Agradecimientos

Agradezco a mis padres y hermanas por acompañarme en todo este proceso desde el principio.
A mis amigos de la facultad. Con su amistad, el paso por la facultad fue mucho más lindo.
A Martín, por hacer divertido este último tramo de la carrera.

Noel.

Agradezco a mis padres y a mi hermano. Cada uno a su manera me ha ayudado para alcanzar este objetivo.
A mis amigos de la facultad, por hacer divertido además de constructivo mi paso por la facultad.
A todos los compañeros con los que preparé un parcial, un final o un trabajo.
A Noel. Compañera de tesis y de la vida.

Martín.

A Claudia. Por la dedicación, predisposición y voluntad que puso en ayudarnos para lograr este objetivo.
A Diego García, por toda la ayuda prestada.

Noel y Martín.

Índice general

1. Introducción 7

2. Motivación 9

2.1. Aportes 9

3. UML, MOF y Metamodelos 10

3.1. UML 10
3.2. Concepto de meta-metamodelo y metamodelo 12
3.3. MOF 12
3.3.1. Arquitectura MOF 13
3.3.2. Construcciones básicas utilizadas por MOF 15
3.4. Paquetes principales del metamodelo de UML 15

4. Uso de OCL en modelos basados en MOF 17

4.1. Restricciones en OCL 18
4.1.1. Invariante 18
4.1.2. Definición 20
4.1.3. Precondición 20
4.1.4. Postcondición 21
4.1.5. Reglas a nivel metamodelo 22
4.2. Expresión de valor inicial 23
4.3. Expresión de valor derivado 24
4.4. Expresión de consulta 24

5. Arquitectura dirigida por modelos 26

5.1. Introducción a MDA 26
5.2. Modelos en MDA 28
5.2.1. PIM 28
5.2.2. PSM 28
5.3. Desarrollo tradicional Vs. Desarrollo con MDA 29
5.3.1. Problemas de desarrollo tradicional 29
5.3.2. Beneficios del MDA 31
5.3.3. El nuevo proceso de desarrollo 32
5.4. Importancia del metamodelado en MDA 33

6. Estrategias de evaluación para las condiciones de refinamiento de modelos MOF 34

6.1. Condiciones de refinamiento 35
6.2. Relación de refinamiento en UML 36
6.3. Estrategia de verificación para patrones de refinamiento UML 36
6.3.1. Patrón State 37
6.3.2. Patrón Object Descomposition 40
6.3.3. Patrón Atomic Operation 42
6.4. Micromundos para la evaluación de las condiciones de refinamiento 45

7. Generación de micromundos representativos 48
7.1. Conceptos 48
7.1.1. Criterio de testeo para diagramas UML 49
7.1.2. Conclusiones 51
7.2. Generando las particiones 51
7.3. Combinando las particiones 53
7.4. Fragmentos de modelo y fragmentos de objeto 54
7.5. Criterios de test 55
7.5.1. Criterios de cobertura simple 55
7.5.2. Criterios de cobertura clase por clase 56
7.6. Conclusiones 60
8. ePlatero. Módulo Generador de Micromundos 61
8.1. Arquitectura de ePlatero 62
8.2. Módulo Generador de Micromundos 62
8.3. Componentes 64
8.4. Metamodelo utilizado 64
8.5. Propiedad de dualidad y función de abstracción 66
8.5.1. Problemas y decisiones 66
8.5.2. Función de abstracción 67
8.5.3. Ejemplo de función de abstracción 67
8.6. Estructura del componente 69
8.6.1. Generación de particiones 70
8.6.2. Generación del fragmento de modelo y creación del micromundo 74
8.7. Patrones de diseño utilizados 77
8.7.1. Strategy 77
8.7.2. Visitor 78
9. Caso de estudio 80
9.1. Dominio 80
9.2. Creación del modelo 81
9.3. Definición de reglas OCL 82
9.4. Parseo del archivo OCL 83
9.5. Generación del micromundo 84
9.5.1. Micromundo para OneRangeCombination y sin límite de instancias 88
9.5.2. Micromundo para AllRangesCombination y sin límite de instancias 90
9.5.3. Micromundo para OneRangeCombination y con límite de instancias 93
9.5.4. Micromundo para AllRangesCombination y con límite de instancias 95
9.5.5. Conclusiones 97
9.6. Evaluación del refinamiento 98
9.7. Comparación con Alloy Analyser 100
10. Conclusiones 102
10.1. Futuros trabajos 103
11. Referencias 104
12. Anexo I - Metamodelo de la sintaxis abstracta de OCL 2.0 106

12.1. Paquete types 106
12.2. Paquete expressions 107
12.2.1. Metaclase ExpressionInOcl 111
12.3. Ejemplo de instanciación del metamodelo de OCL 112
13. Anexo II - Descripción de los módulos de ePlatero 114
13.1. Analizador léxico y sintáctico 114
13.2. Descripción del editor de fórmulas OCL 118
13.2.1. Diseño del editor OCL 121
13.3. Descripción del evaluador OCL 121
13.3.1. Diseño del evaluador OCL 122
13.4. Descripción del evaluador de refinamientos 125

Tabla de figuras

Figura 1 - Ejemplo de modelo de 4 capas 14
Figura 2 - Paquetes del metamodelo UML..... 16
Figura 3 - Ejemplo de diagrama de clases..... 18
Figura 4 - Expresión OCL utilizada como Invariante o Definición 19
Figura 5 - Expresión OCL utilizada como Precondición o Postcondición..... 21
Figura 6 - Expresión OCL utilizada como Valor Inicial de una Propiedad 23
Figura 7 - Expresión OCL utilizada en una Operación de Consulta 25
Figura 8 - Logotipo de MDA..... 27
Figura 9 - Pasos en el desarrollo con MDA..... 28
Figura 10 - Proceso de desarrollo de software tradicional 30
Figura 11 - Proceso de desarrollo con MDA 33
Figura 12 - Proceso de verificación de refinamiento..... 37
Figura 13 - Instancia del patrón de refinamiento State..... 38
Figura 14 - Resultado del proceso aplicado a una instancia del patrón de refinamiento 39
Figura 15 - Instancia del patrón de refinamiento Object Descomposition 40
Figura 16 - Resultado del proceso aplicado a una instancia del patrón de refinamiento 42
Figura 17 - Instancia del patrón de refinamiento Atomic Operation..... 43
Figura 18 - Resultado del proceso aplicado a una instancia del patrón de refinamiento 44
Figura 19 - Invariantes OCL que reducen el espacio de búsqueda 45
Figura 20 - Micromundo generado automáticamente del modelo UML de la figura 15 enriquecido con las restricciones de la figura 19..... 46
Figura 21 - Ejemplo de multiplicidad de asociaciones..... 50
Figura 22 - Ejemplo de modelo para generar particiones..... 52
Figura 23 - Particiones generadas para un modelo de ejemplo 52
Figura 24 - Metamodelo de particiones y fragmentos..... 53
Figura 25 - Ejemplo de fragmento de modelo..... 54
Figura 26 - Cobertura de rangos y particiones en OCL..... 56
Figura 27 - Dos estrategias para combinaciones de rangos..... 57
Figura 28 - Ejemplo para combinaciones de rangos..... 57
Figura 29 - Particiones para el ejemplo de la figura 28..... 57
Figura 30 - Fragmentos de objeto utilizando la estrategia OneRangeCombination..... 58
Figura 31 - Fragmentos de objeto utilizando la estrategia AllRangesCombination..... 58
Figura 32 - Dos estrategias para generar fragmentos de modelo 59
Figura 33 - Criterios de test basados en las combinaciones clase por clase..... 60
Figura 34 - Arquitectura de eclipse, para el desarrollo de plugins..... 61
Figura 35 - Arquitectura de ePlatero 62
Figura 36 - Componentes del Generador de Micromundos 64
Figura 37 - Metamodelo para la generación de micromundos 65
Figura 38 - Ejemplo para la función de abstracción..... 68
Figura 39 - Clase MicroWorldFactory 69
Figura 40 - Colaboradores de MicroWorldFactory para la generación de particiones .. 71
Figura 41 - Generación de particiones para las clases involucradas en un refinamiento 74
Figura 42 - Colaboradores de MicroWorldFactory para la generación de micromundos 75
Figura 43 - Generación del fragmento de modelo y creación del micromundo 77
Figura 44 - Utilización del patrón Strategy 78
Figura 45 - Utilización del Patrón Visitor 79

Figura 46 - Diagrama de clases del caso de estudio 80
Figura 47 - Editor de diagramas UML de ePlatero 81
Figura 48 - Editor OCL de ePlatero 83
Figura 49 - Generación de micromundo en ePlatero 85
Figura 50 - Particiones generadas con ePlatero 86
Figura 51 - Ventana para agregar nuevos rangos en ePlatero 87
Figura 52 - Función de abstracción para el caso de estudio 88
Figura 53 - Micromundo generado por ePlatero para la estrategia OneRangeCombination 88
Figura 54 - Micromundo generado por ePlatero para la estrategia AllRangesCombination y sin establecer límite de cantidad de instancias 91
Figura 55 - Micromundo generado por ePlatero para la estrategia OneRangeCombination y con límite de 2 instancias 94
Figura 56 - Micromundo generado por ePlatero para la estrategia AllRangesCombination y con límite de 3 instancias 96
Figura 57 - Evaluación del refinamiento en ePlatero 98
Figura 58 - Resultado de la evaluación del refinamiento 99
Figura 59 - Resultado de la evaluación del refinamiento modificando alguna precondición 99
Figura 60 - Resultado de la evaluación del refinamiento modificando alguna postcondición 99
Figura 61 - Comparación de tiempos promedio con un límite de 5 instancias 100
Figura 62 - Comparación de tiempos promedio con un límite de 10 instancias 100
Figura 63 - Comparación de respuestas correctas con micromundos de tamaño 5 101
Figura 64 - Comparación de respuestas correctas con micromundos de tamaño 10 101
Figura 65 - Extracto del metamodelo de la sintaxis abstracta para los tipos de OCL 106
Figura 66 - Estructura básica del metamodelo de la sintaxis abstracta para expresiones 108
Figura 67 - Metamodelo de la sintaxis abstracta para FeatureCallExp 110
Figura 68 - Metamodelo para la expresión if 111
Figura 69 - Metamodelo de la sintaxis abstracta para ExpressionInOcl 112
Figura 70 - Instanciación del metamodelo de la Sintaxis abstracta de OCL 113
Figura 71 - Ejemplo analizador léxico y sintáctico 114
Figura 72 - Ejemplo de un árbol de sintaxis concreta 115
Figura 73 - Traducción de IntegerLiteralCS a IntegerLiteralAS 116
Figura 74 - Traducción de PathNameExpCS a VariableExpAS 116
Figura 75 - Traducción de DotSelectionExpCS a PropertyCallExpAS 117
Figura 76 – Traducción de InfixOperationExpCS a OperationCallExpAS 118
Figura 77 - Editor de fórmulas OCL 119
Figura 78 - Relaciones de la clase AbstractTextEditor 120
Figura 79 - Diseño básico del editor de fórmulas OCL 121
Figura 80 - Visualización de errores semánticos 122
Figura 81 - Diagrama de paquetes del evaluador OCL 123
Figura 82 - Diagrama de clases del paquete basics 124
Figura 83 - Diagrama de clases del analizador semántico 125
Figura 84 - Diagrama de clases de la implementación del evaluador de refinamientos 126

1. Introducción

La complejidad de los problemas del mundo real ha llevado a que la construcción de un sistema de software debe ser precedida por la construcción de un modelo. El modelo de un sistema es una representación conceptual obtenida a partir de la identificación, clasificación y abstracción de los elementos que constituyen el problema y su posterior organización en una estructura formal.

De esta forma, el modelo de un sistema actúa como una especificación de los requerimientos que el sistema debe satisfacer, proveyendo un medio de comunicación y negociación entre usuarios, clientes, analistas y desarrolladores, así como también un documento de referencia durante la verificación y validación, y durante la evolución del producto. Es de suma importancia expresar el problema claramente y con precisión; pero esta meta es difícil de lograr, los modelos tienden a contener errores, omisiones e inconsistencias porque ellos son el resultado de una actividad compleja y creativa.

El modelo del sistema se expresa utilizando un lenguaje de modelado. El éxito de los lenguajes gráficos de modelado, como el Unified Modelling Language (UML), se basa principalmente en el uso de construcciones gráficas que transmiten un significado intuitivo. Estos lenguajes son atractivos para los usuarios porque son claros y entendibles. Estas características son vitales ya que el modelo también cumple una función contractual. Sin embargo, es fundamental contar con un lenguaje que permita expresar restricciones semánticas adicionales sobre los objetos del modelo, pudiendo obtener modelos más precisos y verificables.

OCL es un lenguaje de especificación formal fácil de leer y escribir, que fue definido por la OMG (Object Management Group). Permite expresar restricciones semánticas del sistema que no se pueden expresar a partir de una notación grafica. De esta forma, los diagramas complementados con expresiones OCL son más precisos, su documentación es más clara, se mejora la comunicación entre desarrolladores (evitando errores producidos por malas interpretaciones) y la compresibilidad del sistema en etapas iniciales del desarrollo de software es mayor.

Otra característica que es muy importante para la utilización de los lenguajes de modelado, en sistemas de software complejos, es contar con técnicas de refinamientos, permitiendo un desarrollo por etapas con distintos niveles de abstracción y postergando los detalles del problema en etapas posteriores. Esto es la esencia del paradigma de desarrollo de software conocido como MDE (acrónimo de “Model Driven software Engineering”) el cual tiene dos ejes principales:

- por un lado, hace énfasis en la separación entre la especificación de la funcionalidad esencial del sistema y la implementación de dicha funcionalidad usando plataformas tecnológicas específicas. Para ello, el MDE identifica dos tipos principales de modelos: modelos con alto nivel de abstracción e independientes de cualquier tecnología de implementación, llamados PIM (Platform Independent Model) y modelos que especifican el sistema en términos de construcciones de implementación disponibles en alguna tecnología específica, conocidos como PSM (Platform Specific Model);

- por otro lado, la transformación entre modelos constituye el motor del MDE; los modelos son sucesivamente transformados, comenzando con modelos independientes de la plataforma, con el objetivo de obtener a cada paso modelos más específicos, hasta llegar al código ejecutable.

Para asegurar la corrección del producto final, cada una de estas transformaciones entre modelos debe ser validada y/o verificada. Esto genera un nuevo desafió en el área de desarrollo de software ya que las técnicas tradicionales de validación y verificación (V&V) de programas deben ser adaptadas a este nuevo escenario.

2. Motivación

Escribir transformaciones de modelos es una tarea compleja y propensa a errores. Garantizar formalmente la correctitud de estas transformaciones es difícil, siendo requeridas técnicas de verificación formal. Una aproximación alternativa es la validación por testeo.

El testeo de una trasformación de modelos es típicamente realizado chequeando el resultado de la transformación aplicada a un conjunto de modelos de entrada. Para que la transformación sea confiable y esté librada de errores, lo ideal es extraer del modelo un conjunto de instancias con estado representativo y verificar si las condiciones de refinamiento establecidas entre los modelos se satisfacen para esas instancias. Este conjunto de instancias es llamado "micromundo".

En la práctica, la construcción manual y la edición de los micromundos son tareas tediosas (la estructura de los datos puede ser compleja, y los modelos pueden ser difíciles de manejar). La generación de los micromundos debe entonces estar automatizada para evitar la tarea de construirlos manualmente.

La motivación de nuestro trabajo es definir e implementar una estrategia eficiente y confiable para validar transformaciones de modelos UML. Dicha estrategia se basará en la técnica de generación de micromundos la cual ha sido propuesta y explotada para la V&V de sistemas de software tradicionales.

2.1. Aportes

De alguna manera este trabajo pretende contribuir a la mejora de la calidad del desarrollo de software.

En este trabajo se han estudiado diferentes técnicas para la generación de micromundos, de manera que éstos sean adecuados, con valores que tenga sentido poner a prueba. Esto tiene como fin, proporcionar una buena solución para el testeo de las transformaciones de modelos. Como se mencionó en la sección anterior, el desarrollo de estas transformaciones puede introducir errores y asegurar que éstas son correctas no es una tarea simple.

La idea es generar micromundos que permitan determinar la veracidad y correctitud de los refinamientos que pueden ser especificados actualmente en ePlatero.

Quizás el aporte más importante de este trabajo sea la automatización de la generación de micromundos. El plugin desarrollado para la herramienta ePlatero permite generar micromundos para testear refinamientos o para un modelo de clases completo, eligiendo una estrategia para tal fin.

3. UML, MOF y Metamodelos

UML (Unified Modeling Language) es un lenguaje gráfico para visualizar, especificar, construir y documentar los artefactos de sistemas de software.

UML es producto de varios años de trabajo, centrados en la unificación de los métodos más utilizados alrededor del mundo, la adopción de buenas ideas para la industria, y por sobre todas las cosas, el esfuerzo en hacer las cosas simples. UML es el resultado de un esfuerzo en equipo formado por varias compañías, patrocinado por OMG. Cada una de las compañías aportó sus propias perspectivas, sus motivos de preocupación y sus campos de interés, y esta diversidad de experiencias y puntos de vista enriqueció y consolidó el producto final.

UML ofrece una forma estándar para escribir modelos de sistemas. Para poder llegar a un lenguaje estándar, los proponentes de UML buscaron definir con precisión el significado de los símbolos, así como también las formas de usarlos y relacionarlos. Fue necesario que se alcanzara un amplio acuerdo en torno a cuáles debían ser los artefactos a usar durante el análisis y el diseño; es decir, cuáles debían ser los componentes de los modelos. Esta es precisamente la función del metamodelo.

El lenguaje de modelado debería ser coherente con el servicio necesario para proporcionar y transportar los componentes de los modelos. Este servicio es conocido como Meta-Object Facility (MOF). MOF es un meta-metamodelo que se utiliza para especificar metamodelos orientados a objetos.

En este capítulo se describe UML en la sección 3.1, el concepto de meta-metamodelo y metamodelo en la sección 3.2, MOF en la sección 3.3, y los paquetes principales del metamodelo UML en la sección 3.4.

3.1. UML

Como se menciono anteriormente, UML es un lenguaje gráfico para visualizar, especificar, construir y documentar los artefactos de sistemas de software. Ofrece una forma estándar para escribir modelos de sistemas.

Una de las ventajas que se pretenden lograr mediante un lenguaje unificado es permitir el intercambio de diagramas y de formas de representación de sistemas entre diversas herramientas. Por ejemplo, si un grupo de desarrolladores utilizara una herramienta CASE (Computer Aided Software Engineering) o una herramienta de modelado visual, debería existir la facilidad de transportarla a otra herramienta, independientemente del proveedor o fabricante de cada una de ellas.

Pero el problema de estandarización no tiene que ver sólo con la transportabilidad de los diagramas de una herramienta de software a otra. Para mediados de los noventa las llamadas “metodologías” habían proliferado de tal forma que los proyectos y equipos de trabajo se topaban constantemente con dificultades para seleccionar un método de análisis y diseño. Los métodos propuestos por Grady Booch [**6**] [**7**], Ivar Jacobson [**7**], James Martin, James Odell, Edward Yourdon, y muchos más, tenían asociadas sus

formas peculiares de diagramación. En otras palabras, cada uno de estos métodos contaba con su propio lenguaje de modelado. Esta falta de estandarización impedía la reutilización de soluciones de un proyecto a otro, y muchas veces inhibía la inversión en capacitación de personal y en herramientas para diagramar. A lo anterior debemos añadir el esfuerzo y los riesgos inherentes a la curva de aprendizaje de estos métodos de análisis y diseño.

Para poder llegar a un lenguaje estándar, los proponentes de UML buscaron definir con precisión el significado de los símbolos, así como también las formas de usarlos y relacionarlos. Fue necesario que se alcanzara un amplio acuerdo en torno a cuáles debían ser los artefactos a usar durante el análisis y el diseño; es decir, cuáles debían ser los componentes de los modelos. Esta es precisamente la función del metamodelo.

En el desarrollo de software los modelos juegan el mismo papel que los planos y las especificaciones en la industria de la construcción: primero, porque representan la forma de ir plasmando las aproximaciones e ideas necesarias para resolver los requerimientos del cliente; segundo, porque los modelos son esenciales para la comunicación entre los distintos equipos de trabajo y entre las distintas disciplinas que participan en un proyecto; tercero, porque es mucho más económico hacer correcciones sobre un modelo "en papel" que hacerlas durante la construcción misma. Y existen muchas razones más, como es el caso de la facilidad que brinda la documentación para dar mantenimiento a una aplicación.

Las causas de la crisis del software son variadas, pero entre las más importantes se encuentra la falta de una cultura común a quienes realizamos los desarrollos. Y esta carencia está íntimamente relacionada con la falta de medios de comunicación para, por ejemplo, intercambiar técnicas y soluciones exitosas. Es precisamente en estos aspectos donde UML nos puede ayudar.

Además de obviarnos el tener que decidir entre varias formas de representación, UML nos proporciona un lenguaje visual de modelado, y nos permite aprovechar componentes, plantillas y patrones a partir de soluciones exitosas que ya han sido probadas.

Para mantener su característica de estándar abierto, UML es independiente de los lenguajes, de las bases de datos y de la marca del software y del equipo de cómputo. Deliberadamente, los proponentes de este lenguaje unificado lo han dejado únicamente como un medio para especificar, visualizar y documentar los artefactos del desarrollo e implantación de sistemas. Es importante hacer hincapié en el hecho de que UML no es un método de análisis y diseño, ni pretende representar un proceso para guiar las actividades de un desarrollo. Los usuarios de distintos métodos podrán mantener su forma actual de trabajo, adaptando sus diagramas de acuerdo a los símbolos utilizados en UML. En la mayor parte de los casos esta adaptación se va a poder realizar con facilidad.

Debemos tener siempre presente el objetivo central del desarrollo de sistemas: lo que esperan los clientes que solicitan una solución informática es llegar a contar con un sistema confiable y amigable. El modelado es sólo un medio para alcanzar esta meta, y no un fin en sí mismo. Para un equipo de trabajo capaz de adaptarlo a las circunstancias, UML va a ser de gran ayuda; pero los modelos creados no van a sustituir la labor de

programación y aplicación de pruebas. Por el momento UML es sólo un lenguaje de modelado, y no pretende ser un lenguaje para la programación mediante símbolos visuales.

3.2. Concepto de meta-metamodelo y metamodelo

Un **meta-metamodelo** (OMG, 2003) es un modelo que define el lenguaje formal para representar un metamodelo. La relación entre un meta-metamodelo y un metamodelo es análoga a la relación entre un metamodelo y modelo.

Un **metamodelo** (OMG, 2003) es un modelo que define el lenguaje formal para representar un modelo.

Un metamodelo se expresa con notación gráfica y con un lenguaje formal, como OCL, para aumentar su precisión y eliminar ambigüedades. Si bien este acercamiento carece de la precisión de la especificación de los métodos formales, ofrece una especificación intuitiva y comprensible.

3.3. MOF

Para el grupo del OMG el problema no consistía únicamente en solicitar una serie de normas para crear diagramas, sino que estos diagramas y sus componentes deberían ser compatibles con los servicios necesarios para transportarlos de una herramienta de diagramación a otra (lo cual podría significar también el transportar diagramas de una plataforma a otra).

El lenguaje de modelado debería ser coherente con el servicio necesario para proporcionar y transportar los componentes de los modelos. Este servicio es conocido como Meta-Object Facility (MOF).

MOF es un meta-metamodelo que se utiliza para especificar metamodelos orientados a objetos. Define los elementos comunes y estructuras de metamodelos que son utilizados para construir modelos de sistemas orientados a objetos.

La especificación de MOF proporciona:

- ✓ Una definición formal del meta-metamodelo MOF; es decir un lenguaje abstracto para especificar metamodelos MOF.
- ✓ Un conjunto de reglas para el mapeo de metamodelos MOF a interfaces independientes del lenguaje de programación, definidos por medio del estándar de CORBA IDL. Una implementación de estas interfaces para un modelo concreto podría ser utilizada para acceder y modificar cualquier modelo basado en ese metamodelo.
- ✓ Un conjunto de interfaces reflexivas para manipular metadatos independientes del metamodelo.

- ✓ Un formato XMI[1] (XML Metadata Interchange) para el intercambio de metamodelos MOF.

Los metamodelos de UML y OCL, entre otros, están especificados mediante dicho meta-metamodelo.

3.3.1. Arquitectura MOF

MOF define una arquitectura en la que existen cuatro niveles o capas, estos son:

1. Meta-metamodelo
2. Metamodelo
3. Modelo del usuario
4. Instancias en tiempo de ejecución

La responsabilidad primaria de la capa de meta-metamodelo es definir el lenguaje para especificar un metamodelo. Esta capa es conocida como M3, y como se mencionó anteriormente MOF es un ejemplo de un meta-metamodelo. Por lo general este es más compacto que un metamodelo, y a menudo define varios metamodelos. El meta-metamodelo es común al MOF y a UML, lo que asegura que compartan una base común.

Un metamodelo es una instancia de un meta-metamodelo y significa que cada elemento del metamodelo es una instancia de un elemento del meta-metamodelo. La responsabilidad primaria de la capa del metamodelo es definir un lenguaje para especificar modelos. Esta capa es conocida como M2; UML y OCL son ejemplos de metamodelos. En general estos son más detallados que los meta-metamodelos que los describen, sobre todo cuando ellos definen semántica dinámica. Para dar un ejemplo, en el metamodelo de UML se definen los objetos del lenguaje unificado como Class, Property, Operation, etc.

Un modelo es una instancia de un metamodelo. Cuando creamos un modelo estamos definiendo un lenguaje para describir el área que estamos analizando o el sistema que estamos diseñando. Por la tanto, la responsabilidad de esta capa es definir un lenguaje que describa los dominios semánticos, para permitirles a los usuarios modelar una variedad de dominios diferentes, como procesos, requerimientos, etc. Esta capa es conocida como M1. El modelo del usuario contiene los elementos del modelo y los snapshots de instancias de dichos elementos que sirven de moldes para los datos que vamos a introducir, manipular, almacenar y procesar en nuestras aplicaciones.

Por último la capa de instancias es conocida como M0. Esta capa representa las instancias de los elementos del modelo (instancias “reales” del sistema) en tiempo de ejecución. Estos describen el área o dominio a los que está dedicada la aplicación. Los snapshots que son modelados en M1 son versiones reducidas de las instancias en tiempo de ejecución.

[1] Es un estándar de la OMG que mapea MOF a XML. Define cómo deben emplearse las etiquetas XML que son utilizadas para representar modelos MOF serializados en XML.

Se podrían añadir más niveles a los ya descritos (por encima de M3), pero la realidad es que no sería muy útil. En lugar de definir una capa M4, el OMG establece que todos los elementos de M3 se pueden definir con instancias de conceptos de M3, lo que significa que MOF se define a sí mismo.

La figura 1 es un ejemplo de modelo de 4 capas.

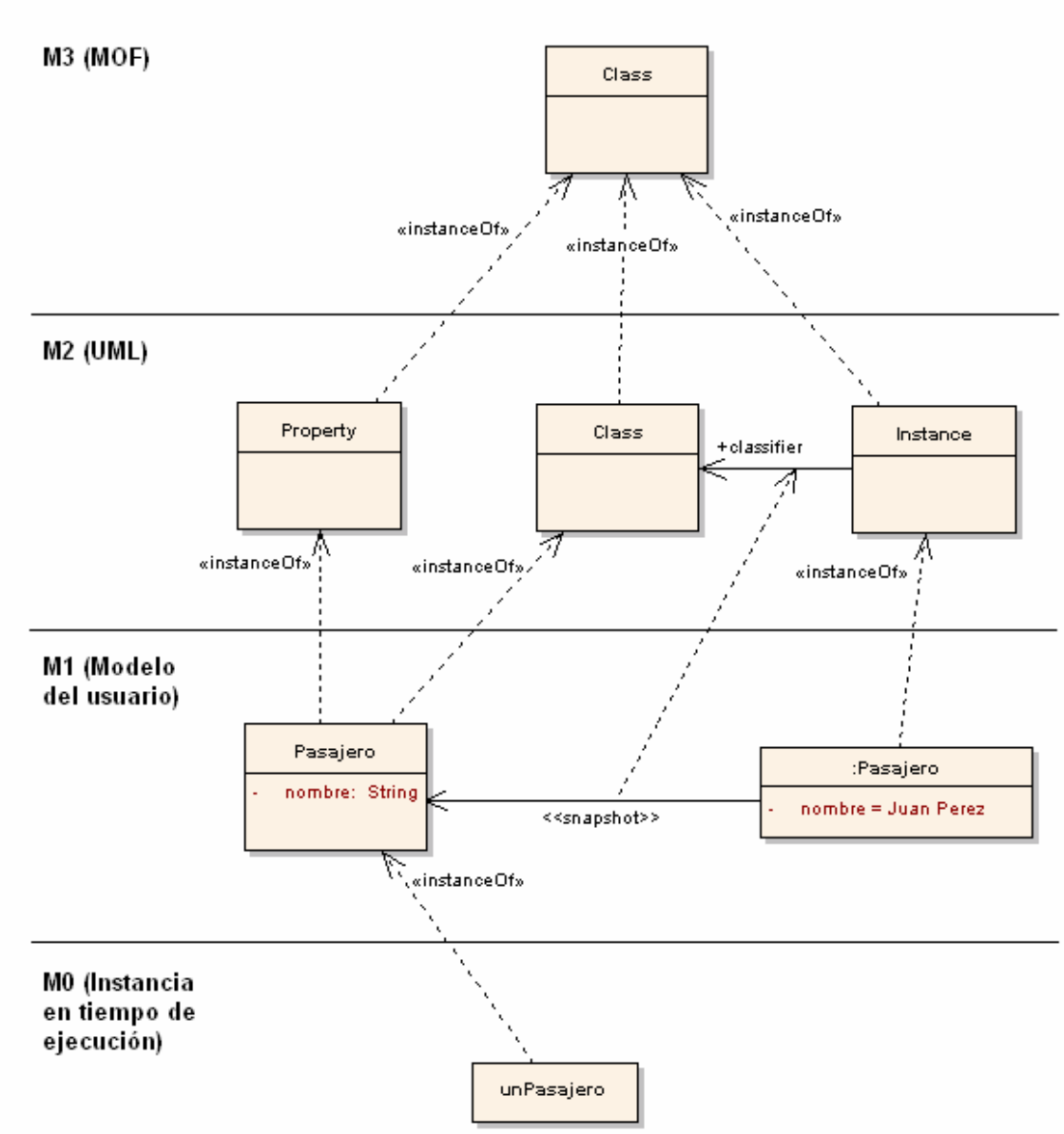


Figura 1 - Ejemplo de modelo de 4 capas

De acuerdo a esta arquitectura, cuando los usuarios creamos el objeto Pasajero en un diagrama estamos creando una instancia del metaobjeto Class; es decir, que estamos usando Class como si fuera el molde para producir una expresión concreta (con características propias) llamada Pasajero. De igual forma, si al objeto Pasajero le añadimos el atributo nombre, en realidad estamos instanciando la metaclase Property.

Por ejemplo, los nombres de pasajeros almacenados en el objeto Pasajero son elementos de la capa M0.

Una de las principales ventajas de este enfoque es que cada uno de los objetos puede ser validado contra su respectivo metaobjeto, en la capa inmediata superior. Por ejemplo, la clase Pasajero debe corresponder a las características y especificaciones de Class en el metamodelo.

MOF define una arquitectura de metamodelado estricta, cada elemento de modelo de cada capa se corresponde estrictamente con un elemento de modelo de la capa superior.

3.3.2. Construcciones básicas utilizadas por MOF

MOF es un estándar del OMG que establece un lenguaje común y abstracto para definir lenguajes de modelado, y cómo acceder e intercambiar modelos expresados en dichos lenguajes. MOF usa cinco construcciones básicas para definir un lenguaje de modelado:

- **Clases:** usadas para definir tipos de elementos en un lenguaje de modelado. Por ejemplo, la relación de dependencia de UML es una clase definida en MOF, que representa el tipo de todas las dependencias que pueden crearse en un modelo UML.

- **Generalización:** define herencia entre clases. Por ejemplo UML::Classifier es una generalización de UML::Class. La subclase hereda todas las características de la clase padre.

- **Atributos:** usados para definir propiedades de elementos del modelo. Los atributos tienen un tipo y una multiplicidad.

- **Asociaciones:** definen relaciones entre clases. Una relación tiene dos extremos, cada uno de los cuales puede tener definido un nombre de rol, navegabilidad y multiplicidad

- **Operaciones:** definen operaciones dentro del ámbito de una clase, junto con una lista de parámetros.

3.4. Paquetes principales del metamodelo de UML

Como podemos ver en la figura 2, el metamodelo de UML está dividido en tres paquetes, que contienen todos los elementos de modelado y los tipos de diagramas que conforman el lenguaje unificado.

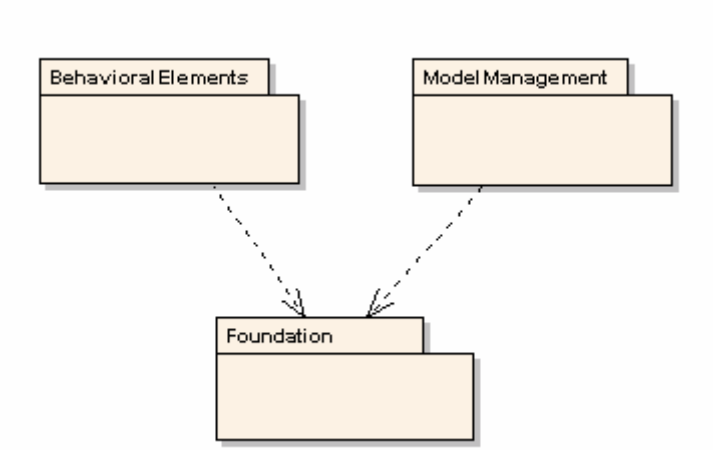


Figura 2 - Paquetes del metamodelo UML

El paquete "Behavioral Elements" (elementos de comportamiento) define los elementos necesarios para representar la dinámica de un sistema, como son los casos de uso, los diagramas de interacción, los diagramas de actividades y los diagramas de estado.

El paquete "Model Management" (manejo de modelo) especifica cómo organizar los elementos de UML en modelos, paquetes y sistemas.

En el paquete "Foundation" (fundamentación) se encuentran los elementos que componen los diagramas de clases, que son utilizados para representar la estructura de un sistema.

Las flechas que aparecen en la figura no representan las interacciones entre los paquetes, sino las dependencias entre éstos. Lo que nos está diciendo el diagrama en la Figura 2 es que si se llegaran a realizar cambios en el paquete Foundation va a ser necesario revisar la repercusión de estos cambios en los paquetes Behavioral Elements y Model Management.

4. Uso de OCL en modelos basados en MOF

Un diagrama UML, como un diagrama de clases, no está lo suficientemente refinado para proveer todos los aspectos relevantes de una especificación. Existe, entre otras cosas, una necesidad por describir restricciones adicionales sobre objetos en el modelo. Tales restricciones son a veces descriptas en lenguaje natural, pero esto puede resultar en ambigüedades. Para escribir restricciones no ambiguas han sido desarrollados los lenguajes formales, que presentan la desventaja de ser utilizados por personas con un gran conocimiento matemático, siendo difíciles de usar por los modeladores de sistemas.

OCL (Object Constraint Language) ha sido desarrollado para llenar este hueco. Es un lenguaje de especificación formal que es fácil de leer y escribir. Al ser un lenguaje de especificación puro se garantiza que toda expresión OCL es libre de efectos laterales; es decir cuando una expresión OCL es evaluada simplemente retorna un valor, sin modificar nada en el modelo.

OCL no es un lenguaje de programación, no es posible escribir lógica de programas o flujo de control en OCL. OCL se puede emplear para especificar restricciones y otras expresiones adjuntas a los modelos MOF.

OCL es un lenguaje tipado, lo que significa que cada expresión tiene un tipo. Para ser bien formada, una expresión debe concordar con los tipos de reglas del lenguaje; por ejemplo, no puede compararse un Integer con un String.

Por lo tanto, el nombre de Object Constraint Language es incorrecto, ya que es posible especificar expresiones que no necesariamente son del tipo Boolean, por ejemplo la definición de valor inicial.

Para los ejemplos se utilizará el diagrama de clases de la figura 3.

En este capítulo se describen las restricciones OCL en la sección 4.1, expresiones de valor inicial en la sección 4.2, de valor derivado en la sección 4.3, y de operación de consulta en la sección 4.4.

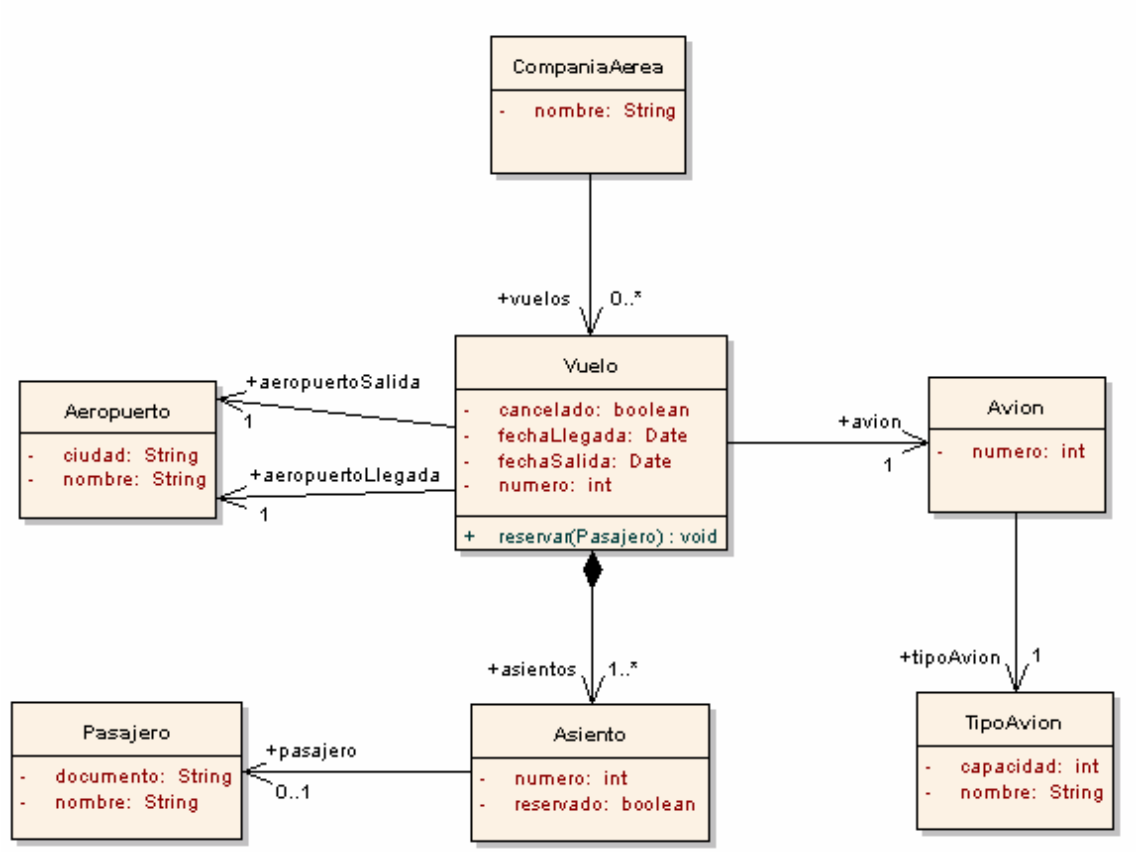


Figura 3 - Ejemplo de diagrama de clases

4.1. Restricciones en OCL

Las restricciones que podemos especificar con OCL son:

- Invariantes
- Definiciones
- Precondiciones
- Postcondiciones

Además de restricciones a nivel modelo podemos definir reglas a nivel metamodelo. Estas son útiles para definir reglas de buena formación (Well – Formedness Rules) y reglas de diseño.

4.1.1. Invariante

Un invariante es una restricción que se liga a un Classifier[2] (Class, Interface, etc). El propósito del invariante es definir una condición que debe ser válida siempre para todas las instancias de un Classifier.

[2] Un Classifier [**15**] es una clasificación de instancias. Describe un conjunto de instancias que tienen características comunes.

La restricción tiene el estereotipo << invariant >>. Los invariantes se ligan a un solo Classifier, y este es el tipo de la variable contextual. En la figura 4 se representa la expresión OCL utilizada como invariante.

Su sintaxis es:

context [VariableName:] TypeName
inv: < OclExpression >

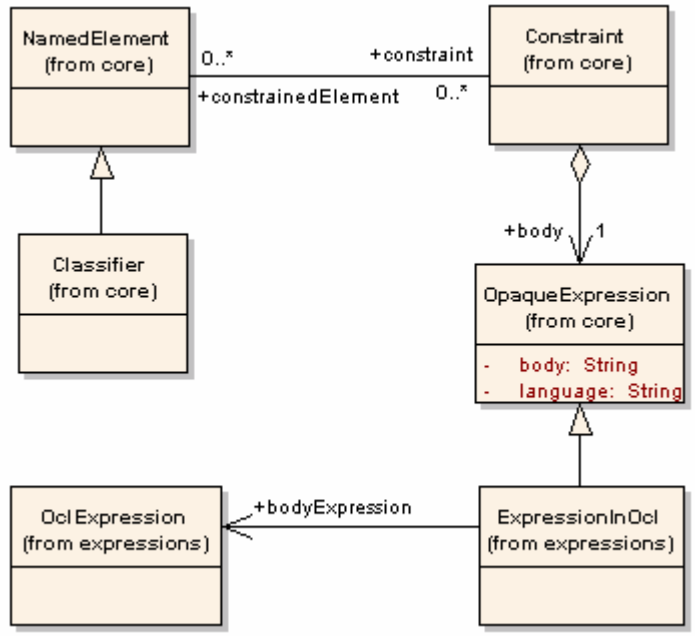


Figura 4 - Expresión OCL utilizada como Invariante o Definición

Ejemplo: la siguiente invariante valida que la fecha de salida de un vuelo no puede ser posterior a la fecha de llegada.

context Vuelo
inv: self.fechaSalida <= self.fechaLlegada

La palabra context introduce el contexto para la expresión. Cada expresión OCL está escrita en el contexto de una instancia de un tipo específico. En una expresión OCL, la palabra reservada self es utilizada para referirse a la instancia contextual. En este caso, self se refiere a una instancia de Vuelo. Otra alternativa, es utilizar un nombre en lugar de self:

context v: Vuelo
inv: v.fechaSalida <= v.fechaLlegada

4.1.2. Definición

Una definición es un "Constraint" que se liga a un Classifier. La variable o función definida puede utilizarse como una propiedad o una operación del correspondiente Classifier. El propósito de esta restricción es definir expresiones OCL reusables.

La restricción tiene el estereotipo << definition >>. Las definiciones se ligan a un solo Classifier, y este es el tipo de la variable contextual. En la figura 4 se representa la expresión OCL utilizada como definición. El concepto de Constraint es incorrecto ya que una restricción debe ser verdadera o falsa y una definición puede retornar cualquier valor.

A continuación se define su sintaxis:

context [VariableName:] TypeName
def: [VariableName] | [OperationName(ParameterName1: Type, ...)]:
ReturnType = < OclExpression >

Ejemplo: en la siguiente expresión OCL se define una variable llamada capacidad, que retorna la capacidad del avión asignado al correspondiente vuelo.

context Vuelo
def: capacidad : Integer = self.avion.tipoAvion.capacidad

La variable capacidad es conocida en el contexto de Vuelo.

context Vuelo
inv: self.asientos -> select (a | a.reservado) -> size() <= self.capacidad

4.1.3. Precondición

Una precondición es una restricción que se liga a un Operation[3] de un Classifier. Esta restricción establece una condición que debe cumplirse antes de ejecutar la operación.

La restricción tiene el estereotipo << precondition >>. Las precondiciones se ligan a un solo BehavioralFeature[4] (Operation), y el tipo de la variable contextual es el Classifier que define la operación. En la figura 5 se ilustra la expresión OCL utilizada como precondición.

