Implementación de técnicas de evaluación y refinamiento para OCL 2.0 sobre múltiples lenguajes basados en MOF

Carlos Diego García

Director: Claudia Pons

Tesis presentada a la Facultad de Informática de la Universidad Nacional de La Plata como parte de los requisitos para la obtención del título de Magíster en Ingeniería de Software.

La Plata, Julio de 2006

Facultad de Informática
Universidad Nacional de La Plata
Argentina

Índice general

Índice de figuras

Índice de tablas

Implementación de técnicas de evaluación y refinamiento para OCL 2.0 sobre múltiples lenguajes basados en MOF

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, tal como ocurre en otros sistemas ingenieriles. 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 modelados como el Unified Modelling Language (UML) son principalmente basado en el uso de construcciones gráficas que transmiten un significado intuitivo. Estos lenguajes son atractivos para los usuarios porque ellos 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. Fue definido por la OMG (Object Management Group), permite expresar restricciones semánticas del sistema que no se pueden expresar a partir de una notación gráfica. 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.

Tanto UML como OCL están definidos a través de MOF (Meta Object Facility) una especificación de tecnología estandarizada por OMG en 1997. MOF es un meta-metamodelo que es utilizado para crear metamodelos. OCL puede ser utilizado para cualquier metamodelo que adhiera a MOF.

Otras características que son muy importantes 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, postergando los detalles del problema en etapas posteriores. En los lenguajes formales como Z [**33**] es posible demostrar si una especificación dada es un refinamiento de otra especificación, o incluso derivar refinamientos a partir de una determinada especificación. Para poder utilizar mecanismos de refinamientos en UML, es necesario aumentar la precisión de dicho lenguaje de modelado, y definir un marco para expresar la noción de refinamiento.

Algunos grupos de investigadores proponen la traducción de UML/OCL en lenguajes formales que soporten mecanismos de refinamientos para disminuir las deficiencias de UML. Este enfoque es valioso pero insuficiente, ya que no resuelve los siguientes inconvenientes:

- Falta de notación para especificar refinamientos: En UML no existen artefactos que permiten definir las relaciones de refinamientos **formalmente**.
- Presencia de refinamientos ocultos: En UML hay distintos artefactos que permiten definir relaciones de abstracción / refinamientos implícitamente, por ejemplo la generalización y composición. Estas relaciones deben descubrirse y documentarse.
- Falta de metodología de refinamientos: aparte de la formalización del lenguaje en si, es además necesario contar con una metodología de refinamiento, basada en una teoría formal.

Otra alternativa, como complemento de la anterior, es definir estructuras de refinamientos en UML/OCL equivalentes a las estructuras de refinamientos en los lenguajes formales. Esta propuesta ha sido explorada en [**29**] y [**25**].

El objetivo de esta tesis es desarrollar una herramienta de soporte para OCL 2.0 sobre múltiples lenguajes basados en MOF. Además se definirán formalmente relaciones de abstracción / refinamiento en UML/OCL para aumentar de este modo su precisión y fomentar el uso de técnicas de refinamientos en dicho lenguaje de modelado.

Esta tesis está organizada de la siguiente manera. En el capitulo 2 se introduce el concepto de MOF y metamodelos. En el capítulo 3 se describe el uso de OCL en modelos basados en MOF. En el capitulo 4 se expone el metamodelo de OCL 2.0. En el capítulo 5 se describe una estrategia de evaluación para las condiciones de refinamientos en modelos UML/OCL. En el capítulo 6 se describe la implementación de la herramienta que asiste al proceso de especificación y evaluación de refinamiento con UML y OCL. En el capitulo 7 se exponen los trabajos relacionados. En el capitulo 8 se expone las conclusiones finales y se citan futuros trabajos.

2. MOF y Metamodelos

La mayoría de los métodos orientados a objetos (métodos OO) tienen escaso rigor; su sintaxis y semántica no está definida formalmente. Una manera para que estos métodos sean más rigurosos es definiendo un metamodelo. Si bien este acercamiento carece de la precisión de la especificación de los métodos formales, ofrece una especificación intuitiva y comprensible. Un metamodelo se expresa con notación gráfica y con un lenguaje formal, como OCL, para aumentar su precisión y eliminar ambigüedades.

En este capitulo se describe el concepto de meta-metamodelo y metamodelo en la sección 2.1, el paquete core en la sección 2.2 y la arquitectura MOF en la sección 2.3.

2.1. 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.

Meta Object Facility (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 orientados a objetos de sistemas.

La especificación de MOF proporciona:

- Una definición formal del meta-metamodelo 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 para manejar cualquier tipo de metadato.
- Una jerarquía de interfaces reflexivas para manipular metadatos independientes del metamodelo.
- Un formato XMI[1] para el intercambio de modelo MOF.

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

2.2. Paquete core

Una de las motivaciones de esta tesis es la implementación de una herramienta con soporte a OCL 2.0 sobre múltiples lenguajes basados en MOF. Para ello se define el paquete core que es reusado por distintos metamodelos (UML, OCL, etc) y por MOF (figura 1).

En el paquete core se encuentran un conjunto de elementos que son comunes a cualquier metamodelo. La intención es que OCL y otros metamodelos puedan reusar todo o parte del paquete, permitiendo que estos sean beneficiados por una sintaxis abstracta y semántica ya definida.

Además el paquete core, provee la información contextual para la evaluación de expresiones OCL. La versión actual de la especificación de la sintaxis abstracta de OCL 2.0 importa un conjunto de metaclases del metamodelo de UML; esto limita la utilización de OCL solo para los modelos UML. La utilización del package core permite relacionar los conceptos de OCL con los elementos comunes a cualquier metamodelo.

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

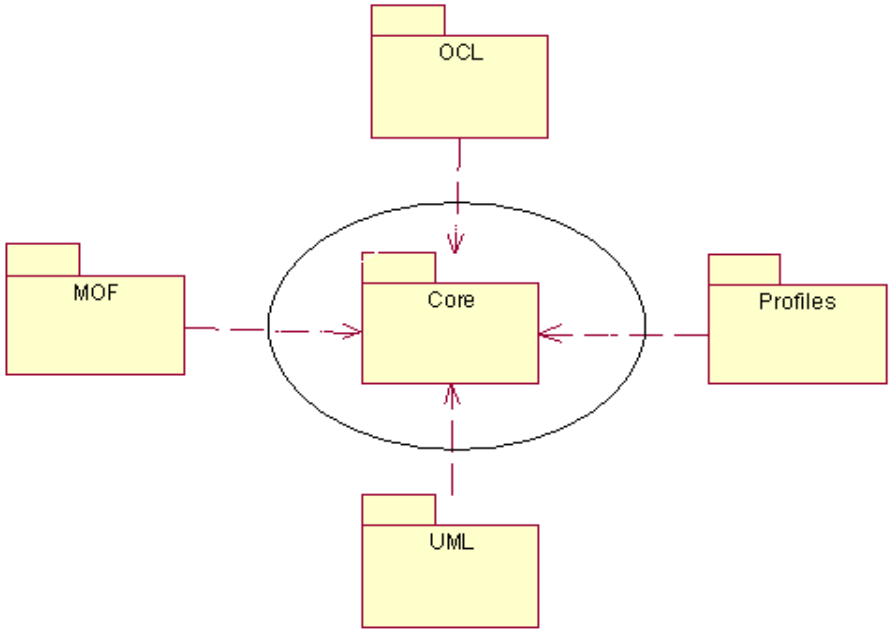

Figura 1 – El rol del paquete core

Para facilitar el reuso, el paquete core se subdivide en varios paquetes: *PrimitiveTypes*, *Abstractions*, *Basic*, y *Constructs* , como se ilustra en la Figura 2. Algunos de estos son divididos para facilitar el uso de los elementos del paquete al definir un nuevo metamodelo.

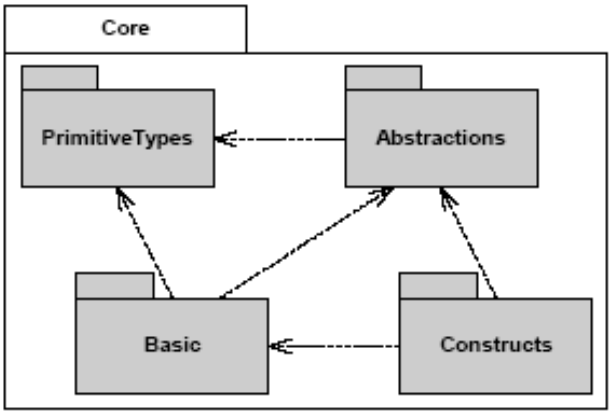

Figura 2 – Package Core

- El *paquete* Primitive*Types* es un paquete simple que contiene varios tipos predefinidos que normalmente se utilizan cuando se metamodela. Ejemplo: Integer, Boolean, etc.

- El *paquete* Ab*stractions* contienen varios paquetes con pocas metaclases cada uno, la mayoría son abstractas. El propósito de este paquete es proporcionar un conjunto muy reusable de metaclases que sean especializadas cuando se definan nuevos metamodelos. Ejemplo: Namespaces, Expressions, Constraints, etc

- *El paquete* C*onstructs* contiene varios paquetes, y reune muchos de los aspectos de las Abstractions. Las metaclases de Constructs tienden a ser concretas en lugar de abstractas**.**

- *El paquete Basic* contiene un subconjunto de estructuras que se emplean principalmente para los propósitos de XMI.

La figura 3 es una pequeña parte del metamodelo de OCL 2.0 que representa la llamada a propiedades (atributos y navegaciones) y operaciones. Como vemos dicho metamodelo utiliza los elementos del paquete core, permitiendo relacionar los conceptos de OCL con cualquier metamodelo MOF.

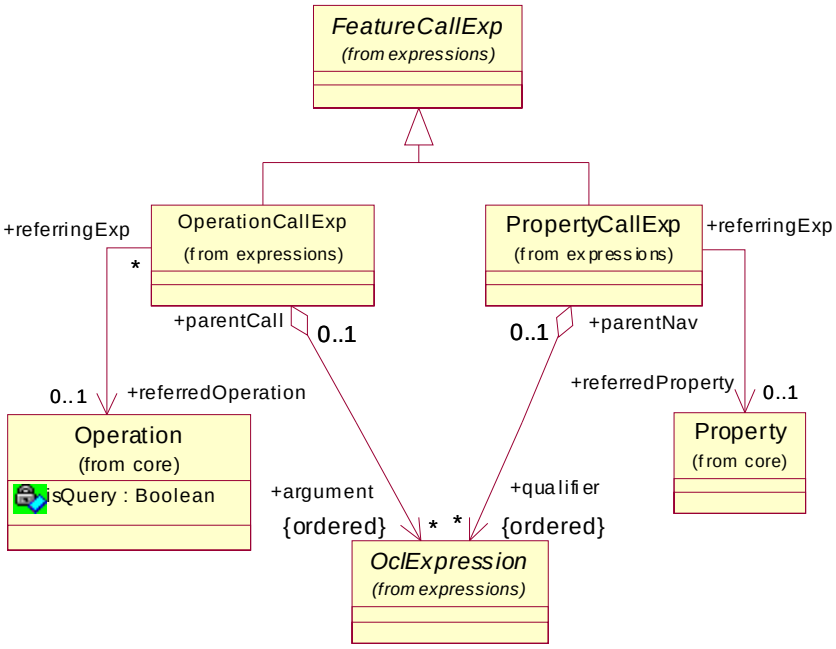

Figura 3 – Metamodelo de la sintaxis abstracta de OCL 2.0 para FeatureCallExp

2.3. 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.

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.

Un *modelo* es una instancia de un metamodelo. La responsabilidad de esta capa es definir un lenguaje que describa los dominios semánticos, es decir, 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.

Por último la *capa de instancias* es conocida como M0, representa las instancias de los elementos del modelo en tiempo de ejecución. El snapshot que son modelado en M1 son versiones reducidas de las instancias en tiempo de ejecución. La figura 4 es un ejemplo de modelo de 4 capas.

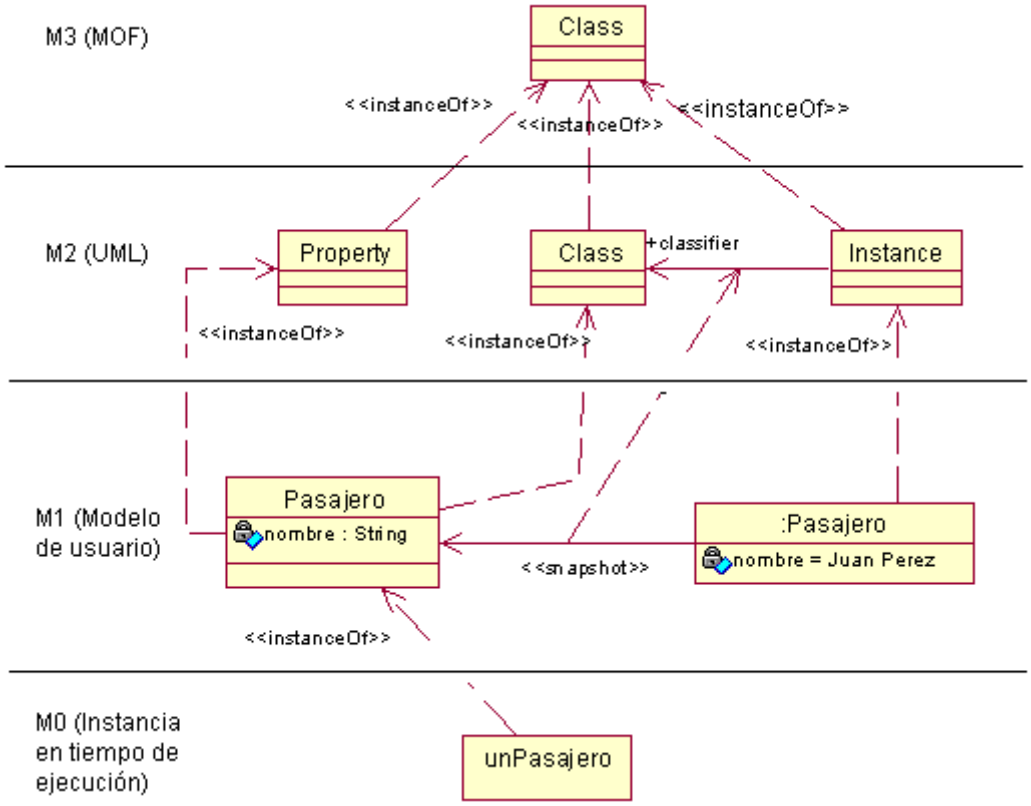

Figura 4 – Ejemplo de modelo de 4 capas

3. Uso de OCL en Modelos basados en MOF

OCL [**20**] es un lenguaje formal de especificación que es fácil de leer y escribir. Se puede emplear para especificar restricciones y otras expresiones adjuntas a los modelos MOF. Al ser un lenguaje de especificación se garantiza que toda expresión OCL es libre de efectos laterales; es decir la expresión no modifica 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 puede ser utilizado para un varios propósitos:

- Como un lenguaje de consulta
- Para especificar invariantes en clases y tipos de un modelo de clases.
- Para describir pre- y postcondiciones en Operaciones y Métodos.
- Para definir operaciones y variables adicionales para los tipos de un modelo de clases.
- Para describir guardas (Guards)
- Para especificar reglas de derivación para una propiedad.
- Para especificar los valores iniciales de las propiedades.
- Para especificar cualquier expresión de un modelo.
- Para especificar restricciones en operaciones.

Por lo tanto, el nombre de Object Constraint Language (OCL) 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 clase de la figura 5.

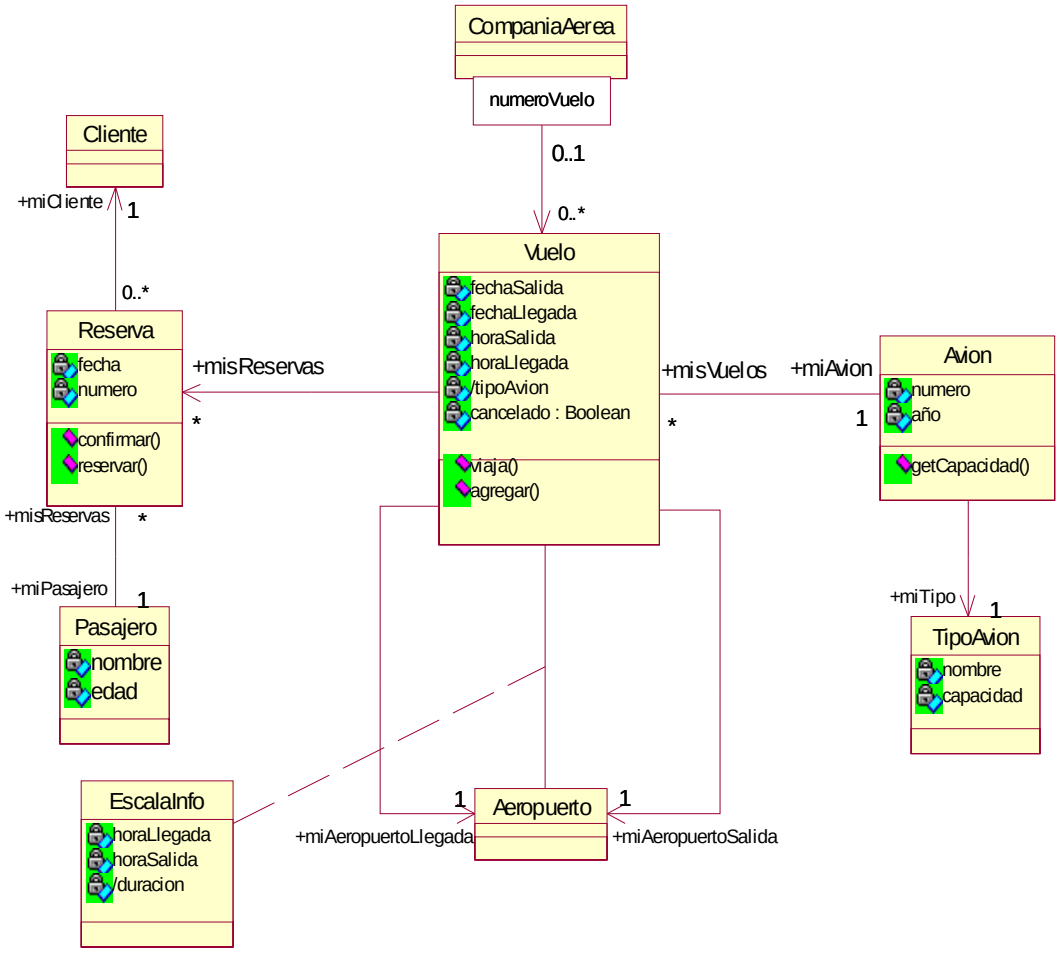

Figura 5 – Ejemplo de diagrama de clase

En este capitulo se describen las restricciones OCL en la sección 3.1, expresiones de valor inicial sección 3.2, de valor derivado sección 4.3, de operación de consulta sección 3.4 y de guarda sección 3.5.

3.1. Restricciones en OCL

Las restricciones que podemos especificar con OCL son:

- Invariantes
- Definiciones
- Precondiciones
- Postcondiciones

3.1.1. 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.

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 6 se representa la expresión OCL utilizada como invariante.

Su sintaxis es:

context [VariableName:] TypeName
inv: < OclExpression >

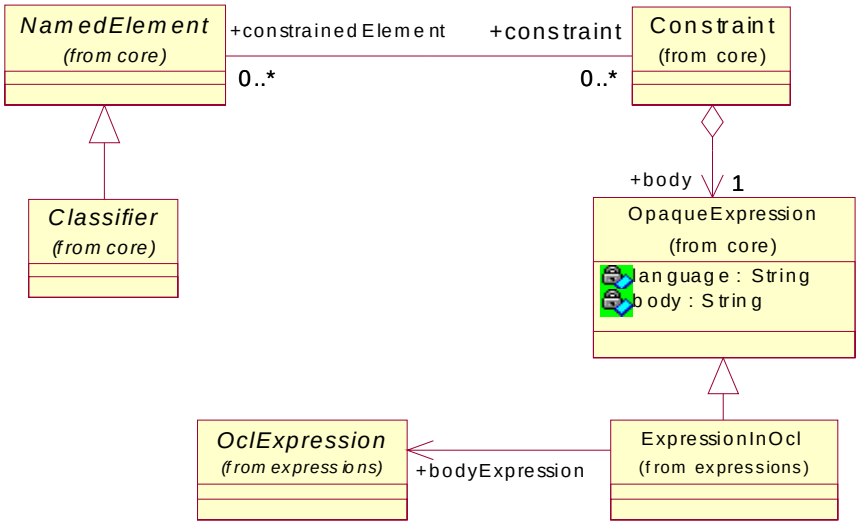

Figura 6 – Expresión OCL utilizada como Invariante o Definición

Ejemplo: El 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

El contexto de la instancia contextual, se escribe a continuación de la palabra reservada context, en este caso self es una variable que hace referencia a una instancia de Vuelo.

[2] Un Classifier [12] es una clasificación de instancias - describe un conjunto de instancias que tienen características comunes.

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 permite mejorar el modelo y pueden ser útiles al momento de generar código, por ejemplo que los Classifers 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 estas 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, Pasajero, etc) tienen alguna operación abstracta entonces estas no pueden ser concretas.

