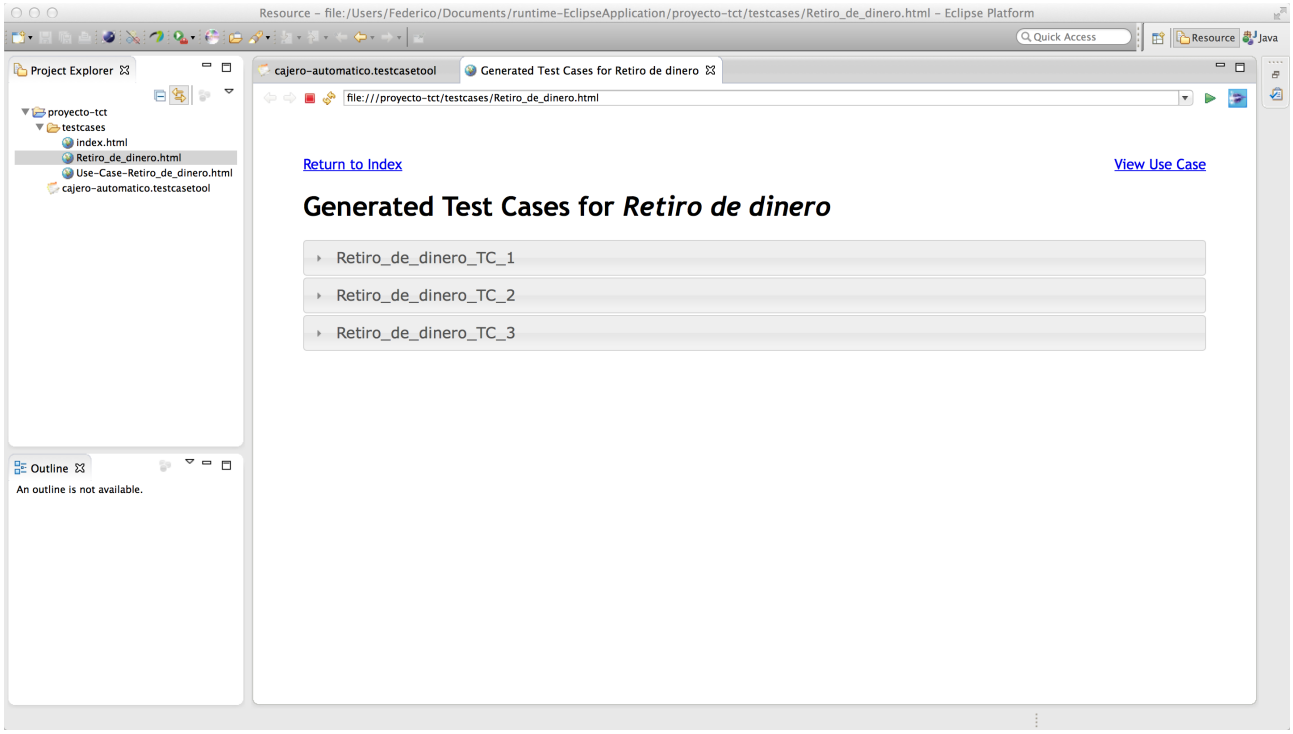

Figura 35: Casos de prueba para Retiro de dinero

Los casos de prueba son representados mediante tablas que contienen los pasos para realizar la prueba. Para acceder a esta información se debe seleccionar alguno de ellos en el editor.

La interfaz de usuario y la funcionalidad que provee son descritas con más detalle en el manual de usuario que se encuentra en el anexo 4 de este informe.

Capítulo 6 Conclusiones

El presente proyecto se enmarca dentro de la necesidad de contar con una herramienta para el *testing* funcional a partir de casos de uso, de tal forma que sea posible automatizar, tanto la creación de pruebas como los valores de prueba a ser usados en las mismas.

6.1 Resultados obtenidos

En la primera etapa del proyecto se construyó un meta-modelo de casos de uso que contiene conceptos e información relevantes para actividades de *testing* y en particular para la generación automática de pruebas.

En él se representan conceptos que están presentes en la descripción textual de los casos de uso y que son independientes de la forma en que estos sean escritos.

El meta-modelo permite resolver en una misma etapa la creación de los casos de uso y la incorporación de conceptos de *testing* lo que hace posible disponer de casos de prueba desde etapas tempranas en el proceso de construcción de un sistema de software.

Se desarrollaron conceptos que representan valores de prueba y que permiten automatizar la generación de estos valores a través de las técnicas de partición de equivalencia y valor límite.

En la segunda etapa del proyecto se crearon dos algoritmos con el objetivo de generar casos de prueba a partir del meta-modelo de casos de uso construido.

El primer algoritmo recibe como entrada el meta-modelo de casos de uso y tiene como resultado un diagrama de actividad que representa la interacción entre actores y sistema descrita en los casos de uso.

El segundo algoritmo tiene como entrada el resultado del primero junto al meta-modelo de casos de uso. El resultado es un conjunto de casos de prueba que son modelados usando conceptos de UML *Testing Profile*.

Los datos de prueba son generados mediante una técnica que utiliza la información almacenada en el meta-modelo de casos de uso. La técnica creada se basa en partición en clases de equivalencia y valores límite.

En la última etapa del proyecto se implementó una herramienta que consiste en un plug-in de Eclipse para crear casos de uso con información representada por conceptos del meta-modelo y que permite generar casos de prueba en forma automática utilizando los algoritmos desarrollados.

6.2 Conclusiones

El trabajo realizado a lo largo del proyecto sirve para demostrar que es posible generar casos de prueba en forma automática a partir de casos de uso definidos en lenguaje natural y que respetan la estructura definida en un meta-modelo de casos de uso.

En la investigación del estado del arte no se encontraron meta-modelos que representan conceptos de testing vinculados a la generación automática de pruebas. Por lo tanto el meta-modelo construido es un aporte importante del proyecto ya que incorpora conceptos no presentes en otros trabajos.

Los casos de uso descritos en lenguaje natural deben respetar una estructura para que sea posible procesar la información que contienen en forma automática.

Los trabajos sobre generación de casos de prueba a partir de casos de uso que se encontraron en el estado del arte describen algoritmos en alto nivel, por lo tanto se debieron construir algoritmos específicos que utilizan el meta-modelo de casos de uso del proyecto.

El caso de estudio en el capítulo 4 permite aplicar la técnica desarrollada a lo largo del proyecto sobre un conjunto de casos de uso. Esto permitió poner en práctica el trabajo realizado y así comprobar la factibilidad de nuestra propuesta.

En relación a la generación automática de valores de prueba, no se encontraron trabajos que consideren el tema. Por lo tanto las técnicas desarrolladas también son un aporte de este proyecto.

Finalmente, la plataforma de Eclipse como ambiente de desarrollo tiene una curva de aprendizaje elevada. Existe mucha documentación en libros y sitios web aunque muchas veces está desactualizada y es necesario inspeccionar el código fuente de Eclipse para entender y extender la plataforma.

El proyecto presenta argumentos sólidos que permiten considerar la herramienta construida para ser extendida y utilizada en un caso real de producción.

6.3 Trabajos a futuro

Durante el desarrollo del proyecto se han identificado algunos puntos donde es posible profundizar en futuras investigaciones. En esta sección se describen posibles caminos a seguir para mejorar los resultados obtenidos en este trabajo.