Su sintaxis es:

context Typename :: operationName(parameter1 : Type1, ...): ReturnType
pre: < OclExpression >

---

[3] Un Operation [**15**] es un BehavioralFeature que declara un servicio que puede ser llevado a cabo por las instancias de un Classifier.

[4] Un BehavioralFeature [**15**] es una característica de un Classifier que especifica un aspecto del comportamiento de sus instancias.

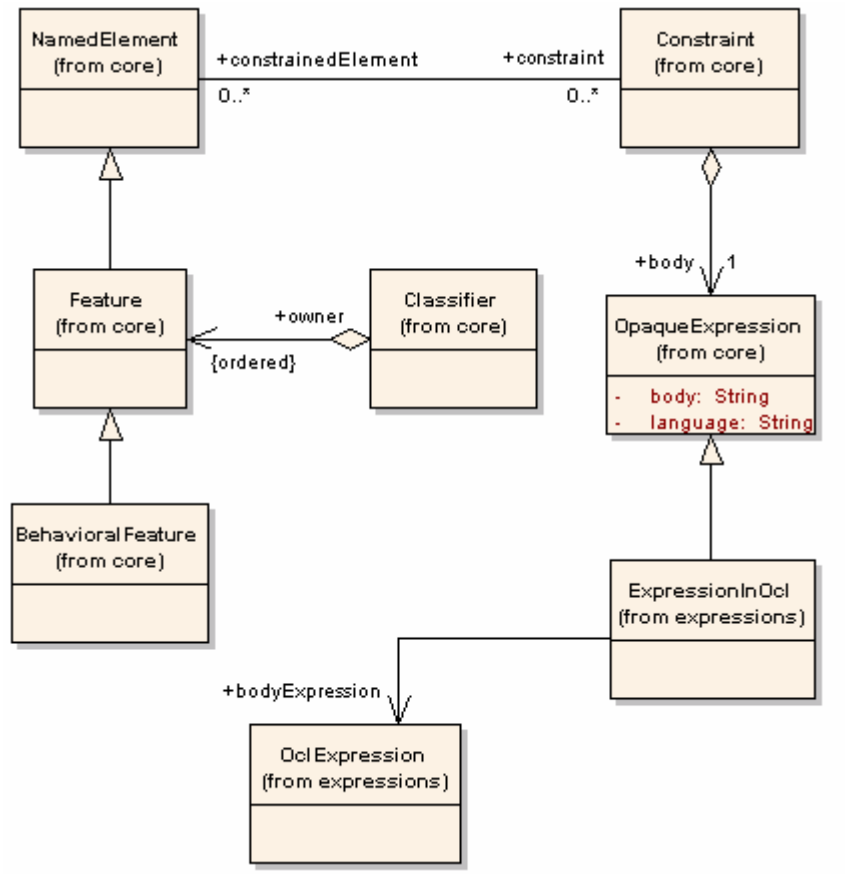


Figura 5 - Expresión OCL utilizada como Precondición o Postcondición

Ejemplo: la precondición de la operación reservar(pasajero) definida en la clase Vuelo, especifica que este no puede estar en estado cancelado y que exista algún asiento disponible.

context Vuelo: reservar(pasajero: Pasajero) : Boolean
pre: not self.cancelado and
self.asientos -> select(a | a.reservado) -> size () < self.capacidad

La variable capacidad es la definida anteriormente en el contexto de Vuelo.

En la expresión se puede utilizar la variable self para referenciar a un objeto del tipo que define la operación.

4.1.4. Postcondición

Una postcondición es una restricción que se liga a un Operation de un Classifier. El propósito de esta restricción es definir la condición que debe cumplirse luego de ejecutar la operación. Una postcondición consiste en una expresión OCL de tipo Boolean. En el caso de los invariantes las restricciones se deben cumplir en todo momento, en cambio en las precondiciones y postcondiciones deben cumplirse antes y después de ejecutar la operación. En una expresión OCL utilizada como postcondición los elementos se pueden decorar con el postfijo "@pre" para hacer referencia al valor del elemento al comienzo de la operación. La variable result se refiere al valor de retorno de la operación.

La restricción tiene el estereotipo << postcondition >>. Las postcondiciones se ligan a un solo BehavioralFeature (Operation), y el tipo de la variable contextual es el Classifier que define dicha BehavioralFeature. En la figura 5 se presenta la expresión OCL utilizada como postcondición.

A continuación se define su sintaxis:

context Typename :: operationName(parameter1 : Type1, ...): ReturnType
post: < OclExpression >

Ejemplo: la operación reservar(pasajero) definida en la clase Vuelo, retorna verdadero si el pasajero dado está asignado al correspondiente vuelo, en caso contrario retorna falso.

context Vuelo: reservar(pasajero: Pasajero) : Boolean
post: result = self.asientos -> exists (a | a.reservado and a.pasajero = pasajero)

El nombre result es el nombre del objeto retornado, si existe alguno. Los nombres de los parámetros también pueden ser utilizados en la expresión OCL.

4.1.5. Reglas a nivel metamodelo

Además de restricciones a nivel modelo podemos definir reglas a nivel metamodelo. Estas son útiles para definir reglas de buena formación (Well – Formedness Rules) y reglas de diseño. Las primeras nos aseguran que nuestro modelo está bien formado, por ejemplo que no existan atributos con el mismo nombre en un Classifier. En cambio las reglas de diseño nos permiten mejorar el modelo y pueden ser útiles al momento de generar código; por ejemplo, que los Classifiers concretos no pueden definir operaciones abstractas.

Ejemplo:

a) Regla de buena formación:

La metaclase Association representa la clase de las asociaciones de los modelos, es decir las asociaciones de los modelos son instancias de dicha metaclase.

context Association
def: allConnections() : Set(AssociationEnd) = self.connection

allConections es una operación adicional definida en Association que retorna todos los extremos de una determinada asociación.

context Association
inv: self.allConections -> forAll (p, q | p.name = q.name implies p = q)

La regla anterior valida que los extremos de cada una de las asociaciones tengan distinto nombre de rol.

b) Regla de diseño:

context Class
inv: self.operations() -> exists (op | op.isAbstract) implies self.isAbstract

Esta regla dice que si las clases del modelo (Vuelo, Avion, etc) tienen alguna operación abstracta entonces estas no pueden ser concretas.

4.2. Expresión de valor inicial

Una expresión de valor inicial es una expresión que se liga a un Property[5]. Una expresión OCL que actúa como el valor inicial debe conformar al tipo definido por la propiedad. Además hay que tener en cuenta su multiplicidad, es decir si la multiplicidad es mayor que uno el tipo es un Set u OrderedSet del tipo de la propiedad. En la figura 6 se representa la expresión utilizada para definir el valor inicial de una propiedad.

Su sintaxis es:

context Typename :: propertyName: Type
init: -- alguna expresión representando el valor inicial de la propiedad

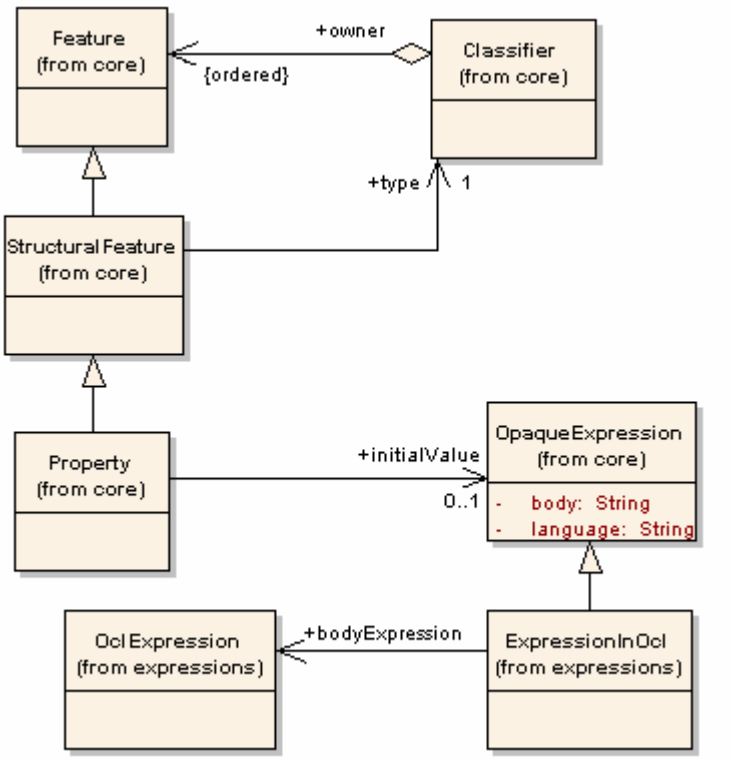


Figura 6 - Expresión OCL utilizada como Valor Inicial de una Propiedad

Ejemplo: la propiedad llamada cancelado de la clase Vuelo tiene asignado un valor inicial de falso.

---

5 Un Property [**15**] es un elemento tipado que representa un atributo de una clase.

context Vuelo :: cancelado : Boolean
init : false

4.3. Expresión de valor derivado

Una expresión de valor derivado es una expresión que se liga a un Property. La expresión OCL que actúa como el valor derivado de una propiedad debe conformar al tipo de esta. De igual modo que cuando definimos el valor inicial, debemos tener en cuenta la multiplicidad de la propiedad. Si esta es mayor que uno el tipo es un Set u OrderedSet del tipo de la propiedad actual.

Una expresión de valor derivado se metamodela como un invariante, el cual especifica el valor de la propiedad.

Su sintaxis es:

context Typename :: propertyName: Type
derive: -- alguna expresión representando la regla de derivación

Ejemplo: se define una propiedad derivada llamada tipoAvion en la clase Vuelo.

context Vuelo :: tipoAvion : TipoAvion
derive: self.avion.tipoAvion

4.4. Expresión de consulta

Una expresión de consulta es una expresión que se liga a una operación de consulta definida en un Classifier. Para indicar que es una operación de consulta se utiliza el atributo isQuery. En la figura 7 se representa el metamodelo de dicha expresión.

Su sintaxis es:

context Typename :: operationName(parameter1: Type1, . . .) : ReturnType
body: -- alguna expresión

La expresión debe ser conforme con el tipo de la operación. Al igual que en las precondiciones y postcondiciones, los parámetros pueden ser utilizados en la expresión.

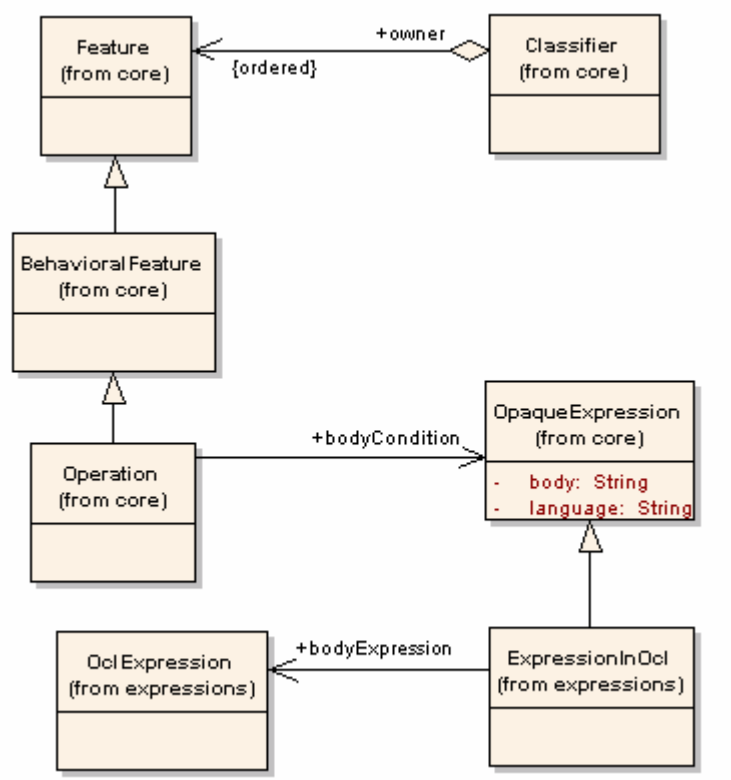


Figura 7 - Expresión OCL utilizada en una Operación de Consulta

Ejemplo: se define la operación de consulta getCapacidad, definida en la clase avión

context Avion :: getCapacidad() : Integer
body: self.tipoAvion.capacidad

Las precondiciones, postcondiciones y expresiones de consulta pueden ser combinadas luego de especificar el contexto de una operación.

context Asiento :: getPasajeroQueReservo() : Pasajero
pre: self.reservado
body: self.pasajero

5. Arquitectura dirigida por modelos

En el año 2001, el OMG estableció el framework MDA (Arquitectura Dirigida por Modelos) como arquitectura para el desarrollo de aplicaciones. MDA representa un nuevo paradigma de desarrollo de software en el que los modelos guían todo el proceso de desarrollo. Este nuevo paradigma se ha denominado "Ingeniería de Modelos" o "Desarrollo basado en Modelos".

En la actualidad, la construcción de software se enfrenta a continuos cambios en las tecnologías de implementación, lo que implica realizar esfuerzos importantes tanto en el diseño de la aplicación, para integrar las diferentes tecnologías que incorpora, como en el mantenimiento, para adaptar la aplicación a cambios en los requisitos y en las tecnologías de implementación. La idea clave que subyace a MDA es que si el desarrollo está guiado por los modelos de software, se obtendrán beneficios importantes en aspectos fundamentales como son la productividad, la portabilidad, la interoperabilidad y el mantenimiento.

Para conseguir estos beneficios, MDA plantea el siguiente proceso de desarrollo: de los requisitos se obtiene un modelo independiente de la plataforma (PIM), luego este modelo es transformado con la ayuda de herramientas en uno o más modelos específicos de la plataforma (PSM), y finalmente cada PSM es transformado en código. Por lo tanto, MDA incorpora la idea de transformaciones entre modelos (PIM a PSM, PSM a código), por lo que se necesitan herramientas para automatizar esta tarea. Estas herramientas de transformación son, de hecho, uno de los elementos básicos de MDA.

En la sección 5.1. se hace una introducción a MDA, en la sección 5.2 se describen los distintos tipos de modelos en MDA, en la sección 5.3 se compara el desarrollo de software tradicional contra el desarrollo en MDA, y por último en la sección 5.4. se detalla la importancia del metamodelado en MDA.

5.1. Introducción a MDA

Según el OMG [**3**], MDA proporciona una solución para los cambios de negocio y de tecnología, permitiendo construir aplicaciones independientes de la plataforma, e implementarlas en plataformas como CORBA, J2EE o Servicios Web. La figura 8, obtenida de la página de MDA del OMG [**3**], pretende resumir lo más importante de este nuevo framework, mencionando los estándares relacionados y las principales ventajas de la aplicación de MDA.

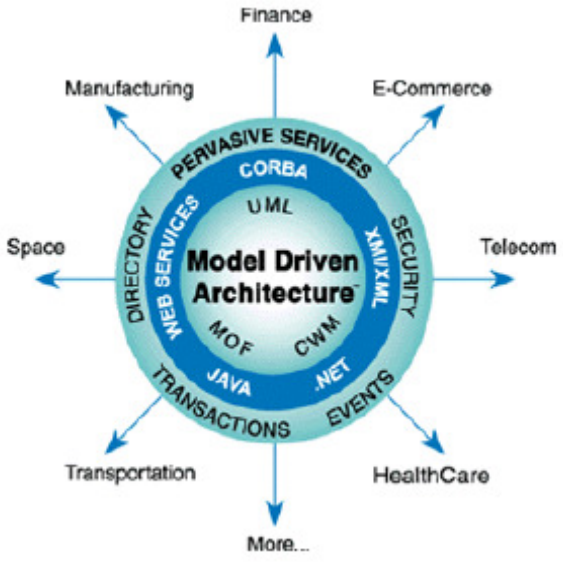


Figura 8 - Logotipo de MDA

MDA es un framework de desarrollo de software que define una nueva forma de construir software en la que se usan modelos del sistema a distintos niveles de abstracción para guiar todo el proceso de desarrollo, desde el análisis y diseño hasta el mantenimiento del sistema y su integración con futuros sistemas.

MDA pretende separar, por un lado, la especificación de las operaciones y datos de un sistema, y por el otro, los detalles de la plataforma en la que el sistema será construido. Para ello, MDA proporciona las bases para:

- Definir un sistema independiente de la plataforma sobre la que se construye.
- Definir plataformas sobre las que construir los sistemas.
- Elegir una plataforma particular para el sistema.
- Transformar la especificación inicial del sistema a la plataforma elegida.

Los principales objetivos de MDA son mejorar la productividad, la portabilidad, la interoperabilidad y la reutilización de los sistemas.

A grandes rasgos, el proceso de desarrollo de software con MDA se puede dividir en tres fases:

- Construcción de un Modelo Independiente de la Plataforma (PIM), un modelo de alto nivel del sistema independiente de cualquier tecnología o plataforma.
- Transformación del modelo anterior a uno o varios Modelos Específicos de la Plataforma (PSM). Un PSM es un modelo de más bajo nivel que el PIM, que describe el sistema de acuerdo con una tecnología de implementación determinada.
- Generación de código a partir de cada PSM. Debido a que cada PSM está muy ligado a una tecnología concreta, la transformación de cada PSM a código puede automatizarse.

El paso de PIM a PSM y de PSM a código no se realiza “a mano”, sino que se utilizan herramientas de transformación para automatizar estas tareas. La figura 9 muestra de

manera resumida el proceso de desarrollo con MDA, suponiendo que tenemos un único PSM.

Figura 9 - Pasos en el desarrollo con MDA

5.2. Modelos en MDA

Un modelo es una descripción de todo o parte de un sistema escrito en un lenguaje bien definido. El hecho de que un modelo esté escrito en un lenguaje bien definido tiene una gran importancia para MDA, ya que supone que el modelo tiene asociadas una sintaxis y una semántica bien definidas. Esto permite la implementación automática por parte de transformadores o compiladores de modelos, fundamentales en MDA.

UML es un lenguaje de modelado bien definido que se ha adoptado como el principal lenguaje de modelado en MDA, aunque se puede usar cualquier lenguaje bien definido.

Entre los distintos tipos de modelos definidos en UML, los Modelos de Clases (que muestran la vista estática del sistema) son los más importantes dentro de MDA, ya que el PIM y la mayoría de los PSMs son modelos de este tipo. Estos modelos se representan mediante Diagramas de Clases de UML.

5.2.1. PIM

Un Modelo Independiente de Plataforma o PIM es un modelo de sistema de alto nivel que representa la estructura, funcionalidad y restricciones del sistema, sin aludir a una plataforma determinada. Este modelo servirá de base para todo el proceso de desarrollo y es el único que debe ser creado íntegramente por el desarrollador.

Al no incluir detalles específicos de una tecnología determinada, este modelo es útil en dos aspectos:

- Es fácilmente comprensible por los usuarios del sistema y, por lo tanto, les resultará más sencillo validar la corrección del sistema.
- Facilita la creación de diferentes implementaciones del sistema en diferentes plataformas, dejando intacta su estructura y su funcionalidad básica.

5.2.2. PSM

Un Modelo Específico de la Plataforma o PSM es un modelo del sistema con detalles específicos de la plataforma en la que será implementado. Se genera a partir del PIM, así que representa el mismo sistema pero a distinto nivel de abstracción. Podemos decir que un PSM es un PIM al que se le añaden detalles específicos para ser implementado en una plataforma determinada.

El PSM es, pues, un modelo de sistema de más bajo nivel, mucho más cercano a la vista de código que el PIM. Puede incluir más o menos detalles, dependiendo de su propósito.

Hay que destacar que a partir de un mismo PIM pueden generarse varios PSMs, cada uno describiendo el sistema desde una perspectiva diferente.

La transformación de PIM a PSM puede llevarse a cabo de varias formas:

- Construyendo manualmente el PSM a partir del PIM.
- De forma semiautomática, generando un PSM esqueleto que es completado a mano.
- De forma totalmente automática, generando un PSM completo a partir del PIM.

Para las transformaciones automáticas se usan herramientas especializadas. Estas herramientas tienen implementados distintos algoritmos de transformación para pasar de un tipo de modelo a otro.

Un PSM también puede refinarse, transformándose sucesivamente en PSMs de más bajo nivel, hasta llegar al punto en que pueda ser transformado a código de manera directa.

A partir del PSM, y gracias nuevamente a una herramienta de transformación, se obtiene gran parte del código que implementa el sistema para la plataforma elegida. El desarrollador tan solo tendrá que añadir aquella funcionalidad que no puede representarse mediante el PIM o el PSM.

5.3. Desarrollo tradicional Vs. Desarrollo con MDA

En el capítulo 1 de MDA Explained [**4**] se realiza una interesante comparativa entre el desarrollo tradicional y el desarrollo con MDA, ilustrando de este modo los beneficios de este nuevo proceso de desarrollo.

5.3.1. Problemas de desarrollo tradicional

Un proceso típico de desarrollo de software incluye las siguientes fases:

1. Especificación de requisitos
2. Análisis
3. Diseño
4. Codificación
5. Prueba
6. Despliegue

La figura 10 esquematiza el proceso de desarrollo de software tradicional.

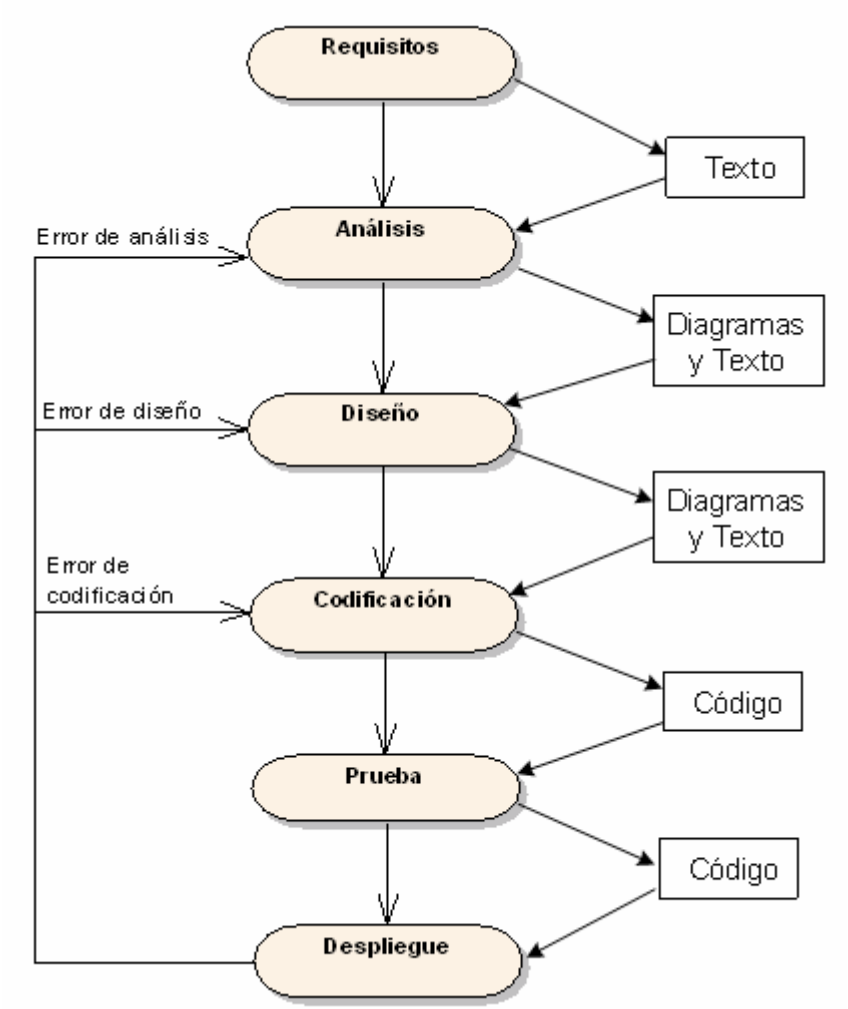


Figura 10 - Proceso de desarrollo de software tradicional

Durante los últimos años se ha progresado bastante en el desarrollo de software, permitiendo construir sistemas más grandes y complejos. Aún así, la construcción de software de la manera tradicional sigue teniendo múltiples problemas:

- **Productividad.** El proceso tradicional produce una gran cantidad de documentos y diagramas para especificar requisitos, clases, colaboraciones, etc. La mayoría de este material pierde su valor en cuanto comienza la fase de codificación, y gradualmente se va perdiendo la relación entre los diagramas. Y más aún cuando el sistema cambia a lo largo del tiempo: realizar los cambios en todas las fases (requisitos, análisis, diseño,…) se hace inmanejable, así que generalmente se realizan las modificaciones sólo en el código. Lo cierto es que para sistemas complejos, los diagramas y la documentación de alto nivel son necesarios; entonces lo que necesitamos es un soporte para que un cambio en cualquiera de las fases se traslade fácilmente al resto.

- **Portabilidad.** En la industria del software cada año aparecen nuevas tecnologías y las empresas necesitan adaptarse a ellas, bien porque la demanda de esta tecnología es alta, o porque realmente resuelve problemas importantes. Como consecuencia el software existente debe adaptarse o migrar a la nueva tecnología. Esta migración no es ni mucho menos trivial, y obliga a las empresas a realizar un importante desembolso.

- **Interoperabilidad.** La mayoría de los sistemas necesitan comunicarse con otros, probablemente ya construidos. Incluso si los sistemas que van a interoperar se construyen desde cero, frecuentemente usan tecnologías diferentes. Necesitamos que la interoperabilidad entre sistemas, nuevos o ya existentes, se consiga de manera sencilla y uniforme.

- **Mantenimiento y documentación.** Documentar un proyecto de software es una tarea lenta que consume mucho tiempo, y que en realidad no interesa tanto a los que desarrollan el software, sino a aquellos que lo modificarán o lo usarán más adelante. Esto hace que se ponga poco empeño en la documentación y que generalmente no tenga una buena calidad. La solución de este problema a nivel de código es que la documentación se genere directamente del código fuente, asegurándonos que este siempre actualizada[6]. No obstante, la documentación de alto nivel (diagramas y texto) todavía debe ser mantenida a mano.

Estos problemas se solucionan con MDA, como veremos a continuación.

5.3.2. Beneficios del MDA

A continuación explicaremos como MDA resuelve cada uno de los problemas del desarrollo tradicional expuestos en la sección anterior.

- **Productividad.** En MDA el foco de desarrollo recae sobre el PIM. Los PSMs se generan automáticamente (al menos en gran parte) a partir del PIM. Por supuesto, alguien tiene que definir las transformaciones exactas, lo cual es una tarea especializada y difícil. Pero una vez implementada la transformación, puede usarse en muchos desarrollos. Y lo mismo ocurre con la generación de código a partir de los PSMs. Este enfoque centrado en el PIM aísla los problemas específicos de cada plataforma y encaja mucho mejor con las necesidades de los usuarios finales, puesto que se puede añadir funcionalidad con menos esfuerzo. El trabajo "sucio" recae sobre las herramientas de transformación, y no sobre el desarrollador.

- **Portabilidad.** En MDA la portabilidad se logra también enfocando el desarrollo sobre el PIM. Al ser un modelo independiente de cualquier tecnología, todo lo definido en él es totalmente portable. Otra vez el peso recae sobre las herramientas de transformación, que realizarán automáticamente el paso de PIM al PSM de la plataforma deseada.

- **Interoperabilidad.** Los PSMs generados a partir de un mismo PIM normalmente tendrán relaciones, que es lo que en MDA se llaman "puentes". Normalmente los distintos PSMs no podrán comunicarse entre ellos directamente, ya que pueden pertenecer a distintas tecnologías. Este problema lo soluciona MDA generando no sólo los PSMs, sino también los puentes entre ellos. Como es lógico, estos puentes serán construidos por las herramientas de transformación, que como vemos son uno de los pilares de MDA.

---

6 Un ejemplo de esto es la herramienta Javadoc para Java.

- **Mantenimiento y Documentación.** Como ya se mencionó, a partir del PIM se generan los PSMs, y a partir de los PSMs se genera el código. Básicamente, el PIM desempeña el papel de la documentación de alto nivel que se necesita para cualquier sistema de software. Pero la gran diferencia es que el PIM no se abandona tras la codificación. Los cambios realizados en el sistema se reflejarán en todos los niveles, mediante la regeneración de los PSMs y del código. Aún así, seguimos necesitando documentación adicional que no puede expresarse con el PIM, por ejemplo para justificar las elecciones hechas para construir el PIM.

5.3.3. El nuevo proceso de desarrollo

Si comparamos el proceso MDA con el proceso tradicional de desarrollo de software, observaremos que lo que cambia son las fases de análisis, diseño y codificación:

- **Análisis:** un grupo especial de personas desarrollarán el PIM, guiados por las necesidades del negocio y la funcionalidad que debe incorporar el sistema.

- **Diseño:** otro grupo diferente de personas se encargarán de la transformación del PIM a uno o más PSMs. Estas personas tendrán conocimientos sobre distintas plataformas y arquitecturas, y conocerán también las transformaciones disponibles en las herramientas que usan. De este modo podrán elegir la plataforma o arquitectura que mejor se adapte a los requisitos del sistema y establecer los parámetros de las distintas transformaciones. Los creadores de PSMs tendrán comunicación constante con los diseñadores del PIM para tener más información sobre el sistema (por ejemplo, para conocer los requisitos no funcionales).

- **Codificación:** esta fase se reduce a generar el código del sistema mediante herramientas especializadas. Los programadores únicamente tendrán que añadir la funcionalidad que no puede reflejarse en los modelos y, si es necesario, "retocar" el código generado.

Pero en este nuevo proceso de desarrollo aun falta un tercer grupo de personas, aquellos que escriben definiciones de transformaciones para empresas de construcción de software o, principalmente, para los vendedores de herramientas de MDA. Estas transformaciones son fundamentales para construir software con MDA, pues permiten a las herramientas pasar de PIM a PSM y de PSM a código.

La figura 11 muestra un esquema sencillo que ilustra este nuevo proceso de desarrollo con MDA. Vemos como los cambios se centran en el PIM, base de todo el desarrollo, y gracias a las herramientas de transformación estos cambios se trasladan rápidamente al resto de las fases.

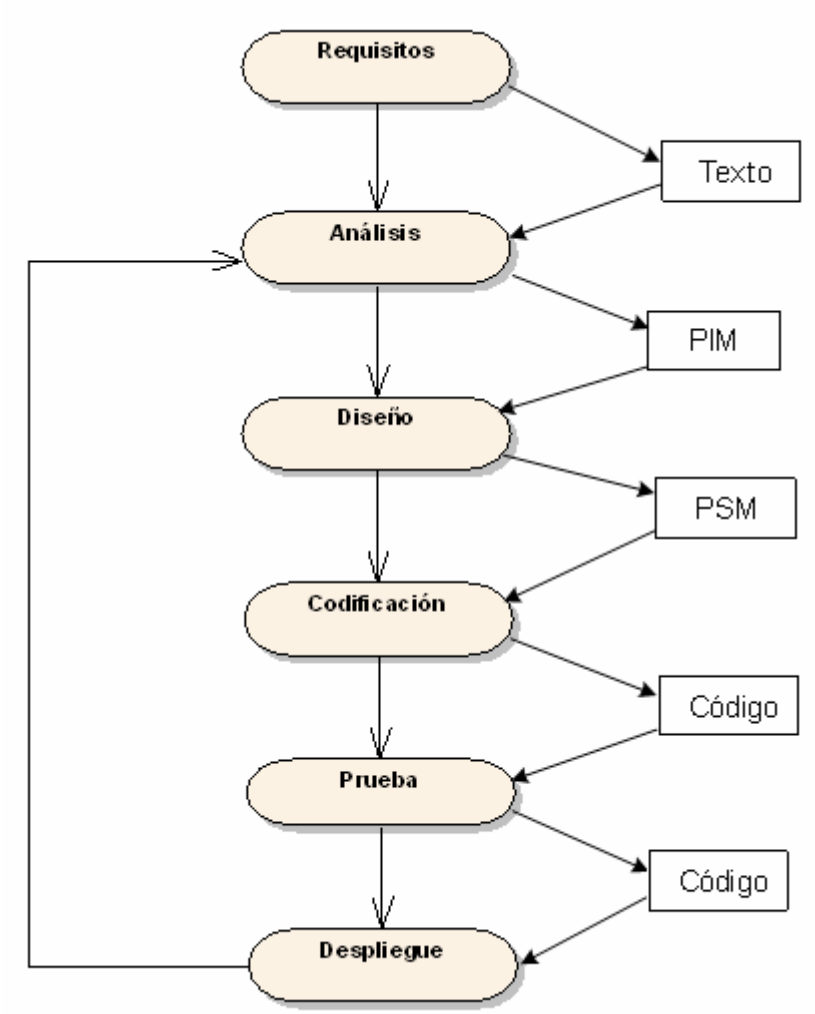


Figura 11 - Proceso de desarrollo con MDA

5.4. Importancia del metamodelado en MDA

En primer lugar, el metamodelado es importante en MDA porque actúa como mecanismo para definir lenguajes de modelado, de forma que su definición no sea ambigua. Esta no ambigüedad permite que una herramienta pueda leer, escribir y entender modelos como los construidos con UML.

Por otro lado, las reglas de transformación usadas para transformar un modelo en lenguaje A en otro modelo en lenguaje B, usan los metamodelos de los lenguajes A y B para definir las transformaciones.

6. Estrategias de evaluación para las condiciones de refinamiento de modelos MOF

En MDA las transformaciones predefinidas, escritas en un lenguaje de transformación estándar QVT [**5**] se aplican en orden para evolucionar de modelo a modelo. Se supone que tales transformaciones se han validado previamente por un experto de MDA, que asegura que estas son refinamientos en el sentido de los lenguajes formales:

Refinamiento es el proceso de desarrollo de un diseño o implementación más detallado que una especificación abstracta, a través de una sucesión de pasos matemáticos que mantienen la correctitud con respecto a la especificación original.

A pesar de la importancia que tiene la técnica de refinamiento en el acercamiento formal de la ingeniería de software, el concepto de refinamiento en MDA se define débilmente y se abre a malas interpretaciones. Este inconveniente surge por la presente semi-formalidad de los lenguajes de modelado utilizados en MDA y también debido a la inmadurez actual en este campo.

Hay dos alternativas para aumentar la robustez del mecanismo de refinamiento del MDA. Una de ellas es traducir el núcleo del lenguaje utilizado en MDA, es decir UML, en un lenguaje formal como Z [**16**], donde las propiedades son definidas y analizadas. La otra alternativa es promover una definición formal de refinamiento, por ejemplo simulación en Z, y expresarlo en términos de MDA. Esta última alternativa, explorada en [**17**] y [**18**], consiste en definir estructuras de refinamientos en UML/OCL equivalentes a las estructuras de refinamientos en los lenguajes formales. En [**19**] se enriquece dicha propuesta definiendo condiciones de refinamiento escritas en OCL.

La ventaja de este acercamiento es que las condiciones de refinamiento se definen completamente en términos de UML y OCL y hacen innecesario la aplicación de lenguajes formales que normalmente no son aceptados por ingenieros de software. A pesar de que las condiciones de refinamiento no se expresan con un lenguaje formal, tienen un fundamento formal proporcionado por Object-Z [**16**].

Por otro lado, los evaluadores de OCL tradicionales son incapaces de determinar si una condición de refinamiento escrita en OCL se cumple en un modelo UML, porque se evalúan fórmulas OCL en una instancia particular del modelo, mientras que las condiciones de refinamiento necesitan ser validadas en todas las posibles instanciaciones. Por lo tanto para evaluar las condiciones de refinamiento, se extrae del modelo UML un número relativamente pequeño de instanciaciones, y se verifica si satisfacen las condiciones mencionadas. Esta estrategia, llamada micromodelos de software, fue propuesta por Daniel Jackson en [**14**] para evaluar fórmulas escritas en Alloy. Luego, Martin Gogolla en [**15**] desarrolló una adaptación de tal técnica para verificar modelos UML y OCL. La extensión consiste en introducir un lenguaje para definir propiedades de los micromodelos deseados (snapshots) y mostrar cómo se generan, con el propósito de reducir la cantidad de micromodelos a ser considerado. Para verificar las condiciones de refinamientos se adapta dicha estrategia.

En la sección 6.1 se describen informalmente las condiciones de refinamiento, en la sección 6.2 se describe cómo expresar la relación de refinamiento en UML, en la sección 6.3 se describe un método utilizado para crear las condiciones de refinamiento OCL para los patrones UML, y por ultimo en la sección 6.4 se explica cómo se aplica la estrategia de los micromodelos para la evaluación de refinamientos.

6.1. Condiciones de refinamiento

En esta sección se describe informalmente cuales son las condiciones que se deben satisfacer para que exista un refinamiento entre una definición abstracta y una definición concreta. La idea es expresar estas condiciones en lenguaje natural dejando los formalismos de lado, los cuales han sido cubiertos en [**19**].

Las tres condiciones que se deben satisfacer son:

- **Inicialización:** para cada instancia de la especificación concreta, si la instancia (llamémosla c_i) tiene en sus propiedades los valores iniciales, entonces para todas las instancias de la especificación abstracta existe una (a_i) tal que tiene en sus propiedades los valores iniciales y además a_i es un mapeo de c_i.

- **Aplicabilidad:** para cada una de las precondiciones de las operaciones op_{ai} de la especificación abstracta, si existe una precondición para la misma operación op_{ci} en la especificación concreta, se debe cumplir que para todas las instancias de la especificación abstracta a_i que son mapeos de alguna instancia de la especificación concreta c_i, si la precondición de op_{ai} se satisface para a_i entonces la precondición de op_{ci} se satisface para c_i.

- **Correctitud:** para cada una de las postcondiciones de las operaciones op_{ai} de la especificación abstracta, si existe una postcondición para la misma operación op_{ci} en la especificación concreta, se debe cumplir que para todas las instancias de la especificación abstracta a_i que son mapeos de alguna instancia de la especificación concreta c_i, si se satisface la precondición de op_{ai} para a_i y se satisface la postcondición de op_{ci} para c_post_i (c_i luego de la ejecución de op_{ci}), entonces existe una instancia a_post_i (a_i luego de la ejecución de op_{ai}) tal que se cumple la postcondición de op_{ai} para a_post_i, y a_post_i es un mapeo de c_post_i.

Estas tres condiciones se van a entender mejor con los ejemplos que daremos a continuación.

Esta definición permite especificar débilmente las precondiciones y reducir el no determinismo. En particular, la condición de aplicabilidad requiere definir una operación concreta que corresponda con la definición de una operación abstracta, sin embargo también permite definir una operación concreta en estados para los cuales la precondición de la operación abstracta es falsa. Es decir, la precondición de la operación se puede definir débilmente. La correctitud requiere que una operación concreta sea consistente con una abstracta siempre que se aplique en un estado donde la operación abstracta está definida. Sin embargo, el resultado de la operación concreta sólo tiene que ser consistente con la abstracta, pero no idéntico. Así si la operación abstracta permite

varias opciones, la operación concreta es libre de usar cualquier subconjunto de estas opciones. En otras palabras, el no determinismo puede resolverse.

6.2. Relación de refinamiento en UML

El lenguaje de modelado estándar UML [**1**] proporciona un artefacto llamado Abstraction (un tipo de Dependencia) con el estereotipo << refine >> para especificar la relación de refinamiento entre elementos nombrados del modelo. En el metamodelo de UML una Abstraction es una relación dirigida de un client (o clients) a un supplier (o suppliers) declarando que el client (el refinamiento) depende del supplier (la abstracción). El artefacto de Abstraction tiene un meta-atributo llamado mapping que es una documentación explícita de como son mapeados las propiedades de un elemento abstracto a sus versiones refinadas, y en la dirección opuesta, como pueden simplificarse los elementos concretos para ajustarse a una definición abstracta. El mapping contiene una expresión en un lenguaje dado que podría ser formal o no. La definición de refinamiento en UML estándar [**1**] se formula empleando el lenguaje natural y permanece abierto a numerosas interpretaciones contradictorias. En la herramienta ePlatero el mapping de la abstracción es expresado a través del lenguaje OCL.

6.3. Estrategia de verificación para patrones de refinamiento UML

Los patrones de refinamiento UML [**17**] [**18**] [**20**] documentan las repetidas estructuras de refinamiento en modelos UML. En esta sección se presentan algunos de los patrones que han sido estudiados en otros trabajos [**17**] [**18**] así como también una breve descripción del proceso aplicado a modelos UML que contienen tales patrones, para crear automáticamente las condiciones de refinamiento OCL.

La figura 12 presenta una descripción del proceso. Está basado en una arquitectura pipeline en la que el análisis es llevado a cabo por una sucesión de pasos. La salida de cada paso proporciona la entrada del próximo. A continuación se describe brevemente cada paso:

- **Instanciación del patrón de refinamiento**. Cada patrón de refinamiento P consiste en dos partes: una descripción de la estructura del patrón M, dado en términos de diagramas UML y una restricción genérica F expresada en Object-Z que representa la condición de refinamiento para tal patrón. Dado un modelo UML M1 conforme con la estructura del patrón P, el primer paso del proceso genera automáticamente una instancia F1 de la fórmula genérica F que establece las condiciones a ser cumplidas a través de M1 para verificar el refinamiento.

- **Transformación a OCL.** Luego, la fórmula Object-Z F1 se traduce automáticamente en la fórmula OCL F1' aplicando la transformación T.

- **Aplicación de la estrategia de micromodelos.** En este paso, se emplea la estrategia de los micromodelos en F1' para producir una fórmula F1" que es analizable dentro de un ámbito limitado.

- **Evaluación OCL.** Finalmente, F1'' se somete a un evaluador tradicional de OCL.

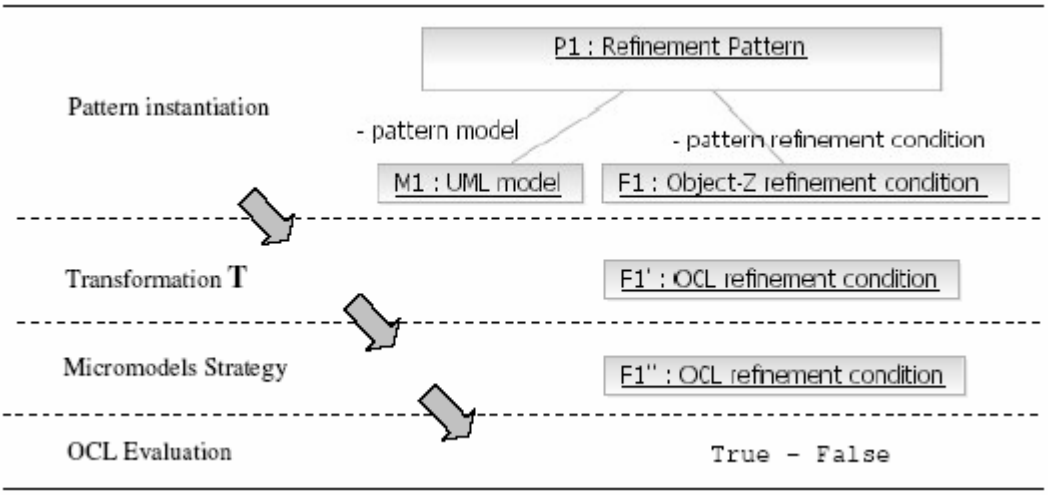


Figura 12 - Proceso de verificación de refinamiento

En las siguientes secciones se presenta el resultado de este proceso aplicado a tres ejemplos concretos:

- Patrón de refinamiento State
- Patrón de refinamiento Object Descomposition.
- Patrón de refinamiento Atomic Operation

En [**10**] se encuentra detallado cada uno de los pasos del proceso aplicados a los tres patrones anteriores.

6.3.1. Patrón State

Un refinamiento State tiene lugar cuando la estructura de datos utilizada para representar los objetos en la especificación abstracta es reemplazada por estructuras más concretas o apropiadas. Se redefinen las operaciones para preservar el comportamiento definido en la especificación abstracta.

