UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Titulo: Verificación de modelos Independientes de la plataforma: un caso de estudio
Autores: Carolina Inés Actis
Director: Claudia Pons
Codirector:
Asesor profesional: Gabriela Alejandra Perez
Carrera: Licenciatura en Informática

Resumen

El lenguaje UML ha sido ampliamente aceptado como el lenguaje estándar de modelado en la industria. El lenguaje OCL es una parte integral de UML, y fue introducido para definir restricciones adicionales que no se pueden expresar en este. Las expresiones OCL son concisas y precisas, y no presentan las ambigüedades del lenguaje natural. Sin embargo, al ser una notación de diseño, OCL no es ejecutable; está definido sobre el modelo, por lo que sus restricciones no se reflejan en el código fuente.
Por otro lado, JML es un lenguaje de especificación formal que puede ser utilizado para especificar clases Java. A diferencia de OCL, las expresiones JML están escritas de forma que pueden ser compiladas y analizadas en tiempo de ejecución.
En este trabajo se propone transformar de forma automática las restricciones OCL a especificaciones escritas en el lenguaje JML. De esta forma se podrán verificar las restricciones en tiempo de ejecución, y se podrá hacer un análisis estático de estas mediante el uso de probadores de teoremas implementados para JML.

Palabras Claves

Acceleo, ATL, Eclipse, Java, JML, Desarrollo dirigido por modelos (MDD), OCL, OpenJML, Traducción OCL a JML, Transformaciones de modelos, UML, Verificación de programas

Conclusiones

Se desarrolló de una herramienta que permite la traducción automática de OCL a JML. De esta forma, a partir de un documento con restricciones OCL se genera automáticamente código Java con especificaciones JML. Esto nos permite aprovechar las herramientas de JML, como por ejemplo, la verificación estática del código mediante probadores de teoremas, y la evaluación en tiempo de ejecución de las restricciones definidas.

Trabajos Realizados

Investigación del MDD, y los lenguajes OCL y JML. Desarrollo de un plugin Eclipse de traducción de OCL a JML, mediante una transformación Modelo a Modelo y una transformación Modelo a Texto. Verificación estática y en tiempo de ejecución del código generado utilizando la herramienta OpenJML.

Trabajos Futuros

Optimizar la traducción considerando patrones comunes en las restricciones OCL.
Analizar la posibilidad de implementar la traducción bidireccional.
Integrar la herramienta desarrollada con OpenJML para realizar la verificación del código directamente

Fecha de la presentación: Noviembre 2017

Tanto el texto de la presente tesina como el software desarrollado para la misma y su código son de libre uso para quien lo desee.

Agradecimientos

A mi familia por brindarme la posibilidad de estudiar esta carrera y por su apoyo durante la misma.

A Claudia, por toda su ayuda y su paciencia.

Índice General

Índice de Figuras

Índice de Tablas

1 Introducción

A lo largo de los últimos años, el Desarrollo de Software Dirigido por Modelos (denominado MDD por su acrónimo en inglés, Model-driven Development) [**1**] ha ido ganando territorio en el ámbito informático, como una nueva área dentro del campo de la ingeniería de software. En MDD, los modelos tienen un papel principal y activo en el proceso de desarrollo de software, logrando la independencia del software y la portabilidad de los sistemas, y separando el diseño de la arquitectura. A través de una serie de transformaciones, estos modelos pueden ser traducidos a código fuente, dependiente de una plataforma específica.

Como consecuencia, se mejora la productividad del sistema, se aumenta su calidad, y se facilita su comprensión, evolución, mantenimiento y reuso/reimplementación en otras tecnologías.

MDA (por Model Driven Architecture, Arquitectura Dirigida por Modelos) [**2**], es una implementación de MDD propuesta por la OMG [**3**]. MDA recomienda el uso inicial de un Modelo Independiente de la Plataforma (PIM, Platform Independent Model), para ser refinado en uno o más Modelos Específicos de la Plataforma (PSM, Platform Specific Model), que son sus especializaciones, teniendo en cuenta las características de la tecnología particular adoptada. El PSM será adaptado o completamente reemplazado de acuerdo con los cambios frecuentes en la tecnología.

En el contexto del MDD, el lenguaje UML [**4**] ha sido ampliamente aceptado como el lenguaje estándar de modelado en la industria. Los elementos gráficos de UML son limitados en cuanto a la capacidad de expresión semántica de sus modelos. El lenguaje OCL [**5**] es una parte integral de UML, y fue introducido para definir restricciones adicionales que no se pueden expresar en este.

OCL es un lenguaje de especificación formal basado en texto, que funciona como extensión de UML. OCL permite añadirle a UML restricciones o comportamiento que no pueden ser definidos de forma gráfica. Las expresiones OCL son concisas y precisas, y no presentan las ambigüedades del lenguaje natural.