3.1.2. 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 6 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.miAvion.miTipo.capacidad
```

La variable capacidad es conocida en el contexto de Vuelo.

context Vuelo *inv*:
self.misReservas -> size() <= capacidad

3.1.3. 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 7 se ilustra la expresión OCL utilizada como precondición.

Su sintaxis es:

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

Ejemplo: La precondición de la operación agregar(unPasajero, unCliente) definida en la clase Vuelo, especifica que este no puede estar en estado cancelado y el pasajero (unPasajero) no debe estar asignado al correspondiente vuelo.

context Vuelo: agregar(unPasajero: Pasajero, unCliente: Cliente) : Reserva
pre: not self.cancelado and not self.viaja(unPasajero)

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

3.1.4. 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 7 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 viaja(unPasajero) definida en la clase Vuelo, retorna verdadero si el pasajero dado está asignado al correspondiente vuelo, en caso contrario retorna falso.

context Vuelo: viaja(unPasajero : Pasajero) : Boolean

[3] Un Operation [12] es un BehavioralFeature que declara un servicio que puede ser llevado a cabo por las instancias de un Classifier.

[4] Un BehavioralFeature [12] es una característica de un Classifier que especifica un aspecto del comportamiento de sus instancias.

post: result = self.misReservas -> exists (r | r.miPasajero = unPasajero)

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.

3.2. 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 8 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

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

context Vuelo :: cancelado : Boolean
init: false

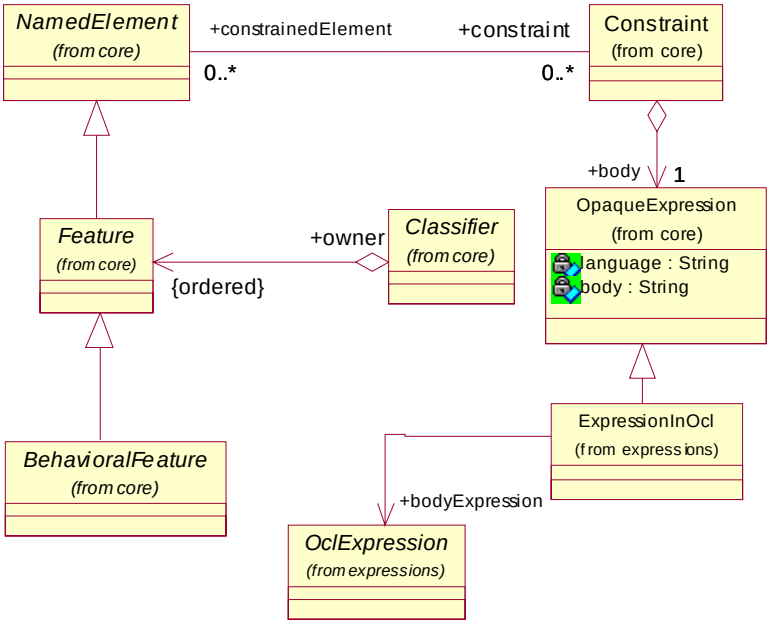

Figura 7 – Expresión OCL utilizada como precondición o postcondición

3.3. Expresión de Valor derivado

Una expresión de valor derivado es una expresión que se liga a una propiedad. 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 tenemos que tener en cuenta la multiplicidad

[5] Un Property [12] es un elemento tipado que representa un atributo de una clase.

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
derive: self.miAvion.miTipo

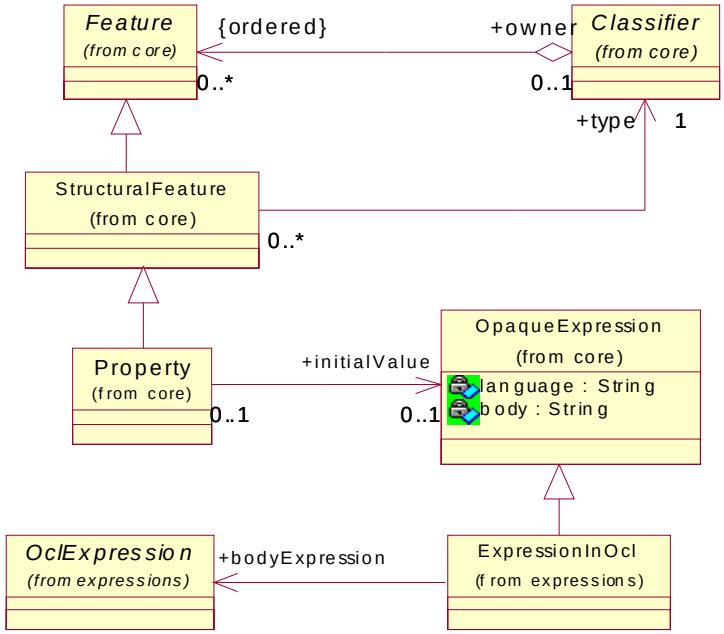

Figura 8 – Expresión OCL utilizada como Valor Inicial de una Propiedad

3.4. 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 9 se representa el metamodelo 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. Las precondiciones, postcondiciones y expresiones de consulta pueden ser combinadas luego de especificar el contexto de una operación.

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

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

3.5. Guarda

Una guarda es una expresión del tipo Boolean que se liga a una transición de una maquina de estados. Una expresión OCL que actúa como guarda se utiliza para restringir la transición. Es decir, la condición debe ser satisfecha para facilitar la activación de la transición asociada. El metamodelo para la guarda se ilustra en la figura 10.

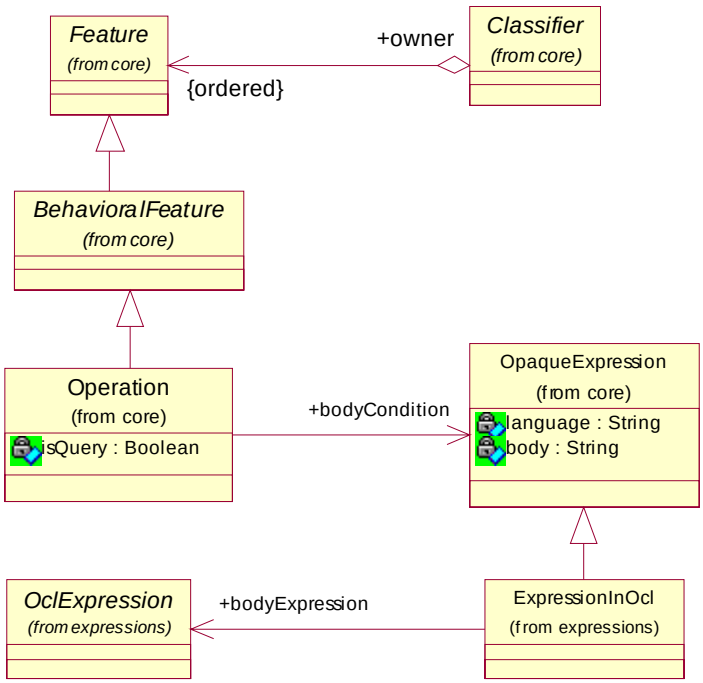

Figura 9 – Expresión OCL utilizada en una Operación de Consulta

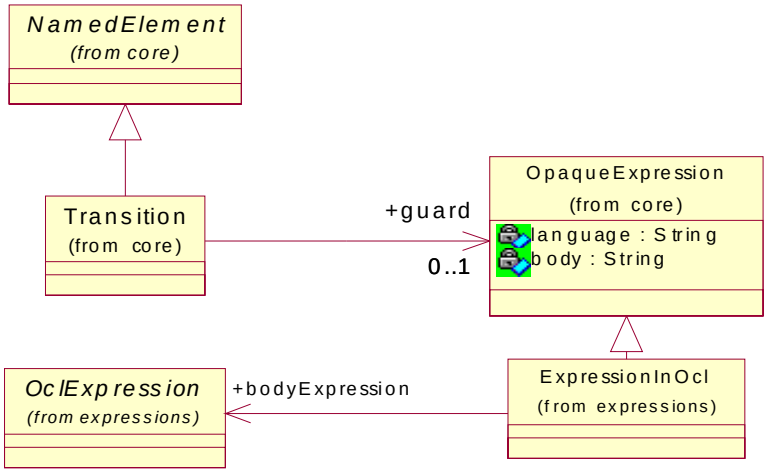

Figura 10 – Expresión OCL utilizada como una Guarda

4. 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 importa un conjunto de metaclases del paquete core presentado en el capitulo 2.

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 4.1 describe el paquete types. En la sección 4.2 se describe el paquete expressions. En la sección 4.3 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.

4.1. 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 tipos. En la Figura 11 se ilustra un extracto del paquete types.

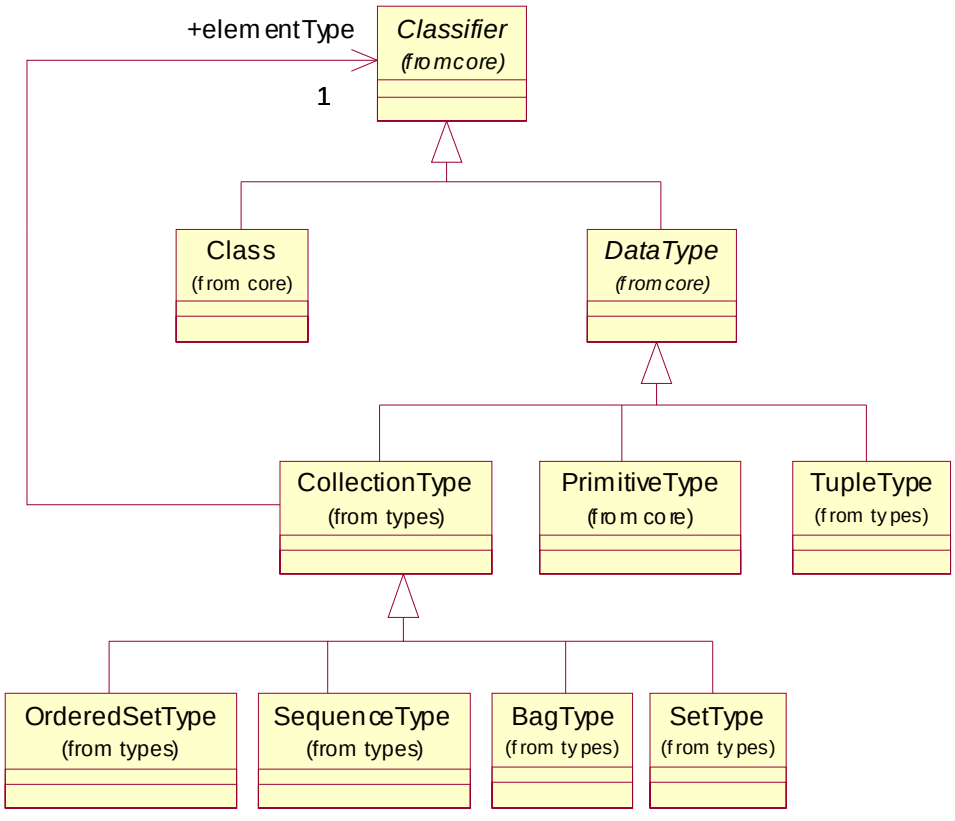

Figura 11 – Extracto del metamodelo de la sintaxis abstracta para los tipos de OCL

BagType

BagType es un tipo de colección que permite múltiples ocurrencia de un elemento. Estos no están ordenados.

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:* El tipo de los elementos de la colección. Todos los elementos que forman parte de la colección deben conformar a este tipo.

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.

4.2. Paquete expressions

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

En la figura 12 se ilustra la estructura básica del paquete expressions.

CallExp

Una CallExp 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 definida en un Classifier del modelo. El resultado es la evaluación de la propiedad correspondiente.

IfExp

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 del la expresión if es idéntica al resultado del thenExpression. En caso contrario el resultado del la expresión if es idéntica 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.

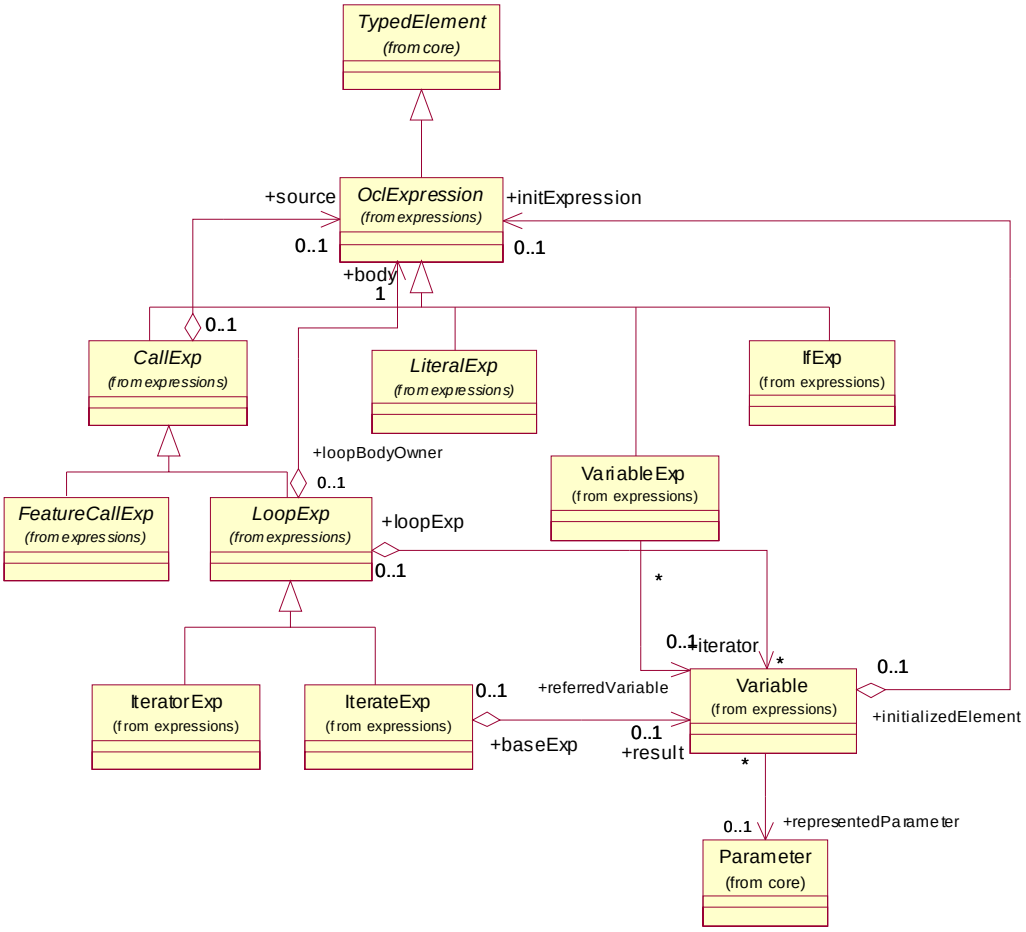

Figura 12 – La estructura básica del metamodelo de la sintaxis abstracta para las expresiones

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.

LiteralExp

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

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:* La expresión OCL que es evaluada para cada elemento de la colección receptora.

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 puede utilizar en las restricciones. En caso contrario para operaciones query, valores iniciales de atributo, etc.

El ambiente (Environment) de una OclExpression define que 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 :* La Variable a la que esta expresión se refiere.

ExpressionInOcl

La metaclase OclExpression está definida recursivamente entonces necesitamos una metaclase que represente el nivel superior del árbol de la sintaxis abstracta. En la figura 13 se representa la metaclase ExpressionInOcl.

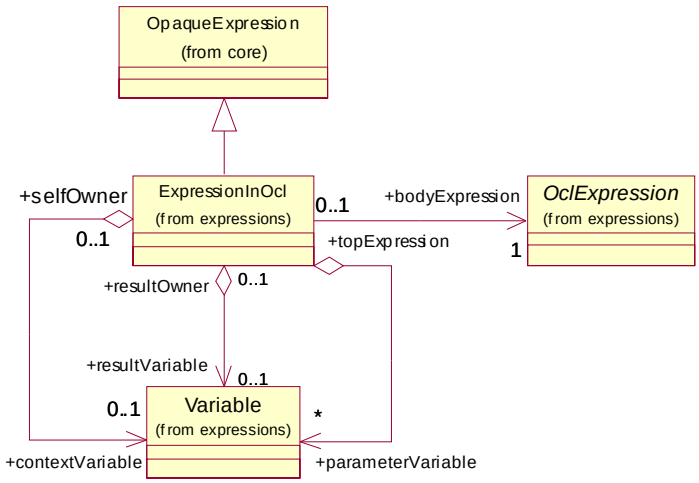

Figura 13 – Metamodelo de la sintaxis abstracta para ExpressionInOcl.

Asociaciones

- *bodyExpression:* El bodyExpression es la raíz de la expresión de 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.

4.3. 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 (anexo 10.2.). A continuación se enriquece al modelo UML presentado en la figura 5 con un invariante.

context Vuelo
inv regla1 : self.fechaSalida <= self.fechaLlegada

En la figura 14 se ilustra un diagrama de objeto que representa la instanciación del metamodelo de OCL para la regla anteriormente especificada.

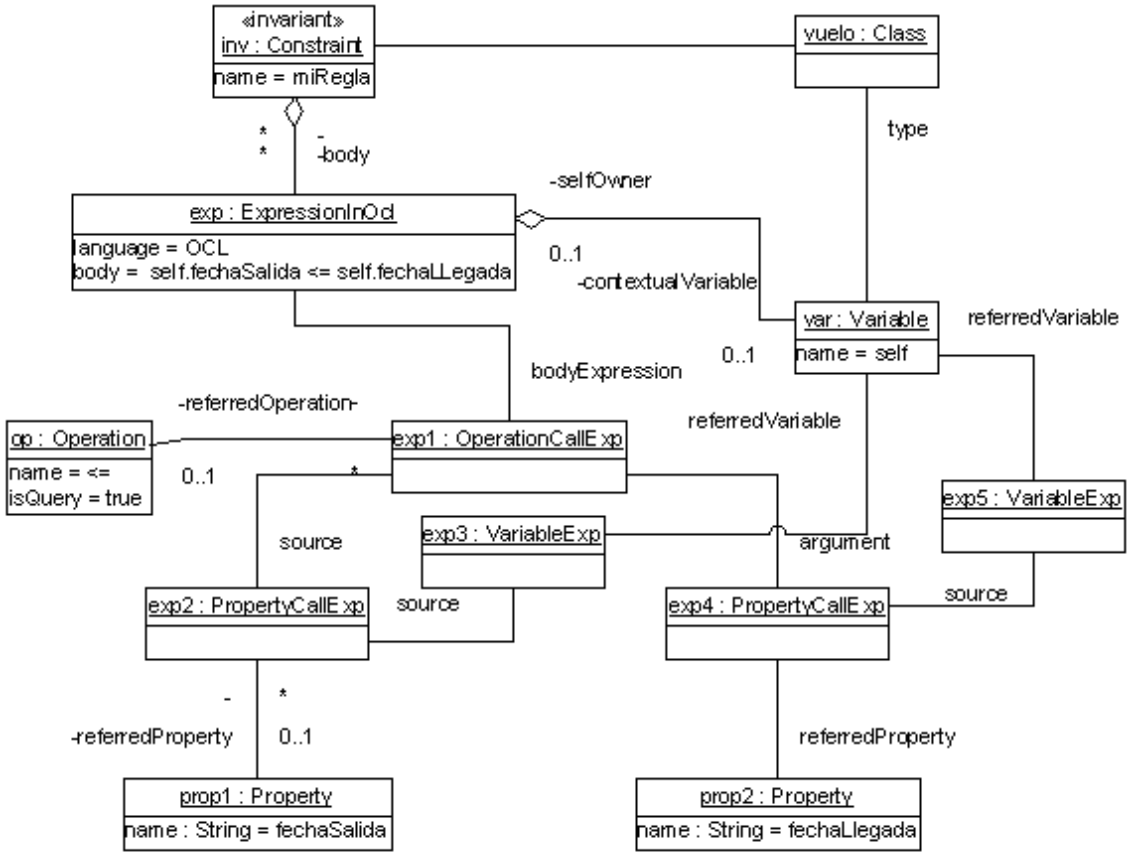

Figura 14 – Instanciación del metamodelo de la Sintaxis abstracta de OCL.

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.

5. Estrategia de evaluación para las condiciones de refinamientos en modelos UML/OCL

La idea promovida por model-driven enginnering (MDE) [**14**] [**34**] [**22**] es utilizar modelos en diferentes niveles de abstracción. El proceso MDE se inicia con la construcción de un modelo independiente de la plataforma. Luego se aplican una serie de transformaciones con el objetivo de especificar el sistema más especifico a la plataforma en cada paso de refinamiento. Tales transformaciones reducen el no determinismo tomando decisiones de diseño, por ejemplo como representar los datos.

En MDE las transformaciones predefinidas, escritas en un lenguaje de transformación estándar, QVT[**30**] se aplican en orden para evolucionar de modelo a modelo. Se supone que tales transformaciones se han validado previamente por un experto de MDE, 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 software, el concepto de refinamiento en MDE se define débilmente y se abre a malas interpretaciones. Este inconveniente surge por la presente semi-formalidad en los lenguaje de modelados utilizados en MDE y también debido a la inmadurez actual en este campo.

Hay dos alternativas para aumentar la robustez del mecanismo de refinamiento del MDE. Una de ellas es traducir el núcleo del lenguaje utilizado en MDE, es decir, UML [**36**], en un lenguaje formal como Z. Donde las propiedades son definidas y analizadas. Los trabajos presentados en [**2**], [**4**], [**9**], [**15**], [**17**], [**18**] y [**38**] entre otros, exploraron dicha línea de investigación. Esta alternativa es apropiada para descubrir y corregir las inconsistencias y ambigüedades del lenguaje gráfico, y en la mayoría de los casos nos permiten verificar y calcular refinamientos en modelos UML.

Sin embargo, tal acercamiento no es constructivo (no proporciona ningún feedback en términos de UML), requiere practica en la lectura y análisis de especificaciones formales y generalmente las propiedades que tienen que ser probada en el ambiente formal es demasiado complejo e indecidible. Una segunda alternativa es promover una definición formal de refinamiento, ej., simulación en Z, y expresarlo en términos de MDE.

Esta última alternativa, explorada en [**29**] y [**25**], consiste en definir estructuras de refinamientos en UML/OCL equivalentes a las estructuras de refinamientos en los lenguajes formales. En esta tesis se enriquece dicha propuesta definiendo condiciones de refinamiento escrita 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 matemáticos que normalmente no son aceptados por ingenieros de software. A pesar de que las condiciones de refinamiento no se expresan con un lenguaje formal, tiene un fundamento formal proporcionado por Object-Z.

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 propuesto por Daniel Jackson en [**13**] para evaluar fórmulas escritas en Alloy. Luego, Martin Gogolla en [**12**] 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 como se generan; con el propósito de reducir la cantidad de micromodelos a ser considerado. Para verificar las condiciones de refinamientos se adapta dicha estrategia.

La sección 5.1 se introduce al problema de especificación de refinamiento en Object-Z y UML. En la sección 5.2 se describe el método utilizado para crear las condiciones de refinamiento OCL para los patrones UML. En la sección 5.3 se explica como se aplica la estrategia de los micromodelos para la evaluación de refinamientos.

5.1. Especificación y verificación de refinamientos en Object-Z y UML

En Object-Z [**32**], una clase se representa como una caja nombrada con cero o más parámetros genéricos. El esquema de la clase puede incluir un tipo local o las definiciones constantes, a lo sumo un esquema de estado, un esquema de estado inicial y puede tener esquemas de operación. Estas operaciones definen el comportamiento de la clase especificando cualquier entrada y salida junto con una descripción de como cambian las variables de estado. Las operaciones se definen en términos de dos copias del estado: una no decorada que representa el estado previo y otra primada que representa el estado posterior.

Por ejemplo, la figura 15 ilustra la especificación de una clase simple llamada Vuelo, tiene un estado (que consiste en dos variables) y una sola operación.

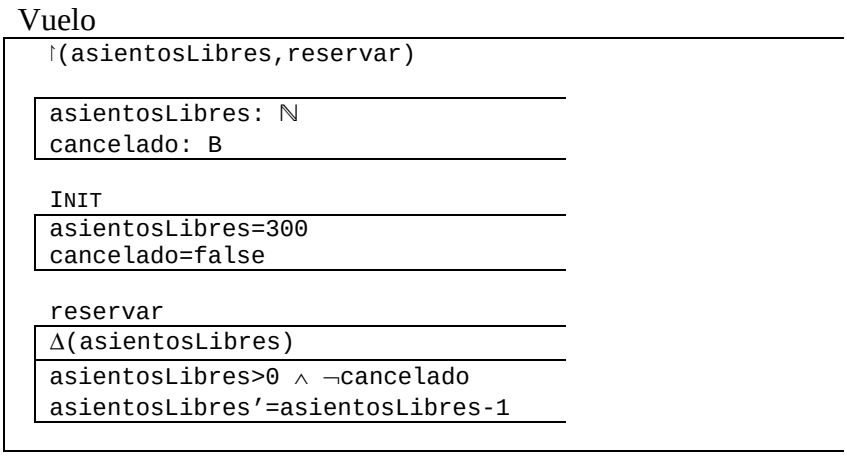

Figura 15 - Esquema simple Object-Z

Object-Z está provisto con un cálculo de esquema (es decir, un conjunto de operadores provistos para manipular esquemas Object-Z). Este hace posible la creación de especificaciones Object-Z que describen propiedades de otras especificaciones Object-Z . Para tratar con refinamientos necesitamos aplicar por lo menos los siguientes operadores:

- Operador STATE denota el conjunto de todos los posibles estados (snapshots) del sistema bajo consideración. Por ejemplo, Vuelo.STATE = {⟪asientosLibres==x, cancelado=t ⟫ | 0≤x≤300 ∧ t∈{true, false}}
- Operador INIT denota el estado inicial de un esquema dado. Por ejemplo, Vuelo. INIT={⟪*asientosLibres*==300 , cancelado=false⟩ | ⟫}
- Operador pre retorna la precondición de un esquema de operación; es decir el conjunto de todos los estados donde la operación puede aplicarse. Por ejemplo, pre reserva = {⟪*asientosLibres*==x, cancelado=false ⟫ | 0<x≤300}
- La conjunción de dos esquemas S y T (S ∧ T) resulta en un esquema que incluye a ambos.
- La implicación del esquema (S ⇒ T) indica la implicación lógica.

En [**6**] se enfoca refinamiento formalmente en el contexto de especificaciones Object-Z como se indica a continuación: una clase Object-Z C es un refinamiento (a través de la simulación descendente) de la clase A si hay una relación R en A.STATE ∧ C.STATE para que cada

operación abstracta visible A.op sea reformulada en una operación concreta visible C.op y se cumpla lo siguiente:

(Inicialización) $\forall$ C.STATE $\bullet$ C.INIT $\Rightarrow$ ($\exists$ A.STATE $\bullet$ A.INIT $\wedge$ R)

(Aplicabilidad) $\forall$ A.STATE $\bullet \forall$ C.STATE $\bullet$ R $\Rightarrow$ (pre A.op $\Rightarrow$ pre C.op)

(Correctitud) $\forall$ A.STATE $\bullet \forall$ C.STATE $\bullet$ $\forall$ C.STATE' $\bullet$ R $\wedge$ pre A.op $\wedge$ C.op $\Rightarrow$
$\exists$ A.STATE' $\bullet$ R' $\wedge$ A.op

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.

El lenguaje de modelado estándar UML [**36**] 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 (equivalente a la relación R de Object-Z) 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 [**36**] se formula empleando el lenguaje natural y permanece abierto a numerosas interpretaciones contradictorias.

5.2. Estrategia de verificación para patrones de refinamiento UML

Los patrones de refinamiento UML [**29**] [**16**] documenta las repetidas estructuras de refinamiento en modelos UML. En esta sección se presentan un proceso aplicado en modelos UML que contienen tales patrones para crear automáticamente las condiciones de refinamiento OCL para analizarlos de una manera rigurosa. La figura 16 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:

Refinement pattern instantiation. 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.

Transformation to OCL. Luego, la fórmula Object-Z F1 se traduce automáticamente en la fórmula OCL F1' aplicando la transformación T (la definición de T se presenta en el anexo 10.1.).

Micromodels strategy application. En este paso, se emplea la estrategia de los micromodelos en F1 ' para producir una fórmula F1 ' ' que son analizable dentro de un ámbito limitado.

OCL Evaluation. Finalmente, F1 ' ' se somete a un evaluator tradicional de OCL.

En las siguientes secciones se presenta este proceso a través de tres ejemplos concretos:

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

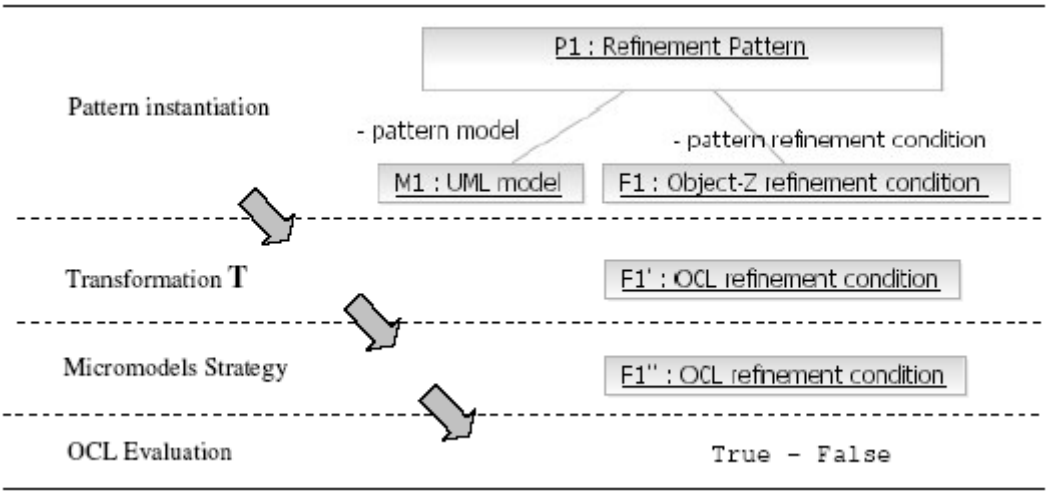

Figura 16 – Proceso de verificación de refinamiento

5.2.1. Patrón de Refinamiento State **estructural**

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.

Instancia de la estructura del patrón:

En la figura 17 se ilustra el modelo UML M1 que es conforme con la estructura del patrón de refinamiento State [**29**]. M1 contienen información sobre un sistema de reserva de vuelo donde cada vuelo es descrito abstractamente por la cantidad de asientos libres en su cabina; entonces 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 Boolean llamado cancelado que representar el estado del vuelo. Las operaciones disponibles son reservar para hacer una reservación de un asiento y cancelar para cancelar el vuelo. Una relación de refinamiento conecta la especificación abstracta a la concreta. El lenguaje OCL [**20**] se utiliza para especificar valores iniciales, precondiciones, postcondiciones y el mapping de la relación de refinamiento.

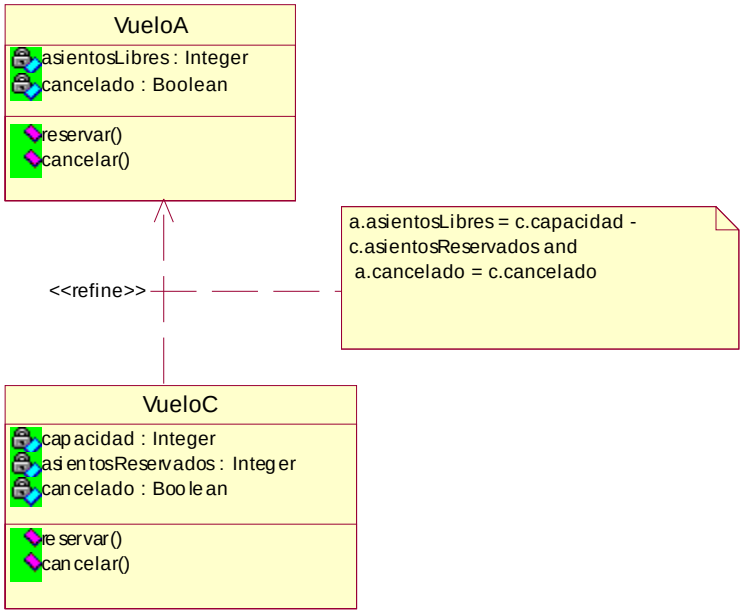

Figura 17 – Instancia del patrón de refinamiento State

Como una convención se utilizan las variables a y c para referenciar a la especificación abstracta y concreta respectivamente.

A continuación se especifican las restricciones OCL que enriquecen al modelo UML presentado en la figura 17:

Restricciones de la especificación abstracta:

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

context VueloA::cancelado: Boolean
    init: false

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

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

Restricciones de la especificación concreta:

```
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
```

Instancia de las condición de refinamiento del patrón:

Las condiciones de refinamiento Object-Z - F1 - para las clases UML VueloA y VueloC sobre alguna relación R se generan automáticamente desde la condición de refinamiento genérica establecida por el patrón [**29**], basada en la definición de simulación descendente en Object-Z descripta en [**6**]. La Figura 18 presenta la fórmula F1.

Inicialización

$$\forall \text{ VueloC.STATE} \bullet \text{VueloC.INIT} \Rightarrow (\exists \text{ VueloA.STATE} \bullet \text{ VueloA.INIT} \wedge \text{R})$$

Aplicabilidad (de la operación reservar)

$$\forall \text{ VueloA.STATE} \bullet \forall \text{ VueloC.STATE} \bullet \text{R} \Rightarrow (\text{pre VueloA.reservar} \Rightarrow \text{pre VueloC.reservar})$$

Correctitud (de la operación reservar)

$$\forall \text{ VueloA.STATE} \bullet \forall \text{ VueloC.STATE} \bullet \forall \text{ VueloC.STATE'} \bullet (\text{R} \wedge \text{pre VueloA.reservar} \wedge \text{VueloC.reservar}) \Rightarrow (\exists \text{ VueloA.STATE'} \bullet \text{R'} \wedge \text{VueloA.reservar})$$

Figura 18 – Instancia de las condiciones de refinamiento para el patrón de refinamiento State

El proceso de la transformación de Object-Z a OCL:

La condición de refinamiento Object-Z - F1 - se transforma automáticamente en la expresión OCL - F1 ' - aplicando la transformación T en el contexto de un modelo UML M1. Además de producir un OclExpression, T retorna un OclFile que contiene definiciones adicionales que se crean durante el proceso de transformación, como se indica a continuación:

***T* : UmlModel -> Zpredicate -> (OclExpression, OclFile)**

Los principales características de la transformación son las siguientes:

Paso #1: La relación Object-Z R es reemplazada por su equivalente OCL.

El mapping de la abstracción (es decir, la relación) se liga a la relación de refinamiento, y describe la relación entre los atributos del elemento abstracto y los atributos del elemento concreto; sin embargo, no se puede escribir expresiones OCL en el contexto de una relación. En Z, el contexto del mapping de la abstracción es la combinación de los estados abstractos y concretos (es decir, A.STATE $\wedge$ C.STATE); sin embargo, una combinación de Classifiers no es

legal en el contexto de OCL. Una solución consiste en traducir el mapping en una fórmula OCL en el contexto del Classifier abstracto, de la siguiente manera:

```
context a: VueloA def:
    mapping(c: VueloC): Boolean =
        a.asientosLibres = c.capacidad – c.asientosReservados
            and a.cancelado = c.cancelado
```

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

Paso #2: La expresión Object-Z INIT es expresada en términos de una operación OCL de tipo Boolean llamada isInit().

Se construye automáticamente una operación de consulta llamada isInit a partir de la especificación de valores iniciales de los atributos. Dicha operación retorna verdadero si todos los atributos de la instancia satisfacen la condición de inicialización, y falso en caso contrario.

Por ejemplo:

```
context VueloA def: isInit(): Boolean =
    (self.asientosLibres = 300 and self.cancelado = false)

context VueloC def: isInit(): Boolean =
    self.capacidad = 300 and self.cancelado = false and
        self.asientosReservados = 0
```

Paso #3: Las expresiones que contienen el operador Object-Z "pre" es traducida a la condición OCL correspondiente.

Por ejemplo, la expresión Object-Z:

```
    pre VueloA.reservar
```

Es transformada en:

```
    vueloA.asientosLibres > 0 and not vueloA.cancelado
```

Mientras que la expresión Object-Z:

```
  pre VueloC.reservar
```

Es transformada en:

```
    vueloC.capacidad - vueloC.asientosReservados > 0 and not
        vueloC.cancelado
```

Paso #4: La expresión Object-Z que representa la invocación a operación es traducida en la correspondiente postcondición OCL.

En Object-Z, los elementos que pertenecen al estado previo son denotados por identificadores no decorados, mientras que los elementos del estado posterior son expresados por identificadores con una decoración. En cambio en OCL la convención es opuesta a la anterior, es decir que los nombres no decorados se refieren a los elementos del estado posterior. Para ser consistente con el resto de la especificación, en las postcondiciones se añade una decoración ("_post") a cada identificador no decorado y la decoración original (@pre) se elimina del resto de los identificadores.

Por ejemplo, la siguiente expresión OCL forma parte de la postcondición de la operación reservar definida en la clase VueloA:

```
self.asientosLibres = self.asientosLibres@pre - 1
```

se renombra de la siguiente manera:

```
self_post.asientosLibres = self.asientosLibres - 1
```

Paso #5: Los cuantificadores y conectores lógicos son traducidos a operadores OCL.

La expresión Object-Z:

```
∀ S.STATE • exp
```

Es traducida a:

S.allInstances() -> forAll(s | T(exp))

La expresión Object-Z:

```
∃ S.STATE • exp
```

Es traducida a:

S.allInstances() -> exists(s | T(exp))

Nota:

El nombre de la clase, en minúscula, se utiliza para nombrar la variable del iterador. El símbolo $\Rightarrow$ se traduce a *implies* y el símbolo $\wedge$ se traduce a *and*.

El anexo 10.1. contiene la definición formal de la función de transformación T de las condiciones de refinamiento a las expresiones OCL.

La tabla 1 muestra la fórmula F1', que es el resultado de aplicar la transformación T al modelo UML M1 (figura 17) y las condiciones de refinamiento Object-Z F1 (figura 18).

Condición de Refinamiento OCL	
Inicialización	`VueloC.allInstances()->`***forAll***`(vueloC | vueloC.isInit()` ***implies*** `(VueloA.allInstances()->` ***exists***`( vueloA| vueloA.isInit()`***and*** `vueloA.mapping(vueloC))))`
Aplicabilidad	`VueloA.allInstances->` ***forAll***`( vueloA | VueloC.allInstances->` ***forAll***`( vueloC | vueloA.mapping(vueloC)` ***implies*** `(vueloA.asientosLibres > 0` ***and*** ***not*** `vueloA.cancelado` ***implies*** `vueloC.capacidad - vueloC.asientosReservados > 0` ***and not*** `vueloC.cancelado)))`
Correctitud	`VueloA.allInstances()->` ***forAll***`(vueloA | VueloC.allInstances() ->` ***forAll***`( vueloC | VueloC.allInstances()->` ***forAll***`( vueloC post| vueloA.mapping(vueloC)` ***and*** `(vueloA.asientosLibres > 0` ***and*** ***not*** `vueloA.cancelado)` ***and*** `(vueloC_post.asientosReservados = vueloC.asientosReservados + 1)` ***implies***

```
VueloA.allInstances()-> exists( vueloA post|
vueloA_post.mapping(vueloC_post) and
vueloA post.asientosLibres = vueloA.asientosLibres -1 ))))
```

Tabla 1 – Condiciones de refinamiento del patrón de refinamiento State.

5.2.2. Patrón de Refinamiento Object Descomposition **estructural**

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

Instancia de la estructura del patrón

Un refinamiento de la clase VueloC se obtiene especificando con más detalle el hecho de que un vuelo contiene una colección de asientos (figura 19). 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.

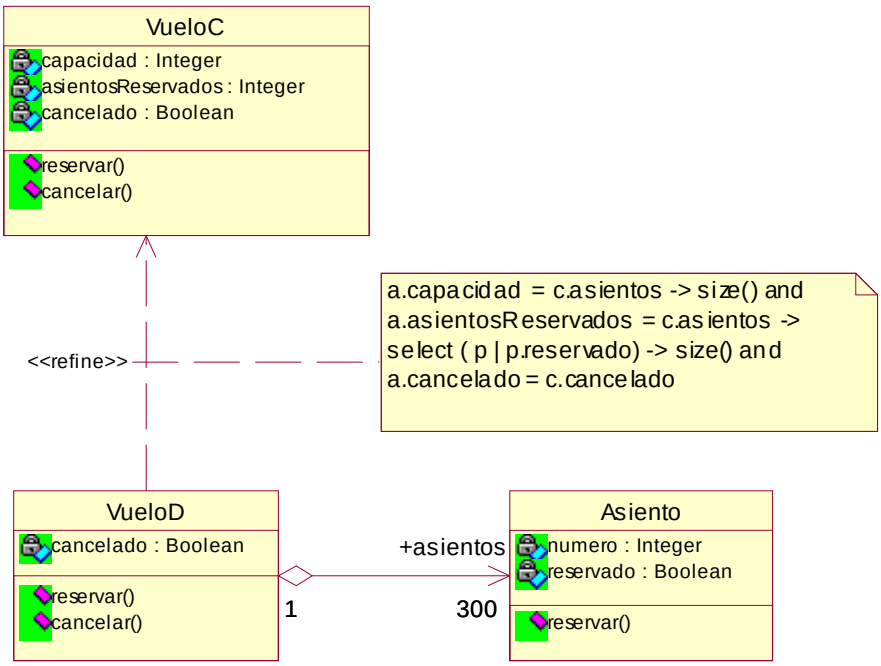

Figura 19 – Instancia del patrón de refinamiento Object Descomposition

A continuación se especifican las restricciones OCL que enriquecen al modelo UML ilustrado en la figura 19:

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 Asiento::reservado: Boolean
    init: false

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

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 ( q | q.numero = p.numero))

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

Instancia de la condición de refinamiento del patrón:

El algoritmo para generar la condición de refinamiento para este patrón es similar al descrito para el patrón de refinamiento State. La figura 20 presenta la especificación de la condición de refinamiento F1, basada de la definición de simulación descendente en Object-Z que es automáticamente generada del modelo UML de la figura 19 por instanciación del patrón Object Descomposition.

Inicialización

$$\forall \text{ VueloD.STATE} \bullet \text{VueloD.INIT} \Rightarrow (\exists \text{ VueloC.STATE} \bullet \text{ VueloC.INIT} \wedge \text{R})$$

Aplicabilidad (de la operación reservar)

$$\forall \text{ VueloC.STATE} \bullet \forall \text{ VueloD.STATE} \bullet \text{R} \Rightarrow (\text{pre VueloC.reservar} \Rightarrow \text{pre VueloD.reservar})$$

Correctitud (de la operación reservar)

$$\forall \text{ VueloC.STATE} \bullet \forall \text{ VueloD.STATE} \bullet \forall \text{ VueloD.STATE'} \bullet (\text{R} \wedge \text{pre VueloC.reservar} \wedge \text{VueloD.reservar}) \Rightarrow (\exists \text{ VueloC.STATE'} \bullet \text{R'} \wedge \text{VueloC.reservar})$$

Figura 20 – Instancia de las condiciones de refinamiento para el patrón de refinamiento Object Descomposition

El proceso de la transformación de Object-Z a OCL:

La condición de refinamiento Object-Z - F1 - se transforma automáticamente en las fórmulas OCL - F1 ' - que representan las condiciones de refinamiento entre las clases UML, aplicando la transformación T en el contexto del modelo UML M1 de la figura 19. Las principales características de la transformación son las siguientes:

Paso #1: La relación R Object-Z es reemplazada por su equivalente OCL.

El mapping de la abstracción es transformada a una definición OCL en el contexto de la especificación abstracta.

```
context a: VueloC def:
    mapping(c: VueloD): Boolean =
        a.capacidad = c.asientos -> size() and
            a.asientosReservados = c.asientos -> select(p|p.reservado)
                -> size() and a.cancelado = c.cancelado
```

Paso #2: La expresión Object-Z INIT es expresada en términos de una operación OCL de tipo Boolean llamada isInit().

Este proceso comienza recolectando los valores iniciales de los atributos especificados en el diagrama UML; luego se obtiene la multiplicidad de las asociaciones compuestas, y por último, se invoca la operación isInit() para cada uno de los componentes. A continuación se especifica la operación isInit() para la clase VueloD:

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

Paso #3: Las expresiones que contienen el operador Object-Z "pre" es traducida a la condición OCL correspondiente.

Por ejemplo, la expresión Object-Z:

```
pre VueloD.reservar
```

Es transformada en:

```
vueloD.asientos -> exists ( q | not q.reservado ) and
    not vueloD.cancelado
```

Paso #4: La expresión Object-Z que representa la invocación a operación es traducida en la correspondiente postcondición OCL.

Este paso es idéntico al presentado en el patrón State.

Por ejemplo, la siguiente expresión OCL forma parte de la postcondición de la operación reservar definida en la clase VueloD:

```
self.asientos -> exists ( p | p.reservado and self.asientos@pre
    -> select ( q | not q.reservado) -> exists ( q | q.numero =
        p.numero ) )
```

se renombra automáticamente de la siguiente manera:

```
self_post.asientos -> exists ( p | p.reservado and self.asientos
    -> select ( q | not q.reservado) -> exists ( q | q.numero =
        p.numero ) )
```

La tabla 2 presenta la fórmula F1', que es el resultado de aplicar la transformación T al modelo UML M1 (figura 19) y las condiciones de refinamiento Object-Z F1 (figura 20).

Condición de Refinamiento OCL	
Inicialización	`VueloD.allInstances()->forAll(vueloD| vueloD.isInit() implies (VueloC.allInstances()-> exists(vueloC| vueloC.isInit() and vueloC.mapping(vueloD))))`
Aplicabilidad	`VueloC.allInstances()-> forAll( vueloC | VueloD.allInstances()-> forAll( vueloD | vueloC.mapping(vueloD) implies ( vueloC.capacidad – vueloC.asientosReservados > 0 and not vueloC.cancelado implies vueloD.asientos -> exists (q| not q.reservado) and not vueloD.cancelado)))`
Correctitud	`VueloC.allInstances()-> forAll(vueloC | VueloD.allInstances() -> forAll(vueloD | VueloD.allInstances()-> forAll(vueloD post| vueloC.mapping(vueloD) and (vueloC.capacidad - vueloC.asientosReservados > 0 and not vueloC.cancelado) and (vueloD_post.asientos -> exists(p | p.reservado and vueloD.asientos -> select(q |not q.reservado) -> exists(q | q.numero = p.numero)) implies VueloC.allInstances()-> exists(vueloC post| vueloC_post.mapping(vueloD_post) and vueloC post.asientosReservados = vueloC.asientosReservados + 1)))))`

Tabla 2 – Condiciones de refinamiento del patrón de refinamiento Object Descomposition.

5.2.3. Patrón de Refinamiento Atomic Operation **comportamiento**

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.

Instancia de la estructura del patrón

En la figura 21 se ilustra un modelo UML M1 que es conforme al patrón de refinamiento Atomic Operation. 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).

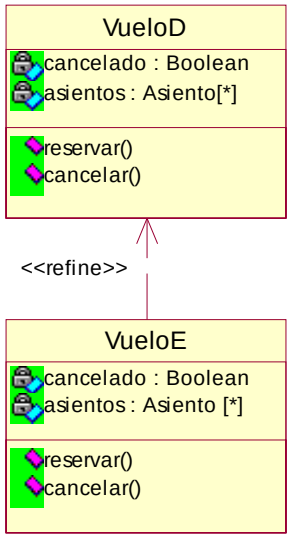

Figura 21 – Instancia del patrón de refinamiento Atomic Operation

Las siguientes restricciones enriquecen al modelo UML planteado en la figura 21:

Restricciones de la especificación abstracta:

```
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 ( q | q.numero = p.numero))




context VueloD::cancelar()
```

```
    pre: not self.cancelado
    post: self.cancelado
```

Restricciones de la especificación concreta:

```
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
```

Instancia de la condición de refinamiento del patrón:

La figura 22 presenta la especificación de las condiciones de refinamiento basada en la definición de simulación descendente en Object-Z. Dichas condiciones son más simples debido a que ambas clases tiene el mismo estado (es decir, la relación R es la función identidad); además la condición de inicialización no necesita ser verificada ya que la operación refinada no está involucrada en el proceso de inicialización.

Aplicabilidad (de la operación reservar)

$$\forall \text{ VueloD.STATE} \bullet (\text{ pre VueloD.reservar} \Rightarrow \text{pre VueloE.reservar [VueloD /VueloE] })$$

Correctitud (de la operación reservar)

$$\forall \text{ VueloD.STATE} \bullet \forall \text{ VueloD.STATE'} \bullet (\text{ pre VueloD.reservar} \wedge \text{VueloE.reservar [VueloD /VueloE] }) \Rightarrow \text{VueloD.reservar}$$

Figura 22 – Instancia de la condición de refinamiento del patrón de refinamiento Atomic Operation

El proceso de la transformación de Object-Z a OCL:

La transformación T se aplica al modelo UML de la figura 21 y las condiciones de refinamientos de la figura 22 para producir automáticamente las fórmulas OCL de la tabla 3, que representan las condiciones de refinamiento conforme al patrón de refinamiento Atomic Operation.

Condición de Refinamiento OCL	
Aplicabilidad	`VueloD.allInstances()-> forAll( vueloD | vueloD.asientos -> exists( q |not q.reservado) and not vueloD.cancelado implies vueloD.asientos->exists( q | not q.reservado) and not vueloD.cancelado )`
Correctitud	`VueloD.allInstances()-> forAll( vueloD | VueloD.allInstances()->forAll( vueloD_post | ( vueloD.asientos -> exists(q |not q.reservado) and not vueloD.cancelado and vueloD_post.asientos -> exists(s| s.reservado and vueloD.asientos`

```
          -> select(q |not q.reservado)
          -> asSequence().first().numero = s.numero ) )
          implies (vueloD post.asientos -> exists(s| s.reservado
          and vueloD.asientos -> select(q|not q.reservado)
          -> exists(q| q.numero = s.numero)))))
```

Tabla 3 – Condiciones de refinamiento del patrón de refinamiento Atomic Operation.

5.3. Micro-Mundos para la evaluación de las condiciones de refinamiento

Incluso los modelos pequeños como el presentado en la figura 17 especifica un numero 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 (micro-mundo) 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 [**13**] expresa que las respuestas negativas tienden a ocurrir en mundos pequeños, entonces las respuestas positivas son muy alentadoras.

Por ejemplo, para generar los micro-mundos utilizados para la evaluación de las condiciones de refinamiento del diagrama de clase de la figura 19, se proporcionan los invariantes de la figura 23 que reduce el tamaño de estos.

```
package vuelos
  context VueloC
    inv: Set { 4 .. 5 } -> includes (self.capacidad)
    inv: Set { 0 .. 5 } -> includes (self.asientosReservados)
  context Asiento
    inv: Set { 1 .. 5 } -> 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 23 – Invariantes OCL que reducen el espacio de búsqueda

Por ejemplo, la figura 24 ilustra uno de los micro-mundos que satisfacen los invariantes presentados en la figura 23. En tal micro-mundo la expresión VueloC.allInstances () retorna un conjunto finito de tamaño dos que contiene a los objetos VueloC1 y VueloC2, mientras que VueloD.allInstances () un conjunto finito de tamaño dos que contiene a los objetos VueloD1 y VueloD2.

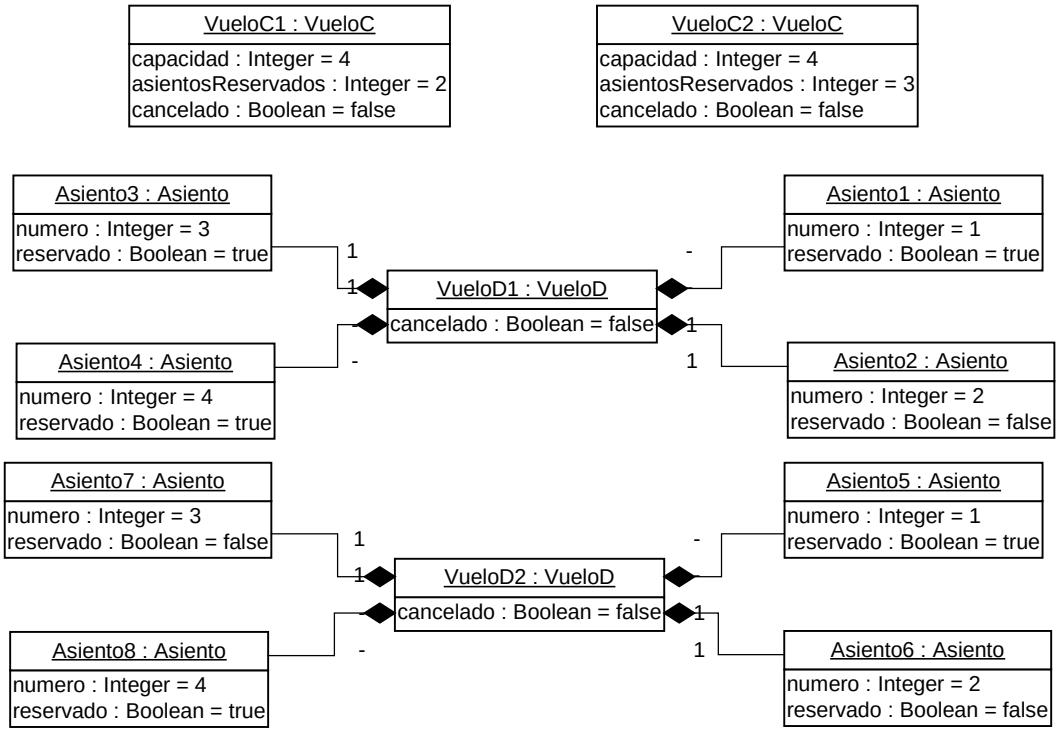

Figura 24 – Micro-mundo generado automáticamente del modelo UML de la figura 19
enriquecido con las restricciones de la figura 23

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 -> select (p* | ***not*** *p.reservado)* -> notEmpty()
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.

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 micro-mundo de la figura 24 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 → VueloD2

Para analizar apropiadamente las relaciones de refinamiento, los micro-mundos que se generan deben satisfacer todos 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 relacionados por el mapping de la abstracción. El proceso de generación automático de micro-mundo llevado a cabo por la herramienta asegura el cumplimiento de la propiedad de dualidad.

6. Implementación de la herramienta ePlatero

Es importante contar con una herramienta automatizada que evalué las propiedades de los modelos y sirva de soporte para las técnicas de refinamientos. Es común encontrar editores UML, sin embargo hay pocas herramientas que permiten la evaluación de reglas semánticas. Eplatero [**10**] es una herramienta CASE educativa que soporta el proceso de desarrollo de software dirigido por modelo 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 [**7**] (figura 25) 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.

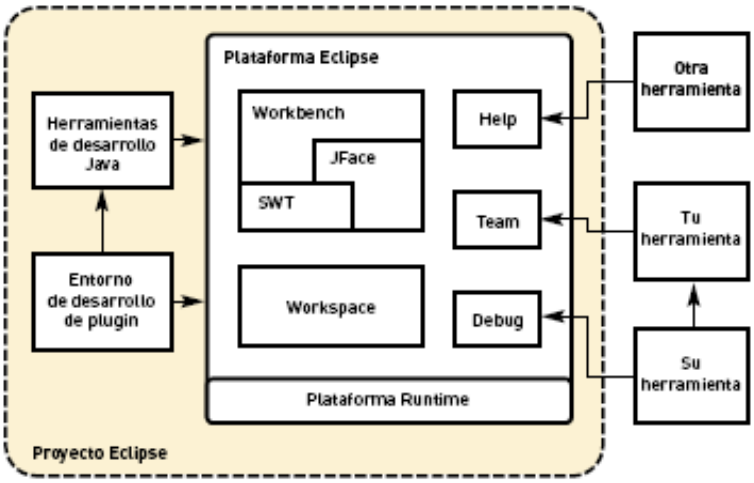

Figura 25 –Arquitectura de eclipse, para el desarrollo de plugins

La sección 5.1 describe la arquitectura de ePlatero. En la sección 5.2 se describe brevemente el proyecto Eclipse Modeling Framework y su importancia para el desarrollo de herramientas compatibles con MOF. En la sección 5.3 se presentan los módulos de ePlatero. En la sección 5.4 se muestran ejemplos de como se comporta la herramienta.