El meta-modelo de casos de uso propuesto en el proyecto puede ser formalizado agregando restricciones de UML que hagan el modelo compatible con ese estándar. Esto no implica una mejora en la construcción de pruebas a partir de casos de uso, sin embargo podría ser de utilidad para fomentar el uso del meta-modelo en otros trabajos.

En relación a los algoritmos desarrollados para generar casos de prueba a partir de casos de uso, existen algunos puntos de mejora que se mencionan a continuación.

Puede ser de utilidad considerar las relaciones de inclusión y extensión de casos de uso para la generación de pruebas. Se debe centrar la atención en la secuencia de pasos que tendrán los casos de prueba al existir estas relaciones en los caso de uso.

El algoritmo que genera los valores de prueba utiliza el concepto de validez de un parámetro para determinar si un escenario alternativo debe ejecutarse. Una mejora en este sentido es permitir la existencia de múltiples escenarios alternativos en un paso del flujo principal del caso de uso.

La condición asociada a un escenario alternativo puede ser extendida permitiendo el uso de más de un parámetro. Esto puede implementarse definiendo condiciones lógicas de “and” y “or” que se apliquen sobre los parámetros.

En la implementación del algoritmo es posible considerar otro tipos de pasos diferentes a *ActionStep*, por ejemplo bloques de repetición o puntos de extensión.

En relación a la generación de valores, es posible definir nuevas restricciones que permitan generar datos de prueba para tipos de datos más complejos como listas u objetos.

Las mejoras relativas al plug-in de Eclipse se describen a continuación.

El diagrama de actividad generado a partir de un caso de uso puede ser visualizado en una vista de Eclipse. Esto puede ser de utilidad para tener una visión global de los flujos y bifurcaciones que se producen en los casos de uso y que dan lugar a los casos de prueba.

Finalmente, puede ser factible generar código JUnit que ejecute los casos de prueba en forma automática. Para esto es necesario simular el sistema real y definir operaciones que sean invocadas usando los valores de prueba generados.

Referencias

[Ahl02] Naresh Ahlowalia, “Testing from use cases using path analysis technique”, International Conference On Software Testing Analysis & Review, Noviembre 4-8, 2002

[BDGSW07] Baker, P., Dai, Z.R., Grabowski, J., Schieferdecker, I., Williams, C, “Model Driven Testing Using the UML Testing Profile”, Springer-Verlag New York, Inc., Secaucus, NJ, 2007

[Coc00] Cockburn, Alistair , “Writing Effective Use Cases”, Addison-Wesley Professional, 2000

[GCEMR08] Gutiérrez, J.J., Clémentine, N., Escalona, M.J., Mejías, M., Ramos, I.M.: “Visualization of Use Cases through Automatically Generated Activity Diagrams”. MODELS2008. Springer

[GEMT09] J.J. Gutiérrez, M.J. Escalona, M. Mejías, J. Torres, “An approach for Model-Driven test generation”, Research Challenges in Information Science, 2009. RCIS 2009

[Heu01] Jim Heumann, “Generating Test Cases From Use Cases”, Rational Software, Junio 2001.

[KS09] Debasish Kundu y Debasis Samanta, “A Novel Approach to Generate Test Cases from UML Activity Diagrams”, Journal of Object Technology, Vol 8, Nº 3, Mayo-Junio 2009

[UTP13] OMG, “UML Testing Profile (UTP)” versión 1.2, OMG, 2013, formal/2013-04-03

[SM11] Siqueira, Fábio Levy; Muniz Silva, Paulo Sérgio. “An Essential Textual Use Case Meta-model Based on an Analysis of Existing Proposals” WER 2011.

[Som08] Somé, Stephane S. “A Meta-Model for Textual Use Case Description” , JOT Vol. 8, No. 7, November- December 2008

[Wil01] Williams, Clay E. “Toward a Test-Ready Meta-model for Use Cases” IBM T.J. Watson Research Center, NY, USA, 2001

[SBPM08] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E., “EMF Eclipse Modeling Framework: Second edition”, Addison-Wesley Professional, Diciembre 2008.

[CR09] Clayberg, E., Rubel, D. “Eclipse: plug-ins 3rd ed.”, Addison-Wesley, 2009.

Glosario

Backtracking

Es una estrategia para encontrar, de manera recursiva, soluciones a problemas que cumplen con un conjunto restricciones definidas para el mismo. En su forma básica, la idea de backtracking se asemeja a un recorrido en profundidad dentro de un grafo dirigido.

Deep-First-Search (DFS)

Una recorrida en profundidad (en inglés DFS o Depth First Search) es un algoritmo que permite recorrer todos los nodos de un grafo o árbol de manera ordenada, pero no uniforme. Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa, de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado.

MetaObject Facility (MOF)

Es un estándar creado por la Object Management Group (OMG) para la creación, manejo, intercambio y gestión de modelos cuando se utiliza el paradigma de MDE.[11]

Model Driven Architecture (MDA)

La arquitectura basada en modelos, es una técnica para diseñar sistemas de software propuesto por OMG para dar soporte a la ingeniería basada en modelos (MDE por sus siglas en ingles). La MDA provee un conjunto de guías para poder estructurar las especificaciones del sistema expresándolas como modelos.[12]

Model Based Testing (MBT)

La técnica de testing basado en modelos, por sus siglas en ingles MBT, consiste en la generación automática de pruebas de software, mediante el uso de los modelos de requerimientos y comportamiento del sistema.

Model Driven Engineering (MDE)

Es un paradigma de desarrollo de software que se enfoca más en la creación y explotación de modelos de dominio (representaciones abstractas de los conocimientos y actividades que rigen un dominio de aplicación en particular), que por sobre los conceptos informáticos y algorítmicos. El concepto fue introducido por la OMG cuando desarrollo la arquitectura basada en modelos (MDA, sigla en ingles para Model-Driven Architecture).

Object Constraint Language (OCL)

11 MetaObject Facility http://www.omg.org/mof/
12 Model Driven Architecture http://www.omg.org/mda/

OCL es un lenguaje para la descripción formal de expresiones sobre los modelos UML y MOF. Sus expresiones pueden representar invariantes, pre-condiciones, post-condiciones, inicializaciones, reglas de derivación, así como consultas a objetos para determinar sus condiciones de estado. Se trata de un lenguaje sin efectos de colaterales, de manera que la verificación de una condición, que se presupone una operación instantánea, nunca altera los objetos del modelo. Su papel principal es el de complementar los diferentes artefactos de la notación de los modelos con requerimientos formalmente expresados y que no pueden ser expresados mediante los diagramas. También forma parte fundamental del estándar de OMG para la especificación de transformaciones de modelos MOF (QVT).

Object Management Group (OMG)

Es un consorcio internacional sin fines de lucro, que promueve estándares para la industria del software. OMG provee especificaciones para dichos estándares pero no brinda implementaciones, quedando estas ultimas en manos de los socios miembros del consorcio.[13]

Query/View/Transformation (QVT)

Es un conjunto estándar de lenguajes utilizados para la especificación de transformaciones entre modelos, definido por OMG. QVT es una pieza clave utilizada en el diseño de software basado en MDA.