Sin embargo, al ser una notación de diseño, OCL no es ejecutable. OCL está definido sobre el modelo, por lo que sus restricciones no se reflejan en el código fuente.

Por otra parte, JML (Java Modeling Language) [**6**] es un lenguaje de especificación formal que puede ser utilizado para especificar clases e interfaces Java. JML proporciona el concepto de diseño por contrato [**7**] al lenguaje Java. Las especificaciones JML pueden ser agregadas al código fuente mediante anotaciones, o agregarse en un archivo separado. En JML se especifica el comportamiento de una clase mediante el uso de, entre otros, invariantes de clase y pre y postcondiciones para sus métodos. Las expresiones están escritas de forma que puedan ser compiladas y detectadas en tiempo de ejecución.

En este trabajo se propone transformar las restricciones OCL a código fuente, específicamente, al lenguaje JML. De esta forma se podrán verificar las restricciones en tiempo de ejecución, y hacer un análisis estático de estas mediante el uso de probadores de teoremas implementados para JML.

La tesina se organiza de la siguiente forma:

- En el capítulo 2 se realiza una introducción al Desarrollo de Software Dirigido por Modelos, y se definen los conceptos básicos de modelos y sus transformaciones.
- En el capítulo 3 se presenta el lenguaje OCL, y se describen sus objetivos y características claves.
- En el capítulo 4 se introduce el lenguaje JML, se describen sus características principales y su utilidad en el desarrollo.
- En el capítulo 5 se describen las diferentes herramientas utilizadas en el desarrollo de la tesina.
- En el capítulo 6 se realiza una comparación entre los lenguajes OCL y JML, y se describe en detalle la función de traducción realizada.
- En el capítulo 7, se describen en detalle los módulos que componen la herramienta desarrollada. Se muestran dos casos de estudio y se aplica la traducción a estos. Luego sobre la traducción realizada, se ejecutan las herramientas de verificación de JML.
- En el capítulo 8 se presentan diversos trabajos relacionados con esta tesina y se destacan los aportes originales de la misma.
- Finalmente, en el capítulo 9 se presentan las conclusiones finales, y se mencionan trabajos futuros que podrían realizarse a partir del presente trabajo.

2 Desarrollo de Software dirigido por modelos

El Desarrollo de Software Dirigido por Modelos [**8**], o MDD, por sus siglas en inglés (Model Driven Development), se ha convertido en un nuevo paradigma de desarrollo software, que promete mejorar el proceso de desarrollo de software basándose en un proceso guiado por modelos y soportado por potentes herramientas.

En este capítulo se describen sus elementos principales y su utilidad en el ámbito del desarrollo software.

2.1 Introducción

Como su nombre lo dice, en MDD los modelos son parte fundamental del proceso de desarrollo. Los procesos de MDD suelen empezar con una fase de requerimientos en la que se define un modelo que describe las necesidades del usuario, de forma independiente de la computación. Luego, este modelo es refinado y/o transformado en uno o más modelos conceptuales que describen el sistema sin tener en cuenta los aspectos tecnológicos. Estos modelos se utilizan principalmente en las fases de análisis. Finalmente, estos son transformados en modelos de diseño que describen el sistema utilizando conceptos de tecnologías específicas y son traducidos a código.

2.2 Características

Los elementos clave de la iniciativa MDD fueron identificados en [**9**] de la siguiente forma:

Representación directa: Consiste en desplazar el enfoque del desarrollo desde el dominio tecnológico hacia las ideas y conceptos del dominio del problema. De esta forma se logra un mayor nivel de abstracción en la especificación tanto del problema a resolver como de la solución correspondiente, en relación con los métodos tradicionales de desarrollo de software. Así se reduce la distancia semántica entre el dominio del problema y su representación.

En el enfoque MDD, esto se consigue mediante la definición de lenguajes de modelado específicos del dominio. Estos lenguajes son cercanos al dominio de problema, y por lo tanto ocultan o minimizan los aspectos relacionados con las tecnologías de

implementación, además de utilizar formas sintácticas que transmiten fácilmente la esencia de los conceptos del dominio. Además, la utilización de modelos permite reducir el impacto de la evolución tecnológica en el desarrollo. Un mismo modelo abstracto puede ser materializado en diferentes plataformas de software.

Automatización: El aumento de confianza en la automatización asistida por computadora para soportar el análisis, el diseño y la ejecución. La automatización es un método eficaz para aumentar la productividad y mejorar la calidad. En MDD se usan herramientas computarizadas para automatizar los aspectos del desarrollo de software que no necesitan intervención humana. Por ejemplo, los modelos específicos del dominio, expresados en conceptos de alto nivel, son transformados automáticamente en programas informáticos ejecutables sobre una plataforma específica. Además, esta transformación automática podría aplicar patrones y técnicas conocidas, lo cual favorece la confiabilidad de los resultados.