6.1. Arquitectura

La arquitectura de ePlatero [**10**] respeta el estándar de Eclipse para el desarrollo de plugins. Consta de 8 módulo (figura 26):

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

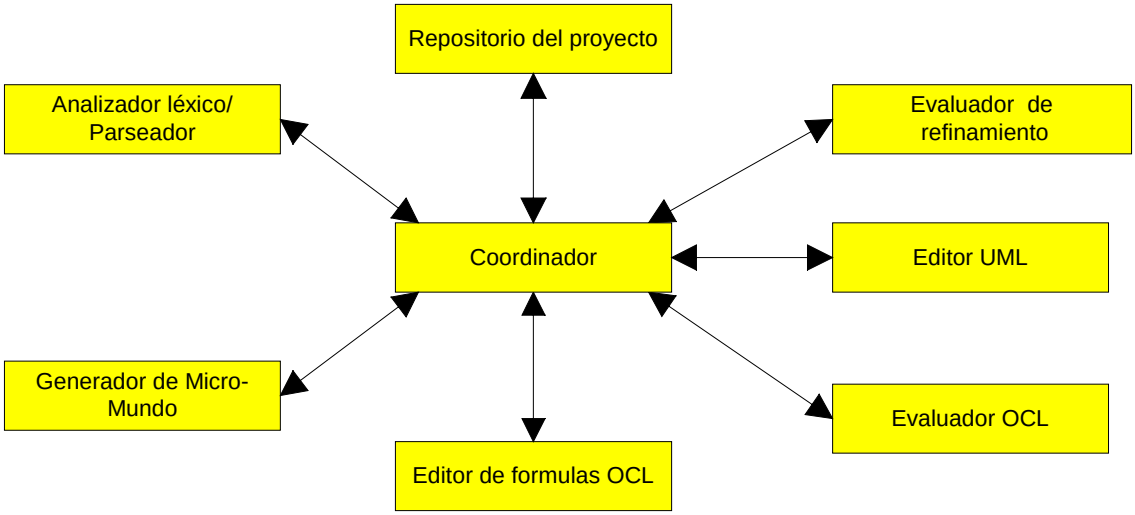

Figura 26 –Arquitectura de ePlatero

6.2. Eclipse Modeling Framework

El Eclipse Modeling Framework (EMF) [**8**] es un framework de modelado para Eclipse y generador de código que es muy útil para el desarrollo de herramientas y aplicaciones.
EMF consiste en dos frameworks:

- Core. Provee la generación básica y soporte a la runtime para crear clases Java para un modelo.
- Edit: Extiende y se construye sobre el framework core y añade soporte para la generación de clases adaptadoras que observan y comandan la edición de un modelo, e incluso provee un editor básico de modelo.

EMF comenzó como una implementación de la especificación de MOF, pero luego evolucionó en base a la experiencia adquirida en la construcción de un conjunto de herramientas basadas en EMF. Sin embargo, soporta la lectura y escritura de serializaciones MOF, y está basado en un meta-metamodelo llamado Ecore (figura 27) equivalente a su progenitor. EMF al igual que MOF respeta el estándar de XMI facilitando de este modo el intercambio de modelo en diferentes herramientas compatibles con MOF.

Un modelo ecore se puede generar a partir de:

1. Diagramas de clases UML
2. Interfaces Java
3. Esquemas XML

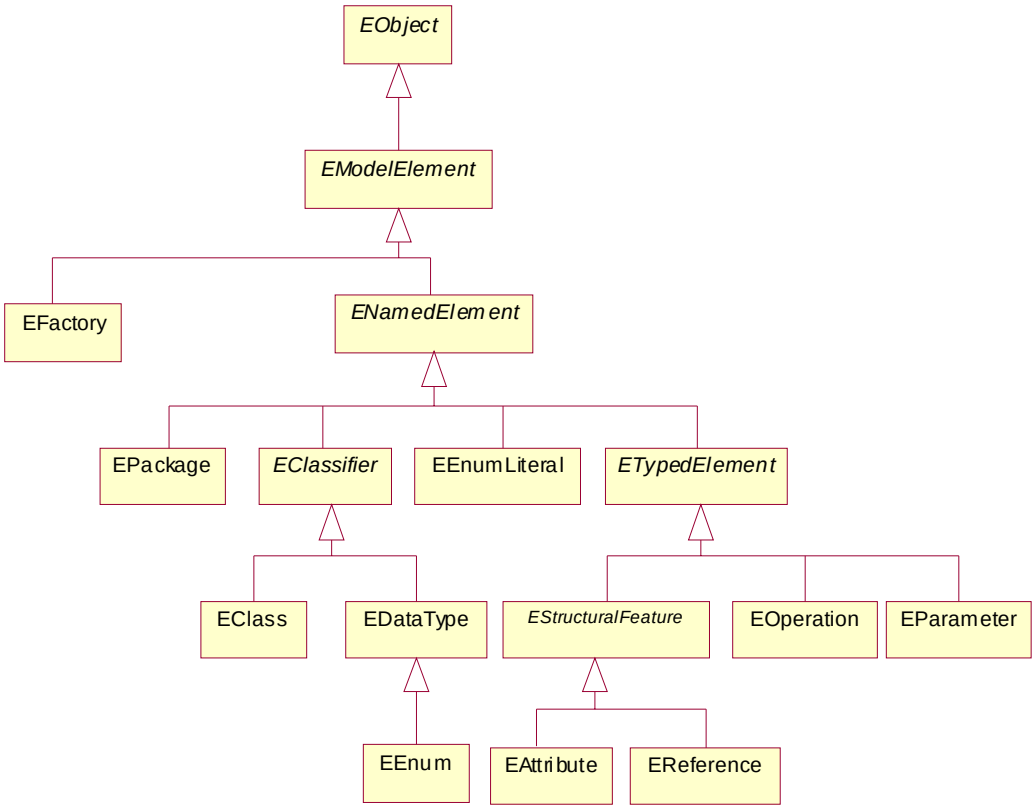

Figura 27 – Modelo ECore

Con cualquiera de la tres alternativas mencionadas anteriormente se utiliza un wizard para transformar el modelo en un modelo ecore y un modelo generator. Este último permite generar el código de implementación JAVA correspondiente

En la figura 28 se presenta como EMF unifica las tres tecnologías: XML, Java y UML.

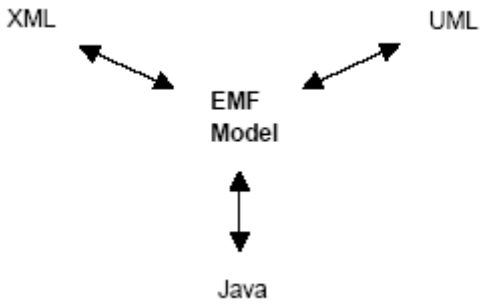

Figura 28 –EMF unifica Java, XML y UML

En nuestro caso en particular se utilizó EMF para generar los metamodelos UML y OCL y al paquete core, presentado en el capitulo 2, a partir de un diagrama de clase UML. En la figura 29 se presenta un pequeño modelo MOF que se utiliza para ilustrar el modelo ecore correspondiente.

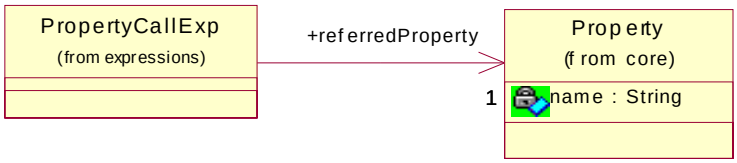

Figura 29 – Modelo UML utilizado para instanciar el modelo ecore

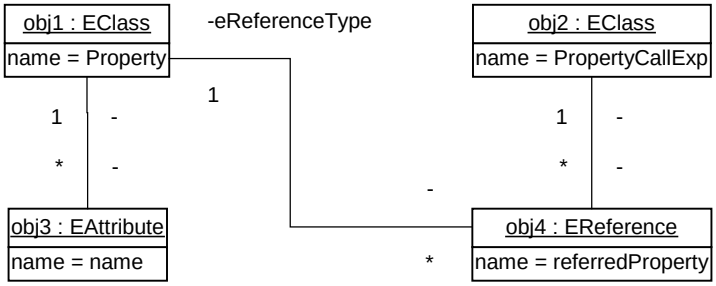

Figura 30 – Modelo ecore correspondiente al modelo UML de la figura 29

Por último la interfaz de EObject da soporte a los mecanismos de reflexivilidad para manipular las instancias que lo equipara a MOF:

```
public interface EObject {

    Object eGet(EStructuralFeature feature);
    Void aSet(EStructuralFeature feature, Object newValue);

    boolean eIsSet(EStructuralFeature feature);
    void eUnset(EStructuralFeature feature);

    ...

}
```

Los métodos especificados anteriormente son una opción para leer y escribir el modelo. A continuación se muestra un fragmento de código que instancia una propiedad con el valor *nombre* para el atributo name:

```
Property aProperty = CoreFactoryImpl.eINSTANCE.createProperty();
EAttribute name = CorePackageImpl.eINSTANCE.getProperty_Name();
aProperty.eSet(name, “nombre”)
```

Para instanciar los elementos del modelo se utiliza una EFactory. Luego es posible manipular los objetos creados independientemente del metamodelo.

6.3. Descripción de los módulos de ePlatero

Para la implementación de la herramienta que se describe a continuación, se re-utilizó algunos componentes de PAMPERO [24].

La sección 6.3.1 describe el analizador léxico y sintáctico. En la sección 6.3.2 el editor de fórmulas OCL . En la sección 6.3.3 se describe el evaluador de OCL . En la sección 6.3.4 se describe el evaluador de las condiciones de refinamiento. En la sección 6.3.5 se describe el generador de micro-mundos.

6.3.1. Descripción del 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:

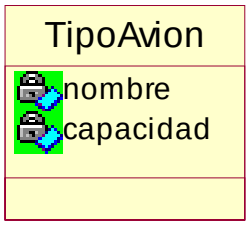

Figura 31 – Especificación de la clase TipoAvion

Dada la siguiente expresión OCL:

$self.capacidad > 0$

El analizador léxico de OCL produciría 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 [**20**] define la sintaxis abstracta (AS), concreta (CS) y la transformación de la CS a la AS. La sintaxis abstracta fue presentada en el capitulo 4. La sintaxis concreta permite a los modeladores escribir expresiones en forma textual.

En [**5**] se representan los conceptos mencionados anteriormente en la arquitectura MOF (figura 32).

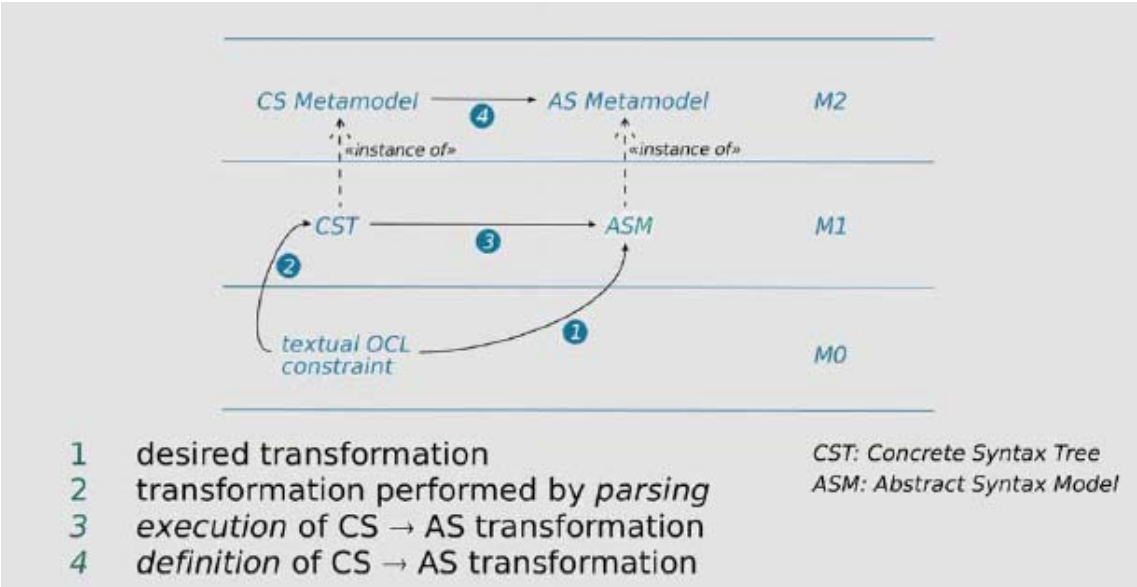

Figura 32 –Capas de la arquitectura MOF en OCL

A partir de la gramática de OCL 2.0 (anexo 10.2.) se construyó el metamodelo de la sintaxis concreta. A continuación se representan las clases que definen los literales a partir de un fragmento de la gramática de OCL 2.0:

<literal> ::= <primitive-literal> | <collection-literal> | <tuple-literal>
<primitive-literal> ::= <boolean-literal> | <integer-literal> | <real-literal>
| <string-literal> | null
<boolean-literal> ::= true | false
<integer-literal> ::= 0..9 { 0..9 }
<real-literal> ::= <integer-literal>.<integer-literal>[(e | E)[-]<integer-literal>]
<string-literal> ::= '{<characters>}'
<identifier> ::= (<letter> | _) {<letter> | _ | <digit>}
<path-name> ::= <identifier> | <path-name>::<identifier>

Los literales OCL son representados por una clase abstracta Literal, y sus subclases son PrimitiveLiteral, CollectionLiteral y TupleLiteral. La clase PrimitiveLiteral es abstracta y sus subclases son BooleanLiteral, StringLiteral, IntegerLiteral, RealLiteral y Null.

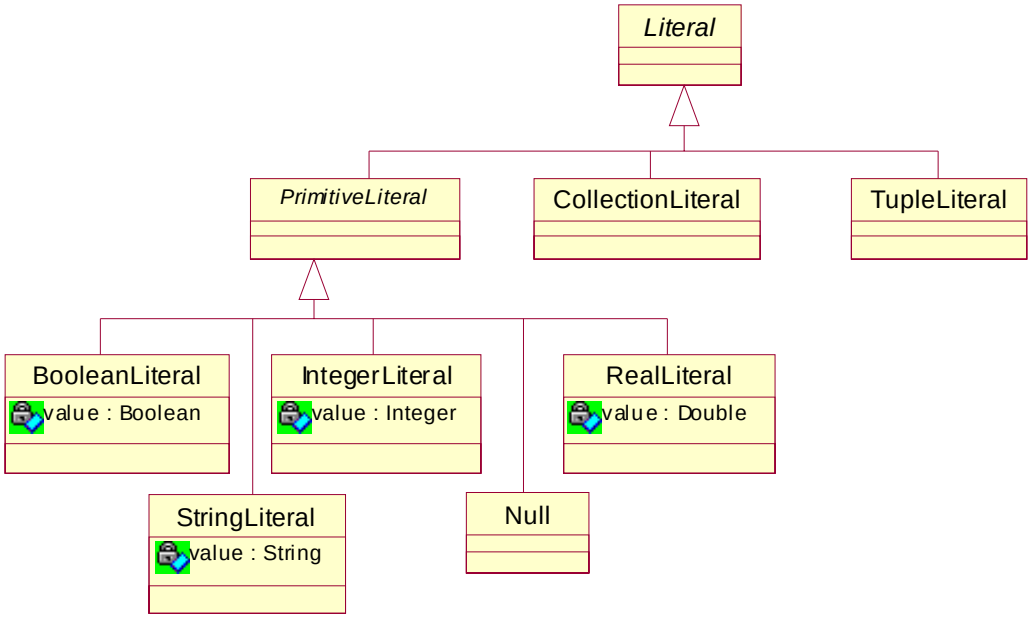

Figura 33 – Metamodelo de la sintaxis concreta para los literales

Para desarrollar el parser se utilizó *Java Compiler Compiler (JavaCC)* [**35**] que es un generador de parser que se puede utilizar en las aplicaciones Java. En el anexo 10.3. se describe la estructura del archivo ".jj" que se emplea para generar el parser y en el anexo 10.2. se describe la gramática de OCL 2.0 que se puede utilizar para generar un parser. La arquitectura del analizador sintáctico de ePlatero es similar a la presentada en [**5**]:

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 [**11**] 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. En la figura 34 se presenta la especificación de la clase Environment propuesta en [**1**], a diferencia de la especificación estándar [**20**] se añado el método addVariable el cual evita tener que construir la Variable antes de agregarla al ambiente.

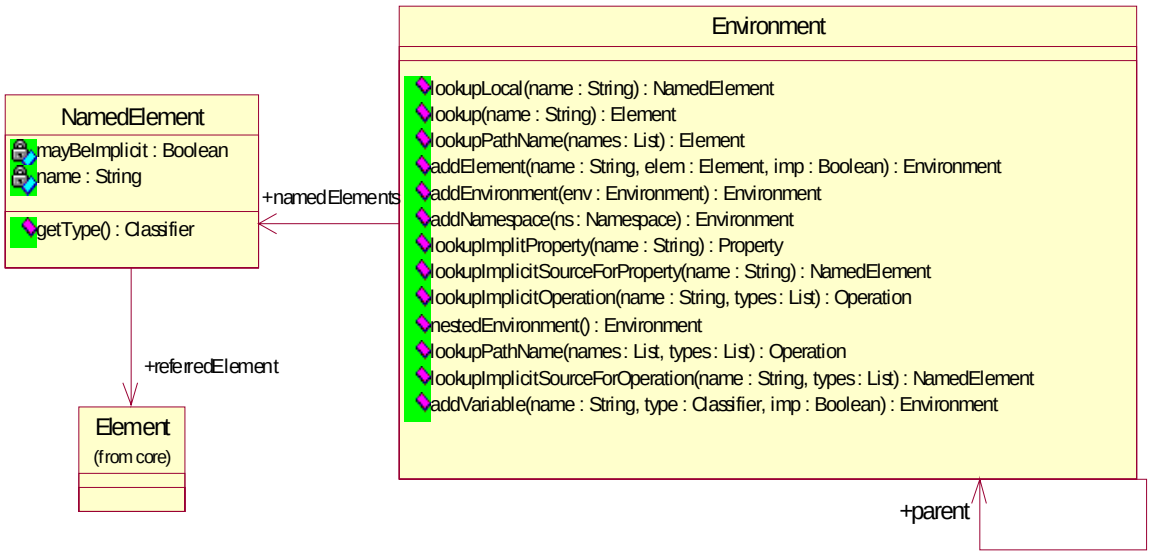

Figura 34 – Especificación de la clase Environment

En la figura 35 se ilustra la especificación de la clase Namespace con las nuevas operaciones que permiten recuperar el contenido del objeto receptor por medio del Environment.

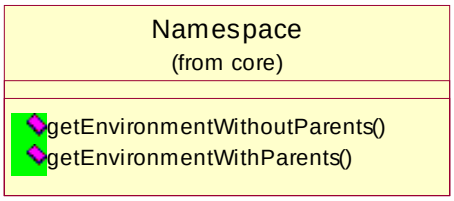

Figura 35 – Especificación de la clase Namespace

En la figura 36 se presenta la especificación de la clase Classifier con las nuevas operaciones definidas en [**20**], que se utilizan para la transformación del árbol de la sintaxis concreta al modelo de la sintaxis abstracta.

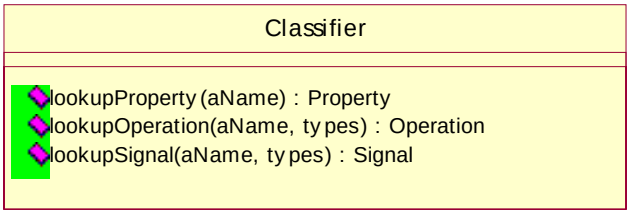

Figura 36 – Especificación de la clase Classifier

Continuando con el ejemplo iniciado en la presentación del analizador léxico se ilustrará el proceso descrito en esta sección. En la figura 37 se presenta el árbol de la sintaxis concreta generado por el OclParser.

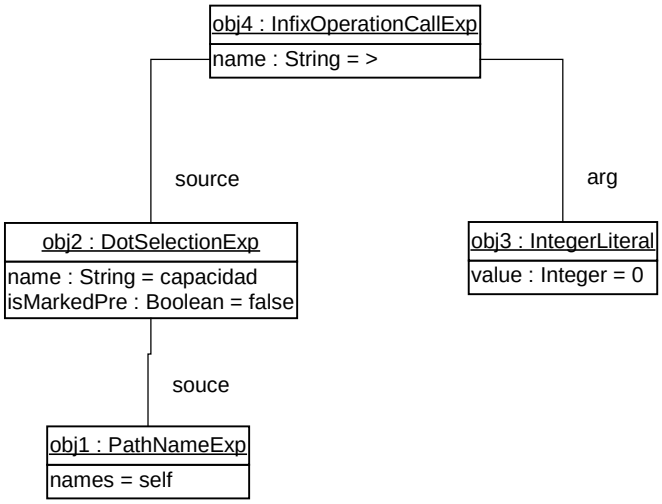

Figura 37 – Ejemplo de un árbol de sintaxis concreta

La instancia de DotSelectionExp (figura 37) representa la invocación de la propiedad capacidad. La gramática definida en [**20**] es ambigua ya que por ejemplo no es posible determinar si dicha invocación corresponde a un PropertyCall o IteratorExp, ambos elementos de la sintaxis abstracta contienen la misma producción ("*oclExpression '.' simpleName*"). Otro problema de la sintaxis concreta presentada en [**20**] es que no hay una definición precisa de nombres, por ejemplo, en esta etapa de análisis no es posible determinar si "self" representa una invocación de variable o propiedad (propiedad implícita). Por tal motivo la llamada a la variable se representa como un PathNameExp. Estos inconvenientes son solucionados definiendo una gramática no ambigua y separando la fases de un parser (análisis léxico, sintáctico y el análisis sensible al contexto) .

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 38).

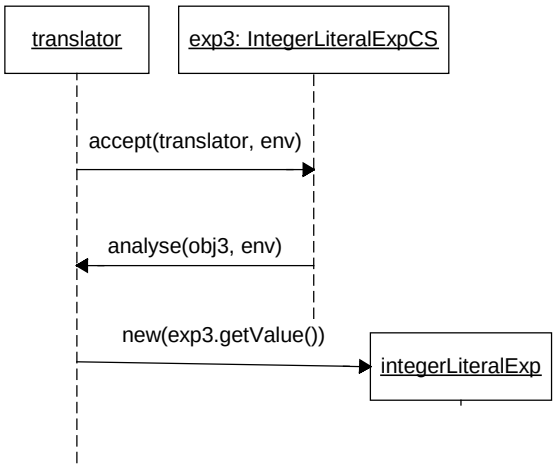

Figura 38 – Traducción de IntegerLiteralExpCS a IntegerLiteralExpAS

En este contexto el objeto obj1 (PathNameExp) se transforma en una VariableExp (figura 39).

Referencia:

1. Una vez que se obtiene el elemento del ambiente se puede determinar la traducción del objeto obj1.

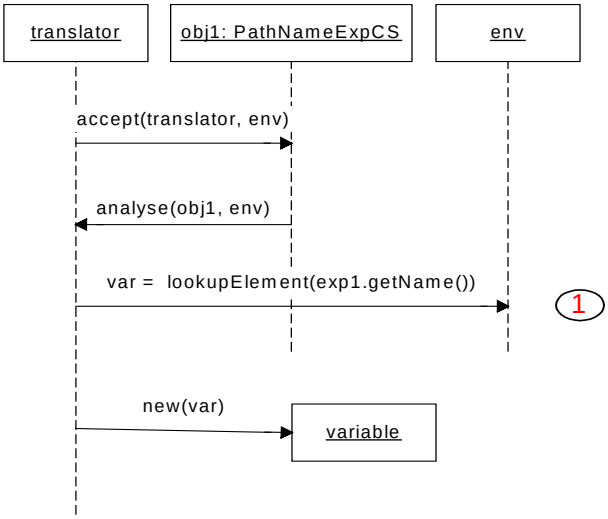

Figura 39 – Traducción de PathNameExpCS a VariableExpAS

En este contexto el objeto obj1 (DotSelectionExp) se traduce como un PropertyCallExp (figura 40).

Referencias:

1. En base al tipo de la propiedad se determina la expresión resultante.

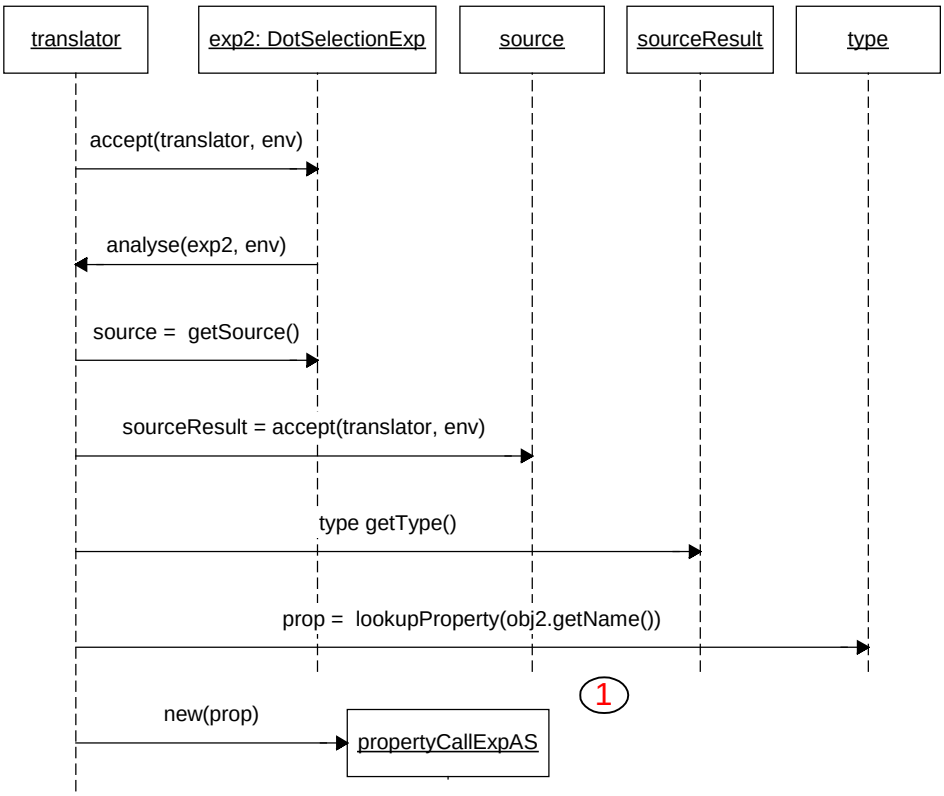

Figura 40 – Traducción de DotSelectionExpCS a PropertyCallExpAS

Por último se transforma el objeto obj4 (InfixOperationCallExp) en un OperationCallExp (figura 41) .