SUT

Acrónimo para Sistema Bajo Pruebas (del inglés, System Under Testing)

Tester

Persona encargada realizar las pruebas sobre un sistema para verificar su correcto funcionamiento.

Unified Modeling Language (UML)

UML es un lenguaje para el desarrollo de modelos en el ámbito de la ingeniería de software, el estándar fue propuesto y patrocinado por la OMG. UML incluye un conjunto de herramientas gráficas que permiten visualizar, especificar, construir y documentar el sistema.[14]

XML Metadata Interchange (XMI)

El XML Metadata Interchange, por sus siglas en ingles XMI, es un estándar desarrollado por el Object Management Group (OMG) para el intercambio de meta-data utilizando Extensible Markup Language (XML). Puede ser utilizado para cualquier meta-data cuyo meta-modelo pueda ser expresado con MOF. Uno de los usos más comunes del XMI es como formato de intercambio de modelos UML.

13 Sitio oficial Object Management Group http://www.omg.org/
14 Unified Modeling Language http://www.uml.org/

Anexos

Anexo 1 Visualización de casos de uso a través de diagramas de actividad

1.1 Introducción

En el trabajo “Visualization of Use Cases through Automatically Generated Activity Diagrams” los autores presentan un algoritmo para generar Diagramas de Actividad UML a partir de requerimientos funcionales de un sistema.

Para ello, crean un meta-modelo de requerimientos funcionales y definen un conjunto de transformaciones, entre dicho modelo y el meta-modelo de diagramas de actividad de UML.

Los autores destacan que los Casos de Uso estructurados, se han convertido en un estándar de la industria y se han realizado muchas investigaciones sobre ellos. Se entiende por Casos de Uso estructurados los definidos por el estándar de UML mejorados con alguna información adicional como pre y pos condiciones y escenarios de ejecución. Muchos de ellos siguen lineamientos de escritura como los propuestos por Cockburn para el uso de lenguaje natural para la descripción de los escenarios.

El caso de uso en UML no contiene mucha información en sí, solo define la frontera del sistema, los actores, y las relaciones entre casos de uso y la vinculación entre actores y casos de usos.

Cada vez más metodologías como RUP o Catalysis, recomiendan a sus usuarios enriquecer la descripción de casos de uso con alguna representación gráfica del mismo. Representaciones gráficas del caso de uso (como Diagramas de Secuencia o de Actividad) tienen la ventaja de poder mostrar fácilmente cuestiones dinámicas de los casos de uso, como pueden ser bucles y escenarios alternativos y diferentes opciones que puedan existir. Además, pueden ser fácilmente analizados por herramientas automatizadas, para por ejemplo generar casos de pruebas.

Según los autores, la contribución principal de su trabajo, es la generación automática de diagramas de actividad a partir de la descripción textual de los escenarios del caso de uso. Dicha generación es realizada a partir de transformaciones entre modelos, que representan el caso de uso textual y los diagramas de actividad de UML. Para ello, definen un meta-modelo de requerimientos funcionales y un conjunto de transformaciones en QVT que son aplicadas a la representación textual para generar el diagrama de actividad.

En el contexto de nuestro trabajo, también definimos un meta-modelo para la representación del caso de uso, al cual le agregamos información extra relativa al testing del sistema. También definimos transformaciones que vinculan elementos de dicho meta-modelo con conceptos del meta-modelo de Diagramas de actividad de UML. La idea es, al igual que los autores de este trabajo, aprovechar las ventajas de representación visual de los diagramas, y ademas de la facilidad para procesarlos de forma automática para generar casos de prueba, valiéndonos para ello de otros trabajos e investigaciones realizadas previamente.

1.2 Meta-modelos de Requerimientos Funcionales y de Diagramas de Actividad

El desarrollo del meta-modelo (Figura 36) se basa en el hecho de que si bien existen muchas maneras de representar los requerimientos funcionales, la mayoría de ellos utilizan el mismo conjunto básico de elementos. Dicho conjunto está compuesto por participantes, pre-condiciones, pos-condiciones, un conjunto de pasos (necesarios para alcanzar el objetivo planteado por el caso de uso) y un conjunto de pasos "excepcionales" (necesarios para manejar los casos de error y de ejecución alternativos).

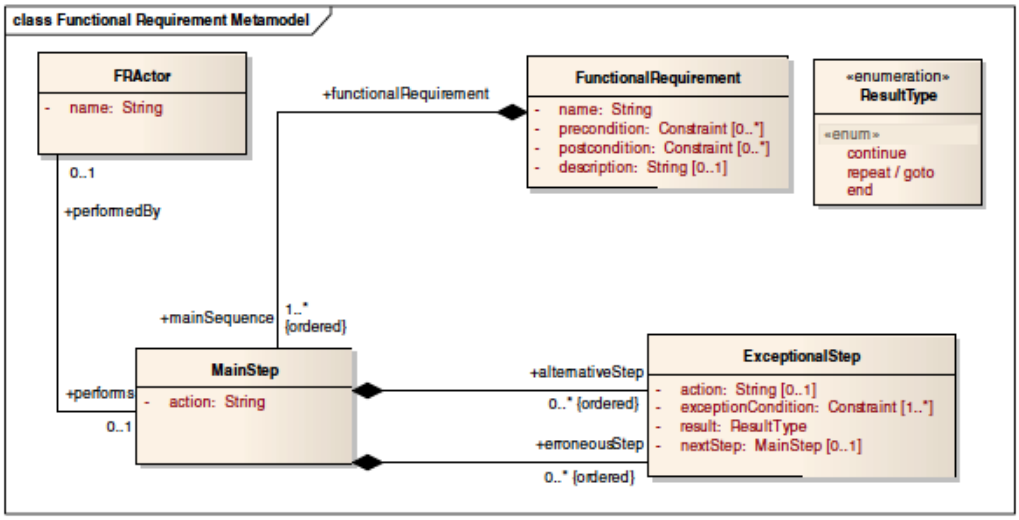

Figura 36: Meta-modelo de requerimientos funcionales

La meta-clase *FRActor* modela a un actor externo que participa en el requerimiento funcional a través de la realización de "*steps*".

La meta-clase *FunctionalRequirement* define las interacciones entre el sistema y los actores externos. El comportamiento de el *FunctionalRequirement* está definido por un conjunto de "*steps*" que pueden ser de tres tipos diferentes.

La secuencia de pasos *mainSequence* define los *steps* realizados por elementos *FRActor* para lograr el objetivo principal del requerimiento funcional. Los pasos alternativos *alternativeSteps* describen variantes a los descriptos en la secuencia principal *mainSequence*. Del mismo modo, los pasos de error *erroneousSteps* indican escenarios erróneos a los que se llega luego de la ejecución de un paso. Tanto los pasos alternativos como los pasos erróneos se modelan con la meta-clase *ExceptionalStep.*

En la meta-clase *MainStep*, la única información relevante es la acción que lleva a adelante el actor (atributo *action*). En un *ExceptionalStep*, la acción no es obligatoria, y este paso es realizado por el mismo actor que realiza el *MainStep*. La condición *exceptionCondition* describe una expresión booleana que debe evaluarse como verdadera para que el paso se lleve a cabo.

