Generación de Pruebas del Sistema a Partir de la Especificación Funcional

Memoria de tesis doctoral presentada para la obtención del grado de Doctor por la Universidad de Sevillla

Autor:

Javier Jesús Gutiérrez Rodríguez

Dirigida por:

Doctor D. Manuel Mejías Risoto
Doctora Dña. María José Escalona Cuaresma

Sevilla, Noviembre 2011.

Departamento de
Lenguajes y Sistemas Informáticos
Universidad de Sevilla

Generación de Pruebas del Sistema a Partir de la Especificación Funcional

Memoria de tesis doctoral presentada para la obtención del grado de Doctor por la Universidad de Sevillla

Autor:

Javier Jesús Gutiérrez Rodríguez

Dirigida por:

Doctor D. Manuel Mejías Risoto
Doctora Dña. María José Escalona Cuaresma

Sevilla, noviembre 2011.

Agradecimientos

Son muchas las personas a las que tengo que agradecer la realidad de la tesis. Quiero comenzar por mis tutores, Manuel Mejías y María José Escalona, por sus siempre acertados consejos y su paciencia. Y a Miguel Toro e Isabel Ramos, por su orientación en todo el trabajo.

A todo el departamento de Lenguajes y Sistemas Informáticos y, en especial, al grupo de profesores con los que he compartido docencia en el departamento, por su apoyo durante la realización de la tesis.

A todos los compañeros de otras universidades con los que he compartido congresos, foros y redes de investigación y, muy especialmente, a Javier Tuya y a sus fantásticos compañeros.

A lodos los investigadores europeos que me han hecho llegar opiniones y sugerencias para enriquecer este trabajar, muy especialmente a Clementine Nebut, Antonia Bertolino, Nora Kosh y Piero Fraternalli.

Al grupo de investigación IWT2. Este trabajo no se podría haber hecho sin su ayuda, desde el primero hasta el último de sus miembros.

A Rogelio, por hacer feliz a una de las personas más importantes de mi vida.

Índice

Índice de Ilustraciones

Índice de Tablas

Capítulo I. Introducción

En este capítulo se presenta una introducción del trabajo desarrollado en esta tesis. Para ello, en la primera sección se presenta un breve resumen cuyo objetivo es centrar y describir el trabajo que se desarrolla en las siguientes páginas. En la segunda sección se introducen las pruebas desde una perspectiva de ingeniería del software. A continuación, en la tercera sección se presentan con más detalle las pruebas funcionales del sistema, sobre las cuáles está basado este trabajo de tesis. Posteriormente, en la cuarta sección, se describe cuál será la estructura de este documento. Finalmente, en la quinta y última sección, se presentan unas breves conclusiones del capítulo.

1. Presentación de la tesis

De manera general, el proceso de prueba de software suele englobar, al menos, tres tareas principales: desarrollo de los casos de prueba, ejecución de los casos de prueba y análisis de los resultados [Burnstein, 2003]. El estudio en profundidad de cualquiera de estas tareas sería un tema lo suficientemente amplio como para poder centrar un trabajo de tesis. Este trabajo, en concreto, se centra en la primera tarea: cómo desarrollar casos de prueba.

Los casos de prueba desarrollados en esta tesis tendrán como objetivo la verificación de la correcta implementación de la especificación funcional del sistema bajo prueba. Por eso, este trabajo se refiere a dichos casos de prueba como casos de prueba funcionales del sistema.

La propuesta de investigación de esta tesis se basa en la generación de casos de prueba funcionales del sistema a partir de una especificación funcional desarrollada mediante requisitos funcionales. La principal idea de esta tesis es ofrecer una respuesta sistemática y automatizada a la pregunta: ¿cómo desarrollar casos de prueba que permitan la verificación del comportamiento definido en los requisitos del sistema?.

La respuesta a esta pregunta se concreta en las siguientes páginas utilizando una técnica de la ingeniería del software, transformaciones, para obtener casos de prueba funcionales a partir de la especificación funcional, y metamodelos para la especificación formal de la información de una especificación y de la información de un conjunto de casos de prueba funcionales del sistema.

2. Las pruebas en la ingeniería del software

Según la definición de Pressman [Pressman, 2010], la ingeniería del software es una disciplina o área de la informática que ofrece métodos y técnicas para el desarrollo y el mantenimiento de software de calidad que resuelva todo tipo de problemas. La IEEE Computer Society define la ingeniería del software como la aplicación de una aproximación sistemática, disciplinada y cuantificable al desarrollo, operación y mantenimiento del software [IEEE, 1990]. Estas definiciones permiten incluir dentro de la ingeniería de software un amplio número de áreas y proyectos muy diversos, desde desarrollo de sistemas operativos a la construcción de compiladores o desarrollos de sistemas en la web o en dispositivos móviles (teléfonos, e-books, tablets, etc.). Además, la IEEE Computer Society incluye las pruebas como una de las áreas de conocimiento del cuerpo de conocimiento de la ingeniería del software [SWEBOK, 2004].

Se puede definir la prueba del software como la verificación dinámica del comportamiento de un programa con un conjunto finito de casos de prueba y la comparación de su resultado con el resultado esperado.

El área de prueba en la ingeniería del software engloba varios tipos de pruebas y de etapas de prueba, por ejemplo, desde pruebas de fragmentos de código hasta pruebas de rendimiento de servidores, pruebas de sistemas empotrados en aviones militares o, incluso, como complemento de los requisitos del sistema, tal y como proponen las metodologías ágiles [Myers, 2004]. Por ello, es necesario definir una clasificación de las pruebas que ayuden a centrar con precisión el tipo de pruebas que se abordan en este trabajo de tesis.

Uno de los criterios más habituales para la clasificación de las pruebas se basa en las fases de desarrollo del ciclo de vida del software. En la figura 1.1 se muestra una clasificación basada en este criterio [Métrica-3].

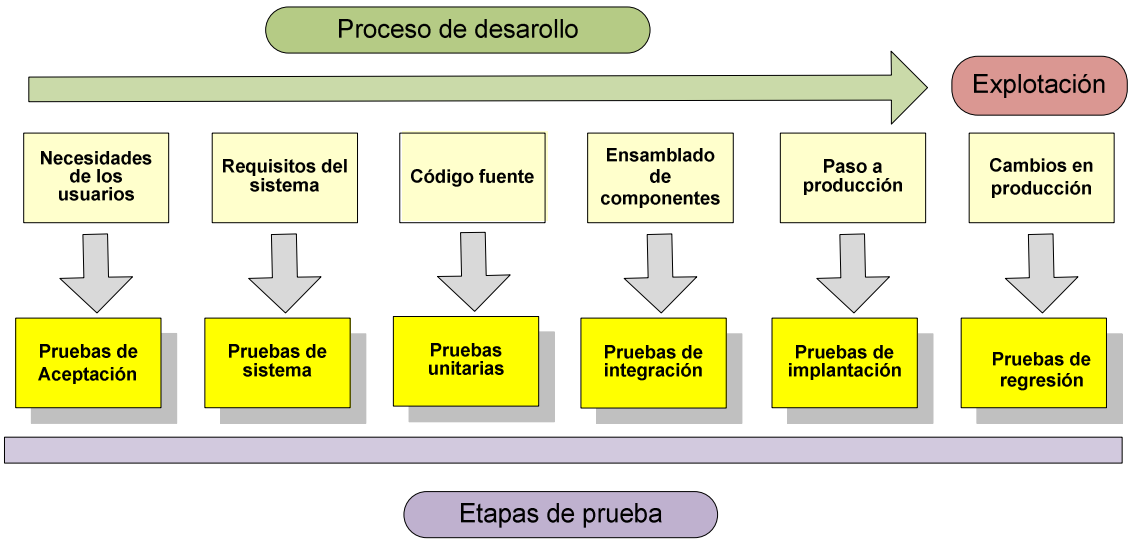


Figura 1.1.Clasificación de las pruebas en función de la etapa de desarrollo.

En la parte superior de la figura 1.1 se representan algunas de las etapas más habituales que se realizan en un proceso de desarrollo software y que guardan una relación más directa con las pruebas. Estas etapas se describen a continuación.

En la etapa de necesidades del usuario se identifican las necesidades que el sistema deberá satisfacer para lograr su aceptación. Posteriormente, estas necesidades se plasman en los requisitos del sistema. Estos requisitos se implementan mediante código ejecutable y este código se ensambla para obtener los componentes del sistema. Finalizado este proceso, el sistema pasa a producción, donde termina el proceso de desarrollo, y comienza su explotación, durante la cual el sistema recibe cambios en función del tipo de mantenimiento realizado.

Para cada una de las etapas anteriores, la figura 1.1 muestra en la fila inferior la etapa de prueba que permite verificar que los artefactos involucrados (ya sea el propio sistema o fragmentos de código) cumplen los objetivos esperados. Estos objetivos serán distintos en función de la etapa de prueba, tal y como se describe a continuación.

Las pruebas unitarias se realizan durante la codificación del sistema. En esta etapa, las pruebas verifican el diseño y el comportamiento de cada uno de los componentes del sistema una vez construidos. Las pruebas de integración verifican la correcta unión de los componentes del sistema entre sí a través de sus interfaces, y si cumplen con la funcionalidad establecida. Las pruebas de sistema verifican que el sistema cumpla con sus requisitos. Las pruebas de implantación verifican el correcto funcionamiento del sistema dentro del entorno real de producción. Las pruebas de aceptación se realizan a continuación de la implantación en el entorno de producción y verifican que el sistema cumple con las expectativas de usuarios y clientes. Por último, las pruebas de regresión se realizan después de realizar modificaciones al

sistema. El objetivo de estas pruebas es verificar que los cambios sobre un componente del sistema no generan errores adicionales en otros componentes no modificados.

La figura 1.2 muestra el mismo proceso de desarrollo y las mismas etapas de pruebas que la figura 1.1, salvo la etapa de explotación del sistema. Sin embargo estas etapas se muestran en un orden distinto como se explica a continuación.

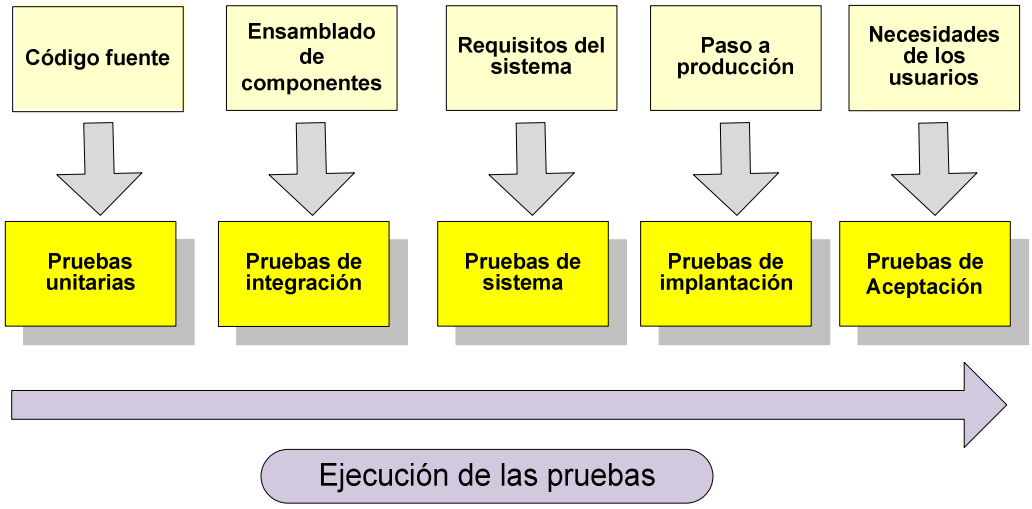


Figura 1.2. Orden de ejecución de las pruebas.

El orden de la figura 1.2 es el orden tradicional en el que se ejecutan las distintas pruebas. Así, las pruebas unitarias serán las primeras en ejecutarse, a medida que se vaya elaborando el código del sistema. Cuando el código se integre para construir los componentes del sistema, se ejecutarán las pruebas de integración. A medida que existan suficientes componentes para realizar los requisitos del sistema se realizarán las pruebas del sistema. Una vez que el sistema realice todos sus requisitos se realizarán las pruebas de implementación y, por último, las pruebas de aceptación para dar por concluido el desarrollo.

Al comparar la figura 1.1 con la figura 1.2 se puede apreciar que, si bien las pruebas no se ejecutan hasta una etapa avanzada del desarrollo, la definición de casos de prueba sí puede comenzar en una etapa más temprana. Por ejemplo, aunque la ejecución de las pruebas del sistema se realice en una etapa avanzada del desarrollo, después de definir los requisitos, realizar el diseño del sistema y codificar, ensamblar y probar los componentes, la definición de estas pruebas, al estar basadas en requisitos, puede comenzar cuando los requisitos estén disponibles.

Esta sección ha presentado una clasificación habitual de las etapas de pruebas según la ingeniería del software. En la próxima sección, se profundiza más en la etapa de pruebas del sistema y, en concreto, de las pruebas funcionales.

3. La etapa de pruebas del sistema

En la sección anterior se han descrito brevemente las etapas de pruebas más comunes. Dado que este trabajo de tesis se centra en las pruebas del sistema, esta sección describe esta etapa con más profundidad.

Como ya se ha mencionado en la sección anterior, las pruebas del sistema constituyen la primera etapa de pruebas donde se prueba el sistema en su conjunto, en lugar de probar cada una de sus partes de manera separada. Dentro de la etapa de pruebas del sistema se pueden identificar distintos grupos de pruebas con distintos objetivos, por lo que se hace necesario presentar una clasificación de las pruebas del sistema. Una posible clasificación de tipos de prueba que puede realizarse en la etapa de pruebas de sistema se muestra en la tabla 1.1 [Métrica-3].

Tabla 1.1. Tipos de pruebas de sistema.

Tipo de prueba de sistema	Resumen
Pruebas funcionales	Verifican la correcta implementación de las especificaciones funcionales del sistema.
Pruebas de comunicaciones	Verifican el funcionamiento de distintos subsistemas a través de sus interfaces
Pruebas de rendimiento	Verifican los tiempos de respuesta del sistema
Pruebas de volumen	Monitorizan el funcionamiento del sistema bajo distintas cargas de trabajo / datos esperados
Pruebas de sobrecarga	Monitorizan el funcionamiento del sistema bajo distintas cargas de trabajo / datos no esperados.
Pruebas de disponibilidad de datos	Verifican la respuesta del sistema ante posibles fallos
Pruebas de facilidad de uso	Verifican la adaptabilidad del sistema a los distintos usuarios esperados
Pruebas de operación	Verifican la correcta implementación de los procedimientos de operación del sistema.
Pruebas de seguridad	Verifican los mecanismos de control de acceso al sistema para evitar alteraciones indebidas en los datos.

Este trabajo de tesis se centra en las pruebas del sistema, y, más concretamente, en las pruebas funcionales del sistema. Como se define en la tabla 1.1, este tipo de pruebas tiene como objetivo definir casos de prueba que permitan evaluar la correcta implementación de la

especificación funcional, realizar dichas pruebas sobre el sistema implementado y evaluar la correcta implementación o no.

Tabla 1.2. Ejemplo de requisito funcional.

Requisito funcional: Alta de nuevo elemento.

Descripción:
Este requisito funcional describe el comportamiento del sistema cuando un usuario desea dar de alta un nuevo elemento.

Precondiciones:
No.

Poscondiciones:
El elemento está dado de alta en el sistema.

Secuencia principal

1. El usuario solicita dar de alta un nuevo elemento en el sistema.
2. El sistema solicita la información del elemento.
3. El usuario introduce la información del elemento.
4. El sistema da de alta al elemento.

Secuencia alternativa

2. Si se ha superado el límite de elementos del sistema, entonces se muestra un mensaje de error y el resultado es finalizar la ejecución.
3. Si la información es incorrecta, entonces se muestra un mensaje de error y el resultado es ir al paso 2.

Según el glosario de la IEEE [IEEE, 1990], un caso de prueba es un conjunto formado por entradas de prueba, condiciones de ejecución y resultados esperados, desarrollados para un objetivo particular. En el caso concreto de un caso de prueba funcional del sistema, esta información se obtendrá de la especificación funcional del sistema. Así, un caso de prueba funcional del sistema, será un fragmento del comportamiento descrito en un requisito funcional perteneciente a la especificación funcional del sistema. Durante la ejecución de un caso de prueba funcional, será necesario contar con un elemento (bien una herramienta de automatización o una persona) que interactúe con el sistema mediante la interfaz adecuada

para realizar el requisito funcional. La misión de este elemento será realizar los pasos del caso de prueba, verificando que el sistema exhibe las repuestas esperadas.

A modo de ejemplo, se muestra en la tabla 1.2, un posible requisito funcional tomado de la especificación funcional de un sistema (que se utilizará de nuevo más adelante a modo de ejemplo).

A partir del requisito de ejemplo de la tabla 1.2, se puede definir, como parte del mismo ejemplo, un posible caso de prueba funcional del sistema en la tabla 1.3.

Tabla 1.3. Ejemplo de prueba funcional del sistema.

Caso de prueba *Estado inicial:* No. *Estado final:* El elemento está dado de alta en el sistema. *Pasos del caso de prueba* 1. El usuario solicita dar de alta un nuevo elemento en el sistema. 2. El sistema solicita la información del elemento (el número de elementos está por debajo o igual al límite de elementos en el sistema). 3. El usuario introduce la información del elemento (la información es correcta). 4. El sistema da de alta al elemento.

En la tabla 1.3 se puede ver cómo la prueba tiene todos los elementos definidos para una prueba en los párrafos anteriores (conjunto de entrada, condiciones de ejecución y resultados esperados). Las entradas, en este ejemplo, son las interacciones que el usuario hace con el sistema. Las condiciones de ejecución se pueden ver en el estado inicial, aunque en este ejemplo no existe ninguna. Los resultados esperados se ven en las respuestas del sistema a los pasos del usuario y en el estado final.

El objetivo de este trabajo de tesis es conseguir la definición de un proceso, lo más automático posible, para sistematizar la generación de pruebas funcionales a partir de la definición de los requisitos funcionales del sistema.

Una vez definido el contexto de prueba en el que se enmarca este trabajo de tesis, en la siguiente sección se presenta la estructura de este trabajo.

4. Estructura de la tesis

A continuación se presenta la estructura de esta tesis describiendo los próximos capítulos.

En el capitulo 2 se ofrece una visión general de cómo es tratada la generación de pruebas funcionales del sistema en el ámbito de la investigación actual. Esta visión general permitirá analizar cómo está definida la generación de pruebas funcionales del sistema en la literatura existente.

A continuación, el capitulo 3 expone el planteamiento del problema. Como un paso natural, este capítulo también define los objetivos a satisfacer por la tesis en base a la descripción del problema presentado. En este capítulo se definirá con más detalle los antecedentes e influencias de este trabajo, así como las técnicas utilizadas para la definición del proceso de transformación y la información involucrada.

Planteado el problema y los objetivos a conseguir, se comienza a trabajar en ellos en el capítulo 4 y en el capitulo 5. En ambos se presenta, de manera formal, la información necesaria para hacer un tratamiento adecuado, tanto de los requisitos funcionales de origen como de los casos de pruebas funcionales obtenidas.

Una vez definida esta información, el capítulo 6 presenta la especificación del proceso de generación de casos de pruebas funcionales del sistema a partir de la información de los requisitos funcionales.

En el capítulo 7, la información y el proceso descritos en los 3 capítulos anteriores se implementa en una herramienta CASE llamada MDETest que permite obtener de manera automatizada un conjunto de casos de prueba funcionales del sistema a partir de los requisitos funcionales. En este capítulo también se presentan distintos formatos para representar la información de los requisitos funcionales y de las pruebas funcionales del sistema.

Finalmente, el capitulo 8 recoge las conclusiones del trabajo y las líneas futuras de investigación que se han abierto en base a los resultados obtenidos

Este trabajo de tesis, además, ofrece 5 anexos. En el primero de ellos se presenta un glosario con las definiciones de los términos más técnicos del ámbito de las pruebas y del desarrollo orientado a modelos. En el segundo anexo se presenta el manual de usuario de la herramienta CASE MDETest. En el tercer anexo se ofrece un ejemplo, basado en un proyecto web real, y se describe la aplicación del proceso de generación de pruebas funcionales del sistema. El cuarto capítulo presenta los perfiles de UML definidos para recoger información presentada en los capítulos 4 y 5. El concepto de perfil de UML se definirá más adelante en

este trabajo de tesis. Finalmente, el quinto y último anexo engloba todas las publicaciones realizadas en torno a la tesis durante los años de su realización.

5. Conclusiones

Se ha visto a lo largo de este capítulo, cómo este trabajo de tesis estudiará, en el capítulo 2, y definirá, en los capítulos del 3 en adelante, un proceso para la generación de pruebas funcionales del sistema a partir de la especificación funcional. En este capítulo se ha definido el nivel de pruebas del sistema y, más concretamente, las pruebas funcionales del sistema que serán el tipo de pruebas abordadas a lo largo de este trabajo.

Capítulo II. Estado del Arte

Esta tesis está motivada por la necesidad de buscar un proceso para la generación de un conjunto de casos de pruebas funcionales del sistema a partir de su especificación funcional. Como primer paso para proponer una aportación original, es necesario conocer el estado de la situación actual mediante la localización de las propuestas existentes actualmente en la literatura y extraer de dicho estado aquellos aspectos que podrán servir como base para guiar la definición de los objetivos del trabajo de tesis. Este capítulo expone el estudio de la situación actual, para ello, en la primera sección se describe el contexto del mismo, en el que se fijarán aquellos parámetros que dirigirán la búsqueda y seleccionarán las propuestas a analizar. A continuación, en la segunda sección se caracterizan y describen las propuestas relevantes encontradas. Posteriormente, en la tercera sección, se compara y se analiza la información presente en dichas propuestas. Finalmente, en la sección cuarta se exponen las conclusiones de este capítulo.

1. Contexto del estudio de la situación actual.

Esta sección se divide en dos secciones. En la primera, se exponen los antecedentes y trabajos anteriores realizados para la definición del estado actual. En la segunda sección se describe cómo se han actualizado dichos trabajos anteriores para su presentación en este trabajo de tesis.

1.1. Antecedentes del estudio de la situación actual.

En el momento de comenzar a desarrollar el trabajo que se presenta en este documento, se buscaron estudios que indicaran cuál era el estado del arte. El único trabajo encontrado fue la referencia [Denger et-al, 2003]. Esto motivó la realización de un estudio de la situación actual propio, que se ha venido manteniendo y actualizando durante todo el tiempo de realización de esta tesis.

Los objetivos propuestos para dicho estudio de la situación actual fueron, en primer lugar, conocer cuáles son las propuestas existentes y, en segundo lugar, resumir la información relevante, de tal manera que se puedan conocer cuáles son los aspectos comunes existentes en todas las propuestas, cuáles son aquellos aspectos más novedosos que introduce cada una

de ellas con respecto a las demás y cuáles son los aspectos cubiertos o los aspectos por cubrir del tema en estudio.

A medida que se iba avanzando en el trabajo de investigación, dischо estudio fue hecho público en varias publicaciones desarrolladas por los participantes en este trabajo. Las principales referencias son: [Gutiérrez et-al, 2005] y [Gutiérrez et-al, 2006]. Finalmente, el resultado del estudio de la situación actual fue publicado en [Escalona et-al, 2011].

A pesar del tiempo transcurrido entre las distintas publicaciones de nuestros trabajos, y de que el número de propuestas analizadas ha ido incrementándose, los resultados y conclusiones se han mantenido estables. Por ello, esta sección presenta un resumen de dichos trabajos. Todo este conjunto de trabajos se referenciarán en las siguientes secciones bajo el nombre de estudios previos.

1.2. Estudio de la situación actual

Esta sección define brevemente el formato y la caracterización de las propuestas presentadas en este capítulo. Para más información sobre las propuestas o una caracterización más detallada se puede acudir a las referencias de los estudios anteriores.

En este capítulo, la información que se ha considerado relevante de una propuesta será la información de entrada y cómo está definida esta información, la información de salida y cómo está definida, el proceso que utiliza para obtener las pruebas y si se propone un procedimiento automático. Por ello, se define en la tabla 2.1 un esquema de caracterización que utilizael mismo formato que el formato utilizado en los estudios previos. Este esquema de caracterización permitirá resumir cada una de las propuestas en la próxima sección y analizarlas más adelante en este capítulo.

Tabla 2.1. Esquema de caracterización.

Atributos	Descripción
Información de entrada	Indica qué información de entrada usa y cómo está definida
Información de salida	Indica qué información de salida produce y cómo está definida dicha información
Tipo de proceso	Indica cómo se generan las pruebas a partir de la información de entrada.
Automático	Indica si el proceso de desarrolla de manera automática o no.

El atributo *información de entrada* describe cómo están definidos los requisitos funcionales que la propuesta utiliza y si, además, utiliza artefactos o información adicional. El atributo *información de salida* describe cómo están definidos los casos de pruebas generados y cuál es su contenido. El atributo *tipo de proceso* indica qué técnica se utiliza para procesar los requisitos funcionales y generar los casos de prueba. Finalmente, el atributo *automático* indica si la propuesta se puede realizar de manera automática con una herramienta.

La mayoría de las propuestas están basadas en técnicas de análisis de caminos. De manera breve, el análisis de caminos consiste en buscar todos los caminos posibles a través de una máquina de estados o artefacto similar. Otras están basadas en Cathegory-Partition Method [Ostrand et-al, 1988], [Balcer et-al, 1990]. De manera breve, este método consiste en buscar las categorías (puntos de variabilidad en un requisito funcional) y particiones (subdominios de las categorías.) de los requisitos funcionales y calcular las posibles combinaciones de las mismas

A continuación, en la siguiente sección, se resumen brevemente las propuestas encontradas y se presentan sus esquemas de caracterización tomadas de los estudios previos.

2. Trabajos analizados

Se ha realizado una selección entre las 24 propuestas incluidas en los estudios previos (sección 1.1), ya que alguna de ellas tienen esquemas de caracterización muy similares y aportan poco a la hora de estudiar el estado el arte desde un punto de vista cualitativo. A continuación, se describen brevemente las propuestas más relevantes de los estudios previos. Cada una de las propuestas incluye una breve descripción y una tabla con su esquema de caracterización visto en la sección anterior.

2.1. Software Requirements and Acceptance Testing (1.997)

Este trabajo [Hsia et-al, 1997] está centrado en la generación de pruebas de aceptación. Sin embargo, el ámbito de las pruebas de aceptación presentado en dicho trabajo es similar a la definición de prueba funcional del sistema recogida en el capítulo 1. Esta propuesta se divide en dos bloques, en el primero se realiza un resumen del proceso de elicitación de requisitos (llamados escenarios). Este proceso se expone con más detalle en [Hsia et-al, 1994]. El segundo bloque consiste en la construcción de pruebas de aceptación a

partir de los escenarios obtenidos en el bloque anterior. Para finalizar, la tabla 2.2 muestra el esquema de caracterización de esta propuesta.

Tabla 2.2. Esquema de caracterización de [Hsia et-al, 1997].

Atributos	Valores
Información de entrada	Requisitos funcionales como árboles de escenarios (*).
Información de salida	Caminos a través de una máquina de estados
Tipo de proceso	Análisis de escenarios (máquina de estados)
Automático	No.

(*) - Un árbol de escenario es una estructura similar a texto tabulado, en el que los posibles caminos alternativos se indican con flechas.

2.2. Extended Use Case Test Design Pattern (1.999)

El punto de partida de esta propuesta [Binder, 1999] son requisitos funcionales descritos en lenguaje natural y con formato tabular, extendidos con variables operacionales. Al final de la aplicación de esta propuesta se obtiene una tabla con todos los posibles casos de prueba para verificar cada combinación de variables operacionales.

Tabla 2.3. Esquema de caracterización de [Binder, 1999].

Atributos	Valores
Información de entrada	Requisitos funcionales en formato tabular
Información de salida	Tabla con todas las combinaciones posibles de variables
Tipo de proceso	Método de Categoría-Partición
Automático	No.

Para finalizar, la tabla 2.3 muestra el esquema de caracterización de esta propuesta.

2.3. Automated Test Case Generation from Dynamic Models (2.000).

Esta propuesta [Fröhlich et-al, 1999], [Fröhlich et-al, 2000] se puede dividir en dos bloques. En el primer bloque se realiza una traducción de un requisito funcional a un diagrama

de estados. En el segundo bloque se genera un conjunto de pruebas de sistema a partir de dicho diagrama de estados. Al final del proceso se obtiene un conjunto de transiciones posibles sobre un diagrama de estados, las cuales están expresadas mediante operadores que indican cambios en el estado del sistema. Además, se obtienen las proposiciones iniciales y finales de dicho conjunto. Dicho conjunto se podrá ejecutar sobre la aplicación para comprobar si se producen las mismas transiciones y el resultado es el esperado.

Tabla 2.4. Esquema de caracterización de [Fröhlich et-al, 2000].

Atributos	Valores
Información de entrada	Requisitos funcionales en formato tabular.
Información de salida	Recorridos sobre una máquina de estados.
Tipo de proceso	Análisis de caminos sobre una máquina de estados
Automático	No.

Como conclusión, en la tabla 2.4 se presenta el esquema de caracterización de esta propuesta.

2.4. Testing from Use Cases Using Path Analysis Technique (2002)

El punto de partida de esta propuesta [Naresh, 2002] es un requisito funcional descrito en lenguaje natural. Al final del proceso de generación de pruebas se obtiene una tabla con todos los caminos posibles que deben ser probados para un requisito funcional. Estos caminos también se describen en lenguaje natural.

Tabla 2.5. Esquema de caracterización de [Naresh, 2002].

Atributos	Valores
Información de entrada	Requisitos funcionales en formato tabular
Información de salida	Caminos sobre un diagrama de flujo
Tipo de proceso	Análisis de caminos
Automático	No.

En la tabla 2.5 se muestra el esquema de caracterización de esta propuesta.

2.5. Test Cases from Use Cases (2.002).

Esta propuesta [Heumann, 2002] no impone un modelo formal de representación de un requisito funcional, pero sí enumera los elementos mínimos que deben aparecer en un requisito funcional. Al final del proceso descrito en esta propuesta se obtiene una tabla donde se describe en lenguaje natural y formato tabular, los casos de prueba para verificar la correcta implantación del requisito funcional.

Tabla 2.6. Esquema de caracterización de [Heumann, 2002].

Atributos	Valores
Información de entrada	Requisitos funcionales en formato tabular
Información de salida	Escenarios
Tipo de proceso	Análisis de caminos
Automático	No

El esquema de caracterización de esta propuesta se muestra en la tabla 2.9.

2.6. Quality Web Systems (2.002)

Esta propuesta está recogida en el capítulo II del libro del mismo título [Dustin et-al, 2002]. En ella se define un proceso para la elicitación de requisitos funcionales y la gestión de otros tipos de requisitos, como requisitos de rendimiento o accesibilidad, para sistemas web. El punto de partida del proceso de generación de pruebas funcionales del sistema propuesto en este libro son los requisitos funcionales definidos siguiendo las plantillas incluidas en esta propuesta. El resultado son casos de prueba funcionales del sistema definidos mediante una plantilla también definida en este libro.

Tabla 2.7. Esquema de caracterización de [Dustin et-al, 2002].

Atributos	Valores
Información de entrada	Requisitos funcionales en formato tabular. Plantilla definida en la propia propuesta.
Información de salida	Escenarios. Plantilla definida en la propia documentación
Tipo de proceso	Análisis de caminos
Automático	No.

La tabla 2.7 muestra el esquema de caracterización de esta propuesta.

2.7. SCENario-Based Validation and Test of Software (2.003)

SCENT [Glinz et-al, 1999] y [Ryser et-al, 2003] es un método basado en escenarios para elicitar, validar y verificar requisitos funcionales y desarrollar casos de pruebas a partir de los mismos. SCENT puede dividirse en dos partes, la parte de creación y refinado de escenarios y la parte de generación de pruebas. El punto de partida de la propuesta de generación de casos de prueba es un conjunto de escenarios definidos con el formato propuesto en SCENT. El resultado de esta propuesta es un conjunto de casos de prueba para verificar la correcta implementación de dichos escenarios.

Tabla 2.8. Esquema de caracterización de [Ryser et-al, 2003].

Atributos	Valores
Información de entrada	Requisitos funcionales en formato tabular. Plantilla definida en la propia propuesta.
Información de salida	Caminos de una máquina de estados.
Tipo de proceso	Análisis de caminos sobre una máquina de estados
Automático	No.

La tabla 2.8 muestra el esquema de caracterización de esta propuesta.

.

2.8. Requirement Base Testing (2.003).

La propuesta Requirement Base Testing (RBT) [Mogyorodi, 2001], [Mogyorodi, 2002] y [Mogyorodi, 2003] se divide en dos actividades principales: revisiones de requisitos y generación y revisión de casos de prueba. El punto de partida de esta propuesta consiste en un conjunto de requisitos escritos en lenguaje natural y formato tabular. El resultado de esta propuesta es un conjunto de casos de prueba expresados en lenguaje natural, compuestos de una serie de causas, o estado del sistema, y sus efectos, o resultados esperados.

Tabla 2.9. Esquema de caracterización de [Mogyorodi, 2003].

Atributos	Valores
Información de entrada	Requisitos funcionales en formato tabular.
Información de salida	No queda claro en la documentación encontrada.
Tipo de proceso	Análisis de caminos sobre diagramas de causa-efecto.
Automático	Sí. Herramienta propietaria

Para finalizar, la tabla 2.9 presenta el esquema de caracterización de esta propuesta.

2.9. Requirements to Testing in a Natural Way (2.004)

La propuesta Requirements to Testing in a Natural Way (RETNA) describe un analizador de lenguaje que puede ser utilizado para generar pruebas del sistema [Boddu et-al, 2004]. Para finalizar, la tabla 2.10 muestra el esquema de caracterización de esta propuesta.

Tabla 2.10. Esquema de caracterización de [Boddu et-al, 2004].

Atributos	Valores
Información de entrada	Texto plano con características concretas (*)
Información de salida	No indicado en la documentación
Tipo de proceso	Análisis de caminos sobre una máquina de estados
Automático	Sí

(*) Las características concretas definidas en la propuesta imponen que el texto debe ser un único párrafo de no más de 399 caracteres.

2.10. A Model-based Approach to Improve System Testing of Interactive Applications (2.004)

El punto de partida de esta propuesta [Ruder et-al, 2004] y [Ruder, 2004] son requisitos funcionales en lenguaje natural. Al final de la aplicación de esta propuesta se obtiene un conjunto de pruebas funcionales del sistema.

Tabla 2.11. Esquema de caracterización de [Ruder et-al, 2004].

Atributos	Valores
Información de entrada	Requisitos funcionales definidos de manera tabular.
Información de salida	Caminos y conjuntos de particiones.
Tipo de proceso	Análisis de caminos sobre un diagrama de actividades y el método Categoría-Partición.
Automático	Sí

El esquema de caracterización se presenta en la tabla 2.11.

2.11. An Automatic Tool for Generating Test Cases from the Systems Requirements (2.007)

Esta propuesta [Ibrahim et-al, 2007], [Ismail et-al, 2007] presenta una herramienta (llamada GenTCase) para la generación de casos de prueba de manera automática. La herramienta permite dibujar un diagrama UML de requisitos funcionales y, completar cada requisito funcional con un flujo de eventos y un diagrama de secuencia UML para cada requisito funcional. A continuación, la herramienta permite generar un conjunto de casos de prueba. El resultado es un archivo de texto con los casos de prueba generados. El esquema de caracterización se presenta en la tabla 2.12.

Tabla 2.12. Esquema de caracterización de [Ibrahim et-al, 2007].

Atributos	Valores
Información de entrada	Texto tabular y diagramas UML de casos de uso y secuencia (*)
Información de salida	Descripción textual a partir del texto tabular.
Tipo de proceso	Análisis de caminos
Automático	Sí

(*) – No se ha identificado que el diagrama de secuencia participe en el proceso de generación de pruebas definido en esta propuesta.

Hasta aquí se han descrito las propuestas estudiadas. A continuación se expone un breve análisis de la situación actual a partir de estas propuestas.

3. Análisis de la situación actual

Los esquemas de caracterización de las secciones anteriores han presentado una visión general de cómo se aborda en la literatura existente la generación de pruebas funcionales del sistema a partir de los requisitos funcionales. Ahora, esta sección se centra en tratar de exponer cuál es la situación actual a partir de las propuestas encontradas. Los resultados encontrados coinciden con los resultados expuestos en los estudios previos, por lo que esta sección ofrece un resumen de dichos resultados.

Las propuestas vistas en la sección anterior, utilizan requisitos funcionales definidos como texto tabular (como el ejemplo de requisito funcional visto en el capítulo anterior), incluso aquellas propuestas que definen un proceso más formal basado en máquinas de estados o similares. Sin embargo, parece no existir un consenso sobre qué información incorporar a los requisitos para la generación de pruebas funcionales del sistema, aunque sí se han detectado aspectos comunes en las distintas plantillas utilizadas por las propuestas.

Se han identificado dos técnicas en todas las propuestas descritas en los estudios previos. Estas técnicas son análisis de caminos y el Método de Categoría-Partición. Ambas técnicas se describen brevemente a continuación.

El análisis de caminos es una técnica clásica de prueba, a menudo mencionada como una técnica de caja blanca, en la que se recorre un grafo explorando todos sus caminos. Dicho grafo puede representar el comportamiento del sistema definido mediante un requisito o bien una representación más formal como una máquina de estados. El Método de Categoría-Partición [Ostrand et-al, 1988] es una técnica consistente en identificar un conjunto de categorías, identificar su dominio y dividirlo en particiones y calcular combinaciones de dichas particiones

Se ha detectado una gran disparidad en los resultados de salida, es decir, en la manera en que están definidas las pruebas funcionales del sistema obtenidas. Algunas propuestas las expresan en base al texto de los requisitos funcionales mientras que otras propuestas las expresan en base a transiciones de una máquina de estados. De la misma manera, existe un número pequeño de propuestas que ofrece información sobre los valores de prueba.

La mayoría de las propuestas estudiadas trabajan con análisis de caminos, bien directamente sobre el requisito o bien expresándolo como un diagrama de estados o una notación similar. También existe un número menor de propuestas que se decantan por el

estudio de valores de prueba, sin embargo, todas las que lo hacen se basan en el Método de Categoría-Partición. No se ha encontrado ninguna propuesta que utilice otra estrategia distinta a las dos mencionadas.

También se puede apreciar en las propuestas estudiadas que muchas no definen la estructura de artefactos de salida, sino que se limitan a aplicar una técnica, habitualmente el análisis de caminos, sin entrar a desarrollar cómo organizar dichos resultados.

Aquí termina el análisis de la situación actual. En la próxima sección se presentan las conclusiones extraídas del estudio de la situación actual visto en la sección anterior y del análisis realizado en esta sección.

4. Conclusiones

Este capítulo ha permitido alcanzar los objetivos expuestos en la introducción. En concreto, ha permitido conocer las propuestas existentes y los estudios comparativos desarrollados y ha resumido la información más relevante mediante el uso de los esquemas de caracterización.

En este capítulo se ha puesto de manifiesto que, aunque existe un número significativo de propuestas, sin ser muy alto, la mayoría de ellas inciden en las mismas ideas. También queda patente en estos trabajos la viabilidad de abordar la generación de casos de prueba funcionales a partir de los requisitos funcionales del sistema y de desarrollar aplicaciones de soporte.

En este sentido, se plantean varias cuestiones. La primera cuestión será: ¿de qué manera se puede definir formalmente la información que los requisitos funcionales deben contener para aplicar un proceso de generación de pruebas funcionales del sistema? ¿Y de qué manera se puede definir formalmente la información que los casos de prueba contendrán? La segunda cuestión será: ¿de qué manera se puede definir formalmente el proceso sistemático de generación de pruebas funcionales del sistema a partir de los requisitos funcionales? La tercera cuestión será: ¿pueden tratarse de manera automática requisitos funcionales para obtener pruebas del sistema a partir de ellos? ¿Hasta qué punto es automatizable el proceso sistemático formal definido?

La definición y descripción de estos aspectos se realizará en el capítulo siguiente con el fin de introducir y acatar el problema a resolver en los capítulos posteriores.

Capítulo III. Planteamiento del Problema

La presentación del estado del arte del capítulo anterior ha servido para caracterizar cómo la generación de pruebas funcionales del sistema a partir de requisitos funcionales está siendo tratada en el momento de desarrollar este trabajo de tesis. El análisis de esta situación ha llevado a la base para concretar el problema a resolver.

La organización de este capítulo se describe a continuación. En la primera sección se detallan cuáles son los aspectos susceptibles de mejora que se detectan en la generación de pruebas del sistema en base a los trabajos existentes. A partir de aquí, en la segunda sección, se presentan los objetivos que se pretenden alcanzar durante el desarrollo de este trabajo de tesis. En la tercera sección se definen las influencias que conforman el contexto de la solución propuesta. En la cuarta sección se describen las tecnologías que se han seleccionado para definir la solución. En la quinta sección se define la estructura de la solución a desarrollar en base a las influencias y tecnologías vistas. Finalmente, en la sexta sección se exponen las conclusiones.

1. Planteamiento del problema.

El primer paso para definir el trabajo a abordar en los próximos capítulos es plantear el problema que se desea resolver. Para ello, en esta sección se toma el estudio de las propuestas, realizado en el capítulo anterior, y las conclusiones extraídas del mismo como base para definir un conjunto de aspectos que podrían ser mejorados.

1.1. Definición de requisitos funcionales y de pruebas funcionales del sistema

Uno de los aspectos relevantes del análisis de la situación actual visto en el capítulo anterior ha sido la definición de los requisitos funcionales utilizados como base para la generación de pruebas del sistema. Los requisitos funcionales de un sistema serán la fuente a partir de la cual se define el proceso de generación de pruebas funcionales, por lo que la información que contengan y la manera en que dicha información esté descrita tiene un impacto relevante en dicho proceso. El estudio de la manera de definir estos artefactos y cómo se puede formalizar dicha definición, es una tarea relevante para la generación de pruebas funcionales.

El problema, en este caso, surge con la disparidad de notaciones y plantillas para la definición de los requisitos funcionales. Esta disparidad evita la compatibilidad de técnicas, ya

que unas técnicas pueden requerir una información con una estructura determinada, que otra representación concreta de requisitos no ofrezca. Además, unas técnicas pueden profundizar o realizar un trabajo adicional debido a la información presente, que otras no pueda realizar. También impide la integración de los procesos de generación de pruebas en un proceso más amplio de ingeniería del software que ya tenga establecida una metodología de requisitos. Más aún, a partir de las propuestas estudiadas, se pueden plantear como preguntas o dudas a resolver, qué es lo que sucede si se modifica el formato de los requisitos funcionales y cómo afecta esto al proceso de generación de pruebas, si los requisitos deben contener toda la información vista en las propuestas, o qué sucedería si se deseara añadir más información adicional en la definición de un requisito funcional. Estas preguntas resaltan puntos sobre los que se puede desarrollar un trabajo adicional para completarlos.

Un razonamiento análogo puede exponerse a la hora de abordar la definición de las pruebas funcionales generadas a partir de los requisitos funcionales. Como se ha visto en el capítulo anterior, muy pocos autores definen con precisión las pruebas obtenidas. En la mayoría de los casos la información de pruebas se expone en plantillas de texto sin que haya ninguna indicación sobre cómo cumplimentarlas o su semántica, ni ninguna otra referencia a trabajos adicionales que aborden esta cuestión. Por ello, si bien es necesario definir las pruebas de una manera precisa y formal para tener un proceso de generación de pruebas del sistema automatizable, no parece adecuado imponer un formato determinado, sino dar la oportunidad de que distintos usuarios o equipos utilicen los formatos (ya sean distintas plantillas de texto, formatos gráficos, etc.) con los que se sientan más cómodos.

1.2. Procesamiento de los requisitos

Otro aspecto relevante del análisis de la situación actual es el procesamiento al que las propuestas de generación de pruebas funcionales del sistema someten a los requisitos. Como se ha podido ver en el estudio de la situación actual, varias propuestas, ([Binder, 1999], [Dustin et-al, 2002], [Heumann, 2002], [Narres, 2002] entre otros; para una lista exhaustiva consultar los esquemas de caracterización del capítulo anterior), abordan la generación de casos de prueba directamente desde los requisitos funcionales definidos como texto. Otras propuestas ([Hsia et-al, 1997], [Boddu et-al, 2004], [Fröhlich et-al, 2000], [Ruder, 2004] entre otros; para una lista exhaustiva consultar los esquemas de caracterización del capítulo anterior), en cambio, dan un paso en la formalización de los requisitos funcionales, construyendo primero un modelo que represente el comportamiento definido en un requisito funcional como paso a la generación de pruebas.

Se ha expuesto en el análisis de la situación actual, que el desarrollar un modelo con un mayor nivel de formalismo a partir de un requisito funcional definido como texto en una plantilla, parece ser una opción adecuada para alcanzar una mayor sistematización del proceso de generación de pruebas y poder desarrollar también herramientas de soporte. Sin embargo, tampoco existe consenso entre las propuestas que abordan esta aproximación, por lo que se puede plantear, como parte del problema a resolver, si sería posible definir un proceso de construcción de la representación formal semiautomática o completamente automática, cuál es la notación más adecuada y si se podría utilizar algún estándar, como UML, para elaborarlo y qué ventajas e inconvenientes supondría esto.

Otro aspecto a tener en cuenta es que, como se ha mencionado en la sección anterior, en el momento de elegir una representación concreta de los requisitos, el proceso de generación de pruebas dependerá de dicha representación, dejando fuera a aquellos requisitos que pudieran tener una representación distinta.

1.3. Sistematización y automatización de los procesos de generación de casos de prueba

Un proceso de generación de pruebas funcionales del sistema a partir de requisitos funcionales debe describir qué información se va a tomar de los requisitos y cuándo se va a tomar, cómo se va a manipular y transformar dicha información y qué dependencias existen con otras informaciones y qué información se va a obtener al final. Como se ha visto en el estudio comparativo, se han encontrado una gran variedad de criterios entre las distintas propuestas, no sólo en su manera de trabajar (algunas a partir de requisitos y otras a partir de modelos) sino también en la manera de describir los tres grandes aspectos anteriormente mencionados. Es necesario por tanto, unificar los criterios y definir la información de los requisitos funcionales, la información de los casos de prueba generados y la descripción del proceso de una manera formal y homogénea.

La pregunta que se puede plantear, por tanto, es de qué manera se podría definir con precisión los aspectos anteriores y, en concreto, cómo se podría definir el proceso de generación de pruebas funcionales del sistema a partir de requisitos funcionales de manera que se evite, en la medida de lo posible, interpretaciones incorrectas, ambigüedades u omisiones. También se plantea si dicho proceso es automatizable, hasta qué nivel es automatizable, si lo es todo el proceso o si lo es solo una parte.

1.4. Otros aspectos relevantes

En esta sección se definen otros aspectos que también se han encontrado en las propuestas analizadas y que resultan relevantes en un proceso de generación de pruebas funcionales del sistema.

La priorización de las pruebas consiste en establecer un indicador para cada prueba que indique su relevancia. Este indicador de prioridad determinará qué pruebas deben ser ejecutadas en primer lugar y cuáles son candidatas a pasar a engrosar el catálogo de pruebas de regresión o serán tomadas como posible base de las pruebas de aceptación. El aspecto de la prioridad cobra un papel importante en un proceso de generación de pruebas sistemático y automático, debido a la posible explosión combinatoria del número de casos de prueba. Computacionalmente sería posible explorar un rango muy alto de combinaciones y alternativas en la ejecución de requisitos funcionales para la construcción de pruebas del sistema. Por ejemplo, si un sistema de información imaginario posee doscientos requisitos y de cada requisito funcional se puede extraer del orden de veinte pruebas del sistema, el número total de pruebas del sistema sería de cuatro mil. Cifras como las de este ejemplo son lo suficientemente pequeñas como para que un computador actual pueda manejarlas sin ningún problema, pero se convierten en cifras elevadas si, por ejemplo, las pruebas son realizadas manualmente por un equipo específico o las pruebas se codifican y complementan con conjuntos de datos manualmente. Por este motivo sería útil plantear la posibilidad de incorporar un mecanismo de priorización que, ante la falta de recursos para la ejecución de todas las pruebas, indique cuáles deben ser las primeras en recibir asignación de recursos.

El segundo aspecto relevante a mencionar es el desarrollo de herramientas de soporte. Como ya se ha visto en las secciones anteriores, existen propuestas que parten directamente de los requisitos funcionales escritos como texto [Binder, 1999] o [Heumann, 2002]. Si bien es posible automatizar esta manera de trabajar, y de hecho existen propuestas que proponen cómo hacerlo [Ruder, 2004] y [Ruder et-al, 2004], esta aproximación introduce un número considerable de restricciones que reducen su aplicación práctica. En cambio, otras referencias [Boddu et-al, 2004], [Fröhlich et-al, 1999] y [Fröhlich et-al, 2000] que se basan en un modelo de comportamiento más formal que los requisitos funcionales en texto, son buenas candidatas para disponer de herramientas de soporte. De hecho, como ya se ha mencionado anteriormente, son estas propuestas las que citan en sus trabajos herramientas de soporte (aproximadamente la mitad de las propuestas).