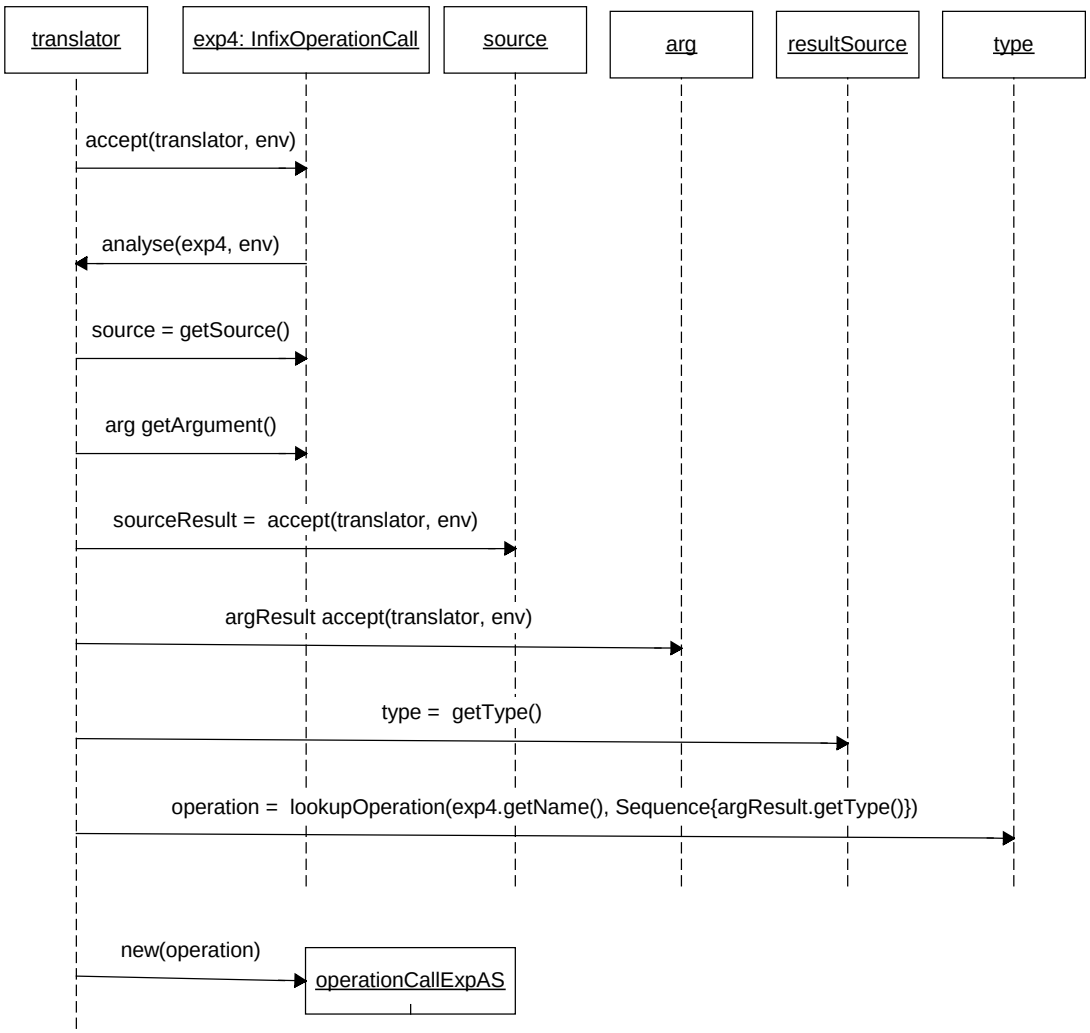

Figura 41 – Traducción de InfixOperationExpCS a OperationCallExpAS

6.3.2. Descripción del editor de fórmulas OCL

El editor de OCL (figura 42) 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.

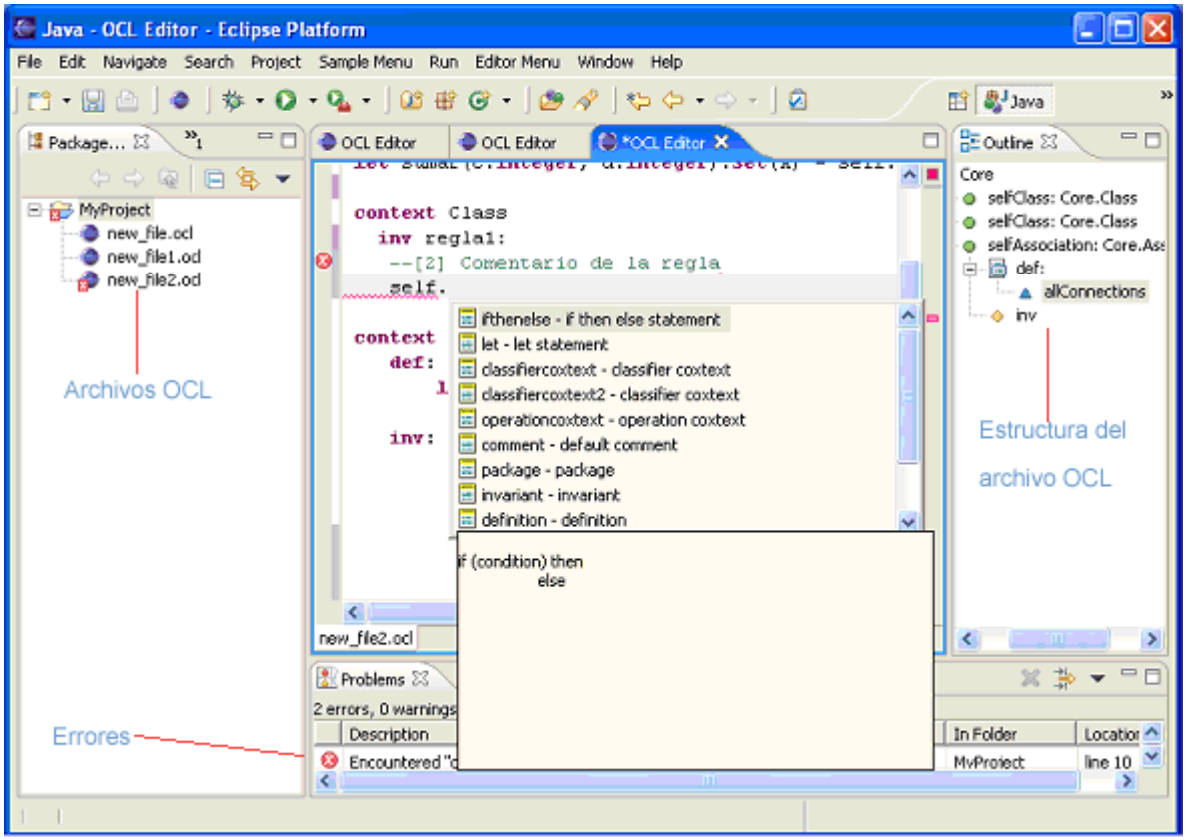

Figura 42 –Editor de fórmulas OCL

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 43 se ilustra 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.

A continuación se describen las responsabilidades o roles de las clases e interfaces asociadas al AbstractTextEditor (figura 43):

- **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 interface del editor a los datos o 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.

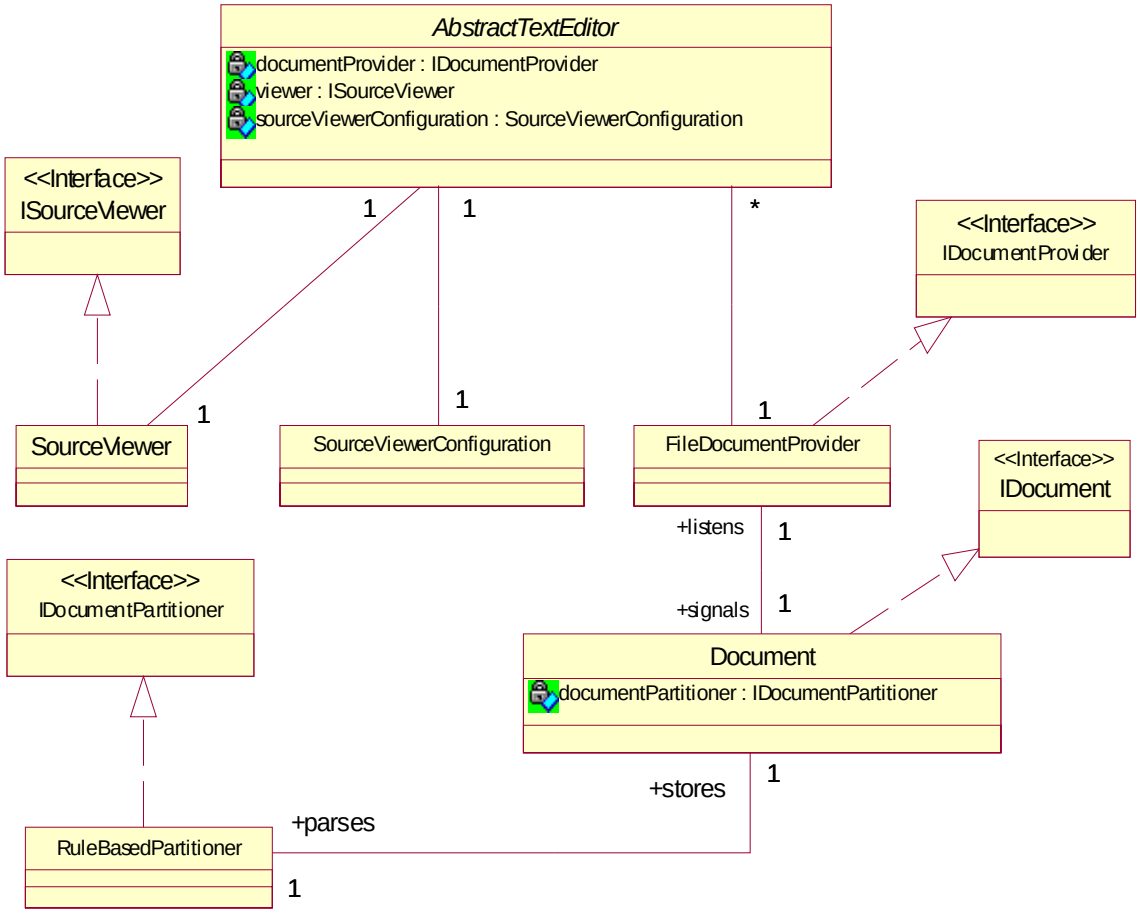

Figura 43 – Relaciones de la clase AbstractTextEditor

Relación Model-View-Controller

El document provider actúa como un intermediario entre el documento y el editor, como se presenta en la figura 44. Los editores con el mismo documento comparten el mismo document provider esto es lo que permite la actualización automática de lo 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.

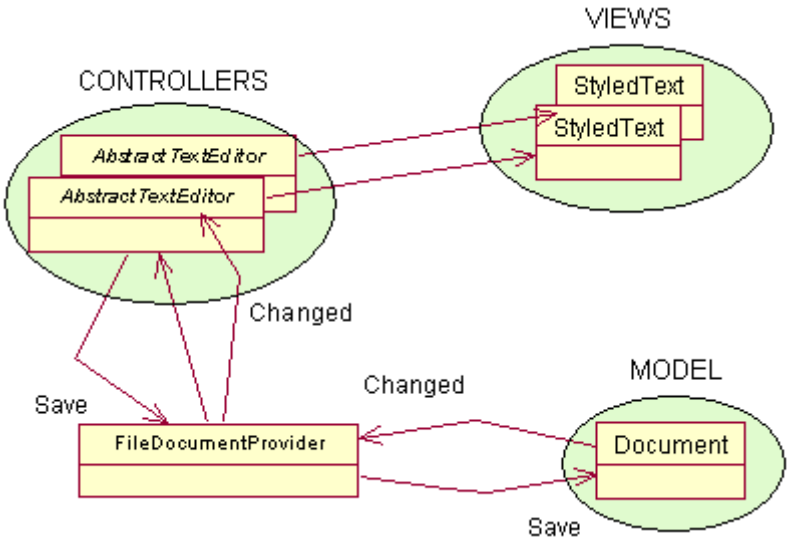

Figura 44 – Relación de Model y View en JFaceText

Diseño del editor OCL

En la figura 45 se presenta el diseño básico del editor de fórmulas OCL, el cual se instancia por especialización del framework JfaceText.

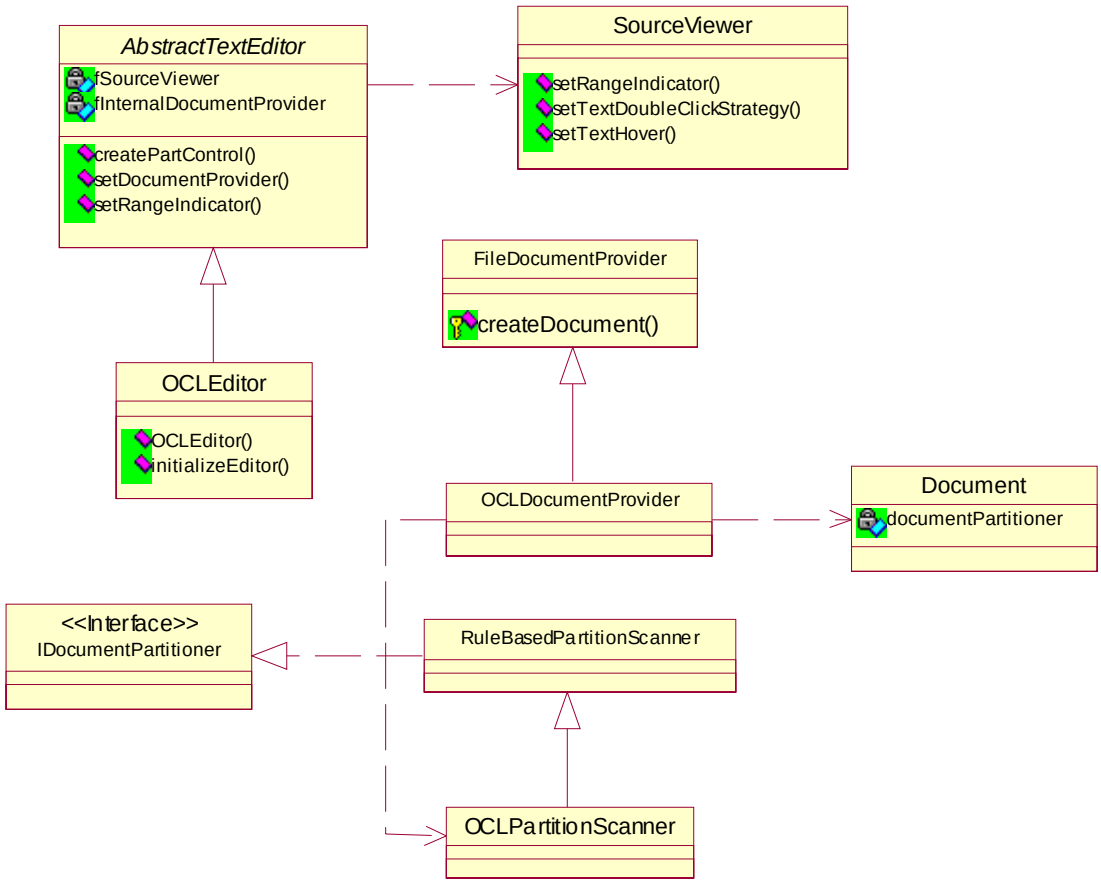

Figura 45 – Diseño básico del editor de fórmulas OCL

6.3.3. 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 meta tipos, es decir evalúa los invariantes del modelo y metamodelo. En la figura 46 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.

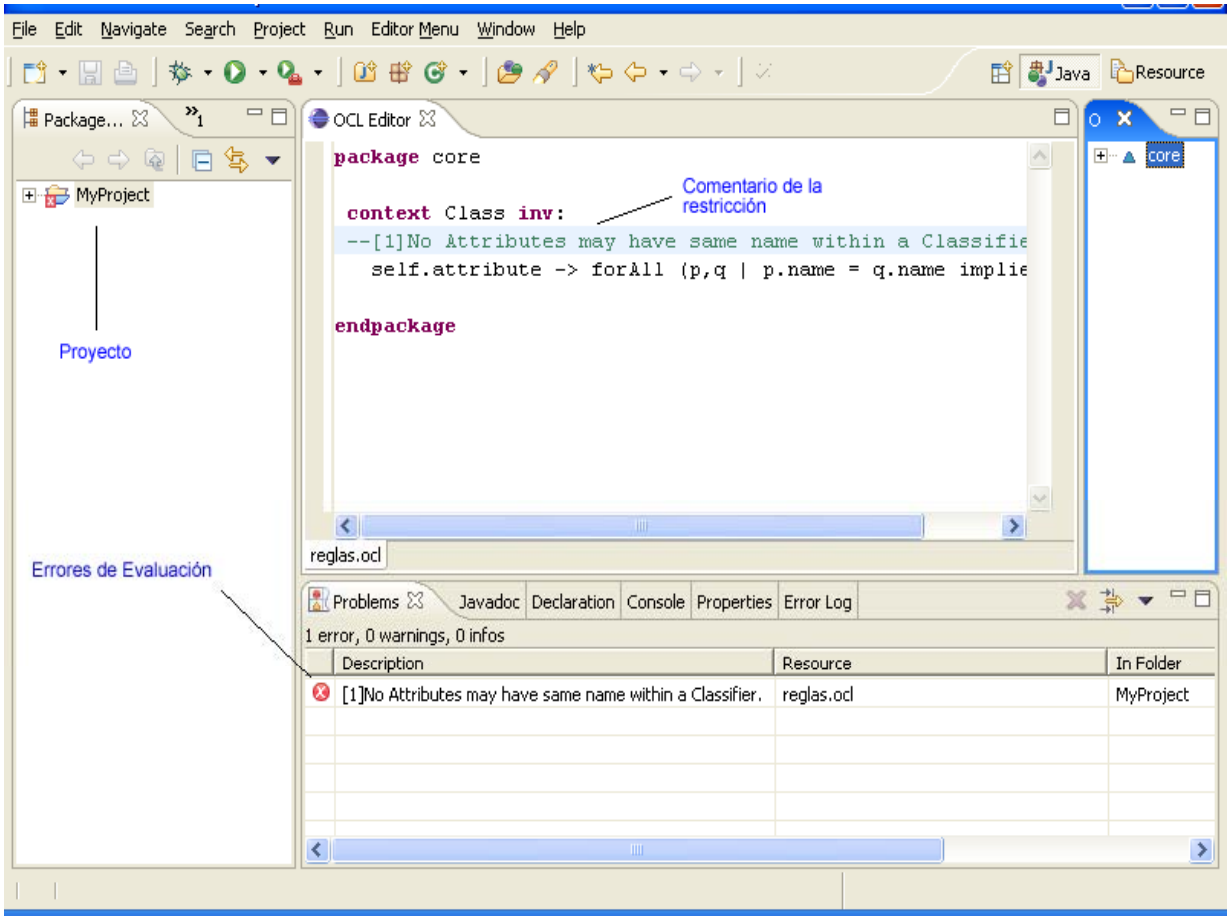

Figura 46 – Visualización de errores semánticos

Diseño del Evaluador de OCL

Paquetes

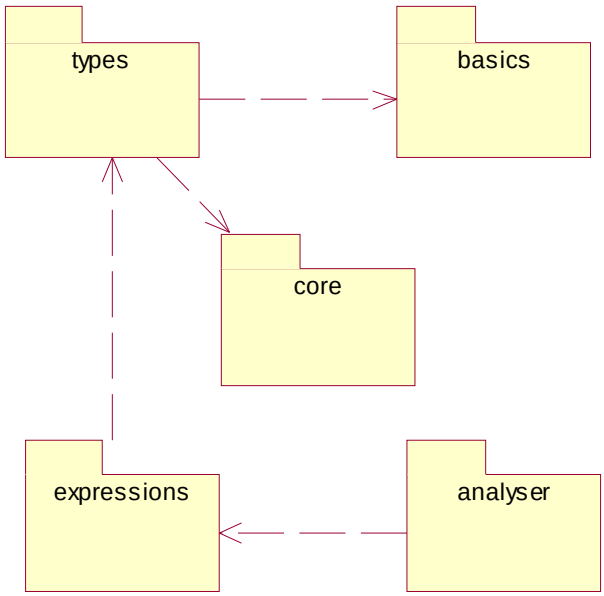

Figura 47 – Diagrama de paquetes del evaluador de OCL

Descripción

En la figura 47 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, contienen los elementos del modelo comunes a todo metamodelo MOF, y provee la información contextual para la evaluación de expresiones OCL. El paquete de basics contienen 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 la reglas semánticas. A continuación se describen las clases de los dos nuevos paquetes.

Estructura del paquete basics

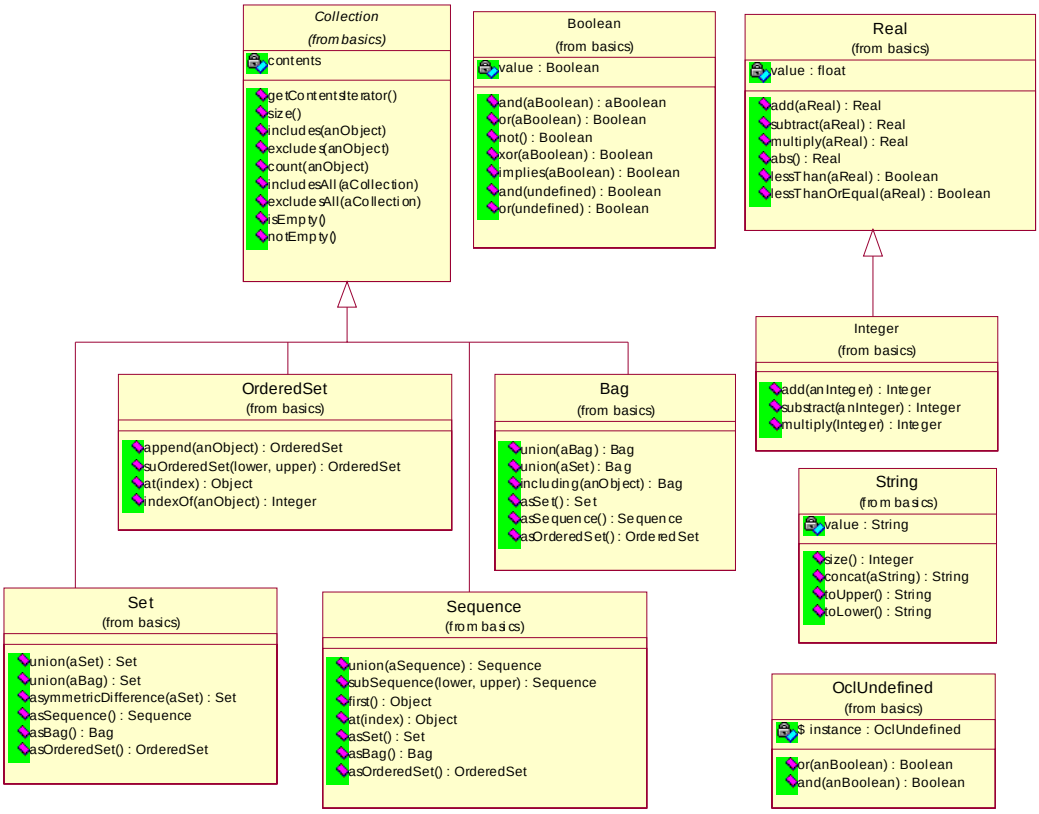

Figura 48 – Diagrama de clase del paquete basics

Participantes y responsabilidades 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 estandar de OCL. Para simplificar el diagrama solo se incluye algunas operaciones, en el anexo 10.4. se encuentran todas las operaciones definidas en la especificación de OCL 2.0. La jerarquía de las colecciones está encabezada por la clase abstracta Collection y define las operaciones comunes a todas las colecciones. Las subclases Set, OrderedSet, 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 ejecudas por el evaluador ya que requiere la evaluación de expresiones. La jerarquia 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 vacia. Por último se incluye otros tipos básicos como son los String y Boolean.

Estructura del analyser

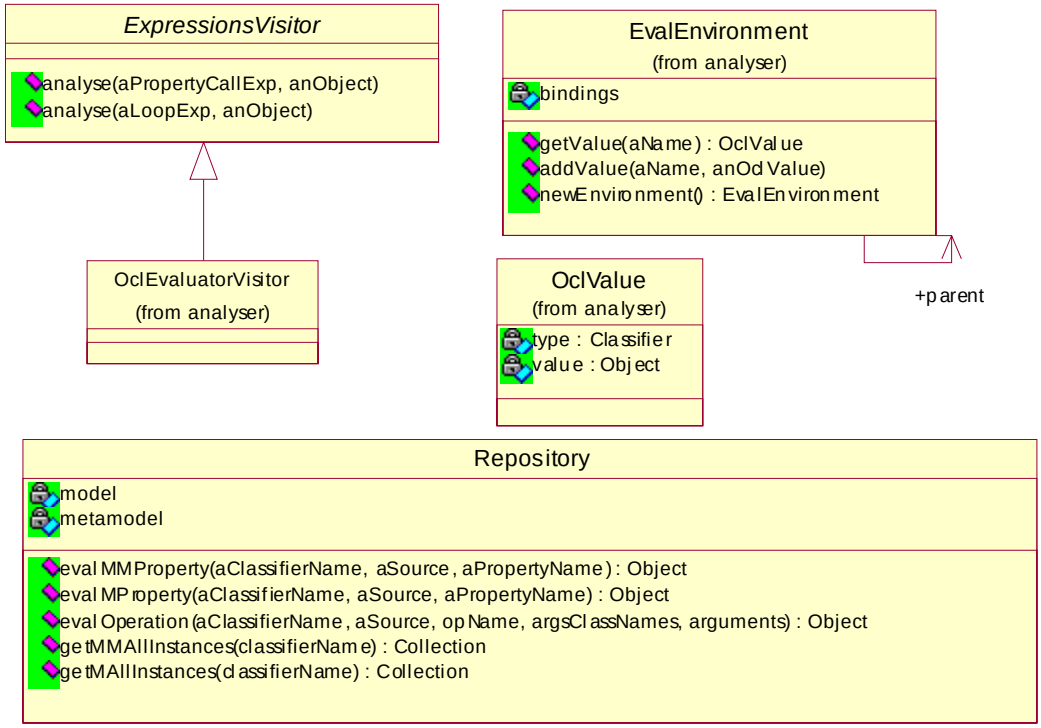

Figura 49 – Diagrama de clase del analizador semántico

Participantes y responsabilidades del paquete analyser

En el diagrama de la figura 49 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 [**11**], 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.