En la figura 13 se ilustra el modelo UML M1 que es conforme con la estructura del patrón de refinamiento State [**17**]. M1 contiene información sobre un sistema de reserva de vuelos, donde cada vuelo es descrito abstractamente por la cantidad de asientos libres en su cabina. Un refinamiento es originado por el registro de la capacidad total del vuelo junto con la cantidad de asientos reservados. En ambas especificaciones se utiliza un atributo de tipo int llamado numero que identifica al vuelo, un atributo de tipo int llamado nroAvion que identifica al avión, y un atributo de tipo boolean llamado cancelado que representa el estado del vuelo. Las operaciones disponibles son reservar para hacer una reservación de un asiento, y cancelar para cancelar el vuelo.

El lenguaje OCL [**2**] se utiliza para especificar valores iniciales, precondiciones, postcondiciones y el mapping de la relación de refinamiento. Como una convención se utilizan las variables a y c para referenciar a la especificación abstracta y concreta respectivamente en el mapping de la relación de refinamiento.

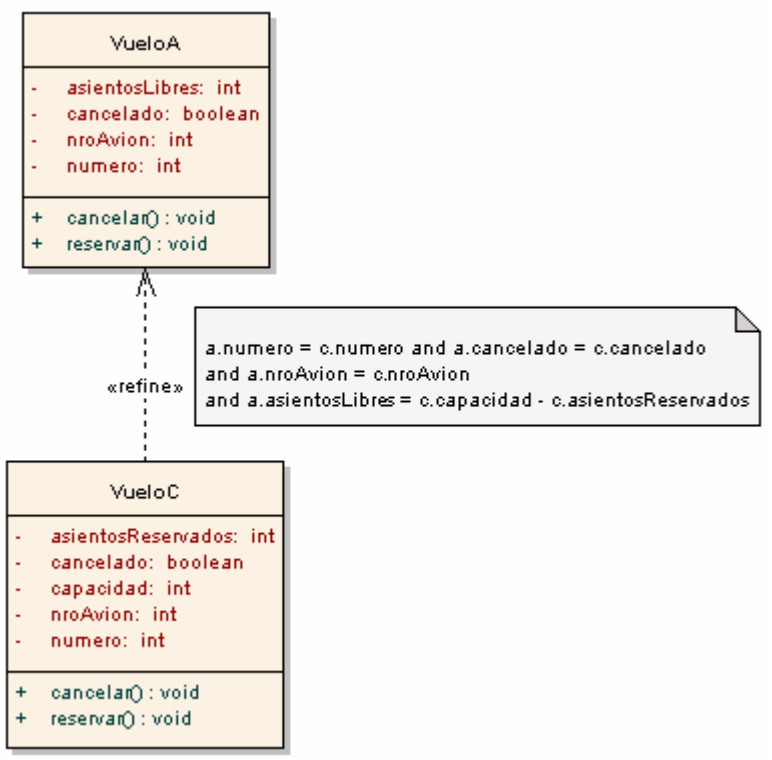


Figura 13 - Instancia del patrón de refinamiento State

A continuación se especifican las restricciones OCL que enriquecen al modelo UML presentado en la figura 13.

Restricciones de la especificación abstracta:

```
context VueloA::asientosLibres: Integer
    init: 300

context VueloA::cancelado: Boolean
    init: false

context VueloA::cancelar()
    pre: not self.cancelado
    post: self.cancelado

context VueloA::reservar()
    pre: self.asientosLibres > 0 and not self.cancelado
    post: self.asientosLibres = self.asientosLibres@pre - 1
```

Restricciones de la especificación concreta:

```
context VueloC::asientosReservados: Integer
    init: 0

context VueloC::cancelado: Boolean
    init: false

context VueloC::capacidad: Integer
    init: 300
```

```
context VueloC::cancelar()
    pre: not self.cancelado
    post: self.cancelado

context VueloC::reservar()
    pre: self.capacidad - self.asientosReservados > 0
         and not self.cancelado
    post: self.asientosReservados = self.asientosReservados@pre + 1
```

Como resultado de los dos primeros pasos del proceso se obtienen las tres condiciones de refinamiento expresadas en OCL, más algunas definiciones OCL utilizadas por las mismas. Este resultado se muestra en la figura 14.

Initialization	`VueloC.allInstances()-> forAll (vueloC | vueloC.isInit() implies (VueloA.allInstances()-> exists (vueloA | vueloA.isInit() and vueloA.mapping(vueloC))))`
Applicability	`VueloA.allInstances()-> forAll (vueloA | VueloC.allInstances()-> forAll (vueloC | vueloA.mapping(vueloC) implies (vueloA.preReservar() implies vueloC.preReservar())))`
Correctness	`VueloA.allInstances()-> forAll (vueloA | VueloC.allInstances()-> forAll (vueloC | VueloC.allInstances()-> forAll (vueloC_post | vueloA.mapping(vueloC) and vueloA.preReservar() and vueloC_post.postReservar(vueloC) implies VueloA.allInstances()-> exists (vueloA_post | vueloA_post.mapping(vueloC_post) and vueloA_post.postReservar(vueloA)))))`

Definitions	`context a: VueloA def: mapping(c: VueloC): Boolean = a.numero = c.numero and a.cancelado = c.cancelado and a.nroAvion = c.nroAvion and a.asientosLibres = c.capacidad - c.asientosReservados`
	
	`context VueloA def: isInit(): Boolean = self.asientosLibres = 300 and self.cancelado = false`
	
	`context VueloC def: isInit(): Boolean = self.capacidad = 300 and self.asientosReservados = 0 and self.cancelado = false`
	
	`context VueloA def: preReservar(): Boolean = self.asientosLibres > 0 and not self.cancelado`
	
	`context VueloC def: preReservar(): Boolean = self.capacidad - self.asientosReservados > 0 and not self.cancelado`
	
	`context VueloA def: postReservar(selfPre: VueloA): Boolean = self.asientosLibres = selfPre.asientosLibres + 1`
	
	`context VueloC def: postReservar(selfPre: VueloC): Boolean = self.asientosReservados = selfPre.asientosReservados + 1`

Figura 14 - Resultado del proceso aplicado a una instancia del patrón de refinamiento

Merece la pena mencionar que la definición del mapping puede traducirse en una fórmula en el contexto del clasificador concreto.

Las operaciones de consulta isInit() retornan verdadero si todos los atributos de la instancia satisfacen la condición de inicialización, y falso en caso contrario.

6.3.2. Patrón Object Descomposition

Un refinamiento Object Descomposition tiene lugar cuando un elemento abstracto se describe con más detalle revelando sus componentes internos.

Siguiendo con el modelo M1 presentado en la figura 13, un refinamiento de la clase VueloC se obtiene especificando con más detalle el hecho de que un vuelo contiene una colección de asientos (figura 15). En este ejemplo el asiento se describe como una entidad individual que tiene estructura y comportamiento. Este conoce su número de identificación y estado (si está reservado o no). La versión refinada de la operación reservar selecciona un asiento (no reservado) de una manera que no reduce el determinismo.

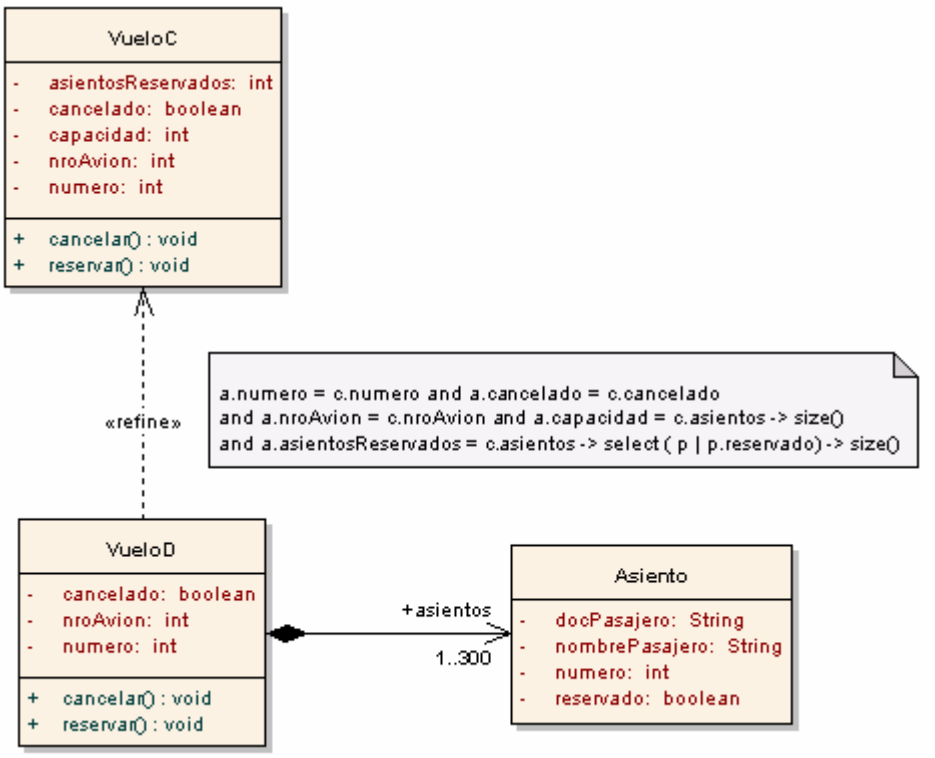


Figura 15 - Instancia del patrón de refinamiento Object Descomposition

A continuación se especifican las restricciones OCL que enriquecen al modelo UML ilustrado en la figura 15.

Restricciones de la especificación abstracta:

```
context VueloC::capacidad: Integer
    init: 300

context VueloC::asientosReservados: Integer
    init: 0

context VueloC::cancelado: Boolean
    init: false

context VueloC::reservar()
    pre: self.capacidad - self.asientosReservados > 0
         and not self.cancelado
    post: self.asientosReservados = self.asientosReservados@pre + 1

context VueloC::cancelar()
    pre: not self.cancelado
    post: self.cancelado
```

Restricciones de la especificación concreta:

```
context VueloD::cancelado: Boolean
    init: false

context VueloD::reservar()
    pre: self.asientos -> exists (p | not p.reservado)
         and not self.cancelado
    post: self.asientos -> exists (p | p.reservado and
          self.asientos@pre -> select (q | not q.reservado)
          -> exists (s | s.numero = p.numero)

context VueloD::cancelar()
    pre: not self.cancelado
    post: self.cancelado

context Asiento::reservado: Boolean
    init: false

context Asiento::reservar()
    pre: not self.reservado
    post: self.reservado
```

Como resultado de los dos primeros pasos del proceso se obtienen las tres condiciones de refinamiento expresadas en OCL, más algunas definiciones OCL utilizadas por las mismas. Este resultado se muestra en la figura 16.

Initialization	`VueloD.allInstances()-> forAll (vueloD | vueloD.isInit() implies (VueloC.allInstances()-> exists (vueloC | vueloC.isInit() and vueloC.mapping(vueloD))))`
Applicability	`VueloC.allInstances()-> forAll (vueloC | VueloD.allInstances()-> forAll (vueloD | vueloC.mapping(vueloD) implies (vueloC.preReservar() implies vueloD.preReservar())))`

Correctness	`VueloC.allInstances()-> forAll (vueloC | VueloD.allInstances()-> forAll (vueloD | VueloD.allInstances()-> forAll (vueloD_post | vueloC.mapping(vueloD) and vueloC.preReservar() and vueloD_post.postReservar(vueloD) implies VueloC.allInstances()-> exists (vueloC_post | vueloC_post.mapping(vueloD_post) and vueloC_post.postReservar(vueloC)))))`

Definitions	`context a: VueloC def: mapping(c: VueloD): Boolean = a.numero = c.numero and a.cancelado = c.cancelado and a.nroAvion = c.nroAvion and a.capacidad = c.asientos -> size() and a.asientosReservados = c.asientos -> select (p | p.reservado) -> size()`
	
	`context VueloC def: isInit(): Boolean = self.capacidad = 300 and self.asientosReservados = 0 and self.cancelado = false`
	
	`context VueloD def: isInit(): Boolean = self.cancelado = false and self.asientos -> size() = 300 and self.asientos -> forAll (s | s.isInit())`
	
	`context Asiento def: isInit(): Boolean = self.reservado = false`
	
	`context VueloC def: preReservar(): Boolean = self.capacidad - self.asientosReservados > 0 and not self.cancelado`
	
	`context VueloD def: preReservar(): Boolean = self.asientos -> exists (p | not p.reservado) and not self.cancelado`
	
	`context VueloC def: postReservar(selfPre: VueloC): Boolean = self.asientosReservados = selfPre.asientosReservados + 1`
	
	`context VueloD def: postReservar(selfPre: VueloD): Boolean = self.asientos -> exists (p | p.reservado and selfPre.asientos -> select (q | not q.reservado) -> exists (s | s.numero = p.numero)`

Figura 16 - Resultado del proceso aplicado a una instancia del patrón de refinamiento

6.3.3. Patrón Atomic Operation

El refinamiento Atomic Operation tiene lugar cuando una especificación más concreta se obtiene de una especificación abstracta reemplazando cualquier operación Aopk por su refinamiento Copk. La operación refinada reduce el no determinismo y/o la parcialidad presente de la operación abstracta.

Para la aplicación de este patrón vamos a seguir con el modelo de la figura 15. La especificación de la operación reservar de la clase VueloD selecciona un asiento (no reservado) de una manera que no reduce el determinismo, mientras que la operación refinada (definida en VueloE) resuelve el no determinismo estableciendo un criterio de selección de asiento (en este caso, el primer asiento disponible del correspondiente vuelo).

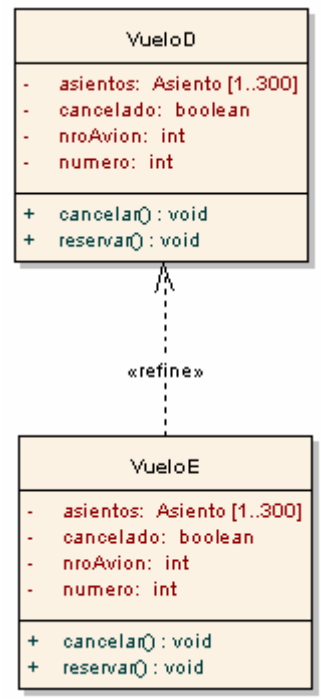


Figura 17 - Instancia del patrón de refinamiento Atomic Operation

Las siguientes restricciones enriquecen al modelo UML planteado en la figura 17.

Restricciones de la especificación abstracta:

```
context VueloD::cancelado: Boolean
    init: false

context VueloD::reservar()
    pre: self.asientos -> exists (p | not p.reservado)
         and not self.cancelado
    post: self.asientos -> exists (p | p.reservado and
          self.asientos@pre -> select (q | not q.reservado)
          -> exists (s | s.numero = p.numero)

context VueloD::cancelar()
    pre: not self.cancelado
    post: self.cancelado
```

Restricciones de la especificación concreta:

```
context VueloE::cancelado: Boolean
    init: false

context VueloE::reservar()
    pre: self.asientos -> exists (p | not p.reservado)
         and not self.cancelado
    post: self.asientos -> exists(p | p.reservado and
          self.asientos@pre -> select (q | not q.reservado)
          -> asSequence().first().numero = p.numero)

context VueloE::cancelar()
    pre: not self.cancelado
    post: self.cancelado
```

Como resultado de los dos primeros pasos del proceso se obtienen las tres condiciones de refinamiento expresadas en OCL, más algunas definiciones OCL utilizadas por las mismas. Este resultado se muestra en la figura 18.

Initialization

```
VueloE.allInstances()-> forAll (vueloE | vueloE.isInit() implies
(VueloD.allInstances()-> exists (vueloD | vueloD.isInit() and
vueloD.mapping(vueloE))))
```

Applicability

```
VueloD.allInstances()-> forAll (vueloD |
VueloE.allInstances()-> forAll (vueloE |
vueloD.mapping(vueloE) implies (vueloD.preReservar() implies
vueloE.preReservar())))
```

Correctness

```
VueloD.allInstances()-> forAll (vueloD |
VueloE.allInstances()-> forAll (vueloE |
VueloE.allInstances()-> forAll (vueloE_post |
vueloD.mapping(vueloE) and vueloD.preReservar()
and vueloE_post.postReservar(vueloE) implies
VueloD.allInstances()-> exists (vueloD_post |
vueloD_post.mapping(vueloE_post) and
vueloD_post.postReservar(vueloD)))))
```

Definitions

```
context a: VueloD def: mapping(c: VueloE): Boolean =
a.numero = c.numero and a.cancelado = c.cancelado
and a.nroAvion = c.nroAvion
and a.asientos -> size() = c.asientos -> size()
and a.asientos -> forAll (p| c.asientos -> exists (q | p.mapping(q)))

context a1: Asiento def: mapping(a2: Asiento): Boolean =
a1.numero = a2.numero and a1.reservado = a2.reservado
and a1.nombrePasajero = a2.nombrePasajero
and a1.docPasajero = a2.docPasajero

context VueloD def: isInit(): Boolean =
self.cancelado = false and self.asientos -> size() = 300
and self.asientos -> forAll (s | s.isInit())

context VueloE def: isInit(): Boolean =
self.cancelado = false and self.asientos -> size() = 300
and self.asientos -> forAll (s | s.isInit())

context Asiento def: isInit(): Boolean =
self.reservado = false

context VueloD def: preReservar(): Boolean =
self.asientos -> exists (p | not p.reservado) and not self.cancelado

context VueloE def: preReservar(): Boolean =
self.asientos -> exists (p | not p.reservado) and not self.cancelado

context VueloD def: postReservar(selfPre: VueloD): Boolean =
self.asientos -> exists (p | p.reservado and
selfPre.asientos -> select (q | not q.reservado)
-> exists (s | s.numero = p.numero)

context VueloE def: postReservar(selfPre: VueloE): Boolean =
self.asientos -> exists (p | p.reservado and
selfPre.asientos -> select (q | not q.reservado) -> asSequence() ->
first().numero = p.numero)
```

Figura 18 - Resultado del proceso aplicado a una instancia del patrón de refinamiento

6.4. Micromundos para la evaluación de las condiciones de refinamiento

Incluso los modelos pequeños como el presentado en la figura 15 especifican un número infinito de instancias; por lo tanto en un principio es imposible afirmar si un modelo cumple con una cierta propiedad.

Para hacer viable la evaluación de las condiciones de refinamiento, se aplica la técnica de micromodelos de software definiendo un límite finito de instancias (micromundo) para luego verificar si estas cumplen con la propiedad, teniendo en cuenta ciertas consideraciones:

- Si la respuesta es positiva, hay una posibilidad de que la propiedad se cumple. En este caso, la respuesta no es definitiva porque puede haber un mundo más grande que no cumple con la propiedad, sin embargo la respuesta positiva es alentadora.
- Si la respuesta es negativa, entonces existe al menos un mundo que viola la propiedad. En ese caso, la respuesta es definitiva ya que la propiedad no se cumple en el modelo.

La hipótesis del pequeño ámbito de Jackson [**14**] expresa que las respuestas negativas tienden a ocurrir en mundos pequeños, entonces las respuestas positivas son muy alentadoras.

Por ejemplo, para generar el micromundo utilizado para la evaluación de las condiciones de refinamiento del diagrama de clases de la figura 15, se proporcionan los invariantes de la figura 19 que reduce el tamaño de estos.

```
package vuelos
  context VueloC
    inv: Set {4} -> includes (self.capacidad)
    inv: Set {0..4} -> includes (self.asientosReservados)
  context Asiento
    inv: Set {1..4} -> includes (self.numero)
  context VueloD
    inv: self.asientos -> forAll (p, q | p.numero = q.numero implies p = q )
  context VueloC
    inv: self.asientosReservados <= self.capacidad
endpackage
```

Figura 19 - Invariantes OCL que reducen el espacio de búsqueda

Por ejemplo, la figura 20 ilustra un micromundo que satisface los invariantes presentados en la figura 19. En tal micromundo la expresión VueloC.allInstances() retorna un conjunto finito de tamaño dos que contiene a los objetos VueloC1 y VueloC2, mientras que VueloD.allInstances() retorna un conjunto finito de tamaño dos que contiene a los objetos VueloD1 y VueloD2. En la figura 20 los objetos de las clases VueloC y VueloD no tienen especificados los valores de las propiedades numero y nroAvion porque estos no son relevantes. Lo mismo ocurre con las propiedades nombrePasajero y docPasajero para los objetos de la clase Asiento.

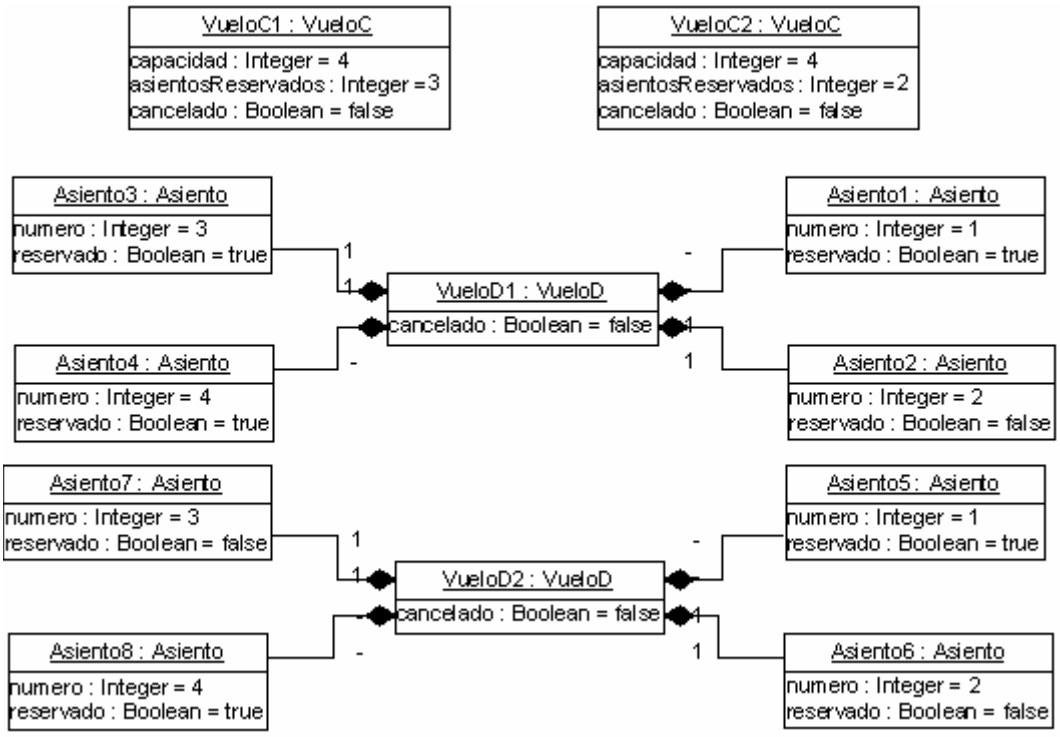


Figura 20 - Micromundo generado automáticamente del modelo UML de la figura 15 enriquecido con las restricciones de la figura 19

En este contexto, por ejemplo, la condición de aplicabilidad para la operación reservar() es la siguiente:

```
Set{<VueloC1>, <VueloC2>} -> forAll (vueloC |
Set{<VueloD1>, <VueloD2>} -> forAll (vueloD |
vueloC.mapping(vueloD) implies
(vueloC.capacidad - vueloC.asientosReservados > 0 and not vueloC.cancelado
implies vueloD.asientos -> exists (p | not p.reservado)
and not vueloD.cancelado)))
```

Esta expresión es evaluada fácilmente por un evaluador tradicional de OCL, retorna una respuesta positiva, pero como se comentó anteriormente esta respuesta no es definitiva.

Cabe aclarar que la expresión:

```
vueloD.asientos -> exists (p | not p.reservado)
```

es equivalente a:

```
vueloD.asientos -> select (p | not p.reservado) -> notEmpty()
```

Para explorar un caso donde las condiciones de refinamiento no se satisfacen; se modifica la siguiente precondición:

```
context VueloD :: reservar()
    pre: self.asientos -> select (p | not p.reservado) -> isEmpty()
         and not self.cancelado
```

La propiedad a ser evaluada es:

```
Set{<VueloC1>, <VueloC2>} -> forAll (vueloC |
Set{<VueloD1>, <VueloD2>} -> forAll (vueloD |
vueloC.mapping(vueloD) implies
(vueloC.capacidad – vueloC.asientosReservados > 0 and not vueloC.cancelado
implies vuelotD.asientos -> select (p | not p.reservado) -> isEmpty() and not
vueloD.cancelado)))
```

La condición de aplicabilidad no se cumple, porque en base al micromundo de la figura 20 se da la siguiente situación:

```
vueloC.mapping(vueloD) = true
vueloC.capacidad-vueloC.asientosReservados > 0 and not vueloC.cancelado = true
vueloD.asientos -> select (p | not p.reservado) -> isEmpty()
and not vueloD.cancelado = false
```

donde*:*

vueloC → VueloC1
vueloD → VueloD1

Para analizar apropiadamente las relaciones de refinamiento, los micromundos que se generan deben satisfacer todos los invariantes OCL que reducen el espacio de búsqueda, y "la propiedad de dualidad", la cual establece que para cada instancia de una clase concreta debe existir por lo menos una instancia de la clase abstracta donde tales instancias están relacionadas por el mapping de la abstracción. El proceso de generación automático de micromundo llevado a cabo por la herramienta asegura el cumplimiento de la propiedad de dualidad.

7. Generación de micromundos representativos

Escribir complejas transformaciones de modelos es propenso a errores, y son requeridas técnicas eficientes de testeo para cualquier desarrollo de programa complejo. El testeo de una trasformación de modelos es típicamente realizado chequeando el resultado de la transformación aplicada a un conjunto de modelos de entrada. Mientras es bastante fácil proveer algunos modelos de entrada, es difícil calificar la relevancia de esos modelos para testear.

Teniendo en mente la hipótesis de Jackson [**14**], que explica que las respuestas negativas tienden a suceder en mundos pequeños, se aplica para la evaluación de las condiciones de refinamiento la técnica de micromodelos de software definiendo un límite finito de instancias (micromundo). Luego se verifica si estas condiciones cumplen con la propiedad, teniendo en cuenta las dos consideraciones mencionadas anteriormente:

- Si la respuesta es positiva, hay una posibilidad de que la propiedad se cumple. En este caso, la respuesta no es definitiva porque puede haber un mundo más grande que no cumple con la propiedad, sin embargo la respuesta positiva es alentadora.
- Si la respuesta es negativa, entonces existe al menos un mundo que viola la propiedad. En ese caso, la respuesta es definitiva ya que la propiedad no se cumple en el modelo.

Los modelos son complejos grafos de objetos. Para generar un micromundo representativo tenemos que, primero determinar valores relevantes para las propiedades de los objetos (atributos y multiplicidades) y luego identificar estructuras pertinentes de objetos. Para la calificación de los valores de las propiedades, adaptamos la técnica de testeo clásica llamada “the category-partition method” [**12**]. La idea es descomponer un dominio de entrada en un número finito de subdominios y elegir los datos de test a partir de estos subdominios.

Las subsecciones siguientes forjan la base para la creación de micromundos, determinando cómo y qué particiones tiene sentido generar, y proponiendo estrategias para combinarlas a la hora de la generación de tales micromundos.

7.1. Conceptos

Un caso de prueba es una secuencia de entradas que determinan el comportamiento que debe ser testeado junto con el comportamiento esperado. Un caso de prueba es exitoso si el comportamiento observado coincide con el comportamiento esperado; en caso contrario, la prueba falla. El éxito puede ser determinado con la ayuda de un oráculo que compara el comportamiento observado con el esperado (correcto). Un conjunto de pruebas es un conjunto de uno o más casos de prueba.

Testear formas ejecutables de modelos es análogo a testear programas. En general, testear un artefacto de software ejecutable (programa o modelo de diseño ejecutable) involucra: (1) la creación de casos de prueba, (2) la ejecución del artefacto usando casos

de prueba, y (3) el análisis de los resultados de la prueba para determinar la correctitud del comportamiento testeado.

7.1.1. Criterio de testeo para diagramas UML

La suficiencia de las pruebas ejecutadas sobre modelos puede ser expresada en términos de la cobertura de los elementos del modelo. En [**13**] se exponen un conjunto de criterios de prueba basados en la cobertura de los elementos de un modelo, los cuales hemos adaptado en este trabajo para la construcción de micromundos. A continuación se detallan algunos de tales criterios.

- **Criterio de multiplicidad de asociaciones (AEM):** dado un conjunto de test T y un modelo del sistema SM, T debe causar que cada par de multiplicidad representativo en el SM sea creado.

- **Criterio de generalización (GN):** dado un conjunto de test T y un modelo del sistema SM, T debe causar que cada especialización definida en una relación de generalización sea creada.

- **Criterio de atributos de clases (CA):** dado un conjunto de test T, un modelo del sistema SM, y una clase C, T debe causar que un conjunto de combinaciones de valores de atributos representativos (en cada instancia de la clase C) sea creado.

Dos de los criterios (AEM y CA) son expresados en término de valores representativos. Para establecer el conjunto de valores representativos, es usada la forma propuesta por Ostrand y Balcer en [**12**] adaptada a UML. Usando este método, el dominio de valores es particionado en clases de equivalencia, y es seleccionado un valor de cada clase para el conjunto de valores representativos. Así mismo en [**13**] se expone cómo las restricciones (expresadas en OCL o alguna otra forma) pueden ser usadas para determinar particiones para un determinado modelo.

Las particiones pueden ser determinadas de las siguientes formas:

1) Particionamiento basado en conocimiento: usando el conocimiento del dominio del problema, el testeador puede determinar las particiones. Tal conocimiento puede ser obtenido examinando las restricciones OCL asociadas con el modelo. El dominio del valor de un constructor UML puede ser caracterizado por una o más restricciones. Una aproximación sistemática para determinar particiones para el dominio del valor puede proceder como sigue:

(a) Usando una de las restricciones, crear una partición inicial (por ejemplo, una partición en la cual la restricción es verdadera para cada elemento en la partición, y una partición en la cual la restricción es falsa para cada elemento de la partición).

(b) Refinando la partición inicial, particionando cada partición inicial usando una restricción distinta. Repetir este paso hasta que no haya más restricciones a ser aplicadas.

2) Particionamiento por defecto: el testeador puede utilizar particiones por defecto que consisten del valor mínimo, del valor no limitado, y del valor máximo. Por ejemplo,

dada una multiplicidad p..n, la partición del valor mínimo es {p}, la partición del valor no limitado es {p+1,…,n-1}, y la partición del valor máximo es {n}. Si la multiplicidad es definida como *, el testeador debe seleccionar un límite alto basado en los requerimientos del sistema.

7.1.1.1. Criterio de multiplicidad de asociaciones (AEM)

Una multiplicidad de asociación especifica cuántas instancias de una clase en el final opuesto de la asociación pueden ser asociadas con una única instancia de la clase del final de la asociación.

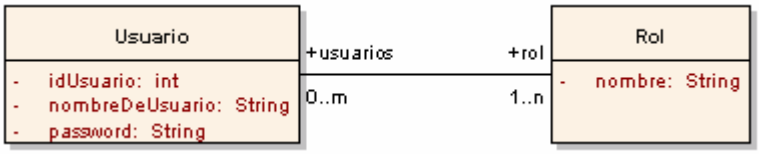


Figura 21 - Ejemplo de multiplicidad de asociaciones

El criterio AEM determina el conjunto de tuplas de multiplicidad representativas que debe ser creado durante el test. Para establecer el conjunto de tuplas de multiplicidad representativas, puede ser usada la forma adaptada del testeo de partición de categorías [**12**]. Usando este método, el dominio de la multiplicidad es particionado en clases de equivalencias, y es seleccionado un valor de cada clase para el conjunto de multiplicidades representativas. En el ejemplo anterior:

1) Crear un conjunto de multiplicidad seleccionando arbitrariamente un valor de cada partición. Un posible conjunto de multiplicidad para la clase Rol es {1, u, n} y uno para Usuario es {0, v, m}, donde m y n son límites altos determinados por el testeador, u es un valor del conjunto {1, …, n-1} y v es un valor del conjunto {0, …, m-1}.
2) Crear un conjunto de configuraciones $\{(r,s)_1, (r,s)_2, \ldots\}$ del producto cartesiano de los conjuntos múltiples de las instancias de Usuario y Rol, donde r es el número de instancias de Usuario relacionadas con s instancias de Rol. Para la asociación, este conjunto de configuración es: {(0,1), (0,v), (0,m), (u,0), (u,v), (u,m), (n,0), (n,v), (n,m)}.

El criterio AEM asegura que los testadores del diseño ejecutan pruebas que ejercitan configuraciones que contienen ocurrencias limitadas y no limitadas de links entre objetos (una instancia de una asociación es un link).

7.1.1.2. Criterio de generalización (GN)

El criterio de generalización define el conjunto representativo de tipos de especialización que deben ser creados durante el testeo del modelo del sistema. Un test que causa que sea creado uno de cada tipo de especialización es adecuado con respecto al criterio GN.

Los tests que satisfacen el criterio GN se espera que revelen fallas que pueden presentarse por la violación del principio de sustitución, el cual establece que una instancia de una subclase puede ser usada en cualquier lugar en donde se espera una instancia de la superclase. Los tests que satisfacen el criterio GN pueden destapar

violaciones de sustitución testeando comportamientos en los cuales son esperados los objetos de la superclase, utilizando especializaciones de la superclase en lugar de objetos de la superclase.

7.1.1.3. Criterio de atributos de clase (CA)

Los valores de los atributos pueden restringir el comportamiento de un objeto. De esta forma, el espacio de valores de atributos provee otra oportunidad de desarrollar el criterio de test. El espacio de valores de un atributo puede estar restringido por restricciones OCL.

La parte crítica del criterio es definir las combinaciones de valores de atributos representativos. Esto es hecho en tres pasos:
1) Usar el particionamiento de categorías para crear conjuntos de valores representativos para cada atributo.
2) Tomar el producto cartesiano de cada conjunto de valores, para crear un conjunto agregado de tuplas de valores de atributos representativos para cada clase.
3) Identificar conjuntos agregados válidos e inválidos.

7.1.2. Conclusiones

En [**13**] el foco está en el testeo del comportamiento de modelo, mientras que para transformaciones de modelos, estamos interesados solo en su estructura. Por este motivo, GN no es interesante en este contexto debido a que, aunque los comportamientos puedan ser sobrescritos en subtipos, las estructuras no pueden serlo. Por lo tanto, proponemos adaptar los dos criterios siguientes para alcanzar la cobertura de un modelo:

- ✓ **AEM**: para cada asociación, debe ser cubierta cada multiplicidad representativa. Por ejemplo, si una asociación tiene multiplicidad [0..N], puede ser instanciada con la multiplicidad 0, N, y algún valor entre 0 y N.
- ✓ **CA:** para cada atributo, debe ser cubierto cada valor representativo. Si el tipo del atributo es simple (integer, string, bolean), tiene que ser computado un conjunto de valores representativos. Si el tipo del atributo es complejo, debe ser procesado como una asociación, acorde al criterio AEM. Los valores representativos de atributos simples de una clase necesitan luego ser combinados usando el producto cartesiano.

7.2. Generando las particiones

En el contexto particular de testing del software, las particiones se han utilizado para definir el testeo de particiones de categoría (category partition [**12**]). Esta técnica consiste en definir clases de equivalencia en el dominio de entrada del software y después testear el programa con un valor de cada clase. La idea básica para generar las particiones consiste en dividir el dominio de entrada en subdominios o rangos y luego seleccionar datos de prueba desde cada uno de estos rangos. Los rangos para un

dominio de entrada definen una partición de los dominios de entrada y de esta forma no deben solaparse unos con otros.

Una definición precisa de partición es la siguiente:

Partición: Una partición de un conjunto de elementos es una colección de n rangos A1,…, An donde A1,.., An no se solapan y la unión de todos los subconjuntos forman el conjunto inicial. Estos subconjuntos son llamados rangos.

Para aplicar esta idea para la selección de modelos de test, proponemos definir particiones para cada propiedad en el modelo de entrada. Estas particiones proveen una forma práctica para seleccionar lo que llamamos "valores representativos". Para una propiedad p y para cada rango R en la partición asociada con P, el modelo de test debe contener al menos un objeto “o” tal que el valor “o.p” está contenido en R.

Con respecto a las multiplicidades de las propiedades, en [**11**] se establece que si una propiedad tiene multiplicidad 0..*, una partición como {{0}, {1}, {2..inf}} es definida para asegurar que el modelo de test contiene instancias de esa propiedad con cero, una, o más de un objeto. Tengamos en cuenta el siguiente modelo de ejemplo:

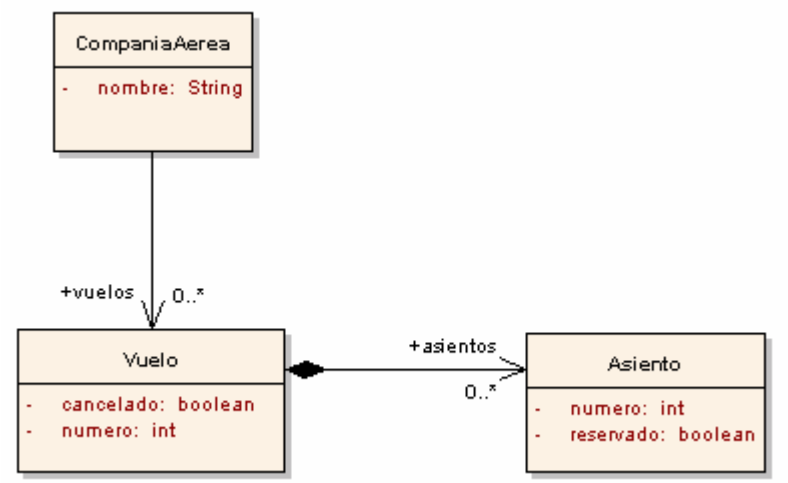


Figura 22 - Ejemplo de modelo para generar particiones

La siguiente figura muestra las particiones que se generan en base a lo propuesto en [**11**] para el modelo del ejemplo anterior.

CompaniaAerea::nombre	{“”}, {“NombreCompania”}
CompaniaAerea::#vuelos	{0}, {1}, {2..MaxInt}
Vuelo::cancelado	{true},{false}
Vuelo::numero	{0}, {1}, {2..MaxInt}
Vuelo::#asientos	{0}, {1}, {2..MaxInt}
Asiento::numero	{0}, {1}, {2..MaxInt}
Asiento::reservado	{true}, {false}

Figura 23 - Particiones generadas para un modelo de ejemplo

La eficiencia de la estrategia de testeo por partición de categorías descansa en la calidad de las particiones que son utilizadas. La idea es aislar los límites y valores singulares en rangos para asegurar que esos valores especiales son usados para el testeo.

7.3. Combinando las particiones

La cobertura independiente de valores y multiplicidades de cada propiedad del modelo de entrada no es suficiente para asegurar la relevancia de los modelos de test; combinar particiones de varias propiedades se transforma en algo necesario. Una aproximación simplista podría ser combinar sistemáticamente las particiones de todas las propiedades del modelo; sin embargo, en la práctica esta aproximación no es conveniente por dos razones:

1) Combinar rangos para todas las propiedades puede generar un número de combinaciones a cubrir poco realista. En el ejemplo de la sección previa, acorde a las particiones definidas en la figura 23, el número de combinaciones será 648 (2*3*2*3*3*3*2). Incluso, algunas combinaciones son más relevantes que otras.
2) Combinar rangos para todas las propiedades no necesariamente es suficiente para asegurar la relevancia de los modelos de test.

Como las estrategias simples no son suficientes para proveer un soporte satisfactorio para la selección de modelos de test relevantes, se propone en [**11**], las nociones de fragmentos de modelo y objeto para definir combinaciones específicas de rangos para propiedades que deben ser cubiertas por los modelos de test. El metamodelo de la siguiente figura captura la noción de particiones asociadas a las propiedades así como también fragmentos de modelos y objetos.

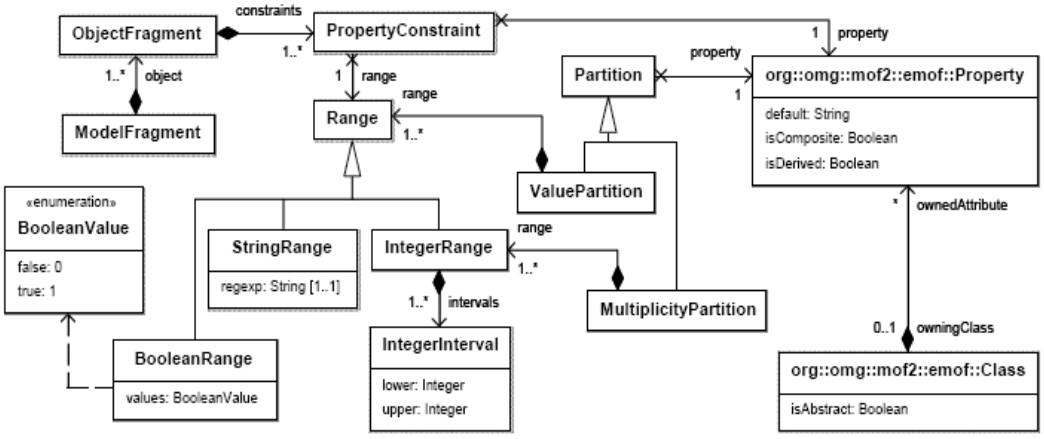


Figura 24 - Metamodelo de particiones y fragmentos

Distinguimos dos tipos de particiones modeladas por las clases ValuePartition y MultiplicityPartition que respectivamente corresponden a las particiones para el valor y la multiplicidad de una propiedad. Para una MultiplicityPartition, cada rango es un rango entero (clase IntegerRange). Para una ValuePartition, el tipo de rangos depende del tipo de la propiedad. En la figura anterior se modelan los tres tipos primitivos que

son definidos en EMOF para el valor de una propiedad; por lo tanto hay modelados tres tipos diferentes de rangos (StringRange, BooleanRange, IntegerRange).

En este trabajo hemos basado nuestro modelo de clases en el metamodelo de la figura 24 que ya analizaremos en la siguiente sección.

7.4. Fragmentos de modelo y fragmentos de objeto

Una noción importante para generar los modelos para testear es tener criterios para calificar modelos. En este trabajo, consideramos que un modelo es relevante para ser testeado si contiene estructuras interesantes de objetos.

La calidad de los modelos de entrada entonces depende directamente de la definición de estructuras de objetos interesantes. Llamamos tales estructuras de objeto a los fragmentos de modelo. Se utilizan los conceptos y las notaciones siguientes:

Un fragmento de modelo MF es un conjunto de fragmentos de objeto:
MF = {object-fragments}. Especifica que por lo menos uno de los modelos de entrada debe incluir todos estos fragmentos de objeto.

Un fragmento de objeto OF especifica una instancia parcial de una clase C. Relaciona clases de equivalencia con propiedades. Es un conjunto de restricciones de propiedades, que asocia una propiedad Pj de una clase C a una clase de equivalencia ECk: OF = {(C::Pj, ECk)}j,k.

Para representar combinaciones de rangos de particiones, el metamodelo de la figura 24 [**11**] propone las nociones de fragmentos de modelo (ModelFragment), fragmentos de objetos (ObjectFragment) y restricciones de propiedades (PropertyConstraint). Un fragmento de modelo está compuesto por un conjunto de fragmentos de objeto. Un fragmento de objeto está compuesto por un conjunto de restricciones de propiedades, que especifican los rangos desde los cuales deben ser tomados los valores de las propiedades de un objeto. Es importante notar que un fragmento de objeto no define necesariamente restricciones para todas las propiedades de una clase, puede restringir parcialmente las propiedades de un objeto en un modelo de test.

La siguiente figura presenta un fragmento de modelo simple que combina rangos para propiedades del modelo de la figura 22.