Un tercer aspecto relevante es el estudio del uso de estándares. Existen en la actualidad una serie de estándares ampliamente probados y utilizados que aportan un considerable valor para evitar el tener que reinventar conceptos básicos y que proporcionan

una lengua común a todos los especialistas en este campo. Sin embargo, se ha puesto de manifiesto en el estudio de las propuestas existentes que no todas utilizan estándares, sino que varias de ellas definen notaciones propias, como [Naresh, 2002], lo cual supone un hándicap importante a la hora de formalizar artefactos y procesos, compartir información con otras personas y en otros foros, utilizar herramientas existentes y completar el trabajo con otros resultados valiosos de la comunidad.

Un cuarto y último aspecto a mencionar son las relaciones de un requisito funcional con otros requisitos funcionales. Este tipo de relaciones, en general, se utilizan para reutilizar el comportamiento definido en otros requisitos funcionales, ganando en concisión, ya que evita tener que repetir varias veces la descripción de un mismo comportamiento. Estas relaciones están muy presentes en algunos trabajos, como pueden ser los diagramas de casos de uso de UML y sus relaciones de inclusión y extensión, por lo que puede resultar adecuado plantear cuál debe ser su tratamiento en un proceso de generación de pruebas funcionales del sistema a partir de requisitos funcionales

2. Objetivos

La identificación y definición de los objetivos de un trabajo de tesis es una tarea importante. Por este motivo, una vez presentado el dominio del problema en el que se enmarca este trabajo (capítulo I) y descritos un conjunto de trabajos relevantes de dicho dominio (capítulo II) se procede a definir los objetivos de este trabajo de tesis. Dichos objetivos se enumeran a continuación.

1. Estudiar la formalización de los artefactos utilizados en el proceso de generación de pruebas del sistema y, en concreto, de los requisitos funcionales y de los casos de pruebas funcionales.

2. Ampliar el uso de estándares que faciliten la interacción del proceso de generación de pruebas en un proceso de ingeniería guiada por modelos y la aplicación de herramientas existentes.

3. Especificar un proceso de generación de pruebas funcionales del sistema que utilice artefactos formalizados.

4. Definir la automatización de la generación de pruebas funcionales del sistema de manera semiautomática o automática.

5. Validar el proceso definido mediante la aplicación a un ejemplo concreto.

Como se puede apreciar, estos objetivos se han definidos para completar los aspectos contemplados en el planteamiento del problema. Un elemento a destacar es la elección de la ingeniería guiada por modelos como la base metodológica sobre la cual desarrollar el trabajo de generación de casos de pruebas funcionales. En posteriores secciones de este trabajo se justificará el por qué se considera que el paradigma de ingeniería guiada por modelos puede ser una herramienta válida para satisfacer los objetivos de formalización, estandarización y automatización planteados aquí.

En la siguiente sección se describen las influencias que servirán de contexto para construir una solución que satisfaga estos objetivos.

3. Influencias

Este trabajo de tesis se apoya en trabajos existentes sobre requisitos, y sobre la ingeniería guiada por modelos. En esta sección se describen cuáles han sido las propuestas más influyentes en los resultados de este trabajo de tesis que se presentan a lo largo de los capítulos posteriores.

3.1. Navigational Development Technique (NDT)

Como se ha visto en el capítulo anterior, algunas de las propuestas de generación de casos de pruebas funcionales del sistema hacen referencias a trabajos existentes sobre la definición de requisitos funcionales, por ejemplo [Labiche et-al, 2002], mientras que otras definen sus propias estructuras de requisitos funcionales, por ejemplo [Heumann, 2002] [Naresh, 2002].

Ya se ha mencionado previamente que la definición de los requisitos funcionales es un aspecto vital para el proceso de generación de pruebas funcionales del sistema, ya que constituyen la información de entrada del proceso de transformación y, tanto su estructura como su contenido, van a condicionar dicho proceso. Más aún, el uso de una propuesta de definición de requisitos funcionales ya existente podría ayudar a reducir el esfuerzo de introducir los procesos de generación de pruebas funcionales del sistema en un proceso más amplio de desarrollo de software. Por este motivo, surge la necesidad de definir con precisión el modelo de requisitos a utilizar en el proceso de generación de prueba del sistema, ya que de ello depende el proceso a aplicar.

La importancia de la definición y el contenido de los requisitos funcionales justifican la elección de una propuesta ya existente, en detrimento de definir una estructura ad-hoc. Para ello, es posible plantear una serie de criterios deseables para que una propuesta de requisitos pueda ser tomada como base de un proceso de generación de pruebas del sistema. Los criterios propuestos son: que los artefactos para la descripción de los requisitos funcionales sean análogos a los requisitos funcionales de los trabajos existentes analizados en el capítulo dos, que exista amplia documentación accesible de la propuesta, que la propuesta esté madura y haya sido aplicada con éxito a proyectos reales y que la propuesta esté en la actualidad en aplicación y que su autor o autores sigan trabajando en ella y no hayan cerrado dicha línea de investigación.

En el seno del grupo de investigación en el que se ha gestado este trabajo de tesis, ya se había realizado una propuesta de definición de requisitos. Al evaluar los criterios anteriores sobre dicha propuesta se vio que cubría las premisas, por lo que fue adoptada como punto de partida para el proceso de generación de pruebas funcionales del sistema. Dicha propuesta, llamada NDT (siglas de Navigational Development Techniques) [Escalona, 2004], [Escalona et-al, 2008], comienza a gestarse en el año 2002, a partir de trabajos anteriores sobre requisitos funcionales que dieron origen a una herramienta llamada REM [Durán et-al, 2000], y cristaliza en el año 2004 como un trabajo de tesis presentado en el seno de la Universidad de Sevilla. Dicha propuesta estaba centrada en potenciar el aspecto navegacional en sistemas web e hipermedia, sin embargo incorporaba un proceso y un conjunto de artefactos completos para abordar los requisitos de un sistema.

Actualmente NDT ha evolucionado hasta convertirse en una metodología web guiada por modelos que ofrece soporte completo para todo el ciclo de desarrollo, pruebas, aseguramiento de calidad y gestión del proyecto. Este soporte se muestra en la figura 3.1 [Escalona et-al, 2008].

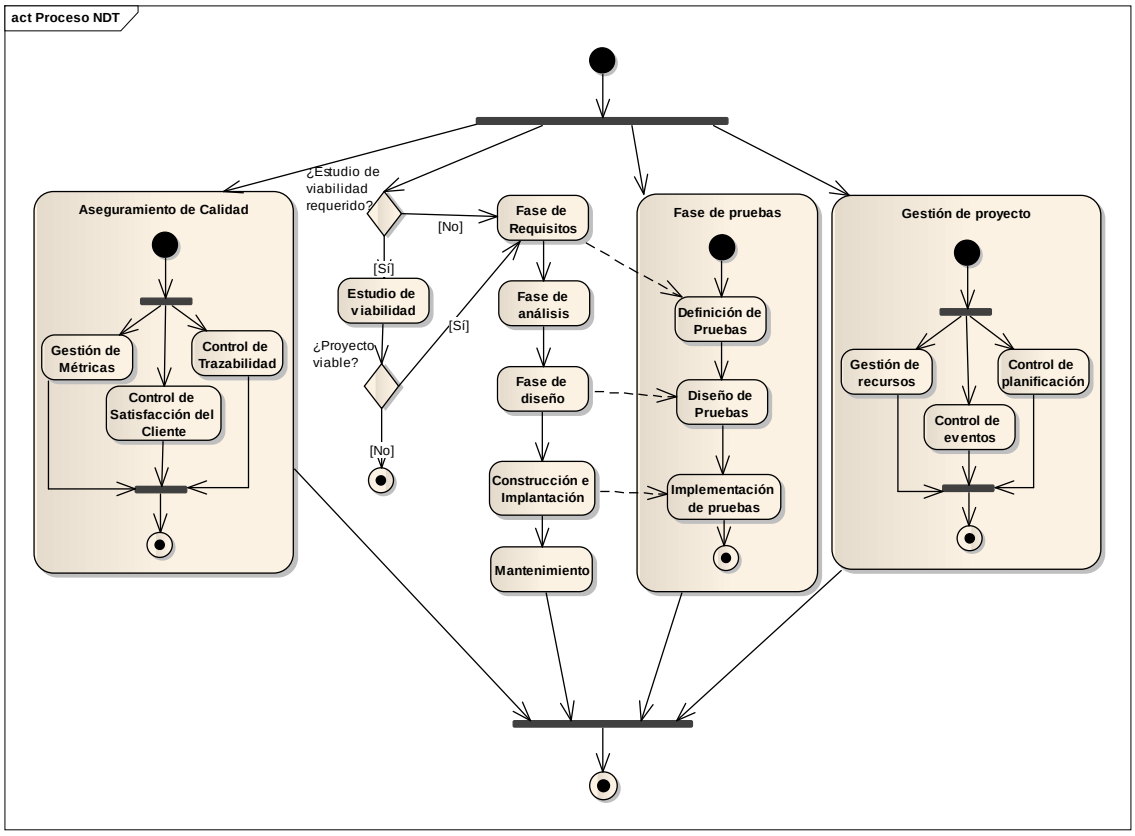


Figura 3.1. El proceso de desarrollo de NDT.

Los metamodelos definidos por NDT se formalizan mediante perfiles de UML, con el aporte de OCL [OMG-OCL, 2010] para la definición de las restricciones sobre los conceptos de los metamodelos, y el soporte de la herramienta Sparx Enterprise Architect como herramienta de modelado UML de referencia de la metodología. Las transformaciones se formalizan mediante QVT.

Todo el proceso de NDT mostrado en la figura 3.1 está completamente soportado por un conjunto de herramientas que se apoyan en Enterprise Architect, agrupadas bajo el nombre de NDT-Suite [NDT-Suite, 2011]. Estas herramientas se han desarrollado implementando en lenguaje Java las transformaciones formalizadas en QVT.

Por último, puede ser interesante mencionar que NDT se ha utilizado con éxito en varios proyectos de desarrollo. En concreto, en la referencia [Escalona et-al, 2008], se describen varios proyectos desarrollados en el seno de la Consejería de Cultura perteneciente a la Junta de Andalucía, un proyecto para la gestión de la empresa de agua de Sevilla, y en proyectos relacionados con el área de la salud.

3.2. Ingeniería dirigida por modelos (MDE)

La principal idea de la ingeniería guiada por modelos (llamada MDE por sus siglas en inglés) es utilizar un conjunto de modelos para ir disminuyendo el nivel de abstracción [Fondement et-al, 2004]. Así, en las primeras etapas del desarrollo se elaboran modelos con un alto nivel de abstracción y, a medida que el desarrollo avanza, se van desarrollando nuevos modelos (a partir de los anteriores) con un nivel de detalle más alto y más cercanos a la implementación final del sistema. En el ámbito de la ingeniería guiada por modelos, el modelo representa un aspecto relevante del sistema, por ejemplo la funcionalidad, la estructura de los datos, la navegación, las pruebas, etc. No existe un único nivel de abstracción posible para estos modelos, sino que se puede definir el nivel de abstracción más necesario para una situación determinada.

Sin embargo, al trabajar con modelos de esta forma, surgen dos necesidades principales que hay que satisfacer [Fondement et-al, 2004]. En primer lugar, es necesario contar con un conjunto de elementos comunes o de bloques de construcción para que todos los modelos se desarrollen utilizando los mismos bloques de construcción. En segundo lugar, se plantea la necesidad de definir mecanismos que hagan posible la obtención de unos modelos a partir de otros modelos diferentes, de una manera sistemática y susceptible de automatización de forma parcial o total.

Para resolver la primera necesidad, surge el concepto de metamodelo. De manera resumida, un metamodelo está constituido por el conjunto de elementos con su semántica y restricciones asociadas, que permite definir los modelos. La misión de un metamodelo es definir las relaciones entre conceptos de un dominio de problema y definir de manera precisa la semántica asociada a dichos conceptos del dominio [Schmidt, 2006]. Así, un metamodelo engloba el conjunto de constructores que representan los conceptos del metamodelo, las asociaciones válidas entre los constructores, las restricciones existentes y la definición del significado [Giachetti et-al, 2008]. Otra terminología utilizada, como una metáfora, nombra a los metamodelos como sintaxis abstracta y a los modelos como sintaxis concreta.

Para resolver la segunda necesidad, que era la búsqueda de un mecanismo para derivar nuevos modelos a partir de modelos ya existentes, se usa el mecanismo de transformaciones. Una transformación entre dos modelos representa una relación entre dos sintaxis abstractas o metamodelos y se define mediante un conjunto de relaciones entre los elementos del correspondiente metamodelo [Thiry et-al, 2009]. Una transformación es una expresión que relaciona los elementos de un metamodelo de origen con los elementos de un metamodelo de destino, de tal manera que cuando la transformación es realizada proporcionando un modelo de entrada conforme al metamodelo de origen, la transformación

sabe qué hacer con dichos elementos y cómo utilizarlos para construir los elementos del modelo de salida conforme al metamodelo de destino. En la figura 3.2 se muestra cómo se combinan ambas necesidades para la generación de nuevos modelos a partir de modelos ya existentes.

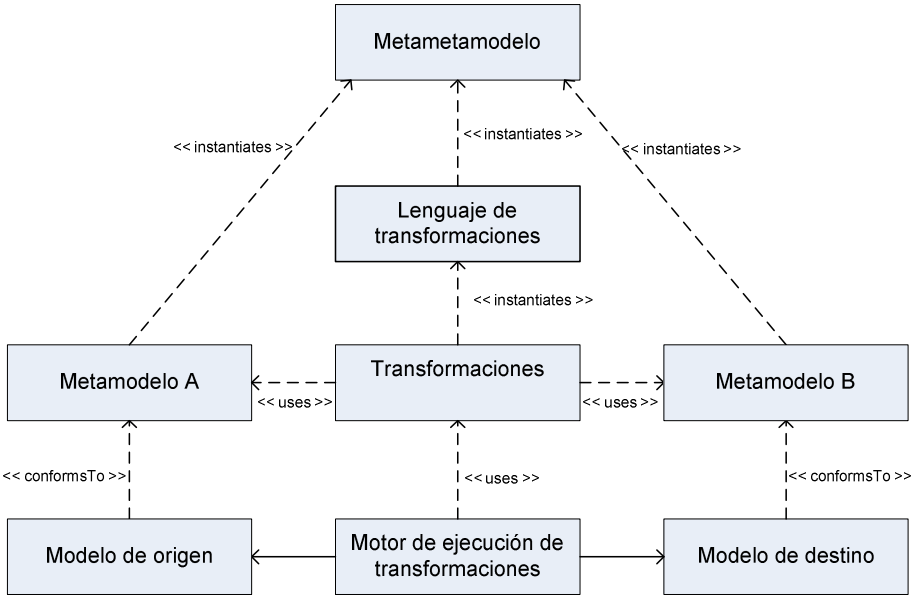


Figura 3.2. Visión del proceso de transformación desde una perspectiva MDE.

En la figura 3.2 se presenta el concepto de metametamodelo, que es la herramienta para construir metamodelos, los cuáles son la base para especificar transformaciones utilizando un lenguaje de transformaciones. A continuación, dichas transformaciones se aplican sobre un modelo concreto, conforme al metamodelo de entrada de la transformación para obtener como resultado otro modelo conforme al metamodelo de salida de la transformación. Esta figura se irá completando a lo largo de este capítulo a medida que se vean herramientas para definir metamodelos y lenguajes de transformaciones.

La ingeniería guiada por modelos aparece como una posible guía para plantear una solución a los objetivos vistos con anterioridad, al principio de este capítulo, pero no es ni mucho menos el único punto de vista posible. Sí es cierto que la elección de este punto de vista viene justificada por varias razones como el hecho, ya visto, de que la ingeniería guiada por modelos trabaja, como su nombre indica, con modelos, los cuáles son una herramienta posible para formalizar la información de los requisitos funcionales y de los casos de prueba funcionales del sistema, o como el hecho también visto de que la ingeniería guiada por modelos trabaja con transformaciones, las cuales son una herramienta posible para formalizar el proceso de transformación. Otras razones que también justifican la elección del desarrollo guiado por modelos son la reutilización que se hace de trabajos plenamente asentados en el

ámbito del software, como se verá en la sección dedicada a tecnología, y su uso exitoso en NDT, como se ha expuesto en la sección anterior.

4. Tecnologías

Para poder trabajar con ingeniería guiada por modelos (MDE), tal y como se ha descrito en la sección anterior) hace falta una serie de herramientas tecnológicas para representar modelos, metamodelos y transformaciones. Esta sección describe las tecnologías utilizadas en la construcción de la solución presentada en este trabajo de tesis.

4.1. Unified Modelling Language

El lenguaje unificado de modelado, ampliamente conocido por sus siglas en inglés UML [OMG-UML, 2011], es una herramienta para la definición de modelos utilizada principalmente en el campo de la ingeniería del software. Este lenguaje nació en 1997 con el objeto de estandarizar distintas técnicas de modelado y, desde entonces, ha cosechado un gran éxito y se ha convertido en uno de los estándares más importantes de la industria e investigación en el campo de la ingeniería del software. UML es desarrollado y mantenido por Object Management Group (más conocida por sus siglas OMG), la cual además, se encarga también de mantener el metamodelo sobre el cual se ha construido (llamado MOF) y otros trabajos relacionados

A la hora de representar un metamodelo, se pueden buscar diferentes notaciones o técnicas que permitan definir los conceptos, sus relaciones y las restricciones de los mismos. Actualmente una de las técnicas más utilizadas es el uso de UML. Una característica muy importante de UML y que será aplicada en los siguientes capítulos, es la incorporación de mecanismos de extensión en el propio lenguaje. Este mecanismo de extensión permite dotar a UML de elementos con una semántica más precisa o más especializada en un campo concreto. Por ejemplo, actualmente el desarrollo de sistemas web tiene su propio conjunto de artefactos ampliamente reconocidos (páginas, componentes del cliente, etc.). Gracias a las posibilidades de expansión de UML se puede trabajar con un conjunto de elementos base que aporten mucha más semántica que una clase, es decir, se puede personalizar el componente clase de UML para generar una paleta de elementos mucho más amplia y que proporcione una semántica más precisa.

Estos mecanismos de extensión se resumen en dos conjuntos de artefactos: estereotipos y perfiles. El primer mecanismo, los estereotipos, es el más sencillo de aplicar. Un

estereotipo es una precisión semántica realizada sobre un elemento. Por ejemplo, el concepto de clase en UML es un concepto muy amplio, retomando el contexto de desarrollo de sistemas web mencionado en los párrafos anteriores, se podrían modelar como clases UML los Servlets, páginas JSP, filtros, controladores centrales, JavaBeans, y otros elementos habituales en este tipo de sistemas. Sin embargo, tal y como salta a la vista, si bien estos elementos tienen aspectos comunes también tienen una semántica, comportamiento y características diferentes. Por lo tanto, el mecanismo de estereotipos va a permitir definir nuevos elementos a partir de la clase de UML. Cada uno de esos elementos será una clase pero con un añadido semántico (será una clase estereotipada, según la terminología de UML).

El segundo mecanismo, perfiles, es más complejo que los estereotipos pero también es más potente. Un perfil está compuesto de un conjunto de estereotipos, valores etiquetados y un conjunto de restricciones sobre estos elementos.

En el ámbito de este proyecto, UML va a ser utilizado para definir los metamodelos de los artefactos de requisitos y de pruebas. Además, se presentarán los perfiles de UML necesarios para elaborar sintaxis concretas para dichos modelos mediante extensiones formales del propio UML.

La definición de un metamodelo como diagrama de clases estima que cada metaclase represente un concepto de aquellos que se quiere definir. Además, la posibilidad de definir invariantes y restricciones mediante OCL que permite UML, resulta de gran utilidad a la hora de establecer las restricciones de los conceptos y sus relaciones. Por ejemplo, UML es un lenguaje definido de manera formal en sí mismo mediante un conjunto de metamodelos. Cada uno de ellos permite definir instancias que a su vez, permitirán construir diagramas de clases, de actividades, de estados, etc.

4.2. El perfil de pruebas de UML

Como ya se ha visto, los perfiles de UML son uno de los mecanismos estándares que proporciona el propio UML para definir extensiones. El perfil de pruebas de UML (llamado a partir de aquí por sus siglas en inglés: UMLTP) [OMG-UMLTP, 2005], aparecido como borrador en el año 2002, proporciona elementos con una semántica específica para el proceso de pruebas, tanto a nivel estructural como a nivel de comportamiento.

Este perfil está basado en UML 2.0 y sus elementos se organizan en cuatro grupos lógicos. Estos grupos son: comportamiento de pruebas, arquitectura de pruebas, datos de prueba y tiempo de prueba. El grupo lógico de comportamiento de prueba define los elementos

utilizados para la definición de artefactos relevantes a aspectos dinámicos de los procedimientos de prueba. Algunos ejemplos de estos artefactos son: Caso de Prueba, Objetivo de Prueba, Observación, etc. El grupo lógico de arquitectura de prueba define los elementos utilizados para la definición de artefactos relacionados con la estructura y configuración de las pruebas. Algunos ejemplos de estos artefactos son: Contexto de Prueba, Configuración de Prueba, Árbitro, etc. El grupo lógico de datos de prueba define los elementos utilizados para la definición de artefactos concernientes a los valores de prueba utilizados en las pruebas. Algunos ejemplos de estos artefactos son: Carácter Comodín, Pool de Datos o Partición de Datos. Finalmente, el grupo lógico de tiempo de prueba define los elementos utilizados para la definición de artefactos concernientes a la cuantificación de valores de tiempo en las pruebas. Algunos ejemplos de estos artefactos son: Zona Horaria o Temporizador.

UMLTP incorpora conceptos ya existentes en el mundo de las pruebas y les dota de una sintaxis concreta basada en UML. Este trabajo de tesis va a incorporar algunos elementos de UMLTP como se verá más adelante.

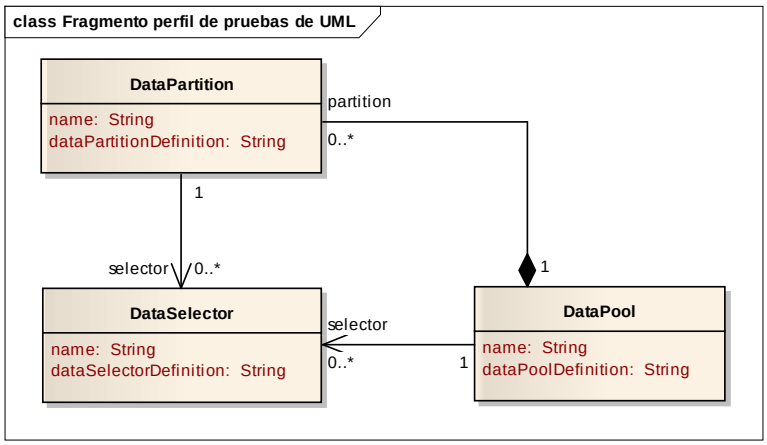


Figura 3.3. Fragmento del perfil de pruebas de UML.

A modo de ejemplo, en la figura 3.3 se muestra el diagrama de clases que describe los elementos principales del grupo lógico de datos de prueba del perfil de pruebas de UML. Se puede ver cómo los valores de prueba se agrupan en particiones, con la misma semántica definida en el método de Categoría-Partición, y, además, se añaden dos elementos adicionales, *DataSelector* y *DataPool*, para gestionar los valores de prueba durante la ejecución de casos de prueba.

4.3. Query-View Transformations

Query-View Transformation (llamado QVT) es un lenguaje para definir transformaciones de modelo a modelo y que trabaja con UML. QVT también es gestionado por el Object Management Group (OMG).

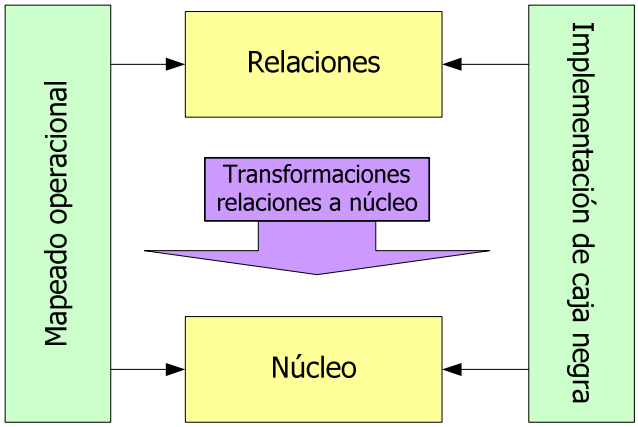


Figura 3.4. Descripción general de QVT.

QVT define tres lenguajes específicos de dominio (figura 3.4): un lenguaje relacional, un lenguaje operacional y un núcleo o core. El lenguaje relacional y el core son lenguajes declarativos (por ejemplo análogo a SQL, el cual es un lenguaje declarativo de amplio uso) pero a dos niveles distintos de abstracción. El lenguaje relacional está basado en el concepto de *pattern matching,* según el cual, se define un patrón o conjunto de patrones que el entorno de ejecución intentará hacer coincidir con elementos del modelo. El lenguaje operacional es una extensión y complemento de los lenguajes relacional y core, para incorporar construcciones imperativas como condicionales, bucles, etc.

QVT, además de los lenguajes y mecanismos mencionados, integra OCL a partir de su versión 2.0 [OMG-OCL, 2010], el cual es otro lenguaje declarativo para la definición de restricciones, por ejemplo en los diagramas definidos mediante UML.

QVT también define el concepto de BlackBox o caja negra, por el cual se pueden incorporar a las transformaciones funciones auxiliares desarrolladas en otros lenguajes (como por ejemplo XQuery o incluso Java). Esto permite incorporar conceptos ya existentes, como cálculo de impuestos, en las transformaciones.

5. Estructura de la solución

Una vez vistas las influencias y herramientas, esta sección presenta una propuesta de solución para el proceso sistemático de generación de pruebas funcionales del sistema a partir de requisitos funcionales orientada a cubrir los objetivos definidos.

El punto de partida para desarrollar un proceso de generación de pruebas funcionales a partir de requisitos funcionales son los trabajos ya existentes analizados en el capítulo II. En el estudio comparativo se han identificado dos técnicas utilizadas por las distintas propuestas para obtener pruebas funcionales del sistema a partir de requisitos funcionales. Estas técnicas son los escenarios de prueba y los valores de prueba. Un escenario se definía como el conjunto de pasos a realizar durante una prueba y los valores de prueba se definían como la información a suministrar durante una prueba.

El proceso de generación de pruebas funcionales del sistema a partir de los requisitos funcionales presentado en esta tesis constará de dos actividades principales: obtener los escenarios de prueba y obtener los valores de prueba. Además, se va a introducir un artefacto adicional, los *casos de prueba funcionales del sistema* cuyo objetivo es expresar de una manera conjunta, en un mismo artefacto, la información recogida en un escenario y en un conjunto de valores de prueba, siguiendo la estela de alguna de las propuestas vistas en el capítulo II, como [Ruder, 2004]. Así, el proceso de generación de pruebas funcionales del sistema tendrá una tercera actividad principal cuya misión será obtener los casos de prueba a partir de los escenarios de prueba y valores de prueba. Este proceso puede verse en la figura 3.5.

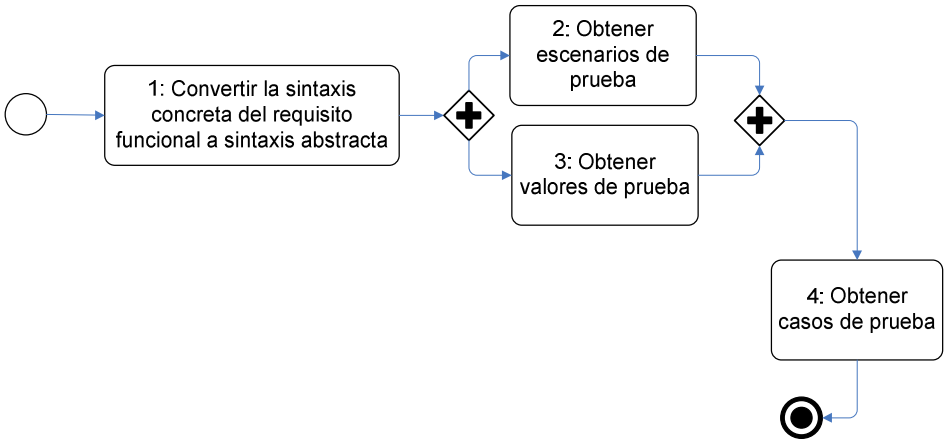


Figura 3.5. Descripción del proceso de generación de pruebas funcionales del sistema.

Como ya se ha justificado, no es conveniente ligarse a una representación concreta de los requisitos funcionales. Por ello, se incluye en el proceso una primera actividad que realiza una conversión de la sintaxis concreta de los requisitos a una sintaxis abstracta, la cuál será el punto de entrada de las siguientes actividades.

El proceso de la figura 3.5 es el proceso que se va a formalizar y definir en los próximos capítulos de este trabajo de tesis utilizando herramientas de ingeniería guiada por modelos.

A continuación, se va a plantear cómo definir este proceso desde una perspectiva de ingeniería guiada por modelos. Para ello, es necesario definir el proceso de la figura 3.5 en base a los artefactos descritos para ingeniería guiada por modelos, por lo que será descrito utilizando metamodelos y transformaciones (figura 3.6).

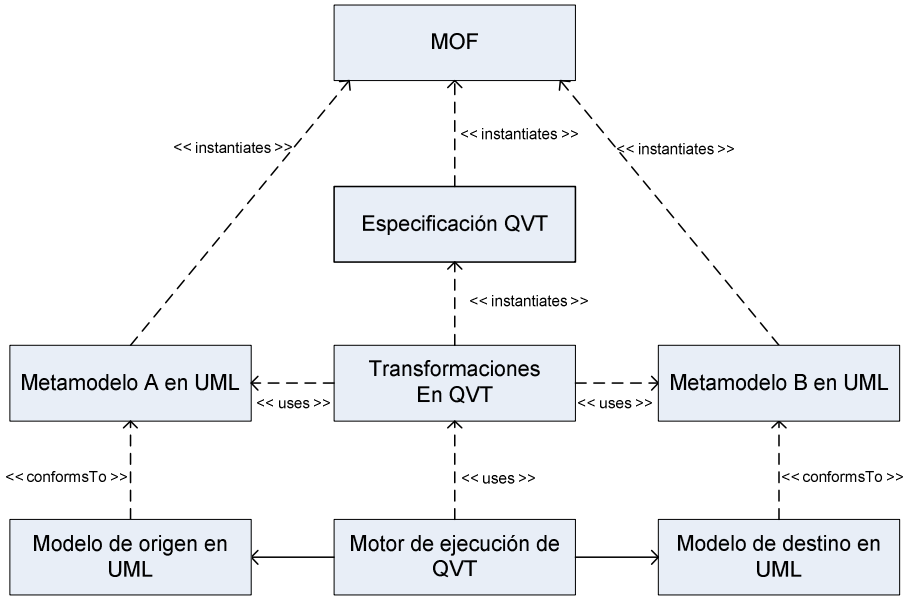


Figura 3.6. Relación entre los metamodelos y transformaciones utilizados.

Cada una de las actividades del proceso de generación de pruebas funcionales del sistema (figura 3.5) se realizará aplicando la propuesta de la figura 3.6. Así, para pasar de una sintaxis concreta a una sintaxis abstracta será necesario desarrollar los metamodelos de ambas sintaxis (utilizando UML en el contexto de este trabajo de tesis) y definir las transformaciones adecuadas (utilizando QVT en el contexto de este trabajo de tesis). Este mismo trabajo, definir metamodelos y transformaciones, será necesario para realizar la obtención de escenarios de prueba, valores de prueba y casos de prueba

La lista de metamodelos y transformaciones necesarios para poner en pie el proceso descrito en la figura 3.6, se enumeran a continuación, en la tabla 3.1.

El primer metamodelo es el metamodelo de requisitos funcionales. Este metamodelo definirá la información necesaria para definir un requisito funcional de manera que sea posible poder aplicar un proceso de generación de pruebas funcionales del sistema de manera sistemática y automatizable.

Tabla 3.1. Metamodelos y transformaciones.

Metamodelos	• Metamodelo de requisitos funcionales. • Metamodelo de escenarios de prueba. • Metamodelo de valores de prueba. • Metamodelo de casos de prueba.
Transformaciones	• Transformación de requisitos funcionales a escenarios de prueba. • Transformación de requisitos funcionales a valores de prueba. • Transformación de escenarios de prueba y valores de prueba a casos de prueba.

El segundo metamodelo, es el metamodelo de escenarios de prueba. Este metamodelo describirá los distintos escenarios de ejecución de cada requisito funcional, esto es, cada posible combinación de pasos.

El tercer metamodelo, es el metamodelo de valores de prueba. Este metamodelo describirá los artefactos utilizados por el Método de Categoría-Partición [Ostrand, 1988], que como se ha visto en el capitulo anterior es aplicado por otros conjunto de trabajos [Binder, 1999], para la generación de los requisitos funcionales.

El último metamodelo, es el metamodelo de casos de prueba. Un caso de prueba se define como la fusión de un escenario concreto con sus valores de prueba asociados. Como ya se ha visto, la ventaja de la definición del metamodelo de escenarios de prueba y del metamodelo de valores de prueba es que permite llevar al ámbito de la ingeniería guiada por modelos, y de sus herramientas de automatización, las propuestas vistas en el capítulo II que siguen estas estrategias. La ventaja principal de añadir un tercer metamodelo es que permite definir artefactos que combinen la información de ambas estrategias en una única prueba, obteniendo resultados semánticamente más completos. Lógicamente, la composición de escenarios y valores de prueba en un caso de prueba permite recoger más información que si consideramos ambos artefactos por separado. En concreto, combinar escenarios y valores de

prueba en un caso de prueba funcional del sistema permite definir información relevante sobre en qué paso del caso de prueba es relevante un valor de prueba.

Respecto a las transformaciones, la primera trasformación es la transformación de requisitos funcionales a escenarios de prueba. El objetivo de esta transformación es generar un modelo de escenarios de prueba conforme al metamodelo de escenarios de prueba, a partir de un modelo de requisitos funcionales conforme al metamodelo de requisitos funcionales. La segunda transformación es la transformación de requisitos funcionales a valores de prueba. El objetivo de esta transformación, al igual que la anterior, es generar un modelo de valores de prueba conforme al metamodelo de valores de prueba a partir de un modelo de requisitos funcionales conforme al metamodelo de requisitos funcionales. La tercera y última transformación combina la información de los modelos de escenarios y valores d prueba en un único modelo. El objetivo de la transformación de escenarios de prueba y valores de prueba a casos de prueba es obtener un modelo de casos de prueba conforme al metamodelo de casos de prueba a partir de un modelo de escenarios de prueba y un modelo de valores de prueba.

En la figura 3.5 se ha presentado una actividad para obtener la sintaxis abstracta del requisito funcional a partir de la sintaxis concreta. Dado que cada sintaxis concreta es distinta, se necesitaría un proceso específico para cada sintaxis concreta que se desee utilizar, lo que hace difícil proponer una solución general.

6. Conclusiones

En este capítulo se ha definido el planteamiento del problema a abordar a partir del estudio comparativo del capítulo anterior, lo que ha motivo el planteamiento de los objetivos a alcanzar en este trabajo de tesis. A partir e dichos objetivos, se han presentado las influencias y tecnologías a utilizar. Por último, se ha definido, de manera general, un proceso de generación de pruebas funcionales del sistema a partir de los requisitos funcionales que ha tomado como base los resultados de las propuestas existentes, que satisface los objetivos planteados, y que utiliza las influencias y tecnologías vistas.

En los siguientes capítulos se desarrollará el proceso descrito en las secciones anteriores mediante la definición de los artefactos contemplados en la descripción del proceso.

Capítulo IV. Metamodelo de Requisitos Funcionales

En el capítulo anterior se ha expuesto un planteamiento para abordar la generación de pruebas funcionales del sistema a partir de requisitos funcionales basado en la ingeniería guiada por modelos. Este capítulo presenta un metamodelo de requisitos funcionales, el cual define los elementos con los que se contará para la generación de pruebas funcionales del sistema, su estructura y relaciones.

La organización de este capítulo se describe a continuación. En la sección 1 se define el metamodelo de requisitos funcionales presentando y describiendo los conceptos que lo componen. En la sección 2 se presenta un ejemplo de uso de dicho metamodelo. Finalmente, en la sección 3 se exponen las conclusiones de este capítulo.

1. Definición del metamodelo de requisitos funcionales.

Esta sección define el metamodelo de requisitos funcionales cuyo objetivo es especificar formalmente aquellos conceptos que se utilizarán en el proceso de generación de pruebas funcionales del sistema más adelante en esta tesis. La figura 4.1 muestra este metamodelo de requisitos funcionales.

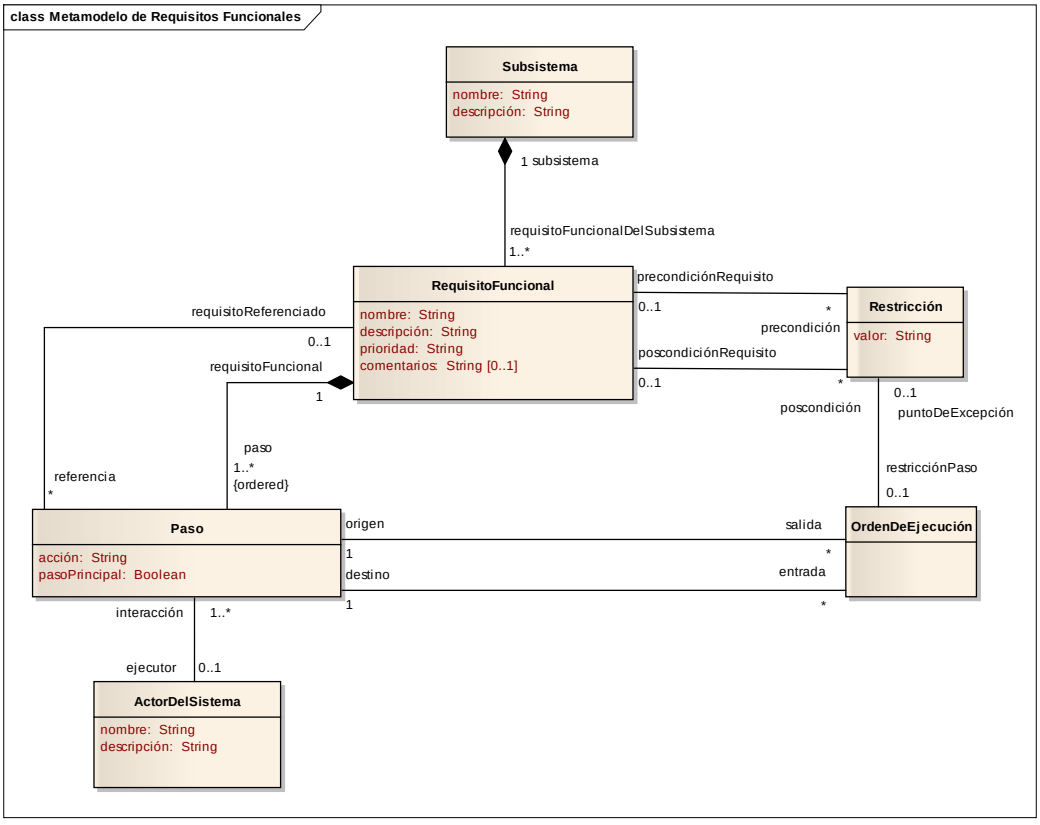


Figura 4.1. Metamodelo de requisitos funcionales.

El concepto más importante de este metamodelo es el concepto de requisito funcional (elemento *RequisitoFuncional* en la figura 4.1). En este ámbito de este metamodelo, un requisito funcional define un comportamiento que el sistema debe implementar y que, por tanto, es susceptible de ser probado mediante casos de prueba funcionales del sistema.

El resto de conceptos presentes en este metamodelo tienen como misión completar la información del elemento requisito funcional, por ejemplo, indicando en qué pasos se descompone el comportamiento del requisito funcional (elemento *Paso*), el orden de dichos pasos (elemento *OrdenDeEjecución*), qué actor realiza el paso si no es realizado por el propio sistema (elemento *ActorDelsistema*), etc. La tabla 4.1 resume todos los conceptos del metamodelo de requisitos funcionales (figura 4.1) ordenados alfabéticamente.

Tabla 4.1. Elementos del metamodelo de requisitos funcionales.

Elemento	Descripción
ActorDelSistema	Descripción de la información referente a las entidades externas que interactúan con el sistema.
OrdenDeEjecución	Relación entre pasos que indica en qué orden y bajo qué restricciones se realizan los mismos
Paso	Fragmento del comportamiento a realizar en un requisito funcional que define una tarea o acción realizada por uno de los participantes en el requisito funcional.
RequisitoFuncional	Descripción de una interacción entre un grupo de actores del sistema y el propio sistema para lograr un resultado.
Restricción	Predicado booleano usado como precondición, poscondición o punto de extensión.
Subsistema	Agrupación de requisitos funcionales y actores según un criterio definido por el usuario.

Como se puede ver en la figura 4.1, los requisitos funcionales están compuestos de pasos, los cuáles describen cada una de las acciones a realizar para satisfacer el requisito funcional. Un paso puede tener un ejecutor (modelado con el elemento *ActorDelSistema*), que es un actor externo al sistema y que interactúa con él durante la realización de un requisito funcional. Si un paso no tiene ejecutor, se asume que es realizado por el propio sistema.

Para indicar la prelación y alternativas a la hora de la realización de los pasos, se utiliza el concepto de orden de ejecución (modelado con el elemento *OrdenDeEjecución).* Este elemento permite conocer qué paso debe realizarse a la finalización de un paso concreto, asumiendo que, si no existe ninguno, el requisito funcional finaliza.

En el metamodelo de la figura 4.1 se contempla la posibilidad de que, a la finalización de un paso, existan varios pasos que puedan realizarse. Para ello, se utilizarán restricciones (modeladas con el elemento *Restricción*), que indicarán bajo qué condiciones pueden realizarse cada uno de los múltiples pasos.

Además, el metamodelo presentado permite definir relaciones entre distintos requisitos funcionales utilizando los pasos. Así un paso de un requisito funcional puede indicar la realización de otro requisito funcional.

Por último, el metamodelo define la clasificación y agrupación de requisitos funcionales en subsistema.

En la siguiente sección, se describen en detalle los elementos de la figura 4.1 y de la tabla 4.1, siguiendo el estilo de documentación utilizado por el Object Modeling Group (abreviadamente OMG) para la documentación de UML y especificaciones relacionadas. Este estilo se describe a continuación. En primer lugar se ofrece una breve descripción del elemento. A continuación, se indican sus generalizaciones, si existen, y las restricciones asociadas a dichas generalizaciones. A continuación se describen sus asociaciones y sus atributos. Posteriormente, se describen las restricciones a los valores de los atributos y asociaciones si la hubiera. Por último, se describe con más detalle la semántica del elemento si es preciso.

1.1. ActorDelSistema

Este elemento representa a un participante externo al sistema involucrado en el comportamiento de un requisito funcional.

Asociaciones

- interacción: Paso [1..*]
  Conjunto de pasos en los que participa el actor del sistema.

Atributos

- nombre: String[1]
  Nombre del actor del sistema.
- descripción: String[1]
  Descripción del actor del sistema.

Semántica

Un *ActorDelSistema* participa en el comportamiento descrito por un requisito funcional. Los momentos concretos en los que se requiere esta participación son aquellos pasos del requisito funcional referenciados por la asociación *interacción*.

1.2. OrdendeEjecución

Este elemento determina el orden en que se realizan los pasos de un requisito funcional.

Asociaciones

- origen: Paso[1]
  Paso de origen del orden de ejecución.
- destino: Paso [1]
  Paso de destino del orden de ejecución.
- puntoDeExcepción: Restricción[0..1]
  Restricción que controla la realización del orden de ejecución.

Restricciones

[1] self.origen **=** self.destino **implies** self.puntoDeExtensión→size() > 0

Semántica

Un elemento *OrdenDeEjecución* representa la transición del fragmento de comportamiento definido en el paso origen al fragmento del comportamiento definido en el paso destino.

La asociación punto de excepción define una condición que debe cumplirse para poder pasar del paso origen al paso de destino. Así, en algunas ocasiones, una transición se realiza de manera automática, realizándose el paso de destino al concluir el paso de origen cuando no se indica ningún punto de excepción. En otras ocasiones, cuando sí se indica un punto de excepción, el inicio del paso de destino está condicionado a que la condición definida en el punto de excepción sea cierta.

Por ejemplo, en un requisito que describa cómo dar de alta un nuevo elemento, un posible paso indicará que un actor externo debe introducir la información del elemento a añadir. A continuación de este paso puede que sea necesario especificar otro paso que indique qué comportamiento se espera si la información introducida es incorrecta. Para indicar que después del paso de introducir la información viene el paso de tratar la información si es incorrecta, se

define un elemento *OrdenDeEjecución* entre ambos pasos. Como en este ejemplo el segundo paso sólo es necesario si la información introducida es incorrecta, el orden de ejecución entre ambos pasos tendrá una restricción para definir dicha condición. Este mismo ejemplo, se define con más detalle en la sección 2 de este capítulo de tesis.

La restricción [1] especifica que es posible que un elemento orden de ejecución comience y termine en el mismo paso, lo cual permite definir bucles en el comportamiento de un requisito funcional. En este caso, dicho orden de ejecución debe llevar asociada una restricción para indicar la restricción que controla la realización del comportamiento repetitivo.

1.3. Paso

Este elemento define un fragmento del comportamiento de un requisito funcional.

Asociaciones

- ejecutor: ActorDelSistema[0..1]
  Actor del sistema que realiza el paso.
- requisitoReferenciado: RequisitoFuncional[0..1]
  Requisito funcional realizado en el paso.
- requisitoFuncional: RequisitoFuncional[1]
  Requisito funcional al que pertenece el paso.
- salida: OrdenDeEjecución[*]
  Órdenes de ejecución que salen de un paso.
- entrada: OrdenDeEjecución[*]
  Órdenes de ejecución que llegan a un paso.

Atributos

- acción: String[1]
  Acción realizada en el paso principal.
- pasoPrincipal: Boolean[1]
  Índica si el paso pertenece a la secuencia principal del requisito funcional.

Restricciones

[1] **not***(* self. requisitoReferenciado → isEmpty())

implies self. requisitoReferenciado **<>** self.requisitoFuncional

[2] self.salida→size() > 1 **implies** self.salida→**select**(oe: OrdenDeEjecucion |
oe.puntodeExcepción.isEmpty()=**false**)→size() >= (self.salida→size()-1)

[3] self.salida→size() = 0 **implies** self.entradas→size() > 0

Semántica

El comportamiento de un requisito funcional se define mediante elementos *Paso.* Todo paso debe ser realizado por un actor o por el sistema. Por ello, si la multiplicidad de la relación *ejecutor* es 0, el paso se considera realizado por el sistema.

Se definen dos tipos posibles de pasos, en función del valor del atributo *PasoPrincipal*. Si el valor de este atributo es cierto, el paso pertenece a la secuencia principal del requisito funcional. Si el valor del atributo es falso, este paso pertenece a una secuencia alternativa del requisito funcional. La secuencia principal es el conjunto de pasos necesario para alcanzar el resultado de un requisito funcional con éxito y satisfacer sus poscondiciones. Lo pasos que no están en la secuencia principal describen el comportamiento a realizar al surgir errores o bien comportamiento alternativo u opcional al comportamiento de la secuencia principal.

Este metamodelo permite la posibilidad de referenciar un requisito funcional desde otro requisito funcional con una semántica similar a, por ejemplo, las asociaciones de inclusión y extensión para casos de uso de UML. En concreto, un paso define la asociación *requisitoReferenciado* para indicar que el paso permite la realización del comportamiento definido en otro requisito funcional. Relacionado con esto, en la restricción [1] se indica que el requisito funcional de la relación *requisitoReferenciado* no puede ser el mismo requisito funcional al que pertenece el paso, es decir, que no tiene sentido que un paso referencie al mismo requisito funcional al que pertenece.

Como se ha mencionado al principio de este capítulo, un paso puede tener más de un paso de salida, lo cual significaría que, a la conclusión de dicho paso, habría varios pasos candidatos para continuar el comportamiento del requisito funcional. Para poder determinar qué paso concreto se realizará es necesario que los órdenes de ejecución tengan condiciones (modelados mediante el elemento *Restricción).* Esto se expresa con la restricción [2]. Esta restricción impone que, si existe más de un orden de ejecución de salida, todos deben incluir restricciones para poder determinar cuál es el orden de ejecución a considerar.

Además, un paso puede no tener órdenes de ejecución de salida, porque sea un paso que termina con el comportamiento del requisito funcional, o bien puede no tener órdenes de ejecución de entrada porque sea un paso inicial. Sin embargo, tal y como indica la restricción

[3], no tiene sentido considerar pasos que no tengan a la misma vez ni órdenes de ejecución de salida ni órdenes de ejecución de entrada.