Se definieron también tres tipos de resultados posibles para ExceptionalStep, que son enumerados en *ResultType* como "*end*", "*repeat/goto*" y "*continue*". La continuación "*continue*" expresa que se debe seguir con la ejecución del paso siguiente al alternativo luego de que este finalice. Un "*repeat/goto*" indica que luego de ejecutar el paso excepcional se debe continuar con el paso indicado en el atributo "*nextStep*". Finalmente, un resultado de tipo "*end*" indica que el caso de uso debe terminar luego de la ejecución de ese *step*.

1.3 El meta-modelo de Diagramas de Actividad de UML

El meta-modelo de UML para diagramas de actividad contiene una cantidad de elementos que ayudan a comprender la semántica de los conceptos y brinda flexibilidad para su uso.

Los autores de este trabajo escogieron un subconjunto de dicho meta-modelo para poder generar los diagramas en forma automática. Dicho subconjunto es presentado en la Figura 37.

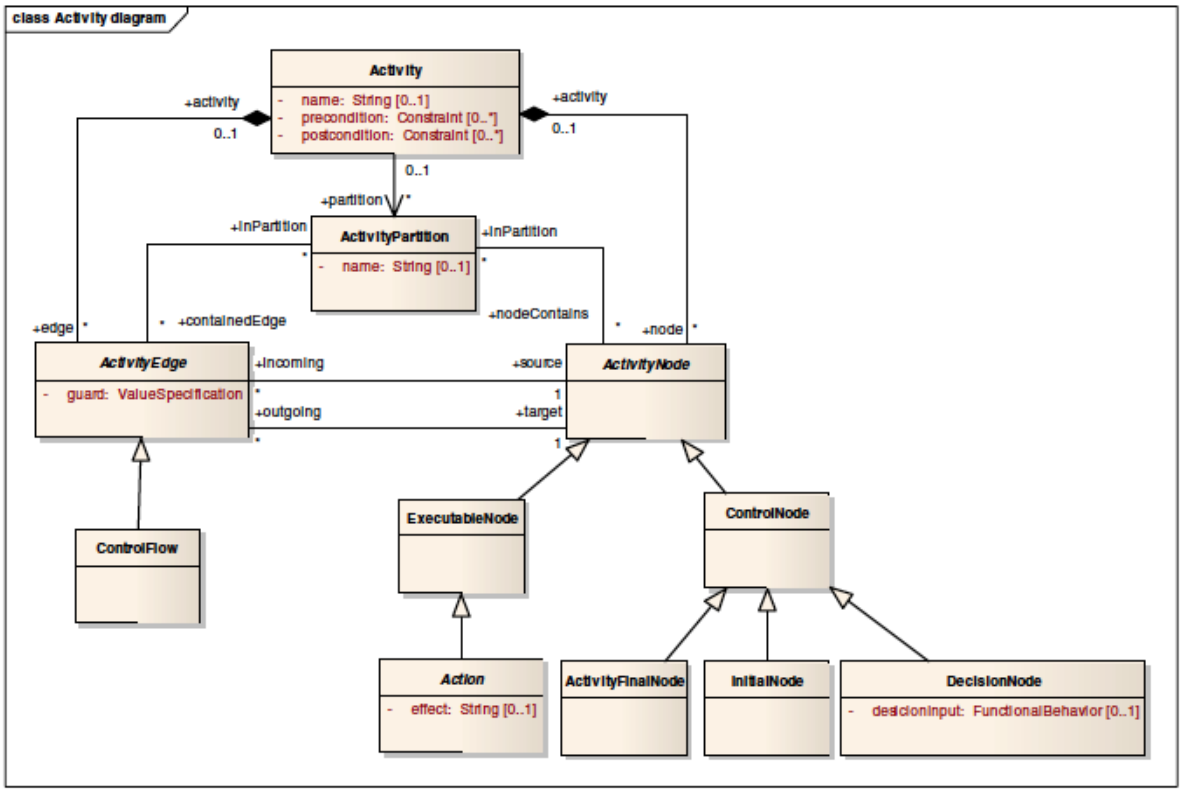

Figura 37: Meta-modelo de Diagramas de Actividad

La meta-clase *Activity* representa el diagrama de actividad y agrupa todos los elementos necesarios para la representación del mismo, como ser flujos de control, nodos y particiones.

Las particiones *ActivityPartition* son agrupaciones de elementos de un diagrama de actividad. Los flujos de control *ControlFlows* representan las entradas y salidas en un nodo.

Los nodos *Action* definen un comportamiento, los nodos decisión *DecisionNode* permiten elegir un flujo dependiendo de la evaluación de una expresión booleana. Finalmente los nodos *ActivityFinalNode* e *InitialNode* representan los nodos finales e iniciales respectivamente.

1.4 Transformaciones de Requerimientos Funcionales textuales a Diagramas de Actividad

A continuación se definen las transformaciones para generar un diagrama de actividad como instancia del meta-modelo de UML a partir de un requerimiento funcional como instancia de el meta-modelo *Functional Requirement*.

El diagrama de actividad debe representar todo el requerimiento funcional, llevando su mismo nombre y manteniendo las pre y pos condiciones. Los diferentes actores son modelados como particiones, los flujos de control representan los caminos de ejecución o escenarios, y finalmente los nodos "*action*" y "*decision*" modelan los "*steps*" y "*excecutionConditions*" respectivamente.

A grandes rasgos las transformaciones necesarias son:

1. Generar un diagrama de actividad por cada *FunctionalRequirement.*
2. Generar un nodo *Action* por cada *step.*
3. Generar un nodo *Decision* y otro *Action* por cada *ExceptionalStep.*
4. Conectar todos los elementos del diagrama de actividad utilizando *ControlFlows.*

Las tareas necesarias para dichas transformaciones son implementadas en QVT.

Para el primer paso, se genera una actividad *Activity* por cada requerimiento funcional, dicho elemento tiene por nombre, pre-condiciones y pos-condiciones, los mismos valores que posee el requerimiento funcional. Además, se crea una partición para cada actor participante y una partición extra para el Sistema.

En el segundo paso, un nodo *Action* es generado para cada *step* del *mainSequence*. El atributo *effect* del *Action* es el atributo *action* del *step*. Los nodos *Action* generados son incluidos dentro de la partición correspondiente al actor, que se identifica por la relación *performedBy*. Los *steps* que son realizados por el sistema no poseen dicha relación, por lo que el nodo *Action* generado se coloca dentro de la partición del sistema.

Para el tercer paso, dos nodos son creados por cada *ExceptionalStep.* Primero un *DecisionNode* que evalúa la *exceptionCondition* y permite o no la ejecución del *Alternative* o *Erroneous step.* La *exceptionCondition* es guardada en el atributo *decisionInput*. Luego, genera un *Action* node, si es que es necesario, para reflejar la acción que el actor realiza en dicho caso.

Por último para cada tipo de resultado, tiene una representación distinta en el diagrama de actividad. Si el resultado es de tipo "end", entonces se genera un nodo *ActivityFinalNode*, que se une al *DecisionNode,* o al nodo *Action* si existe, con un *ControlFlow.* Si es de tipo "*continue*", entonces el *step* indicado por *nextStep* es conectado usando un *ControlFlow* con el nodo de decisión, o con el *Action* si es que existe. Por último, si el resultado es de tipo "*repeat/go-to*" se realiza lo mismo que en el caso de "*continue*".