Ejemplo de Evaluación

Continuando con la expresión OCL utilizada como ejemplo en los módulos anteriores, se explicará el proceso de evaluación mediante diagramas de secuencias. El proceso se inicia con la invocación de la operación >, para ello se evalua su receptor y argumento.

self.capacidad > 0

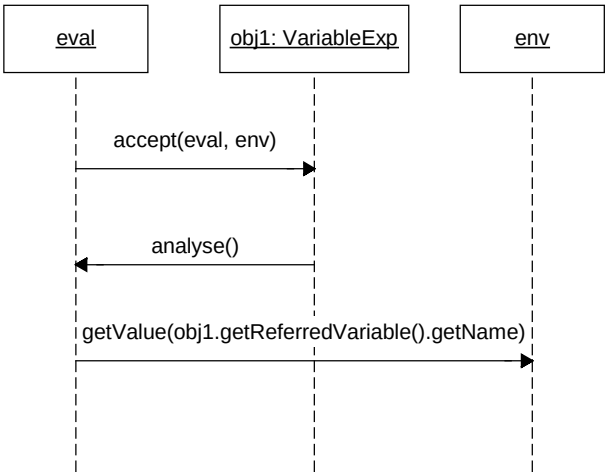

Figura 50 – Diagrama de secuencia de evaluar una invocación de variable

En la figura 50 se presenta el diagrama de secuencia de evaluar una invocación de una variable. Suponiendo, por ejemplo, que la expresión dada forma parte de un invariante el evaluador va asignado un objeto del TipoAvion del repositorio al valor de la variable contextual.

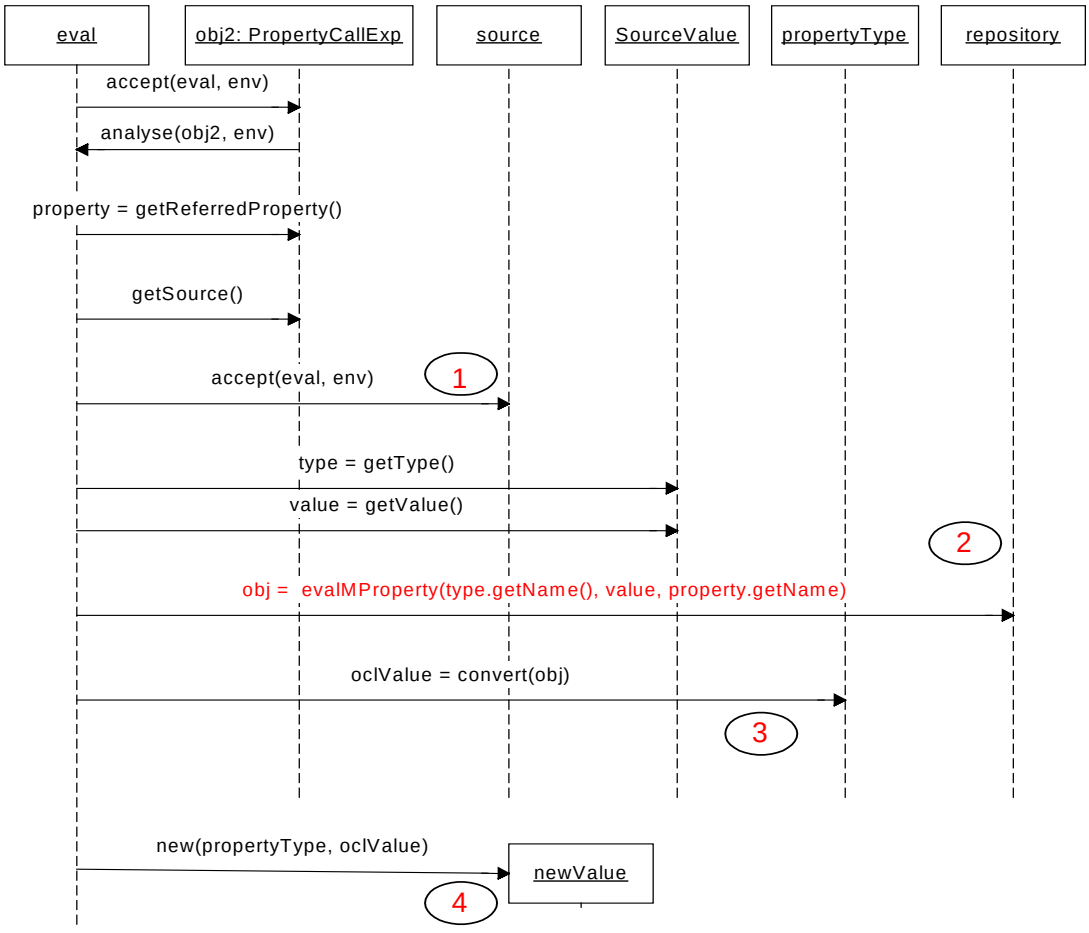

Figura 51 – Diagrama de secuencia de evaluar una invocación de una propiedad

La figura 51 presenta el diagrama de secuencia de evaluar una invocación de una propiedad (capacidad).

Referencias:

1. En este ejemplo la evaluación del source de la propiedad se indica en la figura 50.
2. Dependiendo del tipo del resultado de la evaluación del source de la invocación de la propiedad se solicita el valor al repositorio. Es decir, si el tipo es un elemento del modelo se invoca a evalMProperty, en cambio si es un meta tipo se invoca al método evalMMProperty del repositorio.
3. Convierte un objeto Java al objeto OCL correspondiente.
4. El resultado de esta evaluación consiste en un nuevo valor cuyo tipo es el tipo de la propiedad.

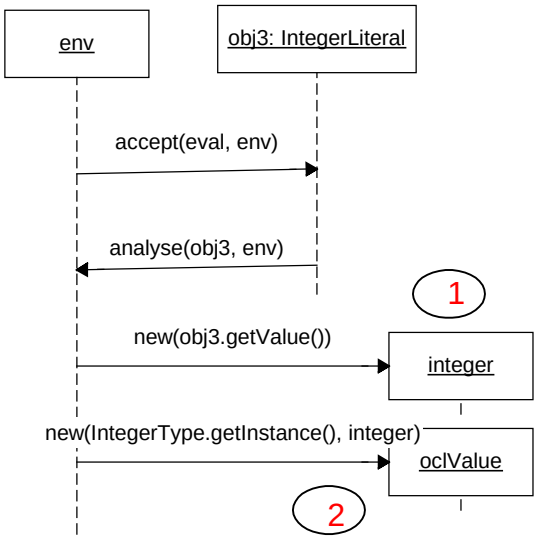

Figura 52 – Diagrama de secuencia de evaluar un literal entero

La figura 52 presenta el diagrama de secuencia de la evaluación de un literal entero (cero)

Referencias:

1. Se instancia un entero OCL (basics) cuyo valor es el valor del literal.
2. El resultado de la evaluación es un nuevo OclValue cuyo tipo es la clase Singleton [**11**] IntegerType (types) y el valor es el entero recientemente instanciado.

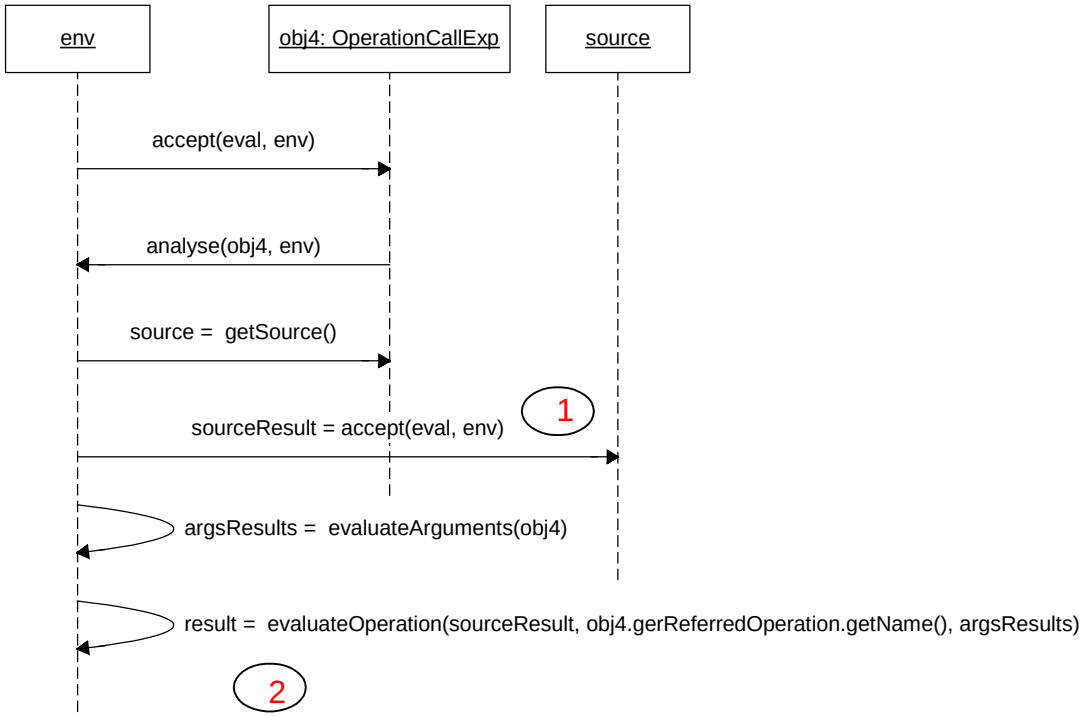

Figura 53 – Diagrama de secuencia de evaluar una invocación de una operación

La figura 53 presenta el diagrama de secuencia de la evaluación de una invocación de una operación (>).

Referencias:

1. Retorna el valor (OclValue) de evaluar el source de la invocación de la operación.
2. Ejecuta la operación e instancia el OclValue correspondiente.

6.3.4. Descripción del Evaluador de Refinamiento

Este módulo se encarga de implementar la estrategia de evaluación de refinamiento en modelos UML/OCL propuesta en esta tesis. En la figura 54 se presenta el diagrama de clase de la implementación del evaluador de refinamiento.

Estructura

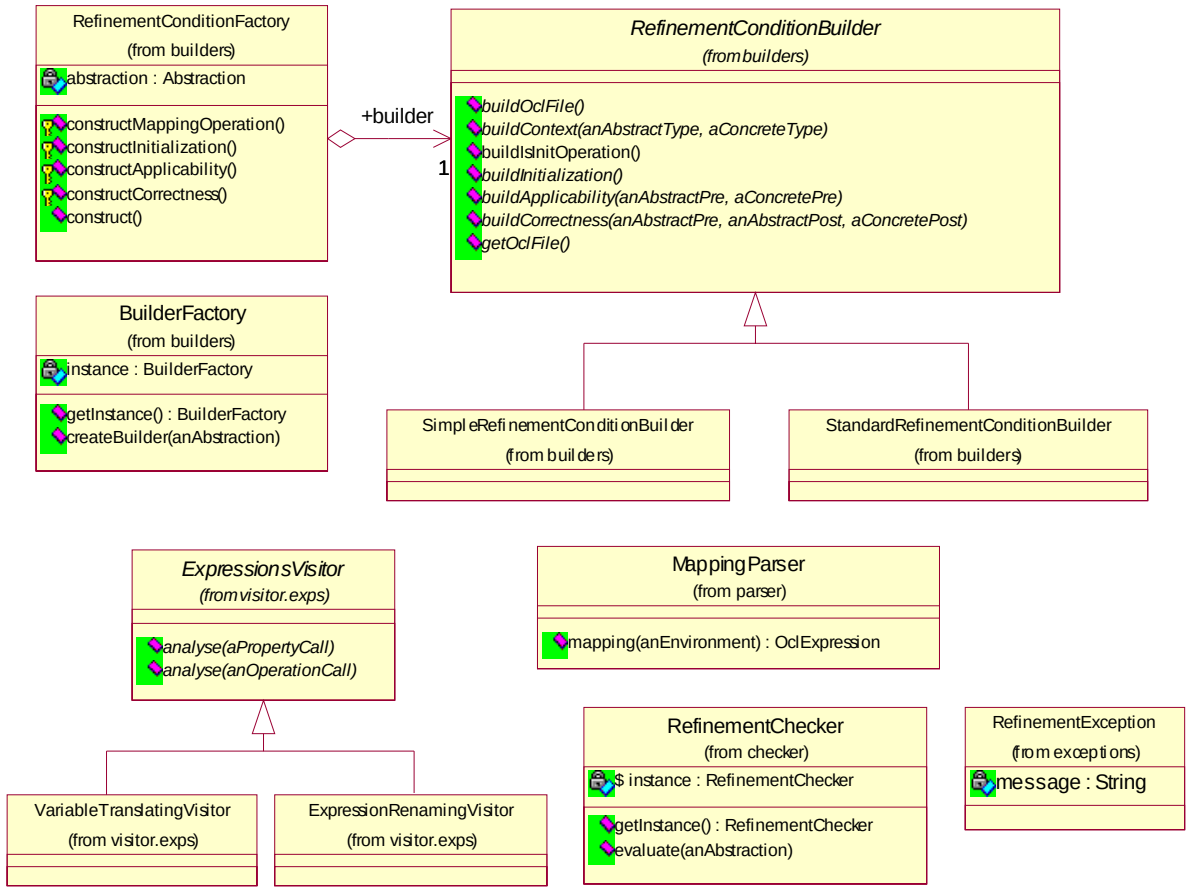

Figura 54 - Diagrama de clase de la implementación del evaluador de refinamiento

En la figura 55 se presentan algunas restricciones OCL que enriquecen al modelo ilustrado en la figura 54.

package checker

context RefinementChecker :: evaluate (anAbstraction: Abstraction)

post: let builder = BuilderFactory ^ getInstance().result() ^
createBuilder(anAbstraction) **in**
builder.oclIsNew() and factory.oclIsNew() and factory.builder = builder and
factory ^ construct().hasReturned() and builder ^ getOclFile().result()
-> forAll (c | coordinator ^ evaluate(c))

endpackage

package builders

context RefinementConditionFactory :: constructMappingOperation()

post: let newOperation: Constraint = abstraction.supplier.constraint - > select (c |
c.name = 'mapping' and c.stereotype.name = 'definition' and
c.constrainedElement = abstraction.supplier
in newOperation.oclIsNew() and newOperation.body.bodyExpression =
self ^ parse(anAbstraction).result() and
newOperation.body.parameterVariable -> size(1) and
newOperation.body.parameterVariable.asSequence().at:1.type =

```
                abstraction.client

    context BuilderFactory :: createBuilder(anAbstraction: Abstraction)
                                                        :RefinementConditionBuilder
      post: if (anAbstraction.mapping.body = '')
              result.oclIsTypeOf(SimpleRefinementConditionBuilder)
            else
              result.oclIsTypeOf(StandardRefinementConditionBuilder)
            endIf
```

La operación adicional *getPreconditions()* retorna todas las precondiciones de las operaciones definidas en el tipo del objeto receptor.

La operación adicional getPrecondition(aBehavioralFeature) a diferencia de la operación adicional anterior tiene en cuenta las operaciones con la misma signatura de la operación dada.

```
    context RefinementConditionFactory :: constructApplicability()
      post: abstraction.supplier.getPreconditions() -> forAll ( p |
              abstraction.client.getPrecondition(p) -> notEmpty implies
              builder ^ buildApplicability (p,
                  abstraction.client.getPrecondition(p.constrainedElement).hasReturned()
endpackage
```

Figura 55 –Restricciones OCL que enriquecen al modelo de la figura 54

Participantes y Responsabilidades

La clase *RefinementChecker* es una clase Singleton[**11**], tiene exactamente una instancia. Su responsabilidad es la de determinar si una abstracción dada es un refinamiento, para ello evalúa las condiciones de refinamiento planteadas en el capítulo anterior.

El objeto *RefinementConditionFactory* es el responsable de instanciar las condiciones de refinamiento utilizando la interfaz de *RefinementConditionBuilder*. En esta sección al conjunto de las restricciones a ser evaluadas por el *RefinementChecker* se denomina *archivo OCL.* El cual 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 [**11**]. 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 puede 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 MappingParser 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 un excepción RefinementException la cual conoce el mensaje del error.

Las operaciones realizadas sobre las expresiones en OCL están modeladas como una clase de la jerarquía encabezada por la clase abstracta *ExpressionsVisitor (* patrón Visitor [**11**] *)*. La clase *VariableTranslatingVisitor* y *ExpressionRenamingVisitor* son utilizadas en el momento de instanciar las condiciones de aplicabilidad y correctitud. La primera, es la responsable de renombrar la variable contextual (self) de las precondiciones y postcondiciones por la variable que tiene el nombre de la clase (en minúscula) que define la operación. La segunda, es la responsable de añadir una decoración ("_post") a cada identificador no decorado y elimina la decoración original (@pre) del resto de los identificadores que forman parte de la postcondición.

En la figura 56 se presenta un diagrama de secuencia de la operación que verifica si una abstracción dada cumple con las condiciones de refinamiento.

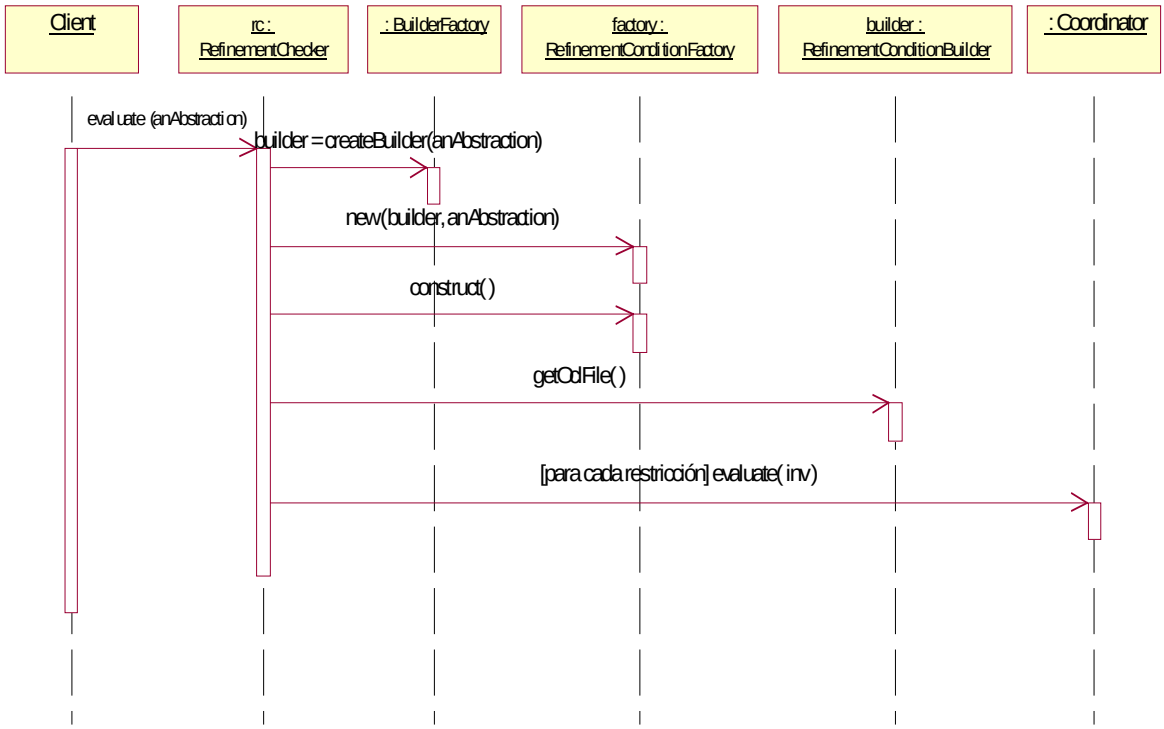

Figura 56 - Diagrama de secuencia de evaluar un refinamiento

En la figura 57 se presenta un diagrama de secuencia de la operación que construye el archivo OCL, el cual contiene las restricciones OCL a ser evaluadas por el evaluador de refinamiento. Para simplificar el diagrama se evita especificar las operaciones privadas de la clase *RefinementConditionFactory.*

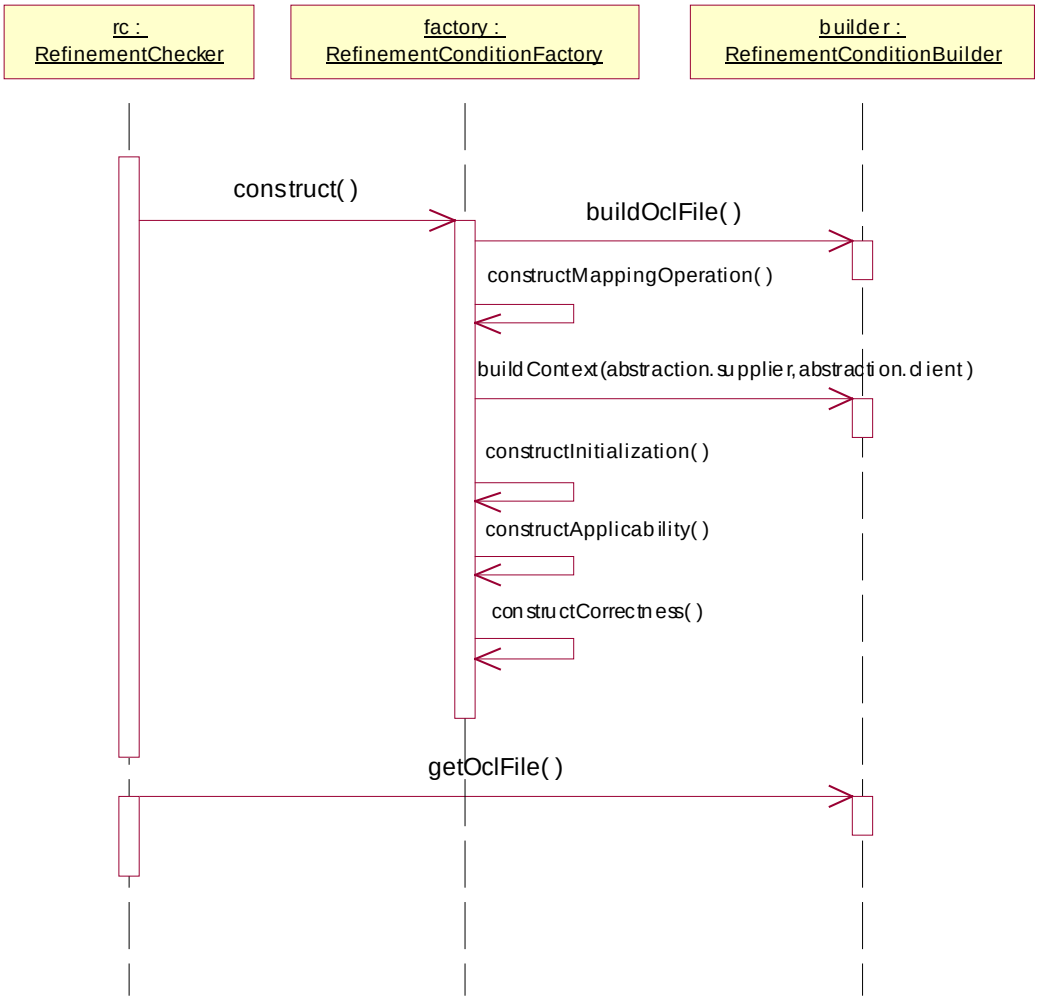

Figura 57 - Diagrama de secuencia del armado del OCLFile

6.3.5. Descripción del generador de Micro-Mundos

El generador de micro-mundos 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 5. Para generar los micro-mundos 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.

El diseño de este módulo es muy sencillo, consta de un MicroWorldFactory el cual es el responsable de instanciar los n objetos para los Classifiers que están involucrados en la abstracción. Además esta provee el mapping (en el caso de los patrones de refinamiento estructurales) que se utiliza para generar la condición de dualidad. El MappingParser es el responsable de parsear el mapping de la abstracción dada. El proceso de generación de micro-mundos crea automáticamente la propiedad de dualidad (si se especificó un mapping) que es necesaria para que los micro-mundos sean válidos para la evaluación de refinamiento. Para restaurar el estado de la especificación concreta se utiliza el patrón de diseño Memento [**11**], ya que se generan restricciones adicionales que son solamente útiles para el proceso de generación de micro-mundos.

El OclDomainAnalyser es el objeto responsable de visitar los invariantes especificados en un determinado Classifier y retornar el conjunto de los objetos de tipo primitivo (si se han definido). Para los tipos primitivos de dominio infinito se deben restringir con expresiones OCL (Integer, Real, String). El MicroWorldFactory obtiene de forma aleatoria los valores de los slots para

cada una de las instancias generadas. El evaluador de OCL colabora con la fábrica de micro-mundos para asegurar que las instancias respetan las restricciones especificadas por el usuario. Si en el modelo se establece una relación de composición se asegura que todos los componentes son referenciados por un solo contenedor.

6.4. Ejemplos

En esta sección se muestra como se comporta la herramienta desarrollada mediante un pequeño ejemplo (figura 58), solo se los emplea a fines explicativos. Los viajes tienen asignado al menos un piloto, y estos tienen una política de bonificación (patrón de diseño Strategy[**11**]), que puede ser estándar o estrella.

Figura 58 - Diagrama de clase de ejemplo

Para poder entender como funciona el evaluador, se modifica la clase Vuelo que se especifica en el diagrama de clases de la figura 58 agregando una operación con dos parámetros con el mismo nombre. Cuando se evalúa la regla de buena formación (well-formedness rules) presentada en la figura 59 se observa en la figura 60 que aparece un error o una entrada en la Vista de Problemas (View of Problems).