1.4. RequisitoFuncional

El elemento requisito funcional define un comportamiento del sistema.

Asociaciones

- paso: Paso[1..*]
  Conjunto ordenado de pasos del requisito funcional.
- subsistema: Subsistema[1]
  Subsistema al que pertenece el requisito funcional.
- referencia: Paso[*]
  Paso de otro requisito funcional que incluye el comportamiento de este requisito funcional.
- precondición: Restricción[*]
  Condiciones a evaluar antes de la realización del requisito funcional.
- poscondición: Restricción[*]
  Condiciones ciertas después de la realización de la secuencia principal del requisito funcional.

Atributos

- nombre: String[1]
  Nombre del requisito funcional.
- descripción: String[1]
  Descripción del requisito funcional.
- prioridad: String[1]
  Prioridad del requisito funcional.
- comentarios: String[0..1]
  Comentarios adicionales del requisito funcional.

Restricción

[1] self.paso→includes(self.referencia) = **false**

Semántica

Como ya se ha mencionado en la introducción de este capítulo, un requisito funcional define un comportamiento del sistema susceptible de ser probado. Este comportamiento se define mediante los pasos asociados con la asociación *paso* y puede ser parte del comportamiento de otro requisito funcional con la asociación *referencia*. Además, un requisito funcional puede definir las condiciones que deben cumplirse para poder realizar su comportamiento con la asociación *precondición* y las condiciones que se evalúan a cierto después del comportamiento del requisito funcional (en concreto de su escenario principal) con la asociación *poscondición*.

El atributo *prioridad* de un requisito funcional permitirá, posteriormente, definir un criterio para organizar los casos de prueba que verifiquen los requisitos funcionales. Es decir, una vez generados los artefactos de prueba (como se verá en los dos próximos capítulos) sería posible ordenarlos en base a su prioridad. En el ámbito de este trabajo se utiliza el concepto de prioridad tal y como se define en la metodología NDT presentada en el capítulo anterior. En concreto, esta metodología propone que sea la propia organización participante la que desarrolle el conjunto de criterios a tener en cuenta.

La restricción [1] indica que un paso que referencie al requisito funcional no puede pertenecer al mismo requisito funcional. Esto impide que un paso de un requisito funcional incluya al propio requisito funcional.

Por último, como se ha mencionado en la definición de atributos, las poscondiciones de este metamodelo sólo se aplican a la finalización con éxito del requisito funcional, es decir, a la realización de todos los pasos de la secuencia principal. La definición de otras poscondiones, como poscondiciones globales que son siempre ciertas a la finalización del requisito funcional, independientemente de los pasos ejecutados, no se contempla en este metamodelo.

1.5. Restricción

Este elemento define un predicado booleano.

Asociaciones

- precondiciónRequisito: RequisitoFuncional[0..1]
  Requisito funcional en el que la restricción funciona como precondición.
- poscondiciónRequisito: RequisitoFuncional[0..1]
  Requisito funcional en el que la restricción funciona como poscondición.

- restricciónPaso: OrdenDeEjecución[0..1]
  Orden de ejecución en la que la restricción condiciona la realización de un paso.

Atributos

- valor: String[1]
  Definición del predicado booleano.

Restricciones

[1] Toda restricción debe estar asociada a un requisito funcional o a un orden de ejecución.
[2] Toda restricción puede tener solo una asociación.

Semántica

Este elemento permite definir los predicados booleanos que deben ser ciertos en las precondiciones y poscondiciones o bien los predicados booleanos que deben ser ciertos para realizar un punto de extensión de un paso excepcional.

La restricción [1] especifica que todo elemento *Restricción* debe tener una relación que lo asocie bien a un elemento *RequisitoFuncional* o bien a un elemento *OrdenDeEjecución*. La restricción [2] impone que un elemento *Restricción* solo puede tomar el rol e precondición, el rol de poscondición o el rol de restricción de paso pero nunca puede asumir más de un rol a la vez.

1.6. Subsistema

Este elemento representa un conjunto de requisitos funcionales.

Asociaciones

- requisitoFuncionalDelSubsistema: RequisitoFuncional[1..*]
  Requisitos funcionales del subsistema.

Atributos

- nombre: String[1]
  Nombre del subsistema.

- descripción: String[1]
  Descripción del subsistema.

Semántica

Este elemento permite agrupar los requisitos funcionales en conjuntos llamados subsistemas.

2. Ejemplos

En esta sección se presenta un ejemplo de instancia del metamodelo de requisitos funcionales.

Tabla 4.2. Ejemplo de requisito funcional en prosa estructurada.

Requisito funcional:
Alta de nuevo elemento.

Descripción:
Este requisito funcional describe el comportamiento del sistema cuando un usuario desea dar de alta un nuevo elemento.

Precondiciones:
No.

Poscondiciones:
El elemento está dado de alta en el sistema.

Secuencia principal

1. El usuario solicita dar de alta un nuevo elemento en el sistema.
2. El sistema solicita la información del elemento.
3. El usuario introduce la información del elemento.
4. El sistema da de alta al elemento.

Secuencia alternativa

1.1. Si se ha superado el límite de elementos del sistema, entonces se muestra un mensaje de error y el resultado es finalizar la ejecución.

3.1. Si la información es incorrecta, entonces se muestra un mensaje de error y el resultado es ir al paso 2.

3.2. Si el elemento ya está en el sistema, el sistema pregunta si se desea actualizar el elemento con la nueva información.

3.2.1. Si la respuesta es afirmativa, entonces el requisito funcional continúa.

3.2.2. En caso contrario, el resultado es ir al paso 2.

La tabla 4.2 define un requisito genérico para el alta de nuevo elemento utilizando una sintaxis concreta de texto tabulado muy similar a la sintaxis utilizada por NDT y la mayoría de propuestas vistas en el capítulo 2. A modo de ejemplo, la figura 4.2 modela un fragmento de dicho requisito funcional utilizando los elementos definidos en el metamodelo de requisitos funcionales.

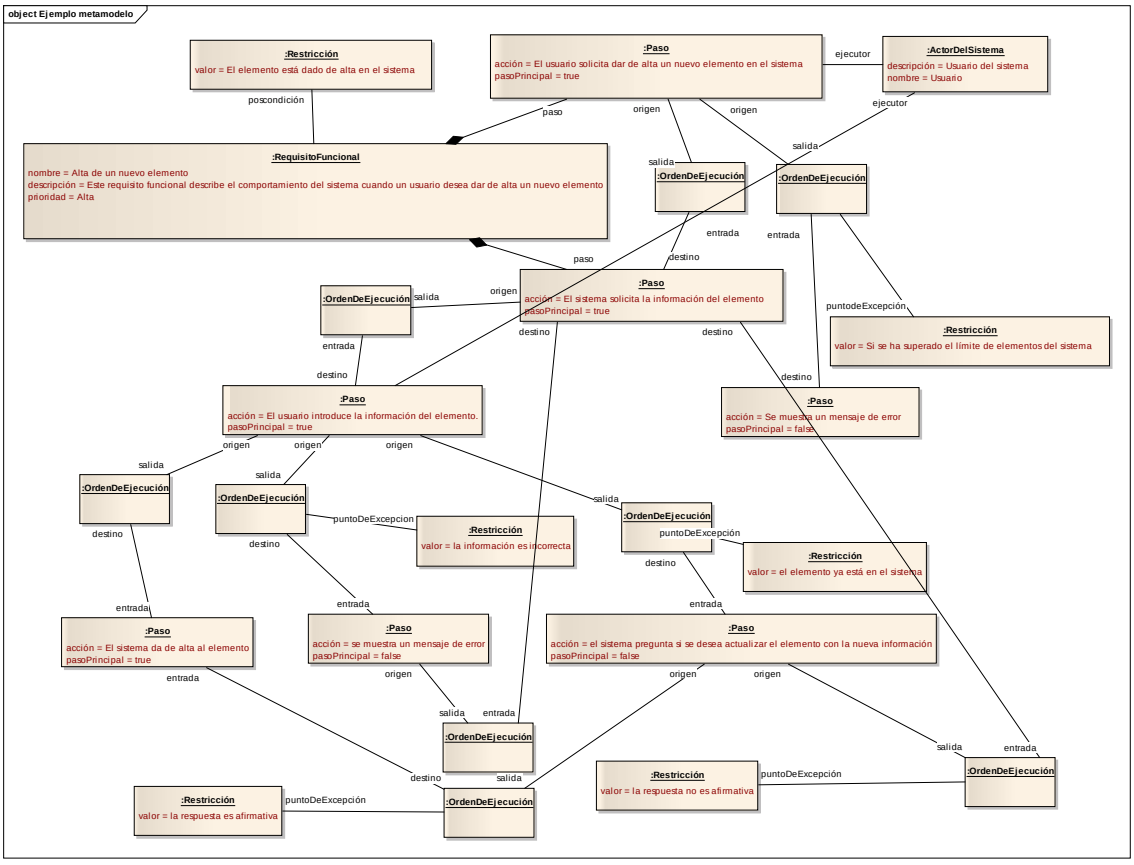


Figura 4.2. Ejemplo de requisito funcional y paso principal.

El fragmento de la figura 4.2 tiene un primer paso principal con la acción “El usuario solicita dar de alta un nuevo elemento”. Dicho paso es realizado por el actor de nombre “Usuario del sistema”. El segundo paso principal es un paso realizado por el sistema, el cual tiene asociado un paso excepcional (indicado por una orden de ejecución con una restricción) que, si se cumple el punto de extensión indicado en el orden de ejecución, muestra un mensaje

de error y finaliza la ejecución del requisito funcional (porque no tiene ningún orden de ejecución de salida). Si dicho punto de extensión (modelado con la restricción) no se cumple, como ya no hay más pasos, el requisito funcional continúa ejecutando el paso 3.

3. Conclusiones

En este capítulo se ha presentado un metamodelo que define la información mínima que debe tener un requisito funcional para poder aplicar las propuestas de generación de pruebas funcionales del sistema definidas en un capítulo posterior. Esto permite que no se tenga que adaptar una propuesta de definición de requisitos funcionales para poder ser utilizada en el proceso de generación de pruebas del sistema, o si se tiene que adaptar, que dichos cambios sean mínimos. En el siguiente capítulo, se presentará el metamodelo de pruebas funcionales del sistema.

Además, este metamodelo de requisitos funcionales, se definirá, también como perfil de UML en los anexos de este trabajo de tesis. Como ya se ha mencionado anteriormente, un perfil de UML es una extensión formal del propio lenguaje UML con el objetivo de definir nuevos conceptos a partir de constructores ya existentes en UML (como, por ejemplo, clasificador, clase, paquete, etc.) con el fin de dotarles de una semántica más precisa y concreta

Como se puede ver en el ejemplo de la sección 2, puede resultar más cómodo, intuitivo y compacto modelar requisitos funcionales utilizando sintaxis concretas, como la plantilla de texto utilizada, en lugar de trabajar directamente con instancias del metamodelo de requisitos funcionales. Este problema se retomará en el capítulo VII, dónde se presentarán ejemplos adicionales de sintaxis concretas para requisitos funcionales.

Capítulo V. Metamodelo de Pruebas Funcionales del Sistema

En el capítulo anterior se ha definido un metamodelo de requisitos funcionales, el cual ha permitido especificar de manera formal la información necesaria que debe tener un requisito funcional para poder generar pruebas funcionales del sistema a partir de él. Continuando con dicho trabajo, este capítulo adopta el mismo enfoque para las pruebas funcionales del sistema. A continuación, se define un metamodelo para las pruebas funcionales del sistema. Este metamodelo incluye los elementos necesarios para poder aplicar otros trabajos existentes basados en análisis de caminos y en cálculo de combinaciones de valores de prueba, que ya se han comentado en capítulos anteriores.

La organización de este capítulo se describe a continuación. En la sección 1 se define el metamodelo de pruebas funcionales del sistema. En la sección 2, se presentan ejemplos del uso de este metamodelo para la definición de un modelo de pruebas funcionales del sistema. Finalmente, en la sección 3, se exponen las conclusiones de este capítulo.

1. Definición del metamodelo de pruebas funcionales del sistema

Como se ha expuesto y justificado en el capítulo de planteamiento de la solución, el metamodelo de pruebas funcionales del sistema se va a organizar en tres grupos lógicos. El primer grupo es el grupo lógico de escenarios de prueba, donde se describen los elementos necesarios para modelar escenarios de prueba de los requisitos funcionales. De manera resumida, un escenario de prueba se define como un conjunto concreto de pasos a realizar sobre el sistema a prueba. El segundo grupo lógico es el grupo lógico de valores de prueba, donde se describen los elementos necesarios para representar variables operacionales y sus combinaciones. De manera resumida, una variable operacional se define como un punto de variabilidad de un requisito funcional, es decir, un punto en el que el comportamiento del requisito funcional puede ser distinto entre dos o más escenarios del mismo requisito funcional. El tercer grupo lógico es el grupo lógico de casos de prueba, donde se describen los elementos necesarios para combinar la información de los grupos anteriores para definir casos de pruebas. Estos grupos lógicos se describen con detalle en las siguientes secciones. Como se ha hecho en el capítulo anterior, se sigue el mismo formato que la documentación de la OMG sobre UML por lo que primero se presentan los diagramas con los grupos lógicos y, a continuación, se describen los elementos de cada uno de dichos grupos lógicos. En concreto, la sección 1.1 define los elementos que componen el grupo lógico de escenarios de prueba, la sección 1.2 define los elementos que componen el grupo lógico de valores de prueba, la sección 1.3 define los elementos que componen el grupo lógico de casos de prueba.

1.1. Escenarios de prueba

Los elementos y relaciones del grupo lógico de escenarios de prueba se definen en el diagrama de clases de la figura 5.1. Como se ha definido anteriormente, un escenario de prueba se define como un conjunto concreto de pasos a realizar sobre el sistema a prueba.

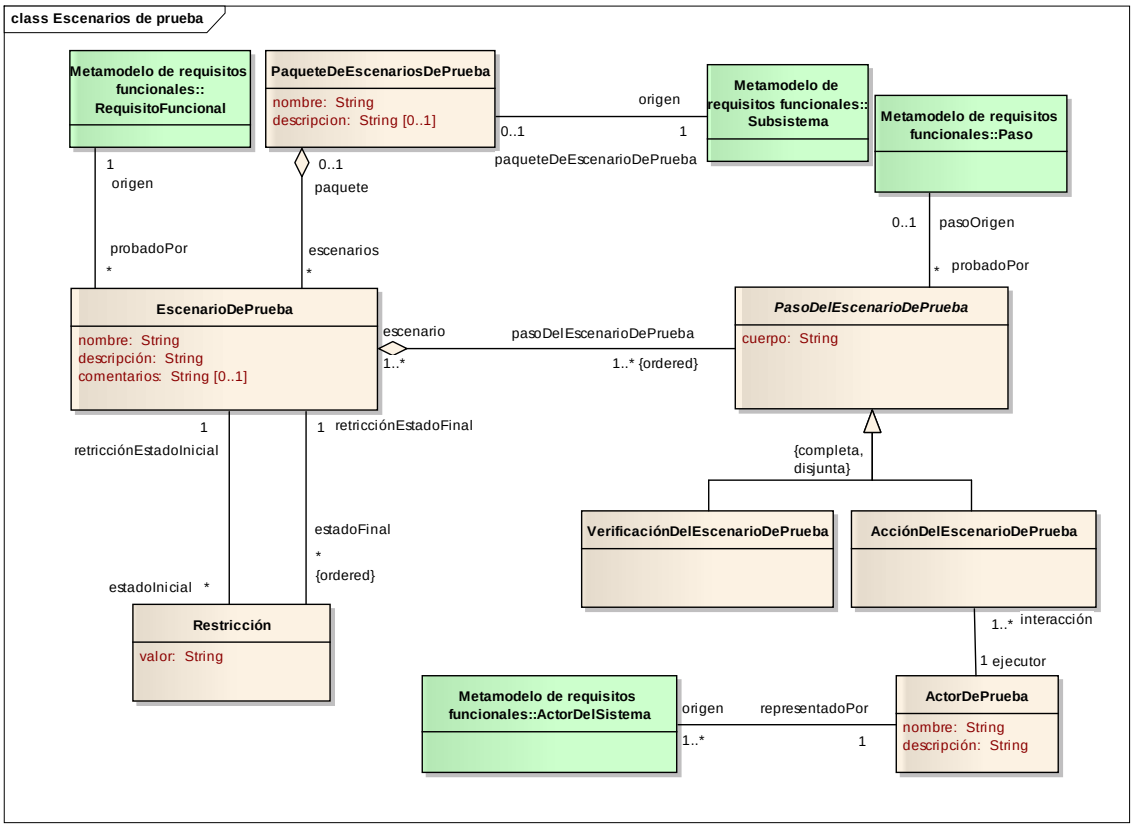


Figura 5.1. Grupo lógico escenarios de prueba.

En la figura 5.1 se muestran los elementos que se han definido para normalizar la información que recoge un escenario de prueba. El elemento principal es *EscenarioDePrueba*, el cual está relacionado con un requisito funcional y compuesto por pasos del escenario de prueba. Estos pasos del escenario de prueba se clasifican según sean realizados por el propio sistema o por un actor externo. Además, se han incluido varios elementos del metamodelo de requisitos funcionales (*Subsistema, Paso, ActorDelSistema* y *RequisitoFuncional*) para definir las relaciones de trazabilidad que permiten conocer el origen de los elementos del escenario de prueba. En la tabla 5.1, a modo de resumen, se citan todos los elementos del grupo lógico de escenarios de prueba y una breve descripción de los mismos (la cual será ampliada en posteriores secciones).

Tabla 5.1. Elementos del grupo lógico de escenarios de prueba.

Elemento	Descripción
AccionDelEscenarioDePrueba	Paso de prueba realizada por un elemento externo al sistema bajo prueba
ActorDePrueba	Elemento externo al sistema bajo prueba que participa en las pruebas
EscenarioDePrueba	Conjunto de pasos de prueba a realizar sobre el sistema a prueba
PaqueteDeEscenariosDePrueba	Agrupación de escenarios de prueba.
PasoDelEscenarioDePrueba	Interacción con el sistema bajo prueba
VerificaciónDelEscenarioDePrueba	Paso de prueba que describe un comportamiento realizado por el sistema bajo prueba que puede ser susceptible de verificación.

Además, se utilizará el elemento *Restricción*, ya definido con anterioridad en el metamodelo de requisitos funcionales.

El concepto central de este grupo lógico es el escenario de prueba (elemento *EscenarioDePrueba*), el cual prueba un fragmento del comportamiento de un requisito funcional. De la misma manera que un requisito funcional está compuesto por pasos, un escenario de prueba está compuesto por pasos del escenario de prueba. Cada paso del escenario de prueba define un acción que realiza el escenario de prueba para probar el fragmento del requisito funcional.

Como se puede ver en la figura 5.1, los pasos del escenario de prueba pueden ser de dos tipos en función del comportamiento realizado. El primer tipo, elemento *VerificaciónDelEscenarioDePrueba*, define un comportamiento en el cual el sistema realiza una acción. Dicho comportamiento podrá ser susceptible de ser verificado para dar por superado el escenario de prueba.

El segundo tipo, elemento *AccciónDelEscenarioDePrueba*, en cambio, define un comportamiento en el cual el sistema espera que un actor externo al mismo (elemento *ActorDePrueba*), realice una acción. Este comportamiento deberá ser realizado, por ejemplo, por la persona que esté realizando el escenario de prueba o, por ejemplo, por un módulo automático que interactúe con el sistema simulando el comportamiento de un actor de prueba.

Por último, los escenarios de prueba se agrupan y clasifican en paquetes de escenarios de prueba.

En las siguientes secciones se describen con detalle los elementos de la tabla 5.1. Se ha seguido en todo momento la estructura de OMG para la documentación de UML y especificaciones relacionadas, la cual es la misma estructura utilizada en los elementos del metamodelo de requisitos funcionales vista con anterioridad y ya descrita en el capítulo anterior.

1.1.1. AcciónDelEscenarioDePrueba

Una *AcciónDelEscenarioDePrueba* representa un comportamiento concreto de un *EscenarioDePrueba*, realizado por un *ActorDePrueba*.

Generalización

PasoDelEscenarioDePrueba (completa, disjunta)

Asociaciones

- ejecutor: ActorDePrueba[1]
  Entidad externa que realiza la acción del escenario de prueba.

Semántica

Este elemento es una especialización de un paso del escenario de prueba. En concreto, las acciones del escenario de prueba son los pasos del escenario de prueba que representan el comportamiento de alguien o algo externo al sistema bajo prueba.

Por ejemplo, cuando un requisito funcional define pasos en los que un actor externo introduce una información, o seleccione una opción del conjunto presentado por el sistema, dichos pasos se convertirán en acciones del escenario de prueba. Por este motivo, este elemento necesita conocer quién es el actor externo (relación con un *ActorDePrueba*) que realiza la acción.

1.1.2. ActorDePrueba

Un *ActorDePrueba* representa un participante externo al sistema involucrado en un escenario de prueba.

Asociaciones

- interacción: AcciónDelEscenarioDePrueba[1..*]

Conjunto de interacciones que los actores de prueba realizan sobre el sistema.

- origen: ActorDelSistema[1..*]
  Relación de trazabilidad con el elemento actor del sistema del metamodelo de requisitos funcionales del sistema.

Atributos

- nombre: String[1]
  Nombre del actor de prueba.

- descripción: String[1]
  Descripción del actor de prueba.

Semántica

Un actor de prueba simboliza un elemento externo que interactúa con el sistema durante la realización de un escenario, realizando acciones del escenario de prueba sobre el propio sistema. Un actor de prueba siempre representa a un actor del sistema, de ahí su relación de trazabilidad, sin embargo no siempre tienen por qué coincidir. Por ejemplo, si un escenario de prueba se realiza manualmente, el ingeniero de prueba puede actuar como *ActorDePrueba* realizando la función del actor del sistema implicado, o bien, el *ActorDePrueba* puede ser implementado como un componente automático dentro de una arquitectura de prueba.

1.1.3. EscenarioDePrueba

Un *EscenarioDePrueba* se define como un escenario concreto de un requisito funcional bajo prueba.

Asociaciones

- estadoInicial: Restricción[*]
  Conjunto de predicados que deben evaluarse a cierto para la ejecución del escenario de prueba.

- estadoFinal: Restricción[*]
  Conjunto de predicados que se evalúan a cierto una vez terminada la ejecución del escenario de prueba.

- pasoDelEscenarioDePrueba: PasoDelEscenarioDePrueba[1..*] {ordered}

Lista ordenada que define el comportamiento a realizar.

- origen: RequisitoFuncional[1]
  Relación de trazabilidad con el requisito funcional que ha dado origen a este escenario.
- paquete: PaqueteDeEscenarioDePrueba[0..1]
  Paquete al que pertenece el escenario de prueba.

Atributos

- nombre: String[1]
  Nombre del escenario de prueba.
- descripción: String[1]
  Descripción del escenario de prueba.
- comentarios: String[0..1]
  Comentarios adicionales.

Semántica

Un escenario de prueba es un conjunto de pasos extraídos del comportamiento del requisito funcional en el que sólo existe una única secuencia de ejecución sin ninguna alternativa.

1.1.4. PaqueteDeEscenariosDePrueba

Un *PaqueteDeEscenariosDePrueba* representa un conjunto de escenarios de prueba.

Asociaciones

- escenarios: EscenarioDePrueba[*]
  Escenarios de prueba de un paquete.
- origen: Subsistema[1]
  Asociación de trazabilidad con el subsistema del modelo de requisitos funcionales.

Atributos

- nombre: String[1]

Nombre del paquete de escenarios de prueba.

- descripción: String[0..1]
  Descripción del paquete de escenarios de prueba.

Semántica

Un paquete de escenarios de prueba tiene la misma semántica que el elemento subsistema, visto en el capítulo anterior, pero aplicado a escenarios de prueba.

1.1.5. PasoDelEscenarioDePrueba

Un *PasoDelEscenarioDePrueba* es un comportamiento realizado por un *EscenarioDePrueba*.

Asociaciones

- pasoOrigen: Paso [0..1]
  Relación de trazabilidad con el paso del requisito funcional que ha dado origen a este paso del escenario de prueba.

- escenario: EscenarioDePrueba[1..*]
  Conjunto de escenarios de prueba que realizan el paso de prueba.

Atributos

- cuerpo: String[1]
  Definición del comportamiento del paso del escenario de prueba.

Semántica

Un *PasoDelEscenarioDePrueba* puede ser una participación de un actor externo (elemento *AcciónDelEscenarioDePrueba*) que interactúa con el sistema, o bien una comprobación del estado del sistema (elemento *VerificaciónDelEscenarioDePrueba*). Esta clase es abstracta ya que no existirá un paso de prueba que no sea una verificación o una acción del escenario de prueba.

1.1.6. VerificaciónDelEscenarioDePrueba

El elemento *VerificaciónDelEscenarioDePrueba* describe un comportamiento realizado por el sistema que es susceptible de ser verificado para comprobar su validez.

Generalización

PasoDelEscenarioDePrueba (completa, disjunta)

Semántica

Este elemento no añade ningún nuevo atributo ni relación a la superclase de la que deriva. Su aportación es de índole semántica, permitiendo conocer que el valor definido en el atributo *cuerpo* define una actividad realizada por el sistema bajo prueba que podría ser verificada como, por ejemplo, el número de intentos a la hora de hacer un acceso al sistema, la información almacenada en un alta del sistema, etc.

1.2. Valores de prueba

En este grupo lógico se describen los elementos para definir variables operacionales y particiones relevantes. Además, se definen los elementos necesarios para poder calcular combinaciones de variables y particiones y mantener la trazabilidad con los requisitos funcionales. Nuevamente se utiliza el elemento *Restricción*, ya definido en el metamodelo de requisitos funcionales. Como se ha definido con anterioridad, una variable operacional se define como un punto de variabilidad de un requisito funcional, es decir, un punto en el que el comportamiento del requisito funcional puede ser distinto entre dos o más escenarios del mismo requisito funcional

En la figura 5.2 se muestran los elementos y relaciones del grupo lógico valores de prueba. También se han definido asociaciones con elementos del metamodelo de requisitos funcionales con el fin de que exista una trazabilidad que permita rastrear el origen de los artefactos de valores de prueba. Por simplicidad no se incluyen en los diagramas los atributos de los elementos del metamodelo de requisitos funcionales.

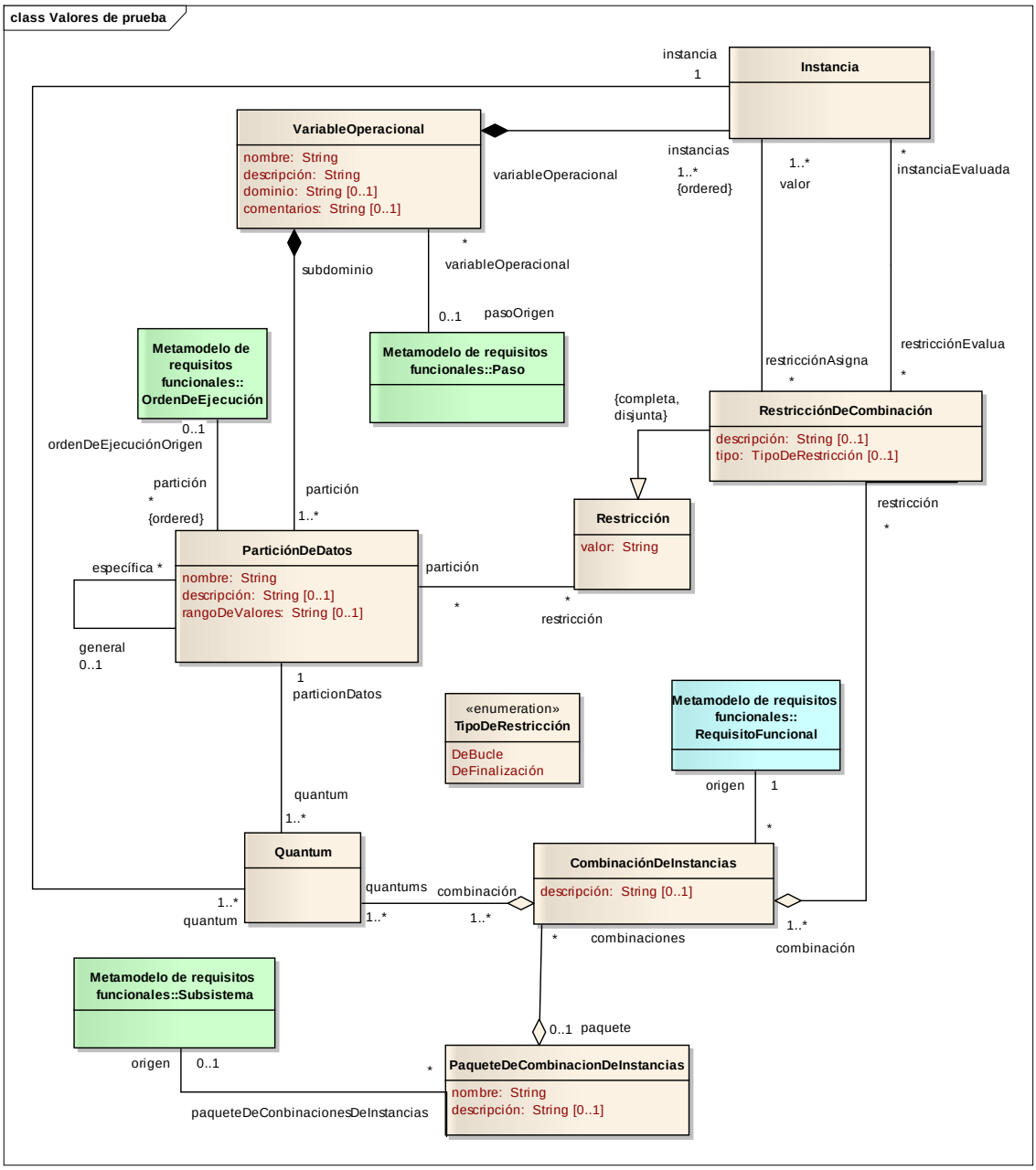


Figura 5.2. Grupo lógico valores de prueba.

En la tabla 5.2, modo de resumen, se citan todos los artefactos del grupo lógico de valores de prueba.

El concepto principal de este grupo lógico es la variable operacional. Las variables operacionales se identifican a partir de los pasos de un requisito funcional que tienen más de un orden de ejecución de salida. Además, las variables operacionales están compuestas de particiones de datos. Cada partición de datos agrupa todos los posibles valores de una variable operacional que provocan el mismo comportamiento.

Tabla 5.2. Elementos del grupo lógico de valores de prueba.

Elemento	Descripción
CombinacionDeInstancias	Conjunto ordenado de variables operacionales y valores que toman cada una de ellas.
Instancia	Variable operacional que toma un valor concreto en una combinación concreta
PaqueteDeCombinacionesDeInstancias	Agrupación de combinaciones de instancias.
ParticionDeDatos	Fragmento del dominio de datos de una variable operacional
Quantum	Instancia de una variable junto con el valor que toma
RestriccionDeCombinacion	Predicado booleano aplicado sobre instancias de variables operacionales
TipoDeRestricción	Enumeración de posibles tipos de restricciones
VariableOperacional	Punto de variabilidad en un requisito funcional.

Una vez identificadas las variables y particiones, las pruebas se realizan calculando combinaciones entre las particiones de todas las variables operacionales. Sin embargo, no todas las posibles combinaciones entre variables y particiones pueden ser posibles. Para ayudar a evitar combinaciones de particiones imposibles de reproducir en el sistema, se utilizan restricciones de combinación. Las restricciones de combinación funcionan como expresiones SI-ENTONCES y permiten condicionar los valores que pueden tomar las variables operacionales en base a los valores que ya han tomado otras variables operacionales y evitar combinaciones imposibles.

En las combinaciones generadas, es posible que una variable operacional tenga que ser evaluada más de una vez, y que tome valores de distintas particiones en cada evaluación. Para poder representar esto se introduce el concepto de instancia. Una instancia representa la participación de una variable operacional en una combinación. Este elemento permite identificar unívocamente cada una de las apariciones de una misma variable operacional en una misma combinación y permite asignar valores diferentes así como permitir definir restricciones de combinación que afecten sólo a determinadas instancias de una misma variable operacional.

Por tanto, los valores de prueba para un requisito funcional están compuestos de un conjunto de combinaciones en los que distintas instancias de variables operacionales toman valores en sus particiones.

Sin embargo, también puede presentarse el caso en que una misma instancia tenga valores distintos en distintas combinaciones del mismo requisito funcional. Por ejemplo, en un requisito funcional en el que un actor pueda elegir entre varias opciones, una misma instancia e una variable operacional tomará distintos valores en distintas combinaciones en base a las opciones disponibles. Por ello, se introduce el concepto de quantum. Un quantum es una asignación de una instancia a una partición en una combinación concreta. Esto permite identificar distintas asignaciones a una misma instancia en distintas combinaciones de instancia.

Finalmente, las combinaciones de instancia se agrupan y clasifican en paquetes de combinaciones de instancias.

En las siguientes secciones se describen con detalle los elementos de este grupo lógico siguiendo la misma estructura que en la descripción de los elementos del grupo lógico de valores de prueba.

1.2.1. CombinaciónDeInstancias

Este elemento representa una secuencia concreta de quantums, es decir, instancias de variables operacionales y los valores que toman.

Asociaciones

- quantums: Quantum[1..*]
  Conjunto de quantums que componen la instancia.
- restricción: RestricciónDeCombinación[*]
  Asociación de trazabilidad con el requisito funcional origen de la variable operacional.
- origen: RequisitoFuncional[*]
  Restricciones que se evalúan a cierto para una combinación concreta de instancias.
- paquete: PaqueteDeCombinacionDeInstancias[0..1]
  Paquete que contiene la combinación de instancias.

Atributos

- descripción: String[0..1]
  Descripción de la combinación.

Semántica

Una combinación de instancias es una secuencia concreta y ordenada de quantums del conjunto de variables operacionales. El orden viene determinado por el orden en que dichas variables operacionales aparecen durante la ejecución del requisito funcional. Así, si, por ejemplo, el comportamiento de un requisito funcional define un primer punto de variabilidad donde el actor puede cancelar o no la operación y un segundo punto de variabilidad donde se define un comportamiento adicional en función de si unos datos de entrada son correctos o incorrectos, una combinación de instancias presentará quantums de las variables correspondientes a ambos puntos de variación en el mismo orden, es decir, primero se tendrá en cuenta el quantum de la variable operacional que indica si se cancela o no y, después, el quantum de la variable operacional que indica si la información es correcta o incorrecta.

1.2.2. Instancia

Este elemento representa una instancia de una variable operacional.

Asociaciones

- restricciónAsigna: RestricciónDeCombinación[*]
  Restricción que asigna un valor a la instancia.
- restricciónEvalúa: RestricciónDeCombinación[*]
  Restricción que evalúa el valor de la instancia.
- variableOperacional: VariableOperacional[1]
  Variable Operacional que es instanciada.
- quantum: Quantum[1..*]
  Quantum en el que la instancia toma valor.

Semántica

Una instancia es la aparición de una variable operacional durante el comportamiento de un requisito funcional, y en donde toma un valor concreto de una de sus particiones. Una instancia solo puede tomar un único valor de una única partición. Este valor puede ser distinto en función del comportamiento. El número de instancias de una variable operacional varía en función del número de veces que dicha variable operacional puede ser evaluada durante el comportamiento de un requisito funcional.

Como puede verse en las asociaciones, una instancia puede participar en una restricción de combinación de valores desempeñando uno de dos roles posibles, siendo

evaluada para determinar si se cumple dicha restricción o siendo asignada para imponer que, en dicha restricción, una instancia toma siempre valores de una determinada partición.

1.2.3. PaqueteDeCombinacionDeInstancias

Un *PaqueteDeCombinaciónDePrueba* representa un conjunto de quantums, es decir, instancias de variables operacionales y los valores concretos que toman.

Asociaciones

- combinaciones: CombinaciónDeInstancias[*]
  Combinaciones de instancias y valores de un paquete.
- origen: Subsistema[1]
  Asociación de trazabilidad con el subsistema del modelo de requisitos funcionales.

Atributos

- nombre: String[1]
  Nombre del paquete de combinación de instancias.
- descripción: String[0..1]
  Descripción del paquete de combinación de instancias.

Semántica

Un paquete de escenarios de prueba tiene la misma semántica que el elemento subsistema o el elemento paquete de escenarios de prueba pero aplicado a combinaciones de instancias.

1.2.4. ParticiónDeDatos

El elemento *ParticiónDeDatos* representa una división del dominio de una variable operacional o de otra partición existente.

Asociaciones

- subdominio: VariableOperacional[1]
  Variable operacional a la que pertenece esta partición.
- general: ParticiónDeDatos[0..1]
  Partición de datos que agrupa varias particiones existentes.

- específica: ParticiónDeDatos[*]
  Partición de datos que refina o subdivide una partición de datos existente.

- restricción: Restricción[*]
  Predicados que deben evaluarse a cierto para una determinada partición.

- ordenDeEjecuciónDeOrigen: OrdenDeEjecución[0..1]
  Relación de trazabilidad con el orden de ejecución.

- quantum: Quantum[1..*]
  Quantums que toman valor en esta partición.

Atributos

- nombre: String[1]
  Nombre de la partición de datos.

- descripción: String[0..1]
  Descripción de la partición de datos.

- rangoDeValores: String[0..1]
  Descripción del rango de valores del dominio de la variable operacional representados en esta partición.

Semántica

Una partición es un conjunto de valores para los que siempre se obtiene el mismo resultado. Este conjunto de valores puede ser un subconjunto de todo el dominio de una variable operacional o un subconjunto de otra partición. Así por ejemplo, a partir una variable operacional que represente la información introducida por un actor externo, podrían definirse dos particiones distintas, una que englobara el conjunto de informaciones correctas y otro que englobara el conjunto de informaciones incorrectas. Si se deseara un refinamiento más detallado, sería posible, por ejemplo, subdividir la partición de información incorrecta en nuevas particiones que indicaran las causas concretas por la que esa información es incorrecta (por ejemplo, ausencia de fragmentos de información, uso de caracteres no válidos, etc.). El atributo *rangoDeValores* es una descripción textual del conjunto de valores abarcado por la partición. Además, es posible definir el rango de una partición utilizando elementos *Restricción*.

1.2.5. Quantum

Este elemento representa una instancia que toma un valor concreto en una combinación concreta.

Asociaciones

- particiónDatos: ParticioneDatos[1]
  Partición de la que se toma el valor de la instancia.
- instancia: Instancia[1]
  Instancia que toma un valor concreto.
- combinación: CombinaciónDeInstancia[*]
  Combinaciones en las que participa un quantum.

Semántica

Una instancia de una variable operacional puede aparecer en distintas secuencias y, en cada secuencia tomar un valor distinto. Por ello, el elemento *Quantum* asocia la aparición de una instancia en una combinación de instancia, junto con el valor concreto que toma en dicha aparición.

Así, por ejemplo, si se tiene un requisito funcional que define el acceso a un sistema mediante un nombre y una contraseña de usuario, una posible variable operacional (punto donde el comportamiento puede variar) serán los valores del nombre y la contraseña. Para este ejemplo se pueden definir dos particiones para la variable operacional: una que agrupe los nombres y claves correctos y otra que agrupe los nombres y claves incorrectos.

En un posible escenario de este requisito funcional, en el que se describe un primer intento fallido de entrar al sistema y un segundo intento con éxito, se contará con dos instancias de esta variable operacional, ya que la variable operacional aparece una vez por cada intento. El primer intento debe tomar un valor n la partición de nombres y claves incorrectos, lo cual se modela con un quantum. De la misma manera, la segunda instancia debe tomar un valor en la partición de nombres y claves correctos, lo cual se modela con otro quantum.

1.2.6. RestricciónDeCombinación

Este elemento representa una restricción que condiciona las instancias que participan en una combinación de instancias.

Generalización

Restricción (incompleta y disjunta).

Asociaciones

- instanciaEvaluada: Instancia[*]
  Instancias que deben aparecer en la restricción para su aplicación.
- valor: Instancia[1..*]
  Instancias que toman valores.
- combinación: CombinaciónDeParticiones[1..*]
  Conjunto de combinaciones en las que se aplica una restricción.

Atributos

- descripción: String[0..1]
  Descripción de la restricción.
- tipo: TipoRestricción[0..1]
  Tipo de la restricción.

.**Restricciones**

[1] self.instanciaEvaluada→**forAll**(i | self.valor → includes(i) = false)

Semántica

Una restricción de combinación de particiones es un refinamiento del elemento restricción. Este elemento define una restricción que se aplicará sobre instancias de variables operacionales. Una restricción de combinación de particiones está compuesta de dos partes. La primera parte, definida mediante la asociación *instanciaEvaluada,* indica aquellas instancias y valores que deben aparecer en la combinación para poder aplicar la restricción de combinación. La segunda parte, definida mediante la asociación *valor*, indica las instancias que deben tomar valor y el valor concreto que deben tomar al aplicarse la restricción. La restricción [1] indica que ninguna instancia puede participar simultáneamente en ambas partes.

Con este elemento es posible definir al menos dos tipos de restricciones de combinaciones de valores. Estos tipos son: restricciones de bucle, que aparecen cuando existe un fragmento de comportamiento de un requisito funcional que puede repetirse y dicho comportamiento incluye variables operacionales y restricciones de finalización, que aparecen

cuando un requisito funcional termina. Estos dos tipos se describen con más detalle en la sección del elemento *TipoDeRestricción*.

1.2.7. TipoDeRestricción

Este tipo enumerado describe dos posibles tipos de restricciones de combinación que se contemplan en este trabajo de tesis. Sus posibles valores y su semántica se definen a continuación.

- DeBucle: este valor indica una restricción que se utiliza para detectar cuándo se genera un bucle cuando una instancia de una variable operacional toma un valor en una partición concreta. Es decir, indica que, por ejemplo, si una instancia toma valor en una partición concreta, el requisito funcional entrará en un bucle, repetirá un conjunto de pasos y se volverá a evaluar una nueva instancia de la misma variable operacional. Con esta restricción se evita entrar en una combinación de instancias que nunca tenga fin, es decir, evita intentar generar combinaciones que constantemente hagan que el requisito funcional repita la misma secuencia de pasos.

- DeFinalización: esta restricción se utiliza para detectar cuándo termina el comportamiento bajo prueba y, por tanto, no es necesario asignar más instancias ni valores a una combinación. Con esta restricción se evitan combinaciones que nunca llegarán a darse según el comportamiento definido en los requisitos.

Los dos tipos de restricciones comentados aquí son dos tipos bastante comunes a la hora de definir restricciones de combinaciones. Sin embargo, se contempla la posibilidad de definir restricciones de combinación específicas que no sean de ninguno de los dos tipos definidos.

1.2.8. VariableOperacional

Este elemento representa un punto en el comportamiento de un requisito funcional que puede cambiar entre dos ejecuciones (escenarios) de dicho caso de uso.

Asociaciones

- partición: ParticiónDeDatos[1..*]
  Particiones del dominio de la variable operacional.

- pasoOrigen: Paso [0..1]

Relación de trazabilidad con el paso del requisito funcional.

- instancias: Instancia[1..*]
  Conunto de instancias de una variable operacional.

Atributos

- nombre: String[1]
  Nombre de la variable operacional.
- descripción: String[1]
  Descripción de la variable operacional.
- dominio: String[0..1]
  Naturaleza del dominio de la variable operacional.
- comentarios: String[0..1]
  Comentarios adicionales.

Restricciones

[1] (self.paso →isEmpty() **implies** self.ordenDeEjecución→isEmpty() = **false**) **and**
(self.ordenDeEjecución→isEmpty()**implies** self.paso →isEmpty()= **false**)

Semántica

Las variables operacionales se identifican a partir de los requisitos funcionales localizando aquellos puntos en los que una instancia o ejecución del requisito funcional pueda ser distintas de otra instancia o ejecución del mismo requisito funcional. Las alternativas posibles conforman las particiones de dicha variable operacional.

1.3. Casos de prueba

En esta sección se describen los elementos del tercer y último grupo lógico, el grupo lógico de casos de prueba. El principal objetivo de los casos de prueba es combinar los elementos de los dos grupos lógicos anteriores, escenarios de prueba y valores de prueba. Para ello, se toma como base el grupo lógico de escenarios de prueba y se extienden sus elementos para incluir relaciones a los elementos del grupo lógico de valores de prueba. En la figura 5.3 se muestran los elementos y relaciones del grupo lógico casos de prueba.

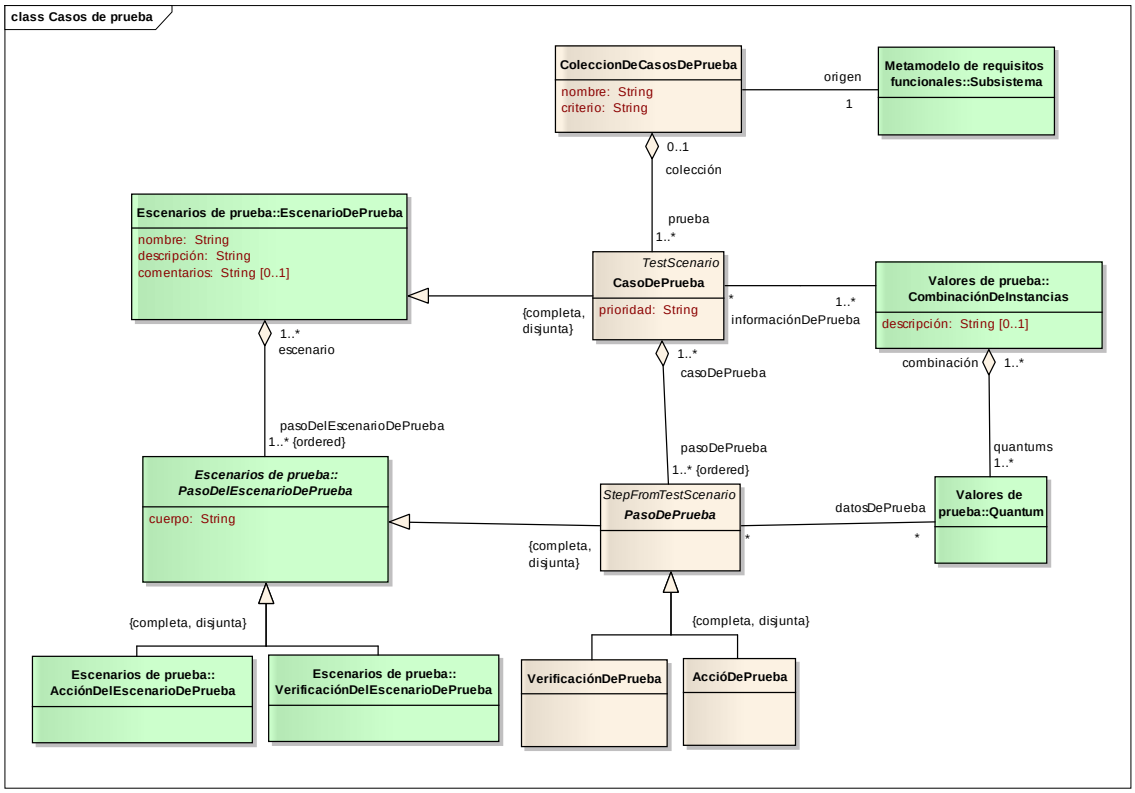


Figura 5.3. Grupo lógico de casos de prueba.

Los elementos del metamodelo se describen brevemente en la tabla 5.3. Las descripciones de los elementos *AcciónDelEscenarioDePrueba, PasoDelEscenarioDePrueba, VerificaciónDelEscenarioDePrueba* por un lado y *AccióndePrueba, PasoDePrueba* y *VerificciónDePrueba* por el otro son las mismas ya que su semántica es la misma, siendo la principal diferencia que los elementos del grupo lógico de casos de prueba también se relacionan con elementos del grupo lógico de valores de prueba.

Tabla 5.3. Elementos del grupo lógico de casos de prueba.

Elemento	Descripción
AccionDePrueba	Paso de prueba realizada por un elemento externo al sistema bajo prueba
CasoDePrueba	Prueba funcional del sistema
ColecciónDeCasosDePrueba	Conjunto de pruebas funcionales del sistema
PasoDePrueba	Interacción con el sistema bajo prueba
VerificaciónDePrueba	Interacción con el sistema bajo prueba

Este grupo lógico toma los conceptos definidos en el grupo lógico de escenarios de prueba y los extiende para relacionarlos con los conceptos el grupo lógico de valores de prueba.

El elemento principal de este grupo lógico es el elemento *CasoDePrueba*, el cual es una extensión de un escenaro de prueba. Por tanto, un caso de prueba contiene toda la información del escenario de prueba al que extiende. Sin embargo, un caso de prueba está compuesto de pasos de prueba. De manera análoga, los pasos de prueba extienden el concepto de paso de escenario de prueba. La misma clasificación que se hizo para los pasos del escenario de prueba (según la acción fuera realizada por el sistema o por un actor externo) se realiza también para los pasos de prueba. En concreto, el elemento *VerificaciónDePrueba*, extiende al elemento *VerificaciónDelEscenarioDePrueba*, mientras que el elemento *AcciónDePrueba* extiende al elemento *AcciónDelEscenarioDePrueba*.