Finalmente, en el cuarto paso de la transformación, teniendo todos los elementos creados, se conectan aquellos que quedaron aislados. La siguiente secuencia de pasos se aplica para poder crear los enlaces que faltan:

1) Unir el nodo inicial al primer elemento y el nodo final con el último elemento.

2) Unir los nodos de decisión entre ellos.

3) Unir los nodos de decisión con nodos de actividad.

4) Unir los nodos de un step con nodos de otro step.

De este modo se completa el diagrama de actividad y todos sus nodos quedan conectados.

Anexo 2 Un enfoque para generación de pruebas en MDT

En el trabajo “An approach for Model-Driven test generation” los autores presentan una forma para generar casos de pruebas a partir de los requerimientos funcionales del sistema.

La metodología propuesta está basada en el paradigma de MDE (Model Driven Engineering) para la generación automática de casos de prueba.

2.1 Estado del arte

Como parte del estado de la literatura los autores destacan los trabajos realizados por Binder para la obtención de variables operacionales y la construcción de la tabla de decisión con todos los valores de dichas variables y los resultados esperados para cada combinación de las mismas. En el trabajo de Binder obtuvieron las ideas para la identificación de las variables operacionales.

Ademas destacan los trabajos de Briand y Labiche (TOTEM) y el CowSuite que representan los casos de uso como diagramas de secuencia de UML, y argumentan que si bien generan buenos resultados, le encontraron problemas al uso de dichos diagramas, ya que por ejemplo es muy difícil representar escenarios de error o alternativos en un único diagrama, y que además es necesaria información sobre la arquitectura y la representación interna del sistema en cuestión (como clases y mensajes intercambiados entre ellas). Por esto último, ellos concluyen que los diagramas de secuencias impide la utilización de dichas soluciones en etapas tempranas del desarrollo. Para su solución ellos eligieron el uso de diagramas de actividad.

En el contexto de nuestro trabajo, compartimos dicho análisis y en consecuencia se eligió también el uso de diagramas de actividad para representar los casos de uso.

2.2 El trabajo propuesto

La solución planteada comienza con la definición de los requerimientos funcionales de un sistema, y propone un conjunto de transformaciones automáticas basadas en QVT para derivar casos de prueba del sistema.

Si bien en la imagen siguiente se muestran todos los niveles de MDE que participan del proceso de generación de las pruebas del sistema, el trabajo cubre solo las transformaciones que ocuren en los niveles de CIM (Computer Independent Model) y a nivel de PIM (Platform Independent Model).

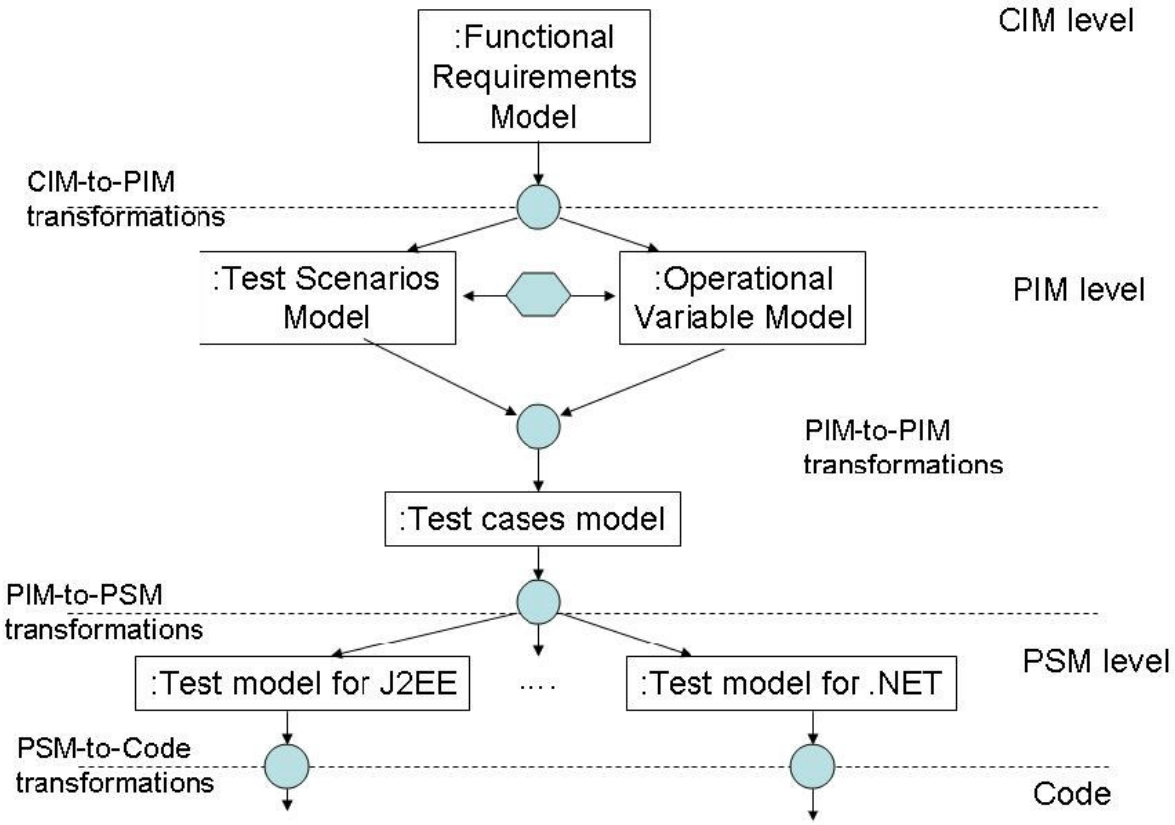

Figura 38: Generación de casos de prueba siguiendo el paradigma de MDE

Todo el proceso de generación de pruebas siguiendo MDE comienza con el conjunto de requerimientos funcionales definidos a nivel de CIM. Luego, con una serie de transformaciones en QVT se pueden obtener los modelos de nivel PIM.

Los escenarios de prueba son el resultado de dichas transformaciones en QVT, un escenario de prueba es una secuencia concreta de ejecución del sistema, derivado de los requerimientos funcionales.

Ademas, se obtienen también como resultado de otras transformaciones QVT un conjunto de variables operacionales, que son elementos del dominio cuyo valor puede cambiar en los distintos escenarios de ejecución de un requerimiento funcional.

Luego, con ambos conjuntos de artefactos, los autores realizan otra transformación, en este casos de PIM a PIM para obtener los casos de prueba sistematizados para los requerimientos funcionales definidos en el CIM.

Para dar fundamento al proceso, se definieron dos meta-modelos, uno para los requerimientos funcionales y otro para las pruebas funcionales.