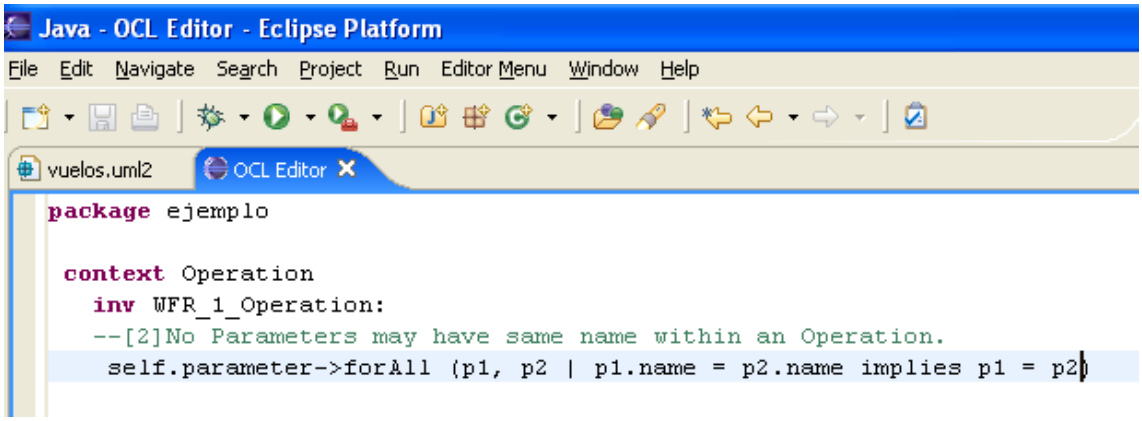

Figura 59 –Regla de buena formación

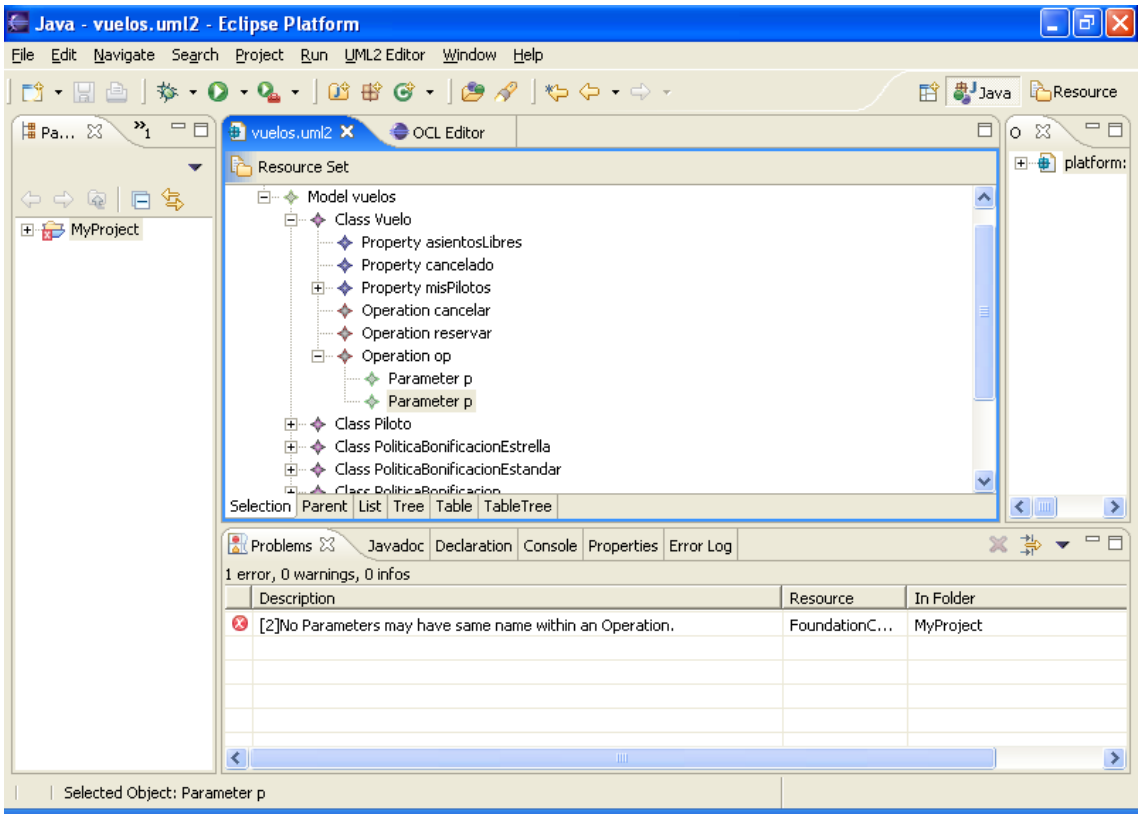

Figura 60 – Validación de la regla de la figura 59

Ahora vamos a evaluar la regla de diseño que establece que las clases del modelo no deben tener demasiadas responsabilidades. Para explorar un caso donde no se cumple con dicha restricción se especifica que toda las clases deben tener menos de dos operaciones . Cuando se evalúa la regla se observa en la figura 61 un error. Si bien el ejemplo es muy simple se puede utilizar la herramienta como soporte para un programa de métricas. De este modo ePlatero ayuda a encontrar debilidades en los diseños.

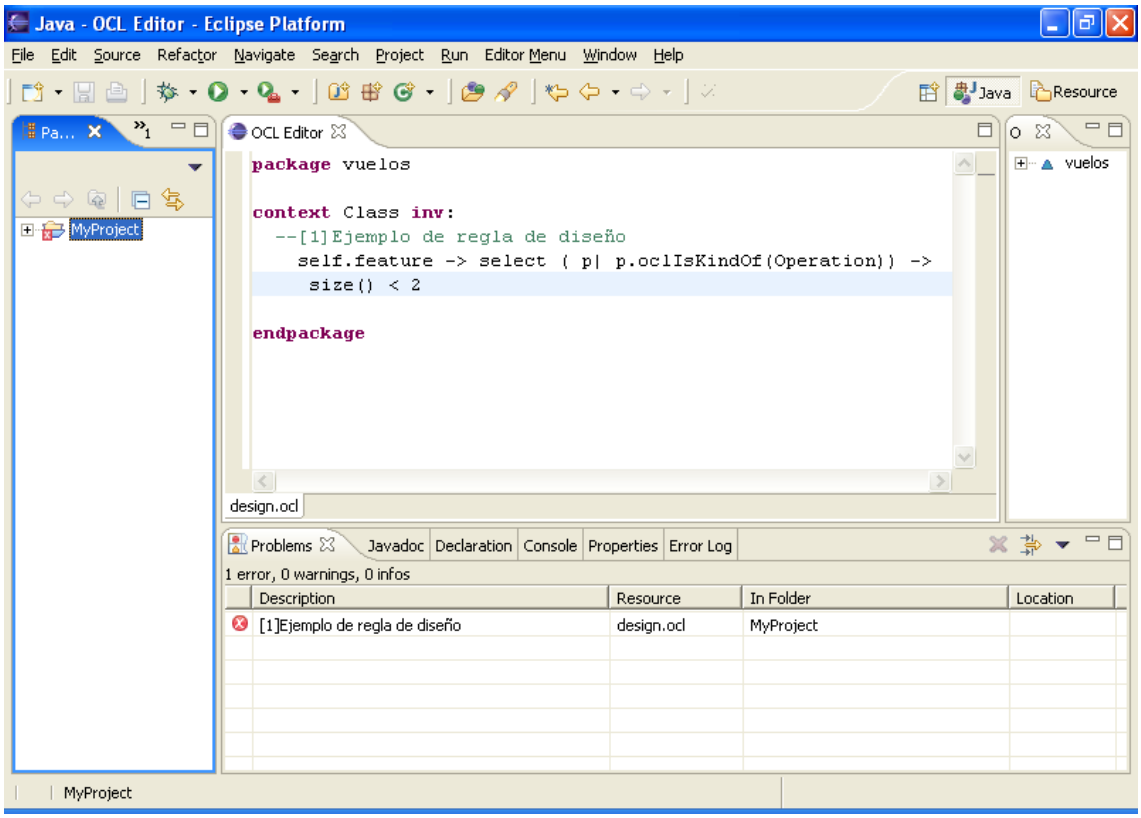

Figura 61 – Validación de la regla de diseño

Para ilustrar el funcionamiento del evaluador de refinamiento y generador de micro-mundos se utiliza el ejemplo del patrón de refinamiento presentado en el capítulo 5 (figura 17). El primer paso para generar el micro-mundo consiste en definir los invariantes del modelo y las restricciones de los dominios (figura 62).

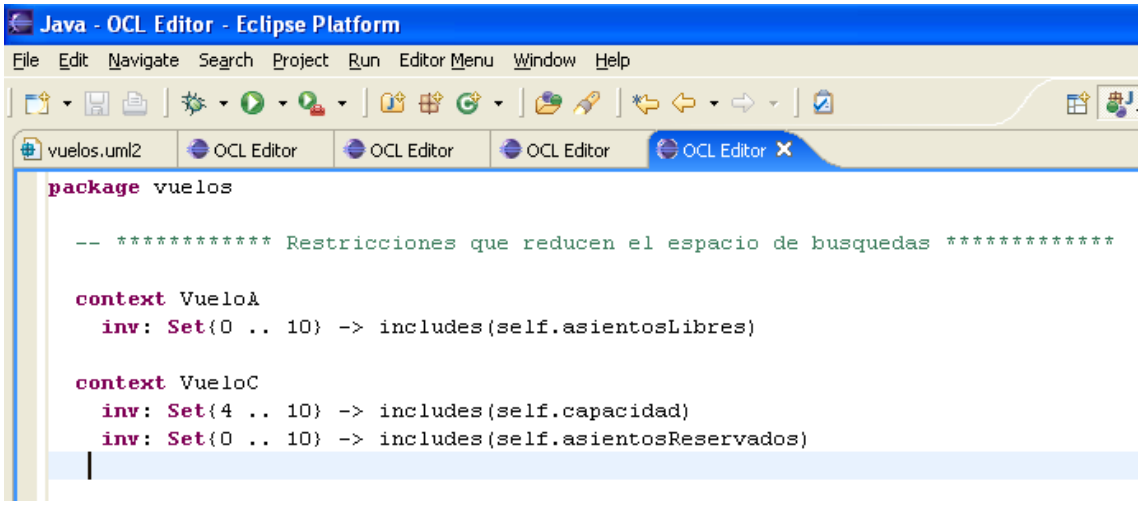

Figura 62 – Restricciones que reducen el espacio de búsquedas especificadas en ePlatero

En la figura 63 se ilustra el menu contextual y el micro-mundo generado automáticamente por la herramienta. Tal micro-mundo además de cumplir con los invariantes especificados en la figura 62 respeta la condición de dualidad.

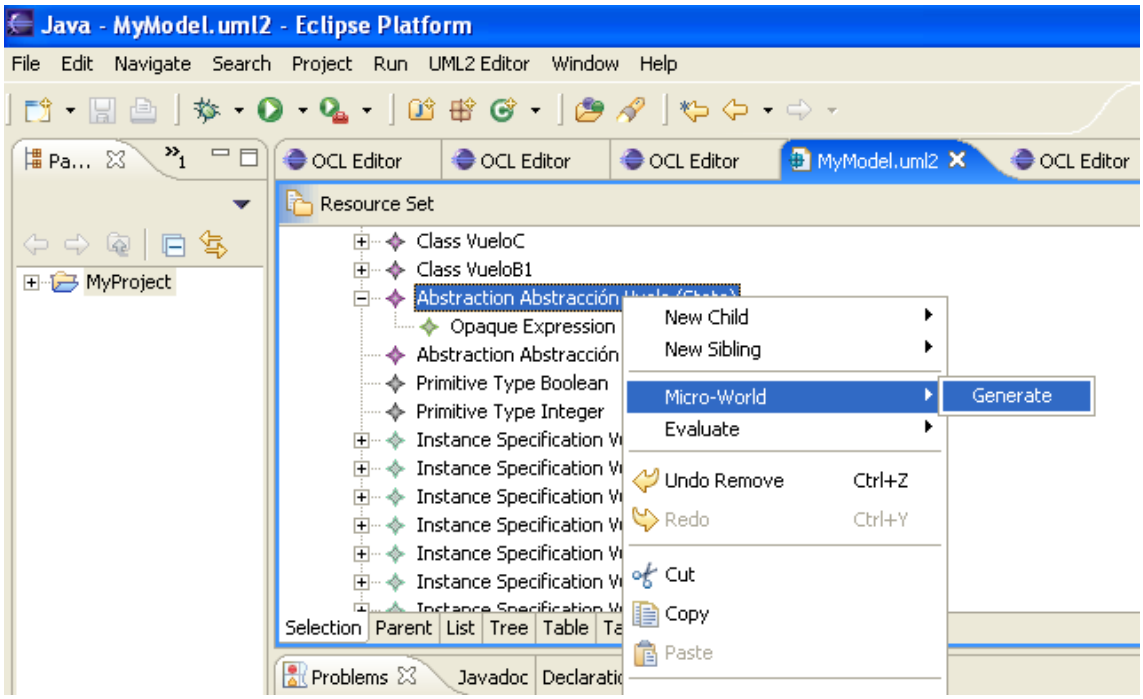

Figura 63 – Micro-mundo generado por ePlatero

Una vez que las restricciones de la especificación abstracta y concreta (ilustradas en el capítulo anterior) son parseadas es posible evaluar el refinamiento (figura 64).

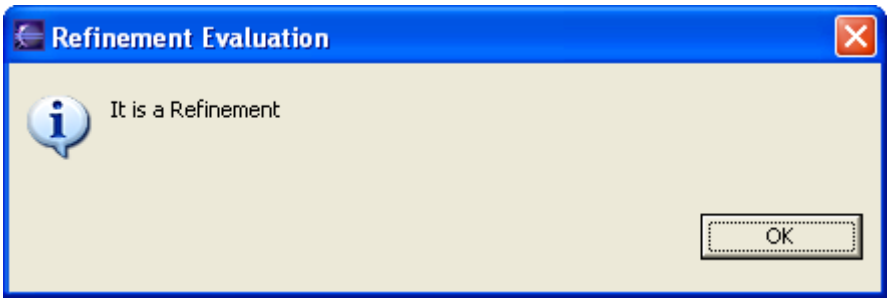

Figura 64 – Evaluación satisfactoria de refinamiento

En la figura 65 se ilustra un caso donde el mapping de la abstracción es sintacticamente incorrecto, el cual es reportado en el momento de la evaluación del respectivo refinamiento.

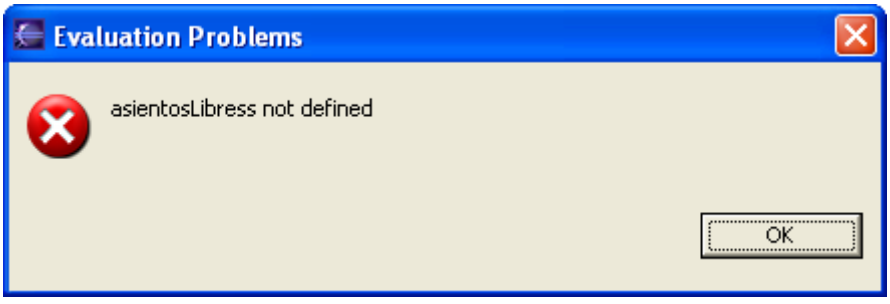

Figura 65 – Mapping sintacticamente incorrecto

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

```
context VueloC::reservar()
    pre: self.capacidad - self.asientosReservados < 0 and not
          self.cancelado
```

En la figura 66 se ilustra el resultado de evaluar la condición de refinamiento. El evaluador informa la primera condición que no se satisface.

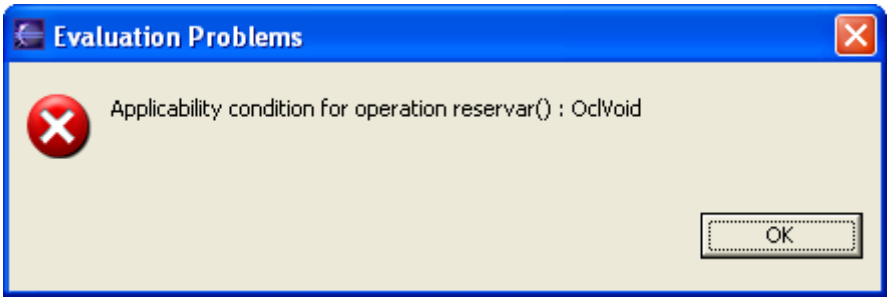

Figura 66 – Evaluación no satisfactoria de refinamiento

7. Trabajos Relacionados

Una de las primeras herramientas que soportan OCL es USE (UML based Specification Environment) [**37**]. Esta permite especificar un modelo con una especificación USE, utiliza un subconjunto de UML para representar la estructura y comportamiento, y OCL para expresar restricciones y queries. Se puede crear un estado del sistema para verificar las restricciones OCL. OCLE 2.0(Object Constraint Language Environment) [**21**] es otra herramienta que soporta edición y evaluación de OCL. Es posible escribir restricciones a nivel metamodelo y modelo. Estas últimas son evaluadas en base a las instancias especificadas en el modelo de usuario. Permite importar modelos UML realizados en otras herramientas CASE y soporta XMI. VOCLEditor [**39**] es un novedoso editor gráfico de OCL que permite escribir expresiones mediante un menú amigable, luego es posible convertir el diagrama VOCL a una representación textual. Eplatero además de la edición y evaluación de expresiones OCL a nivel metamodelo y modelo, agrega la posibilidad de especificar y evaluar refinamientos utilizando modelos UML enriquecidos con OCL. Esta característica es muy importante para el desarrollo de software complejo.

Boiten y Bujorianu en [**3**] exploran los refinamientos indirectamente a través de la unificación considerando varios modelos UML; la formalización es utilizada para descubrir y describir propiedades intuitivas en los refinamientos de UML. El lenguaje Z se utiliza para la formalización de UML. Esta propuesta tiene el mismo espíritu debido al hecho de que la formalización se emplea para descubrir y describir propiedades intuitivas en los refinamientos UML. Paige en [**23**] define refinamiento en términos de consistencia de modelo; Liu, Jifeng, Li y Chen en [**19**] utilizan un lenguaje de especificación orientado a objetos (OOL) para formalizar y combinar modelos UML. Ellos definen tres tipos de refinamientos interrelacionados en un diseño orientado a objetos; entonces, definen un conjunto de leyes de refinamiento en modelos UML para capturar la naturaleza esencial, principios y modelos de diseños orientados a objetos que son consistente con la definición de refinamiento (por ejemplo: “agregando un nuevo método a una clase”, “moviendo algún atributo de una clase a su superclass directa”). Lano en [**16**] describe un catálogo de “Patrones de Refinamiento UML” qué es un conjunto de reglas que se emplean para transformar sistemáticamente, de la forma más cerca, modelos UML al código Java. En contraste con la propuesta de esta tesis, los acercamientos expresados anteriormente son sólo descriptivos y no proporcionan ningún proceso de verificación.

8. Conclusiones finales y futuros trabajos

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 consiguen 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.

En este trabajo hemos desarrollado una herramienta que soporta OCL, la cual resulta útil para la evaluación de propiedades de los diseños orientados a objetos, tales como reglas de buena formación, métricas de calidad (Ej.: estructura de la jerarquía de clases, cohesión, acoplamiento, etc.), especificación de patrones de diseño [**11**] y también reglas del negocio.

Además de la evaluación de este tipo de propiedades, el paradigma de desarrollo de software dirigido por modelo requiere fundamentalmente que sea posible la evaluación de las transformaciones entre modelos; es decir, cada paso de transformación debe ser verificado formalmente, garantizando, de esta forma, la corrección del producto final con respecto a sus modelos. En la actualidad existen mecanismos para verificar cada paso en el proceso de desarrollo del sistema, sin embargo, estos mecanismos exigen el uso de lenguajes formales cuya complejidad obstaculiza su utilización en la práctica. Para disminuir esta brecha entre una metodología segura pero compleja por un lado, y una metodología insegura pero simple por el otro, en esta tesis hemos trabajado en la definición de un método de verificación de transformaciones (refinamientos) que es al mismo tiempo confiable y simple. Dicho método permite crear condiciones de refinamientos para los modelos UML, expresadas en OCL. La utilización de modelos UML / OCL y la existencia de herramientas que soportan el método descrito en esta tesis dan lugar a un marco de trabajo mucho mas amigable para los desarrolladores. Los resultados de esta investigación han sido presentados en [**27**].

Como complemento y con el objetivo de agregar eficiencia al método de evaluación de las condiciones de refinamiento, hemos adaptado una estrategia para reducir el espacio de búsqueda, basada en la creación automática de micro mundos en los cuales las condiciones de refinamiento son evaluadas en un tiempo finito. Este resultado ha sido descrito en [**26**].

En base a los resultados obtenidos nos planteamos los siguientes trabajos futuros:

- extender la herramienta para permitir la evaluación de las propiedades dinámicas del sistema, especificadas mediante pre y postcondiciones. Para ello será necesario enriquecer al evaluador con la capacidad de analizar propiedades que cambian a través del tiempo como resultado de la aplicación de las operaciones cuyas pre y post condiciones se desea evaluar.
- extender el método para soportar el refinamiento de operaciones con parámetros.
- completar nuestro catálogo de patrones de refinamiento.
- extender la herramienta para soportar la transformación automática de las restricciones OCL a través de los pasos de refinamiento, obteniendo de esta forma su traducción del lenguaje abstracto hacia lenguajes más concretos y viceversa.

9. REFERENCIAS

[1] 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.

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

[3] Boiten E.A. and Bujorianu M.C. Exploring UML refinement through unification. Proceedings of the UML'03 workshop on Critical Systems Development with UML, J. Jurjens, B. Rumpe, et al., editors -TUM-I0323, Technische Universitat Munchen. (2003).

[4] Davies J. and Crichton C. Concurrency and Refinement in the Unified Modeling Language. Electronic Notes in Theoretical Computer Science 70,3, Elsevier, 2002.

[5] 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.

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

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

[8] EMF: Eclipse Modeling Framework - http://www.eclipse.org/emf.

[9] Engels G., Küster J., Heckel R. and Groenewegen L. A Methodology for Specifying and Analyzing Consistency ofObject Oriented Behavioral Models. Procs. of the IEEE Int. Conference on Foundation of Software Engineering. Vienna. (2001).

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

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

[12] 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).

[13] 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).

[14] Jean-Marie Favre, Jacky Estublier, Mireille Blay. Beyond MDA : Model Driven Engineering (L'Ingénierie Dirigée par les Modèles : au-délà du MDA) Edition Hezmes-Lavoisier, ISBN 2-7462-1213-7. Février 2006.

[15] Kim, S. and Carrington, D., Formalizing the UML Class Diagrams using Object-Z, proceedings UML´99 Conference, Lecture Notes in Computer Sciencie 1723 (1999).

[16] 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).

[17] Lano,K., Biccaregui,J., Formalizing the UML in Structured Temporal Theories, 2nd. ECOOP Workshop on Precise Behavioral Semantics, TUM-I9813, Technische U. Munchen (1998).

[18] Ledang, Hung and Souquieres, Jeanine. Integration of UML and B Specification Techniques: Systematic Transformation from OCL Expressions into B. Procs. of IEEE Asia-Pacific Software Engineering Conference 2002. December 4-6, (2002).

[19] Liu, Z., Jifeng H., Li, X. Chen Y. Consistency and Refinement of UML Models. 3er Workshop on Consistency Problems in UML-based Software Development III, event of the UML Conference, (2004).

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

[21] OCLE 2.0: Object Constraint Language Environment - lci.cs.ubbcluj.ro/ocle/

[22] Object Management Group, MDA Guide, v1.0.1, omg/03-06-01, June 2003.

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

[24] 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

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

[26] Pons Claudia and Garcia Diego . "An OCL-based Technique for Specifying and Verifying Refinement-oriented Transformations in MDE". Proceedings MoDELS/UML 2006 "Model Driven Engineering Languages and Systems, 9th International Conference, Genoa, Italy, October 2006. Lecture Notes in Computer Science (LNCS). Springer. Volume Editors: Oscar Nierstrasz, Jon Whittle, David Harel, Gianna Reggio.

[27] Pons Claudia and Garcia Diego. "Practical Verification Strategy for Refinement Conditions in UML Models". First International Workshop on Advanced Software Engineering IWASE 2006. co-located with the 19th IFIP World Computer Congress 2006. Santiago, Chile. Springer Science+Business Media. August 25, 2006.

[28] Pons C., Giandini R., Pérez G., et al. Precise Assistant for the Modeling Process in an Environment with Refinement Orientation. In "UML Modeling Languages and Applications: Satellite Activities". Lecture Notes in Computer Science 3297. Springer, (2004).

[29] Pons C.,. Heuristics on the Definition of UML Refinement Patterns. 32nd International Conference on Current Trends in Theory and Practice of Computer Science. SOFSEM (SOFtware SEMinar). January 21 - 27, 2006 . Merin, Czech Republic. Published in the Springer LNCS (Lecture Notes in Computer Science series by Springer-Verlag. (2006)

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

[31] Richters, Mark and Gogolla, Martin. OCL-Syntax, Semantics and Tools. in Advances in Object Modelling with the OCL. Lecture Notes in Computer Science number 2263. Springer. (2001).

[32] Smith, Graeme. The Object-Z Specification Language. Advances in Formal Methods. Kluwer Academic Publishers. ISBN 0-7923-8684-1. (2000).

[33] Spivey, M. The Z notation: a reference manual. Prentice Hall, Englewood Cliffs, NJ, Second edition, 1992.

[34] Stahl, M Voelter. Model Driven Software Development. John Wiley, ISBN 0470025700, April 2006.

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

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

[37] USE: UML based Specification Environment - http://www.db.informatik.uni-bremen.de/projects/USE/

[38] Van Der Straeten, R., Mens,T., Simmonds, J. and Jonckers,V. Using description logic to maintain consistency between UML-models. In Proc. 6th International Conference on the Unified Modeling Language. Lecture Notes in Computer Science number 2863. Springer. (2003).

[39] VOCLEditor - www.tfs.cs.tu-berlin.de/vocl/

10. Anexos

10.1. Transformación de las condiciones de refinamiento de Object- Z a OCL

Gramática para las expresiones de refinamientos en Z

Esta sección describe la gramática para las expresiones de refinamiento de Z que son un subconjunto de la gramática de Object-Z presentada en [**32**].

La descripción de la gramática utiliza la sintaxis de EBNF, donde los símbolos terminales se visualizan con letra negrita, y los opcionales entre [].

```
Predicate::=      ∃ SchemaText • Predicate
                  | ∀ SchemaText • Predicate
                  | Predicate1
Predicate1::=     className.INIT
                  | pre operationName
                  | operationName
                  | relationName
                  | Predicate1 ∧ Predicate1
                  | Predicate1 ⇒ Predicate1
                  | (Predicate)
SchemaText::=     className.STATE [Decoration]
className::=      Word
operationName::=  className.Word
relationName::=   Word [Decoration]
Word              category for undecorated names
Decoration::=     ’
```

Definición de la función de transformación

Esta sección contiene la especificación de la función T que dada una condición de refinamiento escrita en Object-Z retorna la correspondiente condición de refinamiento escrita en OCL.