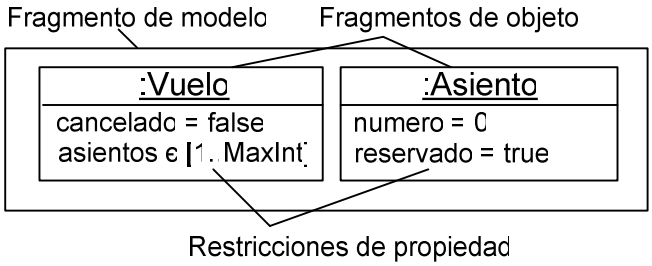


Figura 25 - Ejemplo de fragmento de modelo

El fragmento de modelo está compuesto de dos fragmentos de objeto que especifican que el modelo del test debe contener:
* Un vuelo que no está cancelado con uno o varios asientos.
* Un asiento identificado con el valor 0 y que está reservado.

Usando este formalismo, las combinaciones particulares que deben ser cubiertas por los modelos de test pueden ser fácilmente representadas; aunque la selección de estas combinaciones todavía no se ha resuelto.

7.5. Criterios de test

Como se discutió previamente, la construcción de fragmentos de modelo relevantes no puede ser resuelta con una simple estrategia como crear todas las combinaciones de rangos para todas las propiedades del modelo. Sin embargo, el modelo provee en sí mismo información acerca de las relaciones entre las propiedades que contiene. De hecho, las propiedades están encapsuladas dentro de clases y las clases están relacionadas por asociaciones y herencia.

En esta sección usamos esta información estructural para proponer un conjunto de criterios de test para la construcción de fragmentos de modelo que deben ser cubiertos por el conjunto de modelos de test.

Criterio de Test: un criterio de test especifica un conjunto de fragmentos de modelo para un modelo de entrada. Estos fragmentos de modelo son construidos para garantizar la cobertura de clase y rango como se define en las siguientes reglas:

Regla 1 - Cobertura de clase: cada clase concreta debe ser instanciada en al menos un fragmento de modelo.

Regla 2 - Cobertura de rango: cada rango de cada partición para todas las propiedades del modelo debe ser usado en al menos un fragmento de modelo.

Satisfacción del criterio de test: un conjunto de modelos de test satisface un criterio de test si, para cada fragmento de modelo MF, existe un modelo M tal que todos los fragmentos de objeto definidos en MF corresponden a un objeto en M. Un objeto O corresponde a un fragmento de objeto OF si, para cada restricción de propiedad en OF, el valor de la propiedad en O está incluido en el rango especificado por OF.

7.5.1. Criterios de cobertura simple

Esta sección define dos criterios que aseguran la cobertura de rango combinando restricciones de propiedades de dos maneras distintas. El primer criterio, allRanges, no agrega ninguna restricción a las dos reglas definidas en la sección previa. El segundo criterio, allPartitions, es un poco más fuerte debido a que requiere que los valores de todos los rangos de una propiedad sean utilizados simultáneamente en un único modelo de test. La siguiente figura formaliza estos dos criterios en OCL:

AllRanges criterion

```
Partition.allInst.forAll{ P |
  P.range.forAll{ R |
    ModelFragment.allInst.exists{ MF |
      MF.object.size == 1
      MF.object.one.constraint.size == 1
      MF.object.one.constraint.one.range == R
}}}
```

AllPartitions criterion

```
Partition.allInst.forAll{ P |
  ModelFragment.exists{ MF |
    P.range.forAll{ R |
      MF.object.exists{ OF |
        OF.constraint.size == 1
        OF.constraint.one.range == R
}}}}
```

Figura 26 - Cobertura de rangos y particiones en OCL

En la práctica, el criterio de cobertura simple puede ser usado para crear un conjunto de fragmentos de modelo inicial pero, en la mayoría de los casos, este conjunto de fragmentos de modelo debe ser completado por el tester o usando un criterio más fuerte.

7.5.2. Criterios de cobertura clase por clase

En el modelo, las propiedades están encapsuladas en clases. Basados en esta estructura y en la forma en la que los modelos son diseñados, es natural que las propiedades que pertenecen a una clase tengan una relación semántica más fuerte con cada una de las otras, que con propiedades de otras clases. Para tomar ventaja de esto, se proponen en [**11**] cuatro criterios que combinan rangos, clase por clase. Estos criterios difieren por un lado, en el número de combinaciones de rango que requieren y por el otro, en la forma en que las combinaciones son agrupadas en fragmentos de modelo.

7.5.2.1. Combinando rangos

A continuación se da la definición de dos criterios propuestos en [**11**] para combinar rangos, más adelante veremos que la cantidad de fragmentos de objeto generados utilizando uno y otro difiere considerablemente.

OneRangeCombination: cada rango para cada propiedad de una clase necesita ser usada en al menos un fragmento de objeto.

AllRangesCombination: todas las combinaciones posibles de rangos para las propiedades de una clase deben aparecer en un fragmento de objeto.

La siguiente figura formaliza estos dos criterios en OCL:

OneRangeCombination

```
Range.allInst.forAll{ R |
  ObjectFragment.allInst.exists{ OF |
    OF.constraint.exists{ PC |
      PC.range == R
}}}
```

AllRangesCombination

```
Class.allInst.forAll{ C |
  getCombinations(Partition.allInst.select{ P |
    C.ownedAttribute.contains(P.property)}
  ).forAll{ RSet |
    ObjectFragment.allInst.exists{ OF |
      RSet.forAll{ R |
        OF.constraint.exists{ PC |
          PC.range == R and
          PC.property == R.partition.property
}}}}}
```

Figura 27 - Dos estrategias para combinaciones de rangos

La figura anterior propone dos estrategias para combinar los rangos de las propiedades de una clase. La primera es absolutamente débil, pues solo asegura que cada rango de cada propiedad sea cubierto al menos una vez. La segunda es más fuerte, dado que requiere un fragmento de objeto para cada combinación de rangos posible para todas las propiedades de una clase. La operación *getCombinations* usada para la definición de esta estrategia simplemente computa el producto cartesiano para rangos del conjunto de particiones. Veamos el siguiente ejemplo:

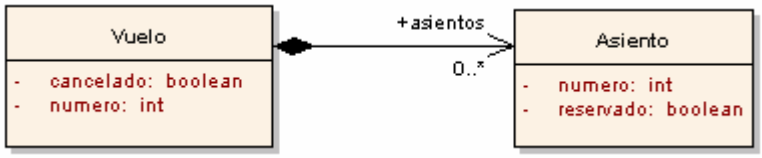


Figura 28 - Ejemplo para combinaciones de rangos

Teniendo en mente la figura 28 y de acuerdo a lo presentado en las secciones anteriores, las posibles particiones para las clases Vuelo y Asiento podrían ser:

Vuelo::cancelado	{true}, {false}
Vuelo::#asientos	{0}, {1},{2..MaxInt}
Asiento::numero	{0}, {1}, {2..MaxInt}
Asiento::reservado	{true}, {false}

Figura 29 - Particiones para el ejemplo de la figura 28

Si estuviésemos empleando la estrategia de combinación de rangos OneRangeCombination un posible conjunto de fragmentos de objeto para las particiones anteriores se ilustra en la figura 30:

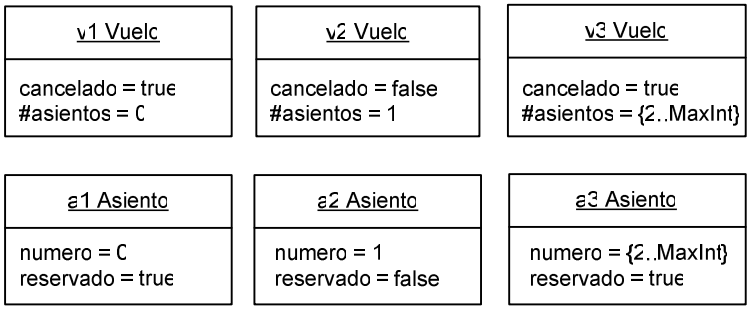


Figura 30 - Fragmentos de objeto utilizando la estrategia OneRangeCombination

La estrategia OneRangeCombination propone que cada rango perteneciente a la partición para una propiedad de una clase debe aparecer al menos en un fragmento de objeto. Esta estrategia es sumamente débil, pero el número de fragmentos de objeto a generar es considerablemente más chico que el número de fragmentos de objeto que pueden llegar a ser creados si se utiliza la estrategia AllRangesCombination. Veamos en la siguiente figura los fragmentos de objeto que se generarían si implementáramos la estrategia AllRangesCombination para el ejemplo anterior:

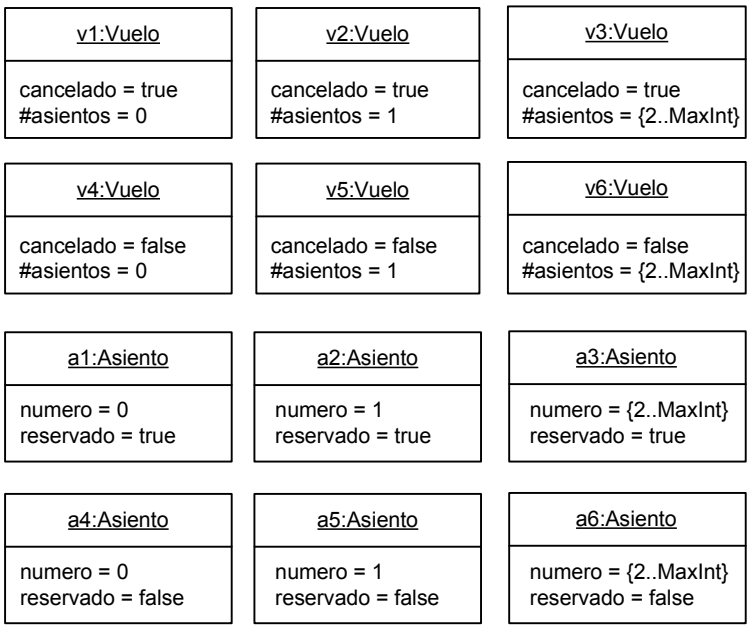


Figura 31 - Fragmentos de objeto utilizando la estrategia AllRangesCombination

En este ejemplo particular, el número de fragmentos de objeto generados es el doble que si aplicamos la estrategia OneRangeCombination. A medida que las clases tienen un mayor número de propiedades, la cantidad de fragmentos de objeto crece significativamente si utilizamos la estrategia AllRangesCombination.

La siguiente tabla compara el número de fragmentos de objeto generado en cada caso:

Criterio	Nº de Fragmentos generados
OneRangeCombination	6
AllRangesCombination	12
Todas las combinaciones [7]	36

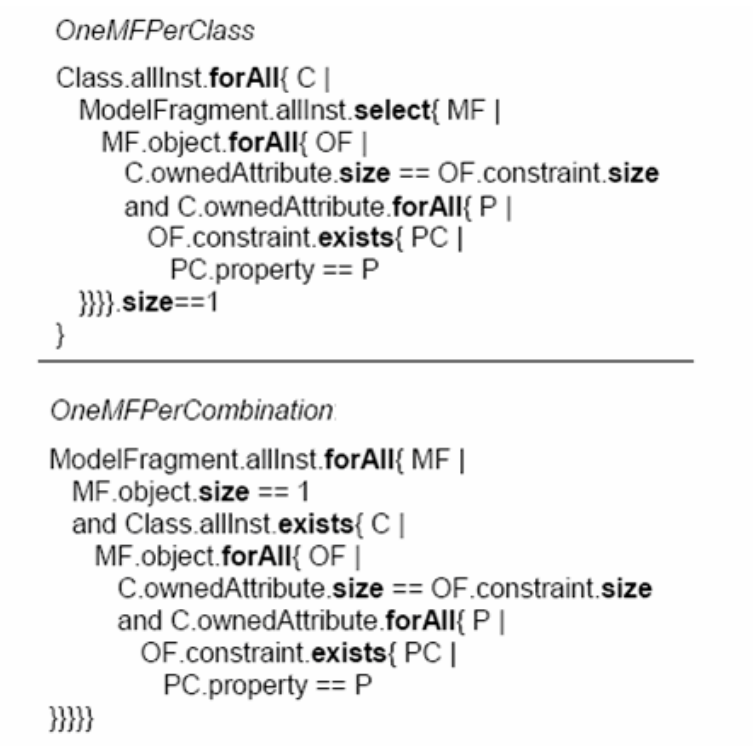
Es interesante notar que todos los criterios propuestos reducen perceptiblemente el número de fragmentos comparado a la estrategia de "todas la combinaciones". Sin embargo, para todos los criterios que requieran el producto cartesiano de rangos, el número de fragmentos sigue siendo absolutamente alto. Pensemos también que estos datos son arrojados para un modelo muy simple de dos clases con pocos atributos.

7.5.2.2. Agrupando combinaciones de rangos

Los criterios que se presentan en esta sección describen dos formas para organizar los fragmentos de objeto generados como resultado de aplicar alguna de los criterios explicados en la sección precedente. La figura 32 formaliza estos dos criterios en OCL.

OneMFPerClass: un solo fragmento de modelo contiene todas las combinaciones de rangos para la clase.

OneMFPerCombination: cada fragmento de modelo contiene una sola combinación de rangos para la clase.



OneMFPerClass

```
Class.allInst.forAll{ C |
  ModelFragment.allInst.select{ MF |
    MF.object.forAll{ OF |
      C.ownedAttribute.size == OF.constraint.size
      and C.ownedAttribute.forAll{ P |
        OF.constraint.exists{ PC |
          PC.property == P
  }}}}.size==1
}
```

OneMFPerCombination

```
ModelFragment.allInst.forAll{ MF |
  MF.object.size == 1
  and Class.allInst.exists{ C |
    MF.object.forAll{ OF |
      C.ownedAttribute.size == OF.constraint.size
      and C.ownedAttribute.forAll{ P |
        OF.constraint.exists{ PC |
          PC.property == P
}}}}}
```

Figura 32 - Dos estrategias para generar fragmentos de modelo

[7] Esta técnica consiste en combinar todos los posibles valores de cada partición con todos los posibles valores de las demás particiones. En el ejemplo sería 2x3x3x2 = 36. El número de fragmentos de objeto generados es alto.

La figura anterior presenta las dos estrategias propuestas en [**11**] para agrupar combinaciones de rangos clase por clase. En ambos casos la idea es crear fragmentos de objeto que contienen restricciones relacionadas con cada propiedad de una clase. Las dos estrategias difieren en la forma en que los fragmentos de objeto son organizados en fragmentos de modelo. La primera estrategia (OneMFPerClass) agrupa todos los fragmentos de objeto relacionados a una clase en un solo fragmento de modelo mientras que la segunda crea un fragmento de modelo por cada fragmento de objeto.

Basados en las estrategias para combinación de rangos y para construir fragmentos de modelo, en [**11**] se proponen los cuatro criterios mostrados en la figura siguiente:

Test criteria	Definition
Comb∑	OneRangeCombination and OneMFPerCombination
CombX	AllRangesCombination and OneMFPerCombination
Class∑	OneRangeCombination and OneMFPerClass
ClassX	AllRangesCombination and OneMFPerClass

Figura 33 - Criterios de test basados en las combinaciones clase por clase

En nuestro trabajo hemos implementado las dos estrategias de combinación de rangos, pudiendo el usuario seleccionar cuál estrategia utilizar. La organización de los fragmentos de objeto dentro de los fragmentos de modelo no está basada en ninguno de los dos criterios mencionados, porque consideramos que, para los fines de nuestro trabajo, lo más relevante es cómo se combinan los fragmentos de objeto en lugar de cómo están organizados.

En la herramienta ePlatero, la estructura utilizada para generar el micromundo equivale a un solo fragmento de modelo que contiene fragmentos de objetos que involucran una o varias clases del modelo.

7.6. Conclusiones

Como conclusión de esta sección podemos decir que para generar un micromundo representativo, los pasos a seguir son los siguientes:

1) Crear particiones representativas para cada una de las propiedades (atributos y multiplicidades) de los objetos del modelo. Para esto se utiliza la técnica de particionamiento de categorías propuesta en [**12**].
2) Combinar los rangos de las particiones creadas utilizando la estrategia OneRangeCombination o la estrategia AllRangesCombination, aprovechando el hecho de que las propiedades que pertenecen a una clase tienen, entre ellas, una relación semántica más fuerte que con propiedades de otras clases.
3) Representar las combinaciones de rangos realizadas en fragmentos de objetos pertenecientes a un fragmento de modelo.
4) A partir del fragmento de modelo generar el micromundo representativo. Esta generación es directa, ya que en el fragmento de modelo está contenida toda la información necesaria para la creación de instancias relevantes.

8. ePlatero. Módulo Generador de Micromundos

ePlatero [**8**] es una herramienta CASE educativa que soporta el proceso de desarrollo de software dirigido por modelos utilizando notación gráfica con fundamento formal. Dicha herramienta es un plugin para la plataforma de eclipse, y puede interoperar con herramientas que soporten MDA.

La arquitectura de plugins de eclipse [**21**] permite, entre otras cosas, enriquecerlo con herramientas que pueden ser útiles para el desarrollo de software. Para ello, la plataforma está estructurada como un conjunto de subsistemas, los cuales son implementados en uno o más plugins que corren sobre una runtime engine. Dichos subsistemas definen puntos de extensión para facilitar la extensión de la plataforma.

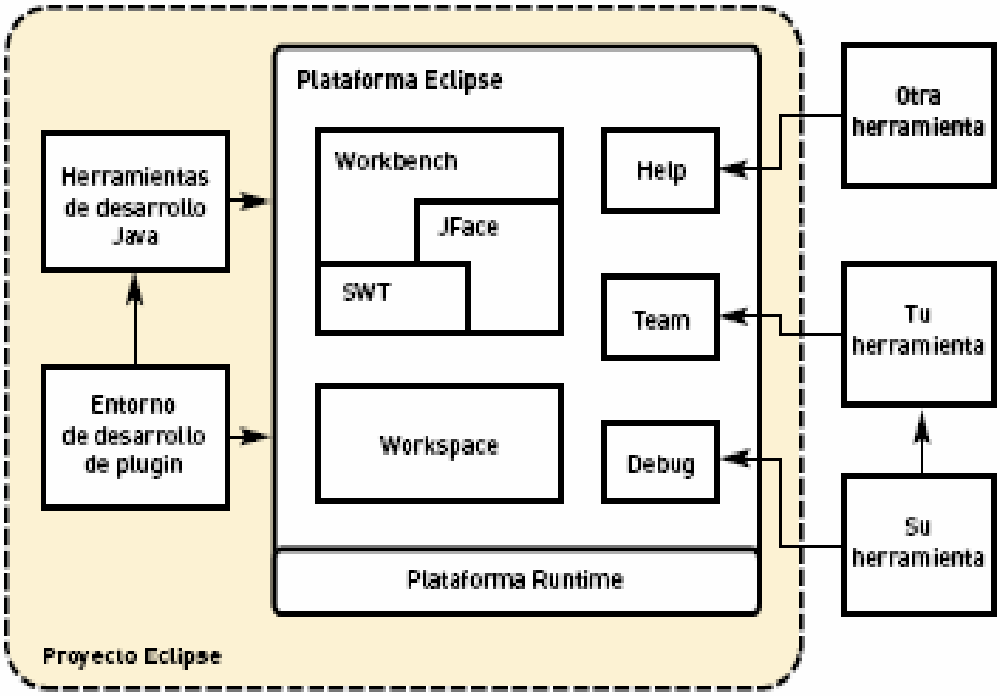


Figura 34 - Arquitectura de eclipse, para el desarrollo de plugins

Este capítulo está organizado de la siguiente manera: en la sección 8.1 se describe la arquitectura de ePlatero mostrando cuáles son sus componentes, en la sección 8.2 se describe el módulo generador de micromundos que es el tema central de nuestra tesis, en la sección 8.3 se encuentra una breve descripción de los componentes del generador de micromundos, en 8.4 se detalla el modelo implementado con una descripción de las clases involucradas, en la sección 8.5 se describe el papel que juega la función de abstracción y los problemas que llevaron a tener que contar con la misma y su relación con la propiedad de dualidad, la sección 8.6 se dedica a describir la estructura del componente junto con los procesos para la construcción de particiones y la generación de micromundos, y por último en la sección 8.7 se describen dos patrones de diseño usados y la importancia de su utilización.

8.1. Arquitectura de ePlatero

La arquitectura de ePlatero [**8**] respeta el estándar de Eclipse para el desarrollo de plugins. Consta de 8 módulos o componentes:

- Analizador léxico / Parseador
- Repositorio del proyecto
- Coordinador
- Evaluador de fórmulas OCL
- Editor de fórmulas OCL
- Evaluador de refinamiento
- Generador de Micromundos
- Editor UML.

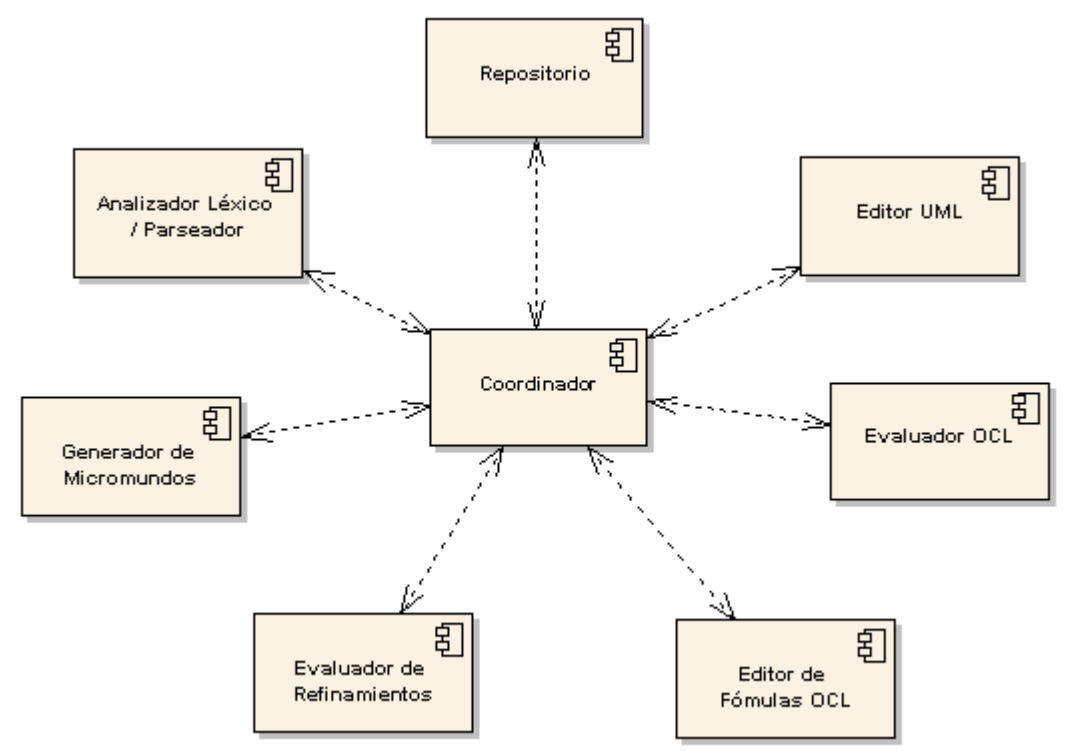


Figura 35 - Arquitectura de ePlatero

En este trabajo pondremos especial atención en el componente Generador de Micromundos, que es el tema central de nuestra tesis, sin detenernos en los demás componentes que han sido objeto de estudio en otros trabajos [**9**] y [**10**] y que en esta versión de ePlatero son reutilizados.

8.2. Módulo Generador de Micromundos

El generador de micromundos es el módulo responsable de instanciar los objetos que representan un estado del sistema, esto es fundamental para la evaluación de los refinamientos como se comentó en el capítulo 7. Para generar el micromundo no es

necesario definir ningún lenguaje, con OCL es suficiente para especificar las restricciones a nivel modelo, restricciones de los dominios y la propiedad de dualidad.

Como ya hemos mencionado, no es una buena opción generar los micromundos utilizando una técnica basada en combinar todos los posibles valores para los atributos y multiplicidades, puesto que el tamaño del micromundo sería muy grande y el proceso de generación automática tendría un gran coste de tiempo, resultando poco útil. Tampoco es suficiente generar micromundos con valores aleatorios para los atributos y multiplicidades, como lo venía haciendo la herramienta, porque en este caso los micromundos creados no son representativos. Por estas razones, el modulo fue implementado para generar micromundos utilizando las estrategias de particionamiento y las técnicas para combinar tales particiones, estudiadas en el capítulo anterior (OneRangeCombination y AllRangesCombination), quedando abierto para soportar nuevas estrategias que puedan surgir de estudios futuros.

El módulo generador de micromundos es capaz de generar micromundos ya sea para un modelo de entrada completo (teniendo en cuenta todas las clases del mismo), o solamente para las clases involucradas en un refinamiento. Además, permite crear micromundos de un tamaño determinado (ingresado por el usuario) y brinda algunas funcionalidades para el tratamiento de las particiones, proporcionando de esta manera mayor flexibilidad a la hora de construir micromundos:

- Crear nuevos rangos para particiones
- Editar rangos de particiones existentes
- Eliminar rangos de particiones existentes

En caso de querer generar un micromundo para un refinamiento, este módulo permite adjuntar un archivo con la función de abstracción (escrita en java) necesaria para crear instancias de la definición abstracta a partir de instancias de la definición concreta, y nos provee de una clase (Helper) que ayuda en la confección de la función. La función de abstracción colabora con el módulo de generación de micromundos para darle valores a las propiedades de la clase abstracta a la hora de generar las instancias, de manera que se cumpla la propiedad de dualidad. La importancia de contar con una función de abstracción, así como también el significado de la propiedad de dualidad se describen más adelante en esta sección.

8.3. Componentes

Desde un nivel más bajo de abstracción podemos descomponer el módulo Generador de Micromundos presentado en la sección anterior en dos grandes componentes que forman parte del mismo.

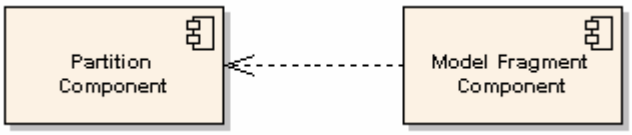


Figura 36 - Componentes del Generador de Micromundos

El componente *Partition Component* es el encargado de generar las particiones para un modelo en particular o para las clases involucradas en una condición de refinamiento. Basado en una estrategia, este componente crea las particiones para las propiedades de las clases del modelo, que serán usadas por el componente ModelFragment Component para la generación de los fragmentos de objeto.

El segundo gran componente que forma parte del Generador de Micromundos es el *ModelFragment Component*. Dentro de sus responsabilidades se encuentra generar el fragmento de modelo para un modelo de entrada y generar instancias para tal fragmento de modelo, es decir el micromundo.

Las flechas que aparecen en la figura no representan las interacciones entre los paquetes, sino las dependencias entre éstos. Lo que nos está diciendo el diagrama de la Figura 36 es que si se llegaran a realizar cambios en el componente Partition va a ser necesario revisar la repercusión de estos cambios en el componente Model Fragment.

8.4. Metamodelo utilizado

En la herramienta ePlatero se ha utilizado el metamodelo de la figura 24, propuesto por Franck Fleurey, Benoit Baudry y Pierre-Alain Muller en [**11**]), con algunas modificaciones, permitiéndonos mantener una estructura útil al momento de generar los micromundos de instancias representativas.

Entre las modificaciones realizadas al metamodelo de la figura 24 se encuentran:

- Lo que antes era una Property de EMOF ahora es un PropertySignature del ePlatero, que esta subclasificado en AttributeSignature y AssociationEndSignature.
- Las particiones ahora están subclasificadas en AttributePartition y AssociationEndPartition dependiendo si estas se refieren a atributos o asociaciones de una clase.
- La clase ObjectFragment ahora conoce cuál es el ModelType que está representando. Esto es necesario a la hora de generar las instancias.

El metamodelo modificado se presenta en la figura 37.

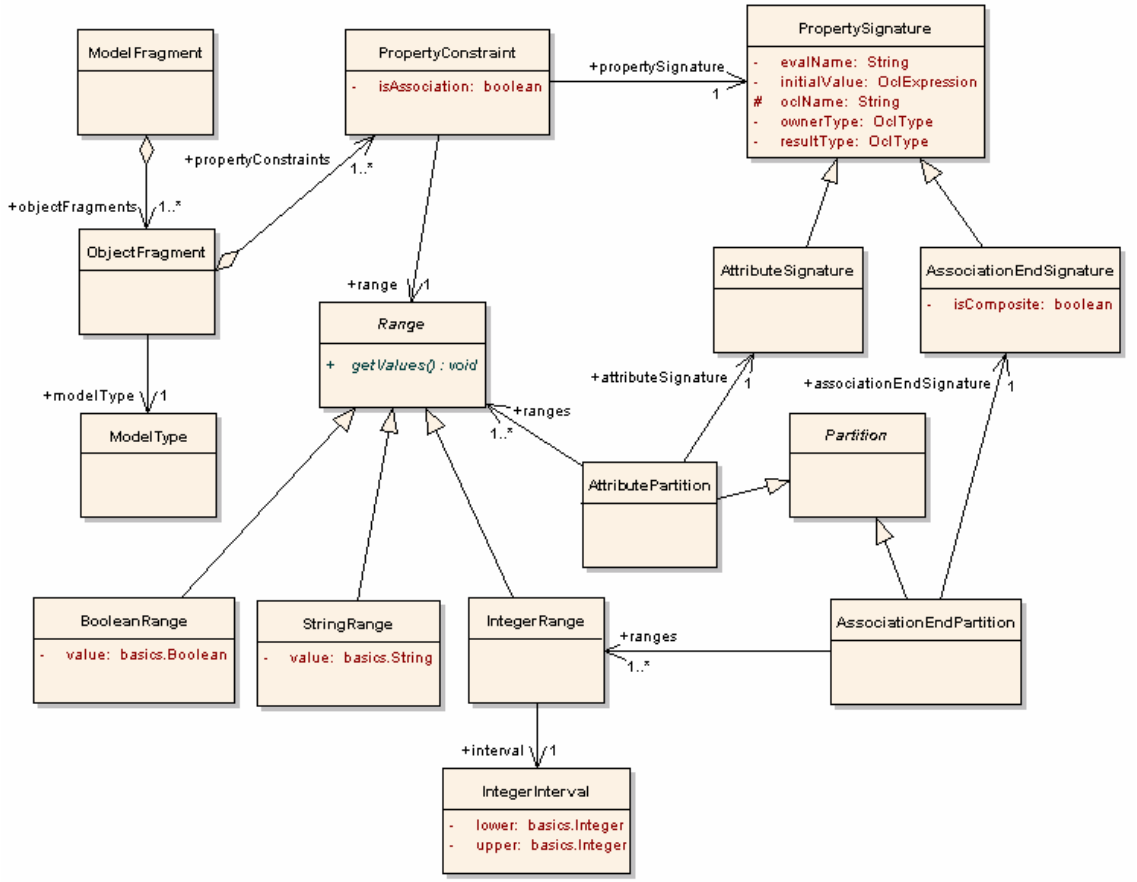


Figura 37 - Metamodelo para la generación de micromundos

A continuación se detalla una breve descripción de cada una de las clases del metamodelo de la figura anterior:

ModelFragment: representa un fragmento de modelo. Agrupa todos los fragmentos de objeto. En nuestra implementación se genera una sola instancia de la clase ModelFragment para todo el modelo (o para todas las clases involucradas en la condición de refinamiento).

ObjectFragment: representa un fragmento de objeto. Como se menciono anteriormente, un fragmento de objeto está compuesto por un conjunto de restricciones de propiedades, que especifican los rangos desde los cuales deben ser tomados los valores de las propiedades de un objeto.

ModelType: representa a una clase del modelo.

PropertySignature: representa una propiedad de una clase del modelo. Tal propiedad puede ser un atributo de la clase (AttributeSignature) o una asociación con otra clase del modelo (AssociationEndSignature).

PropertyConstraint: representa una restricción sobre una propiedad de una clase del modelo, indicando el rango desde el cual se debe tomar un valor para tal propiedad.

Partition: representa una partición del conjunto de posibles valores para una propiedad de una clase del modelo.

AttributePartition: representa una partición para un atributo de una clase del modelo. Tal partición puede estar asociada con rangos de cualquier tipo.

AssociationEndPartition: representa una partición para una asociación de una clase del modelo. Tal partición puede tener solamente rangos de enteros.

Range: clase abstracta que representa los rangos de las particiones.

BooleanRange: representa un rango de valores booleanos. Puede ser instanciada con uno de los dos valores booleanos: True o False.

StringRange: representa un rango con un sólo valor de tipo String.

IntegerRange: representa un rango de valores enteros identificados por un intervalo con un valor mínimo y un valor máximo (IntegerInterval).

8.5. Propiedad de dualidad y función de abstracción

Es sabido que para analizar correctamente relaciones de refinamientos, los micromundos deben satisfacer la "*propiedad de dualidad"*. Esta propiedad establece que para cada instancia de una clase refinada debe existir al menos una instancia de la clase abstracta, siendo relacionadas por el mapping de abstracción. Por lo tanto, el módulo de generación de micromundos solo genera particiones para las propiedades de las clases refinadas (utilizando la estrategia ya mencionada "*the category partition method"*), mientras que los valores de las propiedades de las clases abstractas son automáticamente calculados aplicando una función de abstracción.

De esta manera, alcanzamos dos objetivos; primero, la propiedad de dualidad se mantiene por construcción, y segundo, la cantidad de propiedades a ser analizadas decrece considerablemente.

8.5.1. Problemas y decisiones

Uno de los problemas que surgió cuando se desarrolló la implementación del módulo de micromundos de la herramienta ePlatero fue el hecho de generar las instancias para construir los micromundos.

En una primera etapa se pensó en construir instancias representativas de la clase abstracta y generar instancias aleatorias de la clase concreta, siempre y cuando cumplieran con el mapping de la abstracción. Es decir, se generaban instancias aleatorias de la clase concreta y se evaluaba el mapping de la abstracción: si la instancia de la clase concreta generada no cumplía con el mapping, entonces se desechaba tal instancia; en caso contrario la instancia se mantenía. Todo esto era ineficiente y, además, no era del todo correcto estar generando instancias representativas de la clase abstracta.

Por ello, se decidió cambiar el enfoque y generar instancias representativas de la clase concreta. Sin embargo, seguía siendo un problema generar las instancias de la clase abstracta para que se cumpliera la propiedad de dualidad. Como solución se pensó en tener una función de abstracción que ayudara a darle valores a las propiedades de la clase abstracta a la hora de generar las instancias, de manera que se cumpliera la propiedad de dualidad.

8.5.2. Función de abstracción

La función de abstracción deberá ser escrita en código java, y almacenada en un archivo de texto plano (.txt). Para no hacer tan tediosa la codificación, se contará con un objeto llamado "helper" que proporciona algunos métodos útiles. Entre otros se encuentran:

- getConcreteInstanceProperty(String propertyName) => retorna el valor que tiene la instancia de la definición concreta para la propiedad propertyName.
- getIntanceProperty(InstanceSpecification instance, String propertyName) => retorna el valor que tiene una instancia cualquiera (instance) para la propiedad propertyName.
- createSlotWithValue(String featureName, InstanceSpecification value) => crea un slot para la propiedad featureName en la instancia de la definición abstracta y le setea el value de tipo InstanceSpecification recibido como argumento.
- createSlotWithValue(String featureName, Integer value) => que crea un slot para la propiedad featureName en la instancia de la definición abstracta y le setea el value de tipo Integer recibido como argumento.
- createSlotWithValue(String featureName, Boolean value) => que crea un slot para la propiedad featureName en la instancia de la definición abstracta y le setea el value de tipo Boolean recibido como argumento.
- createSlotWithValue(String featureName, String value) => que crea un slot para la propiedad featureName en la instancia de la definición abstracta y le setea el value de tipo String recibido como argumento.

En el archivo de texto plano solamente se deberá incluir el código de la función de abstracción (sin la signatura), ya que ePlatero se encargará de crear el método correspondiente. Además se puede suponer la existencia de este objeto helper, lo que permite utilizarlo sin necesidad de crearlo previamente.

8.5.3. Ejemplo de función de abstracción

Supongamos que tenemos el siguiente refinamiento, presentado en la sección 6.3.2, figura 15:

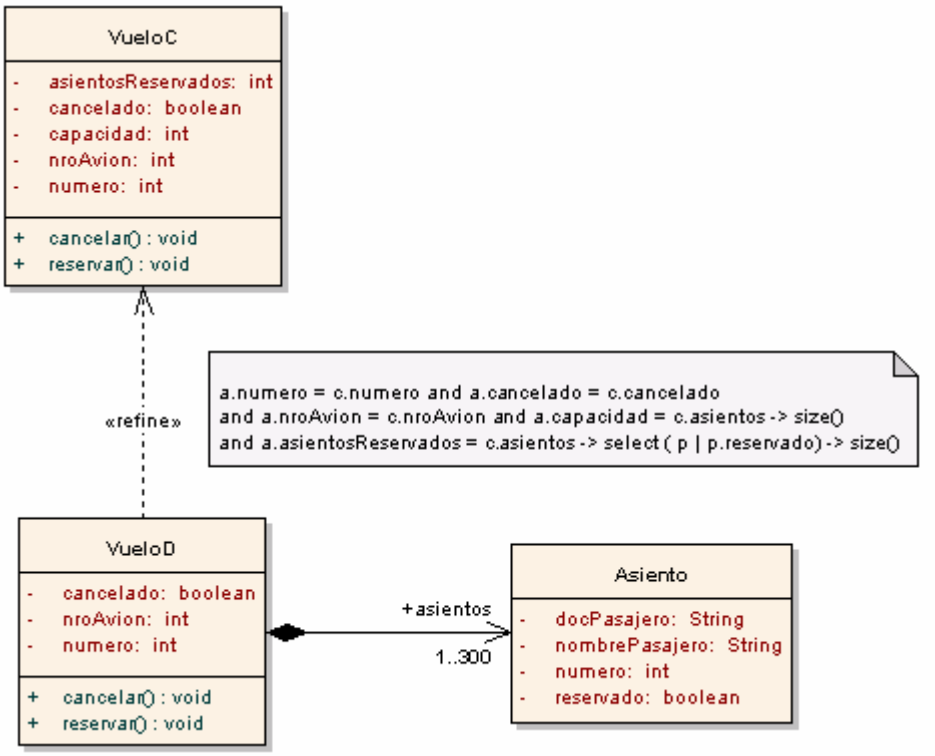


Figura 38 - Ejemplo para la función de abstracción

El código de la función de abstracción para el ejemplo anterior quedaría:

```
Object valorCancelado = helper.getConcreteInstanceProperty("cancelado");
Object valorNroAvion = helper.getConcreteInstanceProperty("nroAvion");
Object valorNumero = helper.getConcreteInstanceProperty("numero");
Object valorAsientos = helper.getConcreteInstanceProperty("asientos");

java.util.Iterator iteradorAsientos = ((java.util.Collection) valorAsientos).iterator();
int asientosReservados = 0;
org.eclipse.uml2.uml.InstanceSpecification asiento;
Object valor;
Object numero;
while (iteradorAsientos.hasNext()) {
        asiento = (org.eclipse.uml2.uml.InstanceSpecification) iteradorAsientos.next();
        valor = helper.getInstanceProperty(asiento, "reservado");
        if (((java.lang.Boolean)valor).booleanValue())
                asientosReservados++;
}

helper.createSlotWithValue("asientosReservados", asientosReservados);
helper.createSlotWithValue("cancelado",
                        ((java.lang.Boolean)valorCancelado).booleanValue());
helper.createSlotWithValue("capacidad", ((java.util.Collection)valorAsientos).size());
helper.createSlotWithValue("nroAvion",
                        ((java.lang.Integer)valorNroAvion).intValue());
helper.createSlotWithValue("numero", ((java.lang.Integer)valorNumero).intValue());
```

Como se puede observar en el código, solamente es necesario especificar cuáles valores serán asignados a la instancia de la definición abstracta, que fue creada anteriormente por ePlatero y que es conocida por el objeto helper. Estos valores van a depender de los valores de las propiedades de la instancia de la definición concreta, también conocida por el helper, los cuales deben ser recuperados a través de uno de sus métodos.

Como trabajo futuro para mejorar la herramienta ePlatero y, en particular, para hacer más dinámica la construcción de micromundos a través de ella, la codificación de la función de abstracción podrá ser reemplazada por la utilización de un convertor de OCL a Java, ya que en realidad el código java de la función de abstracción se corresponde con el mapping del refinamiento (escrito en OCL).

8.6. Estructura del componente

La estructura del módulo para la generación de micromundos consta de una clase muy importante, llamada MicroWorldFactory, la cual contiene métodos necesarios para la generación del micromundo. Esta clase, que se muestra en la figura 39, interactúa con otras clases para generar las particiones, para combinar dichas particiones y para generar los fragmentos de modelo y objeto a partir de los cuales se crearán las instancias que forman el micromundo.

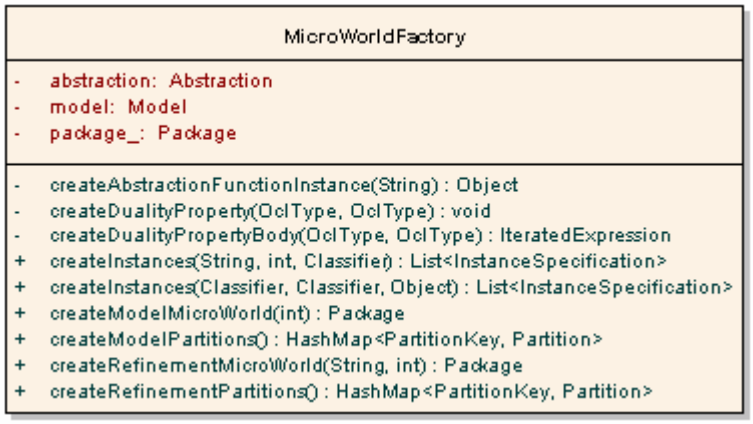


Figura 39 - Clase MicroWorldFactory

MicroWorldFactory es la clase principal del componente de generación de micromundos. A esta clase llegan los mensajes para la construcción de micromodelos desde el resto del plugin. Si bien es muy importante, prácticamente no tiene lógica para la creación de las particiones ni para generar los fragmentos de objeto y de modelo. Para ello, se nutre de otras clases colaboradoras de manera de no tener concentrada en una sola clase toda la lógica necesaria para generar los micromundos. Algunos de sus métodos serán descritos más adelante, puesto que esta clase aparecerá con frecuencia en el resto de esta sección.

En las siguientes subsecciones se mostrarán las clases que colaboran con la clase MicroWorldFactory y se describirá el proceso para crear micromundos de principio a fin. Para entender mejor este proceso, el mismo será dividido en dos pasos: el primer paso involucra todo lo relacionado a la generación de particiones, y el segundo

involucra todo lo referente a la generación del fragmento de modelo, y la creación del micromundo.

Recordemos que ePlatero permite generar micromundos tanto para poner a prueba un refinamiento, en donde solo se generan instancias representativas para las clases involucradas en el mapping de la abstracción, como para un modelo de clases completo. Sólo se explicará el proceso para la primera de estas dos alternativas, ya que la segunda es bastante parecida. La única diferencia existente en el proceso para la construcción de micromundos de un modelo de clases completo es que las particiones se crean para todas las propiedades de todas las clases del modelo, en lugar de para las propiedades de la clase concreta del refinamiento.

8.6.1. Generación de particiones

La siguiente figura muestra los colaboradores de MicroWorldFactory para la generación de particiones.