La relación que se establece con los conceptos del grupo lógico de valores de prueba se realiza en dos puntos. En primer lugar, el caso de prueba se relaciona con las combinaciones de instancias que contienen los quantums (es decir, las instancias de las variabes operacionales implicadas y sus valores asignados) utilizadas en el caso de prueba. Esta relación se lleva a un nivel más alto de detalle, ya que en los pasos en los que hay implicada una variable operacional (por ejemplo porque sea necesario suministrar una información al sistema, o porque sea el sistema el que suministra una información a un actor, o, por ejemplo, porque el sistema realice una acción en función de su estado actual), se relacionarán con los quantums de las combinaciones de instancias implicadas.

Por último, los casos de prueba se agrupan y clasifican en colecciones de casos de prueba.

En las siguientes secciones se describen con más detalle los elementos de la tabla anterior, siguiendo también la misma estructura que en los dos grupos lógicos anteriores.

1.3.1. AcciónDePrueba

Este elemento representa una acción del escenario de prueba con los valores de prueba asociados.

Generalización

PasoDePrueba (completa y disjunta).

Semántica

Una acción de prueba refina la semántica del elemento acción del escenario de prueba añadiendo una relación con los quantums implicados en la acción de prueba, si hubiera alguno.

1.3.2. CasoDePrueba

Este elemento representa un escenario de una prueba con sus valores de prueba asociados.

Generalización

EscenarioDePrueba (completa y disjunta).

Asociaciones

- colección: ColecciónDeCasosDePrueba[0..1]
  Colección de casos de prueba a los que pertenece un caso de prueba.
- informaciónDePrueba: CombinaciónDeInstancias[1..*]
  Combinaciones de instancias utilizadas en el caso de prueba.
- pasoDePrueba: PasoDePrueba[1..*] {ordered}
  Pasos de prueba que pertenecen a un caso de prueba.

Atributos

- prioridad: String
  Prioridad del caso de prueba.

Semántica

Un caso de prueba es una especialización de un escenario de prueba en el cual los pasos del escenario de prueba se relacionan con los quantums (es decir, con una instancia de una variable operacional que toma un valor concreto) del grupo lógico de valores de prueba.

Este elemento añade un nuevo atributo para indicar la prioridad. Como valor inicial de la prioridad de un caso de prueba puede tomarse la prioridad del requisito funcional de origen. Algunos ejemplos del uso de la prioridad en los casos de prueba pueden ser indicar aquellos casos que prueban requisitos más prioritarios, indicar los casos de prueba que deben realizarse en primer lugar, indicar los casos de prueba que tienen preferencia para ser automatizados, etc.

1.3.3. ColecciónDeCasosDePrueba

Un elemento *ColecciónDeCasosDePrueba* representa un conjunto de casos de prueba.

Asociaciones

- prueba: CasoDePrueba[1..*]
  Casos de prueba pertenecientes a una colección.
- origen: Subsistema[1]
  Asociación de trazabilidad con el subsistema del modelo de requisitos funcionales.

Atributos

- nombre: String[1]
  Nombre de la colección de casos de prueba.
- criterio: String[1]
  Descripción del criterio utilizado para la agrupación de casos de prueba.

Semántica

Una colección de casos de prueba recoge la misma semántica que el elemento subsistema pero aplicada a casos de prueba.

El atributo criterio describe la clasificación hecha sobre los casos de prueba. Ejemplos de criterios pueden ser por ejemplo, el actor principal (el cual inicia el caso de uso) participante en el caso de prueba, la prioridad de los casos de prueba, etc.

1.3.4. PasoDePrueba

Este elemento representa una acción concreta realizada por el sistema o por un actor como parte de la ejecución de un caso de prueba.

Generalización

PasoDelEscenarioDePrueba (incompleta y solapada).

Asociaciones

- casoDePrueba: CasodePrueba[1..*]
  Casos de prueba a los que pertenece una acción de pruebas.
- datosDePrueba: Quantum[*]

Quantums de la combinación de instancias con los valores necesarios para la realización del paso.

Semántica

Un paso de prueba es una especialización del elemento *PasoDelEscenarioDePrueba*. La semántica de este elemento es la misma semántica que la de su supertipo, sin embargo, este elemento permite relacionar los pasos de prueba con las instancias de variables operacionales que pudieran participar en la realización de este paso.

Tal y como indican los clasificadores de la generalización, toda acción de prueba debe ser también o una acción de prueba o una verificación de prueba, de manera similar a lo visto en el grupo lógico de escenarios de prueba.

1.3.5. VerificaciónDePrueba

Una verificación de prueba representa una extensión de una verificación del escenario de prueba.

Generalización

PasoDePrueba (completa y disjunta).

Semántica

Una verificación de prueba refina la semántica del elemento verificación del escenario de prueba añadiendo una relación con los quantums implicados en la acción de prueba, si hubiera alguno.

2. Ejemplos

En esta sección se presenta un ejemplo de cómo definir modelos conformes al grupo lógico de pruebas funcionales presentado en las secciones anteriores. Para ello, se va a tomar como base el requisito funcional visto como ejemplo en el capítulo anterior. Para facilitar la comprensión de los ejemplos de esta sección, se vuelve a presentar dicho requisito funcional en la tabla 5.4

Tabla 5.4. Ejemplo de requisito funcional en prosa estructurada.

Requisito funcional:
Alta de nuevo elemento.

Descripción:
Este requisito funcional describe el comportamiento del sistema cuando un usuario desea dar de alta un nuevo elemento.

Precondiciones:
No.

Poscondiciones:
El elemento está dado de alta en el sistema.

Secuencia principal

1. El usuario solicita dar de alta un nuevo elemento en el sistema.
2. El sistema solicita la información del elemento.
3. El usuario introduce la información del elemento.
4. El sistema da de alta al elemento.

Secuencia alternativa

1.1 Si se ha superado el límite de elementos del sistema, entonces se muestra un mensaje de error y el resultado es finalizar la ejecución.

3.1. Si la información es incorrecta, entonces se muestra un mensaje de error y el resultado es ir al paso 2.

En primer lugar, se muestra un modelo del grupo lógico de escenarios de prueba (figura 5.4). El escenario de prueba que contempla la secuencia principal del requisito funcional (para mejorar la legibilidad se han omitido las asociaciones de trazabilidad).

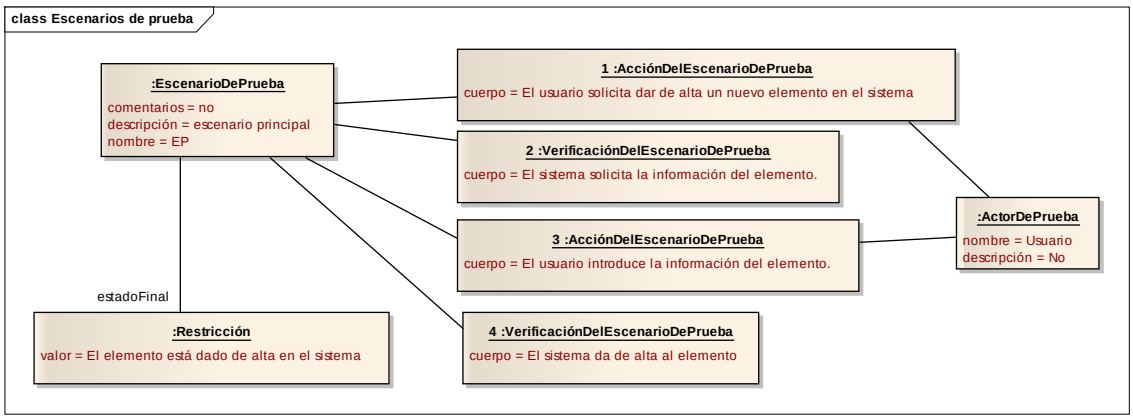


Figura 5.4. Modelo de escenarios de prueba (escenario principal).

En el modelo de la figura 5.4, se presenta un escenario de prueba con cuatro pasos, dos de ellos acciones del escenario de pruebas realizadas por un actor prueba que es quien

realiza las acciones y dos verificaciones del escenario de pruebas que nunca tienen una asociación con un actor, ya que, como su semántica define, siempre son realizadas por el sistema.

Además, el escenario de prueba no cuenta con un estado inicial ya que el requisito funcional de origen no define ninguna precondición. Sin embargo, sí se ha definido una restricción que indica un estado final.

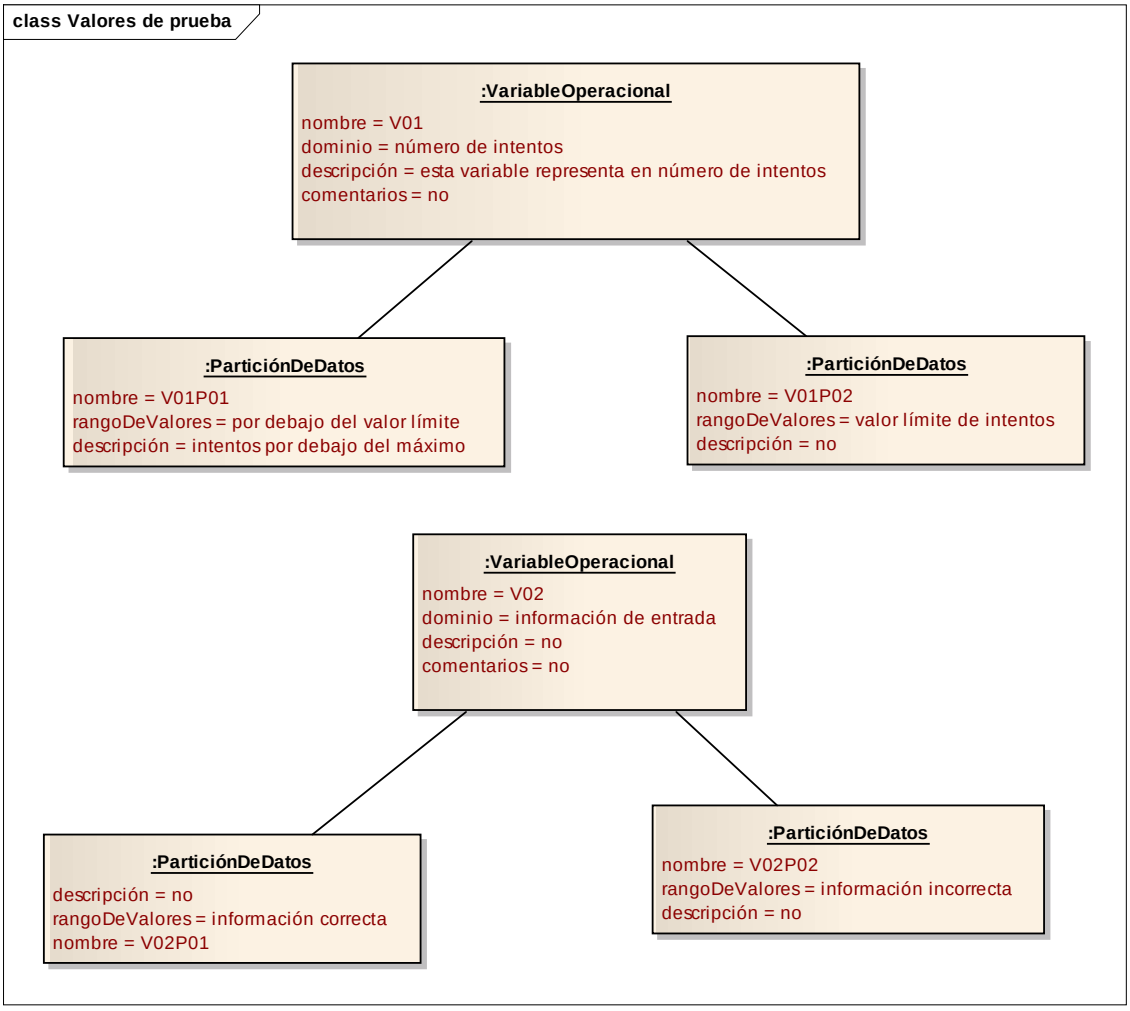


Figura 5.5. Modelo de valores de prueba (variables y particiones).

En la figura 5.5, se muestra una instancia del grupo lógico de valores de prueba con las variables operacionales identificadas y sus particiones. En el requisito funcional de ejemplo, existen dos puntos en los que el comportamiento del requisito funcional puede variar: si se ha superado el límite del sistema y si la información es incorrecta, por lo que habrá dos variables operacionales en el modelo. A pesar de existir una variable de información de entrada, no se han podido definir elementos de tipo atributo para prueba dado que el requisito funcional no describe la estructura de la información de entrada.

Las dos particiones definidas en el modelo de la figura 5.5 surgen de los comportamientos distintos posibles a partir de un punto de variabilidad del requisito funcional de ejemplo. En el requisito funcional se definen dos comportamientos para cada un de los dos puntos de variabilidad, lo que se modela como dos particiones para cada variable operacional.

Dado que hay un bucle, es decir, una parte del comportamiento del requisito funcional que puede repetirse varias veces en un escenario, serán necesarias varias instancias de las mismas variables. Por simplicidad se supondrá que dicho bucle puede ejecutarse ninguna o una vez, por lo que será necesario dos instancias de cada variable operacional. También, se identifican dos restricciones. La primera impide que el bucle se produzca más de una vez, la segunda impide que las instancias de la variable V02 tomen valores si se ha alcanzado el máximo número de intentos. En la figura 5.6 se muestra el fragmento del modelo de valores de prueba correspondiente a instancias y restricciones.

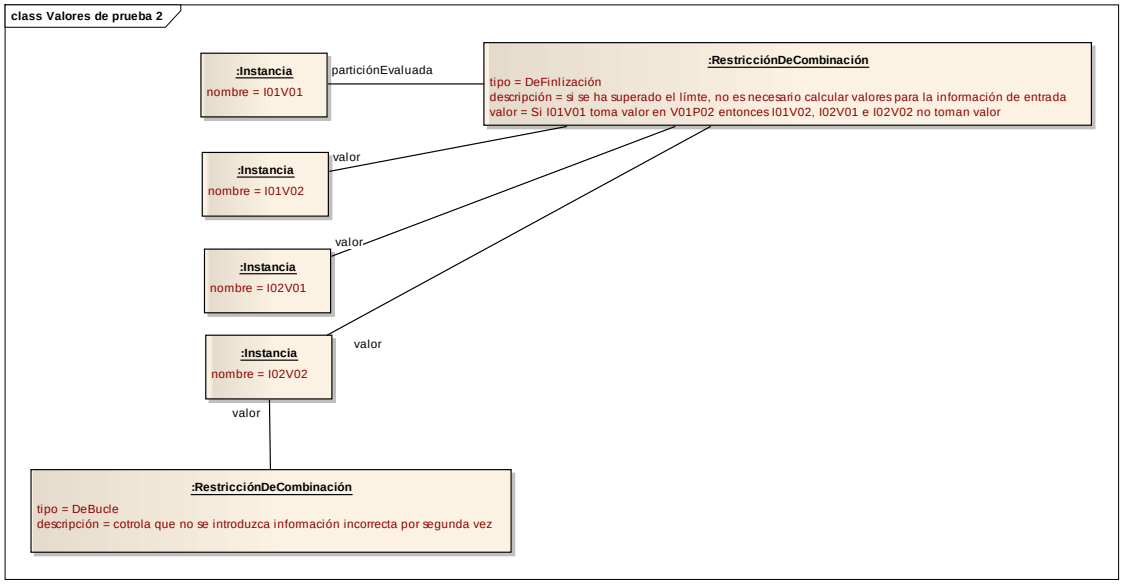


Figura 5.6. Modelo de valores de prueba (instancias y restricciones).

Con cuatro instancias, cada una pudiendo tomar dos posibles valores, el número de combinaciones es de 16. Sin embargo, aplicando las restricciones de ejemplo, es posible reducir este número hasta 9 ya que estas restricciones evitan combinaciones que no pueden presentarse según el requisito del sistema como, por ejemplo, introducir un nombre y contraseña correctos después de introducir un nombre y contraseña ya correctos. En la figura 5.7 se muestra la combinación que corresponde con la secuencia principal del requisito funcional (solo participan dos instancias ya que no hay bucle).

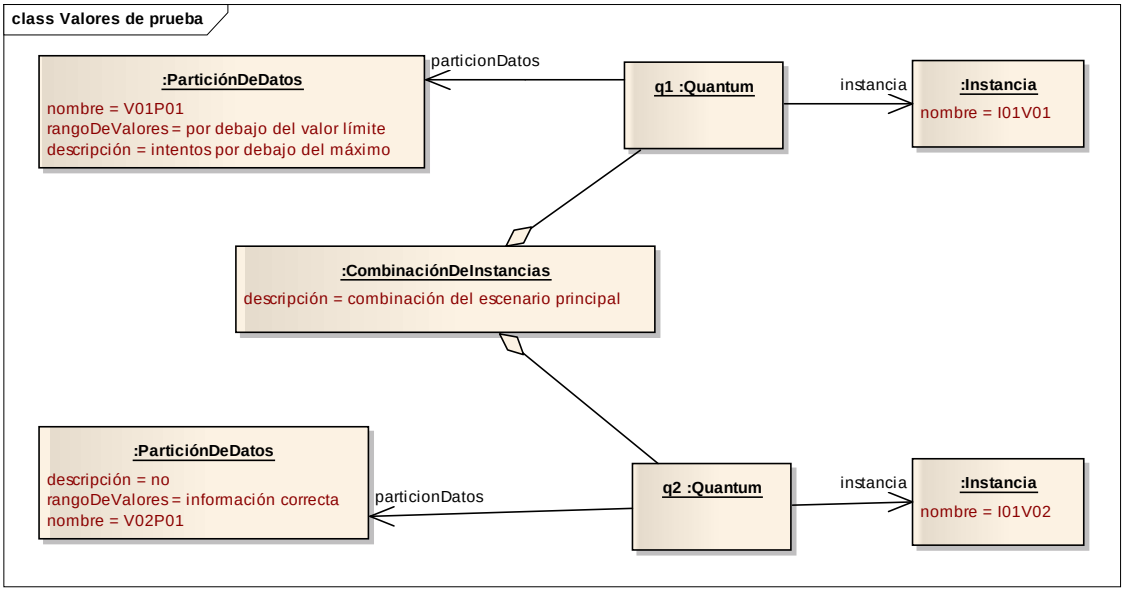


Figura 5. 7. Modelo de valores de prueba (combinación de instancias).

En la combinación de instancias, se puede ver cómo el elemento *quantum* participa en la combinación asociando a cada instancia la partición de donde tomará su valor. Así, el quantum q1 indica que la primera instancia de la combinación será I01V1 y que dicha instancia tomará un valor por debajo del valor límite de intentos.

Por último, en la figura 5.8, se muestra el modelo del grupo lógico de casos de prueba que corresponde al escenario principal del requisito funcional.

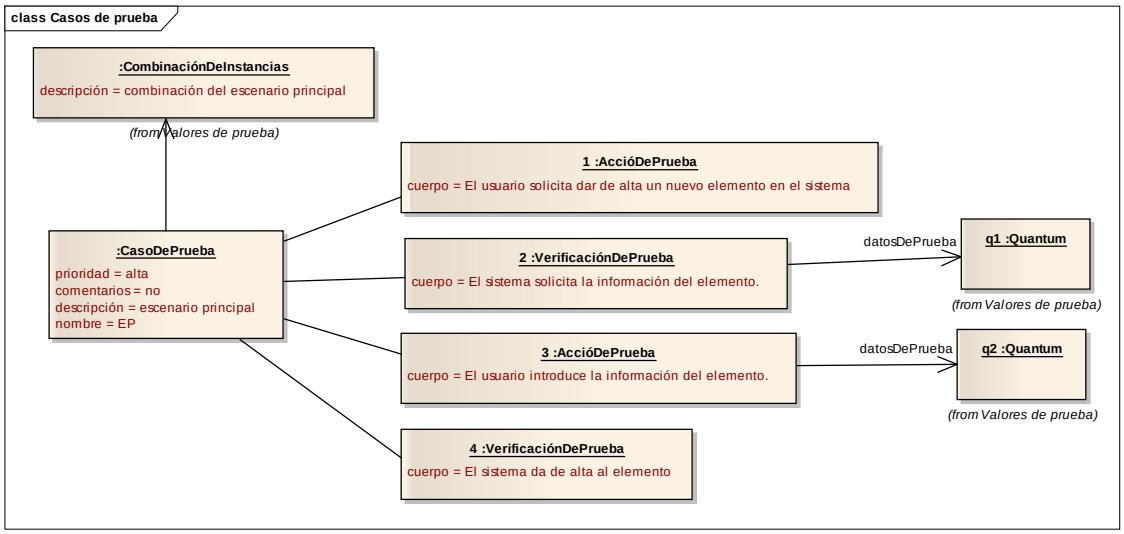


Figura 5.8. Modelo de casos de prueba.

Los elementos del modelo de casos de prueba de la figura 5.8 tienen toda la información de los elementos del modelo de escenarios de prueba de la figura 5.5, tal y como

se especificó en la definición de los elementos en el grupo lógico de casos de prueba (por legibilidad se ha omitido el actor de prueba). Además, tanto el elemento caso de prueba, como las verificaciones de prueba, ahora incluyen relaciones a los elementos del modelo de valores de prueba relacionados.

En la siguiente sección, se presentan las conclusiones de este capítulo.

3. Conclusiones

En este capítulo se ha presentado un metamodelo, dividido en tres grupos lógicos, que define la información de pruebas funcionales del sistema. Como ya se ha expuesto en el planteamiento de la solución, este metamodelo será el resultado obtenido del proceso de transformación que permitirá obtener casos de prueba funcionales del sistema a partir de los requisitos funcionales. En el siguiente capítulo, se presenta este proceso mediante la definición de transformaciones de instancias del metamodelo de requisitos funcionales a instancias del metamodelo de pruebas funcionales del sistema. Además, en los apéndices de este trabajo de tesis se incluye la definición de un perfil de UML para los grupos lógicos del metamodelo de pruebas funcionales del sistema.

Capítulo VI. Transformaciones para la Generación de Pruebas Funcionales del Sistema

En los capítulos anteriores se ha definido y formalizado la información que deben contener los requisitos funcionales y los artefactos de prueba funcionales del sistema (escenarios de prueba, combinaciones de variables operacionales y casos de prueba) mediante metamodelos. El objetivo de este capítulo es especificar formalmente un proceso basado en transformaciones QVT para generar artefactos de casos de prueba a partir de un modelo de requisitos funcionales.

La organización de este capítulo se especifica a continuación. En la sección 1 se expone el proceso de generación de pruebas funcionales del sistema a partir de los metamodelos vistos en los capítulos anteriores. A continuación, en las secciones de la 2 a la 5, se especifican las transformaciones que permitirá enriquecer el modelo de requisitos funcionales, generar escenarios de prueba, valores de prueba y casos de prueba. Para finalizar, en la sección 6 se exponen las conclusiones de este capítulo.

1. Introducción

Esta introducción se divide en dos secciones. En la primera sección se presenta una visión global del proceso de obtención de pruebas funcionales del sistema partir de los requisitos funcionales, utilizando transformaciones basadas en los metamodelos especificados en los dos capítulos anteriores. En la segunda sección se describe la estructura que tendrán las transformaciones definidas en el lenguaje QVT.

1.1. Visión global del proceso de generación de pruebas funcionales del sistema

En el capítulo III se planteó el proceso de generación de pruebas funcionales. A continuación se describe con mayor detalle cómo se va a llevar a cabo dicho proceso a lo largo de este capítulo, para ello se toma como base la figura 6.1, presentada con anterioridad en el capítulo 3.

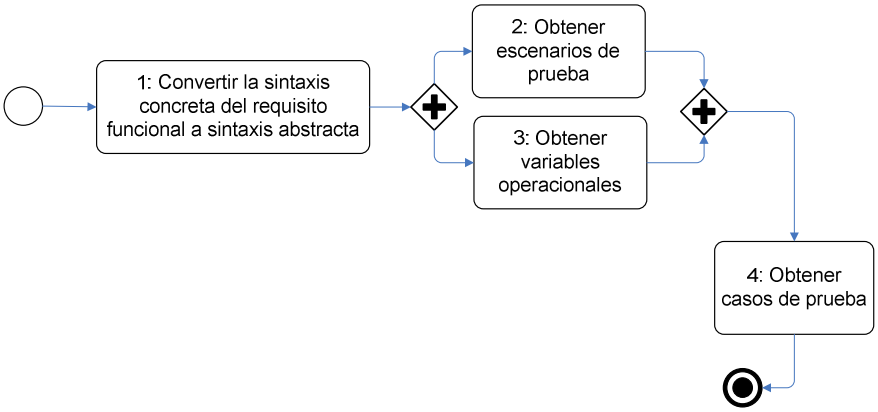


Figura 6.1. Descripción del proceso de generación de pruebas funcionales del sistema.

Tras los dos capítulos anteriores ya se cuenta con la herramienta necesaria para definir un requisito funcional mediante sintaxis abstracta (actividad 1 de la figura 6.1), en este caso, dicha sintaxis abstracta es el metamodelo de requisitos funcionales, También se cuenta con las herramientas necesarias para definir los escenarios de prueba, valores de prueba y casos de prueba. Por tanto, el siguiente punto a abordar es definir cómo realizar las actividades 2, 3 y 4 de la figura 6.1. Para ello, como ya se adelantó en capítulos anteriores, se van a utilizar transformaciones de modelo a modelo. La aplicación de las transformaciones se muestra en la figura 6.2.

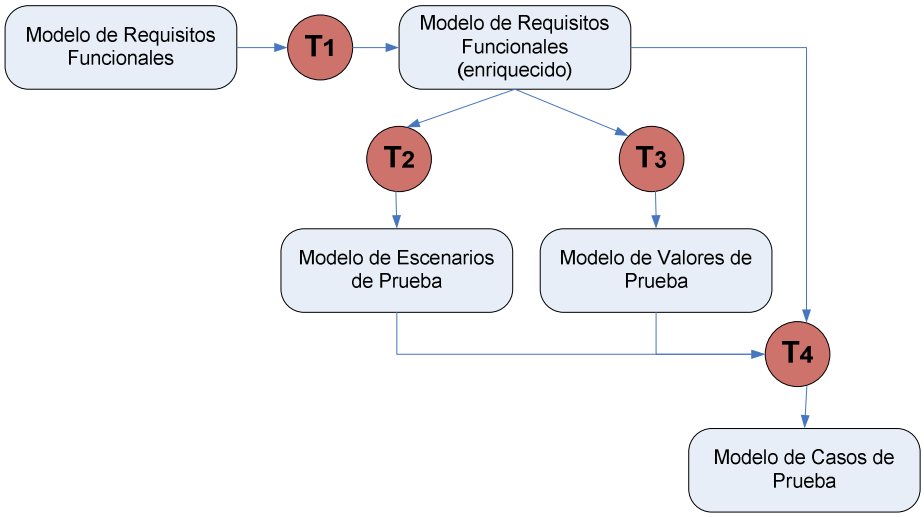


Figura 6.2. Aplicación de las transformaciones.

Como se puede ver en la figura 6.2, los escenarios de prueba, los valores de prueba y los casos de prueba definidos en la figura 6.1 se formalizan y representan mediante las instancias de los metamodelos vistos en los dos capítulos anteriores. De manera análoga, la

obtención de dichos elementos se realiza mediante las transformaciones indicadas en la figura 6.2 con la letra T.

El comportamiento de un requisito funcional viene definido con unos elementos, que, de manera simplificada, componen una máquina de estados. Será necesario, por tanto, explorar dicha máquina de estados recorriendo sus posibles caminos o escenarios (ambos términos se utilizan como sinónimos en este trabajo de tesis). Para evitar la duplicación de código, reducir el tamaño de las transformaciones y facilitar su comprensión, se añade un paso adicional consistentes en aplicar una transformación para obtener los escenarios de un requisito funcional, y tenerlos ya disponibles a la hora de aplicar las transformaciones para obtener escenarios de prueba y valores de prueba. Esta transformación se puede ver en la figura 6.2 como la transformación que permite obtener un modelo de requisitos funcionales enriquecido.

1.2. QVT como lenguaje de especificación formal de transformaciones

El lenguaje QVT, que ya se presentó al principio de este trabajo de tesis, se basa en el concepto de transformación y en el concepto de mapeo. Una transformación se puede definir como un conjunto de pasos para construir un modelo de salida a partir de un modelo de entrada. Un mapeo, en cambio, es definido por QVT como una operación (o un subconjunto de pasos) que implementa una parte de una transformación.

Las transformaciones se especifican en base a los elementos definidos en un metamodelo y se aplican sobre las instancias de dicho metamodelo. Por ejemplo, utilizando un ejemplo clásico de la literatura, una transformación de código fuente (en un lenguaje de programación orientado a objetos) a SQL, podría indicar que una tabla SQL se construye a partir de una clase. Dicha transformación se aplicará sobre un fragmento de código concreto para obtener un conjunto de tablas SQL concretas.

Una transformación definida en QVT, contiene una cabecera, la especificación de las claves y la especificación de todos los mapeos y funciones auxiliares necesarios. La cabecera indicará además cuáles son los metamodelos de entrada y salida. Estos metamodelos permiten especificar de qué tipo serán los modelos sobre los que se pueda aplicar la transformación. Como ejemplo, si una transformación especifica como entrada el metamodelo A y como salida el metamodelo B, será una transformación que se pueda aplicar sobre cualquier modelo conforme al metamodelo A, para obtener como resultado un modelo conforme al metamodelo B.

Una clave es un mecanismo establecido en QVT para identificar unívocamente una instancia de un elemento. Esto permite a QVT distinguir cuándo debe crear una nueva instancia y cuándo debe utilizar una instancia ya creada.

Además de transformaciones y mapeos, el lenguaje QVT también permite definir funciones auxiliares que pueden ser referenciadas desde cualquier mapeo. Las funciones auxiliares utilizadas en la especificación formal de las transformaciones de este capítulo se dividen en dos tipos: helpers y queries (utilizamos los términos originales en inglés). La principal diferencia entre ambas es que una función helper no define el retorno de ningún valor mientras que una función query sí define un valor de retorno.

A continuación, hasta el final del capítulo, se especifican formalmente cuatro transformaciones para generar pruebas funcionales del sistema a partir de requisitos funcionales: una transformación de requisitos funcionales a un modelo intermedio de caminos que enriquecerán el modelo de requisitos funcionales, una transformación de un modelo de requisitos funcionales enriquecido con caminos a un modelo de escenarios de prueba, una transformación de un modelo de requisitos funcionales enriquecidos a un modelo de valores de prueba y una transformación de un modelo de requisitos funcionales enriquecido, un modelo de escenarios de prueba y un modelo de valores de prueba a un modelo de casos de prueba.

2. Transformación de requisitos funcionales a representación intermedia de caminos

En la sección anterior se ha expuesto un primer paso en el proceso de generación de pruebas funcionales del sistema a partir de requisitos funcionales, consistente en definir todos los posibles caminos, ramas o alternativas que se pueden construir a partir del comportamiento definido en un requisito funcional.

Estos caminos no introducen información nueva que no esté ya definida en los requisitos funcionales conforme al metamodelo de requisitos funcionales definido en el capítulo IV. En su lugar estos caminos son una manera adicional y redundante de definir el comportamiento de un requisito funcional. Si estos caminos no están presentes, un requisito funcional sigue teniendo la misma información.

Sin embargo, como se ha justificado al principio de este capítulo, expresar el comportamiento de un requisito funcional como caminos, en lugar de pasos y órdenes de transición que construyen una máquina de estados, facilita la especificación de las transformaciones y evita tener que repetir partes comunes a las mismas.

Esta sección especifica la primera transformación (transformación T1 en la figura 6.3), la cual toma como artefactos de origen los requisitos vistos en el metamodelo de requisitos funcionales ya definido en detalle en un capítulo anterior y genera artefactos de una ampliación de la sintaxis abstracta de requisitos funcionales, para representar los distintos caminos de ejecución de un requisito funcional.

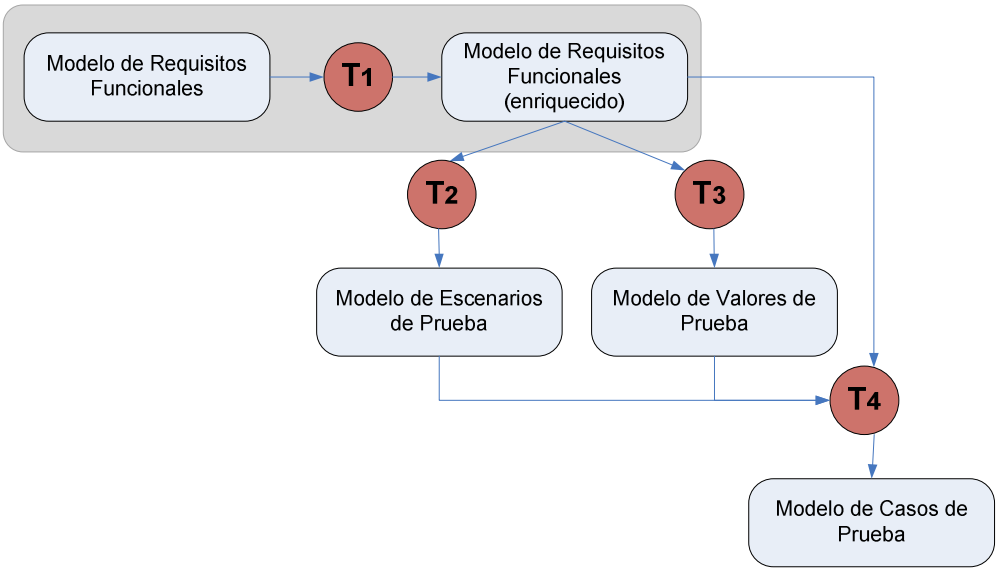


Figura 6.3. Transformación de requisitos funcionales a representación intermedia de caminos.

Los artefactos para la especificación de caminos se muestran en la figura 6.4 y se describen a continuación.

Como se puede ver en la figura 6.4, no es necesario definir un nuevo metamodelo, simplemente añadiendo un par de elementos que enlacen con los pasos y órdenes de ejecución de un requisito funcional es suficiente. Esta ampliación especifica dos nuevos elementos, el elemento *Camino* y el elemento *Nodo.* El elemento *Camino* simplemente es una secuencia de nodos, mientras que el elemento *Nodo* hace referencia a un paso, al paso anterior (salvo que sea el primero) y al paso siguiente (salvo que sea el último).

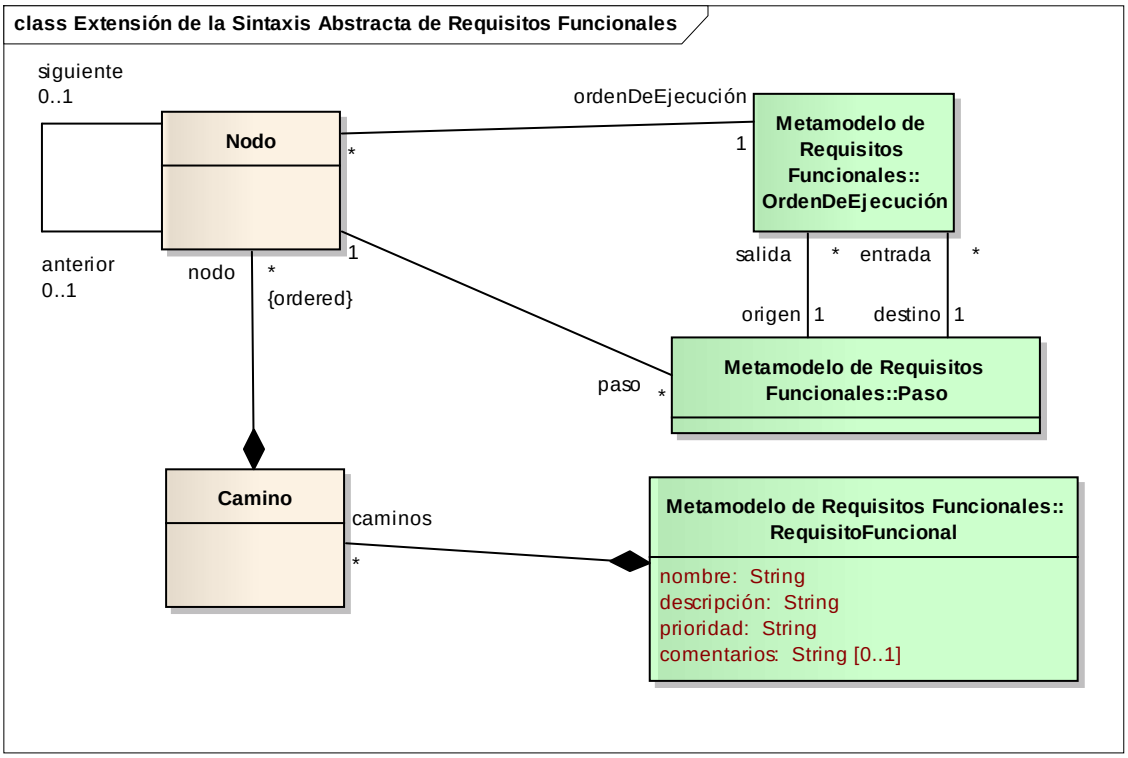


Figura 6.4. Extensión del metamodelo de requisitos funcionales.

A continuación, en las siguientes secciones, se especifica la cabecera de la transformación, así como los mapeos y funciones auxiliares necesarias.

2.1. Especificación de la transformación

La especificación de esta transformación se muestra en la tabla 6.1. Las claves de los elementos del metamodelo de requisitos funcionales enriquecido se han incluido en la transformación.

Tabla 6.1. Transformación de un requisito funcional a la representación intermedia de caminos.

```
transformation RequisitosFuncionales_a_ModeloIntermedioDeCaminos(
        inout mrf: MetamodeloDeRequisitosFuncionales)  {
  main() {
        key Camino(nodo);
        key Nodo(paso, siguiente);
        mrf.objectsOfType(RequisiToFuncional)->map RequisitoFuncional_a_Camino();
  }
}
```

Como se puede apreciar en la tabla 6.1, el mismo modelo de requisitos funcionales es el modelo de entrada y el modelo de salida, ya que esta transformación enriquece un modelo de requisitos funcionales añadiéndole información adicional. Por ello, el modelo de entrada (cabecera *transformation*) está declarado como un parámetro de entrada y salida (*inout*). Dentro del cuerpo de la transformación (método *main*), se realiza una llamada al mapeo RequisitoFuncional_a_Camino (descrito justo a continuación) sobre todos los requisitos funcionales del modelo.

2.2. Especificación del mapeo de requisito funcional a camino

Este mapeo (tabla 6.2) se aplica sobre cada requisito funcional y su misión es definir el punto de entrada de la función auxiliar, en este caso un helper, que especifica cómo generar los caminos de manera recursiva.

Tabla 6.2. Mapro de requisito funcional a camino.

```
mapping RequisitoFuncional::RequisitoFuncional_a_Camino()
{
        var pth : Camino;
        var codel : Set(OrdenDeEjecucion) {};
        self.generarCaminos(pth, self.paso->at(1), codel, mrf.caminos, 1, false);
}
```

La función auxiliar *generarCaminos* se utiliza para generar los caminos a partir de los pasos de un requisito funcional. Esta función se especifica en la siguiente sección.

2.3. Especificación de la función auxiliar generarCaminos

La especificación del proceso para calcular caminos a partir del comportamiento de un requisito funcional se especifica en la tabla 6.3.

Tabla 6.3. Función auxiliar para la especificación de caminos.

```
helper RequisitoFuncional::generarCaminos(pth: Camino, p: Paso,   codel: Set(OrdenDeEjecucion),
                                                pl: List(Camino), nb: Integer, fin: Boolean) {
        var cpss : OrderedSet(OrdenDeEjecucion);
        cpss := p.salida;
        if (fin) {   pl->add(pth);
                return;
        }
        var nb_c: Integer := 0;
        var b: Boolean := false;
        pth.nodos->forEach(n) {
                if (n.paso == p) {
                        nb_c += 1;
                        if (nb == nb_c) {     b := true;
                                              break;
                        }
                }
        }
        if (b) {     nb += 1;
                var i: Integer := pth.nodos->size();
                var Nodo n;
                var OrdenDeEjecucion eo := null;
                while (i >= 1) {
                        n := pth.nodos->at(i);
                        if ((n.paso.salidas->size() > 1) &&
                                (i = pth.nodos->size()) {
                                        eo := pth-nodos
                                    ->at(i).paso.DevuelveOrdenDeEjecucion(p);
                                } else {
                                  eo := pth-nodos->
                            at(i).paso.DevuelveOrdenDeEjecucion(pth-nodos->at(i+1).paso);
                                }
                                codel->asList()->add(eo);
                                if (n.paso.ContarOrdenDeEjecucionEn(codel) <
                                        n.pasos.salida->size()) { break;   }
                        }
                        i -= 1;
                }
        }
        var ode: OrdenDeEjecucion;
        var pth_c : Camino;
        var n_aux : Nodo;
        if (cpss->isEmpty()) {
                pth_c :=pth;
                n_aux := Node { paso := p; ordenDeEjecucion :=ode };
                if (p_c.nodos->size() > 1) { p_c.nodos->at(p_c.nodos->size()).siguiente := n_aux;  }
                p_c.nodos->asList()->add(n_aux);
                generarCaminos(p_c, p, codel, pl, nb, true);
        }
        cpss-> forEach(sp)  {
                p.salidas-> forEach(eo_aux | eo_aux.destino = sp.destino) {
                        if (eo_aux.destino = sp.destino) {         ode := eo_aux;
                                                                   break;
                        }
                }
```

```
                        if (p.EstaEn(sp.destino, codel) = false) {
                                p_c := pth.clone();
                                n_aux := Node { paso := p; ordenDeEjecucion :=ode };
                                if (p_c.nodos->size() > 1) {
                                        p_c.nodos->at(p_c.nodos->size()).siguiente := n_aux;
                                }
                                p_c.nodos->asList()->add(n_aux);
                                generarCaminos(p_c, sp.destino, codel.clone(), pl, nb, false);
                        }
                }
        }
```

En esta función auxiliar, el número de escenarios de prueba a construir (o caminos) y la manera de combinar los pasos de un requisito funcional viene dictada por el criterio de cobertura / suficiencia a la hora de combinar los pasos para formar escenarios, especialmente el tratamiento de bucles, es decir, de pasos que alteran la secuencia redirigiendo a otros pasos concretos.

En la especificación de esta transformación, como se puede ver en el código QVT, se ha elegido un criterio que obtiene todos los caminos para ninguna o una repetición de cada posible bucle, pero sería posible implementar otro criterio distinto si así se decidiera.

Otro aspecto relevante de la función auxiliar que genera los caminos es el uso de la clonación. El mecanismo de clonación ya lo proporciona QVT por defecto en todos los elementos de un metamodelo, por lo que no es necesario definirlo explícitamente para los elementos vistos en los capítulos anteriores. En esta transformación concreta, el clonado se realiza cuando un paso de un camino tiene más de una alternativa para continuar su ejecución. En ese caso, el camino que contiene dicho paso se clona tantas veces como alternativas existan para poder explorar todas esas alternativas.

Además, esta función utiliza otras tres funciones auxiliares. En concreto, se utiliza la función DevuelveOrdenDeEjecucion para recuperar el orden de ejecución de un paso, la función ContarOrdenDeEjecucion para saber cuántos órdenes de ejecución tiene un paso, y la función EstaEn para saber si un paso está en el orden de ejecución de otro paso. Estas tres funciones auxiliares se especifican en la siguiente sección.

2.4. Especificación de otras funciones auxiliares

En esta sección, en la tabla 6.4, se especifican otras funciones auxiliares que se han utilizado como parte de la transformación.

Tabla 6.4. Otras funciones auxiliares.

```
query Paso::DevuelveOrdenDeEjecucion(paso: Paso): OrdenDeEjecucion {
                    return self.salida->select(oe | oe.destino = paso)->at(1);
  }
  query Paso::ContarOrdenDeEjecucionEn(codel: Set(OrdenDeEjecucion)) : Integer {
          return codel->select(oe | oe.destino = self) -> size();
  }
  query Paso::EstaEn(sp: Paso, codel: Set(OrdenDeejecucion)): Boolean {
          return codel->select(oe | oe.origen=self && oe.destino = sp) -> size > 1;
  }
```

Con estas funciones auxiliares ya queda completamente especificada la transformación que enriquece un modelo de requisitos funcionales añadiendo los caminos a cada requisito funcional. Teniendo esta información adicional, se procede a especificar las transformaciones para la generación de los artefactos de prueba en las siguientes secciones.

2.5. Ejemplo de aplicación de la transformación

Para ayudar a comprender la misión de la transformación especificada en esta sección dentro del proceso de generación de pruebas funcionales del sistema se muestra un ejemplo de los resultados de su ejecución.

Tabla 6.5. Ejemplo de posibles caminos.

Camino	Descripción
1	1. El usuario solicita dar de alta un nuevo elemento en el sistema. 2. El sistema solicita la información del elemento. 3. El usuario introduce la información del elemento. 4. El sistema da de alta al elemento.
2	1. El usuario solicita dar de alta un nuevo elemento en el sistema. 2. Si se ha superado el límite de elementos del sistema, entonces se muestra un mensaje de error y el resultado es finalizar la ejecución
3	1. El usuario solicita dar de alta un nuevo elemento en el sistema. 2. El sistema solicita la información del elemento. 3. El usuario introduce la información del elemento. 4. Si la información es incorrecta, entonces se muestra un mensaje de error y el resultado es ir al paso 2. 5. El sistema solicita la información del elemento. 6. El usuario introduce la información del elemento. 7. El sistema da de alta al elemento.
4	1. El usuario solicita dar de alta un nuevo elemento en el sistema. 2. El sistema solicita la información del elemento. 3. El usuario introduce la información del elemento. 4. Si la información es incorrecta, entonces se muestra un mensaje de error y el resultado es ir al paso 2. 5. Si se ha superado el límite de elementos del sistema, entonces se muestra un mensaje de error y el resultado es finalizar la ejecución

Al aplicar esta transformación sobre el requisito funcional de alta de un elemento visto en los capítulos anteriores (por ejemplo, en el capítulo V sección 2), se obtienen los 4 caminos mostrados en la tabla 6.5.

A partir de aquí, las transformaciones para obtener escenarios de prueba y valores de prueba, utilizarán los caminos generados mediante la transformación vista en esta sección. Es decir, a partir de aquí, en este capítulo, siempre que se mencione un modelo de requisitos funcionales, dicho modelo ya estará enriquecido con los caminos definidos en esta sección.

3. Transformación de requisitos funcionales a escenarios de prueba

La transformación definida en esta sección (parte sombreada en la figura 6.5) tiene como objetivo la obtención de los escenarios de prueba. Se utilizarán los requisitos funcionales y los artefactos auxiliares (caminos y nodos) vistos en la sección anterior para obtener un escenario de prueba por cada camino identificado.

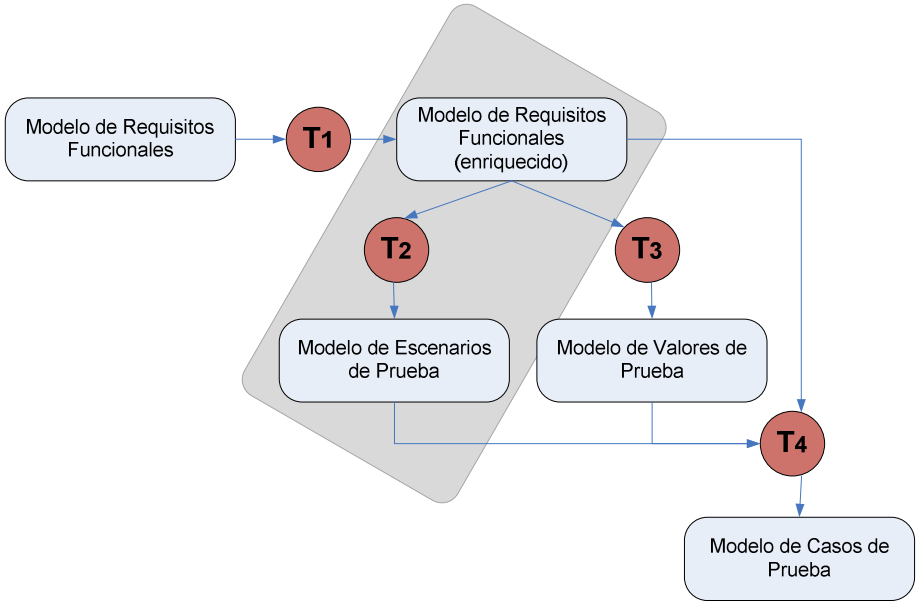


Figura 6.5. Transformación de requisitos funcionales a escenarios de prueba.