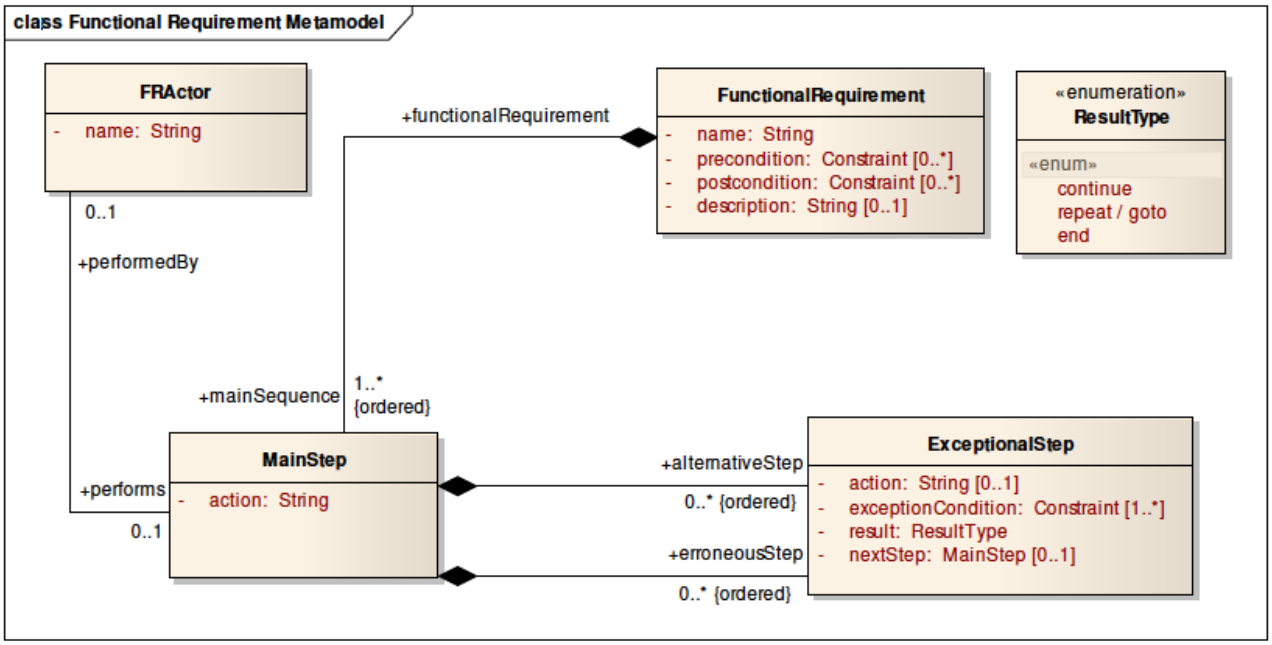

Figura 39: Meta-modelo de requerimientos funcionales

El meta-modelo para los requerimientos funcionales es el estudiado en detalle en “Visualization of Use Cases through automatically generated Activity Diagrams”.

En el contexto de nuestro trabajo, el equivalente a dicho meta-modelo es el definido para la representación de los Casos de Uso. Dicho meta-modelo, al igual que el definido por los autores para requerimientos funcionales, está a nivel de CIM en el paradigma de MDA.

En cuanto al meta-modelo definido para testing funcional, abarca todos los artefactos necesarios para la representación en el nivel PIM. Esto significa que está compuesto por un conjunto de meta-clases que abarcan la definición abstracta de los “Escenarios de prueba”, de “Variables Operacionales” y además de “Casos de prueba del sistema”.

Como todo enfoque de MDE, la propuesta está compuesta por dos partes, los meta-modelos y las transformaciones. Estas últimas son la forma de producir resultados en dicho enfoque.

Para las transformaciones entre los meta-modelos los autores eligieron QVT como lenguaje estándar, y dentro de QVT, una combinación de QVT Relacional y QVT Operacional.

2.3 Solución concreta

Para la representación concreta de los meta-modelos y transformaciones los autores eligieron la utilización de UML Profiles.

Además de eso, los autores requieren el uso de patrones de escritura para los requerimientos funcionales. En concreto eligieron NDT (Navigational Development Techniques).

Para poder analizar los distintos escenarios de ejecución de los requerimientos funcionales, utilizaron diagramas de actividad, ya que estos facilitan la detección de los escenarios gracias a su representación gráfica.

Si bien los patrones de escritura son fáciles de utilizar por los usuarios, ya que usan su propio vocabulario, los diagramas de actividad son más precisos para la generación de casos de prueba.

En el contexto de nuestro trabajo, no aplicamos ninguna restricción en la escritura, y si utilizaremos diagramas de actividad para la generación de casos de prueba.

En el trabajo presentado, los requerimientos funcionales se ingresan con un patrón tabular como el que sigue:

Name	UC-02. Search link by description
Preconditions	NO
Main Sequence	1. The user asks the system for searching links by description. 2. The system asks for the description. 3. The user introduces the description. 4. The system searches for the links which matches up with the description introduced by the user. 5. The system shows the results.
Errors/alternatives	3.1.1. At any time, the user may cancel the search, and then the use case ends. 4.1.p. If the actor introduces an empty description, then the system searches for all stored links and the result is to continue the execution of this use case.4.2.i. If the system finds any error performing the search, then an error message is shown and this use case ends. 5.1.i. If the result is empty, then the system shows a message and this use case ends.
Results	1. The system shows the results of UC-05 3.1.i. Out of the limits of this use case. 4.2.i. Error message. 5.1.p. Message of no found results
Post condition	NO

Figura 40: Requerimiento funcional estructurado

Los pasos descriptos en las secciones “Main Sequence”, “Error/alternatives” y “Results” están ingresados siguiendo el formato de escritura de NDT.

Luego, aplicando las transformaciones definidas en trabajos anteriores de los autores, generan el siguiente diagrama de actividad:

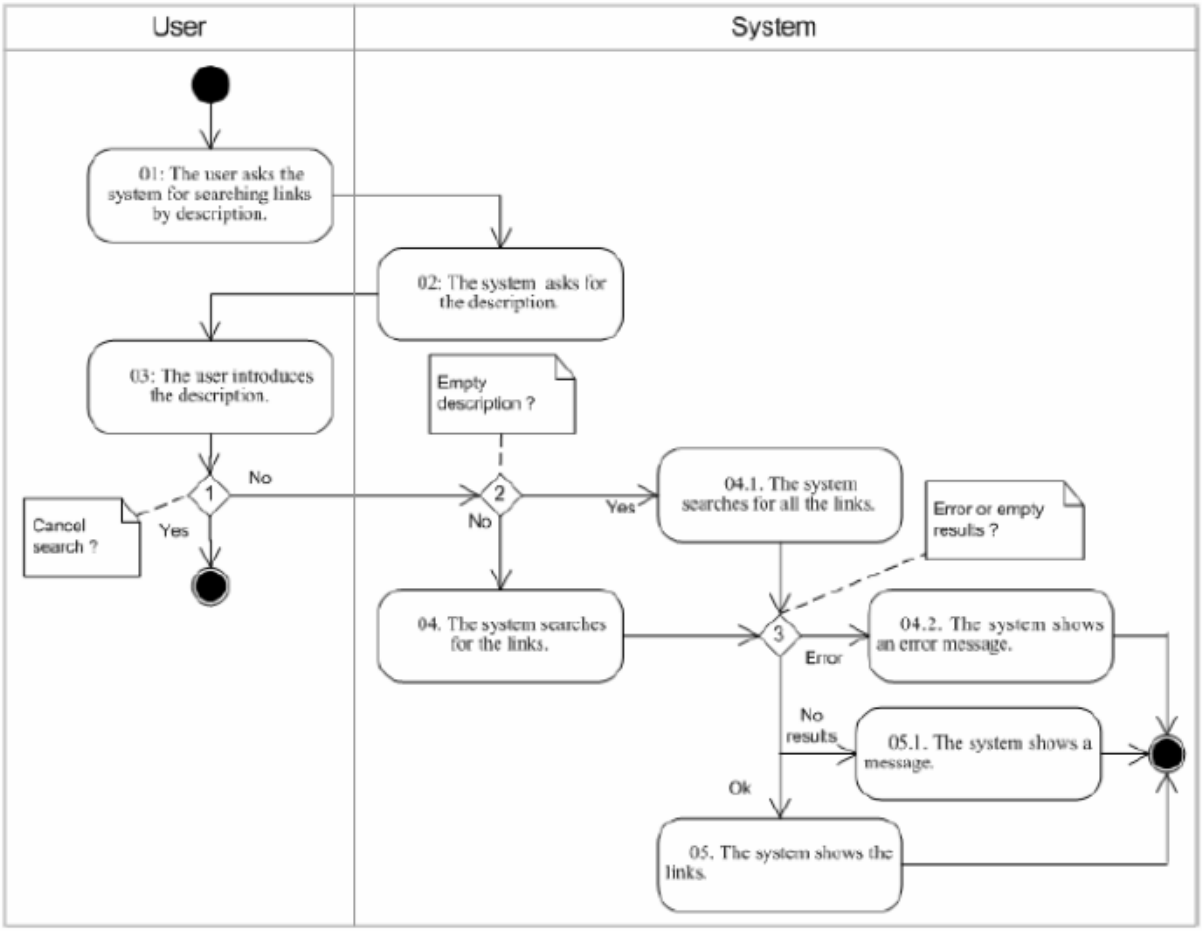

Figura 41: Diagrama de Actividad generado

Luego, aplicando las transformaciones entre modelos propuestas en su estudio, los autores generan los siguientes escenarios de ejecución:

Id	Path
1	01 -> 02 -> 03 -> D1(No) -> D2(No)-> 04 -> D3(No error & Results) -> 05
2	01 -> 02 -> 03 -> D1(No) -> D2(No)-> 04 -> D3(No error & No Results) -> 05.1
3	01 -> 02 -> 03 -> D1(No) -> D2(No)-> 04 -> D3(Error) -> 04.2
4	01 -> 02 -> 03 -> D1(No) -> D2(Yes)-> 04.1 -> D3(No error & Results) -> 05
5	01 -> 02 -> 03 -> D1(No) -> D2(Yes)-> 04.1 -> D3(No error & No Results) -> 05.1
6	01 -> 02 -> 03 -> D1(No) -> D2(Yes)-> 04.1 -> D3(Error) -> 04.2
7	01 -> 02 -> 03 -> D1(Yes)

Figura 42: Escenarios de ejecución generados

Finalmente, los test cases son representados con el siguiente patrón tabular:

Name	TC-01		
Description	-		
Source	UC-02. Search link by description		
Initial State	-		
Final results	-		
Test information	-		
Actions	**Index**	**Body**	**Test data**
	1	The user asks the system for searching links by description	
	2	The system asks for the description.	
	3	The system asks for the description.	D1 = No D2= No
	4	The system searches for the links which matches up with the description introduced by the user	
	5	The system shows the results.	D5 = No errors
Final states	-		
Priority	-		
Comments	-		

Figura 43: Representación tabular del Caso de Prueba

Anexo 3 Un enfoque nuevo para generar casos de prueba a partir de diagramas de actividad

En el trabajo “A Novel Approach to Generate Test Cases from UML Activity Diagrams”, los autores proponen una metodología para generar test cases a partir de los diagramas de actividad de UML. Dicha metodología consiste en tres pasos:

1) Agregar la información necesaria para el testing a los diagramas de actividad.

2) Convertir el diagrama de actividad en un grafo de actividad.

3) Generar los casos de prueba a partir del grafo de actividad.

El trabajo cubre diagramas de actividad con nodos de Fork y Join para el manejo de threads que pueden ser ejecutados en paralelo.

El tipo de fallas que los autores pretenden identificar con los casos de pruebas son de tres tipos diferentes, fallas en los nodos de decisión, fallas en los loops y fallas de sincronización. A este conjunto de fallas, los autores lo denominan “modelo de fallas”.

En la primer etapa agregan al Diagrama de actividad información relativa a cambios de estado y creación de nuevos objetos, además simplifican las decisiones, loops y threads que aparezcan anidados dentro de threads, cambiándolos por nodos de Actividad de mayor nivel de abstracción.

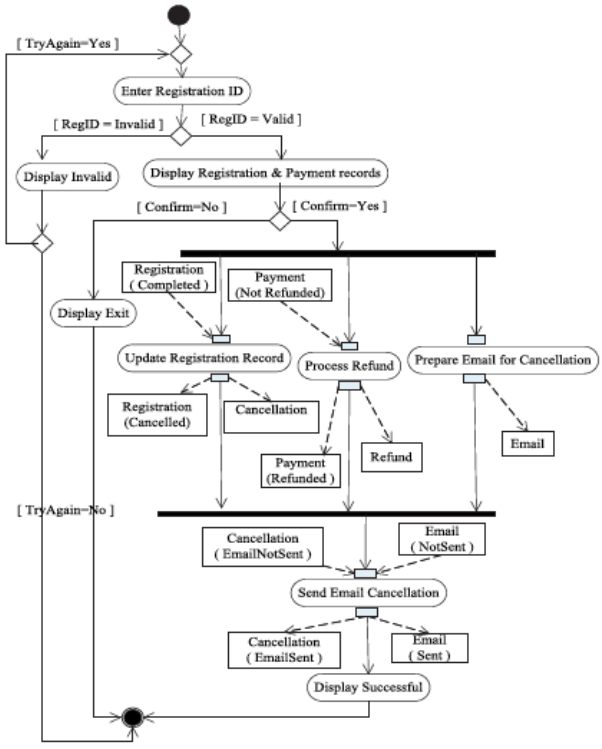

Figura 44: Diagrama de Actividad de ejemplo

Para el segundo paso, cada nodo representa un elemento “ActivityNode” y los enlaces representan los “ControlFlow” o flujos del Diagrama de actividad.

Los autores definen un conjunto de reglas que mapean los elementos del diagrama de actividad a nodos o enlaces del grafo de actividad que se muestran a continuación.

No	Constructs of Activity Diagram	Node of Activity Graph
1	Initial Node	Node of type S with no incoming edge
2	Activity Final Node	Node of type E with no outgoing edge
3	Flow Final Node	Node of type E with no outgoing edge
4	Decision Node	Node of type D
5	Guard Condition associated decision node	Node of type C and associated with condition string. Its parent node is of type D
6	Merge Node	Node of type M and having single outgoing edge
7	Fork Node	Node of type F with single incoming edge
8	Guard condition associated with fork node	Node of type C and its parent node is of type F
9	Join Node	Node of type J and will have one outgoing edge.
10	An object 'OB' at input/output pin of an activity 'AC'	Node of type O and associated object name is 'OB'. Its parent node will be of type 'A' and associated activity name 'AC'. If same object 'OB' is in both input and output pin of the activity 'AC', then only one node is to be used.
11	Object state 'S' of an object 'OB'	Node of type OS. If 'OB' is at input of an activity, then this node is left child of node of type O and associated object name is 'OB' otherwise this node is right child of parent node associated object name with 'OB'.
12	Activity Node	Node of type A. Its associated string is activity name.