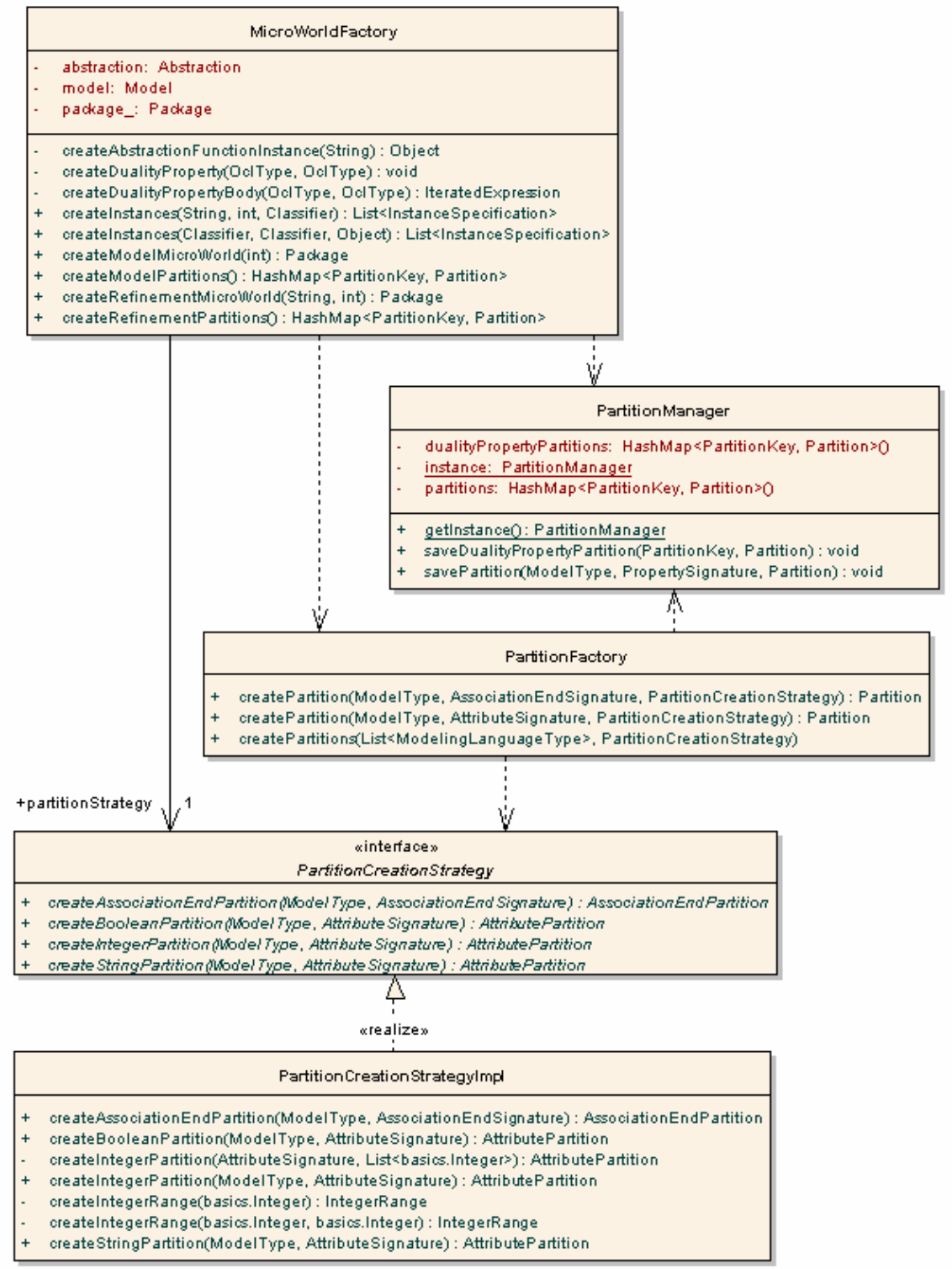


Figura 40 - Colaboradores de MicroWorldFactory para la generación de particiones

MicroWorldFactory

En esta parte es importante notar que implementa métodos, tanto para la creación de particiones para las propiedades que forman parte del mapping de una abstracción (refinamiento), como para las propiedades de todas las clases de un modelo:

- createModelPartitions( ): crea particiones para todas las propiedades de todas las clases presentes en el modelo UML.

- createRefinementPartitions( ): crea particiones para los atributos y asociaciones de la definición concreta, separando por un lado las particiones de propiedades involucradas en el refinamiento, y por el otro las particiones de propiedades no involucradas en el refinamiento. Para verificar qué particiones están involucradas en el refinamiento se crea la propiedad de dualidad entre la definición abstracta y la concreta (a partir del mapping de la abstracción), y luego se verifica qué propiedades de la definición concreta están dentro de la propiedad de dualidad.

PartitionManager

Esta clase implementa el patrón Singleton [**22**]. Tiene como responsabilidad administrar las particiones generadas.

PartitionFactory

Es la encargada de construir las particiones para las propiedades de las clases. Todos sus métodos reciben como parámetro una estrategia (clase que implementa la interfaz PartitionCreationStrategy) en quién delega la creación de las particiones. Una vez creada la partición para una propiedad de una clase, interactúa con la clase PartitionManager para guardar la partición creada. Implementa métodos para crear particiones para todas las propiedades de una lista de modelType y para crear particiones para una propiedad particular de un modelType.

PartitionCreationStrategy

Esta interfaz define los métodos que deben ser implementados por cualquier clase dedicada a la creación de particiones. ePlatero tiene, actualmente, solo una estrategia para construir particiones. Si en un futuro se desea implementar una nueva estrategia, la nueva clase deberá implementar la interfaz PartitionCreationStrategy.

PartitionCreationStrategyImpl

Clase que implementa la interfaz antes descrita. Su principal responsabilidad es la de crear particiones para los diferentes tipos de datos soportados por ePlatero. Entre los métodos que implementa están:

- createIntegerPartition(…): retorna una partición de enteros para el atributo recibido como argumento. El atributo pasado como parámetro debe representar a un atributo de tipo Integer. Esta estrategia retorna una partición con a lo sumo tres rangos: un rango que cubre el valor mínimo para el atributo, otro que cubre el intervalo [valorMínimo + 1 .. valorMaximo - 1] y el último rango que cubre el valor máximo para el atributo.

- createAssociationEndPartition(…): retorna una partición para la asociación recibida como argumento. La cantidad de rangos generados depende de la multiplicidad de la asociación; a lo sumo son generados tres rangos: un rango

que cubre la multiplicidad mínima, otro que cubre el intervalo [multMinima + 1 .. multMaxima - 1] y el último rango que cubre la multiplicidad máxima para la asociación.

- createStringPartition(…): retorna una partición de strings para el atributo recibido como argumento. La partición tendrá dos rangos, uno con el string vacío y otro con un string no vacío.

- createBooleanPartition(…): retorna una partición de valores booleanos para el atributo recibido como argumento. Genera dos rangos, uno para cada posible valor booleano (true y false).

Como se puede observar, los métodos que crean particiones para atributos enteros y para asociaciones son una adaptación de lo propuesto en [**11**]. En lugar de generar una partición {{0}, {1}, {2..inf}} consideramos que es más representativo generar una partición {{valor mínimo}, {valorMínimo + 1 .. valorMaximo - 1}, {valorMaximo}}.

8.6.1.1. Proceso para la generación de particiones

Hasta aquí hemos visto las clases involucradas en la creación de las particiones, con las responsabilidades de cada una de ellas.

Ahora, vamos a describir el proceso de generación de particiones para las clases involucradas en un refinamiento:

1) Se recupera la clase abstracta y su correspondiente clase concreta, las cuales están involucradas en el mapping de la abstracción. Estas clases son instancias de ModelingLanguageType, subclase de OCLType.
2) Se delega en la clase PartitionFactory la creación de las particiones de las propiedades de la clase concreta, pasándole una estrategia para generar las particiones, es decir, una estrategia para generar los rangos de las particiones. La clase PartitionFactory trabaja junto a la estrategia para generar las particiones, tanto para los atributos como para las multiplicidades de las asociaciones, interactuando con la clase PartitionManager que es la clase encargada de administrar las particiones generadas.
3) Se crea la propiedad de dualidad entre la definición abstracta y la concreta, y se recuperan las propiedades de la definición concreta involucradas en el mapping de la abstracción. Para esto se utiliza una instancia de la clase DualityPropertyVisitor (que implementa el patrón Visitor [**22**]) que "visita" la propiedad de dualidad.
4) Luego, se indica al PartitionManager que mueva las particiones de las propiedades identificadas en el paso anterior a una colección diferente. De esta manera la clase PartitionManager tendrá dos colecciones de particiones, una con las particiones de las propiedades involucradas en el mapping, y otra con el resto de las particiones. El motivo por el cual se diferencian estas dos colecciones de particiones es que, en la generación de los fragmentos de objeto, van a ser tenidos en cuenta todos los rangos de las particiones de propiedades involucradas en el mapping, mientras que para las particiones de propiedades no involucradas solo se tendrá en cuenta un rango aleatorio dentro de sus rangos posibles.

La siguiente figura muestra un diagrama de interacción resumido del proceso descrito anteriormente:

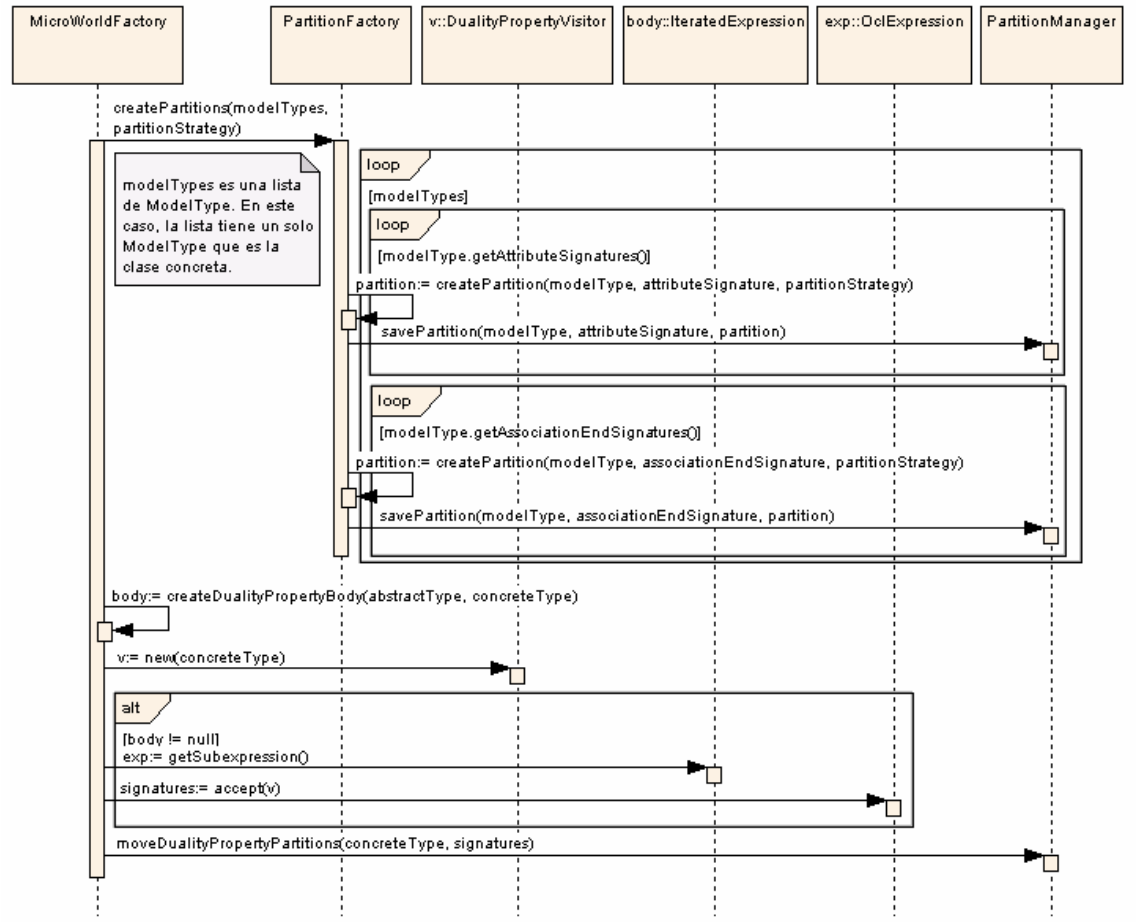


Figura 41 - Generación de particiones para las clases involucradas en un refinamiento

La figura anterior muestra como la clase MicroWorldFactory delega en el PartitionFactory la creación de particiones. Luego crea una instancia (body) de la clase IteratedExpression que representa la propiedad de dualidad. A través del patrón visitor recupera todas las propiedades de la definición concreta involucradas en el mapping de la abstracción, y finalmente le dice al PartitionManager que mueva las particiones de esas propiedades a una colección diferente.

8.6.2. Generación del fragmento de modelo y creación del micromundo

Una vez generadas las particiones para un modelo con un refinamiento es posible generar micromundos. Nuevamente recordemos, que aquí estamos explicando la generación de micromundos para un modelo que contiene un refinamiento.

La figura 42 muestra las clases más importantes que participan en la generación del micromundo. Nuevamente aparece la clase MicroWorldFactory como encargada de dirigir la construcción de los micromundos. También aparece una clase que se explicó en la sección 8.4., la clase ModelFragment. Esta última clase representa a un fragmento de modelo y es la que contiene a todos los fragmentos de objeto.

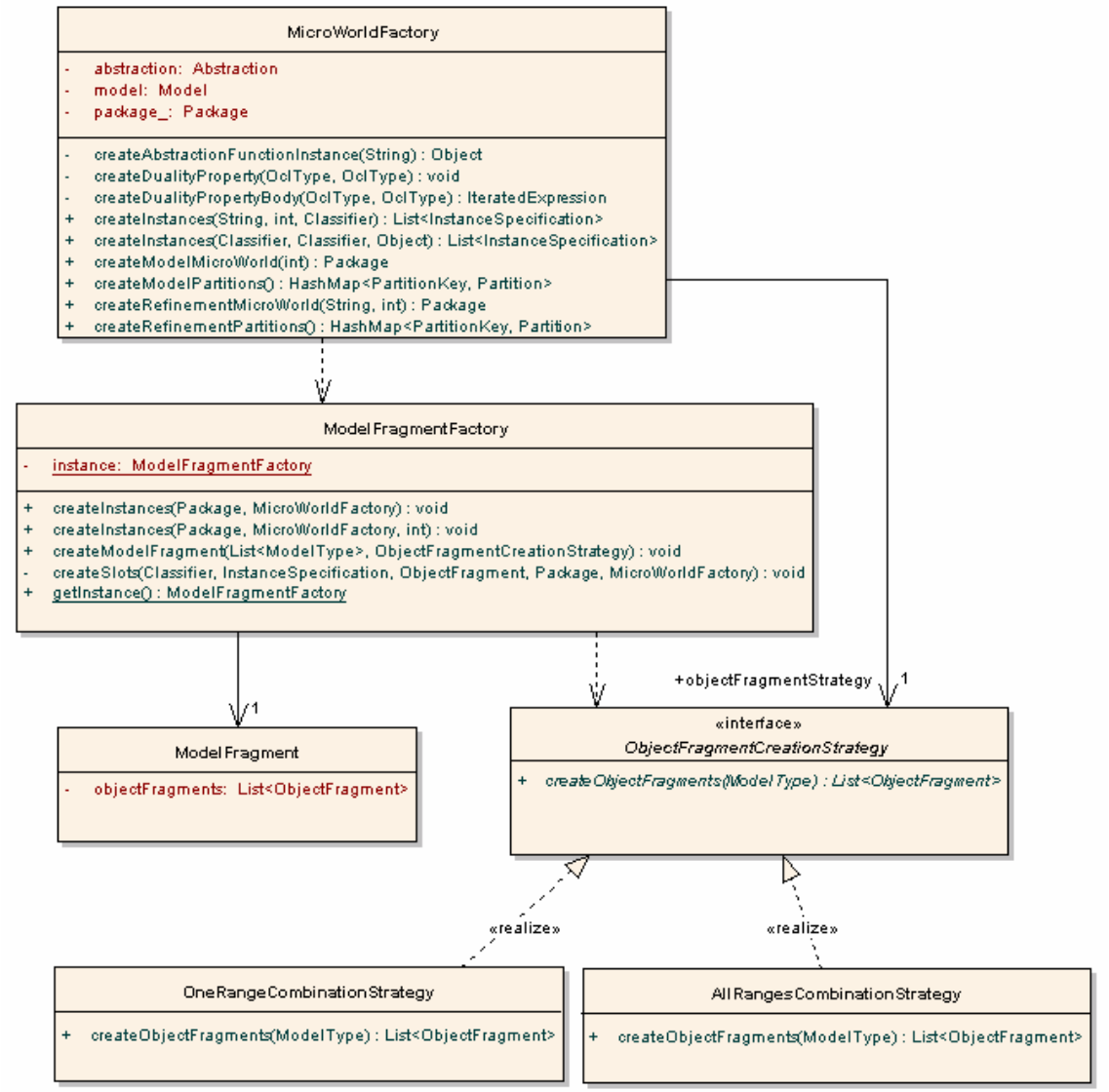


Figura 42 - Colaboradores de MicroWorldFactory para la generación de micromundos

MicroWorldFactory

En esta parte es importante notar que implementa métodos, tanto para la creación de micromundos para un modelo, como para un refinamiento:

- createModelMicroWorld(…): crea un fragmento de modelo teniendo en cuenta todas las clases del modelo UML, y utilizando una estrategia para la creación de fragmentos de objeto determinada. A partir de este fragmento de modelo crea el micromundo, el cual puede contener todas las instancias representativas o una cantidad determinada de instancias haciendo, quizá, el micromundo menos representativo.

- createRefinementMicroWorld(…): crea un fragmento de modelo teniendo en cuenta solamente la clase concreta del refinamiento, y utilizando una estrategia para la creación de fragmentos de objeto determinada. A partir de este fragmento de modelo crea el micromundo, el cual tiene solamente instancias de la definición concreta, y puede contener todas las instancias representativas o una

cantidad determinada de instancias haciendo, quizá, el micromundo menos representativo. A partir de todas las instancias de la definición concreta y utilizando una función de abstracción determinada, crea las instancias de la definición abstracta, cumpliéndose de esta forma la propiedad de dualidad.

ModelFragment

Como se dijo, representa un fragmento de modelo. Agrupa todos los fragmentos de objeto. En nuestra implementación se genera una sola instancia de la clase ModelFragment para todas las clases involucradas en la condición de refinamiento.

ModelFragmentFactory

Esta clase se utiliza para crear un fragmento de modelo con un conjunto de fragmentos de objeto generados de acuerdo a una estrategia; y para, a partir de ese fragmento de modelo, poder crear las instancias correspondientes.

ObjectFragmentCreationStrategy

Esta interfaz define los métodos que deben ser implementados por cualquier clase dedicada a la creación de fragmentos de objeto. ePlatero tiene, actualmente, las dos estrategias explicadas en la sección 7.5.2.1: OneRangeCombinationStrategy y AllRangesCombinationStrategy. Si en un futuro se desea implementar una nueva estrategia, la nueva clase deberá implementar ObjectFragmentCreationStrategy.

8.6.2.1. Proceso para la generación del fragmento de modelo y creación del micromundo

A continuación se va a describir el proceso de generación del fragmento de modelo y la creación directa, a partir de este, del micromundo:

1) Se recupera la clase abstracta y su correspondiente clase concreta, las cuales están involucradas en el mapping de la abstracción. Estas clases son instancias de ModelingLanguageType, subclase de OCLType.
2) Se delega en la clase ModelFragmentFactory la creación del fragmento de modelo, teniendo en cuenta solamente la clase concreta y utilizando la estrategia para la generación de fragmentos de objeto seleccionada por el usuario en la utilización del plugin.
3) Luego se verifica si existe una cantidad de instancias limitada a generar, si es así se delega en la clase ModelFragmentFactory la creación de a lo sumo esa cantidad de instancias, en caso contrario se crean todas las instancias correspondientes a los fragmentos de objeto generados. Para crear las instancias que formarán parte del micromundo se utiliza el fragmento de modelo creado en el paso previo.
4) Se crea una clase que contiene un método con el código de la función de abstracción, contenido en el archivo adjuntado al proceso en la utilización del plugin.
5) Utilizando la clase creada en el paso anterior se crea, para cada una de las instancias de la definición concreta, la instancia correspondiente de la definición abstracta, agregándose de esta forma al micromundo resultante.

La figura 43 muestra un diagrama de interacción resumido del proceso descrito anteriormente:

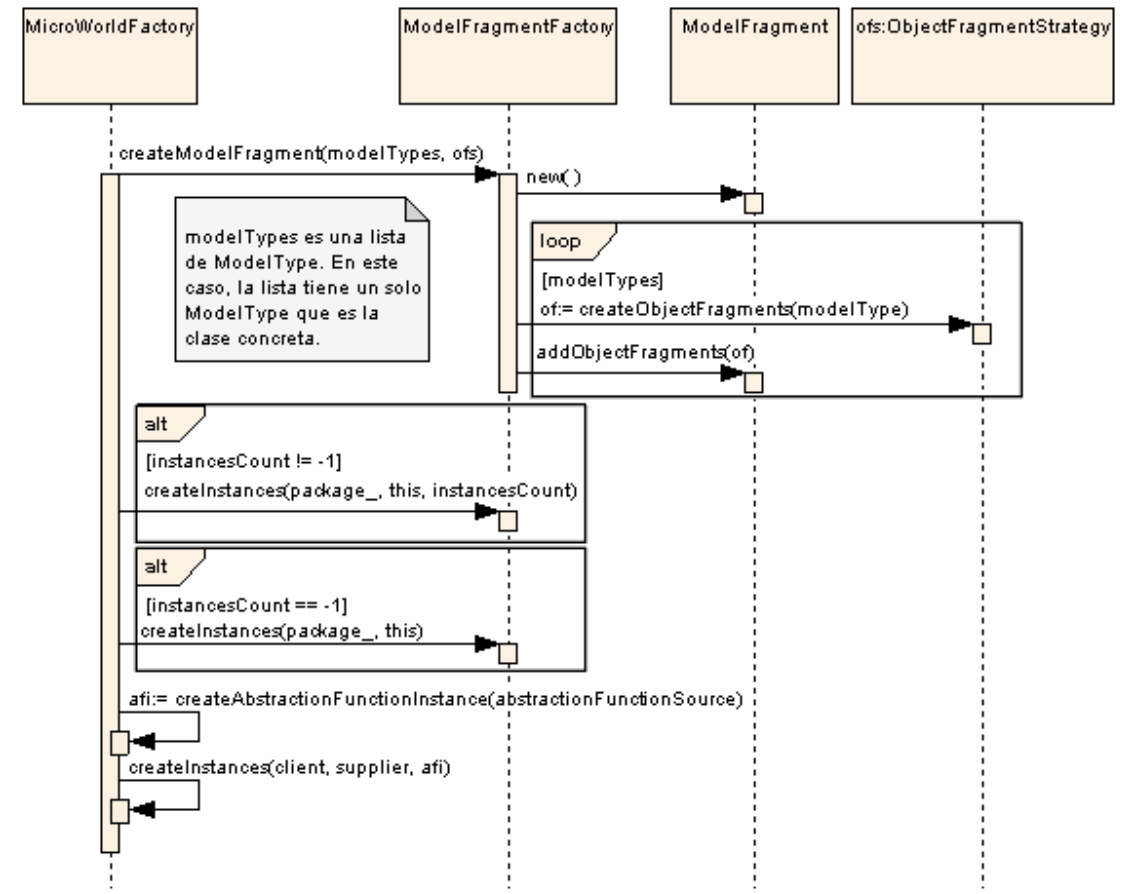


Figura 43 - Generación del fragmento de modelo y creación del micromundo

8.7. Patrones de diseño utilizados

Los patrones de diseño son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces.

Un patrón de diseño es una solución a un problema de diseño no trivial que es efectiva (ya se resolvió el problema satisfactoriamente en ocasiones anteriores) y reusable (se puede aplicar a diferentes problemas de diseño en distintas circunstancias).

A continuación presentamos algunos de los patrones de diseño que han sido fundamentales para el desarrollo el módulo de generación de micromundos.

8.7.1. Strategy

Como se mencionó en otras secciones ePlatero implementa dos estrategias para combinar las particiones de rangos. Por un lado tenemos la estrategia

AllRangesCombinations, en la cual todas las combinaciones posibles de rangos para las propiedades de una clase aparecen en un fragmento de objeto, y por otro, la estrategia OneRangeCombination, la cual establece que cada rango para cada propiedad de una clase necesita ser usada en al menos un fragmento de objeto.

Estas estrategias fueron implementadas siguiendo el patrón de comportamiento Strategy [**22**], el cual fue adaptado.

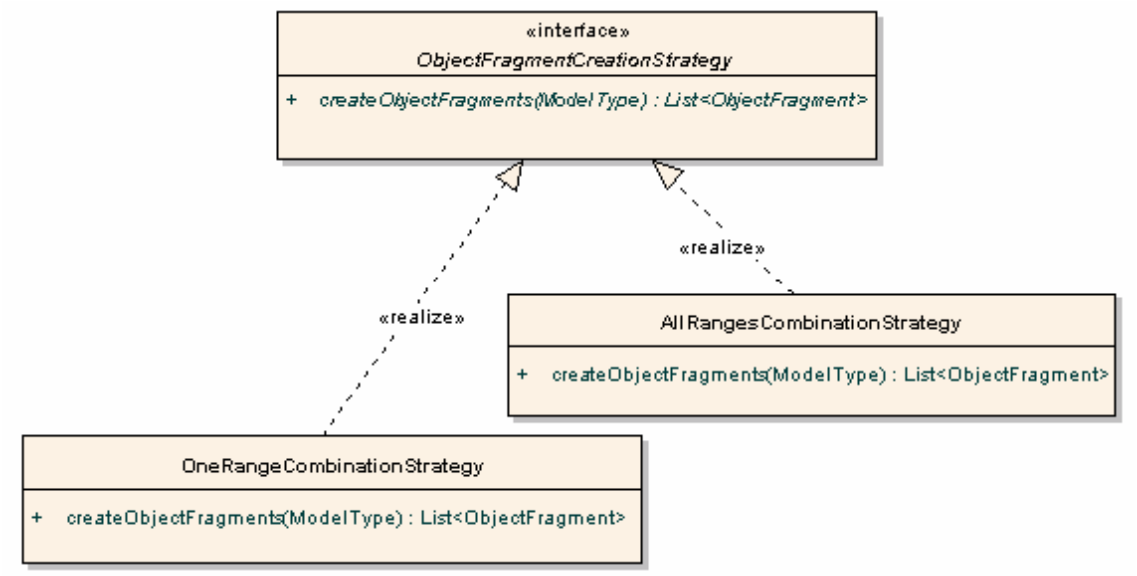


Figura 44 - Utilización del patrón Strategy

En la implementación se definió una interfaz ObjectFragmentCreationStrategy que define un único método llamado createObjectFragments el cual recibe como parámetro un ModelType y retorna una lista de fragmentos de objeto para ese ModelType. De esta manera cada estrategia de generación de fragmentos de objeto debe ser desarrollada como una clase que implementa esta interface, implementando ese método de acuerdo a su propio criterio. Dado que existe un factory que genera un fragmento de modelo y que espera recibir algo de tipo ObjectFragmentCreationStrategy, puede ser agregada una nueva estrategia para generar fragmentos de objeto a ePlatero simplemente agregando una nueva clase que implemente la interfaz ObjectFragmentCreationStrategy y reimplementando el método createObjectFragments.

8.7.2. Visitor

En programación orientada a objetos, el patrón visitor [**22**] es una forma de separar el algoritmo de la estructura de un objeto. En la implementación del modulo de generación de micromundos fue necesario implementar este patrón para determinar las propiedades de la definición concreta que están involucradas en la propiedad de dualidad. En la herramienta, la propiedad de dualidad está representada como una expresión OCL la cual es procesada con una clase que implementa el patrón Visitor, para determinar las propiedades involucradas en ella.

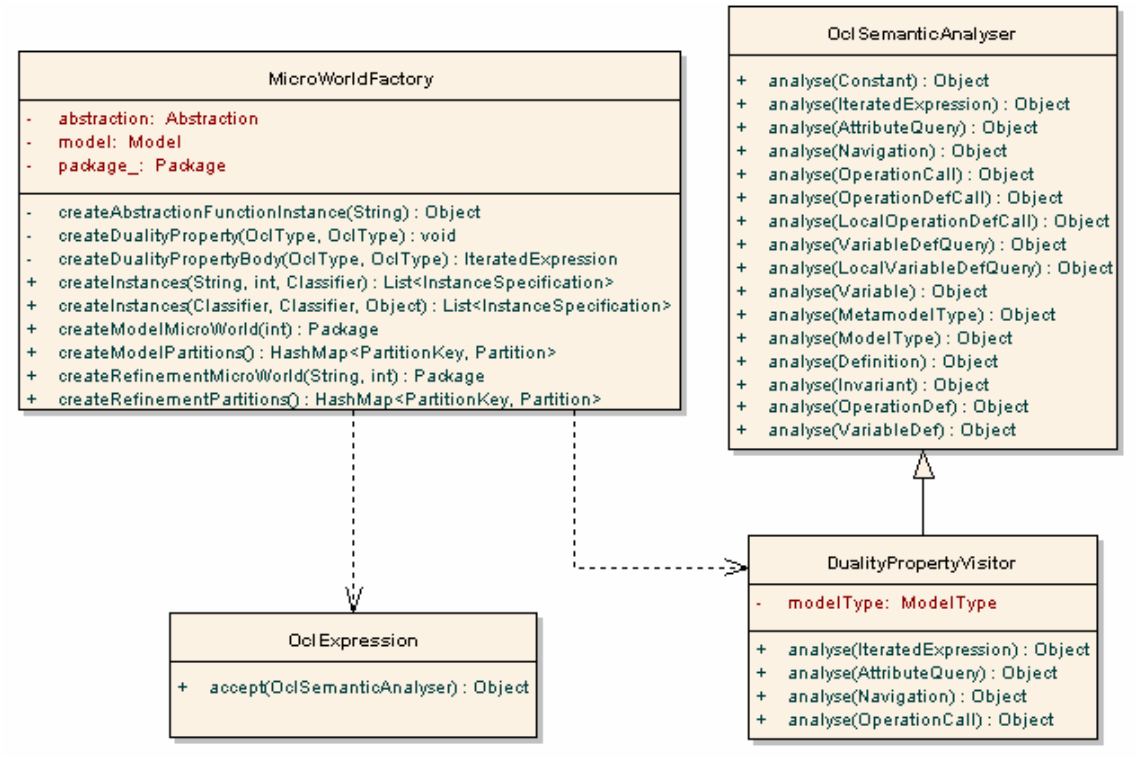


Figura 45 - Utilización del Patrón Visitor

9. Caso de estudio

Este capítulo tiene como principal objetivo mostrar la funcionalidad implementada en la herramienta ePlatero para la generación de micromundos. Para ello, tomaremos un caso de estudio y describiremos paso a paso el proceso para generar micromundos y evaluar refinamientos. El caso de estudio elegido ya ha sido presentado a lo largo de este trabajo, por lo que le será familiar al lector. Finalmente se comparan los resultados obtenidos y los tiempos necesarios para alcanzar los mismos, con una herramienta existente llamada Alloy Analyser [**24**].

En la sección 9.1 se presenta el dominio del caso de estudio, en la sección 9.2 se explica cómo especificar el modelo en ePlatero, en 9.3 se detalla como especificar las reglas OCL del caso de estudio en la herramienta, en 9.4 se explica cómo parsear el archivo OCL, en 9.5 se explica cómo generar los micromundos de acuerdo a alguna de las estrategias implementadas, en la sección 9.6 se explica cómo evaluar un refinamiento y en 9.7 se da una comparación de tiempos con Alloy.

Este capítulo no pretende ser un guía de usuario de ePlatero, por lo que algunos detalles del manejo de la herramienta y del IDE eclipse han sido omitidos.

9.1. Dominio

El siguiente diagrama de clases muestra las clases y el refinamiento que usaremos como caso de estudio.

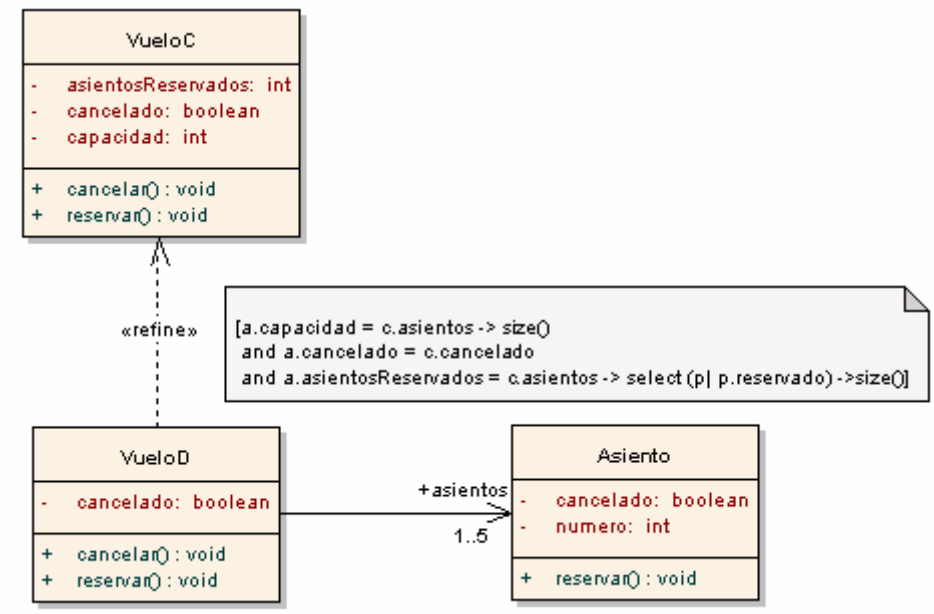


Figura 46 - Diagrama de clases del caso de estudio

Como vemos existe una clase llamada VueloC. Esta clase es la clase abstracta, es decir, la clase que queremos refinar. También aparece la clase VueloD que representa a la clase VueloC refinada. Finalmente, podemos observar en el modelo que existe un

refinamiento, es decir, un conjunto de reglas OCL que se deberán cumplir cuando generemos el micromundo. En el mapping de este refinamiento se establece que las propiedades capacidad y asientosReservados de la clase VueloC son, ahora, representadas a través de una relación entre la clase VueloD y la clase Asiento.

9.2. Creación del modelo

Como primer paso debemos pasar nuestro modelo de clases a ePlatero. Para ello necesitamos crear cualquier tipo de proyecto (por ejemplo un proyecto Java), y luego agregar al proyecto un "ePlatero Class Diagram".

Para poder crear el modelo de clases, la herramienta nos provee de un editor UML2, con ciertas bondades que facilitan el modelado:

- ✓ tool palette and overview
- ✓ layout and selection tools
- ✓ diagram image export (svg, bmp, jpeg, gif)
- ✓ tabbed properties view
- ✓ font and color options for selected element
- ✓ pop-up bars and connection handles
- ✓ notes and geometric shapes
- ✓ animated zoom and layout
- ✓ diagram printing

La figura 47 muestra el diagrama presentado en la sección previa modelado con ePlatero.

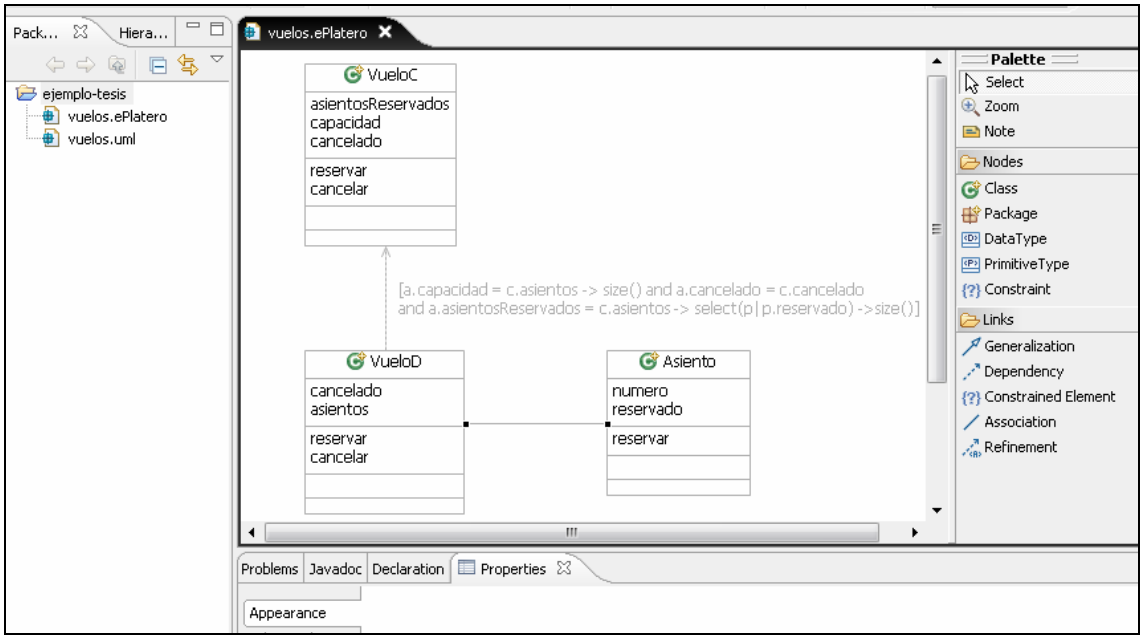


Figura 47 - Editor de diagramas UML de ePlatero

Ahora debemos escribir las reglas OCL para nuestro modelo.

9.3. Definición de reglas OCL

El siguiente paso es escribir las reglas OCL para nuestro modelo de clases. Para ello debemos agregar al proyecto un OCL File. ePlatero cuenta con un editor OCL (descripto en los anexos de este trabajo) que nos brinda:

- ✓ highlighting de sintaxis
- ✓ asistencia de código
- ✓ corrección de errores

En este editor debemos escribir las reglas OCL que ayudan a determinar si el refinamiento planteado es correcto o no. Las reglas OCL para nuestro modelo son las siguientes:

Restricciones OCL para la definición abstracta:

```
context VueloC
  inv:  Set{1..5} -> includes(self.capacidad)
  inv:  Set{0..5} -> includes(self.asientosReservados)

context VueloC::asientosReservados: Integer
  init: 0

context VueloC::cancelado: Boolean
  init: false

context VueloC::cancelar()
  pre:  not self.cancelado
  post: self.cancelado and self.capacidad = self.capacidad@pre and
        self.asientosReservados = self.asientosReservados@pre

context VueloC::reservar()
  pre:  self.capacidad - self.asientosReservados > 0 and
        not self.cancelado
  post: self.asientosReservados = self.asientosReservados@pre + 1
        and self.cancelado = self.cancelado@pre and
        self.capacidad = self.capacidad@pre
```

Restricciones OCL para la definición concreta:

```
context VueloD::cancelado: Boolean
  init: false

context VueloD
  inv:  self.asientos -> forAll( p,q | p.numero = q.numero
        implies q = p )

context VueloD::cancelar()
  pre:  not self.cancelado
  post: self.cancelado
        and self.asientos -> size() = self.asientos@pre -> size()
        and self.asientos -> forAll( p |
            self.asientos@pre -> exists( q | p.numero = q.numero
                                    and p.reservado = q.reservado) )
```

```
context VueloD::reservar()
  pre:  self.asientos -> exists( q | not q.reservado )
        and not self.cancelado
  post: self.cancelado = self.cancelado@pre and
        self.asientos -> size() = self.asientos@pre -> size() and
        self.asientos -> exists ( s | s.reservado and
        self.asientos@pre -> select( q | not q.reservado ) ->
        exists ( q | q.numero = s.numero ) and
        self.asientos -> forAll( p | p <> s implies
        self.asientos@pre -> exists( q | s.reservado = q.reservado
        and s.numero = q.numero)))

context Asiento
  inv:  Set{1..5} -> includes(self.numero)

context Asiento::reservado: Boolean
  init: false

context Asiento::reservar()
  pre:  not self.reservado
  post: self.reservado and self.numero = self.numero@pre
```

En ePlatero, se ve de la siguiente manera:

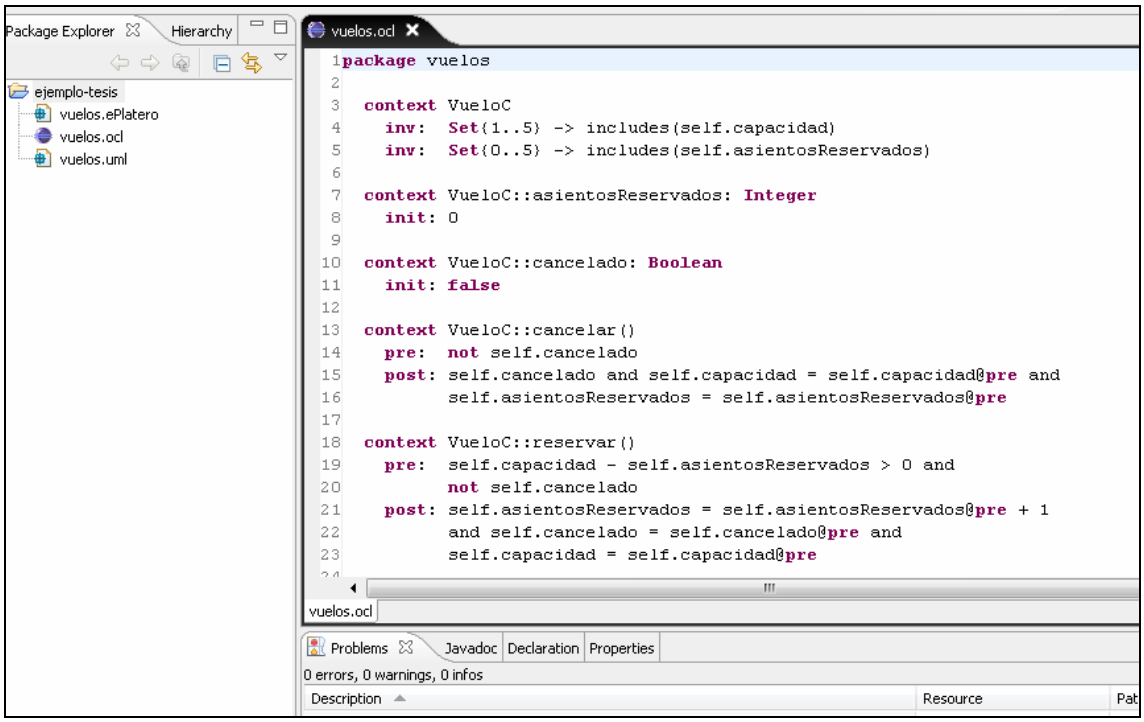


Figura 48 - Editor OCL de ePlatero

9.4. Parseo del archivo OCL

Una vez generado el archivo con las reglas OCL debemos parsearlo. Al parsear el archivo se levanta el modelo de clases creado mediante el Editor de ePlatero a clases de UML2, y luego se mapea ese modelo en UML2 a clases de OCL, propias del plugin.

Una vez que tenemos el modelo representado en clases de OCL, se procesa el contenido del archivo OCL y se completa el modelo de clases; es decir se agregan al modelo los invariantes, valores iniciales para atributos de clases, pre y post condiciones para los métodos de las clases, etc., definidos en el archivo OCL.

Cada cambio que hagamos en el modelo o en el archivo OCL, mediante alguno de los editores de ePlatero, va a ser tenido en cuenta para la generación de micromundos sólo si previamente a esta generación se parsea nuevamente el archivo OCL. En caso contrario, los cambios serán ignorados por el modulo generador de micromundos.

9.5. Generación del micromundo

Para poder generar un micromundo, tanto para evaluar un refinamiento como para crear instancias representativas de todo el modelo de clases, es necesario abrir el archivo con extensión .uml que se genera al crear el "ePlatero Class Diagram" (cuya extensión es .ePlatero). Este archivo permite navegar todos los objetos del diagrama de clases, mostrándolo en forma de árbol.

Como nuestra intención en este caso de estudio es evaluar un refinamiento, vamos a seleccionar la abstracción que representa al mismo y vamos a generar el micromundo. Esto se muestra en la figura 49.