De manera resumida, el proceso de transformación consiste en generar un escenario de prueba con todos los elementos menos los pasos del escenario de prueba. Después, ese escenario se copiará tantas veces como caminos tenga el requisito funcional y, a partir de cada uno de dichos caminos, se generarán los pasos del escenario de prueba de cada escenario.

La especificación de la transformación en lenguaje QVT, así como los mapeos y funciones auxiliares que la componen se exponen en las siguientes secciones.

3.1. Especificación de la transformación

La especificación de la transformación de requisitos funcionales, complementados con la información sobre caminos, a escenarios de prueba, se muestra en la tabla 6.6. Las claves de los elementos del grupo lógico de escenarios de prueba se especifican en la transformación.

Tabla 6.6. Transformación de requisitos funcionales a escenarios de prueba.

```
transformation RequisitosFuncionales_a_EscenariosDePrueba(in mrf: MetamodeloDeRequisitosFuncionales,
   out mep: MetamodeloDeEscenariosDePrueba) {
  main () {
    key EscenarioDePrueba(acciones);
    key AcciónDelEscenarioDePrueba(cuerpo);
     key EstadoDelSistema(estado);
    mrf.objectsOfType(Subsistema) ->  map Subsistema_a_PaqueteDeEscenariosDePrueba();
    mrf.objectsOfType(ActorDelSistema)  ->map ActorDelSistema_a_ActorDePrueba();
mrf.objectsOfType(RequisitoFuncional)->map RequisitoFuncional_a_EscenarioDePrueba(mic.objectsOfType(Path));
    mt.objectsOfype(EscenarioDePrueba) -> select (v|v.pasos->isEmpty()) {  mte.removeelement(v)  };
  }
}
```

Esta transformación toma como entrada un modelo conforme al metamodelo de requisitos funcionales visto en este trabajo de tesis y devuelve como salida un modelo conforme al metamodelo del grupo lógico de escenarios de prueba también visto en este trabajo de tesis. Esta transformación se compone de tres mapeos que se aplican sobre los elementos del modelo de requisitos funcionales para obtener los artefactos de escenarios de prueba, por ejemplo, el mapeo *ActorDelSistema_a_ActorDePrueba*, se aplica sobre los actores del sistema presentes en el modelo de requisitos funcionales de entrada, para generar los actores de prueba del modelo de escenarios de prueba de salida.

A continuación, en las siguientes secciones se especifican los mapeos referenciados en la transformación de la tabla 6.6.

3.2. Especificación del mapeo de subsistema a paquete de escenario de prueba

Se especifica en la tabla 6.7 el mapeo que permite crear los paquetes donde de almacenarán los escenarios de prueba a partir de los subsistemas del modelo de requisitos funcionales.

Tabla 6.7. Mapeo de subsistemas a paquetes de escenarios de prueba.

```
mapping Subsistema::Subsistema_a_PaqueteDeEscenariosDePrueba()
        : PaqueteDeEscenariosdePrueba {
                nombre = self.nombre;
                descripción := self.descripción;
                origen := self;
```

```
        }
```

Los paquetes creados tendrán el mismo nombre y la misma descripción que los subsistemas.

3.3. Especificación del mapeo de actor del sistema a actor de prueba

Este mapeo especifica cómo crear los actores de prueba a partir de los actores del sistema. Dicho mapeo se especifica en la tabla 6.8.

Tabla 6.8. Mapeo de actor del sistema a actor de prueba.

```
mapping ActorDelSistema::ActorDelSistema_a_ActorDePrueba(): ActorDePrueba {
        nombre := self.nombre;
        descripcion := self.descripcion;
        origen := self.ActorDelSistema;
}
```

El nombre y la descripción del actor de prueba generado como resultado de este mapeo, coinciden con el nombre y la descripción del actor del sistema.

3.4. Especificación del mapeo de requisitos funcionales a escenarios de prueba

Este mapeo especifica cómo crear un prototipo de escenario de prueba a partir de un requisito funcional. Este prototipo de escenario de prueba tiene como misión servir de patrón o molde para crear tantos escenarios de prueba como caminos distintos se hayan generado a partir de los pasos del requisito funcional con la transformación vista en la sección anterior.

Por ejemplo, a partir del ejemplo de la sección 2.5, se generaría un único prototipo escenario de prueba con los mismos actores, descripción, subsistema, etc. A partir de este prototipo, aplicando las transformaciones de las próximas secciones, se generan cuatro escenarios de prueba.

El mapeo para crear un prototipo de escenario de prueba a partir de un requisito funcional se especifica en la tabla 6.9.

Tabla 6.9. Mapeo de requisito funcional a escenario de prueba.

```
mapping RequisitoFuncional::RequisitoFuncional_a_EscenarioDePrueba(pths: Set(Pasos)):
            EscenarioDePrueba
{
            nombre := self.nombre;
            descripcion := "Escenario de prueba para el requisito funcional " + self.descripcion;
            comentarios := self.comentarios;
            estadoInicial := self.precondicionRequisito;
            origen := self;
            paquete:= mep.objectsOfType(paqueteDeEscenariosDePrueba)->
                        select(p | p.origen = self.subsistema)->at(1);
}
when {
            Set peps := self.pasos -> map Paso2PasoDelEscenarioDePrueba ();
            result.generarEscenariosDePrueba(pths, peps);
            self.crearEstadoFinal();
}
```

Posteriormente a la realización del mapeo de un requisito funcional a un prototipo escenario de prueba (fragmento *when* del mapeo de la tabla 6.9, al final del mismo) se especifica una referencia a otro mapeo (*generarEscenariosDePrueba*) para transformar los pasos del requisito funcional a pasos del escenario de prueba. Este mapeo se especifica en la siguiente sección. A continuación, dentro de la cláusula *when*, se especifica una referencia a una función auxiliar que genera tantos escenarios como sea preciso a partir del requisito funcional. Por último, se ejecuta el cuerpo principal del mapeo y se inicializan los atributos del escenario de prueba con los atributos del requisito funcional.

Además, la asignación de las poscondiciones, y a diferencia de las precondiciones, solo se realiza para aquel escenario que contiene el escenario principal del requisito funcional. Por ese motivo la cláusula *when* de la transformación termina aplicando un helper, el cual, una vez generados todos los escenarios de prueba, localiza aquel escenario que contiene el camino principal del requisito funcional y le añade las poscondiciones. La especificación de este helper se incluye en la sección 3.6.

3.5. Especificación del mapeo de paso de requisito funcional a paso del escenario de prueba

Este mapeo especifica cómo crear el conjunto de pasos del escenario de prueba que corresponden a los pasos del requisito funcional. Los pasos del escenario de prueba pueden ser, o bien verificaciones de prueba, o bien acciones del escenario de prueba, por lo que el

mapeo deberá crear uno u otro, según el paso principal sea realizado por un actor del sistema, o por el propio sistema. Este mapeo se especifica en la tabla 6.10.

Tabla 6.10. Mapeo entre un paso del requisito funcional y un paso del escenario de pruebas.

```
mapping Paso::Paso2PasoDelEscenarioDePrueba() : result:PasoDelEscenarioDePrueba {
        if (self.ejecutor->isEmpty()) {  result := object VerificacionDelEscenarioDePrueba { };
        } else {
                result := object AccionDelEscenarioDePrueba {
                        ejecutor := getActorDePrueba(self.ejecutor.nombre);
                        ejecutor.interaccion := result;
                }
        }
        cuerpo := self.accion;
        pasoDeOrige := self;
}
query getActorDePrueba(name: String):ActorDePrueba {
        return mep.objectOfType(ActorDePrueba )-> collect(i|i.name = name)-> at(1);
}
```

El criterio para determinar cuál de los elementos utilizar, viene determinado por el ejecutor del paso del requisito funcional. Es decir, si el paso es realizado por el propio sistema, entonces el mapeo creará una verificación del escenario de prueba, pero si el paso es realizado por una entidad externa, entonces el mapeo creará una acción del escenario de prueba. Este mapeo utiliza la función *getActorDePrueba*, la cual permite localizar un objeto actor a partir de su nombre.

3.6. Funciones auxiliares para la generación de escenarios de prueba

El último paso para completar la transformación de un requisito funcional a un conjunto de escenarios de prueba es definir cómo generar tantos escenarios de prueba como sean necesarios a partir del requisito funcional. Cada escenario será el comportamiento a verificar a la hora de probar el sistema.

Tabla 6.11. Función auxiliar para el cálculo de escenarios.

```
helper TestScenario::generarEscenariosDePrueba(pths: Set(Path), peps: Set(Paso)) {
        var ep_c : EscenarioDePrueba
        pths->forEach(pth) {
                ep_c := self.clone();
                pth.nodos->forEach(n) {
                        ep_c.pasoDelEscenarioDePrueba += n.buscarPasoDelEscenarioDePrueba(peps);
                }
        }
}

query Nodo::buscarPasoDelEscenariodePrueba(peps: Set(PasoDelEscenarioDePrueba))
 : PasoDelEscenarioDePrueba {
        return peps->select(PasoDelEscenarioDePrueba: pep | pep.cuerpo = n.paso.accion);
}

helper RequisitoFuncional::crearEstadoFinal() {
        var me: EscenarioDePrueba;
        me = mep.objectsOfType(EscenarioDePrueba) -> select(e |
                        e.pasoDelEscenarioDePrueba -> forAll(p | p.pasoOrigen.pasoPrincipal = true) );
        me.estadoFinal := self.poscondicionRequisito;
}
```

El número de escenarios de prueba a construir depende de los caminos generados en la representación intermedia obtenida a partir de la transformación vista en la sección anterior. Se construirán tantos escenarios de prueba como caminos haya creado la transformación de la sección anterior. Así, en el ejemplo de la sección 2.5, se generarán cuatro escenarios de prueba, uno a partir de cada uno de los caminos.

Cada uno de esos escenarios obtenidos con esta transformación contendrá el comportamiento definido en uno de esos caminos. En la tabla 6.11 se especifica esta función auxiliar.

Lo primero que se puede ver, en la propia cabecera de la función auxiliar es que no tiene valor de retorno, ya que no crea ningún elemento nuevo, sino que se limita a clonar un escenario de prueba y a completarlo a partir de elementos ya existentes (en concreto el conjunto de pasos del metamodelo intermedio de pasos, el conjunto de pasos del escenario de prueba y el escenario de prueba). En el primer bucle de la función auxiliar se recorren todos los caminos y para cada uno de ellos (en el segundo bucle) se clona el escenario de prueba, se localizan los pasos del escenarios de prueba correspondientes a cada uno de los pasos y se añaden a dicho escenario de prueba en el mismo orden, con lo que se obtienen tantos escenarios de prueba como caminos se hubieran identificado a partir del requisito funcional y cada escenario de prueba tiene un comportamiento diferente, que coincide con uno de dichos caminos.

Además, se ha visto en un mapeo anterior que las poscondiciones se añaden como parte del estado final de un escenario de prueba solo si dicho escenario de prueba contiene el camino principal. El helper que realiza dicha operación (*crearEstadoFinal*) se muestra también en la tabla 6.11

En la siguiente sección se especifican las transformaciones para la generación de variable operacionales, restricciones y combinaciones a partir de los requisitos funcionales y la representación intermedia de caminos vista en las secciones anteriores.

4. Transformación de requisitos funcionales a valores de prueba

La transformación definida en esta sección (parte sombreada de la figura 6.6) tiene como objetivo la obtención de los artefactos definidos en el metamodelo de valores de prueba.

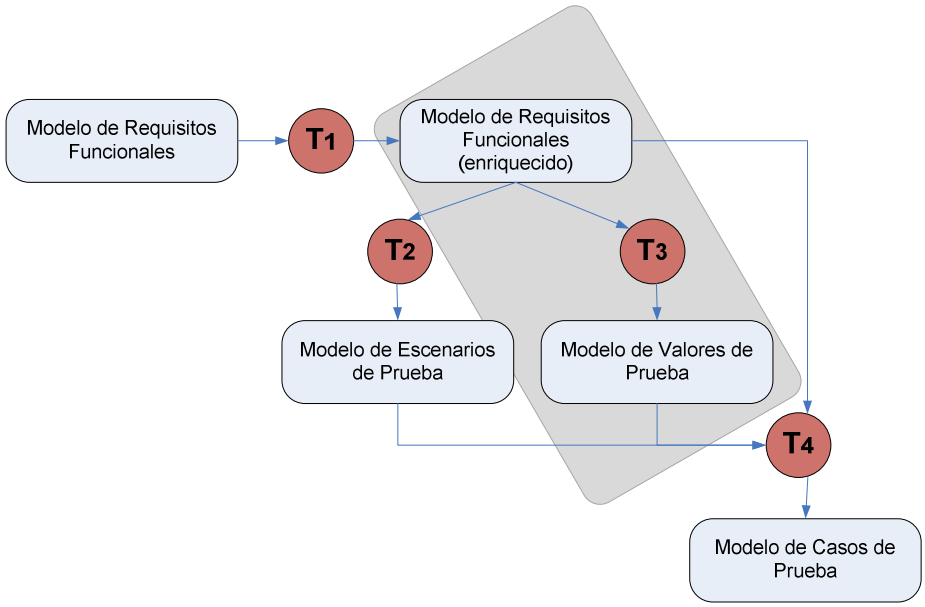


Figura 6.6. Transformación de Requisitos Funcionales a Valores de Prueba

Se han definido en el metamodelo de valores de prueba tres tipos distintos de variables operacionales: de selección, de actor del sistema y de información (que, a su vez, puede ser de entrada al sistema o de salida). En este trabajo de tesis no se ha definido una heurística que, a partir de la información contenida en los requisitos funcionales sea capaz de clasificar las variables según los tipos vistos en el capítulo anterior. Por este motivo, no se crean tipos específicos de variables sino que, en su lugar, se especifica una transformación para construir variables operacionales sin tipo que, posteriormente puedan ser refinadas en cada uno de los

tipos recogidos en el metamodelo de valores de prueba. El desarrollo de una manera de clasificar las variables operacionales queda propuesto para una continuación de este trabajo de tesis.

4.1. Especificación de la transformación

La transformación de requisitos funcionales a valores de prueba se especifica a continuación, en la tabla 6.12 las claves de los elementos del grupo lógico de valores de prueba se especifican en la transformación.

Tabla 6.12. Transformación de requisitos funcionales a valores de prueba.

```
transformation RequisitosFuncionales_a_ValoresDePrueba(in mrf: MetamodeloDeRequisitosFuncionales,
        out mvp: MetamodeloDeValoresDePrueba) {
        main () {
                key VariableOperacional(requisitoFuncional, Origen);
                key ParticiónDeDatos(rangoDeValores);
                key AtributoParaPrueba(attribute);
                key CombinaciónDeParticiones(valores);
                key RestricciónDeCombinaciónDeParticiones(evaluadas);
                mrf.objectsOfType(Subsistema) ->
                        map Subsistema_a_PaqueteDeCombinacionesDeInstancias();
                mrf.objectsOfType(Paso)  -> map RequisitoFuncional_a_VariableOperacional();
                mrf.objectsOfType(RequisitoFuncional)
                        -> map RequisitoFuncional_a_Instancia(mvp.objectsOfType(VariableOperacional));
                mrf.objectsOfType(RequisitoFuncional) ->
                        map RequisitoFuncional_a_CombinacionDeInstancias();
        }
```

Esta transformación se especifica en base a cuatro mapeos que generarán los paquetes de combinaciones de instancias, las variables operacionales, y a partir e ahí sus particiones, las instancias y las combinaciones de instancias. Dichos mapeos se especifican en la siguiente sección.

4.2. Especificación del mapeo de subsistema a paquete de combinaciones de instancias

Se especifica en la tabla 6.13 el mapeo que permite crear los paquetes donde se almacenarán las combinaciones de instancias a partir de los subsistemas del modelo de requisitos funcionales.

Tabla 6.13. Mapeo de subsistema a paquete de combinaciones de instancias.

```
mapping Subsistema::Subsistema_a_PaqueteDeCombinacionesDeInstancias()
        : PaqueteDeCobinacionesDeInstancia {
                nombre = self.nombre;
                descripción := self.descripción;
                origen := self;
}
```

Un paquete tendrá el mismo nombre y descripción que el subsistema.

4.3. Especificación del mapeo de requisito funcional a variable operacional

Se especifica en la tabla 6.14 el mapeo que permite crear las variables operacionales a partir de los pasos de un requisito funcional que tienen más de una alternativa.

Tabla 6.14. Mapeo de requisito funcional a variable operacional.

```
mapping Paso::RequisitoFuncional_a_VariableOperacional() : VariableOperacional
when { self.salida.size() > 1 }
where {   partición:= self.destino -> map ordendeEjecución_a_Partición (result);  }
{
        object result:VariableOperacional {
                nombre := self.requisitoFuncional.nombre;
                descripción := "Variable operacional para el requisito funcional "
                        + self.requisitoFuncional.nombre;
                pasoOrigen := self.requisitoFuncional;
        };
}
```

Las variables operacionales se construyen a partir de aquellos puntos en el requisito funcional donde se puedan realizar distintos comportamientos. Para ello se identificarán las variables operacionales a partir de aquellos pasos del requisito funcional que tengan más de un orden de ejecución de salida. Además, se especifica la participación de un mapeo adicional para construir las particiones de una variable operacional a partir del orden de ejecución, una vez terminada la construcción de variables operacionales.

4.4. Especificación del mapeo de orden de ejecución a partición de variable operacional

Se especifica en la tabla 6.15 el mapeo correspondiente para la creación de particiones de variables operacionales a partir de órdenes de ejecución.

Tabla 6.15. Mapeo de Orden de Ejecución a Partición.

```
mapping OrdendeEjecución::ordendeEjecución_a_Partición(in vo: VariableOperacional): Partición
{
        nombre := "Particion: " +vo.nombre;
        subdominio := vo;
        ordenDeEjecucionOrigen := self;
        restriccion := self.puntoDeExcepcion;
}
```

Las particiones toman su restricción del punto de excepción del orden de ejecución.

4.5. Especificación del mapeo de requisito funcional a instancias y restricciones de bucle

Para construir las instancias de las variables operacionales y crear restricciones, es necesario especificar un mapeo que recorra el comportamiento de un requisito funcional para calcular las instancias existentes. Como ayuda, se utilizarán los caminos construidos por la primera transformación definida en este capítulo. Este mapeo se muestra en la tabla 6.16.

Tabla 6.16. Mapeo de requisito funcional a instancia.

```
mapping RequisitoFuncional::RequisitoFuncional_a_Instancia(Set(VariableOperacional) vos) {
        self.generarInstancias(self.objectsOfType(Camino), vos);
}
helper RequisitoFuncional::GeneraInstancias (cs: Set(Camino),  vos: Set(VariableOperacional)) {
        var cvop: Integer;
        var cvopMax: Integer;
        vos -> forEach(v) {
                cvopMax := -1;
                cs -> forEach(c) {
                        cvop := 0;
                        c.nodo-> forEach(n) {
                                if (n.paso = v.pasoOrigen) {    cvop := cvop +1;     }
                        }
                        if (cvop > cvopMax) { cvopMax = cvop;               }
                }
                var i: Integer := 1;
                while (i <= cvopMax) {
                        var inst: Instancia;
                        inst := object: Intancia {
                                nombre := v.nombre + indIns;
                                variableOperacional := self;
                        }
                        v.instancias += inst;
                        mvp.objectsOfType(Instancia)->asList()->add(inst);
                        i += 1;
                }
        }
}
```

Las instancias obtenidas no tienen un valor asignado, ya que pueden aparecer en distintas combinaciones tomando distintos valores. El valor que toma cada instancia en una combinación concreta se le asocia en el momento de generar las combinaciones de instancias, como se verá a continuación.

4.6. Especificación del mapeo para la construcción de combinaciones de instancias

Una vez definido cómo crear el conjunto de variables, con sus particiones y sus instancias, el último paso es unirlo todo para obtener combinaciones válidas de instancias, con los Quantums correspondientes a cada combinación. El artefacto Quantum se definió en el metamodelo de valores de prueba y representa la asignación de una instancia de una variable operacional a un valor perteneciente a una de las particiones de la variable operacional.

La generación de combinaciones de instancias se realiza con el mapeo mostrado en la siguiente tabla. Una combinación válida cumplirá todas las restricciones identificadas. La especificación del mapeo correspondiente se muestra en la tabla 6.17.

Tabla 6.17. Mapeo de requisito funcional a combinación de instancias.

```
mapping RequisitoFuncional::RequisitoFuncional_a_CombinacionDeInstancias() {
        self.caminos -> generarCombinacionDeInstancias(mvo.objectsOfType(VariableOperacional), self);
}
helper Camino::generarCombinacionDeInstancias(in cvo: Set(VariableOperacional), in rf: RequisitoFuncional)
        : result: combinacionDeInstancias
{
        paquete:= mvo.objectsOfType(paqueteDeCombinacionesDeInstancia)->
                        select(p | p.origen = rf.subsistema)->at(1);
        var ic:Dict(VariableOperacional, Integer);
        cvo->forEach(vo) { ic->put(vo, 0); };
        var nod:List(Nodo):
        self.nodos->select(n | n.paso.salida->size()> 1) { nod->add(n);};
        var ov: VariableOperacional;
        var oe: OrdenDeEjecucion;
        var dp: ParticionDeDatos;
        var q: Quantum;
        nod->forEach(n) {
                ov := ovl->select(v| v.paso = n.paso)->at(1);
                q.particionDeDatos := ov.instancias->at(ic->get(ov));
                ic->put(ov, ic->get(ov)+1);
                oe := ov.paso.salidas ->
                   select(eo | eo.origen = n.paso && eo.destino = n.siquiente.paso) -> at(1);
                dp := ov.particion->selet(pd | do.ordenDeEjecucionOrigen = oe) -> at(1);
                q.particionDeDatos := dp;
                ic.quantums->asList()->add(q);
                result.quantums->asList()->add(q);
        }
        cvo -> forEach(vo) {
                var ind := ic.get(vo);
                while(ind < vo.instancias->size()) {
                        var q: Quantum {
                                particionDeDatos := vo.particionSinValor();
                                instancia = vo.instancias->at(ind);
                        };
                        result.quantums->asList()->add(q);
                }
                ind += 1;
        }
}
query VariableOperacional::particionSinValor(): ParticionDeDatos {
        result.nombre = "*";
        result.descripcion = "sin valor";
}
```

A cada instancia se le añade un Quantum con el valor que toma dicha instancia en esa combinación concreta. En el caso de que no tome ningún valor, se utiliza una partición especial que indica que no se utiliza dicha instancia en esa combinación en concreto.

Una vez definido este mapeo, ya se cuenta con un modelo de valores de prueba. El siguiente paso será proponer una transformación para aunar la información de los escenarios y valores de prueba en un único modelo de casos de prueba.

5. Transformación de escenarios de prueba y valores de prueba a casos de prueba

La transformación definida en esta sección (sombreada en la figura 6.7) tiene como objetivo unir en un único artefacto de prueba los escenarios y las combinaciones de variables operacionales obtenidas mediante las transformaciones definidas en las secciones anteriores.

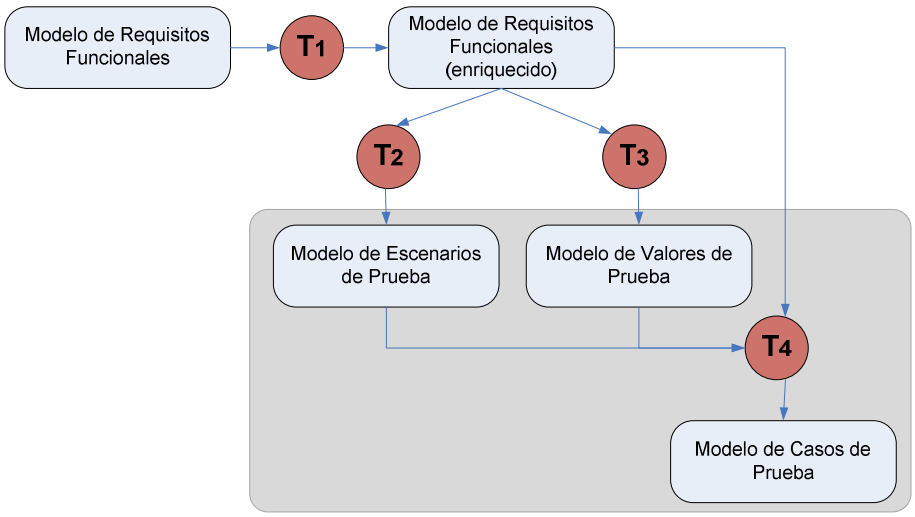


Figura 6.7. Transformación de escenarios de prueba y valores de prueba a casos de prueba.

Como se ha visto en el capítulo anterior, los artefactos del metamodelo de casos de prueba funcionales del sistema derivan de los artefactos de escenarios de prueba. Por este motivo las claves de los artefactos de casos de prueba son las mismas que las claves definidas en este capítulo para los escenarios de prueba.

5.1. Especificación de la transformación

El cuerpo principal de la transformación para crear casos de prueba se especifica a continuación, en la tabla 6.18.

Tabla 6.18. Transformación de escenarios de prueba y valores de prueba a casos de prueba.

```
transformation EscenariosdePruebaYValoresdePrueba_a_CasosDePrueba(
        in rfm: ModeloDeRequisitosFuncionales, in esm: ModeloDeEscenariosDePrueba,
        in vom: ModeloDeValoresDePrueba,    out cpm: ModeloDeCasosDePrueba)
{
        main() {
                rfm.objectsOfType(Subsistema) -> map Subsistema_a_ColecciónDeCasosDePrueba();
                esm.objectsOfType(EscenarioDePrueba) ->
                map EscenarioDePrueba_a_CasodePrueba(vom.objectsOfType(CombinacionDeInstancia),
                                vom.objectsOfType(VariableOperacional)          );
                esm.objectsOfType(ActorDePrueba) -> foreach(a) {
                        cpm.objectsOfType(ActorDePrueba)->asList()->add(a);
                }
        }
}
```

En las posteriores secciones se especifican los mapeos especificados en esta transformación.

5.2. Especificación del mapeo de subsistema a colección de casos de prueba

El objetivo de este mapeo es crear los elementos colección de casos de prueba, los cuales son los paquetes donde se almacenarán los casos de prueba. Este mapeo se especifica a continuación, en la tabla 6.19.

Tabla 6.19. Mapeo de subsistema a colección de casos de prueba.

```
mapping Subsistema::Subsistema_a_ColeccionDeCasosDePrueba(): ColeccionDeCasosDePrueba
{
        nombre := self.nombre;
        criterio := self.descripción;
        origen := self;
}
```

El nombre y la descripción de la colección de casos de prueba son los mismos que los del subsistema.

5.3. Especificación del mapeo de escenarios de prueba a casos de prueba

El objetivo de este mapeo es crear un conjunto de casos de prueba a partir de los escenarios de prueba. Este mapeo se especifica en la tabla 6.20.

Tabla 6.20. Mapeo de escenarios de prueba a casos de prueba.

```
mapping EscenarioDePrueba::EscenarioDePrueba_a_CasoDePrueba(in cci: Set(CombinacionDeInstancia)
        , in vos: set(VariableOperacional)
) : CasoDePrueba
where {
        pasoDePrueba := self.pasoDelEscenarioDePrueba->
            map PasoDelEscenarioDePrueba_a_AcciónDePrueba(self);
}
{
        nombre := self.nombre;
        descripción := self.descripción;
        comentarios := self.comentarios;
        estadoInicial := self.estadoInicial;
        estadoFinal := self.estadoInicial;
        origen := self.origen;
        prioridad := self.origen.prioridad;
        colección := cpm.objectsOfType(ColeccioDeCasosDePrueba)->select(cp | cp.origen=self.origen)->at(1);
        cpm.objectsOfType(ColeccioDeCasosDePrueba)->select(cp | cp=self.origen.subsistema)->at(1).prueba +=
result;
        informacionDePrueba := GetCombinacionesDeInstancias(cci, vos);
}
```

Como se puede ver, este mapeo también especifica la asignación de los valores de prueba adecuados a la hora de crear el caso de prueba. Las funciones auxiliares utilizadas en la especificación de este mapeo se detallan en la sección de funciones auxiliares, más adelante.

5.4. Especificación del mapeo de paso del escenario de prueba a acción de prueba

Este mapeo se especifica como parte del mapeo de escenario de prueba a caso de prueba. Este mapeo especifica cómo construir acciones de prueba del caso de prueba y cómo asociar dicha acción de prueba con la variable operacional y la partición correspondiente. Este mapeo se especifica en la tabla 6.21.

Tabla 6.21. Mapeo del paso del escenario de prueba a paso de prueba.

```
mapping PasoDePrueba::PasoDePrueba_a_AcciónDePrueba(in e: EscenarioDePrueba):
        result: AcciónDePrueba
{

        if (self.isInstanceOf(VerificacionDelEscenarioDePrueba)) {
                result := object VerificaciondePrueba { };
        } else {
                result := object AccionDePrueba {
                        ejecutor := self.ejecutor;
                }
        }
        origen := self.origen;
```

```
        cuerpo := self.cuerpo;
};
```

Como se puede ver en la transformación, los pasos de prueba se construyen a partir de los casos del escenario de prueba, enlazándolos con las instancias de variables operacionales, si es que hay alguna, para ese paso en cuestión.

5.5. Especificación de otras funciones auxiliares

En las secciones anteriores (secciones de la 5.1 a la 5.4), se han definido los mapeos de la transformación de requisitos funcionales a casos de prueba. Para terminar, se especifican en la tabla 6.22 algunas funciones auxiliares utilizadas en los mapeos vistos en las secciones anteriores.

Tabla 6.22. Funciones auxiliares.

```
helper EscenarioDePrueba::GetCombinacionesDeInstancias(in cci: Set(CombinacionDeInstancia)
        in vos: set(VariableOperacional)) : result: List(CombinacionDeInstancias) {
        result := List(CombinacionDeInstancias) {};
        var i: Integer;
        var vo: VariableOperacional;
        var oe: OrdenDeEjecucion;
        var usados := List(Quantum) {};
        cci->forEach(ci) {    i := 0;
                              self.pasoDelEscenarioDePrueba->forEach(p) {
                                      i += 1;
                                      vo := p.VariableAsociadaA(vos);
                                      if (vo = null) {
                                              continue;
                                      }
                                      ic.quantums->forEach(q) {
                                              if (usados->contains(q))
                                                      continue;
                                              if(q.instancia.variableOperacional <> ov)
                                                      continue;
                                              if (np < (self.pasoDelEscenarioDePrueba->size())) {
                                                      p.pasoOriginal.salida->select(eo |
                                                        eo.origen: = p.pasoOrigen && eo.destino =
                                                                self.pasoDePrueba->at(i+1).pasoOrigen)
                                                        { oe = eo; };
                                                      if (q.dataPartition = vo.BuscarParticionCon(oe)) {
                                                              usados->add(q);
                                                              break;
                                                      }
                                              }
                                      }
                              }
                              if (usados->size() = ci.numeroInstanciasEnUso() ) {
                                      result += ci;
                                      break;
                              }
```

```
                        }
                        return result;
                }
}
query PasoDelEscenarioDePrueba::VariableAsociadaA(in vos: Set(VariableOperacional)): vo : VariableOperacional
{
                return vos -> select (vo | vo.pasoOrigen = self.pasoOrigen)->at(1);
}
query VariableOperacional::BuscarParticionCon(in OrdenDeEjecucion oe) : Particion {
                return self.particion->(select p | p.ordenDeEjecucionOrigen = oe)->at(1);
}
query CombinacionDeInstancias::NumeroInstanciasEnUso(): Integer {
                return self.quantums->select(q | q.particionDeDatos.rangoDeValores = "*") -> size();
}
```

El helper *GetCombinacionesDeInstancias* especifica cómo calcular todas las combinaciones de instancias válidas en función de las restricciones construidas. La query *variableAsociadaA* permite encontrar la variable operacional asociada a un paso del escenario de prueba. La query *buscarParticionCon* especifica cómo encontrar la partición de una variable operacional asociada a un orden de ejecución. La query *numeroDeInstanciasEnUso* especifica cómo calcular el número de instancias en uso.

6. Conclusiones

Durante este capítulo se han estudiado las transformaciones entre los metamodelos definidos en los dos capítulos anteriores. Estas transformaciones se especifican mediante el lenguaje QVT. Aunque existen prototipos de herramientas que permiten ejecutar código en QVT, no se aprecia que sea posible ni sencillo ejecutar estas transformaciones tal cual se muestran en este capítulo, por las limitaciones de las herramientas existentes. Sin embargo, como se verá en el siguiente capítulo, esta especificación de las transformaciones es fácilmente traducible a código ejecutable en un lenguaje de propósito general.

Capítulo VII. MDETest-Tool.

En capítulos anteriores se han definidos los metamodelos de requisitos funcionales y de pruebas funcionales del sistema y las transformaciones que permiten obtener modelos de prueba a partir de modelos de requisitos funcionales. Uno de los objetivos marcados en este trabajo de tesis es la posible automatización de dichas transformaciones. Este capítulo tiene como objetivo mostrar el desarrollo de una herramienta que permite aplicar las transformaciones de manera automática. Sin embargo, para definir una herramienta es necesario abordar las sintaxis concretas para los modelos implicados, por lo que, en primer lugar, este capítulo propondrá ejemplos de posibles sintaxis concretas, como paso previo al desarrollo de la herramienta.

A modo de recordatorio, en esta tesis se ha definido la sintaxis concreta como una notación o lenguaje, que también puede estar definido mediante un metamodelo, para la construcción de modelos de una manera agradable para un usuario humano y se ha definido la sintaxis abstracta como el conjunto de conceptos o información para un ámbito concreto. Por ejemplo, una sintaxis concreta para un requisito funcional podría ser un lenguaje gráfico para definir requisitos y una sintaxis abstracta sería el metamodelo visto en el capítulo 4 que define los conceptos que un requisito funcional debe tener, independientemente de cómo estén representados.

La organización de este capítulo se describe a continuación. En la primera sección se describen dos posibles ejemplos de sintaxis concreta para requisitos funcionales y se aborda el tratamiento de relaciones de inclusión y extensión. En la segunda sección se describen dos posible ejemplos de sintaxis concreta para casos de prueba. En la tercera sección se describe la implementación de la herramienta MDETest. En la última sección se exponen las conclusiones de este capítulo.

1. Ejemplos de sintaxis concretas para requisitos funcionales

En esta sección, a modo de ejemplo, se propondrán dos posibles sintaxis concretas. Una de ellas será una sintaxis gráfica construida a partir de los diagramas de actividades propuestos por UML [OMG-UML, 2011] y utilizada por NDT en la actualidad. Otra será una sintaxis textual tabular similar a las notaciones textuales utilizadas en las primeras versiones de NDT y en muchas de las propuestas analizadas en el capítulo II. Debido a que la mayoría de los requisitos funcionales se definen bien utilizando texto o bien utilizando un modelo gráfico, estos dos ejemplos (sintaxis textual tabular y sintaxis de diagramas de actividades) cubren la mayoría de representaciones existentes.

1.1. Sintaxis concreta a partir de diagramas de actividades

Los diagramas de actividades definidos por UML enfatizan la secuencia y condiciones para la coordinación de comportamientos [OMG-UML, 2011] y son utilizadas en la actualidad por técnicas de definición de requisitos funcionales, por ejemplo NDT descrita en el capítulo 3 para la definición del comportamiento de un requisito funcional.

UML define los distintos conceptos que pueden representarse en un diagrama de este tipo. Por tanto, para utilizar diagramas de actividades como una sintaxis concreta para requisitos funcionales será necesario realizar una selección de aquellos conceptos útiles para representar los conceptos definidos en el metamodelo de requisitos funcionales. Una posible selección se muestra en la tabla 7.1.

Tabla 7.1. Conceptos de UML para la definición de requisitos funcionales como diagramas de actividades.

Concepto de requisitos	Representado por
ActorDelSistema	*ActivityPartition*
OrdeDeEjecución	*DesicionNode* y *ControlFlow*
Paso	*Action, ActivityFinalNode* y *InitialNode*
RequisitoFuncional	*Activity*
Subsistema	*Package*

Tabla 7.2. Conceptos de los elementos del metamodelo de diagrama de actividades.

Elemento	Descripción
Action	Operación realizada por un actor o sistema.
Activity	Conjunto de elementos que conforman un diagrama de actividades.
ActivityFinalNode	Finalización de la ejecución del caso de uso.
ActivityPartition	Agrupación de las acciones realizadas por un mismo actor o por el sistema.
ControlFlow	Representación del flujo de ejecución y del orden de ejecución de los pasos de un caso de uso.
DesicionNode	Evaluación de la condición de un paso de la secuencia alternativa o de la secuencia errónea.
InitialNode	Comienzo de la ejecución del caso de uso.
Package	Agrupación de actividades.

En la tabla 7.2 se describen brevemente los conceptos seleccionados del metamodelo de diagramas de actividades presentado en UML.

A partir de las dos tablas anteriores ya sería posible realizar un modelo de requisitos funcionales que recoja la semántica definida en el metamodelo de requisitos funcionales presentado en esta tesis, utilizando para ello los elementos de los diagramas de actividades definidos en UML. Por ejemplo, cuando se desee modelar un requisito funcional, a partir de las tablas anteriores ya se conoce que dicho requisito se representará con el elemento *Activity*.

A continuación se describe otra sintaxis concreta alternativa a ésta y se mostrará un ejemplo de ambas sintaxis concretas.

1.2. Sintaxis concreta a partir de texto con formato

Esta sintaxis permite definir requisitos funcionales como un texto con un formato concreto, llamado plantilla, que permite identificar cada uno de los distintos elementos de un requisito funcional.

Al igual que en la sintaxis concreta anterior, en la tabla 7.3 se muestran los elementos del metamodelo de requisitos así como la manera de trasladarlos a plantillas.

Tabla 7.3. Conceptos de Requisitos Funcionales.

Concepto de requisitos	Representado por
ActorDelSistema	Plantilla
OrdeDeEjecución	Entrada en la plantilla de requisitos funcionales
Paso	Entrada en la plantilla de requisitos funcionales
RequisitoFuncional	Plantilla
Subsistema	Entrada en la plantilla correspondiente

El elemento *Plantilla*, mencionado en la tabla 7.3, denota un fragmento de texto que contiene un conjunto de entradas a completar. En este caso concreto, tanto un *ActorDelSistema* como *RequisitoFuncional* se definen como fragmentos de texto que especifican el contenido de dichos elementos (tal y como se vio en el capítulo del metamodelo de Requisitos Funcionales) e incluyen las entradas correspondientes para especificar dicho contenido. De manera relacionada, el elemento *Entrada* denota un fragmento de información de una plantilla.

Además de la tabla 7.3, los atributos y relaciones de cada uno de los elementos del metamodelo de requisitos funcionales también se definen como entradas en sus plantillas correspondientes. Cualquier plantilla que permita expresar los conceptos definidos en el

metamodelo de requisitos funcionales, independientemente de su formato visual, se podrá considerar como una sintaxis concreta válida para expresar requisitos funcionales. Análogamente, si se desea representar un requisito funcional, la tabla 7.3 indica que se debe utilizar una plantilla.

En la siguiente sección se muestra un ejemplo de cada una de las dos sintaxis concretas propuestas.

1.3. Ejemplo de sintaxis concretas para requisitos funcionales

En esta sección se define un ejemplo de un mismo requisito funcional definido con las dos sintaxis concretas presentadas en las secciones anteriores.

Tabla 7.4. Ejemplo de requisito funcional en plantilla.

Requisito funcional: Alta de nuevo elemento.

Descripción:
Este requisito funcional describe el comportamiento del sistema cuando un usuario desea dar de alta un nuevo elemento.

Precondiciones: No.

Poscondiciones: El elemento está dado de alta en el sistema.

Secuencia principal

5. El usuario solicita dar de alta un nuevo elemento en el sistema.
6. El sistema solicita la información del elemento.
7. El usuario introduce la información del elemento.
8. El sistema da de alta al elemento.

Secuencia alternativa

2.1. Si se ha superado el límite de elementos del sistema, entonces se muestra un mensaje de error y el resultado es finalizar la ejecución.

1.1. Si la información es incorrecta, entonces se muestra un mensaje de error y el resultado es ir al paso 2.

En primer lugar, en la tabla 7.4 se muestra el requisito funcional tomado como ejemplo, modelado utilizando una sintaxis textual (una plantilla). La manera de dar formato al texto no es relevante siempre y cuando dicho texto contenga, al menos, la información definida en el metamodelo de requisitos funcionales.

Por simplicidad, en el ejemplo anterior, se ha omitido la prioridad y los comentarios de los requisitos funcionales, la descripción del actor participante y la definición del subsistema. Sin embargo, el resto de conceptos sí están presentes. Por ejemplo, a partir de la plantilla de texto, se pueden identificar los distintos pasos, si un paso concreto es un paso del sistema o no, qué actor realiza el paso, etc. En trabajos preliminares, al comienzo del trabajo de investigación que culmina esta tesis, se definieron formatos textuales y prototipos de herramientas que los manipulaban automáticamente. Algunas referencias a estos trabajos son [Gutiérrez et-al, 2006-2].

A continuación, en la figura 7.1, se muestra este mismo requisito funcional pero representado mediante un diagrama de actividades según la notación de UML.

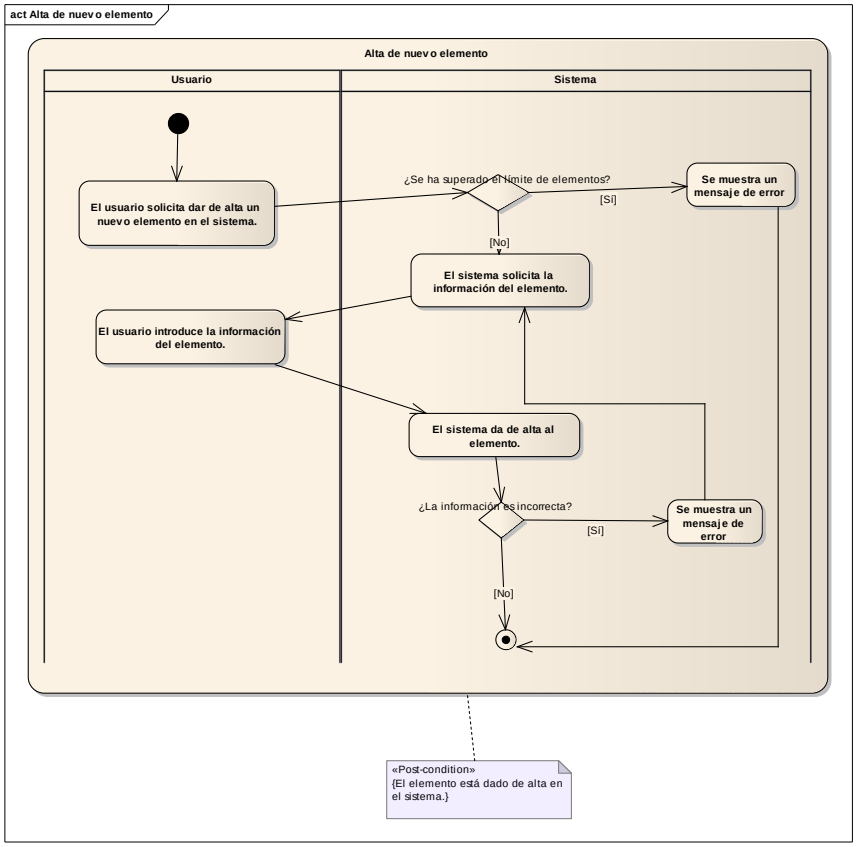


Figura 7.1. Ejemplo de requisito funcional definido como diagrama de actividades.

Como puede observarse, la semántica del requisito funcional definido en la tabla 7.4 y la semántica del requisito funcional definido en la figura 7.1 es la misma, variando sólo su representación o formato. Al igual que en el ejemplo anterior, este diagrama de actividades también define la información relevante especificada en el modelo de requisitos funcionales de este trabajo.

1.4. Gestión de relaciones de inclusión y extensión

Un aspecto identificado en el planteamiento del problema es la conveniencia de contemplar las relaciones de inclusión y extensión en la generación de pruebas funcionales del sistema. La solución propuesta en este trabajo de tesis para gestionar estas relaciones en el proceso de generación de pruebas funcionales del sistema, consiste en definir una etapa inicial en el proceso de construcción de un modelo de requisitos funcionales, conforme al metamodelo presentado en esta tesis, que realice un pre-procesado inicial de los requisitos funcionales de entrada. Esta etapa inicial tendrá como objetivo generar una instancia del metamodelo de requisitos funcionales que recoja el comportamiento del requisito funcional, conjuntamente con el comportamiento de todos sus requisitos funcionales incluidos y de aquellos requisitos funcionales que lo extiendan, Por ejemplo, si un conjunto de requisitos funcionales contienen un requisito funcional A, que incluye a un requisito funcional B y a otro requisito funcional C será necesario pre-procesar el modelo de requisitos funcionales y obtener un nuevo modelo que contenga al requisito funcional A y cuyo comportamiento incluya también los pasos del requisito funcional B y del requisito funcional C. Sobre este requisito A será sobre el que se aplique las transformaciones para la generación de artefactos de prueba. De manera análoga se resuelven las relaciones de extensión.

En este trabajo de tesis se propone que esta etapa de pre-procesamiento inicial se incluya como parte del proceso de traducción de sintaxis concreta a sintaxis abstracta, de tal manera que las transformaciones especificadas en el capítulo anterior no tendrán la necesidad de tener en cuenta dichas relaciones, dado que han sido resueltas a la hora de crear los modelos de requisitos funcionales. De esta manera, las pruebas funcionales del sistema generadas ya incluirán comportamiento de los requisitos funcionales incluidos o que extiendan al requisito funcional de origen.

2. Ejemplos de sintaxis concretas para pruebas funcionales del sistema

De manera análoga a la sección anterior, en esta sección se proponen dos sintaxis concretas para trabajar con modelos de pruebas funcionales del sistema y con cualquiera de los tres grupos lógicos mencionados. En este caso, se presentan las mismas sintaxis, una basada en texto y otra sintaxis gráfica, también por las mismas justificaciones que en la sección anterior. Las descripciones de ambas sintaxis se han agrupado en la sección 2.1.

2.1. Sintaxis concretas para los grupos lógicos del metamodelo de pruebas funcionales del sistema

De los tres grupos lógicos vistos en el metamodelo de pruebas funcionales del sistema, dos de ellos representan información de comportamiento (escenarios de prueba y casos de prueba) y uno de ellos representa información estática (valores de prueba). Para los grupos que representan información de comportamiento se puede utilizar como sintaxis concreta los diagramas de actividades de UML ya presentados, mientras que para la información estructural se pueden utilizar diagramas de clases. A continuación se describe con más detalle la sintaxis gráfica.

Al igual que en la sección anterior, para los grupos lógicos de escenarios de prueba y casos de prueba es necesario identificar los elementos de los diagramas de actividades que representarán los conceptos de dichos grupos. A continuación, en la tabla 7.5 se muestran los conceptos del grupo lógico de escenario de prueba junto con los elementos de los diagramas de actividades de UML elegidos para su representación como sintaxis concreta.

Tabla 7.5. Conceptos para escenarios de prueba.

Concepto de escenarios de prueba	Representado por
ActorDelSistema	ActivityPartition
EscenarioDePrueba	Activity
PasoDelEscenarioDePrueba y VerificaciónDelEscenarioDePrueba	Action, ActivityFinalNode y InitialNode
PaqueteDeEsenarioDePrueba	Package

Como se puede ver en la tabla anterior, los elementos elegidos de los diagramas de actividades de UML son los mismos que los elementos elegidos como sintaxis concreta de los

requisitos funcionales. La principal diferencia es la ausencia de los elementos *DecisionNode* de los diagramas de actividades dado que no son necesarios puesto que un escenario de prueba no tiene alternativas y siempre tiene un único camino de ejecución.

En el caso del grupo lógico de los valores de prueba, los elementos a utilizar para una de las posibles sintaxis concretas son los diagramas de clases. Como ya se ha mencionado, esta es la sintaxis concreta que se utiliza en el perfil de pruebas de UML. De nuevo, es necesario identificar qué elementos utilizados en los diagramas de clases de UML van a representar cada uno de los conceptos del grupo lógico de valores de prueba. Dicha información se muestra en la tabla 7.6.

Tabla 7.6. Conceptos para valores de prueba.

Conceptos de valores de prueba	Representado por
AtributoParaPrueba	Property
CombinaciónDeInstancias	Class
Instancia	Class
PaqueteDeCombinaciónDeInstancias	Package
ParticiónDeDatos	Class y Generalization
Quantum	Class
Restricción y *RestricciónDeCombinación*	Constraint
TipoDeRestricción y TipoeVariableOperacional	No tienen representación en esta sintaxis concreta

Los conceptos de la tabla 7.6 se describen con más detalle en el apéndice dedicado a los perfiles de UML.

Los diagramas de actividades también son una posible sintaxis concreta para los casos de prueba. En la tabla 7.7, se indican los conceptos de los casos de prueba junto con los elementos de los diagramas de actividades de UML elegidos para su representación como sintaxis concreta.

Tabla 7.7. Conceptos para casos de prueba.