Los elementos UML se obtienen del modelo M utilizando las operaciones *lookup* estándares en su ambiente como se define en [**20**]

T : Model -> Predicate -> (OclExpression, OclFile)

T_M(Predicate1 ∧ Predicate2)= (e,Φ)

Where

T_M(Predicate1)= (e1, Φ1)

T_M(Predicate2)= (e2, Φ2)

e= e1 **“and”** e2

Φ = Φ1 *merge* Φ2

$\boldsymbol{T}_{\mathbf{M}}$(Predicate1 $\Rightarrow$ Predicate2)= (e,Φ)

Where

$\mathbf{T}_{\mathbf{M}}$(Predicate1)= (e1, Φ1)

$\mathbf{T}_{\mathbf{M}}$(Predicate2)= (e2, Φ2)

e= e1 **“implies”** e2

Φ = Φ1 *merge* Φ2

$\mathbf{T}_{\mathbf{M}}$($\forall$ className**.STATE** • Predicate) = (e,Φ)

Where

$\mathbf{T}_{\mathbf{M}}$(Predicate)= (e1, Φ)

e=className“.allInstances()->forAll(”iteratorName“|”e1“)”

iteratorName= toLowerCase(className)

$\mathbf{T}_{\mathbf{M}}$($\forall$ className**.STATE’** • Predicate) = (e,Φ)

Where

$\mathbf{T}_{\mathbf{M}}$(Predicate)= (e1, Φ)

e=className“.allInstances()->forAll(”iteratorName“|”e1“)”

iteratorName= toLowerCase(className) “_post”

$\mathbf{T}_{\mathbf{M}}$($\exists$ className**.STATE** • Predicate) = (e,Φ)

Where

$\mathbf{T}_{\mathbf{M}}$(Predicate)= (e1, Φ)

e=className“.allInstances()->exists(”iteratorName“|”e1“)”

iteratorName= toLowerCase(className)

$\mathbf{T}_{\mathbf{M}}$($\exists$ className**.STATE’** • Predicate) = (e,Φ)

Where

$\mathbf{T}_{\mathbf{M}}$(Predicate)= (e1, Φ)

e=className“.allInstances()->exists(”iteratorName“|”e1“)”

iteratorName= toLowerCase(className) “_post”

$\boldsymbol{T}_{\mathbf{M}}$ (className.***INIT***) =(e,Φ)

Where

e= toLowerCase(className) “.isInit()”

Φ = ”**Package”** packageName_1

"**context”** className “**def**: isInit(): Boolean =”

attributeName_1“=”exp_1“**and”**...“**and”** attributeName_n“=”exp_n **“and”** navigationName_1“->size() =” size_1 **“and”** navigationName_1 “->forAll(p| p.isInit())”...“**and”** navigationName_n“->size() =” size_n **“and”** navigationName_n “->forAll(p| p.isInit())”

“**endPackage”**

packageName = class.package.name

class : UMLClass =

M.getEnvironmentWithParents().lookup(className)

attributes: Sequence(UMLProperty) =

class.allProperties()->select(p|p.initialValue->notEmpty())

$\forall$j•1≤j≤attributes->size()• attributeName_j = attributes->at(j).name

```
            ∧ exp_j = attributes->at(j).initialValue.body
    navigations: Sequence(UMLProperty) =
      class.allProperties()->select(p|p.association->notEmpty() and
    p.isComposite())
    ∀j•1≤j≤navigations->size()• navigationName_j =
      navigations->at(j).name
            ∧ size_j = navigations->at(j).lower
```

T_M(pre className.operationName) = (e, ∅)[1]

```
    Where:
    e = operation.precondition.specification.body.tranlated(className)
    operation : UMLOperation  =
      M.getEnvironmentWithParents().lookup(className).
      getEnvironmentWithParents()
      .lookupImplicitOperation(operationName, Sequence{})
```

Where:

La función translated(className) se aplica a un OclExpression y retorna una copia de la expresión donde el nombre de la variable contextual es className en minúscula.

T_M (className.operationName)= (e, ∅)

```
    Where:
    e = operation.postcondition.specification.body
    .translated(className).renamed()
    operation : UMLOperation  =
      M.getEnvironmentWithParents().lookup(className).
      getEnvironmentWithParents()
       .lookupImplicitOperation(operationName,Sequence{})
```

Where:

La función translated(className) se aplica a un OclExpression y retorna una copia de la expresión donde el nombre de la variable contextual es className en minúscula.

La función renamed () se aplica a un OclExpression y retorna una copia de la expresión donde cualquier propiedad no decorada se renombra la variable receptora v (source) como v_post y cualquier elemento decorado p@pre se la renombra como p.

T_M (relationName) = (e,Φ)

```
    Where:
    relationName ∈ Word -- it is an undecorated name
    e = absInstance ".mapping(" refInstance ")"
    Φ = "package" packageName
       "context a:" AbstractClass  "def:"
       "mapping(c:" RefinedClass "):Boolean =" exp
       "endPackage"
    Where:
    d : Abstraction =
          M.getEnvironmentWithParents().lookup(relationName)
    AbstractClass  = d.supplier.name
    RefinedClass = d.client.name
```

[1] In this document the symbol ∅ is an abbreviation denoting the empty package.

```
    absInstance = toLowerCase(AbstractClass)
    refInstance = toLowerCase(RefinedClass)
    exp = d.mapping.body
    packageName = abstractClass.package.name
```

T_M **(**relationName’) = (e,∅)

Where:

```
e = absInstance “.mapping(” refInstance “)”
```

Where:

```
d : Abstraction =
      M.getEnvironmentWithParents().lookup(relationName)
AbstractClass  = d.supplier.name
RefinedClass = d.client.name
absInstance = toLowerCase(AbstractClass) “_post”
refInstance = toLowerCase(RefinedClass)  “_post”
```

10.2. Gramática de OCL 2.0

La gramática que se muestra a continuación se puede emplear para generar un parser de OCL 2.0.

```
packageDeclaration ::= ’package’ pathname contextDeclList ’endpackage’ | contextDeclList
contextDeclList ::= contextDeclaration*
contextDeclaration ::= propertyContextDecl | classifierContextDecl | operationContextDecl
propertyContextDecl ::= ’context’ pathname ‘::’ simpleName ’:’ type initOrDerValue+
initOrDerValue ::= ’init’ ’:’ oclExpression | ’derive’ ’:’ oclExpression
classifierContextDecl ::= ’context’ [simpleName’:’] pathname invOrDef+
invOrDef ::= ’inv’ [simpleName] ’:’ oclExpression
              | ’def’ [simpleName] ‘:’ defExpression
defExpression ::= simpleName ‘:’ type ‘=’ oclExpression
                  | operation ‘=’ oclExpression
operationContextDecl ::= ’context’ operation prePostOrBodyDecl+
prePostOrBodyDecl ::= ’pre’ [simpleName] ’:’ oclExpression
                        | ’post’ [simpleName] ’:’ oclExpression |
                         ’body’ [simpleName] ’:’ oclExpression
operation ::= pathName ’(’ [variableDeclarationList] ’)’ [’:’ type]
variableDeclarationList ::= variableDeclaration ( ’,’ variableDeclaration )*
variableDeclaration ::= simpleName [’:’ type] [’=’ oclExpression]
type ::= pathname | collectionType |  tupleType
collectionType ::= collectionKind ’(’ type ’)’
tupleType ::= ’TupleType’ ’(’ variableDeclarationList ’)’
oclExpression ::= logicalImpliesExpression | letExpression
letExpression ::= ‘let’ variableDeclarationList ‘in’ oclExpression
logicalImpliesExpression ::= logicalXorExpression
                             | logicalImpliesExpression ’implies’ logicalXorExpression
logicalXorExpression ::= logicalOrExpression
                         | logicalXorExpression ‘xor’ loficalOrExpression
logicalOrExpression ::= logicalAndExpression
                        |  loficalOrExpression ‘or’ logicalAndExpression
logicalAndExpression ::= relationalExpression
                         | logicalAndExpression ‘and’ relationalExpression
relationalExpression ::= addExpression
                         | addExpression relationalOperator addExpression
relationalOperator  ::=  < | <= | > | >= | <> | =
addExpression ::= mulExpression
                  | addExpression  ‘+’ mulExpression
                  | addExpression ‘-‘ mulExpression
mulExpression ::= unaryExpression
                  | mulExpression ‘*’ unaryExpression
                  | mulExpression ‘/’ unaryExpression
unaryExpression ::= primaryExpression
                    | ‘-‘ unaryExpression
                    | ‘not’ unaryExpression
primaryExpression ::=  literalExp | ‘(‘ oclExpression ‘)’ |  postfixExpression | ifExpression
                       | propertyCall
ifExpression ::= ‘if’ oclExpression ‘then’ oclExpression ‘else’ oclExpression ‘endif’
postfixExpression ::= primaryExpression ‘.’propertyCall
                      | primaryExpression ‘->’ propertyCall
                      | primaryExpression ‘^’ messageCall
                      | primaryExpression ‘^^’ messageCall
```

propertyCall ::= simpleName [@pre] [qualifiers] [propertyCallParams]
propertyCallParams ::= ‘(‘ [declarator] oclExpression [‘,’ ocExpression]* ’)’
declarator :.= simpleName [‘,’ simpleName]* [‘:’type]
[‘;’ simpleName ‘:’ type ‘=’ oclExpression] ’|’
messageCall ::= pathName ‘(‘ [messageCallArgument] [‘,’ messageCallArgument]* ‘)’
messageCallArgument ::= ‘?’ [:type] oclExpression
qualifiers ::= ‘[‘ oclExpression [‘,’ oclExpression]*‘]’
formalParameter ::= simpleName ‘:’ type
literalExp ::= collectionLiteralExp | tupleLiteralExp | primitiveLiteralExp
collectionLiteralExp ::= collectionKind ’{’ collectionLiteralParts ’}’ | collectionKind ’{’ ’}’
collectionKind ::= ’Set’ | ’Bag’ | ’Sequence’ | ’Collection’ | ’OrderedSet’
collectionLiteralParts ::= collectionLiteralPart (’,’ collectionLiteralPart)*
| oclExpression | collectionRange
collectionRange ::= oclExpression ’..’ oclExpression
tupleLiteralExp ::= ’Tuple’ ’’ variableDeclarationList ’’
primitiveLiteralExp ::= integer | real | string | ’true’ | ’false’
pathname ::= simpleName | pathName ’::’ simpleName
integer ::= [0-9]+
real ::= integer[.]integer[eE][+-]?integer | integer[eE][+-]?integer | integer[.]integer
string ::= [’][^’]*[’]
simpleName ::= [a-zA-Z][a-zA-Z0-9]*

10.3. Ejemplo de un archivo “.jj”

En este anexo se describe la estructura del archivo “.jj” que se utiliza para generar un parser con javacc. Las palabras que están en negrita son las palabras claves de la gramática de “javacc”. Las secciones en verde son comentarios sobre cada parte del documento.

Estructura del archivo

```
options {
            DEBUG_PARSER = false;
          }

/*
*     Comienza la definición de la clase parser
*/

PARSER_BEGIN(EjemploParser)

/*
*    Imports que necesitan los archivos java generados por la herramienta.
*/

import java.io.InputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

/*
*    Nombre de la clase que se encarga de hacer el parser.
*/

public class EjemploParser {

   public static void parse(InputStream aInSt) throws ParseException {
      // create a parser (this object)
      EjemploParser parser = new EjemploParser(aInSt);
     // parse!
     parser.ejemplo();
   }

   public static void main(String[] args) throws ParseException,
     FileNotFoundException {
       parse(new FileInputStream(args[0]));
   }
}

PARSER_END(EjemploParser)

/*
* Se definen los tokens. Los Token son las palabras claves (keywords) de la
* gramática
*/

TOKEN :
{
     <LETTER: [ "A" -"Z", "a"-"z" ] >
   | <DIGIT: [ "0"-"9" ] >
   | <NAMECHAR: ( <LETTER> | <DIGIT> | "." | "-" | "_" | ":" ) >
   | <NAME: ( <LETTER> | "_" | ":" ) ( <NAMECHAR> )*>
   | <PRECONDITION : "pre" >
   | <POSTCONDITION : "post" >
   | <INVARIANT : "inv" >
```

```
   | <DEFINITION : "def" >
}

/*
*    Este comando es para especificar que palabras debe ignorar el parser.
*    Se puede usar también para definir el formato de los comentarios, que siempre
*    son ignorados por el parser.
*/

/* WHITE SPACE */

SKIP:
{
     " "
   | "\t"
   | "\n"
   | "\r"
   | "\f"
}

/*
*  Similar el skip. También se utiliza para ignorar ciertas palabras.
*  Por ejemplo se puede ignorar las " --" que se emplean para escribir los comentarios OCL.
*/

MORE :
{
     "--"
}

/*
 * Este método es traducido a código java y se añade en el parser.
*/

void ejemplo() :
{
        /* Código java. Generalmente inicializaciones del método */
}

{
        ( elementDecl() | attListDecl() )* <EOF>
}
```

10.4. Tipos básicos de OCL

En este anexo se incluye los tipos básicos de OCL con sus respectivas operaciones. Toda implementación de OCL debe incluir esta librería.

Real

Este tipo representa el concepto matemático de real.

OPERACION	DESCRIPCION
+ (r : Real) : Real	Retorna la suma del receptor y el argumento, r. La suma es un nuevo Real.
- (r : Real) : Real	Retorna la diferencia entre el receptor y el argumento, r. La diferencia es un nuevo Real.
* (r : Real) : Real	Retorna el resultado de multiplicar al receptor por el argumento, r. El resultado es un nuevo Real.
- : Real	Retorna un Real que es la negación del receptor.
/ (r : Real) : Real	Retorna el valor de dividir el receptor por el argumento, r.
abs() : Real	Retorna el valor absoluto del receptor.
floor() : Integer	El entero mayor que es menor o igual al receptor.
round() : Integer	El entero que es más acotado al receptor. Cuando hay dos enteros, el más grande.
max(r : Real) : Real	El máximo del receptor y el argumento, .
min(r : Real) : Real	El mínimo del receptor y el argumento, r.
< (r : Real) : Boolean	Retorna true si el receptor es menor que el argumento y false en caso contrario.
> (r : Real) : Boolean	Retorna true si el receptor es mayor que el argumento y false en caso contrario.
<= (r : Real) : Boolean	Retorna true si el receptor es menor o igual que el argumento y false en caso contrario.
>= (r : Real) : Boolean	Retorna true si el receptor es mayor o igual que el argumento y false en caso contrario.

Tabla 4 – Operaciones para Real

Integer

Este tipo representa el concepto matemático de entero. La superclase de Integer es Real, para que cada uno de los parámetros del tipo Real, se pueda usar un entero como dicho parámetro.

OPERACION	DESCRIPCION
+ (i : Integer) : Integer	Retorna la suma del receptor y el argumento, r. La suma es un nuevo Integer.
- (i : Integer) : Integer	Retorna la diferencia entre el receptor y el argumento, r. La diferencia es un nuevo Integer.
* (i : Integer) : Integer	Retorna el resultado de multiplicar al receptor por el argumento, r. El resultado es un nuevo Integer.
- : Integer	Retorna un Integer que es la negación del receptor.
/ (i : Integer) : Real	Retorna el valor de dividir el receptor por el argumento, r.
abs() : Integer	Retorna el valor absoluto del receptor.
div(i : Integer) : Integer	El número de veces que i se ajusta completamente dentro del receptor.
mod(i : Integer) : Integer	El resto de la división entera.
max(i : Integer) : Integer	El máximo del receptor y el argumento, .
min(i : Integer) : Integer	El mínimo del receptor y el argumento, r.

Tabla 5–Operaciones para Integer

String

El tipo String representa una secuencia de caracteres que pueden ser ASCII o Unicode.

OPERACION	DESCRIPCION
size() : Integer	Retorna el números de caracteres del receptor.
concat(s : String) : String	Retorna un nuevo String formado por la concatenación del receptor y el argumento.

substring(lower : Integer, upper : Integer) : String	Retorna un nuevo String que consta del substring del receptor. Los argumentos indican los caracteres a tomar.
toInteger() : Integer	Convierte al receptor en un valor entero.
toReal() : Real	Convierte al receptor en un valor real.

Tabla 6–Operaciones para String

Boolean

El tipo Boolean representa los valores true / false.

OPERACION	DESCRIPCION
or (b : Boolean) : Boolean	True si el receptor o el argumento son true.
xor (b : Boolean) : Boolean	True si el receptor o b son true, pero no ambos.
and (b : Boolean) : Boolean	True si el receptor y b son true.
not : Boolean	True si el receptor es false y false en caso contrario.
implies (b : Boolean) : Boolean	True si el receptor es false, o si el receptor es true y b es true.

Tabla7–Operaciones para Boolean

Collection

A continuación se detallan las operaciones de colecciones predefinidas.

OPERACION	DESCRIPCION
col->size():Integer	El número de elementos en la colección.
col->includes(object: T): Boolean	True si el objeto está incluido en la Colección.
col->excludes(object: T): Boolean	True si el objeto no está incluido en la colección.
col->count(object: T): Boolean	El número de veces que object está presente en la colección.

col->includesAll(c2: Collection(T)):Boolean	Verifica si la colección col contiene todos los elementos de c2.
col->includesAll(c2: Collection(T)):Boolean	Verifica que la colección col no contenga ninguno de los elementos de c2.
col->isEmpty(): Boolean	True si la colección es vacía.
col->notEmpty():Boolean	True si la colección no es vacía.
col->sum():T	La suma de todos los elementos de la colección. Los elementos deben ser de un tipo que soporte la operación +.
col->product(c2: Collection(T2)):Set(Tuple(first:T, second:T2))	El producto cartesiano de la colección col y c2.
col->exists(iterators | body):Boolean	True si la expresión body se evalúa a true al menos para un elemento de la colección.
col->forAll(iterators | body):Boolean	True si la expresión body se evalúa a true para cada elemento de la colección, false en caso contrario
col->isUnique (iterators | body):Boolean	True si el valor resultante de evaluar la expresión body es único para cada elemento de la colección.
col->any(iterator | body) = T	Retorna cualquier elemento de la colección que al evaluar la expresión body sea verdadera. Si no hay ningún elemento que cumpla con dicha condición se retorna null.
col->one(iterator | body) = T	True si hay un solo elemento de la colección para el cual la expresión body es verdadera, false en caso contrario.

Tabla 8–Operaciones para Collection

OPERACION	DESCRIPCION
col->union(s: Set(T)):Set(T)	La unión de col y s.
col->union(bag: Bag(T)):Bag(T)	La unión de col y bag.
col->=(s:Set(T)):Boolean	True si el receptor y s tienen los mismos elementos.

col->intersection(s: Set(T)):Set(T)	La intersección de col y s.
col->intersection(bag: Bag(T)):Set(T)	La intersección de col y bag.
col->-(s: Set(T)): Set(T)	Los elementos de col que no están en s.
col->including(object:T):Set(T)	Retorna un conjunto que contiene todos los elementos de col más object.
col->excluding(object:T):Set(T)	El conjunto que contiene todos los elementos de col menos object.
col-> symmetricDiference(s:Set(T)):Set(T)	El conjunto que contiene todos los elementos que están en col o en s, pero no en ambos.
col->count(object:T):Integer	El número de ocurrencias de object en col.
col->flatten():Set(T)	Si el tipo de elemento no es un tipo colección el resultado es col, de otro modo el resultado es una colección que contiene todos los elementos de todos los elementos de col.
col->asSet():Set()	Un conjunto idéntico a col. Esta operación existe por razones de conveniencia.
col->asOrderedSet():OrderedSet(T)	Un OrderedSet que contiene todos los elementos de col.
col->asSequence():Sequence(T)	Una secuencia que contiene todos los elementos de col.
col->asBag():Bag(T)	El Bag que contiene todos los elementos de col.
col->select(iterator | body) =Set(T)	Retorna un subconjunto de la colección para el que la expresión body es verdadera.
col->reject(iterator | body) =Set(T)	El subconjunto de la colección para el que la expresión body es falso.
col->collect(iterators | body) =Bag(T)	El Bag de elementos con los resultados de aplicar la expresión body a cada miembro de la colección.
col->sortedBy(iterator | body) =OrderedSet(T)	Retorna el OrderedSet que contiene todos los elementos de la colección. El elemento para el que la expresión body

	tiene el valor más bajo es el primer elemento de la colección resultante, y así sucesivamente. El tipo de la expresión body debe soportar la operación <. Esta operación debe retornar un valor del tipo Boolean y deben ser transitiva si a <b y b <c entonces a <c.

Tabla 9–Operaciones para Set

OPERACION	DESCRIPCION
col->append(object:T):OrderedSet(T)	El conjunto de elementos, que consiste de todos los elementos de col, seguidos de object.
col->prepend(object:T):OrderedSet(T)	La secuencia que consiste en object, y todos los elementos de col.
col->insertAt(index: Integer, object:T):OrderedSet(T)	El conjunto que consiste de col pero object se inserta en la posición index.
col->subOrderedSet(lower:Integer, upper: Integer):OrderedSet(T)	El subconjunto de col que comienza en la posición lower y finaliza en la posición upper.
col->at(i:Integer): T	El i-ésimo elemento de col.
col->indexOf(obj:T):Integer	El índice del objeto obj en la secuencia.
col->first():T	El primer elemento en col.
col->last():T	El último elemento en col.

Tabla 10–Operaciones para OrderedSet

OPERACION	DESCRIPCION
col->=(bag:Bag(T)):Boolean	True si col y bag tienen los mismos elementos.
col->union(bag: Bag(T)):Bag(T)	La unión de col y bag.
col->union(set: Set(T)):Bag(T)	La unión de col y set.
col->intersection(bag: Bag(T)):Bag(T)	La intersección de col y bag.
col->intersection(set: Set(T)):Set(T)	La intersección de col y set.

col->including(object:T):Bag(T)	El bag que contiene todos los elementos de col más object.
col->excluding(object:T):Bag(T)	El bag que contiene todos los elementos de col menos todas las ocurrencias de object.
col->count(object:T):Integer	El número de ocurrencias de object en col.
col->flatten():Bag(T2)	Si el tipo de elemento no es un tipo colección el resultado es col, de otro modo el resultado es el bag que contiene todos los elementos de todos los elementos de col
col->asBag():Bag(T)	Un Bag idéntico a self. Esta operación existe por razones de conveniencia.
col->asSequence():Sequence(T)	Una secuencia que contiene todos los elementos de col.
col->asSet():Set(T)	El conjunto que contiene todos los elementos de col.
col->asOrderedSet():OrderedSet(T)	El conjunto ordenado que contiene todos los elementos de col.
col->select(iterator | body) =Bag(T)	Retorna un sub-bag de la colección para el que la expresión body es verdadera.
col->reject(iterator | body) =Bag(T)	El sub-bag de la colección para el que la expresión body es falso.
col->collect(iterators | body) =Bag(T)	El Bag de elementos con los resultados de aplicar la expresión body a cada miembro de la colección.
col->sortedBy(iterator | body) =Sequence(T)	Retorna la secuencia que contiene todos los elementos de la colección. El elemento para el que la expresión body tiene el valor más bajo es el primer elemento de la colección resultante, y así sucesivamente. El tipo de la expresión body debe soportar la operación <. Esta operación debe retornar un valor del tipo Boolean y deben ser transitiva si a <b y b <c entonces a <c.

Tabla 11–Operaciones para Bag

OPERACION	DESCRIPCION
col->count(object:T):Integer	El número de ocurrencias de object en el receptor.
col->=(s:Sequence(T)):Boolean	True si col consiste de todos los elementos de s en el mismo orden.
col->union(s:Sequence(T)): Sequence(T)	La secuencia que consiste de todos los elementos de col, seguido por todos los elementos en s
col->flatten():Sequence(T2)	Si el tipo de elemento no es un tipo colección el resultado es la misma secuencia col, de otro modo el resultado es la secuencia que contiene todos los elementos de todos los elementos de col.
col->append(object:T):Sequence(T)	La secuencia de elementos que consiste de todos los elementos de col seguidos de object.
col->prepend(object:T):Sequence(T)	La secuencia de elementos que consiste de object, y todos los elementos de col
col->insertAt(index: Integer, object: T):Sequence(T)	La secuencia que consiste de col con object insertado en la posición index.
col->subSequence(lower:Integer, upper: Integer):Sequence(T)	La subsecuencia de col que comienza en la posición lower y finaliza en la posición upper.
col->at(i:Integer): T	El i-ésimo elemento de la secuencia col.
col->indexOf(obj:T):Integer	El índice del objeto obj en la secuencia.
col->first():T	El primer elemento en col.
col->last():T	El último elemento en col.
col->including(object:T):Sequence(T)	La secuencia que contiene todos los elementos de col más object agregado como último elemento.
col->excluding(object:T):Sequence(T)	La secuencia que contiene todos los elementos de col menos todas las ocurrencias de object.
col->asBag():Bag()	El bag que contiene todos los elementos de self, incluyendo duplicados.
col->asSequence():Sequence(T2)	La secuencia idéntica a col. Esta operación existe por razones de

	conveniencia.
col->asSet():Set(T)	El conjunto que contiene todos los elementos de col, con duplicados removidos.
col->asOrderedSet():OrderedSet(T)	El conjunto ordenado que contiene todos los elementos de col, en el mismo orden, y con duplicados removidos.
col->select(iterator | body) =Sequence(T)	Retorna un subsecuencia de la colección para el que la expresión body es verdadera.
col->reject(iterator | body) = Sequence (T)	La subsecuencia de la colección para el que la expresión body es falso.
col->collect(iterators | body) = Sequence (T)	La subsecuencia de elementos con los resultados de aplicar la expresión body a cada miembro de la colección.
col->sortedBy(iterator | body) =Sequence(T)	Retorna la secuencia que contiene todos los elementos de la colección. El elemento para el que la expresión body tiene el valor más bajo es el primer elemento de la colección resultante, y así sucesivamente. El tipo de la expresión body debe soportar la operación <. Esta operación debe retornar un valor del tipo Boolean y deben ser transitiva si a <b y b <c entonces a <c.

Tabla 12–Operaciones para Sequence