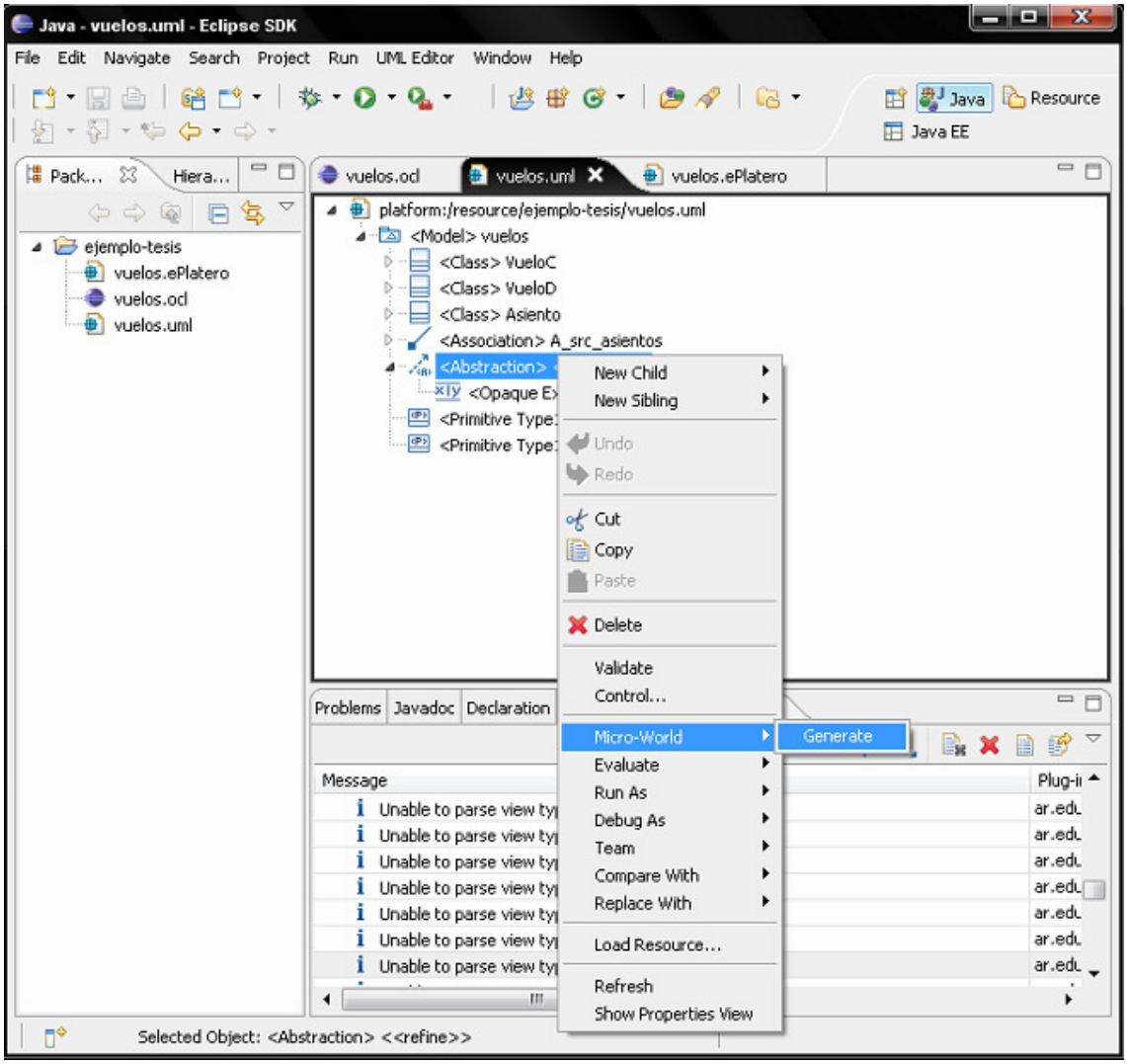


Figura 49 - Generación de micromundo en ePlatero

A continuación se abrirá una ventana que muestra las particiones generadas para los atributos de la definición concreta involucrados en el mapping de la abstracción. Esta ventana se muestra en la figura 50.

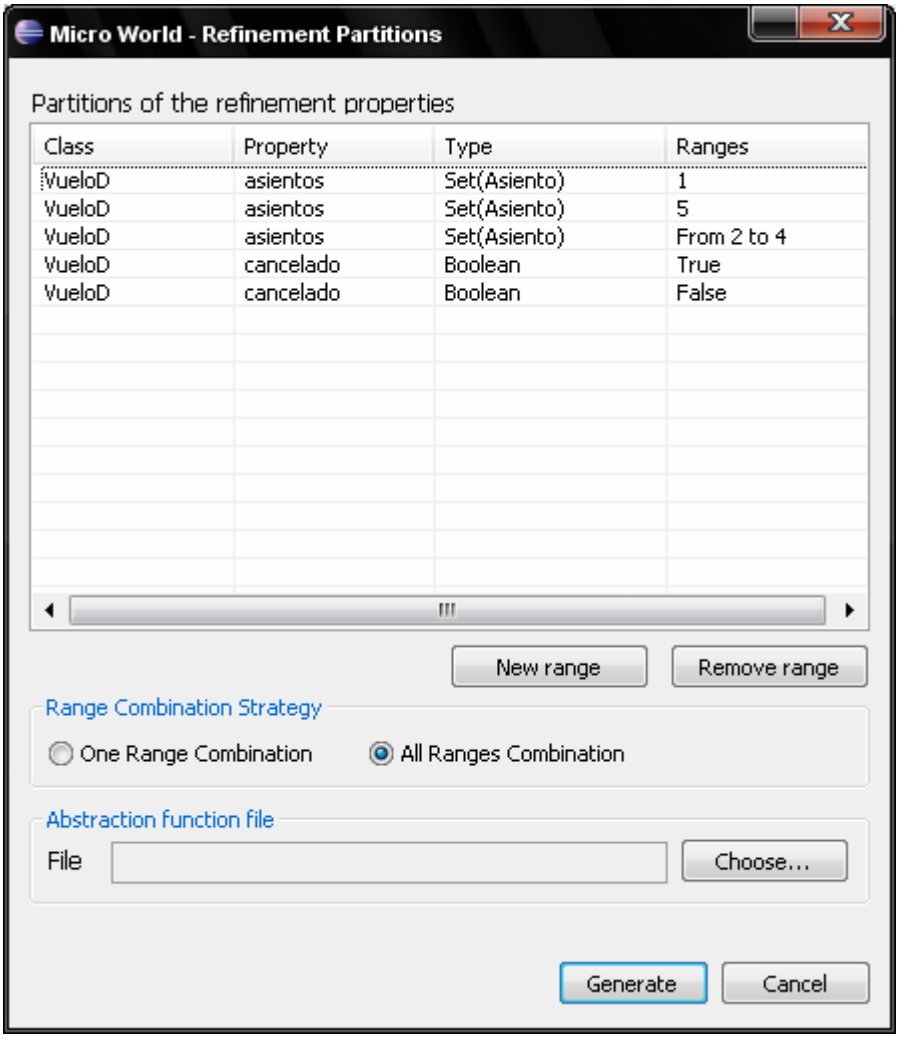


Figura 50 - Particiones generadas con ePlatero

Como se puede observar en la figura es posible agregar nuevos rangos, modificar y eliminar rangos existentes, seleccionar una estrategia de combinación de rangos, adjuntar una función de abstracción y establecer un límite de instancias a generar.

Sólo se podrán agregar rangos a particiones existentes, quedando a criterio del usuario introducir rangos representativos. La forma de escribir un rango válido es la siguiente:

- Si el tipo de la propiedad es Boolean, el rango deberá ser:
  - True
  - False
- Si el tipo de la propiedad es Integer o si la propiedad es una asociación con otra clase, el rango deberá ser:
  - Un número entero
  - Un intervalo de la forma “from a to b” donde a y b son enteros y a <= b
- Si el tipo de la propiedad es String, el rango deberá ser cualquier cadena de caracteres, incluyendo el string vacío.

La figura 51 muestra la ventana para agregar nuevos rangos.

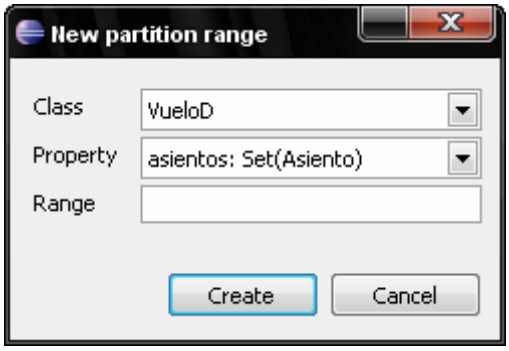


Figura 51 - Ventana para agregar nuevos rangos en ePlatero

Para modificar rangos se deberá seleccionar en la tabla de particiones de la figura 50 el rango que se desea modificar y escribir directamente sobre él, el nuevo rango. Los rangos se deben escribir siguiente las reglas mencionadas anteriormente.

Para eliminar rangos se deberá seleccionar la fila que contiene el rango para luego eliminarlo. Un punto a tener en cuenta es que sólo se podrá eliminar un rango si existe otro rango para la misma partición; es decir, no es posible eliminar todos los rangos de una partición.

Además es posible establecer un límite de cantidad de instancias a generar que, en el caso del micromundo para testear un refinamiento, limita la cantidad de instancias de la definición concreta que se van a crear, y en el caso del micromundo para un modelo de clases completo, limita la cantidad de instancias representativas a crear (sumando las instancias de todas las clases). En este último caso, cuando hablamos de instancias representativas nos referimos a las instancias generadas a partir de los fragmentos de objeto creados, sin tener en cuenta las instancias que son creadas por ser un conocimiento de una instancia representativa. El límite ingresado permite generar micromundos más pequeños, pero a la vez menos representativos.

Por último, para generar el micromundo para testear un refinamiento, estamos obligados a adjuntar la función de abstracción, la cual nos permite crear una instancia de la definición abstracta a partir de una instancia de la definición concreta, de forma que se cumpla la propiedad de dualidad. La función de abstracción para este caso de estudio es la siguiente:

```
Object asientos = helper.getConcreteInstanceProperty("asientos");
Object cancelado = helper.getConcreteInstanceProperty("cancelado");

java.util.Iterator iteradorAsientos =
                        ((java.util.Collection)asientos).iterator();
int asientosReservados = 0;
org.eclipse.uml2.uml.InstanceSpecification asiento;
Object reservado;
while (iteradorAsientos.hasNext()) {
      asiento = (org.eclipse.uml2.uml.InstanceSpecification)
                iteradorAsientos.next();
      reservado = helper.getInstanceProperty(asiento, "reservado");
      if (((java.lang.Boolean)reservado).booleanValue())
            asientosReservados++;
}
```

```
helper.createSlotWithValue("capacidad",
                        ((java.util.Collection)asientos).size());
helper.createSlotWithValue("asientosReservados", asientosReservados);
helper.createSlotWithValue("cancelado",
                        ((java.lang.Boolean)cancelado).booleanValue());
```

Figura 52 - Función de abstracción para el caso de estudio

A continuación se mostrarán los micromundos generados combinando las distintas posibilidades.

9.5.1. Micromundo para OneRangeCombination y sin límite de instancias

Si seleccionamos la estrategia OneRangeCombination y no establecemos límite de cantidad de instancias a generar, el resultado es el de la figura 53.

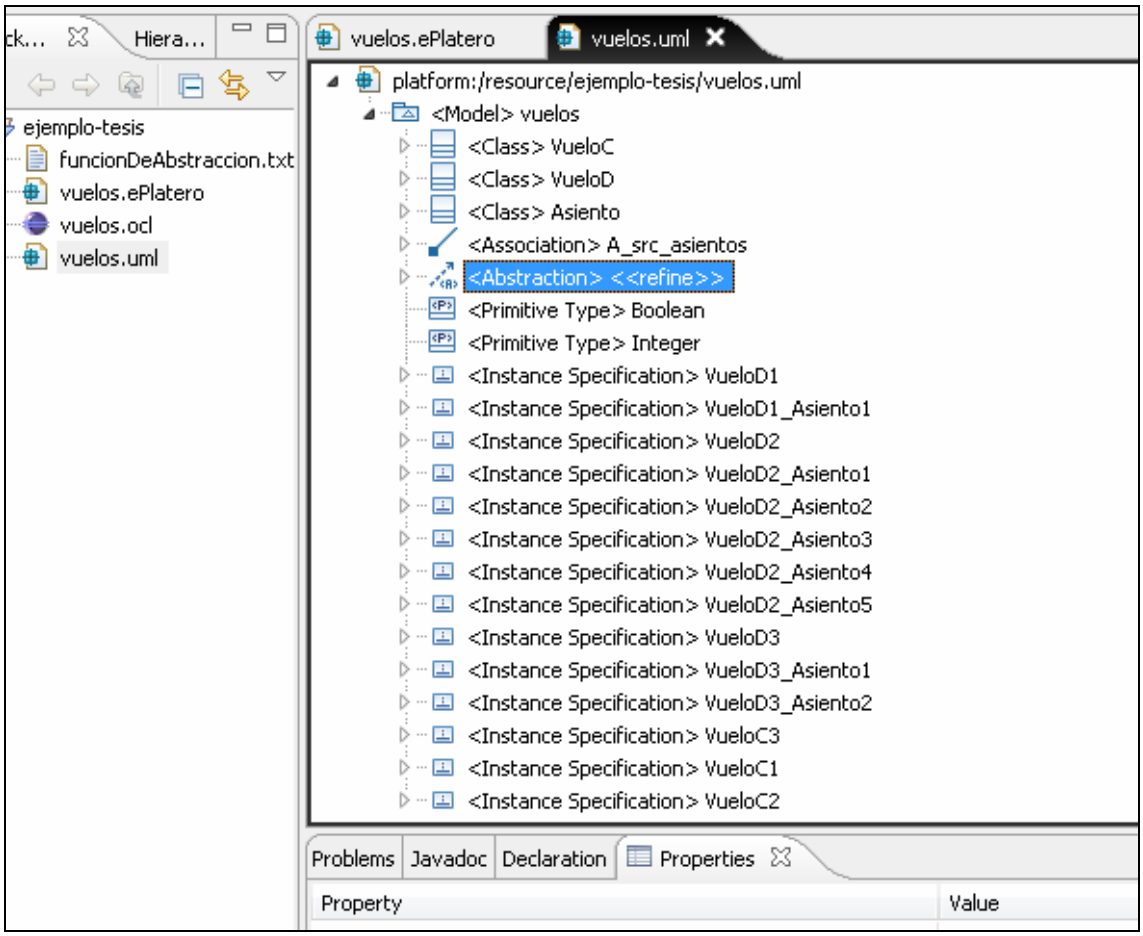
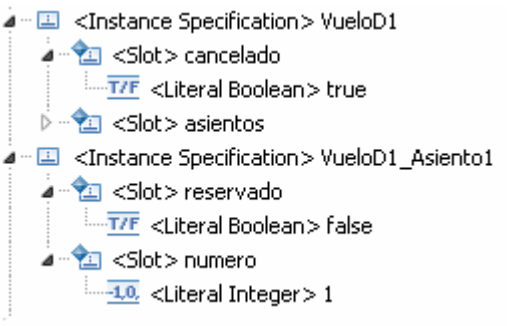


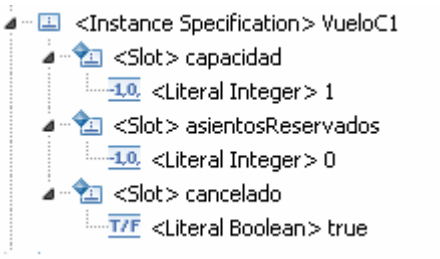
Figura 53 - Micromundo generado por ePlatero para la estrategia OneRangeCombination y sin establecer límite de cantidad de instancias

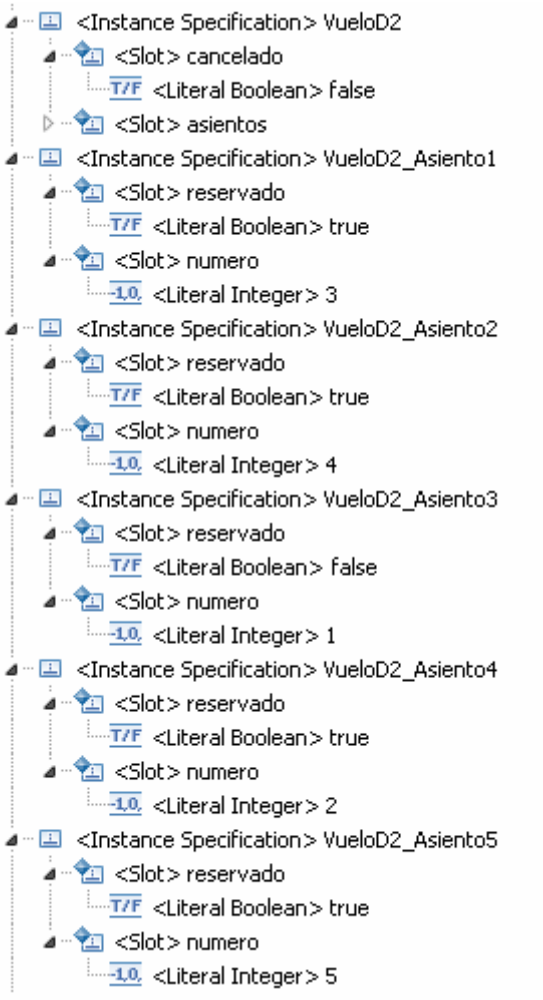
En la figura se puede apreciar que las instancias generadas tienen nombres representativos. Por ejemplo, los asientos asociados con los objeto VueloD2 reciben el nombre VueloD2_Asiento# y a su vez el objeto que representa la abstracción de VueloD2 se llama VueloC2. Estos nombres los elegimos para facilitar la lectura del

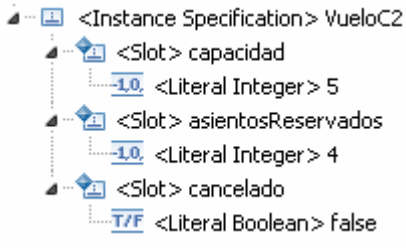
micromundo y poder verificar en forma rápida si se cumple la propiedad de dualidad, determinando de esa forma la correctitud de la función de abstracción escrita.

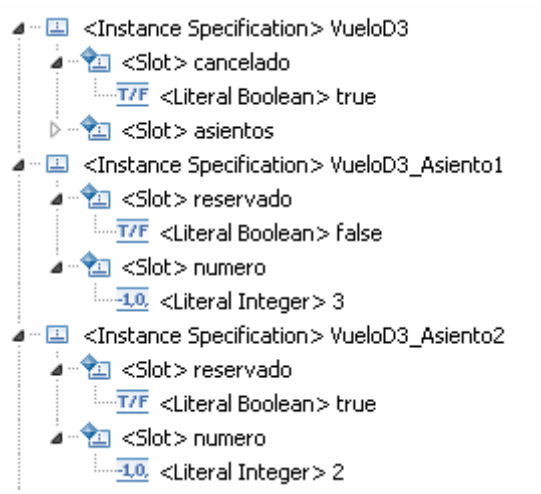
Si expandimos cada una de las instancias podemos verificar que se cumple la propiedad de dualidad.

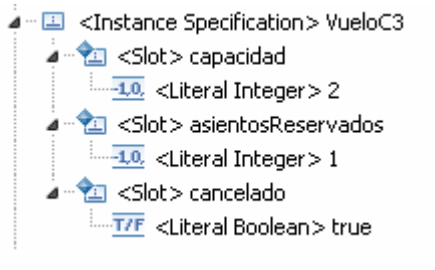












Al expandir las instancias también verificamos que se han utilizado todos los rangos de particiones en al menos una instancia, y que los asientos pertenecientes a un mismo vuelo tienen números distintos, siguiendo lo que indica el invariante:

```
context VueloD
  inv:  self.asientos -> forAll( p,q | p.numero = q.numero
        implies q = p )
```

En los siguientes micromundos no se mostrarán los números de los asientos porque la intención es mostrar las instancias generadas de manera de verificar que se cumple la propiedad de dualidad.

9.5.2. Micromundo para AllRangesCombination y sin límite de instancias

Si seleccionamos la estrategia AllRangesCombination y no establecemos límite de cantidad de instancias a generar, el resultado es el de la figura 54.

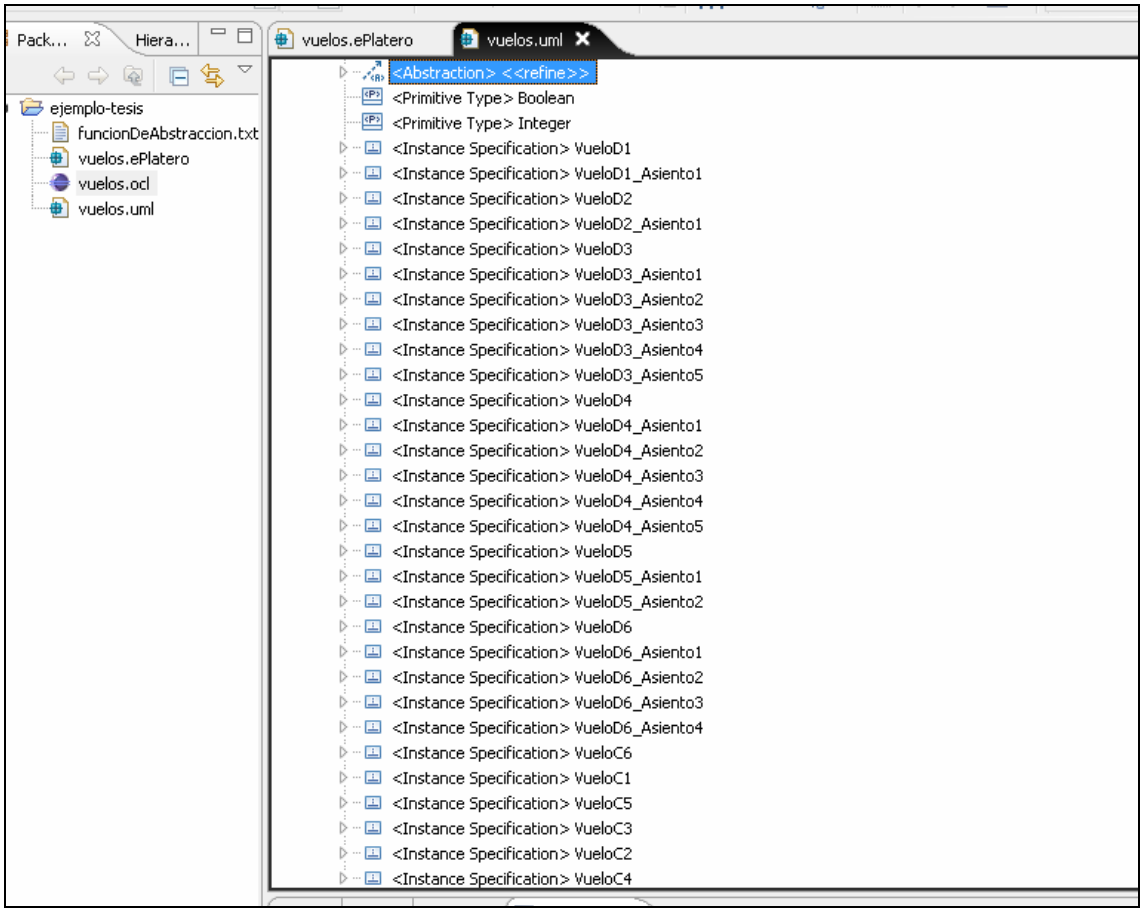
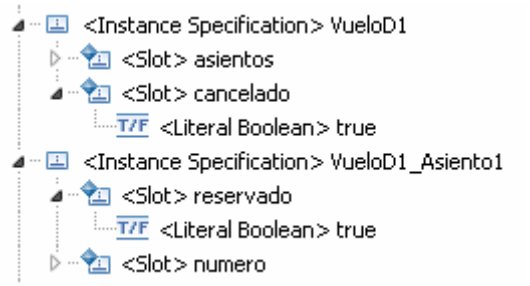
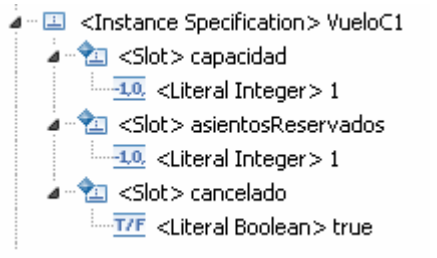
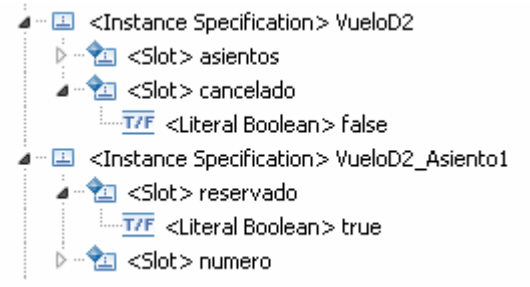


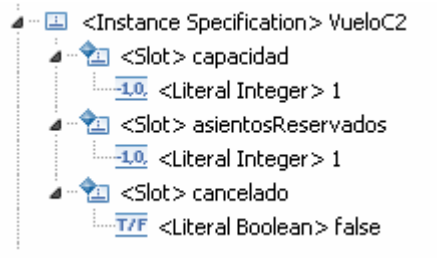
Figura 54 - Micromundo generado por ePlatero para la estrategia AllRangesCombination y sin establecer límite de cantidad de instancias

Si expandimos cada una de las instancias podemos verificar que se cumple la propiedad de dualidad.

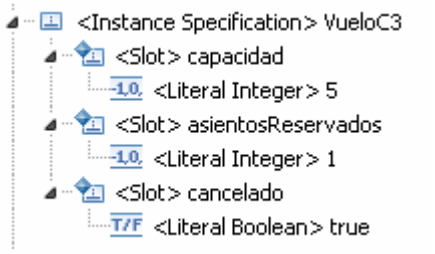




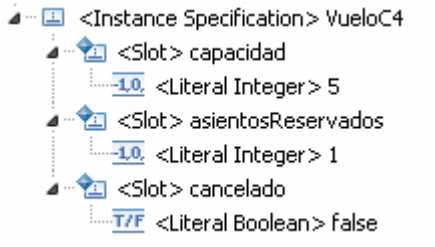


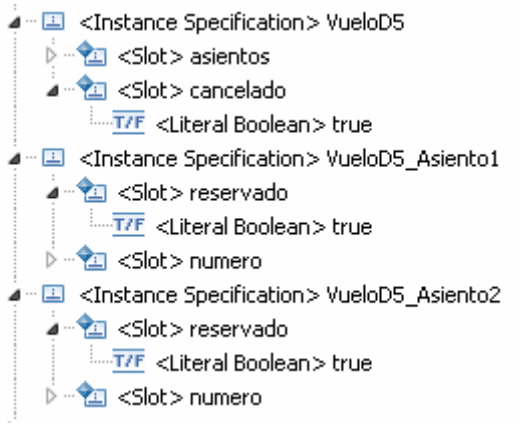


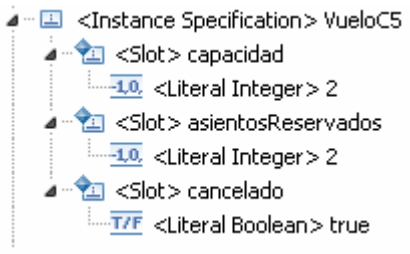

<Instance Specification> VueloD3
<Slot> asientos
<Slot> cancelado
<Literal Boolean> true
<Instance Specification> VueloD3_Asiento1
<Slot> reservado
<Literal Boolean> false
<Slot> numero
<Instance Specification> VueloD3_Asiento2
<Slot> reservado
<Literal Boolean> true
<Slot> numero
<Instance Specification> VueloD3_Asiento3
<Slot> reservado
<Literal Boolean> false
<Slot> numero
<Instance Specification> VueloD3_Asiento4
<Slot> reservado
<Literal Boolean> false
<Slot> numero
<Instance Specification> VueloD3_Asiento5
<Slot> reservado
<Literal Boolean> false
<Slot> numero

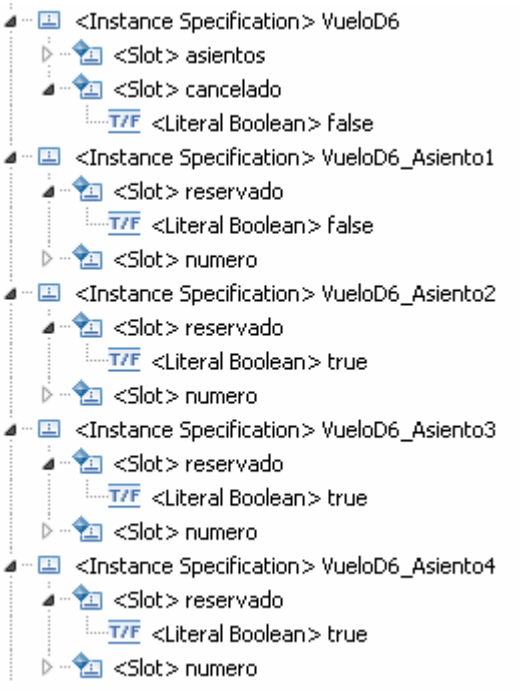

<Instance Specification> VueloC3
<Slot> capacidad
<Literal Integer> 5
<Slot> asientosReservados
<Literal Integer> 1
<Slot> cancelado
<Literal Boolean> true

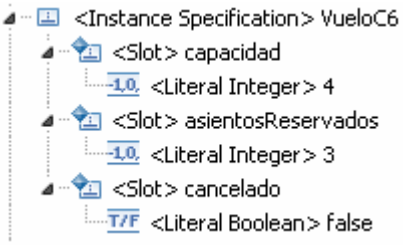

<Instance Specification> VueloD4
<Slot> asientos
<Slot> cancelado
<Literal Boolean> false
<Instance Specification> VueloD4_Asiento1
<Slot> reservado
<Literal Boolean> false
<Slot> numero
<Instance Specification> VueloD4_Asiento2
<Slot> reservado
<Literal Boolean> false
<Slot> numero
<Instance Specification> VueloD4_Asiento3
<Slot> reservado
<Literal Boolean> false
<Slot> numero
<Instance Specification> VueloD4_Asiento4
<Slot> reservado
<Literal Boolean> true
<Slot> numero
<Instance Specification> VueloD4_Asiento5
<Slot> reservado
<Literal Boolean> false
<Slot> numero


<Instance Specification> VueloC4
<Slot> capacidad
<Literal Integer> 5
<Slot> asientosReservados
<Literal Integer> 1
<Slot> cancelado
<Literal Boolean> false









Al expandir las instancias también verificamos que existe una instancia para cada posible combinación de rangos.

9.5.3. Micromundo para OneRangeCombination y con límite de instancias

Si seleccionamos la estrategia OneRangeCombination y establecemos un límite de 2 instancias a generar, el resultado es el de la figura 55.

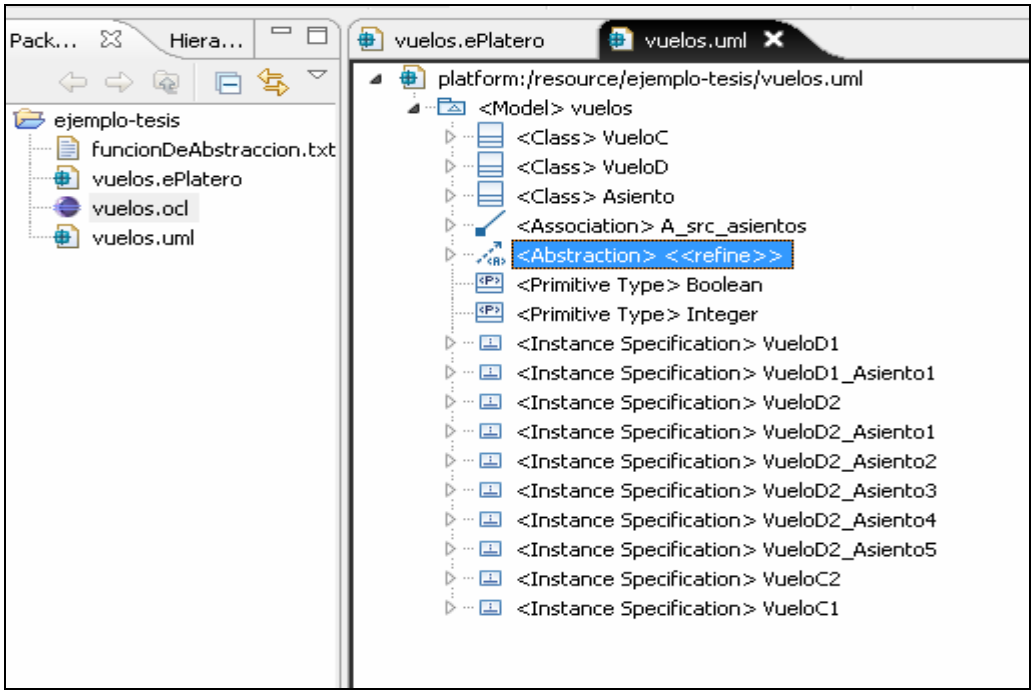
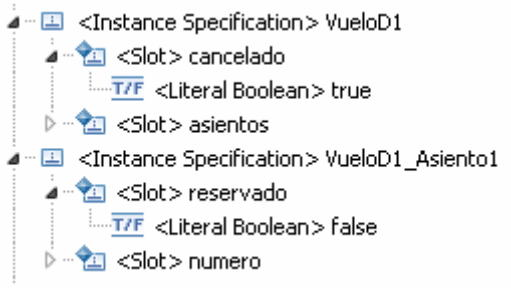
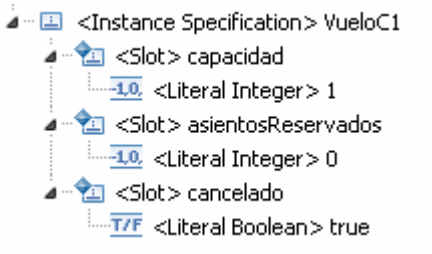
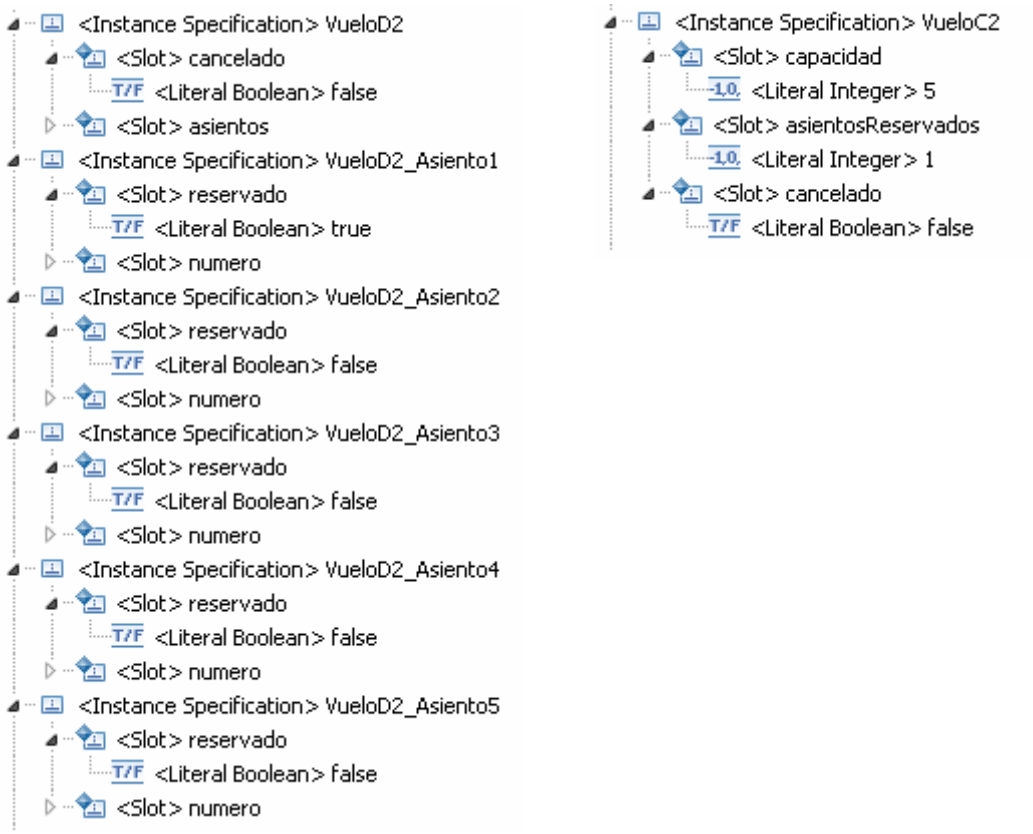


Figura 55 - Micromundo generado por ePlatero para la estrategia OneRangeCombination y con límite de 2 instancias

Si expandimos cada una de las instancias podemos verificar que se cumple la propiedad de dualidad.







Al limitar la cantidad de instancias a generar de la definición concreta, el micromundo resultante es menos representativo. Por ejemplo, en las instancias expandidas anteriormente podemos ver que no existe ningún vuelo que tenga un entre 2 y 4 asientos.

9.5.4. Micromundo para AllRangesCombination y con límite de instancias

Y, por último, si seleccionamos la estrategia AllRangesCombination y establecemos un límite de 3 instancias a generar, el resultado es el de la figura 56.

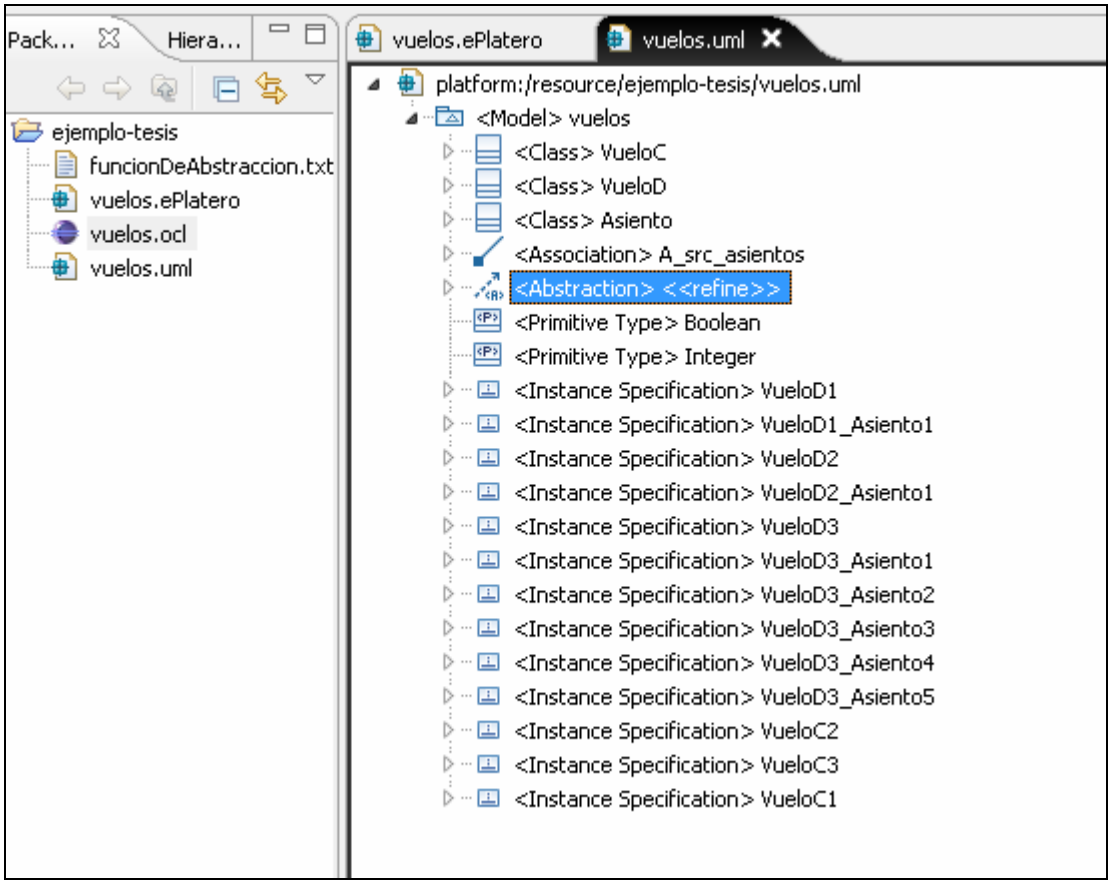
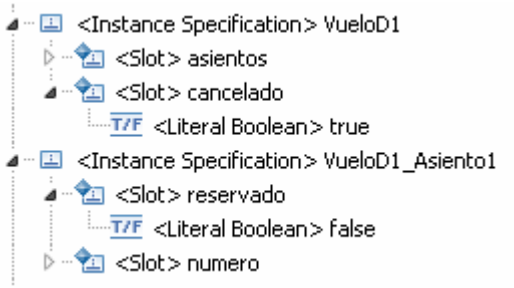
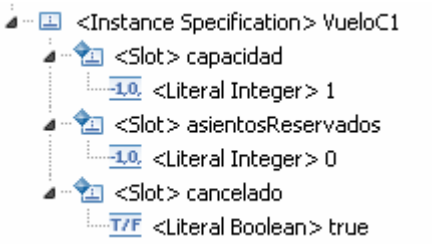
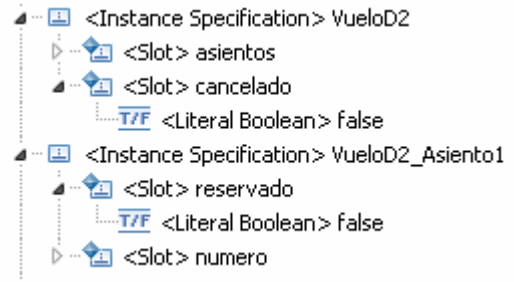


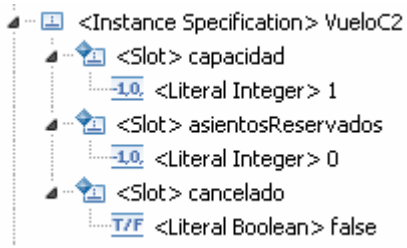
Figura 56 - Micromundo generado por ePlatero para la estrategia AllRangesCombination y con límite de 3 instancias

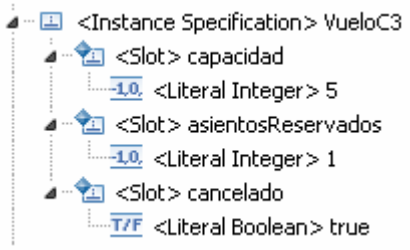
Si expandimos cada una de las instancias podemos verificar que se cumple la propiedad de dualidad.











En este caso, al limitar la cantidad de instancias no existe ningún vuelo que tenga entre 2 y 4 asientos, ni tampoco un vuelo con 5 asientos que no esté cancelado.

9.5.5. Conclusiones

De los cuatro micromundos generados anteriormente podemos concluir que si limitamos la cantidad de instancias a generar, el micromundo generado es más pequeño pero también menos representativo. El hecho de limitar la cantidad de instancias a generar de la definición concreta influye en la estrategia de combinación de rangos elegida, ya que es probable que no se satisfaga esta estrategia; es decir, si seleccionamos OneRangeCombination es probable que algún rango no aparezca en ninguna instancia, y si seleccionamos AllRangesCombination es probable que alguna combinación de rangos no esté representada por ninguna instancia.

En nuestro caso de estudio no tiene sentido limitar la cantidad de instancias debido a que los objetos generados son muy pocos. En diagramas de clases más complejos o con multiplicidades de asociaciones con limites superiores más grandes, limitar la cantidad de instancias generará micromundos menos representativos, pero lo tiempos de procesamiento disminuirán notablemente.

Por lo tanto, a la hora de restringir el número de instancias generadas debemos tener en cuenta la complejidad del modelo de clases y/o los límites superiores de las asociaciones presentes en el diagrama.

9.6. Evaluación del refinamiento

Una vez generado el micromundo podemos evaluar el refinamiento. Como vimos en la sección 6.1 esta evaluación implica verificar que se cumplan tres condiciones:

- ✓ Inicialización
- ✓ Aplicabilidad
- ✓ Correctitud

Si alguna de estas tres condiciones no se cumple, entonces el refinamiento especificado entre la definición abstracta y la concreta es incorrecto.