Concepto de casos de prueba	Representado por
CasoDePrueba	Activity
PasoDePrueba y VerificaciónDePrueba	Action, ActivityFinalNode y InitialNode
ColecciónDeCasosDePrueba	Package

Un aspecto adicional que presenta el grupo lógico de casos de prueba es la relación de sus elementos con los elementos del grupo lógico de valores de prueba. En el caso de la

sintáis concreta basada en diagramas de actividades introducida así, se puede utilizar el flujo de datos que UML permite introducir en los diagramas de actividades para asociar a cada acción del diagrama de actividades el quantum (modelado por una clase) adecuado. En la siguiente sección se presenta un ejemplo de su uso.

Además de la sintaxis gráfica utilizando diagramas de actividades, también sería posible utilizar una sintaxis textual. Esta sintaxis textual será análoga a la ya vista par los requisitos funcionales y consistirá en definir como texto tabulado toda la información de atributos y relaciones de los elementos del metamodelo de pruebas funcionales del sistema. Debido a su similitud con la sintaxis concreta textual vista para los requisitos funcionales y la facilidad para entender esta sintaxis concreta, no se incluye en este trabajo una definición formal de la misma, sino que se define directamente en los ejemplo de la próxima sección.

En la siguiente sección se muestra un ejemplo de sintaxis gráfica y sintaxis textual para cada uno de los grupos lógico del metamodelo de pruebas funcionales del sistema.

2.2. Ejemplos de sintaxis concretas para pruebas funcionales del sistema

A partir del requisito funcional visto como ejemplo en la sección anterior, se van a definir, a modo de ejemplo, algunos artefactos del metamodelo de pruebas funcionales del sistema utilizando las sintaxis concretas presentadas en los párrafos anteriores.

En primer lugar se presenta un escenario del grupo lógico de escenarios de prueba (en concreto, el escenario principal de requisito funcional). Un ejemplo utilizando una posible sintaxis concreta textual, muy similar a la utilizada para requisitos funcionales se puede ver en la tabla 7.8.

Tabla 7.8. Ejemplo de Escenario de Prueba.

Escenario de prueba: Prueba 01

Requisito funcional origen: Alta de nuevo elemento.

Descripción:
Este requisito funcional describe el comportamiento del sistema cuando un usuario desea dar de alta un nuevo elemento.

Estado inicial: No.

Estado final: El elemento está dado de alta en el sistema.

Pasos del escenario de prueba

1. El usuario solicita dar de alta un nuevo elemento en el sistema.
2. El sistema solicita la información del elemento.
3. El usuario introduce la información del elemento.
4. El sistema da de alta al elemento.

Como se puede ver en la tabla 7.8, esta sintaxis concreta es prácticamente autodescriptiva y, tanto la plantilla como las entradas reflejan la información especificada en el metamodelo de casos de prueba (grupo lógico de escenarios de prueba). Este escenario de prueba del sistema verifica la secuencia principal del requisito funcional visto como ejemplo anteriormente. Este mismo escenario de prueba representado con la sintaxis abstracta de diagramas de actividades se muestra en la figura 7.2.

De nuevo, y al igual que en el ejemplo del requisito funcional visto con anterioridad, se observa que la semántica es la misma que la semántica del escenario de prueba de la tabla 7.8, variando únicamente el formato.

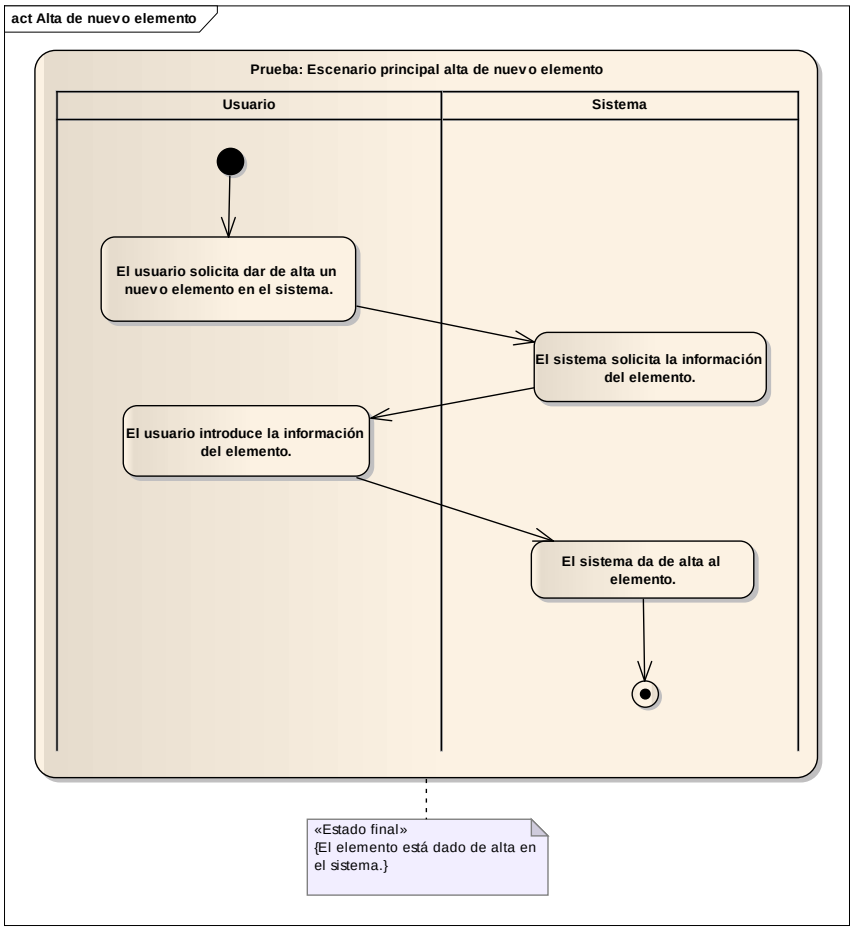


Figura 7.2. Ejemplo de escenario de prueba definido con diagrama de actividades.

A continuación, en las tablas 7.9 y 7.10 se muestran dos plantillas textuales que representan las dos variables operacionales presentes en el requisito funcional tomado como ejemplo.

Tabla 7.9. Ejemplo de sintaxis concreta para variable operacional I.

Variable Operacional *Nombre:* V01 *Tipo:* Del sistema *Descripción:* Límite de elementos en el sistema. .

Tabla 7.10. Ejemplo de sintaxis concreta para variable operacional II.

Variable Operacional Nombre: V02 *Tipo:* De Información de entrada. *Descripción:* Información del nuevo elemento. .

Para cada una de estas dos variables se definen dos particiones para separar los valores de sus dominios. Dichas particiones se muestran en las tablas 7.11 y 7.12 para la variable V01 y 7.13 y 7.14 para la variable V02.

Tabla 7.11. Ejemplo de sintaxis concreta para partición I.

Partición *Nombre*: V01P01. *Rango de valores:* Por debajo o igual al límite de elementos en el sistema.

Tabla 7.12. Ejemplo de sintaxis concreta para partición II.

Partición *Nombre*: V01P02. *Rango de valores:* Superior al límite de elementos en el sistema.

Tabla 7.13. Ejemplo de sintaxis concreta para partición III.

Partición *Nombre*: V02P01. *Rango de valores:* Información correcta.

Tabla 7.14. Ejemplo de sintaxis concreta para partición VI.

Partición *Nombre*: V02P02. *Rango de valores:* Información incorrecta.

A partir del requisito funcional de ejemplo, es posible descubrir que hay una instancia de la primera variable operacional y dos de la segunda. Como ejemplo, se va a modelar la combinación de instancias para el camino principal. La combinación de instancias que coincide con el escenario de prueba tomado como ejemplo (escenario principal del requisito funcional) se muestra en la tabla 7.15, utilizando de nuevo una sintaxis concreta de texto tabulado.

Tabla 7.15. Ejemplo de sintaxis concreta para combinación.

Combinación: I01V01 = V01P01, I01V02 = V01P02

En la notación textual de la tabla 7.15, cada par de asignación es un quantum. Así, la primera asignación: I01V01 = V01P01 representaría el quantum que asigna la instancia I01V01 con la partición V01P01. En la combinación de la tabla 7.15 sólo aparecen dos instancias de variables (I01V01 e I01V02) a pesar de que existen tres instancias de variables operacionales. Esto es así ya que la segunda instancia de la segunda variable operacional (I02V02) no toma ningún valor en la combinación elegida, se ha omitido de la combinación de la tabla 7.15.

Este mismo ejemplo también se puede definir con una sintaxis concreta similar a la instancia del metamodelo vista en el capítulo correspondiente o bien, de manera muy similar a utilizar un perfil de UML, como se verá en los apéndices. Un ejemplo de cómo quedarían las anteriores variables operacionales y particiones con una sintaxis concreta gráfica se muestra a continuación, en la figura 7.3.

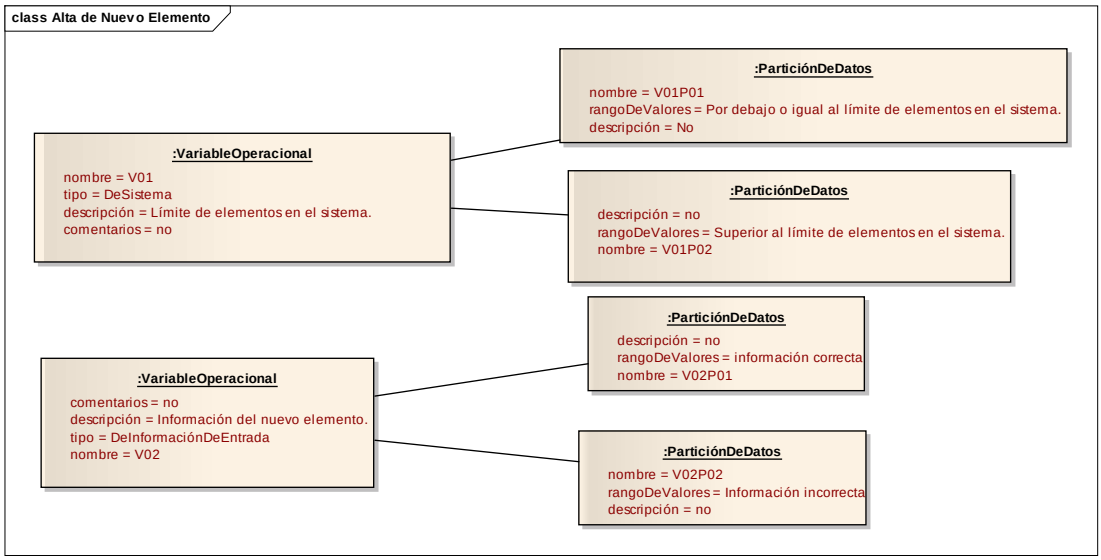


Figura 7.3. Ejemplo de sintaxis concreta gráfica para valores de prueba

Como se puede ver en la figura 7.3, y ya se adelantó en la sección anterior, una sintáis concreta gráfica basada en diagramas de clases es análoga al ejemplo de uso del grupo lógico de valores de prueba visto en el capitulo 5, por lo que no se continúa incluyendo aquí el diagrama para la combinación. En este trajo de tesis se presentan dos ejemplos de uso de diagramas de clases como sintaxis concreta adicionales: el primero de ellos se ha utilizado a la hora de poner ejemplos de uso de la sintaxis concreta (o metamodelo), en el capítulo 5, mientras que el segundo ejemplo se presentará al definir el perfil de UML para el grupo lógico de valores de prueba.

Por último, después de exponer un ejemplo para los grupos lógicos de escenarios de prueba y de valores de prueba, se presenta a continuación un último ejemplo para el grupo lógico de casos de prueba. Ahora, la información del escenario de prueba se combina con la información de los valores de prueba para obtener un caso de prueba. Al igual que en los ejemplos anteriores, el primer ejemplo de caso de prueba se define mediante una sintaxis concreta textual, tal y como se muestra en la tabla 7.16.

Tabla 7.16. Ejemplo de sintaxis concreta textual para casos de prueba.

Caso de prueba: Prueba 01

Requisito funcional origen: Alta de nuevo elemento.

Descripción:
Este requisito funcional describe el comportamiento del sistema cuando un usuario desea dar de alta un nuevo elemento.

Estado inicial: No.

Estado final: El elemento está dado de alta en el sistema.

Pasos del escenario de prueba

1. El usuario solicita dar de alta un nuevo elemento en el sistema.
2. El sistema solicita la información del elemento (el número de elementos está por debajo o igual al límite de elementos en el sistema).
3. El usuario introduce la información del elemento (la información es correcta).
4. El sistema da de alta al elemento.

Como puede verse en el ejemplo anterior, esta sintaxis es muy similar a la sintaxis utilizada para escenarios de prueba y para requisitos funcionales. Además, se ha añadido a cada paso del caso de prueba que lo requería información sobre el elemento del valor de prueba adecuado.

Al igual que en los escenarios de prueba, también se puede utilizar una sintaxis concreta basada en diagramas de actividades para definir un caso de prueba. Un ejemplo de cómo representar el anterior caso de prueba utilizando esta sintaxis concreta se muestra a continuación.

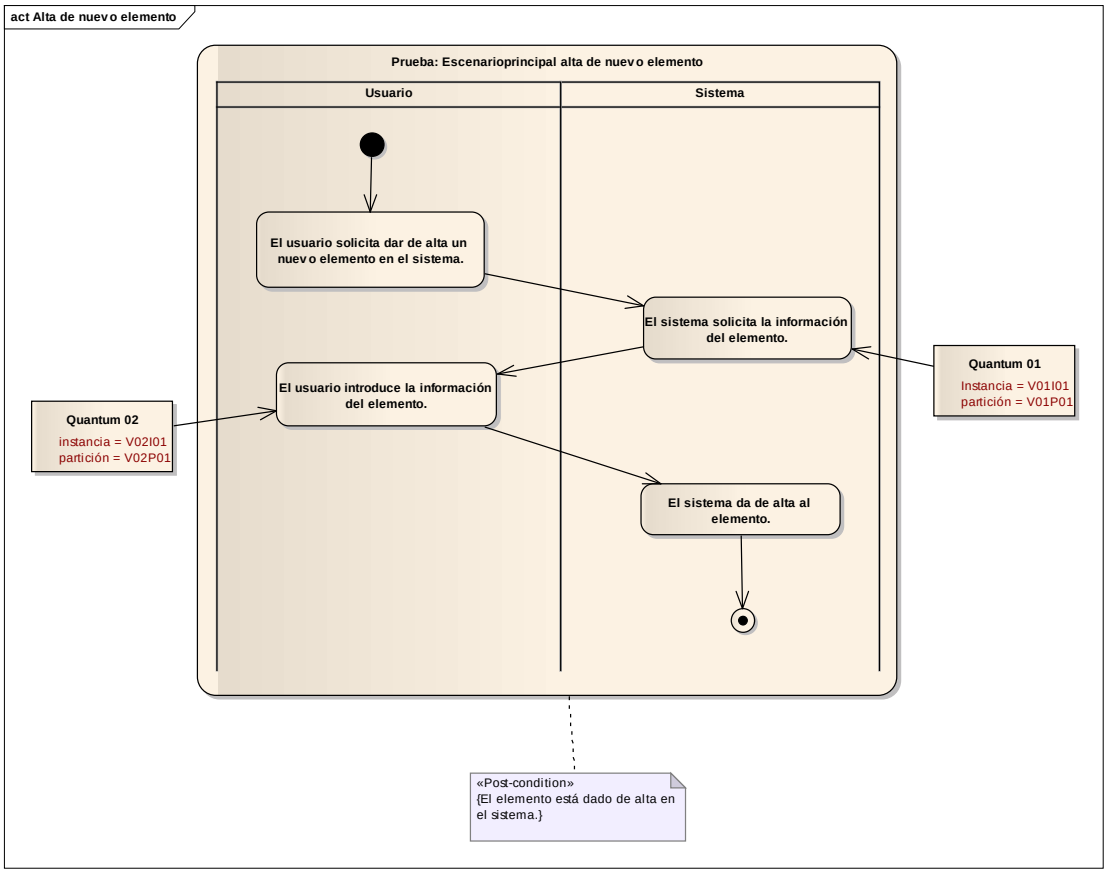


Figura 7.4. Ejemplo de sintaxis concreta gráfica para casos de prueba.

Como se puede ver en estos ejemplos de sintaxis concreta, los casos de prueba son los propios escenarios de prueba enriquecidos con la información de los valores de prueba. Además, en el ejemplo de la figura 7.4, se ve cómo se utiliza el flujo de objetos de los diagramas de actividades de UML para indicar los quantums utilizados en los pasos de prueba asociados a una variable operacional.

3. Herramienta MDETest

La herramienta MDETest implementa las transformaciones vistas en el capítulo anterior, permitiendo la automatización del proceso de generación de pruebas funcionales del sistema a partir de los requisitos funcionales del sistema bajo prueba. El objetivo principal de esta herramienta es implementar de manera fiel los metamodelos y transformaciones

especificadas en este trabajo de tesis, con el fin de ofrecer una justificación sobre la automatización del proceso de generación de pruebas funcionales del sistema.

En los apéndices de este trabajo de tesis, puede encontrarse un manual de usuario de la herramienta así como ejemplos.

3.1. Implementación de los modelos y transformaciones

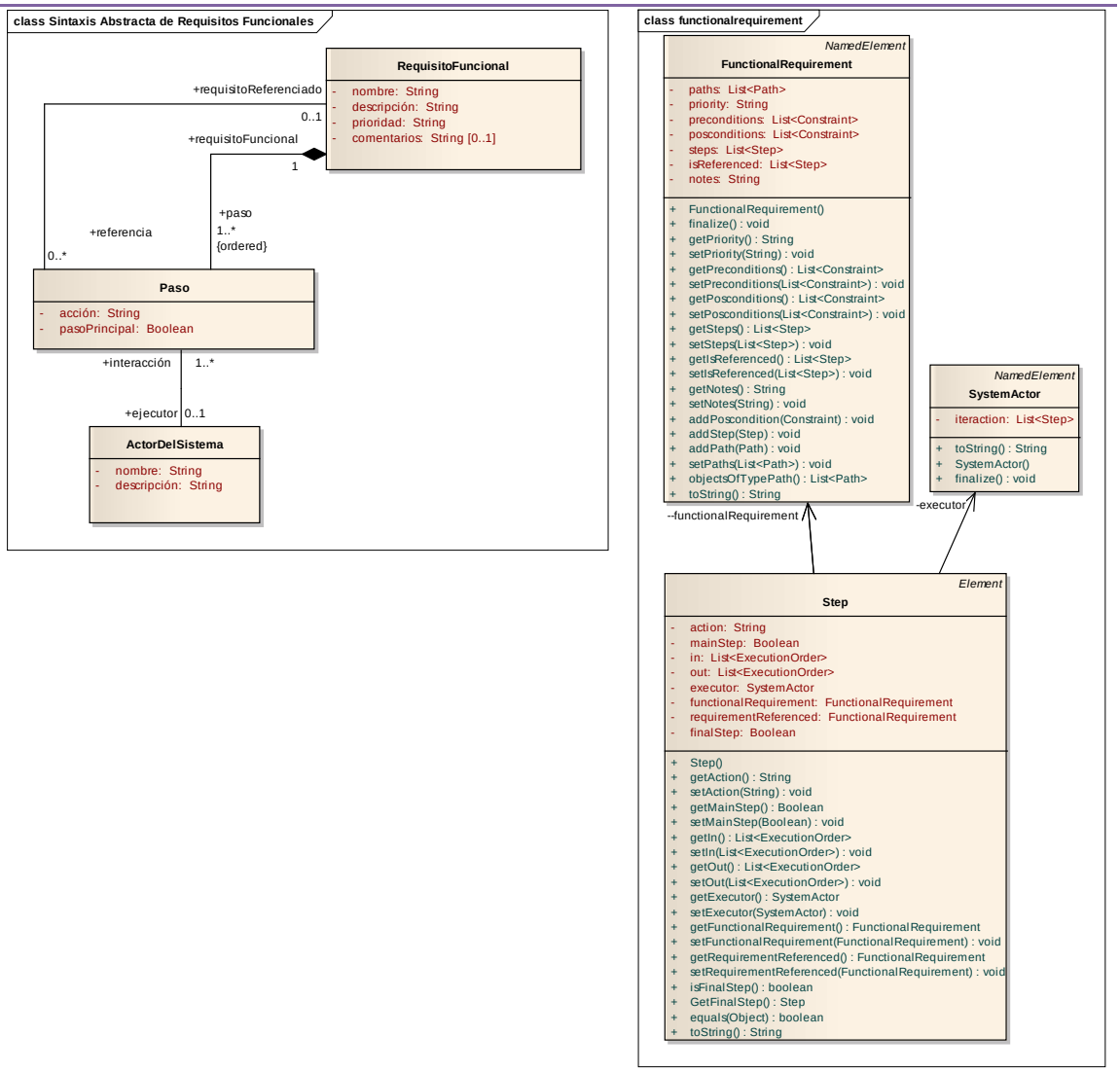
Aunque las transformaciones han sido definidas en lenguaje QVT Procedimental, se han implementado en la herramienta MDETest en lenguaje Java. La elección de lenguaje Java no es determinante ya que otro lenguaje de propósito general como C#, Python, Ruby, etc. podría haber sido utilizado de la misma manera.

Para garantizar la fidelidad del código Java con las transformaciones QVT especificadas en secciones anteriores de este trabajo de tesis, se han aplicado durante el desarrollo de la herramienta una serie de pasos destinados a garantizar que el código Java realiza el mismo proceso de transformación que el proceso especificado en QVT.

En un primer paso, se ha implementado en lenguaje Java una jerarquía de clases que modela de manera fiel los conceptos de los metamodelos presentados para que el código que implementa las transformaciones trabaje con las mismas estructuras de datos, con los mismos atributos y asociaciones, que las transformaciones definidas. Como ejemplo, en la tabla 7.17, se muestra un fragmento del metamodelo de requisitos funcionales (*RequisitoFuncional, ActorDelSistema* y *Paso*) comparado con la jerarquía de clases Java que lo implementa en la herramienta.

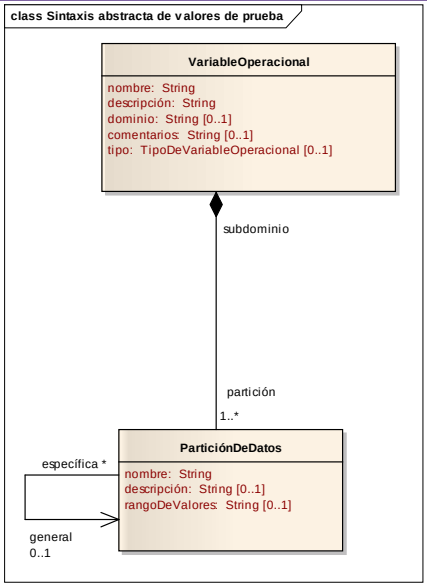
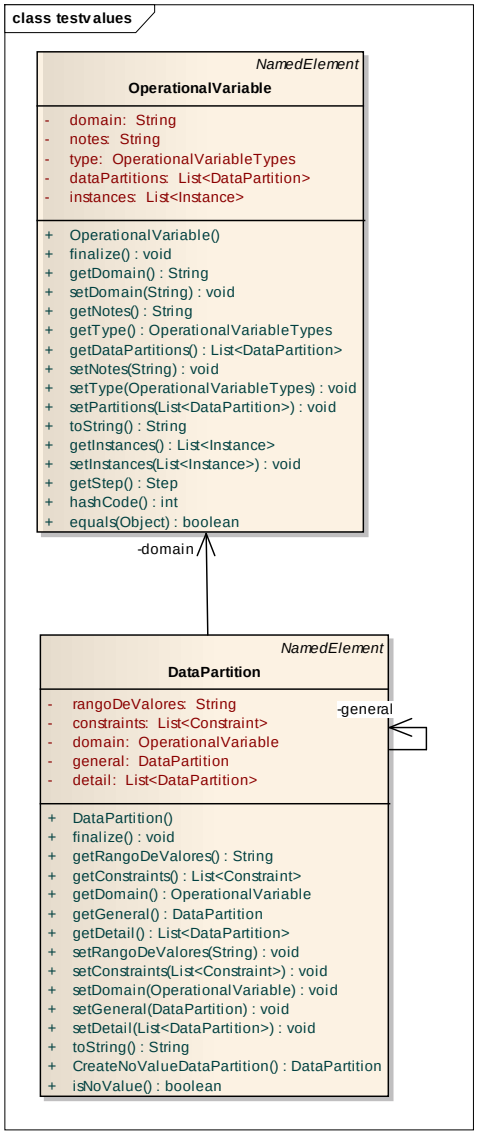
Tabla 7.17. Fragmento de metamodelo de requisitos funcionales y las clases Java de su implementación.

Metamodelo de requisitos funcionales	Clases Java



La tabla 7.18 muestra un fragmento del metamodelo de pruebas funcionales del sistema, en concreto del grupo lógico de valores de prueba (*VariableOperacional* y *ParticiónDeDatos*), comparado con la jerarquía de clases Java que implementa dicho metamodelo en la herramienta MDETest.

Tabla 7.18. Fragmento de grupo lógico de valores de prueba y las clases Java de su implementación.

Grupo lógico de valores de prueba	Clases Java
	

En un segundo paso, se ha definido una estructura de clases y métodos que representan las transformaciones en QVT. Los métodos de dichas clases tienen nombres y argumentos en consonancia con los nombres y argumentos de los mapeos y funciones auxiliares definidas en las transformaciones en QVT. Esto hace que las llamadas y parámetros sean análogos en ambos códigos y, además, permite identificar fácilmente qué fragmento de código QVT corresponde con un fragmento de código Java y viceversa. A continuación, en la tabla 7.19, se muestran dos ejemplos de un fragmento de transformación en QVT ya visto anteriormente y su código Java correspondiente.

Tabla 7.19. Ejemplo transformación QVT-Java I.

Transformación QVT

```
transformation RequisitosFuncionales_a_ModeloIntermedioDeCaminos(
    inout mrf: MetamodeloDeRequisitosFuncionales)
{
    main() {
        mrf.objectsOfType(RequisiToFuncional)->
            map RequisitoFuncional_a_Camino();
    }


    mapping RequisitoFuncional::RequisitoFuncional_a_Camino()
    {
        var pth : Camino;
        var codel : Set(OrdenDeEjecucion) {};
        self.generarCaminos(pth, self.paso->at(1),
            codel, mrf.caminos, 1, false);
    }


    //.....

}
```

Transformación Java

```
public void transform(FunctionalRequirementsModel frm){
    this.frm = frm;
    for (FunctionalRequirement fr:
      this.frm.objectsOfTypeFunctionalRequirement()) {
        mappingRequisitoFuncional_a_Camino(fr);
    }
}

void mappingRequisitoFuncional_a_Camino(FunctionalRequirement fr) {
    List<Path> paths = new ArrayList<Path>(30);
    this.helpperGenerate(new Path(), fr.getSteps().get(0),
            new HashSet<ExecutionOrder>(), paths,
            1, false);

    fr.setPaths(paths);

}
```

En la tabla 7.20 se puede ver, en el lado izquierdo, un fragmento de la transformación de requisito funcional a modelo intermedio de caminos y, en el lado derecho, la implementación de dicha transformación. El punto de entrada *main* de QVT corresponde con el método transform en Java, mientras que el mapeo QVT *RequisitoFuncional_a_Camino* corresponde con el método Java *mappingRequisitoFuncional:a_Camino.* Como se puede apreciar, la codificación en Java es prácticamente directa. Otro ejemplo se muestra en la tabla 7.20, en concreto la transformación de órdenes de ejecución a particiones de variables operacionales.

Tabla 7.20. Ejemplo transformación QVT-Java II.

Transformación QVT

```
mapping OrdendeEjecución::ordendeEjecución_a_Partición
(in vo: VariableOperacional)
    : Partición
{
    nombre := "Particion: " +vo.nombre;
    subdominio := vo;
    ordenDeEjecucionOrigen := self;
    restriccion := self.puntoDeExcepcion;
}
```

Transformación Java

```
private DataPartition
    mappingExecutionOrder2DataPartition
        (ExecutionOrder eo, OperationalVariable ov) {
    DataPartition p = new DataPartition();

    p.setName("Particion: " + ov.getName());
    p.addTrace(eo);
    p.setDomain(ov);
    p.getConstraints().add(eo.getExceptionPoint());

    return p;
}
```

3.2. Sintaxis concretas utilizadas en la herramienta

Uno de los principales aspectos a abordar a la hora de automatizar el proceso de transformación presentado en este trabajo es elegir las sintaxis concretas a utilizar, tanto para la definición de la información de los requisitos funcionales de entrada como para representar la información de los artefactos de pruebas obtenidos. En secciones anteriores se han expuesto algunos ejemplos de notaciones para estas sintaxis.

De las sintaxis concretas vistas en secciones anteriores para los requisitos funcionales, se ha optado por utilizar los diagramas de actividades para la herramienta MDETest. Además, se ha optado por utilizar la herramienta de modelado UML Sparx Enterprise Architect para el modelado de requisitos funcionales y diagramas de actividades. Estos diagramas están almacenados en un formato propio, por lo que es necesario añadir a la herramienta MDETest un paso adicional para generar un modelo de requisitos funcionales conforme al metamodelo presentado en esta tesis a partir del formato de Enterprise Architect. Este formato es una base de datos en MS Access, así, la sintaxis concreta de un fragmento del modelo de la figura 7.1 (es decir, cómo almacena la herramienta dicho modelo de clases) se muestra en la figura 7.5.

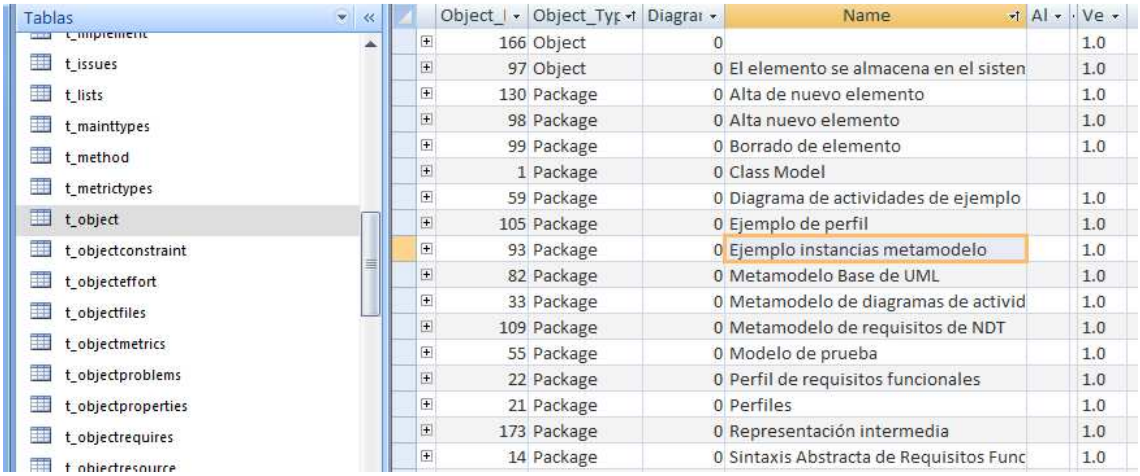


Figura 7.5. Ejemplo de sintaxis concreta de la herramienta Enterprise Architect.

Tabla 7.21. Código Java para leer un modelo de requisitos funcionales.

```
public FunctionalRequirementsModel
   readFunctionalRequirementsModel(String fileName, String p) {
    String frSubsystem = p;
    FunctionalRequirementsModel frm = new FunctionalRequirementsModel();
    EAPConnectionFacade.Connect(fileName);
    List<Subsystem> lp =
        loadSubsystemList(frSubsystem);
    EAPConnectionFacade.Close();
    EAPConnectionFacade.Connect(fileName);
    List<FunctionalRequirement> frl = this.loadAllFRStartingIn(lp);
    EAPActivityDiagramDAO adDAO = EAPConnectionFacade.getEAPActivityDiagramDAO();
    for (FunctionalRequirement fr: frl)
        fr.setSteps(adDAO.loadAllSteps(
            EAPConnectionFacade.getEAPFunctionalRequirementDAO().getFRId(fr), fr));
    List<SystemActor> sal = this.loadAllSAStartingIn(lp);
    EAPConnectionFacade.Close();
    frm.objectsOfTypeSubsystem().addAll(lp);
    for (FunctionalRequirement fr: frl)
        frm.addFunctionalRequirement(fr);
    for (SystemActor sa: sal)
        frm.addSystemActor(sa);
    return frm;
}
```

En la tabla 7.21 se muestra un ejemplo del código Java que permite crear un modelo de requisitos funcionales a partir de un diagrama de actividades almacenado en una base de datos Access con el formato de Enterprise Architect.

Una vez obtenido el modelo de requisitos funcionales, la herramienta aplicará todas las transformaciones vistas en el capítulo de transformaciones para obtener los caminos a partir de los requisitos funcionales, los escenarios de prueba, los valores de prueba y los casos de prueba. Al trabajar directamente con los metamodelos definidos en este trabajo de tesis no se necesita incorporar ningún detalle específico a las transformaciones vistas. Para la definición de los artefactos de salida, esto es de los modelos de los tres grupos lógicos de prueba, no se ha implementado ninguna sintaxis concreta. Un ejemplo de los resultados de esta herramienta se muestra en el anexo B.

La herramienta MDETest permite la incorporación de sintaxis concretas alternativas. Para ello, como ya se ha comentado al principio de este capítulo, solo será necesario proporcionar el mecanismo que permita convertir de una sintaxis concreta a sintaxis abstracta, en el caso de que sea una sintaxis concreta para requisitos funcionales, o de sintaxis abstracta a sintaxis concreta, en el caso de que sea una sintaxis concreta para pruebas funcionales del sistema.

4. Conclusiones

En este capítulo se han presentado ejemplos de sintaxis concretas para los dos metamodelos definidos en este trabajo de tesis, y también se ha presentado la herramienta MDETest, desarrollada para la implementación del proceso de transformación de requisitos funcionales a pruebas funcionales del sistema. Como se mencionó en capítulos anteriores, uno de los objetivos planteados en esta tesis es la definición de un proceso automatizable. El desarrollo de una herramienta que implementa las transformaciones vistas en el capítulo anterior viene a cumplir este objetivo y a reforzar la idea de que el proceso presentado puede ser realizado de una manera automática.

En las primeras etapas de este trabajo de investigación, se desarrollaron 2 herramientas (llamadas ObjectGen y ValueGen, aunque tiempo después ambas se distribuían con el nombre genérico de TestGen), con el fin de explorar la viabilidad de generar automáticamente casos de prueba a partir de requisitos funcionales definidos como una plantilla de texto. Una vez vista dicha viabilidad, estas herramientas se abandonaron para adaptar un enfoque de desarrollo guiado por modelos que permitió, entre otros beneficios, flexibilizar el formato de entrada de los requisitos funcionales

En un trabajo complementario a esta tesis [Gutiérrez et-al, 2008] se han definido un conjunto de transformaciones necesarias para construir diagramas de actividades a partir de requisitos funcionales definidos como texto, dado que, como se ha visto, son semánticamente compatibles. Estas transformaciones adicionales permiten seguir trabajando con requisitos textuales de texto junto con la herramienta MDETest.

Más adelante, en los anexos de esta tesis, se aplicará la herramienta descrita en este capítulo a un caso práctico y se evaluará hasta qué punto es eficaz la automatización aquí presentada.

Capítulo VIII. Conclusiones

En los capítulos anteriores se han sentado las bases para definir un entorno formal y metodológico para la generación de pruebas funcionales del sistema. También se han presentado las motivaciones que han fundamentado el desarrollo de este trabajo, se ha analizado la situación actual y las soluciones propuestas para abordar el problema de la generación de pruebas funcionales del sistema a partir de requisitos funcionales y se han propuesto y analizado metamodelos y transformaciones para resolver este problema utilizando una perspectiva de ingeniería guiada por modelos, permitiendo la formalización, sistematización y posible automatización, como se ha visto en el capítulo anterior, del proceso propuesto. A este respecto, se ha presentado una herramienta que justifica la automatización presentada por los metamodelos y transformaciones. También, durante los capítulos anteriores, se han comentado y referenciado trabajos que han servido como fuente de inspiración en alguna de las propuestas del trabajo. Sin embargo, es necesario dejar constancia detallada de lo que se ha asumido de los trabajos tomados como referencia y de cuáles han sido las aportaciones reales de este trabajo. En este capítulo se analizan las aportaciones que se han realizado en esta tesis, así como los elementos adoptados de otras propuestas y en la medida en la que se han asumido sus ideas.

La organización de este capítulo se describe a continuación. En la sección 1 se exponen las aportaciones de este trabajo de tesis. A continuación, en la sección 2, se plantean los trabajos futuros que continúan la línea de investigación abierta con la realización de esta tesis. Finalmente, en la sección 3, se introducen una serie de conclusiones finales.

1. Aportaciones de este trabajo de tesis

Como se ha desarrollado en capítulos anteriores, esta tesis se ha nutrido de ideas que vienen de otros trabajos ya existentes. El primer punto de referencia lo conforman los trabajos previos de generación de pruebas funcionales a partir de requisitos funcionales (las referencias pueden consultarse en el capítulo 2). Otro punto de referencia ya citado y descrito con más detalle en el capítulo 3 son: la metodología para elicitación de requisitos NDT, el cuál es un trabajo previo del grupo de investigación en cuyo seno se ha gestado este trabajo de tesis, y algunos trabajos de referencia para poder trabajar desde una ingeniería guiada por modelos (cuyas principales referencias pueden consultarse en el capítulo 3), así como herramientas necesarias para trabajar con esta perspectiva, principalmente UML y QVT.

A continuación, en las siguientes secciones, se describen en detalle las aportaciones vistas a lo largo de este trabajo de tesis.

1.1 Aportaciones generales

Una vez que se ha planteado la solución propuesta en esta tesis para el problema de la generación sistemática y automatizable de casos de prueba funcionales del sistema a partir de requisitos funcionales (principalmente en los capítulos 4, 5 y 6), es posible compararla con las propuestas estudiadas en el capítulo 2 e identificar cuáles son las aportaciones originales presentadas. A continuación, a modo de resumen, se hace una enumeración de estas aportaciones.

1. Un estudio del estado del arte sobre las propuestas existentes para la generación de prueba funcionales del sistema a partir de requisitos funcionales.
2. La definición formal de un metamodelo de requisitos funcionales y de un metamodelo de pruebas del sistema.
3. La aportación de las herramientas necesarias para la definición de estos metamodelos en la forma de sintaxis abstracta y extensión formal de UML, así como la especificación de este proceso en transformaciones en lenguaje QVT.
4. La definición de procesos sistemáticos de derivación que permiten relacionar los metamodelos unos con otros para conseguir una derivación sistemática.
5. La disponibilidad de una herramienta que permite ejecutar las transformaciones propuestas y que, por tanto, elevan los procesos sistemáticos de derivación a procesos automáticos, aumentando así la potencia de estos procesos.
6. El enriquecimiento de trabajos previos del grupo de investigación cristalizados en la metodología NDT mediante la reutilización del modelo de requisitos y la adición del proceso de generación de pruebas funcionales del sistema.

Estas aportaciones pueden resumirse en una aplicación práctica de la ingeniería guiada por modelos mediante el desarrollo formal y automatizable, bajo los escenarios vistos en la sección anterior de un proceso para la obtención de pruebas funcionales del sistema a partir de los requisitos funcionales.

Además, como ya se ha mencionado con anterioridad, este trabajo también aporta un enriquecimiento a otros trabajos previos del grupo de investigación, principalmente,

cristalizados en el uso de NDT como una de las bases de este trabajo, tal y como se discutió en el tercer capítulo.

Las aportaciones específicas de este proceso se describen con más detalle en la siguiente sección.

1.2. Proceso aportado

Al igual que otros procesos desarrollados mediante ingeniería guiada por modelos (como, por ejemplo, NDT), el proceso de generación de pruebas funcionales del sistema presentado en esta tesis está formado por dos bloques básicos: metamodelos y transformaciones. La primera mitad de esta sección describe dichos metamodelos y la segunda mitad describe las transformaciones desde la perspectiva de aportaciones originales. Con esta idea, los metamodelos tratados son los siguientes:

1. Metamodelo de requisitos funcionales. Para este metamodelo, se ha planteado durante el capítulo 4 la definición formal y las técnicas para la definición del metamodelo.

2. Metamodelo de pruebas funcionales del sistema. La definición de los tres metamodelos para representar las pruebas funcionales del sistema es una aportación original de este trabajo de tesis. El grupo lógico de escenarios de pruebas ha servido para formalizar toda la información referente a escenarios de prueba. Los escenarios de prueba son los artefactos utilizados con más frecuencia en los trabajos de prueba estudiados en el capítulo 2. El grupo lógico de valores de prueba ha tomado como base el perfil de pruebas de UML, en concreto el concepto de variable operacional y partición, tal y como se definen en la propuesta Cathegory-Partition Method presentada en el capítulo 2, pero lo ha ampliado con elementos adicionales para formalizar combinaciones de valores de prueba. Por último, ambos grupos lógicos se han combinado en un tercer grupo lógico, el grupo lógico de los casos de prueba, el cual formaliza la representación combinada de la información de los grupos lógicos de escenarios de prueba y valores de prueba.

Una vez presentado un recorrido por los metamodelos del proceso presentados en este trabajo de tesis, en los siguientes párrafos se presentan las técnicas aportadas. En cuanto a las técnicas de definición de los metamodelos, ya se ha presentado en los apartados anteriores cuáles han sido las influencias. De los patrones y técnicas de definición, cabe destacar la

influencia de UML y, en concreto, de los diagramas de clases y en las técnicas para extender formalmente UML y con más detalle, la técnica de los perfiles de UML.

. Otro gran grupo de técnicas aportadas son las relacionadas con los procesos sistemáticos que se definen desde las relaciones de derivación establecidas entre los metamodelos. Todas estas relaciones, así como los procesos sistemáticos son aportaciones de este trabajo. En concreto, se han especificado formalmente los procesos para obtener escenarios de prueba, valores de prueba y casos de prueba a partir de la información de requisitos funcionales, utilizando para ello el lenguaje QVT. Las transformaciones aportadas en este trabajo de tesis, especificadas en el capítulo VI, se enumeran a continuación.

1. Transformación para enriquecer el modelo de requisitos funcionales con caminos generados a partir de los propios requisitos funcionales.
2. Transformación para obtener un modelo de escenarios de prueba a partir de un modelo de requisitos funcionales enriquecido con caminos.
3. Transformación para obtener un modelo de valores de prueba a partir de un modelo de requisitos funcionales enriquecido con caminos.
4. Transformación para obtener un modelo de casos de prueba a partir de un modelo de requisitos funcionales enriquecidos con caminos y sus correspondientes modelos de escenarios de prueba y de valores de prueba.

La formalización de este proceso ha permitido su implementación en la herramienta presentada en el capítulo anterior. Esta herramienta, además de ser una aportación original de este trabajo de tesis, es un ejemplo de que es posible obtener casos de prueba funcionales automáticamente a partir de los requisitos funcionales. Un detalle relevante, es que se ha conseguido que la herramienta no requiera de opciones de configuración específica ni necesite información adicional para su funcionamiento aparte de la propia información de los requisitos funcionales. Un ejemplo de esto se verá más adelante, en los anexos de la tesis, en dónde se aplica la herramienta en un caso práctico que ya estaba realizado antes de la construcción de dicha herramienta, que no ha sido realizado para ser usado en esta tesis, que no ha sido desarrollado por nadie que haya participado en esta tesis, y que no ha sido necesario modificar para que la herramienta pueda trabajar con él.

1.3. Herramientas aportadas

Como se ha presentado en el capítulo anterior y se volverá a describir con más detalle en el anexo B, una de las aportaciones originales de este trabajo de tesis es una herramienta que implementa las trasformaciones para la generación de artefactos de prueba de manera que

pueden realizarse de manera automática y no supervisada a partir del modelo de requisitos funcionales.

En el capítulo II se ha discutido la existencia de herramientas de soporte entre las propuestas estudiadas y se ha visto que no todas las propuestas cuentan con una herramienta, ni siquiera todas las propuesta son susceptibles de automatización.

Un punto relevante de la herramienta desarrollada, como se verá en el anexo B, es que su modificación está abierta para la incorporación de nuevas sintaxis concretas. En este sentido, un valor añadido de la herramienta como aportación de este trabajo de tesis es que un usuario de la herramienta no tiene por qué adaptarse al formato de los modelos sino que puede extender la herramienta para que el proceso de transformación de este trabajo de tesis trabaje con la sintaxis concreta que requiera el usuario.

Figura 8.1. Herramienta NDT para la generación de escenarios de prueba.

Otra aportación de esta tesis al apartado de herramientas es la integración de parte del proceso presentado en este trabajo en el conjunto de herramientas de soporte de NDT, figura 8.1. Sin embargo este conjunto de herramientas solo tiene implementada en la actualidad la parte encargada de obtener los escenarios de prueba.

2. Trabajos futuros y nuevas líneas de investigación

A lo largo de este trabajo de tesis han ido apareciendo algunos puntos abiertos que podrían ser objeto de un estudio posterior a partir de los resultados vistos a lo largo de este

trabajo. En esta sección se retoman dichos puntos y se exponen cuáles pueden ser las líneas de investigación a plantear a partir del trabajo aquí realizado.

Uno de estos puntos es cómo identificar de manera sistemática el tipo de la variable operacional. Como se ha visto, si bien cada tipo tiene una semántica claramente definida, no se ha definido en las transformaciones un criterio que permita identificar el tipo a partir de la información contenida en los requisitos, por lo que se haría necesario estudiar si se debería añadir información adicional al metamodelo de requisitos funcionales expuesto para poder definir dicho criterio.

Otro punto abierto es la integración de este trabajo con NDT. Como se ha mencionado con anterioridad, este trabajo de tesis continúa NDT y lo enriquece añadiendo un proceso de generación de pruebas a partir de un metamodelo de requisitos funcionales que encaja con la propuesta de requisitos funcionales presentada en NDT.

Un tema detectado durante la realización de este trabajo es la necesidad de poder seleccionar un subconjunto de artefactos de prueba atendiendo a criterios como las pruebas que verifiquen requisitos de más alta prioridad o las pruebas que ejerciten una mayor cantidad de comportamiento. Un unto abierto como posible trabajo futuro es definir dichos criterios e integrarlos en un proceso automático.

Otro posible punto abierto sería la traducción de sintaxis concreta a sintaxis abstracta y viceversa. El trabajo principal de este trabajo de tesis se centra en formalizar la información de los requisitos funcionales y las pruebas y los procesos de generación de pruebas funcionales del sistema. Sin embargo, ya se ha visto que no es cómodo trabajar directamente con instancias de metamodelos. Aunque se han propuesto algunas sintaxis concretas de ejemplo para poder gestionar más cómodamente la información recogida en los metamodelos presentados en esta tesis, no se ha definido de manera formal ninguna sintaxis concreta ni ningún mecanismo para obtener una instancia de uno de los metamodelos vistos a partir de su sintaxis concreta. En este caso, la implementación de la sintaxis concreta de diagramas de actividades UML de la herramienta de modelado Enterprise Architect ha sido ad-hoc.

Un primer punto de partida sobre este punto abierto se exploró en el trabajo [Gutiérrez et-al, 2008], donde se proponía un conjunto de transformaciones para obtener un diagrama de actividades UML a partir de un modelo de requisitos funcionales escrito en formato texto tabulado, según el formato definido por NDT en sus primeros trabajos [Escalona, 2004].

3. Conclusiones

Para finalizar esta tesis, se describen en los siguientes párrafos las conclusiones finales.

En esta tesis se ha presentado un trabajo que ha comenzado con un planteamiento del problema basado en los resultados y conclusiones resultantes de trabajos comparativos.

El trabajo presentando resulta novedoso a la vista de las carencias detectadas en el capítulo 2, de los objetivos planteados en el capítulo 3 y de los metamodelos y transformaciones desarrollados en los capítulos 4, 5 y 6.

Finalmente, el capítulo 7 ha intentado ofrecer una justificación de la automatización que permite el formalismo de los metamodelos y las transformaciones.

Como resumen final, resaltar que durante todo el período de realización de la tesis se han publicado varios artículos en diferentes foros. Una bibliografía completa de los trabajos publicados puede encontrarse en los apéndices de este trabajo.

Referencias

[Balcer et-al, 1990] M. Balcer, W. Hasling, T. Ostrand. 1990. Automatic Generation of Test Scripts from Formal Test Specifications. Proceedings of ACM SIGSOFT'89 - Third Symposium on Software Testing, Verification, and Analysis (TAVS-3), ACM Press, pp. 257-71.

[Bertolino et-al, 2003] Bertolino, A., Gnesi, S. 2003. Use Case-based Testing of Product Lines. ESEC/FSE'03. Helsinki, Finland.

[Binder, 1999] Binder, R.V. 1999. Testing Object-Oriented Systems. Addison Wesley.

[Boddu et-al, 2004] Boddu R., Guo L., Mukhopadhyay S. 2004. RETNA: From Requirements to Testing in Natural Way. 12th IEEE International Requirements Engineering RE'04.