Figura 45: Reglas de mapeo de conceptos

Luego de aplicar las reglas de conversión, el diagrama de actividad de ejemplo se representa con el siguiente grafo de actividad.

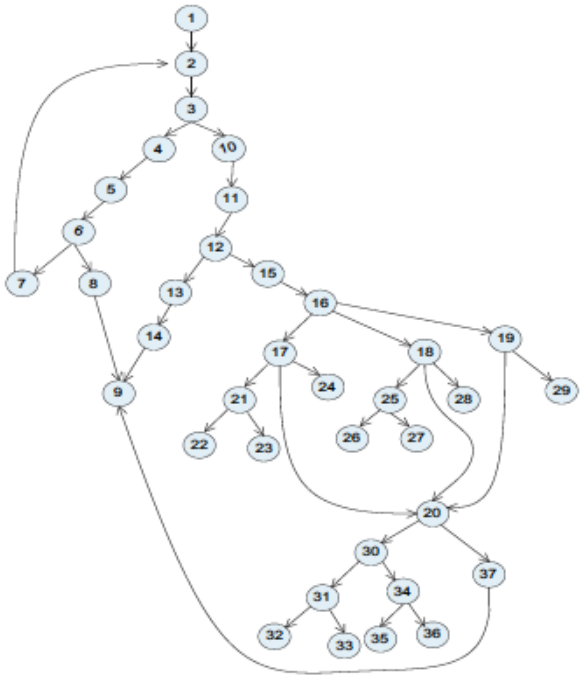

Figura 46: Grafo de Actividad generado

Además del grafo, arman una tabla NDT (Node-description-Table) donde guardan la descripción detallada del nodo con la información tomada desde el diagrama de actividad. Esta información será utilizada al momento de armar el test case final.

Para la generación de los casos de prueba a partir del grafo de actividad y teniendo en cuenta el modelo de fallas buscado, los autores definen un criterio de cobertura, que lo llaman “Activity path”. Un activity path es un camino de nodos en el grafo de actividad donde se considera que los nodos de loops son visitados en un máximo de dos oportunidades y además mantiene la precedencia de todos los nodos, concurrentes y no concurrentes.

Luego los autores definen un algoritmo para recorrer el grafo de actividad que genera dichos “activity paths”, y que se fundamenta en recorridas en DFS (deep-first-search) y BFS (breadth-first-search) para las secciones concurrentes.

Utilizando la salida de dicho algoritmo, aplican un par de reglas para derivar nuevos casos a partir de los ya generados. La primer regla implica simplificar caminos que contienen loops, y generar caminos simples, que son de hecho escenarios de prueba diferentes y que ejercitan distintas combinaciones de condiciones en comparación con las ejercitadas por el camino original. La segunda regla implica expandir las actividades que habían sido cambiadas en el primer paso, manteniendo la precedencia en el orden de los nodos de los caminos.

Finalmente, con este nuevo conjunto de caminos arman los test cases teniendo en cuenta que para el trabajo un test case está formado por la combinación de cuatro componentes que son la secuencia de las condiciones de branch, la secuencia de los nodos de actividad, cambios en los estados de objetos y finalmente los objetos que son creados en el test case.

Recorriendo el conjunto de caminos, y utilizando la tabla NDT el caso de prueba CPi se van armando recorriendo los nodos del camino Ci siguiendo los siguientes pasos:

1. Si el tipo del nodo actual es 'S' or 'E' or 'D' or 'F' or 'J', entonces el nodo es ignorado
2. Si el tipo del nodo es 'C', entonces la condición de branch asociada a ese nodo sera la siguiente condición en la sección de “Condiciones de branch” del CPi.
3. Si el tipo del nodo es 'A', entonces el nombre de la Activity asociada a ese nodo es la siguiente actividad en la sección de 'Secuencia de nodos de Actividad' del caso de prueba CPi.
4. Si el tipo del nodo actual es 'O' y el tipo del siguiente nodo es 'OS', entonces el nombre del objeto asociado al nodo actual se agrega a la sección “Cambio de estado de objetos” del caso de prueba CPi.
5. Si el tipo del nodo actual es 'OS' y el tipo del nodo anterior es 'O', entonces el estado asociado al nodo actual sera el “estado anterior” del objeto asociado al nodo anterior.
6. Si el tipo de los nodos actual y el siguiente es 'OS', entonces el estado del nodo actual sera el “nuevo estado” del objeto asociado al nodo precedente del nodo anterior.
7. Si el tipo del nodo actual es 'O' y el tipo del siguiente nodo es distinto de 'OS', entonces el objeto asociado con este nodo aparecerá en la sección de “Objetos creados” del caso de prueba CPi.

Los test cases son estructurados en una tabla como la que se muestra a continuación:

Test Case No	Sequence of Branch Conditions	Activity sequence	Object State changes [Object (Old state, New state)]	Object created
1	RegID = Invalid, TryAgain = Yes, RegID = Invalid, TryAgain = No	Enter Registration ID, Display InValid, Enter Registration ID, Display Invalid		
2	RegID = Invalid, TryAgain = Yes, RegID = Valid, Confirm = No	Enter Registration ID, Display InValid, Enter Registration ID, Display Registration and Payment Records, Display Exit		
3	RegID = Invalid, TryAgain = Yes, RegID = Valid, Confirm = Yes	Enter Registration ID, Display InValid, Enter Registration ID, Display Registration and Payment Records, Update Registration Record, Process Refund, Prepare Email for Cancellation, Send Email Cancellation, Display Successfully	Registration (Complete, Cancelled), Payment (NotRefunded, Refunded), Cancellation(EmailNotSent, EmailSent), Email (NotSent, Sent)	Cancellation, Refund, Email
4	RegID = Invalid, TryAgain = No	Enter Registration ID, Display InValid		
5	RegID = Valid, Confirm = No	Enter Registration ID, Display Registration and Payment Records, Display Exit		
6	RegID = Valid, Confirm = Yes	Enter Registration ID, Display Registration and Payment Records, Update Registration Record, Process Refund, Prepare Email for Cancellation, Send Email Cancellation, Display Successfully	Registration (Complete, Cancelled), Payment (NotRefunded, Refunded), Cancellation(EmailNotSent, EmailSent), Email (NotSent, Sent)	Cancellation, Refund, Email

Figura 47: Caso de Prueba en forma de Tabla

Anexo 4 Manual de Usuario

En conjunto con la herramienta construida se creo un manual de usuario en formato wiki que se encuentra en el repositorio del proyecto.

Este manual pretende ser una guía del uso de la herramienta. En su contenido se encuentra una guía de instalación y los pasos necesarios para su uso.

El contenido incluye:

- Guía de instalación.
- Creación de un editor de la herramienta en Eclipse.
- Creación del sistema, actores y casos de uso.
- Creación de los flujos de ejecución de los casos de uso.
- Definición de variables operacionales.
- Generación de los casos de prueba.

El manual de usuario puede encontrarse en la siguiente dirección:

https://code.google.com/p/use-case-based-testing/wiki/ManualDeUsuario