En ePlatero se debe seleccionar la abstracción que representa el refinamiento y se debe evaluar el mismo. Esto se muestra en la figura 57.

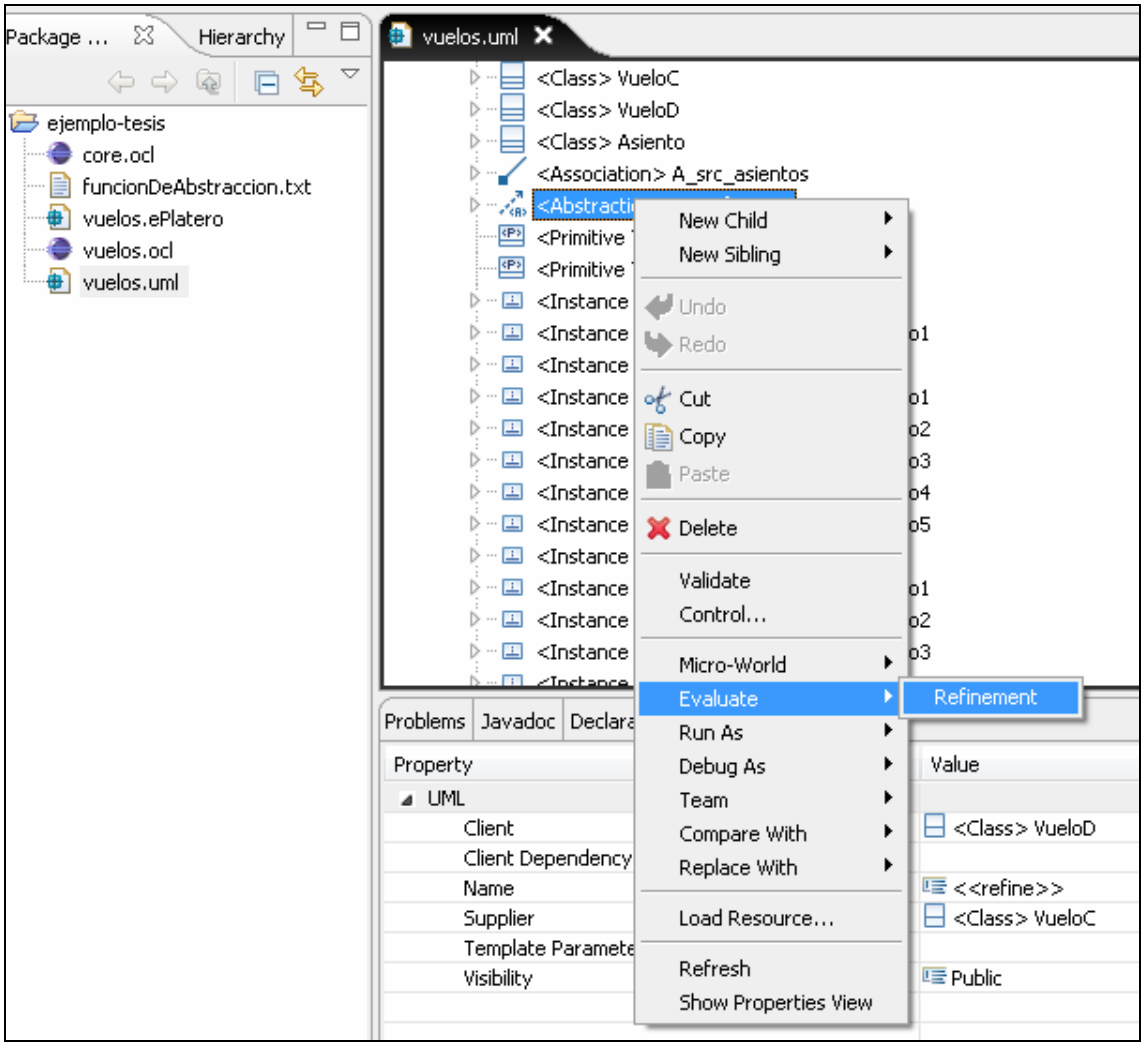


Figura 57 - Evaluación del refinamiento en ePlatero

Como resultado de la evaluación del refinamiento obtenemos:

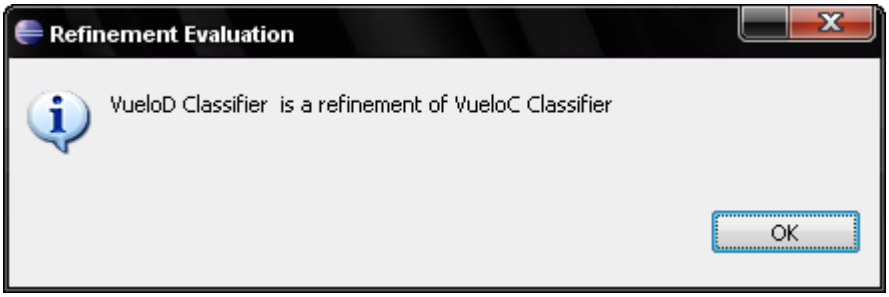


Figura 58 - Resultado de la evaluación del refinamiento

Para explorar un caso donde las condiciones de refinamiento no se satisfacen, se modifica la siguiente precondición:

```
context VueloD::reservar()
  pre:  self.asientos -> exists( q | not q.reservado )
        and self.cancelado
```

En la figura 59 se ilustra el resultado de evaluar la condición de refinamiento. El evaluador informa la primera condición que no se satisface.

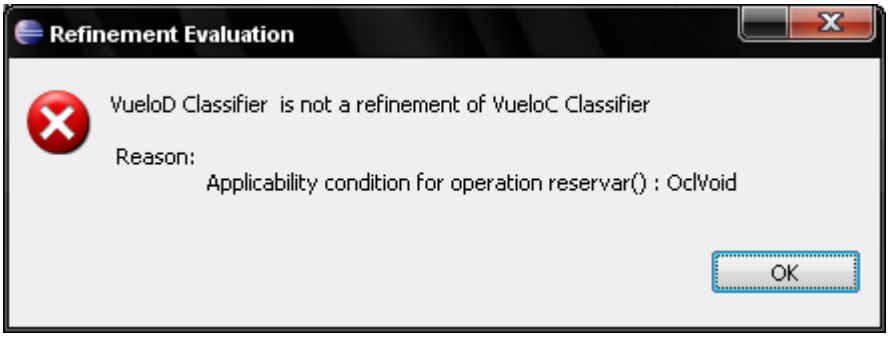


Figura 59 - Resultado de la evaluación del refinamiento modificando alguna precondición

Si en lugar de modificar la precondición de cancelar, modificamos su postcondición:

```
context VueloD::cancelar()
  pre:  not self.cancelado
  post: self.cancelado
```

el resultado de evaluar la condición de refinamiento es el siguiente:

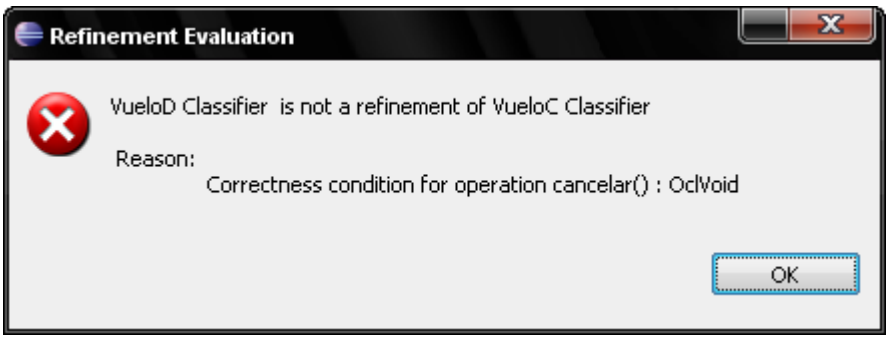


Figura 60 - Resultado de la evaluación del refinamiento modificando alguna postcondición

9.7. Comparación con Alloy Analyser

Alloy Analyzer es una herramienta desarrollada por el Software Design Group para analizar modelos escritos en Alloy, un lenguaje declarativo de lógica de primer orden que está basado en el lenguaje formal Z y que permite describir precisamente modelos de sistemas.

Alloy Analyser proporciona dos funcionalidades principales, simulación y verificación de aserciones. Las aserciones son enunciados que queremos comprobar si son o no propiedades del sistema especificado; cuando una aserción no se cumple el analizador de Alloy genera un contraejemplo.

La simulación produce una instancia del modelo que conforma la especificación. La verificación de aserciones determina si la restricción se cumple en el sistema especificado. Para ello se extrae del modelo UML un número relativamente pequeño de instanciaciones, y se verifica si satisfacen las restricciones especificadas.

En esta sección se presentan los resultados obtenidos de la evaluación de las condiciones de refinamiento empleando las herramientas ePlatero y Alloy Analyser. Los aspectos que nos parece importante señalar son la exactitud y la complejidad computacional, con el fin de verificar las condiciones de refinamiento expresadas en OCL y Alloy respectivamente. Ambas herramientas garantizan que si el refinamiento es correcto, es decir si se cumplen las condiciones de refinamiento, la respuesta es positiva (no se producen negativos falsos).

Las tablas de las figuras 61 y 62 muestran una comparación respecto al tiempo de respuesta promedio generando un micromundo con un número de instancias limitado y evaluando las condiciones de refinamiento. El límite establecido fue de 5 y 10 instancias, ya que para este caso de estudio existen 6 instancias representativas de la definición concreta con lo cual no tiene sentido establecer un límite más grande.

Se evaluaron las condiciones de refinamiento 50 veces en un AMD Athlon 3000 de 1Gb de memoria RAM.

Herramienta	**Tiempo de respuesta**	
ePlatero	OneRangeCombination: 110 mseg.	AllRangesCombination: 235 mseg.
Alloy Analyser	407 mseg.	

Figura 61 - Comparación de tiempos promedio con un límite de 5 instancias

Herramienta	**Tiempo de respuesta**	
ePlatero	OneRangeCombination: 94 mseg.	AllRangesCombination: 359 mseg.
Alloy Analyser	2657 mseg.	

Figura 62 - Comparación de tiempos promedio con un límite de 10 instancias

Los tiempos en ePlatero para la estrategia OneRangeCombination, si establecemos un límite de 5 o 10 instancias, son bastante cercanos. Esto se debe a que, en ambos casos, sólo se van a generar 3 instancias representativas, por cómo está definida la estrategia.

Para explorar un caso donde las condiciones de refinamiento no se satisfacen, se introdujeron errores en el modelo de la figura 46. Evaluamos las condiciones 50 veces y

observamos que ambas herramientas producen algunas respuestas incorrectas (positivos falsos). La siguiente tabla muestra la comparación entre ePlatero y Alloy Analyser con respecto al porcentaje de respuestas correctas según el tamaño de los micromundos.

Herramienta	**Respuestas correctas**	
ePlatero	OneRangeCombination: 60 %	AllRangesCombination: 80 %
Alloy Analyser	33 %	

Figura 63 - Comparación de respuestas correctas con micromundos de tamaño 5

Herramienta	**Respuestas correctas**	
ePlatero	OneRangeCombination: 66 %	AllRangesCombination: 93 %
Alloy Analyser	66 %	

Figura 64 - Comparación de respuestas correctas con micromundos de tamaño 10

A partir de los resultados obtenidos llegamos a la conclusión que ePlatero es más rápido y produce menor cantidad de respuestas incorrectas. Además, cabe mencionar que no hemos considerado el tiempo requerido para transformar el modelo expresado en UML/OCL en Alloy.

Más allá de estos resultados, no descartamos la posibilidad de que la diferencia de tiempos pueda cambiar con modelos más complejos o cuando se pongan a prueba refinamientos que respondan a patrones que surjan de futuras investigaciones.

10. Conclusiones

Debemos tener siempre presente el objetivo central del desarrollo de sistemas: lo que esperan los clientes que solicitan una solución informática es llegar a contar con un sistema confiable y amigable. El modelado es sólo un medio para alcanzar esta meta, y no un fin en sí mismo.

Los lenguajes gráficos de modelado, como UML, son ampliamente aceptados en la industria, sin embargo su falta de precisión ha originado la necesidad de utilizar otros lenguajes de especificación, como OCL, para definir restricciones adicionales. Con el uso de OCL se ofrece al diseñador la posibilidad de crear modelos precisos y completos del sistema en etapas tempranas del desarrollo. Sin embargo para estimular su uso en la industria es necesario contar con herramientas que permitan la edición y evaluación de las especificaciones expresadas en OCL.

Una característica que es muy importante para la utilización de los lenguajes de modelado, en sistemas de software complejos, es contar con técnicas de refinamientos, permitiendo un desarrollo por etapas con distintos niveles de abstracción y postergando los detalles del problema en etapas posteriores. Esto es la esencia del paradigma de desarrollo de software conocido como MDE.

Escribir complejas transformaciones de modelos es propenso a errores, y son requeridas técnicas eficientes de testeo para cualquier desarrollo de programa complejo. El testeo de una trasformación de modelos es típicamente realizado chequeando el resultado de la transformación aplicada a un conjunto de modelos de entrada. Mientras es bastante fácil proveer algunos modelos de entrada, es difícil calificar la relevancia de esos modelos para testear.

Teniendo en mente la hipótesis de Jackson [**14**], que explica que las respuestas negativas tienden a suceder en mundos pequeños, se aplica para la evaluación de las condiciones de refinamiento la técnica de micromodelos de software definiendo un límite finito de instancias (micromundo).

En este trabajo se han estudiado diferentes técnicas para la generación de micromundos, de manera de enriquecer la técnica de micromodelos de Jackson, para generar micromundos relativamente pequeños pero formados por instancias con valores que tenga sentido testear.

En [**13**] se expone el criterio de multiplicidad de asociaciones (AEM), criterio de generalización (GN) y criterios de atributos de clase (CA), basados en la cobertura de los elementos de un modelo, los cuales hemos estudiado y adaptado en este trabajo para la construcción de micromundos.

Además se ha desarrollado un plugin para Eclipse, que forma parte de ePlatero, basado en todo lo estudiado acerca de los criterios de test y micromodelos. El plugin permite generar micromundos con sentido para testear un refinamiento (cuando se quiere poner a prueba una transformación), o para un modelo de clases completo (sin refinamientos). Ha sido desarrollado implementando dos estrategias para la generación de

micromodelos, de manera de poder contrastar una con otra, quedando abierta la implementación para agregar nuevas estrategias para la generación de micromundos.

10.1. Futuros trabajos

Dentro de los trabajos que consideramos que pueden enriquecer la implementación del plugin para la generación de micromundos, como a toda la herramienta ePlatero se encuentran:

- Estudiar e implementar nuevas estrategias para la generación de particiones.
- Estudiar e implementar nuevas estrategias para la combinación de rangos de particiones. Como se detallo antes, ePlatero ahora cuenta con dos estrategias para tal fin quedando abierta la implementación para soportar estrategias que puedan surgir de investigaciones futuras.
- Extender el método para soportar el refinamiento de operaciones con parámetros.
- Reemplazar la codificación de la función de abstracción por la utilización de un convertor de OCL a Java, ya que en realidad el código java de la función de abstracción se corresponde con el mapping del refinamiento (escrito en OCL). Esto ayudará a hacer más dinámica la construcción de micromundos.

11. Referencias

[1] UML 2.0. The Unified Modeling Language Superstructure version 2.0 – OMG final Adopted Specification.. http://www.omg.org/. August 2003

[2] OCL 2.0. OMG Final Adopted Specification. October 2003.

[3] OMG Model Driver Architecture - http://www.omg.org/mda/

[4] Kleppe A., J. Warmer, and W. Bast, MDA Explained. 2003, Addison-Wesley.

[5] QVT. OMG Specification http://www.omg.org/

[6] The Booch Method. http://salmosa.kaist.ac.kr/BoochReferenz/index.html

[7] Metodologías para generación de Sistemas Orientados a Objetos. Análisis y Diseño (Tecnologías) Orientado a Objetos. Dr. Leopoldo Altamirano Robles. Septiembre, 2003. Alicia Morales Reyes, Alma Rosa Rugerio Ramos.

[8] ePlatero.http://sol.info.unlp.edu.ar/eclipse

[9] PAMPERO: Precise Assistant for the Modeling Process in an Environment with Refinement Orientation. C. Pons, R.Giandini, G. Pérez, P. Pesce, V.Becker, J. Longinotti, J.Cengia. In "UML Modeling Languages and Applications: UML 2004 Satellite Activities, Revised Selected Papers" . Lecture Notes in Computer Science number 3297. -- New York :Springer-Verlag. Portugal, October 11-15, 2004 . ISBN: 3-540-25081-6

[10] Implementación de técnicas de evaluación y refinamiento para OCL 2.0 sobre múltiples lenguajes basados en MOF. Diego Gracia, Claudia Pons

[11] Towards Dependable Model Transformations: Qualifying Input Test Data. Franck Fleurey, Benoit Baudry and Pierre-Alain Muller, Yves Le Traon *France Télécom R&D.*

[12] T.J. Ostrand and M.J. Balcer. The category partition method for specifying and generating functional tests. Communications of the ACM, 1988. 31(6): 676 - 686.

[13] A. Andrews, R. France, S. Ghosh, and G. Craig. Test adequacy criteria for UML design models. Software Testing, Verification and Reliability, 2003. 13(2): 95 -127.

[14] Jackson, Daniel, Shlyakhter, I. and Sridharan. A micromodularity Mechanism. In proceedings of the ACM Sigsoft Conference on the Foundation of Software Engineering FSE'01. (2001).

[15] Gogolla , Martin, Bohling, Jo¨rn and Richters, Mark. Validation of UML and OCL Models by Automatic Snapshot Generation. In G. Booch, P.Stevens, and J. Whittle, editors, Proc. 6th Int. Conf. Unified Modeling Language (UML'2003). Springer, Berlin, LNCS 2863, (2003).

[16] Derrick, J. and Boiten,E. Refinement in Z and Object-Z. Foundation and Advanced Applications. FACIT, Springer. (2001)

[17] On the definition of UML refinement patterns. Claudia Pons. Workshop MoDeVa at ACM/IEEE 8th Int. Conference on Model Driven Engineering Languages and Systems (MoDELS) Jamaica. October 2005.

[18] Heuristics on the Definition of UML Refinement Patterns. Claudia Pons. Publication: Lecture Notes in Computer Science ISSN 0302-9743. volume 3831, J. Wiedermann et al. (Eds.).© Springer-Verlag Berlin Heidelberg 2006.

[19] Practical Verification Strategy for Refinement Conditions in UML Models. Claudia Pons and Diego Garcia. IFIP International Federation for Information Processing series. ISSN: 1571-5736. Volume 219. Special issue on Advanced Software Engineering: Expanding the Frontiers of Software Technology, (Boston:Springer), pp. 47-61. (2006).

[20] Lano, Kevin, Androutsopolous, Kelly and Clark David. Refinement Patterns for UML. Proceedings of REFINE'2005. Elsevier Electronic Notes in Theoretical Computer Science 137. pages 131-149 (2005).

[21] - Eclipse -http://www.eclipse.org.

[22] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design patterns elements of reusable object-oriented software, Addison-Wesley Pblishing Company, 1995.

[23] Akehurst David and Patrascoiu Octavian. OCL 2.0 – Implementing the Standard for Multiple Metamodels. University of Kent at Canterbury . Published by the Computing Laboratory. 2003.

[24] The Alloy Analyser. http://alloy.mit.edu/

[25] Sun Microsystems (2000) Javacc – the java parser generator- http://javacc.dev.java.net/.

[26] Demuth, Birgit and Heinrich Hussmann, and Ansgar Konermann. Generation of an OCL 2.0 Parser. Workshop co-located with MoDELS'05: ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Jamaica. October 4, 2005.

[27] Paige, R., Kolovos D. and Polack,F. Refinement via Consistency Checking in MDD. Electronic Notes in Theoretical Computer Science 137. pg. 151-161 (2005).

[28] Astesiano E., Reggio G. An Algebraic Proposal for Handling UML Consistency", Workshop on Consistency Problems in UML-based Software Development. UML Conference (2003).

12. Anexo I - Metamodelo de la sintaxis abstracta de OCL 2.0

La sintaxis abstracta representa los conceptos de OCL empleando MOF. Para representar el metamodelo de OCL se importan metaclases del metamodelo UML, las cuales son mostradas en el modelo con la anotación "(from <UML Package>)".

La sintaxis abstracta está dividida de la siguiente manera:

- El paquete Types: describe los conceptos que definen los tipos de OCL.
- El paquete Expressions: describe la estructura de las expresiones OCL.

La sección 12.1 describe el paquete types. En la sección 12.2 se describe el paquete expressions. En la sección 12.3 se muestra un ejemplo de instanciación del metamodelo de la sintaxis abstracta de OCL 2.0 a partir de una expresión textual y un modelo MOF.

12.1. Paquete types

OCL es un lenguaje tipado. Cada expresión tiene un tipo que se declara explícitamente o puede derivarse estáticamente. Antes de definir expresiones es necesario proveer un modelo para el concepto de tipo. En la Figura 65 se ilustra un extracto del paquete types.

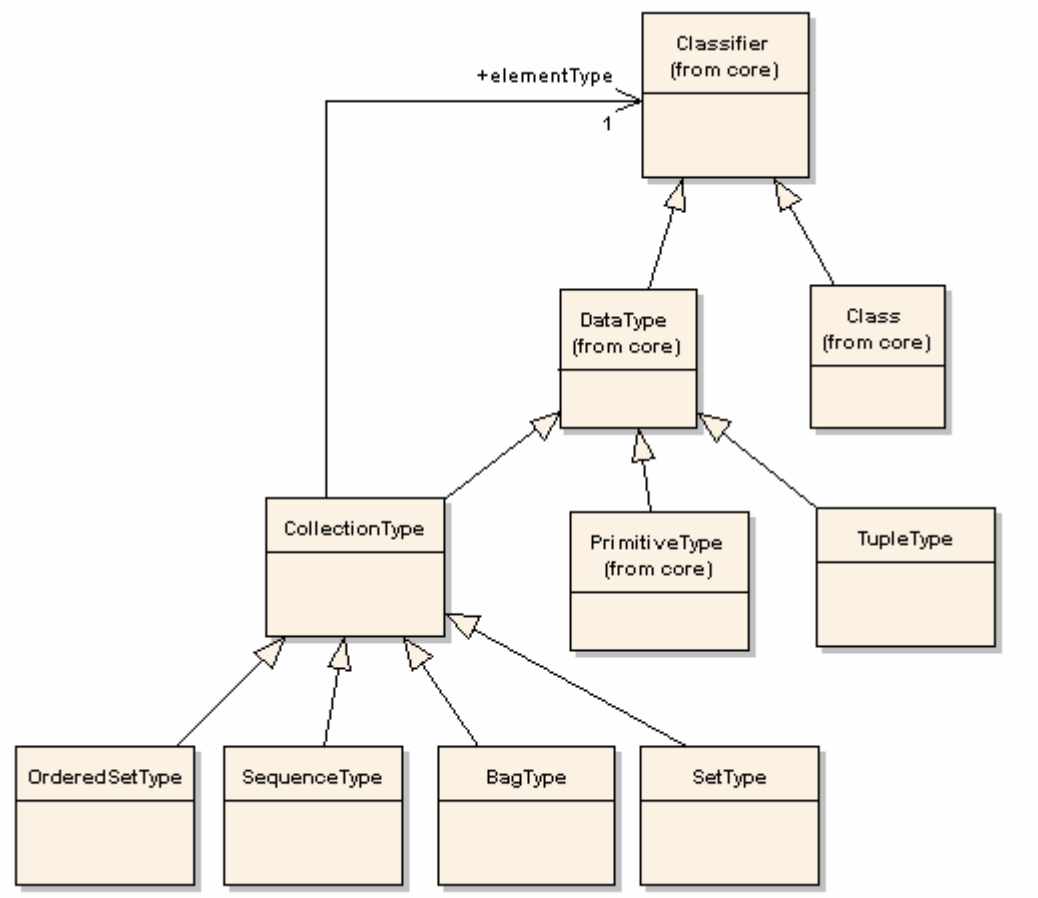


Figura 65 - Extracto del metamodelo de la sintaxis abstracta para los tipos de OCL

CollectionType

CollectionType describe una lista de elementos de un tipo en particular. CollectionType es una clase abstracta. Sus subclases son SetType, SequenceType, OrderedSetType y BagType. Las colecciones son parametrizadas con un tipo en particular. No hay ninguna restricción sobre los tipos de una colección, por ejemplo podemos tener una colección cuyo tipo es otra colección.

Asociaciones:

- elementType: es el tipo de los elementos de la colección. Todos los elementos que forman parte de la colección deben conformar a este tipo.

BagType

BagType es un tipo de colección que puede contener elementos duplicados. Estos no están ordenados.

OrderedSetType

OrderedSetType es un tipo de colección que describe un conjunto de elementos donde estos no están repetidos. Los elementos son ordenados por su posición en la secuencia.

SequenceType

SequenceType es un tipo de colección que describe una lista de elementos donde cada uno de estos puede estar repetido en la secuencia. Los elementos son ordenados por su posición en la secuencia.

SetType

SetType es un tipo de colección que describe un conjunto de elementos donde estos no están repetidos. Los elementos no están ordenados.

TupleType

TupleType (conocido como registro) contiene un conjunto de propiedades (atributos), las cuales tienen un nombre y un tipo. Cada componente es identificado por su nombre.

12.2. Paquete expressions

En esta sección se define la sintaxis abstracta del paquete de las expresiones**.** Este paquete define la estructura que pueden tener las expresiones OCL.

En la figura 66 se ilustra la estructura básica del paquete expressions.

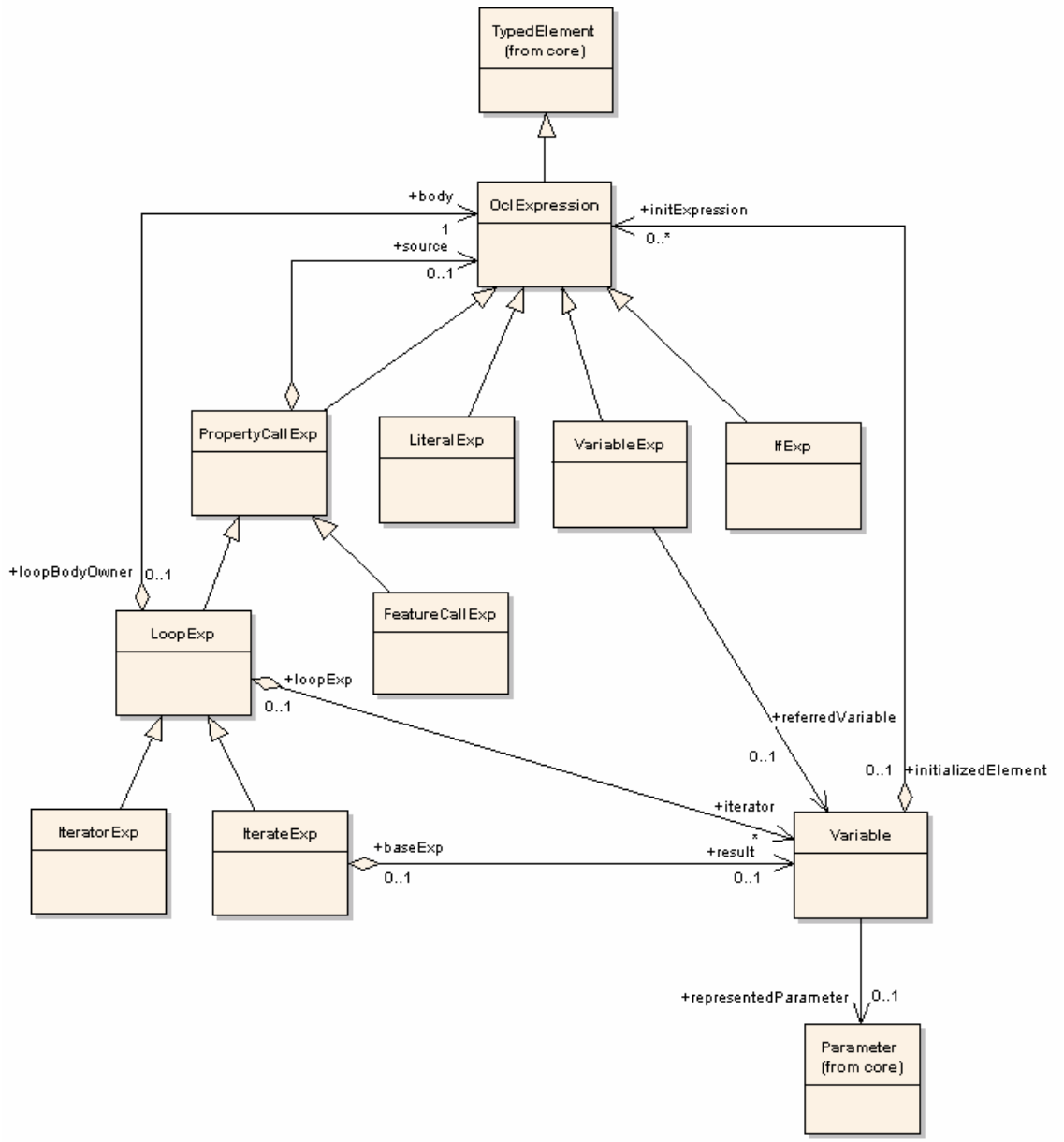


Figura 66 - Estructura básica del metamodelo de la sintaxis abstracta para expresiones

OclExpression

Una OclExpression es una expresión que puede ser evaluada en un ambiente dado. Esta metaclase es la superclase de todas las expresiones en el metamodelo. Toda expresión OCL tiene un tipo que se puede determinar estáticamente analizando la expresión y su contexto. La evaluación de la expresión retorna un valor. Las expresiones cuyo tipo es un boolean se pueden utilizar en las restricciones; en caso contrario, para operaciones query, valores iniciales de atributo, etc.

El ambiente (Environment) de una OclExpression define qué elementos del modelo son visibles y pueden ser referenciados en una expresión. El ambiente puede ser definido por el elemento del modelo que está ligado a la expresión OCL, por ejemplo un Classifier si la restricción es un invariante. Los iteradores de la expresión también pueden ser introducidos en el ambiente.

Variable

Las variables son elementos tipados para pasar datos en las expresiones. Esta metaclase representa entre otras las variables self y result.

Asociaciones:

- initExpression: expresión OCL que representa el valor inicial de la variable.
- representedParameter: parámetro de la operación actual que representa la variable. Cualquier acceso a esta representa un acceso al valor del parámetro.

VariableExp

La VariableExp es una expresión que consiste en una referencia a una variable.

Asociaciones:

- referredVariable: variable a la que ésta expresión se refiere.

LiteralExp

Un LiteralExp es una expresión sin argumentos que representa un valor. Algunas de sus subclases son: IntegerLiteralExp, BooleanLiteralExp, etc.

PropertyCallExp

Una PropertyCallExp es una expresión que se refiere a un Feature (operación, propiedad) o a un iterator predefinido para las colecciones. El resultado se obtiene a partir de la evaluación del feature correspondiente. Esta es un metaclase abstracta.

Asociaciones

- source: el receptor de la invocación de la propiedad.

FeatureCallExp

Un FeatureCallExp es una expresión que se refiere a un Feature definido en un Classifier del modelo. El resultado es la evaluación de la propiedad correspondiente.

La figura 67 muestra las subclases de FeatureCallExp las cuales no se agregaron en la figura 66 por cuestiones de legibilidad.

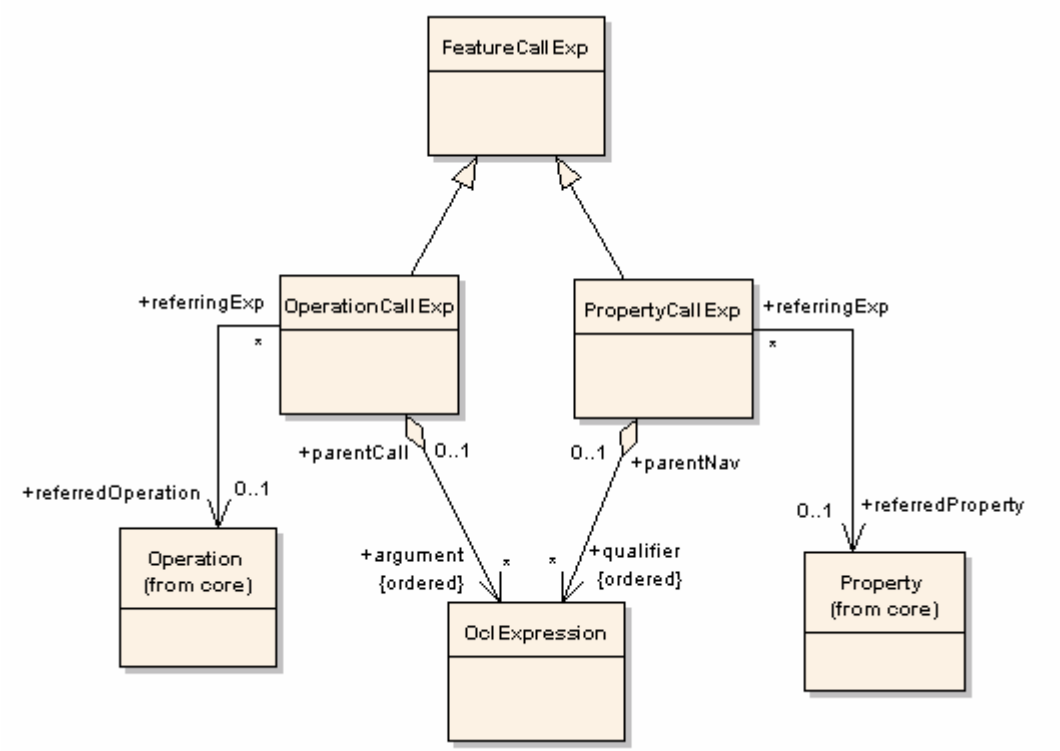


Figura 67 - Metamodelo de la sintaxis abstracta para FeatureCallExp

LoopExp

Un LoopExp es una expresión que representa un bucle sobre una colección. Tiene una variable que se utiliza para iterar sobre los elementos de dicha colección. La expresión body es evaluada para cada elemento contenido en la colección. El tipo del resultado de la expresión loop se indica en sus subclases.

Asociaciones:

- iterator: representa la variable que se utiliza para iterar sobre los elementos de la colección en el momento de la evaluación.
- body: es la expresión OCL que es evaluada para cada elemento de la colección receptora.

IterateExp

Una IterateExp es una expresión que evalúa la expresión indicada por la asociación body para cada elemento de la colección receptora. Cada uno de los valores resultantes de la evaluación forma parte de un nuevo valor de la variable result. El resultado puede ser de cualquier tipo y es definido por la asociación result.

Asociaciones:

- result: representa la variable resultante.

IteratorExp

Una IteratorExp es una expresión que evalúa la expresión indicada por la asociación body para cada elemento de la colección receptora. El resultado de la evaluación es un valor cuyo tipo depende del nombre de la expresión. En algunos casos puede ser del

mismo tipo que el receptor. La metaclase IteratorExp representa todas las operaciones predefinidas de las colecciones que se definen a través de la expresión iterator, por ejemplo select, exists, collect, forAll, etc.

IfExp

La expresión if está incompleta en el diagrama por cuestiones de legibilidad. Esta se encuentra más detallada en la figura 68.

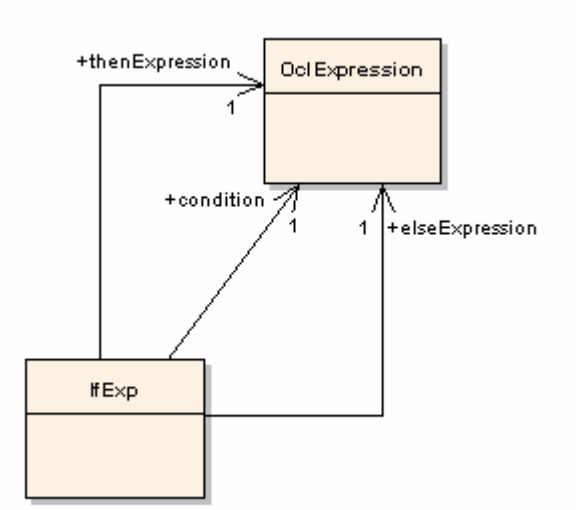


Figura 68 - Metamodelo para la expresión if

La expresión if ofrece dos alternativas a seguir, en base a la comprobación de la condición. Tanto el thenExpression y elseExpression son obligatorios ya que una expresión siempre debe resultar en un valor. El tipo de la expresión es el supertipo común a las dos expresiones alternativas.

Asociaciones:

- condition: el OclExpression que representa la condición. Si esta condición evalúa a true, el resultado de la expresión if es idéntico al resultado del thenExpression. En caso contrario el resultado de la expresión if es idéntico al resultado del elseExpression.
- thenExpression: la OclExpression que representa la parte del then de la expresión if.
- elseExpression: la OclExpression que representa la parte del else de la expresión if.

12.2.1. Metaclase ExpressionInOcl

Debido a que la metaclase OclExpression está definida recursivamente necesitamos una metaclase que represente el nivel superior del árbol de la sintaxis abstracta. Esta metaclase es llamada ExpressionInOcl, y está definida como una subclase de la metaclase OpaqueExpression del paquete core de UML.

En la figura 69 se representa la metaclase ExpressionInOcl.

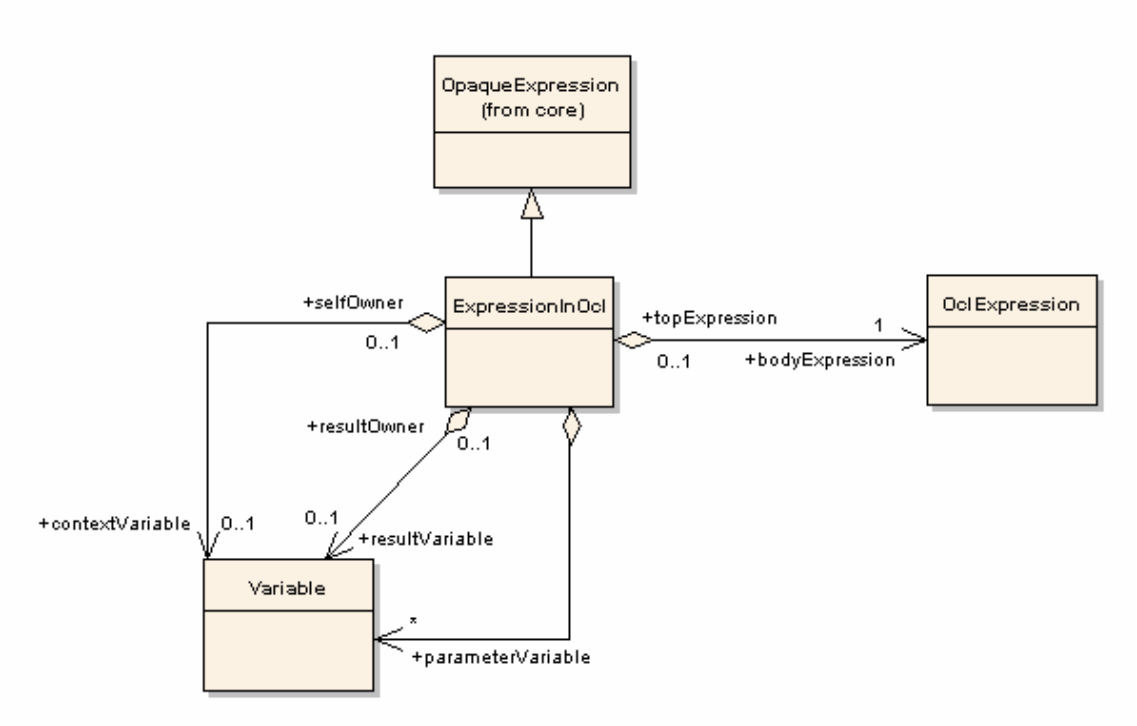


Figura 69 - Metamodelo de la sintaxis abstracta para ExpressionInOcl

Asociaciones:

- bodyExpression: es la raíz de la expresión OCL.
- contextVariable: es la variable contextual utilizada en el bodyExpression correspondiente.
- resultVariable: representa el valor a ser retornado por la operación.
- parameterVariable: las variables que representan los parámetros de la operación actual.

12.3. Ejemplo de instanciación del metamodelo de OCL

En esta sección se describe un ejemplo de instanciación del metamodelo de OCL 2.0 a partir de una expresión textual conforme a la gramática de OCL 2.0 [**2**]. A continuación se enriquece al modelo UML presentado en la sección 4 figura 3 con un invariante.

context Vuelo
inv miRegla: self.fechaSalida <= self.fechaLlegada

En la figura 70 se ilustra un diagrama de objetos que representa la instanciación del metamodelo de OCL para la regla anteriormente especificada.

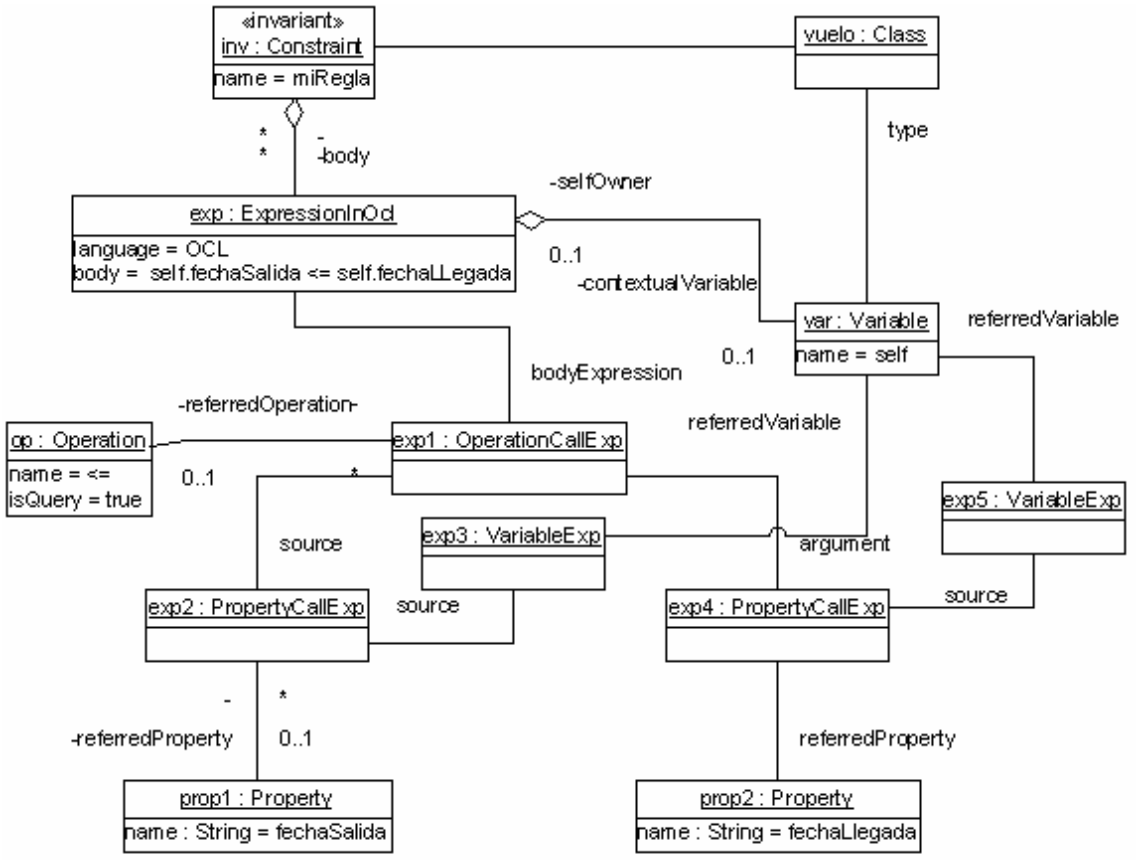


Figura 70 - Instanciación del metamodelo de la Sintaxis abstracta de OCL

La instancia inv representa al invariante definido en el contexto de vuelo identificado con el nombre de miRegla. El cuerpo de la restricción es una expresión en OCL especificada por la instancia exp, la cual conoce la variable contextual y la expresión OCL. Esta tiene un receptor (source) y un argumento representados por las invocaciones de las propiedades fechaSalida y fechaLlegada respectivamente. El tipo de las expresiones exp3 y exp5 es el tipo de la variable var, es decir la clase Vuelo. Las propiedades prop1 y prop2 definidas en dicha clase determinan el tipo de las expresiones exp2 y exp4 respectivamente. El tipo de la exp2 define una operación con el nombre <=, y el tipo de la exp4 es conforme con el tipo del parámetro de dicha operación.