[Burnstein, 2003] Burnstein, I. 2003. Practical software Testing. Springer Professional Computing. USA.

[Denger et-al, 2003] Denger, C. Medina M. 2003. Test Case Derived from Requirement Specifications. Fraunhofer IESE Report. Germany.

[Durán et-al, 2000] Durán A., Bernardez B. 2000. Metodología para la Elicitación de Requisitos de Sistemas Software (versión 2.1). Informe Interno. http://www.lsi.us.es/docs/informes/lsi-2000-10.pdf

[Dustin et-al, 2002] Dustin E., Rashka J., McDiarmid D. 2002. Quality Web Systems. Addison-Wesley 1st edition. USA.

[Escalona, 2004] Escalona M.J. 2004. Modelos y técnicas para la especificación y el análisis de la Navegación en Sistemas Software. Ph. European Thesis. Department of Computer Language and Systems. University of Seville. Seville, Spain

[Escalona et-al, 2008] Escalona M. J., Aragón, 2008 NDT. A model-driven approach for web requirements. IEEE Transactions on Software Engineering, vol. 34, pp. 377-394.

[Escalona et-al, 2011] Escalona M.J., Gutiérrez J.J., Mejías M., Aragón G., Ramos I., Torres J., Domínguez F.J. 2011. An overview on test generation from functional requirements. The Journal of Systems and Software. Elsevier ISSN: 0164-1212.

[Fondement et-al, 2004] Fondemenet F. Silaghi R. 2004. Defining Model Driven Engineering Process. 3th Workshop in Software Model Engineering (WISME2004). October 11-15, Lisbon, Portugal

[Fröhlich et-al, 1999] Fröhlich, P, Link, J. 1999. Modelling Dynamic Behaviour Based on Use Cases. Proceedings of Quality Week Europe. Brussels.

[Fröhlich et-al, 2000] Fröhlich, P, Link, J. 2000. Automated Test Case Generation from Dynamic Models. ECOOP 2000. pp. 472-491.

[Fuentes et-al, 2004] Fuentes L. Vallecito A. 2004. An Introduction to UML Profiles. UPGADE vol. V, nº 2, april 2004.

[Giachetti et-al, 2008] Giachetti G. Marín B. Pastor O. 2008. Perfiles UML y Desarrollo Dirigido por Modelos: Desafíos y Soluciones para Utilizar UML como Lenguaje de Modelado Específico de Dominio. Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos. Vol. 2. No. 3, 2.008

[Glinz et-al, 1999] Martin Glinz, Johannes Ryser. 1999. A Practical Approach to Validating and Testing Software Systems Using Scenarios Quality. Week Europe QWE'99 in Brussels, Institut für Informatik, Universität Zürich.

[Gutiérrez et-al., 2005] Gutiérrez, J.J., Escalona M.J., Mejías M., Torres, J. 2005. Analysis of proposals for the generation of system test cases from system requirements. Proceedings of the CAISE'05 Forum; Portugal (2005-06)

[Gutiérrez et-al, 2006] Gutiérrez, J.J., Escalona M.J., Mejías M., Torres, J. 2006. Generation of test cases from functional requirements. A survey. 4º Workshop on System Testing and Validation. Potsdam. Germany.

[Gutiérrez et-al, 2006-2] Gutiérrez J.J. Escalona M.J. Mejías M. Torres J. 2006. Modelos Y Algoritmos para la Generación de Objetivos de Prueba. Jornadas sobre Ingeniería del Software y Bases de Datos JISBD 06. Sitges. Spain.

[Gutiérrez et-al, 2008] Gutiérrez J.J., Nebut C., Escalona M.J., Mejías M., Ramos I. 2008. Visualization of use cases through automatically generated activity diagrams. Lecturer Notes in Computer Science. Volumen: 5301, Páginas: 83-96.

[Heumann, 2002] Heumann , J. 2002. Generating Test Cases from Use Cases. Journal of Software Testing Professionals.

[Hsia et-al, 1994] Hsia, P, Kung D., Sell C. 1994. A Usage-Model Based Approach to Test the Therac-25. Safety and Reliability in Emerging Control Technologies. IFAC Workshop. Pp55-63. Florida. USA.

[Hsia et-al, 1997] Hsia, P, Kung D., Sell C. 1997. Software Requirements and Acceptance Testing. Anuals of software Engineering. Pag 291-317.

[Ibrahim et-al, 2007] Ibrahim R. Saringat M. Z. Ibrahim N. Ismail N. 2007. An Automatic Tool for Generating Test Cases from the System's Requirements. IEEE 7th International Conference on Computer and Information Technology CIT2007. Fukushima, Japón

[IEEE, 1990] IEE Standards Board. 1990. IEEE Standard Glossary of Software Engineering Terminology.

[Ismail et-al, 2007] Ismail N., Ibrahim R., Ibrahim N. 2007, Automatic Generation of Test Cases from Use-Case Diagram International Conference on Electrical Engineering and Informatics Institut Teknologi Bandung, Indonesia June 17-19, 2007

[Labiche et-al, 2002] Labiche Y., Briand, L.C. 2002. A UML-Based Approach to System Testing, Journal of Software and Systems Modelling (SoSyM) Vol. 1 No.1 pp. 10-42.

[Métrica-3] Métrica v3. Portal de la Administración Electrónica. http://administracionelectronica.gob.es/?_nfpb=true&_pageLabel=P60085901274201580632&langPae=es. Visitada el 31/05/2011.

[Mogyorodi, 2001] Gary Mogyorodi. 2001. Requirements-Based Testing: An Overview. 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39). pp. 0286

[Mogyorodi, 2002] Gary E. Mogyorodi. 2002. Requirements-Based Testing: Ambiguity Reviews. Journal of Software Testing Professionals. p 21-24.

[Mogyorodi, 2003] Gary E. Mogyorodi .What Is Requirements-Based Testing?. 15th Annual Software Technology Conference. Apr. 28-May 1. Salt Lake City, USA

[Myers, 2004] Myers, G. J. The Art of Software Testing, Second Edition . 2004. John Wiley & Sons, Inc. New York, NY, USA

[Naresh, 2002] Naresh, A. 2002. Testing From Use Cases Using Path Analysis Technique. International Conference On Software Testing Analysis & Review.

[NDT-Suite, 2011] NDT-Suite. www.iwt2.org. Visitada el 31/05/2.011.

[OMG-OCL, 2010] Object Management Group. OMG OCL Specification 2.0. 2010. http://www.omg.org. Visitada el 24/06/2.011

[OMG-QVT, 2010] Object Management Group. Query View Transformation Specification 1.0. 2010. http://www.omg.org. Visitada el 24/06/2.011

[OMG-UML, 2011] Object Management Group. 2011. Unified Modelling Language 2.4. www.omg.org. Visitada el 24/06/2.011

[OMG-UMLTP, 2005] Object Management Group. 2005. The UML Testing Profile 1.0. www.omg.org. Visitada el 24/06/2.011

[Ostrand et-al, 1988] Ostrand, TJ, Balcer, MJ. 1988. Category-Partition Method. Comunications of the ACM. 676-686.

[Pressman, 2010] Pressman, R.S. 2010. Ingeniería del Software, Un Enfoque Práctico. McGraw-Hill, séptima edición.

[Ruder et-al, 2004] Ruder A. et-al. 2004. A Model-based Approach to Improve System Testing of Interactive Applications. ISSTA'04. Boston, USA.

[Ruder, 2004] Ruder A.. 2004. UML-based Test Generation and Execution. Rückblick Meeting. Berlin.

[Ryser et-al, 2003] Ryser J., Glinz M. 2003. Scent: A Method Employing Scenarios to Systematically Derive Test Cases for System Test. Technical Report 2000/03, Institut für Informatik, Universität Zürich.

[SWEBOK, 2004] Varios autores. 2004. SWEBOK. Guide to the Software Engineering Body of Knowledge. IEEE Computer Society.

[Schmidt, 2006] Schmidt, D.C. 2006. Guest Editor's Introduction: Model-Driven Engineering. Computer. Volume: 39 Issue: 2

[Thiry et-al, 2009] Thiry L., Thirion B. 2009. Functional Metamodels for Systems and Software. Accepted Manuscript for Journal of systems and Software.

Apéndice A. Glosario

En este primer anexo se ofrecen, a modo de referencia de consulta, definiciones breves para algunos de los términos técnicos utilizados durante este trabajo de tesis.

Caso de prueba funcional del sistema:

En el contexto de este trabajo de tesis, un caso de prueba funcional del sistema es un conjunto de pasos, resultados esperados y valores de prueba para la verificación de un fragmento del comportamiento funcional del sistema.

Categoría:

Según la definición del Método Categoría-Partición, una categoría es un parámetro o una característica de un elemento del entorno. De manera global, en este trabajo de tesis se llama característica a cualquier elemento de la funcionalidad del sistema que puede variar.

Escenario de prueba funcional del sistema:

Conjunto de pasos y resultados esperados para la verificación de un fragmento del comportamiento de un requisito funcional.

Mapeo:

Este término es la traducción al español utilizado en esta tesis del término *mapping operation,* definido en QVT como una operación que implementa una parte de una transformación.

MDE:

Siglas de Model Driven Engineering. Estas siglas denotan una filosofía de desarrollo guiada por modelos, en la cual, el principal objetivo de una fase (análisis, diseño, etc.) es desarrollar los modelos adecuados a partir del refinamiento de los modelos obtenidos en la fase anterior.

Metamodelo:

En el contexto de este trabajo un metamodelo es la definición de una gramática (elementos y reglas de construcción) con la que poder elaborar modelos que sean conformes a dicho metamodelo.

Modelo:

En el contexto de este trabajo, un modelo es una representación mediante un lenguaje concreto, con un mayor o menor grado de abstracción, de un aspecto de un sistema de información. Por ejemplo, un modelo de pruebas será la representación de un conjunto de pruebas a realizar sobre el sistema.

NDT:

Navigational Development Technique es el nombre de la propuesta para la elicitación de requisitos funcionales tomada como referencia en este trabajo de tesis.

Partición:

Este concepto está presente tanto en el Método de Categoría-Partición como en el perfil de prueba de UML. Este concepto está asociado al concepto de categoría o de variable operacional. Una partición es un subconjunto del rango de valores posibles que puede tomar una categoría o variable operacional y que tiene la condición de que cualquier elemento del subconjunto produce el mismo resultado.

QVT:

Query View Transform es el nombre del lenguaje de transformaciones de modelo a modelo definido por la OMG y utilizado en este trabajo de tesis.

Requisito funcional:

Este término define un comportamiento realizado por el sistema en colaboración con uno o más actores externos. En el contexto de este trabajo, los requisitos funcionales se utilizan como información de entrada del proceso de generación de pruebas funcionales del sistema.

Sintaxis abstracta:

En este trabajo de tesis, este término es sinónimo del termino metamodelo y se utiliza con el mismo significado.

Sintaxis concreta:

En este trabajo de tesis, este término es sinónimo del termino modelo y se utiliza con el mismo significado.

Transformación:

En el contexto de este trabajo una transformación es una especificación de cómo construir un modelo conforme con un metamodelo, tomando como entrada otro modelo conforme a otro metamodelo, que puede ser el mismo o distinto.

UML:

Unified Modelling Language es un lenguaje de modelado gráfico, utilizado en este trabajo principalmente para definir los metamodelos y modelos presentados.

Valor de prueba funcional del sistema:

En este trabajo de tesis, un valor de prueba es una información necesaria en un caso de prueba funcional del sistema

Variable operacional:

En este trabajo de tesis, este concepto es un sinónimo de categoría aplicado a un requisito funcional. Es decir, una variable operacional define un punto que puede variar entre varias realizaciones del requisito funcional.

Apéndice B. Manual de Referencia y Caso Práctico de MDETest

Este anexo se divide en dos partes. En la primera, se describe el uso de la herramienta MDETest. Esta herramienta se presentó en el capítulo VII. En dicho capítulo se expuso que la implementación actual cuenta con una sintaxis concreta para requisitos funcionales basados en requisitos funcionales y diagramas de actividades elaborados en la herramienta UML Enterprise Architect No se ha implementado una sintaxis concreta de salida, sino que, en su lugar, la herramienta realiza un volcado de texto con los metamodelos y artefactos resultantes.

En la segunda parte se expone un caso práctico desarrollado con la herramienta MDETest.

La organización de este anexo se describe a continuación. La primera sección describe el uso de la herramienta desde una perspectiva de usuario. La segunda sección describe brevemente cómo ampliar la herramienta para incluir soporte a nuevas sintaxis concretas, tanto de requisitos funcionales como de pruebas funcionales del sistema. Finalmente, la tercera sección describe el caso práctico.

1. Manual de referencia de MDETest

La herramienta actual ofrece una interfaz en modo línea de comandos. Las opciones de línea de comandos se muestran a continuación en la tabla B.1 y se describe en los siguientes párrafos.

Tabla B.1. Opciones de la herramienta.

Opción	Descripción
[Fichero]	Nombre y ruta del archivo de Enterprise Architect que contiene los requisitos funcionales a partir de las cuáles se generarán las pruebas.
[Paquete base]	Nombre del paquete a partir del cuál se buscarán los requisitos.
[Actores base]	Nombre del paquete a partir del cuál se buscarán los actores.

Las dos primeras opciones son obligatorias. Si los actores del modelo de requisitos funcionales están definidos en los mismos paquetes que los propios requisitos funcionales, entonces no es necesario indicar el nombre del paquete de actores, pero si están definidos en un conjunto de paquetes separados, entonces sí es necesario indicar el nombre de dicho paquete. A continuación, en la tabla B.2 se muestran varios ejemplos de uso.

Tabla B.2. Ejemplos de uso de la herramienta.

	Ejemplo
1	java –jar MDETest.jar mdetest. `MDETestCommandLine proyecto.eap requisitos`
2	java –jar MDETest.jar mdetest. `MDETestCommandLine c:/proyecto.eap requisitos`
3	java –jar MDETest.jar mdetest `MDETestCommandLine c:/proyecto.eap requisitos actores`

El primer ejemplo indica a la herramienta que genere las pruebas funcionales del sistema a partir de los requisitos funcionales que encuentre en el paquete “requisitos” y en todos los paquetes contenidos en éste, del fichero proyecto.eap, el cual se encuentra en la misma carpeta que la herramienta. El segundo ejemplo indica lo mismo que el primer ejemplo, pero en este caso el archivo EAP se encuentra en la raíz de la unidad C. Finalmente, el tercer ejemplo indica lo mismo que el segundo pero, en este caso, los actores se van a buscar no en el paquete “requisito”, sino en el paquete “actores” y en todos los paquetes que contenga.

2. Desarrollo de nuevas sintaxis concretas en la herramienta

Es posible extender la herramienta MDETest para que trabaje con nuevas sintaxis concretas. Es necesario que la información gestionada por una sintaxis concreta sea coherente con los metamodelos presentados en este trabajo de tesis, es decir, debe poderse obtener una instancia de los metamodelos vistos en los capítulos anteriores a partir de la información recogida en la sintaxis concreta. Actualmente, la manera de extender la herramienta MDETest es mediante la modificación de su código fuente. Dicha modificación se explica a continuación, en los siguientes párrafos.

La herramienta, en su versión actual, tiene un único punto de entrada y tres puntos de salida. El único punto de entrada acepta el modelo de requisitos funcionales que se va a utilizar como modelo origen de la transformación. Los tres puntos de salida, en cambio, ofrecen cada uno de los modelos de los tres grupos lógicos obtenidos al aplicar las transformaciones.

Un ejemplo de código que genera un conjunto de pruebas funcionales del sistema a partir de un requisito funcional se muestra a continuación, en la tabla B.3.

Tabla B.3. Ejemplo de generación de pruebas funcionales del sistema a partir del código Java.

```
public static void main(String[] args) {
 // (1)
 FunctionalRequirementsModel frm= tl.readFunctionalRequirementsModel("ar.eap",
                      "Subsistema principal");

 FunctionalRequirements2Paths fr2p =  new FunctionalRequirements2Paths();
 fr2p.transform(frm);

 FunctionalRequirements2TestScenarios f2t= new  FunctionalRequirements2TestScenarios();
    TestScenariosModel tsm = f2t.transform(frm);

    FunctionalRequirements2TestValues fr2tv = new FunctionalRequirements2TestValues();
    TestValuesModel tvm = fr2tv.transform(frm);

    TestScenarioTestValue2TestCase tstv2tc = new TestScenarioTestValue2TestCase();
    TestCasesModel tcm = tstv2tc.transform(frm, tsm, tvm);

    // (2)
}
```

Los modelos, en la herramienta MDETest, están definidos mediante un conjunto de objetos que instancian clases Java que representan los conceptos definidos en los metamodelos vistos en los capítulos anteriores. A modo de ejemplo, se muestra a continuación, en la figura B.1, el modelo de clases correspondientes a la implementación en la herramienta del metamodelo de requisitos funcionales del sistema.

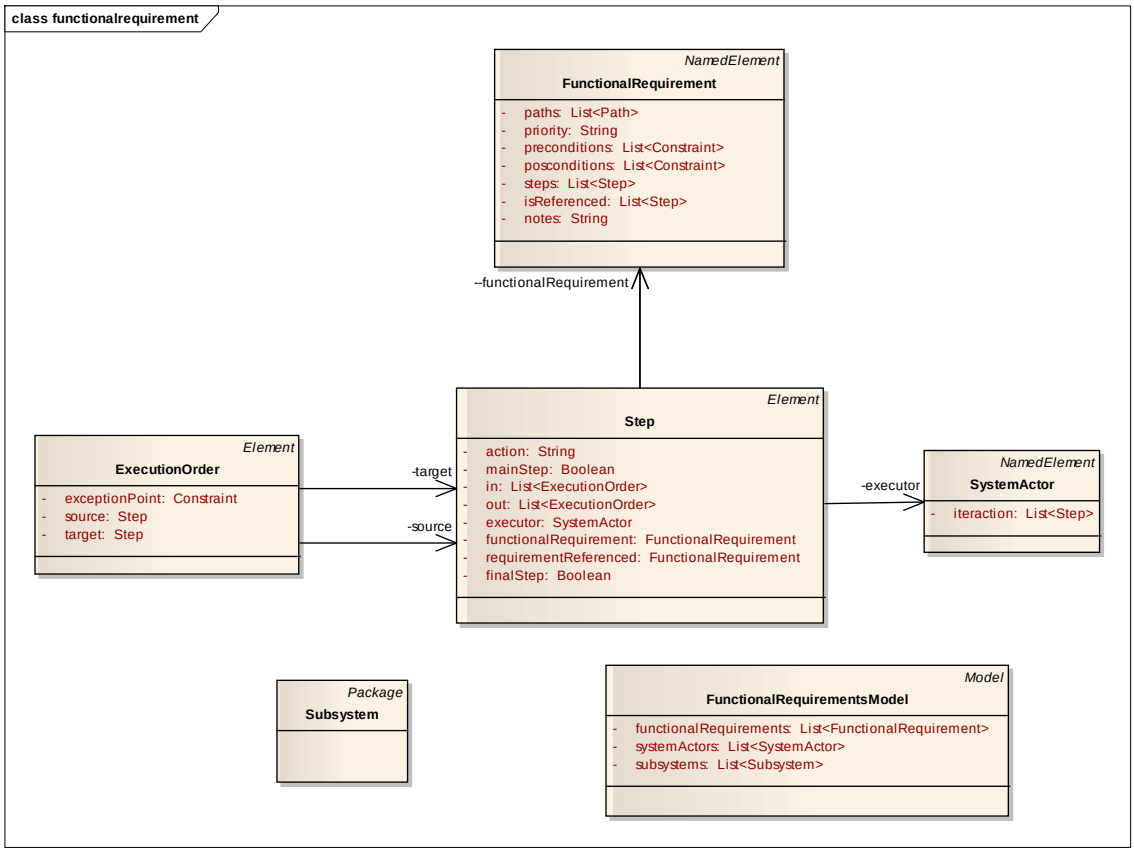


Figura B.1. Modelo de requisitos funcionales implementado en código Java en la herramienta MDETest.

El desarrollo de nuevas sintaxis concretas consiste, por tanto, en el desarrollo de un modelo de código que, o bien permita construir un objeto que implemente la interfaz *FunctionalRequirementModel* o bien permita construir distintos artefactos a partir de los objetos que implementan las interfaces *TestScenariosModel*, *TestValuesModel*, y *TestCasesModel*.

En el código de ejemplo de la tabla B.3 se han marcado los dos puntos (punto (1) y punto (2)) que indican el lugar donde se debe colocar el código para manejar una sintaxis concreta de requisitos funcionales (punto (1)) y el código para manejar sintaxis concreta para artefactos de pruebas funcionales del sistema (punto (2)).

En concreto, en el punto (1) sería donde se crearía una instancia del metamodelo de requisitos funcionales a partir de la sintaxis concreta de requisitos funcionales. En el ejemplo de código de la tabla B.3, la línea que hay justo debajo del comentario (1) es la que se encarga de procesar la sintaxis concreta y elaborar una instancia del metamodelo de requisitos funcionales que contiene la misma información. Si se deseara utilizar otra sintaxis concreta, simplemente habría que cambiar dicha línea por el mecanismo que fuera capaz de crear una instancia a partir de dicha sintaxis concreta.

En el punto (2) del código de la tabla B.3, sería donde se crearían sintaxis concretas de los artefactos de prueba a partir de los objetos *tsm*, *tvm* y *tcm*, de manera análogo a lo visto para el punto (1).

3. Caso práctico

En esta sección se presenta un caso práctico que se ha desarrollado con la herramienta MDETest. La sección 3.1 describe el caso práctico. La sección 3.2 aplica la herramienta MDETest al proceso y describe los resultados.

3.1. Descripción del caso práctico

El caso práctico está basado en el desarrollo de un sistema de gestión hotelera para un hotel ficticio llamado Ambassador. Este proyecto ha sido desarrollado utilizando NDT y sus herramientas de soporte, incluyendo la herramienta Enterprise Architect y, actualmente es un ejemplo de referencia en la aplicación de esta propuesta metodológica.

Para este proyecto se han desarrollado todas las fases recogidas en la metodología NDT, sin embargo la herramienta solo utiliza los dieciocho requisitos funcionales definidos en la fase de requisitos del sistema. Los dieciocho requisitos funcionales se agrupan en tres subsistemas, mostrados en la figura B.2.

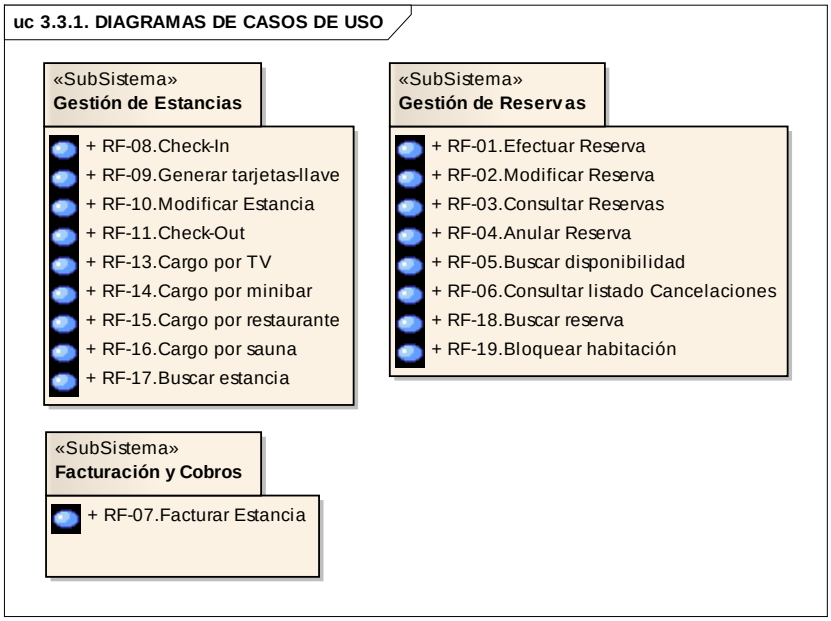


Figura B.2. Requisitos funcionales del caso práctico Hotel Ambassador.

Como se ha visto con anterioridad, para poder aplicar la herramienta MDETest es necesario que el comportamiento de los requisitos funcionales esté definido mediante un diagrama de actividades.

La figura B.3 muestra un ejemplo de diagrama de actividades de este caso práctico, en concreto el diagrama de actividades que describe el comportamiento del requisito funcional RF-01. Efectuar Reserva.

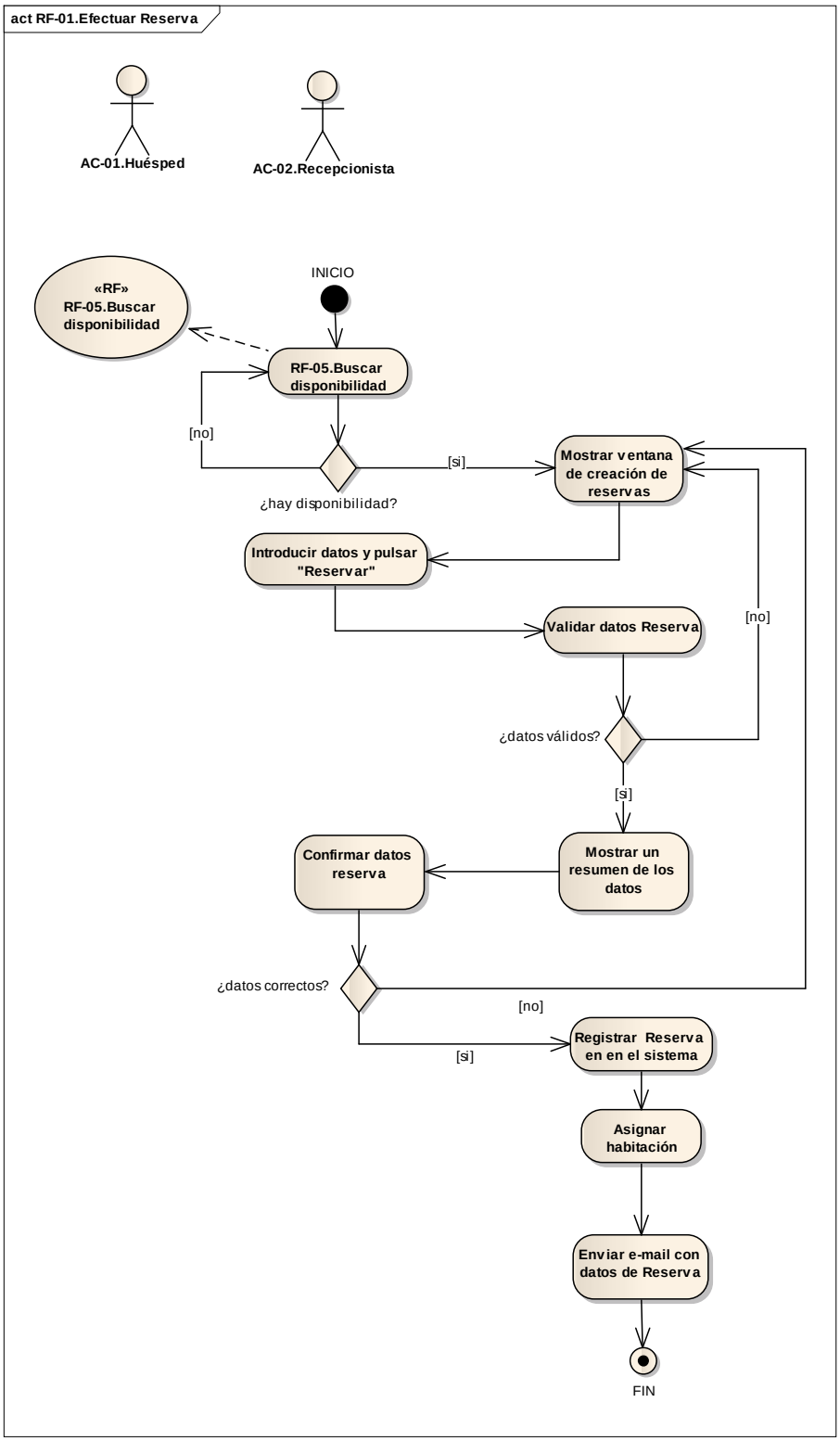


Figura B.3. Diagrama de actividades.

Tabla B.4. Estadística del modelo de requisitos funcionales del caso práctico.

Descripción	Valor
Requisitos funcionales	19
Diagramas de actividades	9
Requisitos funcionales con diagramas de actividades	RF-01.Efectuar Reserva RF-02.Modificar Reserva RF-03. Consultar Reservas RF-04. Anular Reserva RF-05. Buscar disponibilidad RF-07. Facturar estancia RF-08. Check-in RF-09. Generar tarjetas-llave RF-10.Modificar Estancia
Actores del sistema	6
Paquetes de requisitos funcionales	3.3. REQUISITOS FUNCIONALES
Paquetes de actores del sistema	3.2. DEFINICIÓN DE ACTORES

Para terminar, la tabla B.4 describe las estadísticas del modelo de requisitos funcionales de este caso práctico. A continuación se documenta el uso de la herramienta MDETest para la generación de pruebas funcionales del sistema.

3.2. Aplicación de la herramienta

Con la información descrita en la sección anterior, se ha ejecutado la herramienta utilizando la sintaxis de línea de comandos presentada al principio de este anexo. A continuación se resumen los resultados obtenidos de cada una de las transformaciones.

La primera transformación es la que permite enriquecer el modelo de requisitos funcionales del sistema con los caminos. La tabla B.5, resume los resultados obtenidos por la herramienta al aplicar esta transformación.

Tabla B.5. Resumen de los caminos obtenidos.

Descripción	Resultado
Caminos totales obtenidos	38
Caminos para RF-01.Efectuar Reserva	16
Caminos para RF-02.Modificar Reserva	7
Caminos para RF-03. Consultar Reservas	2
Caminos para RF-04. Anular Reserva	2
Caminos para RF-05. Buscar disponibilidad	2
Caminos para RF-07. Facturar estancia	2
Caminos para RF-08. Check-in	2
Caminos para RF-09. Generar tarjetas-llave	2
Caminos para RF-10.Modificar Estancia	3

A continuación, a partir del modelo de requisitos funcionales obtenido de la transformación anterior, se aplica la transformación para generar los escenarios de prueba. La tabla B.6, resume los resultados obtenidos por la herramienta al aplicar esta transformación.

Tabla B.6. Resumen de los escenarios de prueba obtenidos.

Descripción	Resultado
Escenarios de prueba	38
Actores del escenarios de prueba	6
Paquete de escenarios de prueba	6
Escenarios de prueba para RF-01.Efectuar Reserva	16
Escenarios de prueba para RF-02.Modificar Reserva	7
Escenarios de prueba para RF-03. Consultar Reservas	2
Escenarios de prueba para RF-04. Anular Reserva	2
Escenarios de prueba para RF-05. Buscar disponibilidad	2
Escenarios de prueba para RF-07. Facturar estancia	2
Escenarios de prueba para RF-08. Check-in	2
Escenarios de prueba para RF-09. Generar tarjetas-llave	2
Escenarios de prueba para RF-10.Modificar Estancia	3

Como se puede comprobar en la tabla B.6, el número de escenarios de prueba de cada caso de uso coincide con el número de caminos de cada caso de uso, ya que se ha generado un escenario para probar cada uno de los caminos. Además, se ha generado un escenario de prueba por cada requisito funcional que no tenía ningún camino.

A continuación, en la tabla B.7, se muestra un ejemplo concreto, de un escenario de prueba funcional del sistema generado por la herramienta MDETest.

Tabla B.7. Ejemplo de escenario de prueba generado por MDETest.

```
finalState=[],
initialState=[],
notes=null,
testSteps=[
TestVerificationStep [body=RF-17.Buscar estancia],
TestVerificationStep [body=¿existe estancia?],
TestVerificationStep [body=Mostrar datos estancia],
TestVerificationStep [body=Modificar datos estancia],
TestVerificationStep [body=Validar datos],
TestVerificationStep [body=RF10-¿son datos válidos?],
TestVerificationStep [body=Mostrar datos estancia],
TestVerificationStep [body=Modificar datos estancia],
TestVerificationStep [body=Validar datos],
TestVerificationStep [body=RF10-¿son datos válidos?],
TestVerificationStep [body=Actualizar datos estancia],
TestVerificationStep [body=End.]],
getDescription()=Escenario El sistema deberá comportarse tal y como se
describe en el siguiente caso de uso y que representa la funcionalidad
del sistema que permite que los datos de una estancia.
getInPackage()=null,
getName()=RF-10.Modificar Estancia]
```

Tabla B.8. Resumen de los valores de prueba obtenidos.

Descripción	Resultado
Variables operacionales:	15
Instancias de VO:	24
Combinaciones de instancias de Variables Operacionales:	38
Variables operacionales para RF-01.Efectuar Reserva	3
Variables operacionales para RF-02.Modificar Reserva	4
Variables operacionales para RF-03. Consultar Reservas	1
Variables operacionales para RF-04. Anular Reserva	1
Variables operacionales para RF-05. Buscar disponibilidad	1
Variables operacionales para RF-07. Facturar estancia	1
Variables operacionales para RF-08. Check-in	1
Variables operacionales para RF-09. Generar tarjetas-llave	1
Variables operacionales para RF-10.Modificar Estancia	2

A continuación, y también a partir del modelo de requisitos funcionales enriquecido, se aplica la transformación para generar los valores de prueba del sistema. La tabla B.8, resume los resultados obtenidos por la herramienta al aplicar esta transformación.

Como se puede ver en la tabla anterior, de nuevo se ha generado una combinación de instancias de valores de prueba por cada camino. No se ha podido generar ningún artefacto en esta transformación para aquellos requisitos funcionales que carecían de comportamiento. Además, dado que cada nodo decisión de los diagramas de actividades tiene dos flujos de control como salida, todas las variables operacionales generadas en la transformación cuentan con dos particiones.

A continuación, en las tablas B.9 y B.10 se muestra un ejemplo de un conjunto de valores de prueba funcional del sistema generado por la herramienta MDETest.

Tabla B.9. Ejemplo de variable operacional generado por MDETest.

```
name=> V1<. ,
partitions=[
Particion: > V1<. ::[[value=si]],
Particion: > V1<. ::[[value=no]]
]]
Example
```

Tabla B.10. Ejemplo de combinación de instancias generado por MDETest.

```
InstanceCombination [
quantums=[
1 > V1<. =Particion: > V1<. ::[[value=si]],
1 > V2<. =Particion: > V2<. ::[[value=no]],
2 > V2<. =Particion: > V2<. ::[[value=si]],
1 > V3<. =*::[],
2 > V3<. =*::[],
1 > V4<. =*::[],
2 > V4<. =*::[],
3 > V4<. =*::[],
4 > V4<. =*::[],
1 > V5<. =*::[],
2 > V5<. =*::[],
3 > V5<. =*::[],
1 > V6<. =*::[],
1 > V7<. =*::[],
2 > V7<. =*::[],
1 > V8<. =*::[],
1 > V9<. =*::[],
2 > V9<. =*::[]]]
```

Como se puede apreciar en la tabla B.10, en esa combinación concreta sólo toman valores la primera instancia de la variable V1 y las dos primeras instancias de la variable V2. Las demás instancias no participan y, por tanto, no toman valor, lo que se indica con un "*"

Tabla B.11. Resumen de los valores de prueba obtenidos.

Descripción	Resultado
Casos de prueba:	38
Actores de casos de prueba:	6
Colecciones de casos de prueba:	6

Para finalizar, y a partir de todos los modelos anteriores, se aplica la transformación para generar los casos de prueba del sistema. La tabla B.11 resume los resultados obtenidos por la herramienta al aplicar esta transformación.

Como se puede ver en la tabla B.8, los resultados son análogos a los resultados obtenidos para los escenarios de prueba ya que los casos de prueba son un refinamiento de estos artefactos.

A continuación se muestra un ejemplo de un caso de prueba funcional del sistema generado por la herramienta MDETest.

Tabla B.12. Ejemplo de caso de prueba generado por MDETest.

```
TestCase::RF-02.Modificar Reserva
InitialState:[],
FinalState:[],
testInfo=[InstanceCombination [quantums=[
1 > V6<. =Particion: > V6<. ::[[value=no]],
1 > V1<. =*::[],
1 > V2<. =*::[],
2 > V2<. =*::[],
1 > V3<. =*::[],
2 > V3<. =*::[],
1 > V4<. =*::[],
2 > V4<. =*::[],
3 > V4<. =*::[],
4 > V4<. =*::[],
1 > V5<. =*::[],
2 > V5<. =*::[],
3 > V5<. =*::[],
1 > V7<. =*::[],
2 > V7<. =*::[],
1 > V8<. =*::[],
1 > V9<. =*::[],
2 > V9<. =*::[]]]]]
```

```
steps:4
RF-18.Buscar reserva
¿existe reserva?
Test data:[1 > V6<. ]
Mostrar mensaje informativo
End.
```

Posteriormente a este caso práctico, en la siguiente sección, se exponen las conclusiones de este anexo.

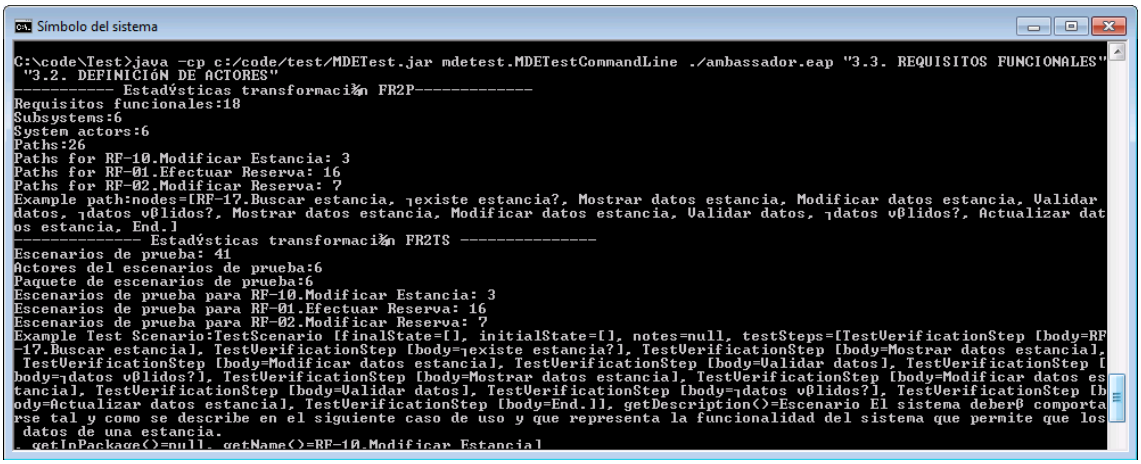


Figura B.4. Captura de pantalla de la herramienta METest.

Finalmente, la figura B.4 muestra una captura de pantalla de la herramienta durante la ejecución de este caso práctico.

4. Conclusiones

Ha sido necesario realizar algunas pequeñas modificaciones a la herramienta para poder desarrollar este caso práctico. En concreto, la herramienta no había sido desarrollada contemplando la posibilidad de trabajar con una estructura de paquetes para casos de uso y otra para actores del sistema, por lo que fue necesario realizar una pequeña modificación. Otra modificación necesaria permitió que la herramienta pudiera trabajar con requisitos funcionales sin comportamiento, ya que esa posibilidad no había sido contemplada en la primera implementación de las transformaciones.

Si bien fue necesario hacer modificaciones muy pequeñas a la herramienta, no ha sido necesario modificar nada del caso práctico, a pesar de que este caso práctico ha sido realizado

por personas ajenas a este trabajo de tesis, en un tiempo anterior a la codificación de la herramienta que implementa el proceso descrito en los capítulos anteriores.

Este caso práctico viene a justificar que sí es posible automatizar la generación de pruebas funcionales del sistema a partir de sus requisitos funcionales, gracias a la formalización del proceso, en este caso, mediante metamodelos y transformaciones.

Apéndice C. Perfiles de UML

Este anexo describe dos perfiles de UML. El primero de ellos permite extender UML para recoger los artefactos y la semántica definidos en el metamodelo de requisitos funcionales. El segundo de ellos permite extender UML para recoger los artefactos definidos en los tres grupos lógicos de artefactos de pruebas.

La organización de este anexo se describe a continuación. La sección 1 presenta el perfil UML de requisitos funcionales. A continuación, la sección 2 presenta el perfil UML del metamodelo de pruebas funcionales del sistema. Por último, la sección 3 presenta las conclusiones de este anexo.

1. Perfil UML de requisitos funcionales

En [Giachetti et-al, 2008] y en [Fuentes et-al, 2004] se propone una guía para la elaboración de perfiles UML. Ambas apuestan por desarrollar, en primer lugar, un metamodelo con los conceptos semánticos para, a continuación, establecer la correspondencia de dichos conceptos con las clases UML y elaborar el perfil. En este caso concreto, el metamodelo con los conceptos semánticos se corresponden con los metamodelos presentando en los capítulos anteriores. En esta sección se realiza una extensión a UML mediante un perfil que incluirá dichos conceptos. Dado que un perfil de UML enriquece semánticamente constructores ya definidos en UML, en primer lugar se identificarán cuáles son los elementos de UML adecuados para representar los conceptos definidos en el metamodelo de la sección anterior. Por ello, la sección 1.1 resume la definición de los elementos de UML tomados como base para el perfil UML de requisitos funcionales. A continuación, en la sección 1.2, se definen los estereotipos necesarios que extenderán dichos elementos para representar los conceptos e información del metamodelo de requisitos funcionales.

1.1. Elementos de UML

Es muy importante identificar cuáles van a ser las clases de UML base sobre la que se van a aplicar los estereotipos que definan los distintos conceptos del metamodelo de requisitos funcionales definidos en la sección anterior. Estas clases se exponen en la tabla C.1 y en la figura C.1. Se incluye en la tabla C.1 el concepto del metamodelo de requisitos funcionales que estará asociado a dicho elemento.

Tabla C.1. Elementos del metamodelo de UML tomados como base del perfil.

Elemento UML	Descripción	Concepto del metamodelo
Actor	Descripción de un rol que interactúa con el sistema.	ActorDelSistema
Constraint	Definición de una condición o restricción	Restricción
Package	Agrupación de elementos bajo un espacio de nombre	Subsistema
State	Situación durante la que se cumple una invariante	Paso
Transition	Relación dirigida entre un vértice origen y u vértice destino	Orden de ejecución.
UseCase	Comportamiento realizado por el sistema.	RequisitoFuncional

Al igual que en la descripción del metamodelo de requisitos funcionales, la figura C.1 escribe de manera gráfica la sintaxis abstracta de los elementos de UML utilizados como base para el perfil de requisitos de sistemas que se desarrolla en esta sección. En esta figura se pueden ver las principales asociaciones y atributos que tienen los elementos utilizados y que se describirán en más detalle a lo largo de esta sección. Es interesante destacar que UML es un lenguaje de propósito muy general y flexible por lo que se hace necesario acotar los elementos y atributos a utilizar. Además, se han omitido de la figura C.1 las relaciones jerárquicas con las superclases de las clases utilizadas siempre que ha sido posible, ya que no se utilizan directamente, su semántica no es relevante y los elementos necesarios ya se describen en la propia clase.

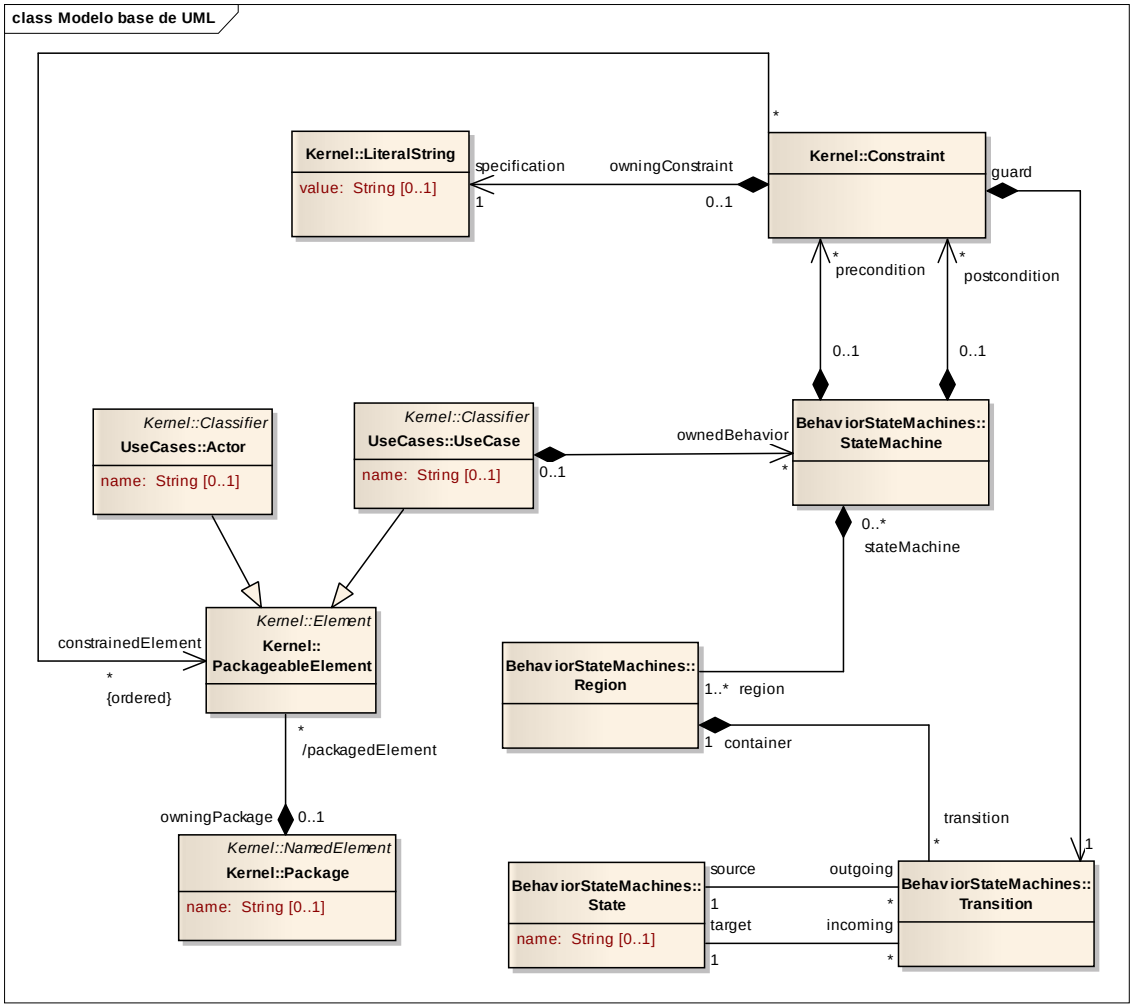


Figura C.1. Modelo de los elementos de UML utilizados en el perfil.

UML permite asociar máquinas de estados a un caso de uso (la manera de hacerlo se muestra en la figura C.1, mediante la relación con el rol *ownedBehaviour* (del elemento *UseCase*) y se explica con más detalle a continuación), por tanto, usar una máquina de estados ha sido la opción elegida para representar en UML el comportamiento de un requisito funcional.

Por último, es conveniente destacar que UML ofrece un lenguaje muy genérico, por lo que, en aras de fomentar la flexibilidad, UML cuenta con una jerarquía de clase profusa y poblada, por lo que puede resultar complejo representar un fragmento de la misma. Por ejemplo, solo las clases *UseCase* y *Actor* generalizan de 11 clases cada una, siendo una de ellas repetida debido a herencia múltiple. Por tanto, para facilitar la compresión, se han omitido de las descripciones de los elementos definidas a continuación aquellos aspectos, atributos, métodos, restricciones, superclases, etc. que no son estrictamente imprescindibles para cada elemento. Para facilitar la trazabilidad de los elementos a través de la jerarquía de clases, se ha anotado la procedencia de los atributos y asociaciones incluidas en las siguientes

definiciones. Es interesante aclarar que en esta simplificación, en ningún momento se realiza ningún cambio o modificación a lo definido en la documentación de UML (lo que no está permitido en un perfil).

Sin embargo, ha sido imposible evitar añadir algunas clases adicionales a las clases con las que se modelan los conceptos del metamodelo para poder utilizar algunos de sus elementos. Estas clase son: *LiteralString*, *PackageableElement, Region* y *StateMachine* (figura C.1) y no se describen en la siguientes secciones porque no son relevantes por sí mismas, sino que se describen como parte de los elementos a los que complementan, como se verá con más detalle justo a continuación.

1.1.1. Actor

El elemento *Actor* de UML define un papel desempeñado por una persona u otro sistema externo que interactúa con el sistema bajo estudio.

Atributos

- name: String[0..1]
  Nombre del actor (atributo heredado de la clase *Kernel::NamedElement*).

Restricciones

[1] name → notEmpty()

Semántica

La restricción [1], definida en el metamodelo de UML e incorporada textualmente en este trabajo de tesis, impone que toda instancia de la clase Actor tenga un valor para su atributo *name*.

1.1.2. Constraint

El elemento *Constraint* de UML define una única condición o restricción expresada en lenguaje natural, OCL, o en un lenguaje de programación. Una restricción es un predicado booleano que, por tanto, se evalúa a cierto o a falso.