13. Anexo II - Descripción de los módulos de ePlatero

ePlatero es una herramienta CASE educativa que soporta el proceso de desarrollo de software dirigido por modelos utilizando notación gráfica con fundamento formal. Esta herramienta está formada por varios módulos o componentes que interactúan unos con otros cumpliendo una labor específica. En este capítulo se describen brevemente los componentes que, junto con el componente dedicado a la generación de micromundos, forman parte de ePlatero.

En la sección 13.1 se describe el analizador léxico y sintáctico; en la sección 13.2 el editor de fórmulas OCL; en la sección 13.3 el evaluador de OCL, y por último en la sección 13.4 se describe el evaluador de las condiciones de refinamiento.

13.1. Analizador léxico y sintáctico

El analizador léxico recibe el contenido del archivo ocl, este es un archivo de texto con extensión ocl. En este proceso se agrupan los diferentes caracteres del flujo de entrada en tokens. Los tokens son los símbolos léxicos del lenguaje. Estos están identificados con símbolos y suelen contener información adicional (como el archivo en el que están, la línea donde comienzan, etc.). Una vez identificados, son transmitidos al analizador sintáctico.

Para comprender como funciona este analizador mostramos el siguiente ejemplo:

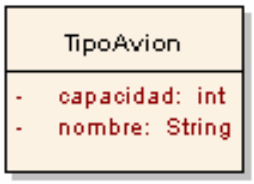


Figura 71 - Ejemplo analizador léxico y sintáctico

Dada la siguiente expresión OCL:

self.capacidad > 0

El analizador léxico de OCL produce la siguiente serie de tokens:

NAME DOT NAME GT INTEGER

En la fase de análisis sintáctico se aplican las reglas sintácticas del lenguaje analizado al flujo de tokens. En caso de no haberse detectado errores, el intérprete representará la información codificada en el código fuente en un Árbol de Sintaxis Concreta (CST), que es una representación arbórea de los diferentes patrones sintácticos que se han encontrado al realizar el análisis, salvo que los elementos innecesarios (signos de puntuación, paréntesis) son eliminados.

La especificación de OCL 2.0 [**2**] define la sintaxis abstracta (AS), concreta (CS) y la transformación de la CS a la AS. La sintaxis abstracta fue presentada en el anexo I. La sintaxis concreta permite a los modeladores escribir expresiones en forma textual.

Para desarrollar el parser se utilizó *Java Compiler Compiler (JavaCC)* [**25**] que es un generador de parser que se puede utilizar en las aplicaciones Java. La arquitectura del analizador sintáctico de ePlatero es similar a la presentada en [**26**]: el parser de OCL generado por javacc transforma el texto de entrada en el modelo de sintaxis abstracta (ASM). El ASM representa una instancia del metamodelo de OCL 2.0. El primer paso del proceso consiste en transformar el texto de entrada en el árbol de sintaxis concreta (CST), esta transformación es realizada automáticamente por el parser. La segunda parte del proceso consiste en transformar el CST en ASM, para llevar a cabo esta última transformación se aplicó el patrón de diseño Visitor [**22**] el cual recorre el árbol de sintaxis concreta y lo transforma en el modelo de sintaxis abstracta. Para poder realizar dicha transformación es necesario llevar a cabo un análisis sensible al contexto

Continuando con el ejemplo iniciado en la presentación del analizador léxico (figura 71) se ilustrará el proceso descrito en esta sección. En la figura siguiente se presenta el árbol de la sintaxis concreta generado por el OclParser.

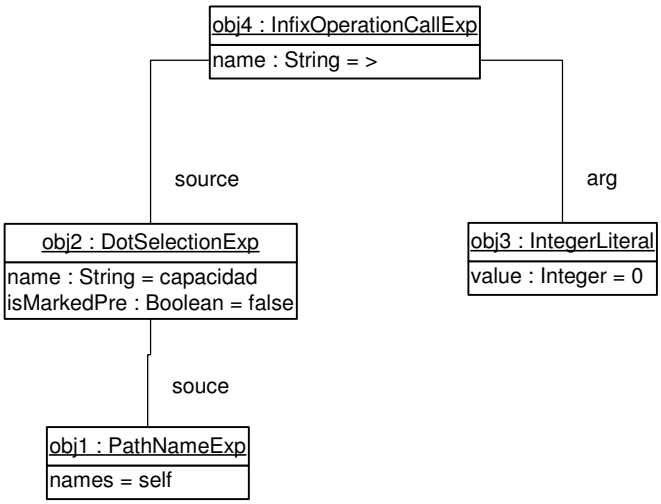


Figura 72 - Ejemplo de un árbol de sintaxis concreta

La siguiente transformación consiste en utilizar la información del contexto para poder instanciar correctamente el modelo de la sintaxis abstracta.

El objeto obj3 (IntegerLiteralCS) se traduce directamente a un IntegerLiteralAS (figura 73).

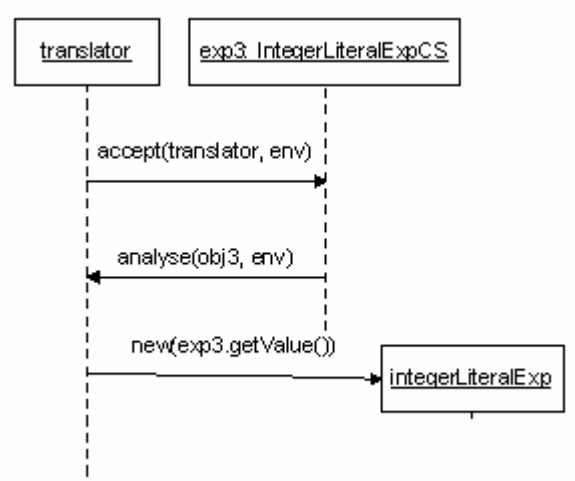


Figura 73 - Traducción de IntegerLiteralCS a IntegerLiteralAS

En este contexto el objeto obj1 (PathNameExp) se transforma en una VariableExp (figura 74).

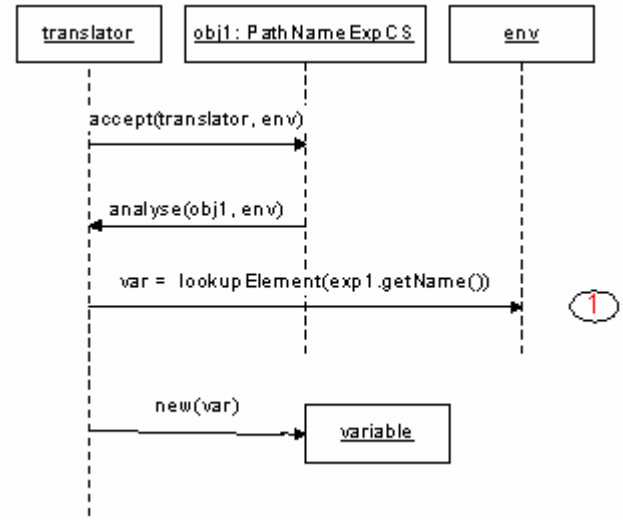


Figura 74 - Traducción de PathNameExpCS a VariableExpAS

Referencia 1 de la figura: una vez que se obtiene el elemento del ambiente se puede determinar la traducción del objeto obj1.

En este contexto el objeto obj1 (DotSelectionExp) se traduce como un PropertyCallExp (figura 75).

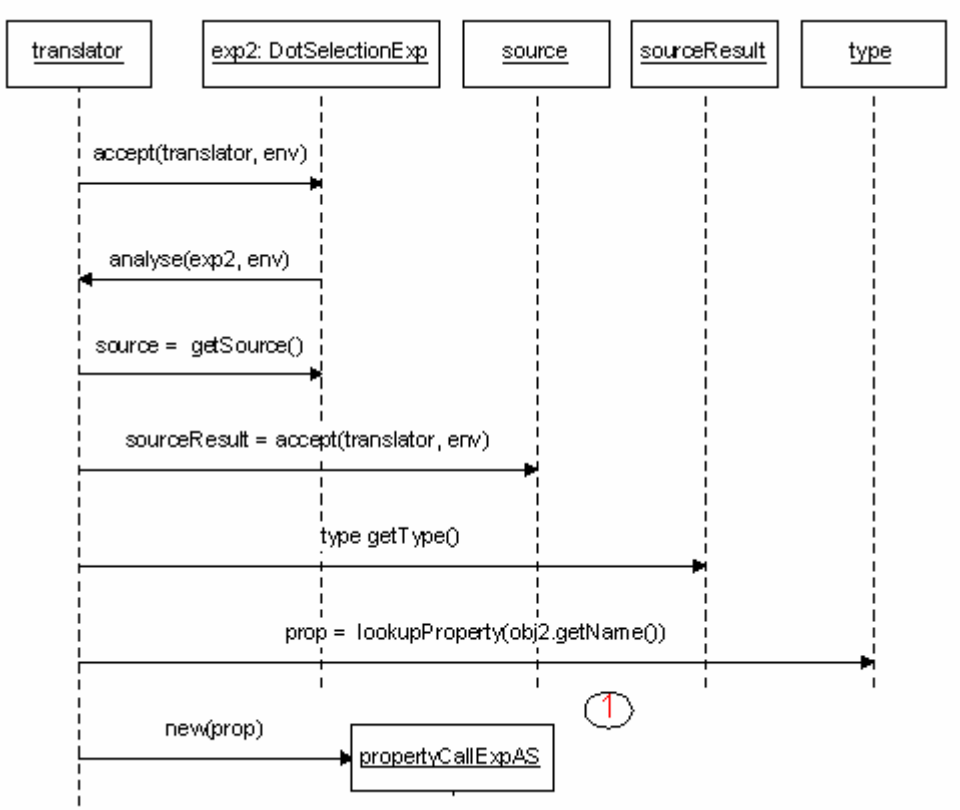


Figura 75 - Traducción de DotSelectionExpCS a PropertyCallExpAS

Referencias 1 de la figura: en base al tipo de la propiedad se determina la expresión resultante.

Por último se transforma el objeto obj4 (InfixOperationCallExp) en un OperationCallExp (figura 76).

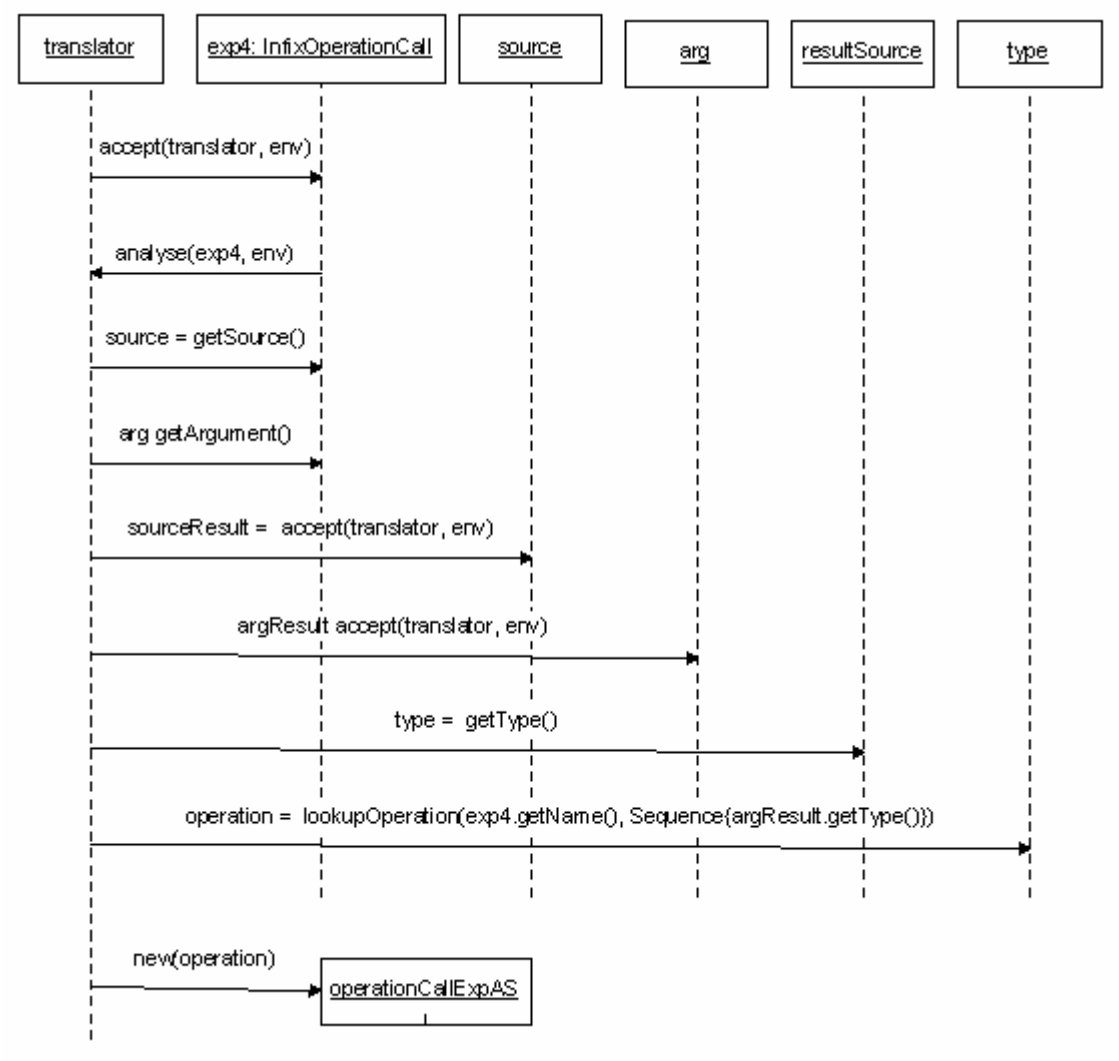


Figura 76 – Traducción de InfixOperationExpCS a OperationCallExpAS

13.2. Descripción del editor de fórmulas OCL

El editor de OCL (figura 77) se utiliza para editar los archivos OCL que contienen las reglas a evaluar sobre el modelo. Posee operaciones de edición, syntax highlighting, asistencia y corrección de errores. Posee un outline que despliega la estructura de los archivos OCL.

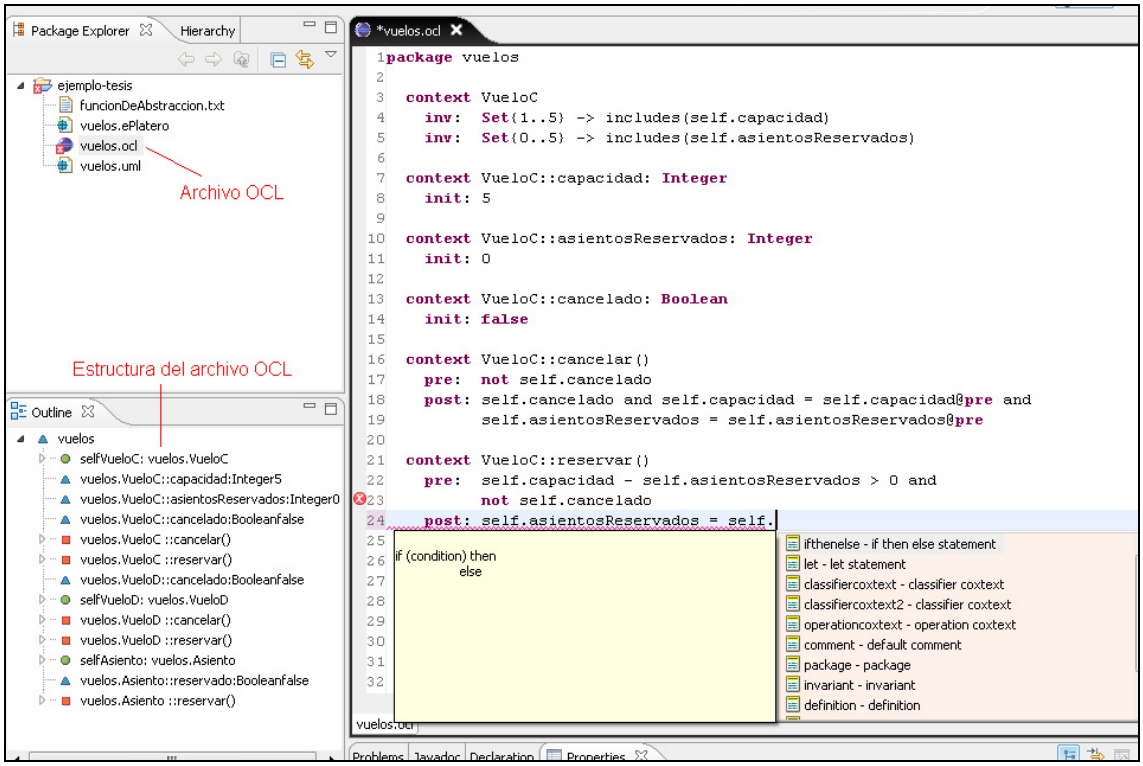


Figura 77 - Editor de fórmulas OCL

En Eclipse, un editor es una parte principal del Workbench y tiene su propio punto de extensión que permite implementar cualquier editor personalizado.

Para el desarrollo del editor de fórmulas OCL se utilizó el framework JFace Text proporcionado por eclipse. Dicho framework aporta un editor que permite la edición de documentos de texto independientemente del dominio.

En la figura 78 se ilustran las principales clases del framework Jface Text. La clase AbstractTextEditor es la clase base del editor de texto predefinido. Para implementar un editor de texto se debe subclasificar dicha clase. El AbstractTextEditor trabaja sobre el contenido del modelo del documento.

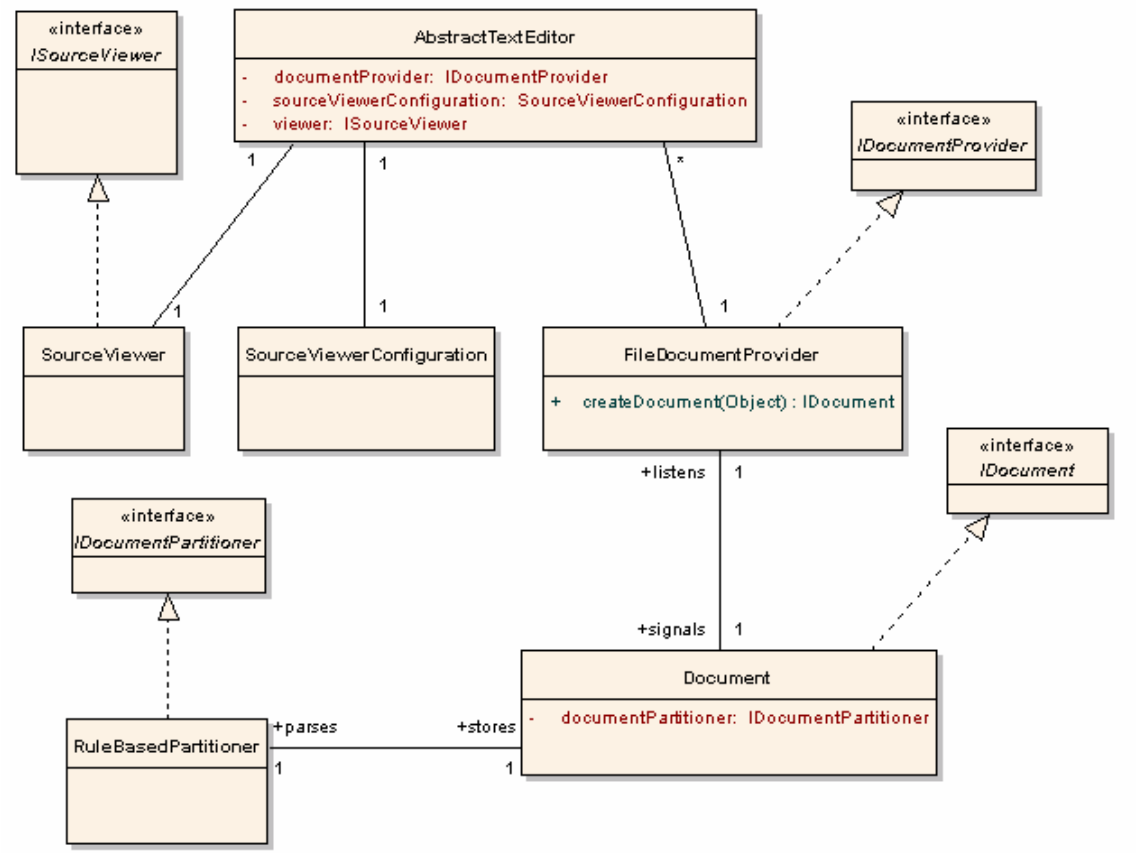


Figura 78 - Relaciones de la clase AbstractTextEditor

A continuación se describen las responsabilidades o roles de las clases e interfaces asociadas al AbstractTextEditor:

- **SourceViewerConfiguration:** se utiliza para describir qué características se añaden a la instancia del editor. Hay que subclasificar dicha clase y sobrescribir los métodos para habilitar el asistente, la estrategia de identación automática, syntax highlighting, etc.

- **SourceViewer**: provee un viewer de texto al editor y permite una configuración explícita.

- **IDocument:** es la representación del modelo de texto del documento, proporciona: manipulación de texto, manejo de la partición del documento, manejo de búsquedas, y notificación de cambio.

- **IDocumentProvider:** es la interfase del editor a los datos u objetos del modelo. De esta manera es posible tener editores concurrentes que abren el mismo documento. Crea y maneja el contenido del documento.

- **IDocumentPartitioner:** se emplea para dividir un documento en secciones para que el texto puede tratarse o manipularse de diferentes maneras dependiendo de la partición.

Relación Model-View-Controller

El document provider actúa como un intermediario entre el documento y el editor. Los editores con el mismo documento comparten el mismo document provider esto es lo que permite la actualización automática de los otros editores cuando uno cambia.

Sustituyendo el método changed es la manera que el provider propaga notificaciones de cambio a cada uno de los editores activos. Un document provider entrega una presentación textual (IDocument) del elemento de entrada del editor a la parte de vista.

13.2.1. Diseño del editor OCL

En la figura 79 se presenta el diseño básico del editor de fórmulas OCL, el cual se instancia por especialización del framework JfaceText.

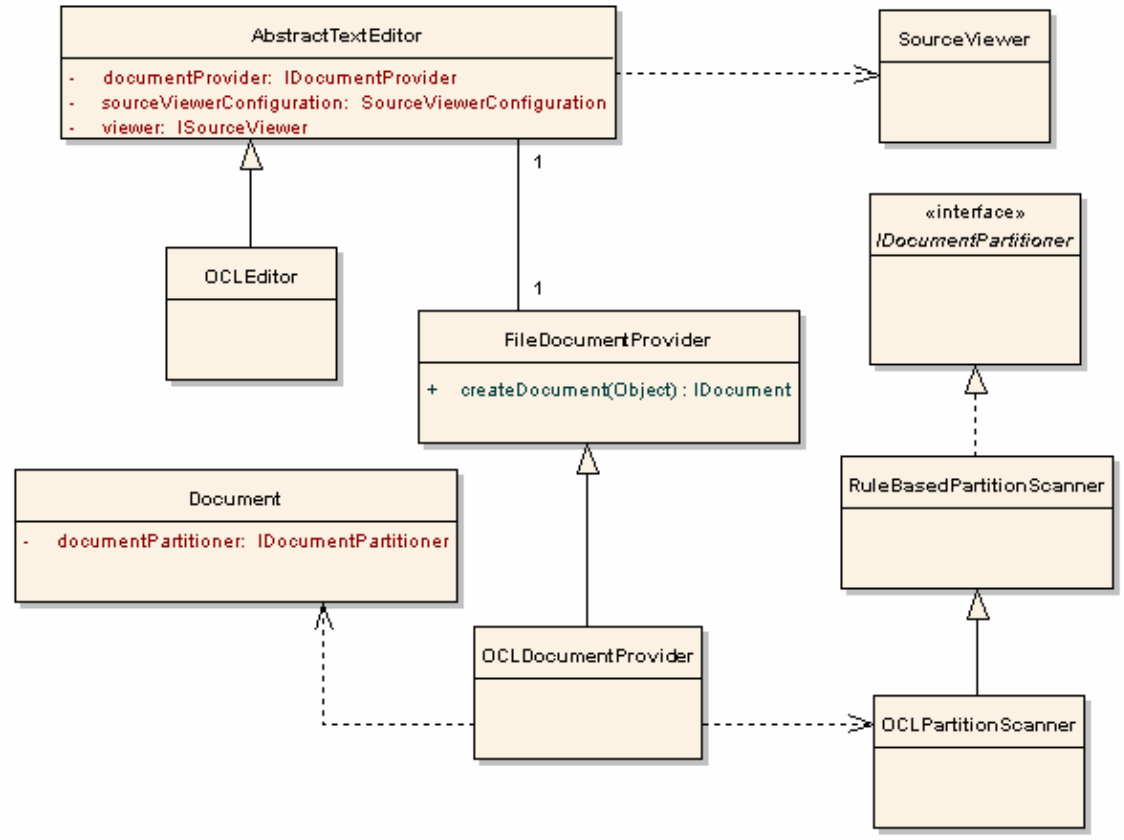


Figura 79 - Diseño básico del editor de fórmulas OCL

13.3. Descripción del evaluador OCL

El evaluador de OCL es el responsable de realizar el análisis semántico de las fórmulas OCL, parseadas anteriormente. El evaluador actual de ePlatero permite evaluar los invariantes de los tipos y metatipos, es decir evalúa los invariantes del modelo y metamodelo. En la figura 80 se ilustra como el editor de fórmulas OCL presenta los errores de evaluación. Los comentarios son utilizados para clarificar las restricciones

y/o para que el usuario pueda comprender rápidamente cuales son los errores semánticos.

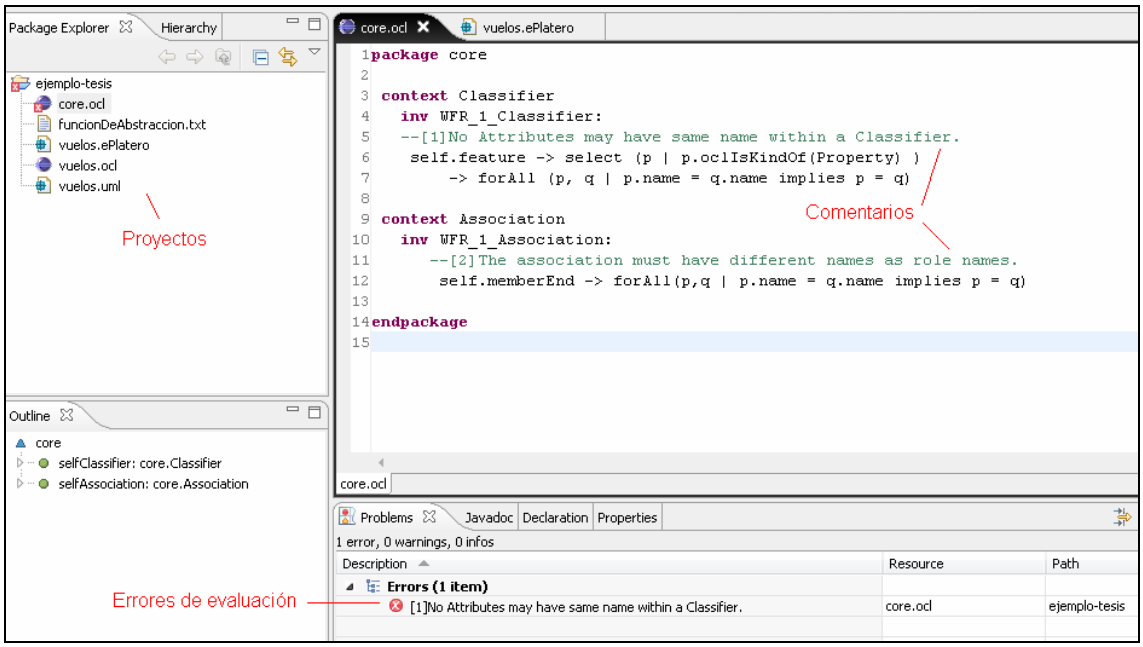


Figura 80 - Visualización de errores semánticos

13.3.1. Diseño del evaluador OCL

En la figura 81 se presenta el diagrama de paquetes del evaluador OCL. Los paquetes expressions y types son los paquetes del metamodelo de OCL 2.0 que contienen las expresiones y tipos de OCL respectivamente. El paquete core, contiene los elementos del modelo comunes a todo metamodelo MOF, y provee la información contextual para la evaluación de expresiones OCL. El paquete de basics contiene los tipos básicos de OCL con las operaciones primitivas que son invocadas por reflexión por el evaluador, y por último el paquete analyser contiene las clases responsables de la evaluación de las reglas semánticas. A continuación se describen las clases de los dos nuevos paquetes.

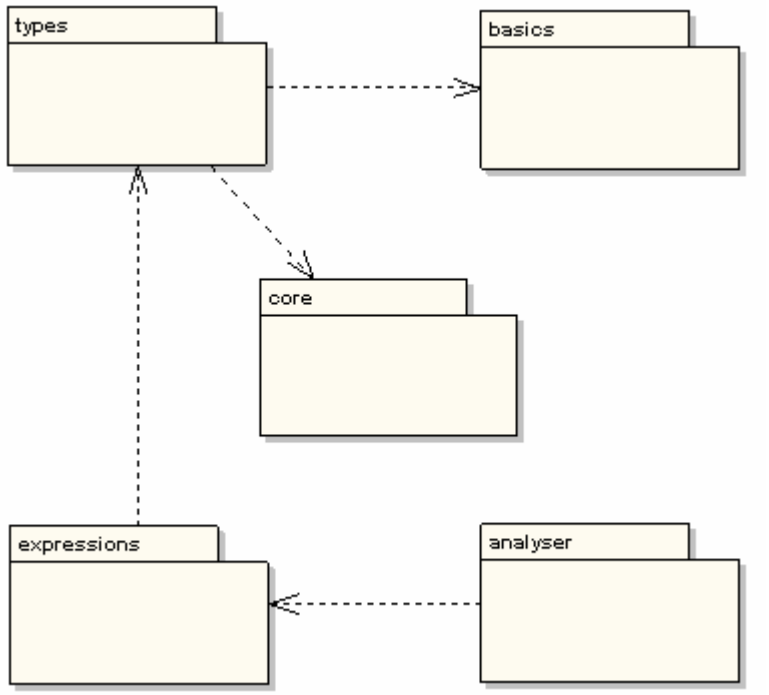


Figura 81 - Diagrama de paquetes del evaluador OCL

13.3.1.1. Estructura del paquete basics

Como se mencionó anteriormente el evaluador hace reflexión sobre las clases que forman parte de este paquete para ejecutar las operaciones primitivas de la librería estándar de OCL. La jerarquía de las colecciones está encabezada por la clase abstracta Collection y define las operaciones comunes a todas las colecciones. Las subclases Set, Bag y Sequence son las clases concretas y cada una define nuevas operaciones. Las operaciones de las colecciones definidas en este paquete no incluyen las expresiones con iteradores (LoopExp), estás son ejecutadas por el evaluador ya que requiere la evaluación de expresiones. La jerarquía de los números está encabezada por la clase Real y su subclase es Integer. La clase OclUndefined identifica a valores indefinidos, por ejemplo cuando se pide el primer elemento de una secuencia vacía. Por último se incluye otros tipos básicos como son los String y Boolean.

En la figura 82 se muestran las clases de este paquete, y para simplificar el diagrama solo se incluye algunas de sus operaciones.

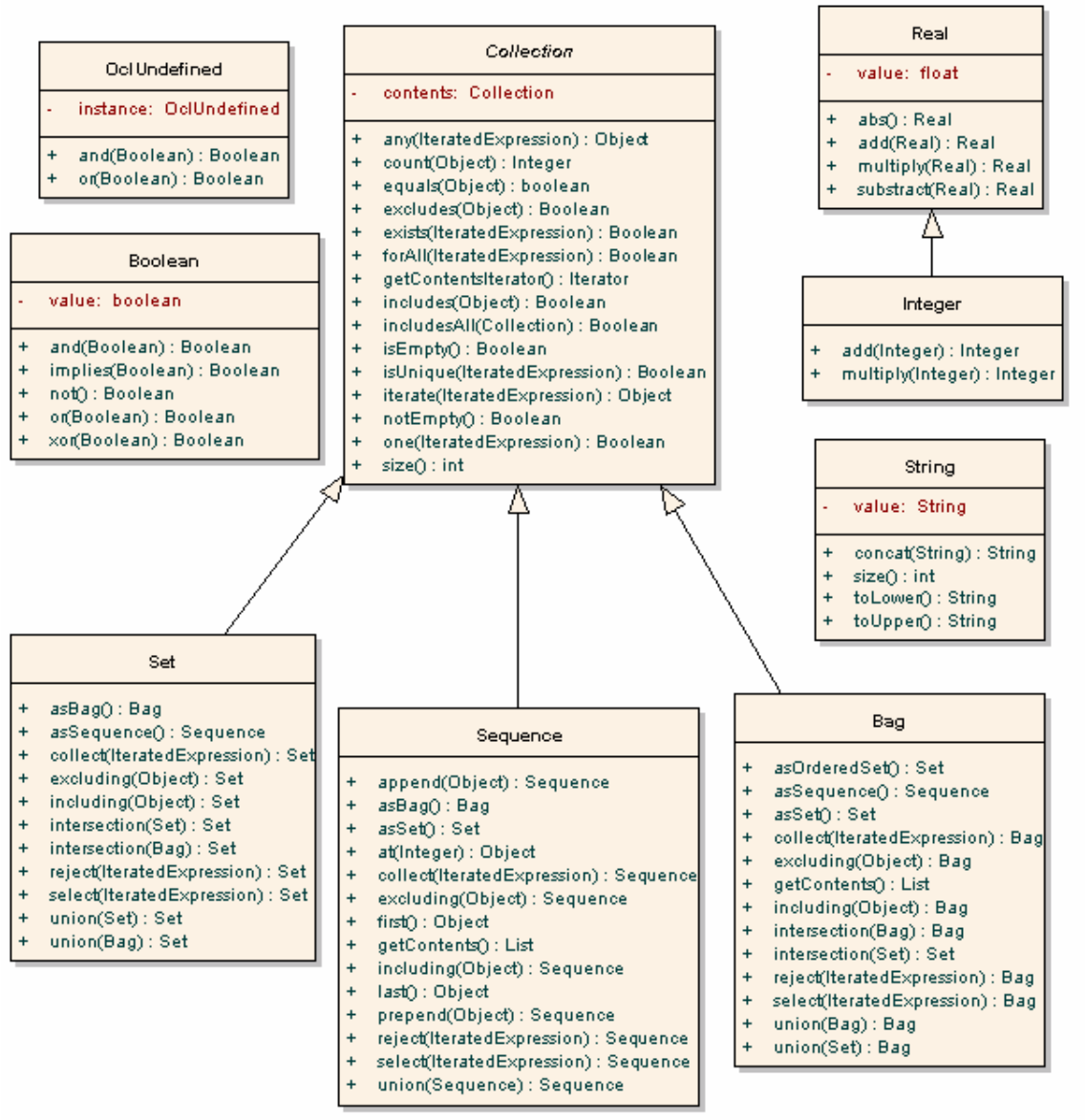


Figura 82 - Diagrama de clases del paquete basics

13.3.1.2. Estructura del paquete analyzer

En el diagrama de la figura 83 se ilustra la especificación de la clase Repository que es utilizada por el evaluador para acceder al modelo y metamodelo. De este modo, el evaluador de OCL se independiza de la implementación del metamodelo, por ejemplo si se implementa como objetos Java comunes se puede acceder al metamodelo por reflexión, si se implementa con EMF se puede utilizar la interfaz reflexiva proporcionada por este.

El objeto OclValue es el valor resultante de la evaluación (de cualquier expresión), este conoce el objeto resultante y el tipo que es un Classifier.

El evaluador de OCL se diseño como un Visitor [**22**], define un método analyse por cada una de las expresiones OCL, el cual además de recibir por parámetro la expresión recibe un objeto que en el caso del evaluador semántico es el ambiente evaluable de la

expresión dada. La evaluación de cada una de las expresiones resulta en un valor (OclValue).

La clase EvalEnvironment representa el ambiente de la expresión y permite relacionar el valor (OclValue) que tiene un elemento identificado por su nombre. El ambiente puede tener un padre, por ejemplo en una LoopExp la expresión del bucle tiene un nuevo ambiente que a diferencia de su padre posee los iteradores utilizados para recorrer la colección receptora. El nuevo ambiente (ej, el del bucle) se desecha una vez utilizado.

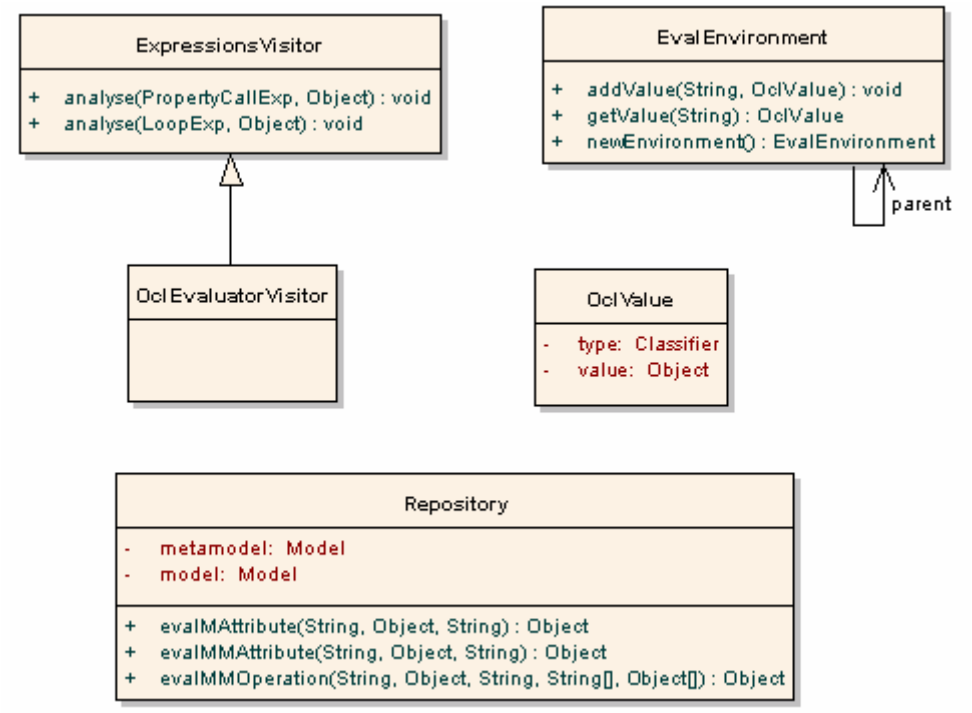


Figura 83 - Diagrama de clases del analizador semántico

13.4. Descripción del evaluador de refinamientos

Este módulo se encarga de implementar la estrategia de evaluación para las condiciones de refinamiento de modelos MOF propuesta en esta tesis. En la figura 84 se presenta el diagrama de clases de la implementación del evaluador de refinamiento.

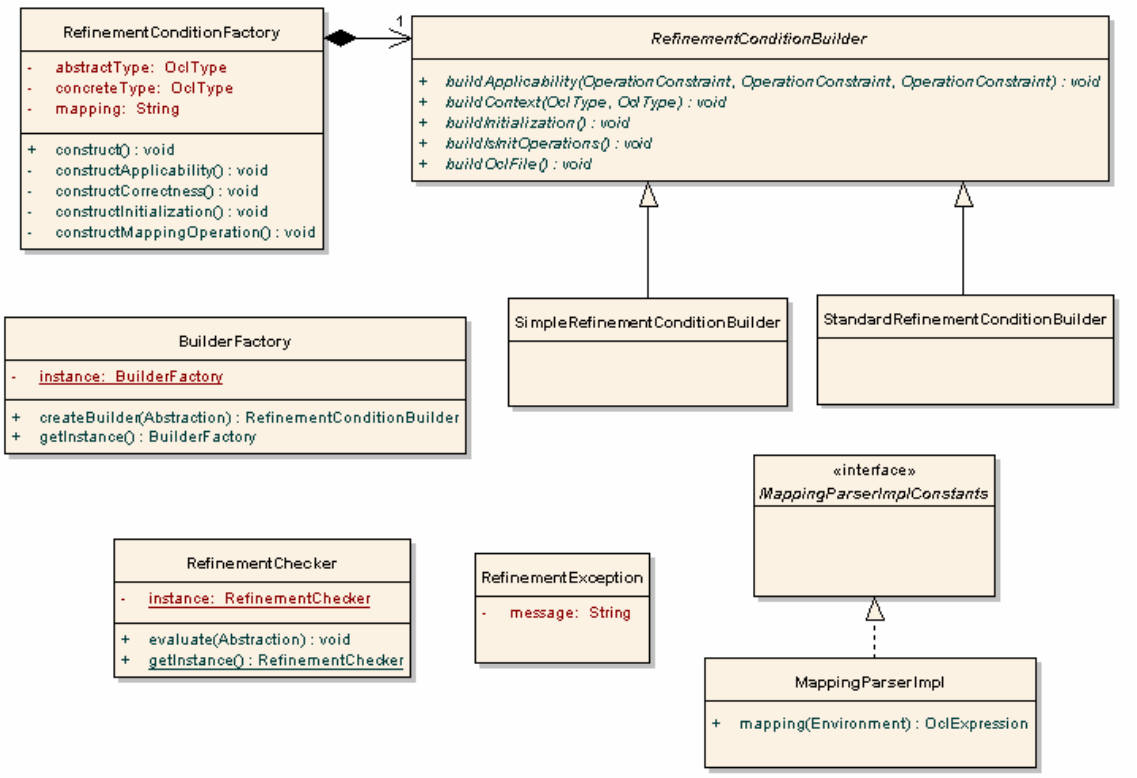


Figura 84 - Diagrama de clases de la implementación del evaluador de refinamientos

La clase *RefinementChecker* es una clase Singleton [**22**], tiene exactamente una instancia. Su responsabilidad es determinar si una abstracción dada es un refinamiento, para ello evalúa las condiciones de refinamiento planteadas en la sección 6.1. El objeto *RefinementConditionFactory* es el responsable de instanciar las condiciones de refinamiento utilizando la interfaz de *RefinementConditionBuilder*. Al conjunto de las restricciones a ser evaluadas por el *RefinementChecker* se denomina *archivo OCL*; este archivo está integrado por la condición de inicialización y por las condiciones de aplicabilidad y correctitud de cada una de las operaciones especificadas en el diagrama. Para crear el archivo OCL anteriormente mencionado se aplicó el patrón Builder [**22**]. En el caso que la abstracción a evaluar tenga asignado un mapping, por ejemplo, cuando se aplica el patrón de refinamiento State, se utiliza el constructor estándar. En cambio cuando no se especifica el mapping, por ejemplo, cuando se aplica el patrón de refinamiento Atomic Operation, se pueden crear las condiciones de refinamiento simplificadas, en este caso se utiliza un constructor simple.

La clase singleton *BuilderFactory* es la responsable de determinar el builder apropiado para una abstracción dada. El objeto MappingParserImpl es el responsable de llevar a cabo el análisis sintáctico del mapping asociado a la relación de refinamiento. En el caso que el mapping tenga errores de sintaxis el evaluador de refinamiento dispara una excepción RefinementException la cual conoce el mensaje del error.