Asociaciones

- specification: LiteralString[0..1]

Definición del predicado a evaluar.

- constrainedElement: Element[*]
  Conjunto de elementos a los que se aplica una restricción.

Restricciones

[1] El valor de *specification* debe poder evaluarse para obtener un resultado booleano (esta restricción no puede expresarse en OCL).

Semántica

Para el perfil de UML de requisitos funcionales, se ha optado por el lenguaje natural para la definición de las restricciones, por lo que el atributo *specification* es de tipo *LiteralString* (en la especificación original, la asociación *specification se establece con un elemento ValueSpecificaction*, de la que *LiteralString* es una especialización, sin embargo como solo se va a utilizar *LiteralString,* se ha utilizado directamente este elemento). La clase *LiteralString* funciona de manera análoga al tipo de datos básico *String* definido en UML y OCL.

La restricción [1] ha sido tomada de la especificación de UML e indica que la cadena de texto no puede contener cualquier valor, sino que debe contener un texto evaluable como predicado booleano.

1.1.3. Package

El elemento *Package* de UML define un grupo de elementos y un espacio de nombres para dichos elementos.

Atributos

- name: String[0..1]
  Nombre del paquete (atributo heredado de la clase *Kernel::NamedElement*).

Asociaciones

- /packagedElement: PackageableElement[*]
  Elementos contenidos en este paquete. Esta relación puede derivarse a partir de las relaciones de su superclase *NamedElement*

Semántica

Todo elemento que derive de *PackableElement* puede estar almacenado en un paquete. Como se puede apreciar, la asociación *packagedElement* es derivada, ya que es un subconjunto de una asociación de su superclase (en concreto de los elementos que pertenecen a un *NameSpace*, ya que un paquete es una especialización de un espacio de nombres). Sin embargo en este trabajo se ha optado por no utilizar el mecanismo de espacio de nombres (*NameSpace*), ya que incorpora una abstracción y un nivel de flexibilidad que no es necesario, y, en su lugar, se ha utilizado un mecanismo más definido, como son los paquetes.

1.1.4. State

El elemento *State* de UML modela una situación durante la que se cumple una condición que puede ser implícita.

Atributos

- name: String[0..1]
  Nombre del estado (atributo heredado de la clase *Kernel::NamedElement*).

Asociaciones

- outgoing: Transition[0..*]
  Conjunto de transiciones que salen del estado (asociación heredada de la clase *BehaviorStateMachines::Vertex*).

- incoming: Transition[0..*]
  Conjunto de transiciones que llegan al estado (asociación heredada de la clase *BehaviorStateMachines::Vertex*).

Semántica

Una de las posibles condiciones implícitas recogidas en la documentación de UML es la permanencia en un estado mientras se realiza la ejecución de un comportamiento. Ésta será la condición utilizada a la hora de definir el perfil del metamodelo de requisitos funcionales.

1.1.5. Transition

El elemento *Transition* de UML define una relación entre un vértice de origen y un vértice de destino.

Asociaciones

- guard: Constraint[0..1]
  Restricción que controla la realización de la transición.
- source: State[1]
  Origen de la transición.
- target: State[1]
  Destino de la transición.
- container: Region[1]
  Región a la que pertenece la transición.

Semántica

Esta clase también tiene una asociación con la región que contiene. El elemento *Region* representa una agrupación de estados y transiciones (por ejemplo para construir máquinas de estados compuestas de otras máquinas de estados) y, como se ha dicho antes, es un elemento que debe aparecer para poder acceder a los elementos de una máquina de estados (elemento *stateMachine*, tal y como se muestra en la sintaxis abstracta), ya que una máquina de estados tendrá al menos una región que contendrá todos los estados y transiciones, pero el elemento *Region* no aporta nada por sí mismo, por lo que no se ha descrito con más profundidad.

1.1.6. UseCase

El elemento *UseCase* de UML define la especificación de un conjunto de acciones realizadas por el sistema, las cuales generan un resultado observable de valor para uno o más actores del sistema.

Atributos

- name: String[0..1]
  Nombre del caso de uso (atributo heredado de la clase *Kernel::NamedElement*).

Asociaciones

- ownedBehavior: Behavior[*]
  Comportamiento del caso de uso. (asociación heredada de la clase *BasicBehaviors::BehavioredClassifier*)

Restricciones

[1] self.name → notEmpty ()

Semántica

La asociación *ownedBehavior* a un elemento *Behavior* de un *UseCase* incluye sus precondiciones, poscondiciones y, en este caso concreto, una máquina de estados que representa el comportamiento del caso de uso.

La restricción [1] establece que, al igual que el elemento *Actor,* todo caso de uso debe tener un nombre. Es necesario destacar que, tal y como se ha visto en la sintaxis abstracta, es posible definir precondiciones y poscondiciones para un caso de uso mediante su comportamiento (rol *ownedBehavior*)

En UML se definen los puntos de excepción mediante la clase *ExtensionPoint*. Esta clase identifica un punto en el comportamiento de un caso de uso en el cuál dicho comportamiento puede extenderse con el comportamiento de otro caso de uso. Este punto se describe, en UML, sin ningún formalismo en particular, de la misma manera que se podría escribir un comentario. Sin embargo, en el metamodelo de requisitos funcionales se ha definido que el punto de extensión debe ser una restricción booleana, cuya evaluación indicará si se realiza la extensión o no. Por ese motivo la clase *ExtensionPoint* no ha sido utilizada y, en su lugar se ha utilizado la clase *Constraint*.

Para terminar esta sección, se incluyen a continuación algunos comentarios finales sobre el perfil de UML. Como se puede ver, no existe una asociación directa entre los casos de uso y los actores. Esto es así porque dichas asociaciones se modelan a través de un tercer elemento que representa la asociación en sí (de manera similar a cómo se relacionan dos elementos *State* en la figura C.1). Además de las asociaciones vistas en las secciones anteriores, todos los elementos tienen también una asociación con un elemento *Comment*, el cual permite añadir comentarios a cada uno de los elementos mencionados.

1.2. Definición del perfil de requisitos funcionales

En esta sección se definen los estereotipos y restricciones adicionales que componen el perfil de requisitos funcionales del sistema. La relación entre los elementos definidos en el

metamodelo de requisitos funcionales y los estereotipos definidos en el perfil de UML se muestra en la figura C.2.

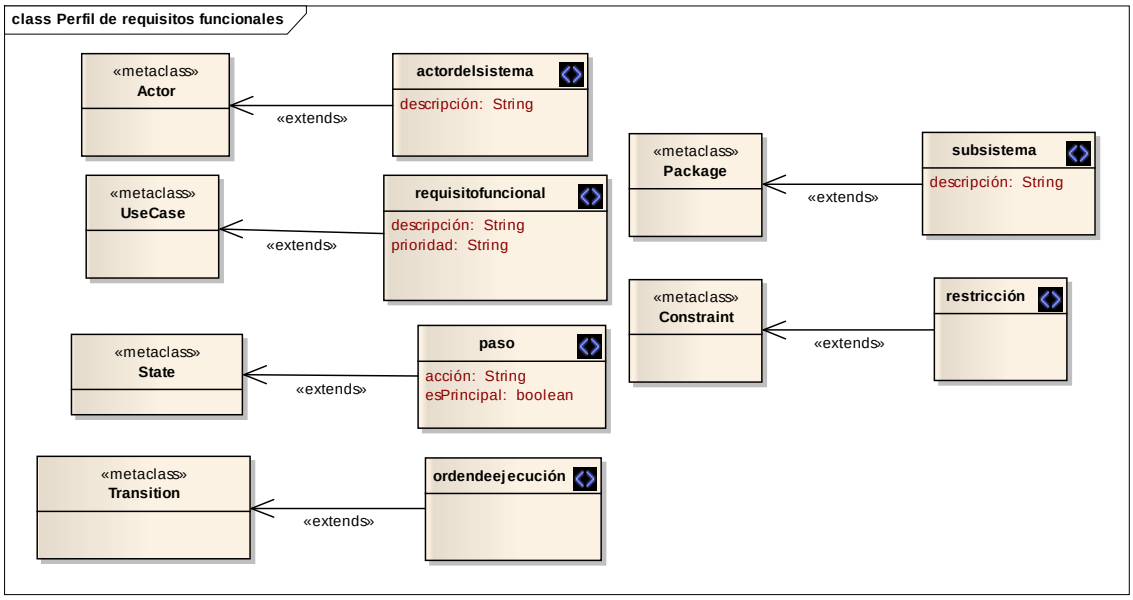


Figura C.2. Perfil del metamodelo de requisitos funcionales.

Como se puede apreciar, se ha utilizado la asociación estereotipada como *extends* (tal y como propone UML) para definir nuevos estereotipos (representados como clases estereotipo) a partir de elementos ya existentes. Es necesario indicar que los atributos de estas clases estereotipos (como el atributo *descripción* de la clase estereotipo *subsistema*) definen valores etiquetados accesibles al estereotipar el elemento original (por ejemplo si en una herramienta UML que contenga este estereotipo creamos un nuevo elemento *package* y lo estereotipamos como *subsistema,* entonces tendremos a nuestra disposición el valor etiquetado *descripción* para definir la descripción del subsistema).

Además, se añaden al perfil las restricciones definidas en la tabla C.2 para adaptar con más precisión la semántica de los constructores de UML a los conceptos vistos en el metamodelo.

Tabla C.2. Restricciones del perfil.

Elemento	Restricción	Descripción
Package	not (self.name→isEmpty())	Todo *Package* debe tener nombre
State	not (self.name→isEmpty())	Todo *State* debe tener un nombre

En la definición de los elementos del perfil (mediante clases estereotipo) solo se han añadido aquellos valores etiquetados necesarios para definir la información del metamodelo de requisitos funcionales de la sección anterior, que no pueden ser definidos con los atributos ya existentes en las clases tomadas como base (y mencionados en las secciones anteriores). Por simplicidad, los atributos existentes en las clases de UML (figura C.1) han sido omitidos de la figura C.2, así como las asociaciones entre los distintos elementos definidos en el metamodelo de requisitos funcionales y las asociaciones entre las clases de UML (ya vistas en la figura C.2).

2. Perfil UML del metamodelo de pruebas funcionales del sistema

Al igual que se ha hecho anteriormente con el metamodelo de requisitos funcionales, en esta sección se realiza una extensión a UML mediante tres perfiles para enriquecer sus elementos con la semántica definida en los tres grupos lógicos presentados en las secciones anteriores.

La manera de proceder será la misma que en el capítulo anterior, por lo que, en primer lugar, se identificarán los elementos de UML que se tomarán como base para el perfil y, después, se añadirán las restricciones, asociaciones y valores etiquetados que sean necesarios para adaptarlo a la semántica de los tres grupos lógicos. En la sección 2.1 se resume la definición de los elementos de UML tomados como base para el perfil. En la sección 2.2, se definen los tres perfiles, uno para cada grupo lógico.

2.1. Elementos de UML utilizados

Como ya se ha visto en el capítulo anterior, para definir un perfil de UML es necesario, como primer paso, identificar el conjunto de elementos de UML que se van a extender mediante los estereotipos definidos en el perfil. Por tanto, en la tabla C.3 se describen los elementos de UML que se han seleccionado por considerarse los más idóneos como base de los tres perfiles de pruebas funcionales del sistema.

Tabla C.3. Elementos del metamodelo de UML tomados como base de los perfiles.

Elemento	Descripción	Concepto representado
Actor	Descripción de un rol que interactúa con el sistema.	*ActorDePrueba*
Class	Definición de patrón para elementos que comparten las mismas características.	*EscenarioDePrueba, VariableOperacional, ParticiónDeDatos, CasodePrueba, Quantum*
Constraint	Definición de una condición o restricción.	*Restricción, CombinaciónDeInstancias*
Operation	Comportamiento para un clasificador	*PasoDePrueba, VerificaciónDePrueba, AcciónDelEscenaioDePrueba, AcciónDePrueba*
Package	Agrupación de elementos bajo un espacio de nombre.	*ColecciónDeCasosDePrueba*

La sintaxis abstracta de los elementos de UML utilizados en el perfil se muestra en la figura C.3. Al igual que en el capítulo anterior, se muestran únicamente los elementos y atributos relevantes para facilitar el entendimiento del perfil de UML.

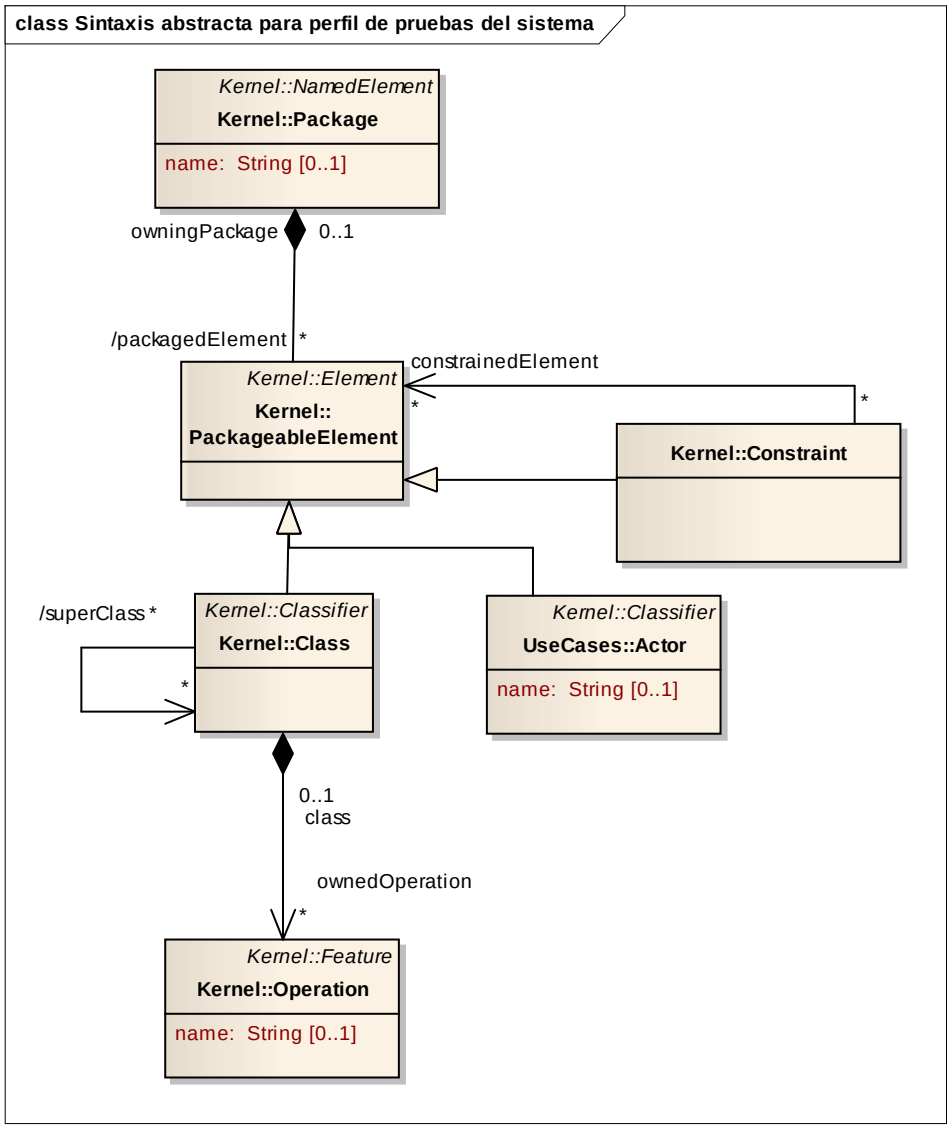


Figura C.3. Sintaxis abstracta de los elementos de UML utilizados en el perfil.

Los elementos *Constraint*, *Actor, PackageableElement* y *Package* (tabla C.3) ya han sido comentados con anterioridad en el capítulo dedicado al metamodelo de requisitos funcionales. En los siguientes párrafos se describen los elementos *Class, Operation* y *Property*, detallando solo lo necesario para el perfil de pruebas funcionales del sistema.

2.1.1. Class

Este elemento define un conjunto de elementos que comparten las mismas características, restricciones y semántica.

Generalizaciones

Kernel::Classifier (se ha omitido de la figura 4).

Atributos

- name: String[0..1]
  Nombre de la clase (atributo heredado de la clase *Kernel::NamedElement*).

Asociaciones

- ownedOperation: Operation[*]
  Operaciones de la clase.

Semántica

Las características de una clase se definen mediante atributos y operaciones. Los atributos relevantes de esta metaclase para el perfil son la lista de atributos y operaciones y el nombre.

2.1.2. Operation

El elemento *Operation* de UML define un comportamiento asociado a un elemento *Classifier* y, por herencia, también a un elemento *Class*.

Generalizaciones

- *Kernel::Feature*

Atributos

- name: String[0..1]
  Nombre de la operación (atributo heredado de la clase *Kernel::NamedElement*).

Semántica

El elemento *Operation* define atributos para manejar la información relevante del comportamiento, tales como si es una operación única, si es una operación que no modifica el estado del elemento sobre el que se invoca o la multiplicidad de los parámetros devueltos. Dado que en los modelos de requisitos funcionales aún es pronto para definir las operaciones del sistema con tanto detalle, dichos atributos no se utilizarán.

2.2. Definición del perfil de pruebas funcionales del sistema

Dado que el metamodelo de pruebas funcionales está dividido en tres grupos lógicos, se ha desarrollado un perfil de UML para cada uno de dichos grupos lógicos A continuación, se definen en la figura C.4, la figura C.5 y la figura C.6 los perfiles UML para cada uno de los grupos lógicos del metamodelo de pruebas funcionales del sistema. En todos los diagramas se han omitido la definición de los atributos de cada estereotipo ya que coinciden con los atributos del elemento del metamodelo del mismo nombre. Por el mismo motivo se han omitido las asociaciones entre los elementos del metamodelo de pruebas funcionales y las subclases de los elementos que son extendidos.

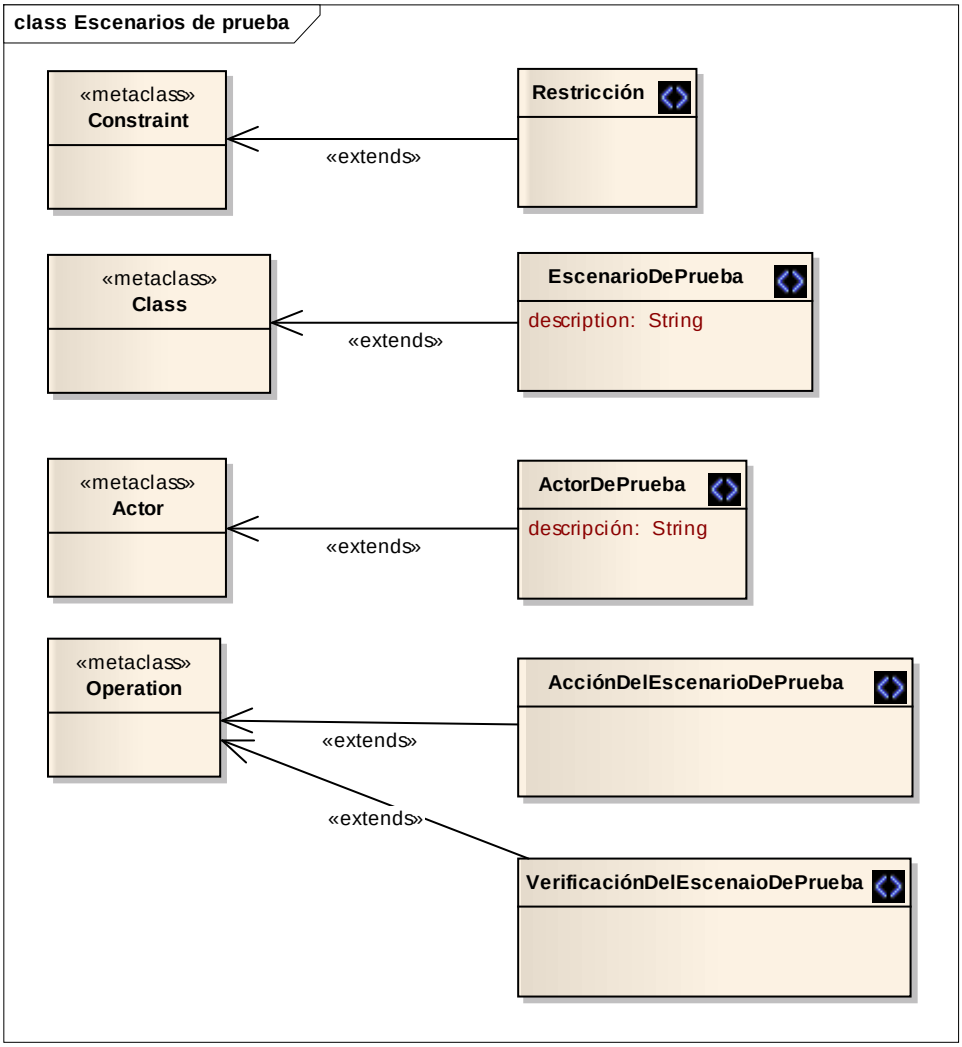


Figura C.4. Perfil para el grupo lógico Escenarios de prueba.

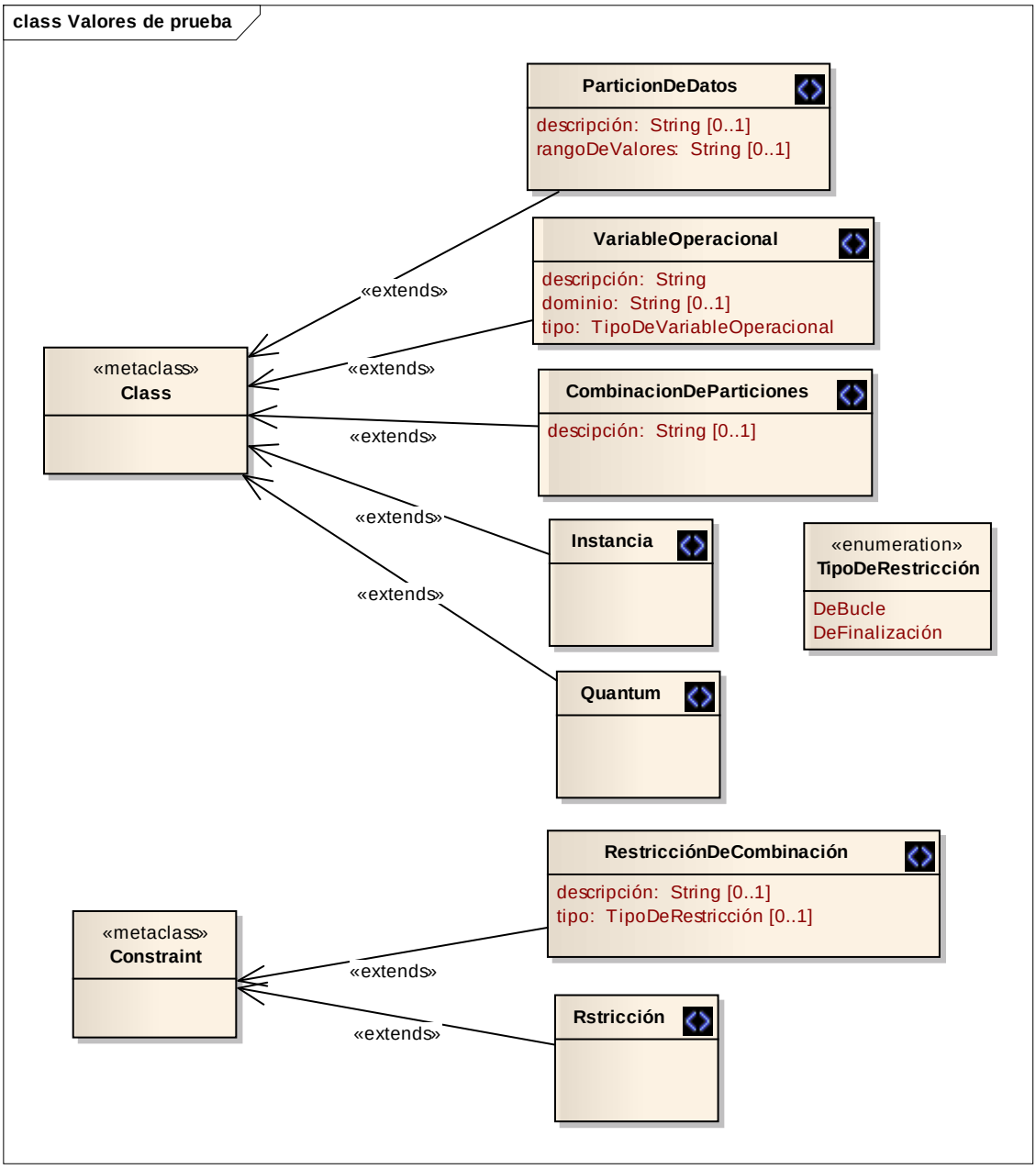


Figura C.5. Perfil para el grupo lógico Valores de Prueba.

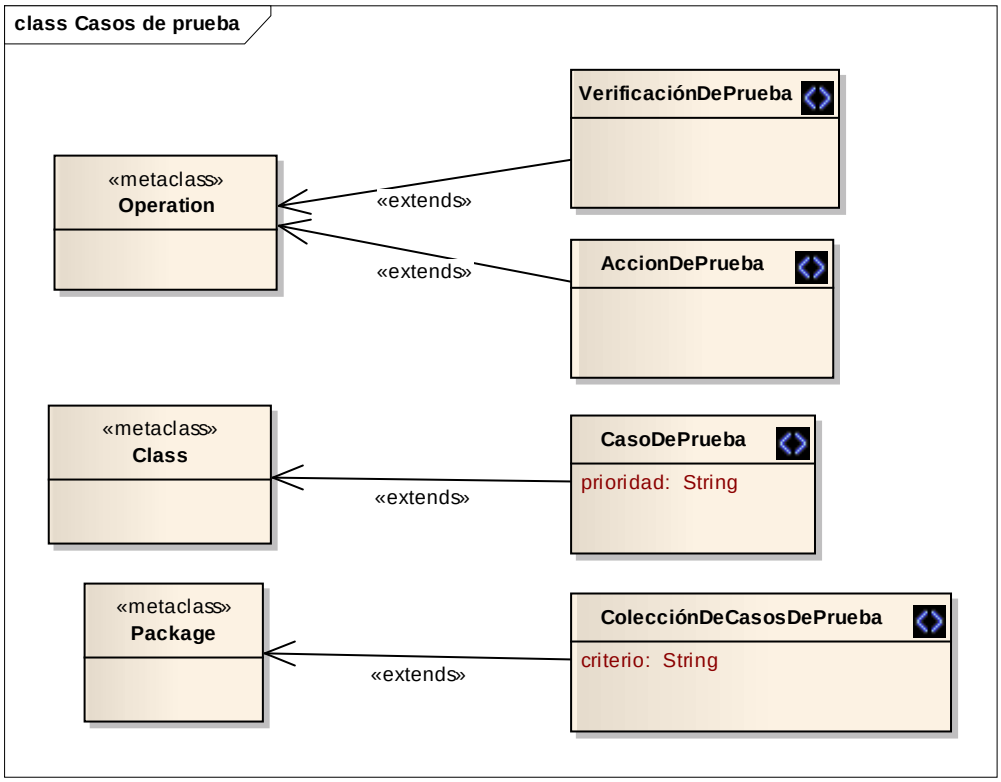


Figura C.6. Perfil para el grupo lógico Casos de prueba.

Por simplicidad, los atributos existentes en las metaclases de UML han sido omitidos de la figuras C.4, C.5 y C.6.

3. Conclusiones

En esta sección se han presentado dos perfiles de UML que permiten representar la información y semántica definida en este trabajo de tesis para los requisitos funcionales y las pruebas funcionales del sistema como una extensión del propio UML. Se han trasladado a dichos perfiles los mismos conceptos definidos en los metamodelos anteriores. Para ello, ha sido necesario estudiar la jerarquía y los elementos de UML para encontrar los elementos más adecuados a estereotipar.

Las ventajas de desarrollar un perfil de UML es que se extiende al propio UML con uno e los mecanismos recogidos en la documentación de UML para su extensión, lo que permite trabajar con cualquier herramienta que recoja dicho mecanismo de extensión.

Apéndice D. Publicaciones.

1. Introducción

En este anexo, se recogen las publicaciones que se han realizado sobre el trabajo presentado en la tesis así como los proyectos de investigación en los que se ha participado durante la realización de la misma. Las publicaciones se han catalogado según su año de publicación.

2. Año 2.004

J.J. Gutierrez, M.J. Escalona, M. Mejías, J.Torres
Comparative Analysis of Methodological Proposes to Systematic Generation of System Test Cases from System Requirements
SV'2004
Proceedings of the 3rd International Workshop on System Testing and Validation
PP. 151-160, ISBN: 3-8167-6677-3
Francia 2004

J.J. Gutiérrez, M.J. Escalona, M. Mejías, J.Torres
Aplicando técnicas de testing en sistemas para la difusión Patrimonial
TURITEC'2004
Actas del V Congreso Nacional de Turismo y Tecnologías de la Información y las comunicaciones;
PP. 237-252, ISBN: 84-608-0181-0
España, 2004

J.J. Gutiérrez, J. Torres, M. Mejías, M.J. Escalona
Métodos de testing sobre la Ingeniería de Requisitos Web de NDT
CIAWI 2004
Actas da conferência IberoAmericana WWW/Internet 2004;
PP. 353-360, ISBN/ISSN: 972-99353-1-9
España, 2004

M. Mejías, J. Torres, M. J. Escalona, J. J. Gutiérrez, J. A. Álvarez
Análisis de Propuestas para la Generación de Casos de Prueba para el Control de Calidad
I SIMPOSIO EN AVANCES EN GESTIOS DE PROYECTOS Y CALIDAD DEL SOFTWARE

Salamanca, España 2004

J.J. Gutiérrez, D. Villadiego, M.J. Escalona, M. Mejías
Aplicación de la Programación Orientada a Aspectos en el Diseño e Implementación de Pruebas Funcionales
JISBD 2004
Actas del Segundo Taller de Desarrollo de Ingeniería del Software Orientado a Aspectos
PP. 99-106, ISBN/ISSN: 84-688-8889-3
España, 2004

J.J. Gutiérrez, M.J. Escalona
FIREBIRD, un servidor de Bases de Datos Relacionales de Código Abierto
Solo Programadores
Revistas Profesionales S.L.
España, 2004

J.J. Gutiérrez, M.J. Escalona
Jakarta Commons. Componentes Java Multiplataforma
Todo Programación
Studio Press;
Vol. 5, pp. 10-15
España, 2004

3. Año 2.005

J.J. Gutiérrez, M.J. Escalona, M. Mejías, J. Torres
Estudio comparativo de propuestas para la generación de casos de prueba a partir de requisitos funcionales
Internal Report LSI-2005-01
Department of Computer Languages and System; España (2005-03)

J.J. Gutiérrez, M.J. Escalona, M. Mejías, J. Torres
Analysis of proposals for the generation of system test cases from system requirements
CAISE'05
Proceedings of the CAISE'05 Forum; Portugal (2005-06)
PP. 125-130, ISBN: 972-752-078-2

Índice de calidad: Otros índices: - Congreso Tier Conference A - Recogido en ISI proceedings - Recogido en DBLP - Artículo fruto de resultados de investigación

J.J. Gutiérrez, M.J. Escalona, M. Mejías, J. Torres
Testing web applications in practice
WWv 2005
Proceedings of the 1st International Workshop on Automated Specification and Verification of Web sites
PP. 65-76
España 2005

J.J. Gutiérrez, R. Pineda, D. Villadiego, M.J. Escalona, M. Mejías
Pruebas funcionales y pruebas de carga sobre aplicaciones Web
Mundo Internet 2005
Proceedings of Mundo Internet 2005
Vol. II, pp. 208-219
España 2005

J. J. Gutiérrez, M.J. Escalona, J. Torres, M. Mejías
A Practical Approach of Web System Testing
Lecture Notes in Computer Science, ISSN: 0302-9743
Vol. Págs. 659 -680
Editorial SPRINGER Año 2005
Índice de impacto: Año: 2005. Categoría: Computer Science, Artificial Intelligence, índice de impacto : 0,302 . Ranking dentro de la Categoría : 70 / 103

M.J. Escalona, A. Martín-Pradas, L.F. de Juan, D. Villadiego, J.J. Gutiérrez
El Sistema de Información de Autoridades del Patrimonio Histórico Andaluz
JBiDi 2005
V Jornadas de Bibliotecas Digitales; España (2005-09)
PP. 87-92, ISBN/ISSN: 84-9732-453-6

M.J. Escalona, J.J. Gutiérrez, J. Torres, M. Mejías, D. Villadiego
System testing in web Modelling languages
WISM 2005
International Workshop on Web Information Systems Modeling; Australia (2005-07)

D. Villadiego, J.J. Gutiérrez, M.J. Escalona, J. Torres, M. Mejías

Una aplicación real de NDT en un sistema para el reconocimiento, declaración y calificación del grado de minusvalía

Mundo Internet 2005

Proceedings of Mundo Internet 2005; España (2005-04)

Vol. II, pp. 116-123

J.J. Gutiérrez, M.J. Escalona

XQuery. La consulta sobre XML

Solo Programadores

Solo Programadores; España (2005-01)

Vol. 121, pp. 44-49, ISBN/ISSN: 1134-4792

J.J. Gutiérrez, M.J. Escalona, M. Mejías, J. Torres, D. Villadiego

XQuery

Internal Report LSI-2005-02

Department of Computer Languages and System; España (2005-03)

J.J. Gutiérrez, M.J. Escalona, M. Mejías, D. Villadiego

Comparativa de herramienta para la enseñanza de lenguajes relacionales

JENUI 2005

Proceedings of XI Jornadas de la Enseñanza Universitaria de la Informática; España (2005-07)

PP. 297-304, ISBN/ISSN: 84-9732-421-8

J.J. Gutiérrez, M.J. Escalona, D. Villadiego, M. Mejías

Herramientas libres para enseñanza del álgebra relacional

CEDI 2005

I Simposio Nacional de Tecnologías de la Información y de las Comunicaciones en la Educación; España (2005-09)

PP. 243-250, ISBN/ISSN: 84-9732-437-4

4. Año 2.006

J.J. Gutiérrez, M.J. Escalona, M. Mejías, J. Torres

Modelos y algoritmos para la generación de objetivos de pruebas

JISBD 06
Actas de las XV Jornadas de Ingeniería del Software y Bases de Datos.
PP. 119-130
España, 2006

J.J. Gutiérrez, M.J. Escalona, M. Mejías, J. Torres
Towards a Complete Approach to Generate System Test Cases
DCEIS 2006
Proceedings of the 4th ICEIS Doctoral Consortium; Chipre (2006)
PP. 38-50, ISBN: 972-8865-58-9

J.J. Gutiérrez, M.J. Escalona, M. Mejías, J. Torres
System test cases from use cases
ICSoft 2006
Proceeding of the 1st International Conference on Software and Data Technologies; Portugal (2006)
PP. 283-286, ISBN: 972-8865-69-4

J.J. Gutiérrez, M.J. Escalona, A.H. Torres, M. Mejías, J. Torres
Hacia una propuesta de pruebas tempranas del sistema
PRIS 2006
Taller sobre Pruebas en Ingeniería del Software.
España (2006)

J.J. Gutiérrez, M.J. Escalona, M. Mejías, A.M. Reina
Modelos de pruebas para Pruebas de Sistemas
DSDM 2006
Taller sobre Desarrollo de Software Dirigido por Modelos, MDA y Aplicaciones.
España (2006)

J.J. Gutiérrez, M.J. Escalona, M. Mejías, J. Torres
An approach to generate test cases from use cases
ICWE 2006
ACM International Conference Proceeding Series;
PP. 113-114, ISBN: 1-59593-352-2
USA 2006
Índice de calidad: El congreso ICWE es un congreso que cuenta con 10 ediciones y es referente en el área de ingeniería Web. Aunque la publicación no es un libro, se ha colocado en

esta sección por la relevancia del congreso. Otros índices: - Congreso Tier Conference C - Recogido en ISI proceedings - Recogido en DBLP - Artículo fruto de resultados de investigación.

J.J. Gutiérrez, M.J. Escalona, M. Mejías, J. Torres

Generation of test cases from functional requirements. A survey

SV 2006

Proceeding of the 4th Workshop on System Testing and Validation

PP. 117-126, ISBN: 3-8167-7072-X

Alemania 2006

J.J. Gutiérrez, M.J. Escalona, M. Mejías, J. Torres

Generating Test Cases From Sequences of Use Cases

WebIST 2006

Proceedings of the 2nd International Conference on Web Information Systems and Technologies

PP. 473-476, ISBN: 978-972-8865-46-7

Portugal 2006

Índice de impacto: El congreso WebIST es un congreso que cuenta con 7 ediciones y es referente en el área de ingeniería del Software orientada a la web. Aunque la publicación no es un libro, se ha colocado en esta sección por la relevancia del congreso. Otros índices: - Congreso Tier Conference C – Recogido en ISI proceedings - Recogido en DBLP - Artículo fruto de resultados de investigación con gran aplicación práctica.

J.J Gutiérrez, M.J Escalona, M. Mejías, J. Torres,

Building Web Applications with XQuery

1st International Conference on Web Information Systems and Technologies

972-8865-20-1

Miami, USA. 2006

M.J. Escalona, D. Villadiego, J.J. Gutiérrez, J. Torres, M. Mejías

A practical experience with NDT. The system to Measure the grade of handicap

ICEIS 2006

Proceedings of the 8th International Conference on Enterprise Information Systems; Portugal (2006)

Vol. 3, pp. 212-217, ISBN/ISSN: 972-8865-43-0

5. Año 2.007

J.J. Gutiérrez, M.J. Escalona, M. Mejías, J. Torres

Derivation of test objectives automatically

Libro: Advances in Information Systems Development. New Methods and Practice for the Networked Society

Editorial: Springer

Volumen: 2, Páginas: 435-446

ISBN: 13-978-0-387-70801-0

United States. 2007

Índice de calidad: Nº de citas: 2. Este libro surge como una selección de los artículos publicados en el congreso ISD. El congreso ISD es un congreso que cuenta con 19 ediciones y es referente en el desarrollo de sistemas de información. Otros índices: - Congreso Tier Conference A - Recogido en ISI proceedings - Artículo fruto de resultados de investigación

J.J. Gutiérrez, M.J. Escalona, M. Mejías, A.H. Torres

Using Use Case Scenarios and Operational Variables for Generating Test Objectives

STV 2007

Proceedings of the 5th Workshop on Systems Testing and Validation

PP. 23-32, ISBN: 978-3-8167-7475-4

France 2007

J.J. Gutiérrez, M.J. Escalona, M. Mejías, A.H. Torres

Generación Automática de Objetivos de Prueba a Partir de Casos de Uso Mediante Partición de Categorías y Variables Operacionales

JISBD 2007

Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos

PP. 105-114, ISBN: 978-84-9732-595-0

España 2007

J.J. Gutiérrez, M.J. Escalona, M. Mejías, A.H. Torres, J. Torres

Implementación de Pruebas de Sistema

JISBD 2007

Actas del II Taller sobre Pruebas de Sistema

Vol. 1, ISBN: 1988-3455

España 2007

J.J. Gutiérrez, M.J. Escalona, M. Mejías, A.H. Torres, J. Torres

Generación e implementación de pruebas del sistema a partir de casos de uso

Revista: Revista Española de Innovación, Calidad e Ingeniería del Software

Editorial: ATI

Volumen: 3; Nº 3, Páginas: 7-27

ISSN: 1885-4486

España. 2007

Índice de calidad: Este artículo fue seleccionado como uno de los mejores del Workshop de pruebas en las Jornadas de Ingeniería del Software y Bases de Datos y ha sido revisado para la publicación en esta revista. Esta revista está incluida en los siguientes índices: E-revistas (CSIC): http://www.erevistas.csic.es/ ICYT, Revistas de Ciencia y Tecnología (CSIC): Directorio de ICYT y Sumarios de ICYT. Redalyc: http://redalyc.uaemex.mx/ Latindex: Directorio DOAJ: http://www.doaj.org/ Ulrich: http://www.ulrichsweb.com/ Google Académico (Google Scholar): http://scholar.google.es/

J.J. Gutiérrez, M.J. Escalona, M. Mejías, J. Torres

A practical Approach of Web System Testing

Libro: Advances in Information Systems Development. Editorial: Springer

Volumen: 2, Páginas: 669-680

ISBN: 0-387-30834-2

United States. 2007

Índice de impacto: Este libro surge como una selección de los artículos publicados en el congreso ISD. El congreso ISD es un congreso que cuenta con 19 ediciones y es referente en el desarrollo de sistemas de información. Otros índices: - Congreso Tier Conference A - Recogido en ISI proceedings - Artículo fruto de resultados de investigación

M. J. Escalona, J. Torres, M. Mejías, J. J. Gutiérrez, D. Villadiego

The Treatment of Navigation in Web Engineering

ADVANCES IN ENGINEERING SOFTWARE, ISSN: 0965-9978

ELSEVIER SCI LTD

Año 2007

Índice de impacto: Año 2007, categoría Computer Science, Interdisciplinary Applications, impacto: 0,529. ranking dentro de la categoría : 67 / 95, año 2007, categoría : Computer Science, Software Engineering, impacto : 0,529, ranking dentro de la categoría : 61 / 93

M. J. Escalona, J.J. Gutiérrez, D. Villadiego; Á. León, A. H. Torres

Practical Experiences in Web Engineering
ADVANCES IN INFORMATION SYSTEMS DEVELOPMENT
ISBN: 978-0-387-708
SPRINGER SCIENCE+BUSINESS MEDIA, LCC
NEW YORK, USA 2007
Índice de impacto: número de citas: 5. Este libro surge como una selección de los artículos publicados en el congreso ISD. El congreso ISD es un congreso que cuenta con 19 ediciones y es referente en el desarrollo de sistemas de información. Otros índices: - Congreso Tier Conference A - Recogido en ISI proceedings - Artículo fruto de resultados de investigación

7. Año 2.008

J.J. Gutiérrez, M.J. Escalona, M. Mejías, I. Ramos
Test cases from Functional Requirements using Model transformations
STV 2008
Proceedings of 6th Workshop on System Testing and Validation;
France 2008

J.J. Gutiérrez, M.J. Escalona, C. Gómez, M. Mejías, I. Ramos
Aplicación automática del método de categoría-partición a requisitos funcionales con soporte para bucles
EATIS 2008
Proceedings of the Euroamerican Conference on Telematics and Information Systems;
ACM Digital Library (ACM-DL). ISBN: -1-59593-988-3
Brasil 2008

J.J. Gutiérrez, M.J. Escalona, M. Mejías, A.H. Torres, J. Torres
A case Study for Generating Test Cases from Use Cases
Proceedings of IEEE International conference on Research Challenges in Information Science
IEEE Morroco Section
PP. 223-228
Morroco 2008

J.J. Gutiérrez, C. Nebut, M.J. Escalona, M. Mejías, I. Ramos
Visualization of use cases through automatically generated activity diagrams

Lecturer Notes in Computer Science
Editorial: Springer Verlag
Volumen: 5301, Páginas: 83-96
ISSN: 0302-9743
USA. 2008
Índice de impacto: 15 citas según el buscador scholar.google.com. Último acceso, 13/09/2011

M.J. Escalona, A.H. Torres, J.J. Gutiérrez, E. Martins, R.S. Torres, M. Cecilia, C. Baranauskas

A Development Process for Web Geographic Information System A Case of Study

ICEIS 2008
Procedings of the International conference on enterprise information system; España (2008)
Vol. HCI, pp. 112-117, ISBN/ISSN: 978-989-8111-40-1

A.H. Torres, M.J. Escalona, J.J. Gutiérrez

Hacia el diseño de aplicaciones reusables

XII International Conference on Project Engineering
AEIPRO; España (2008-07)
Vol. T-12, pp. 1-13, ISBN/ISSN: 978-84-936430-3-4

M.J. Escalona, J.J. Gutiérrez, J.A. Ortega, G. Aragón, M. Pérez-Pérez, J. Ponce

NDT-Suite. Una solución práctica para NDT

X Jornadas Arca
Actas de las X Jornadas Arca. Sistemas Cualitativos y Diagnosis. Robótica. Sistemas Domóticos y Computación Ubicua; España (2008)
PP. 73-80, ISBN/ISSN: 978-84-89315-54-9

M.J. Escalona, J.J. Gutiérrez, J.A. Ortega, I. Ramos

NDT & METRICA V3. An approach for Public Organizations based on Model Driven Engineering

INSTICC Press
Lisboa. Portugal
2008
ISBN: 978-989-8111-26-5

Índice de calidad: número de citas: 1. El congreso WebIST es un congreso que cuenta con 7 ediciones y es referente en el área de ingeniería del Software orientada a la web. Aunque la publicación no es un libro, se ha colocado en esta sección por la relevancia del congreso. Otros índices: - Congreso Tier Conference C – Recogido en ISI proceedings - Recogido en DBLP - Artículo fruto de resultados de investigación con gran aplicación práctica

5. Año 2.009

J.J. Gutiérrez, M.J. Escalona, M. Mejías, I. Ramos, J. Torres

An Approach for Model-Driven Test Generation

Proceeding of the IEEE International Conference on Research Challenges in Information Science. RCIS 2009

IEEE Morroco

PP. 337-346, ISBN: 978-1-4244-2864-9

Morroco 2009

M. J. Escalona, M. Mejías, J. J. Gutiérrez.

Oráculos de Prueba: Un Planteamiento Heurístico de Apoyo a la Decisión

ACTAS DE TALLERES DE INGENIERÍA DEL SOFTWARE Y BASES DE DATOS, ISSN: 1988-3455

Vol. 3 Págs. 104 -113

Universidad de Zaragoza 2009

M.J. Escalona, J.J. Gutiérrez, L. Rodríguez-Catalán, A. Guevara

Model-driven in reverse. The practical experience of the AQUA Project

Proceeding of the Euro American Conference on Telematics and Information Systems EATIS 2009; Czech Republic (2009)

PP. 90-95, ISBN/ISSN: 978-1-60558-398-3

A.H. Torres, M.J. Escalona, M. Mejías, J.J. Gutiérrez

A MDA-Based Testing. A Comparative Study

Proceeding of 4th international conference on Software and Data Technologie

ICSOFT 2009; Bulgaria (2009-07)

Vol. 1, pp. 269-274, ISBN/ISSN: 978-989-674-009-2

M. J. Escalona, M. Mejías, J. J. Gutiérrez

Una Aproximación a las Pruebas de Aplicaciones Web Basadas en un Contexto MDWE
REF. X REVISTA: ACTAS DE TALLERES DE INGENIERÍA DEL SOFTWARE Y BASES DE DATOS
ISSN: 1988-3455
Editorial UNIVERSIDAD DE ZARAGOZA Año 2009

M. J. Escalona, J.J. Gutiérrez, C. Parra
A Practical Environment to Apply Model-Driven Web Engineering
INFORMATION SYSTEMS DEVELOPMENT
ISBN: 978-0-387-848
SPRINGER SCIENCE+BUSINESS MEDIA, LCC Año 2009
Índice de impacto: Nº de citas: 1. Este libre surge como una selección de los artículos publicados en el congreso ISD. El congreso ISD es un congreso que cuenta con 19 ediciones y es referente en el desarrollo de sistemas de información. Otros índices: - Congreso Tier Conference A - Recogido en ISI proceedings - Recogido en DBLP - Artículo fruto de resultados de investigación aplicados al Servicio Andaluz de Salud.

J. J. Gutiérrez y el grupo de profesores de la asignatura
Estructuras de Datos y Algoritmos
ISBN: 978-84-692-52
Universidad de Sevilla. Año 2009

J. J. Gutiérrez y el grupo de profesores de la asignatura
Análisis y Diseño de Algoritmos
ISBN: 978-84-692-52
Universidad de Sevilla. Año 2009

5. Año 2.010

M. J. Escalona, J.J. Gutiérrez
Measuring the Quality of Model-Driven Projects with NDT-Quality
INFORMATION SYSTEM DEVELOPMENT
SPRINGER SCIENCE+BUSINESS MEDIA, LCC
ESTADOS UNIDOS
ISBN: 978-1-4419-73

Año 2010

M. J. Escalona, J. J. Gutiérrez, J. Torres
La Carrera de Informática Tras la Universidad
ISBN: 978-84-693-82
Universidad de Sevilla. Año 2010

J. J. Gutiérrez y el grupo de profesores de la asignatura
Estructura de Datos y Algoritmos
ISBN: 978-84-693-82
Universidad de Sevilla. Año 2010

J. J. Gutiérrez y el grupo de profesores de la asignatura
Análisis y Diseño de Algoritmos
ISBN: 978-84-693-82
Universidad de Sevilla. Año 2010

5. Año 2.011

Escalona M.J., Gutiérrez J.J., Mejías M., Aragón G., Ramos I., Torres J., Domínguez F.J.
An overview on test generation from functional requirements.
The Journal of Systems and Software.
Elsevier. ISSN: 0164-1212.
2011.