Estándares abiertos: Los estándares han sido una de las formas más efectivas de acelerar el progreso en la historia de la tecnología. Los estándares industriales permiten reducir diversidad innecesaria en las herramientas utilizadas. El desarrollo de código abierto ayuda a que los estándares sean implementados consistentemente e impulsa la adopción de estos.

MDD se implementa mediante estándares industriales abiertos. Los estándares permiten a los fabricantes de herramientas centrar su atención en su principal área de experticia, sin tener que recrear y competir con funcionalidades implementadas por otros proveedores.

El poder de MDD está en construir modelos que representan directamente los conceptos del dominio. Con el uso de frameworks que modelan explícitamente asunciones del dominio de aplicación y el ambiente de implementación, las herramientas automatizadas pueden analizar los modelos y transformarlos en implementaciones, evitando la necesidad de escribir grandes cantidades de código y eliminando los errores causados por la intervención humana. De esta forma se aumenta la productividad en el desarrollo; se evita escribir manualmente una gran cantidad de código repetitivo que implementa patrones estándares, siendo este generado automáticamente.

2.3 Modelos

El modelo de un sistema es una conceptualización del dominio del problema y de su solución [1]. Los modelos están basados en el mundo real: identifican, clasifican y abstraen los elementos que constituyen el problema organizándolos en una estructura formal.

MDD identifica los siguientes tipos de modelos, según el nivel de abstracción de estos: **El modelo independiente de la computación (CIM, Computation Independent Model):** Un CIM representa las funciones del sistema sin mostrar detalles de la estructura del mismo. Se lo suele llamar modelo del dominio, y en su construcción se utiliza un vocabulario que resulte familiar para los expertos del dominio. El CIM juega un papel muy importante para reducir la brecha entre los profesionales del dominio y los desarrolladores encargados de implementar el sistema.

El modelo independiente de la plataforma (PIM, Platform Independent Model): Un PIM es un modelo del sistema con un alto nivel de abstracción que es independiente de cualquier tecnología o lenguaje de implementación, de manera de luego poder ser proyectado a una o más plataformas. Esta abstracción suele lograrse mediante la definición de un conjunto de servicios cuyos detalles técnicos no son descritos, sino que serán luego especificados por otros modelos de manera dependiente de la plataforma.

El modelo específico de la plataforma (PSM, Platform Specific Model): Como siguiente paso, un PIM se transforma en uno o más PSMs. Un PSM combina las especificaciones del PIM con los detalles requeridos para especificar cómo el sistema usa una plataforma específica. Cada PSM, entonces, representa la proyección del PIM en una plataforma. Un PIM puede generar múltiples PSMs, cada uno para una tecnología en particular. Generalmente, los PSMs deben colaborar entre sí para lograr una solución completa y consistente.

El modelo de la implementación (Código). En el último paso en el desarrollo se transforma cada PSM en código fuente. Como el PSM está orientado al dominio tecnológico específico, esta transformación es bastante directa.

2.3.1 Transformaciones de Modelos

Una herramienta que soporte MDD, toma un PIM como entrada y lo transforma en un PSM. La misma herramienta u otra tomará ese PSM y lo transformará a código. Estas transformaciones son esenciales en el proceso de desarrollo de MDD. En la Figura 2.1: Definiciones de transformaciones dentro de las herramientas de transformación se muestra la herramienta de transformación como una caja negra, que toma un modelo de entrada y produce

otro como salida.

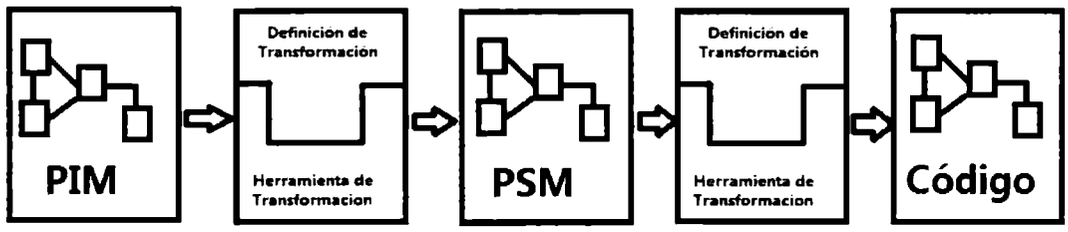


*Figura 2.1: Definiciones de transformaciones dentro de las herramientas de transformación [**1**]*

En algún lugar dentro de la herramienta hay una definición que describe cómo se debe transformar el modelo fuente para producir el modelo destino. Esta es la definición de la transformación. Hay una diferencia entre la transformación misma, que es el proceso de generar un nuevo modelo a partir de otro, y la definición de la transformación. Se podría por ejemplo definir una transformación que relaciona elementos de UML con elementos Java, y esta describiría cómo se pueden generar los elementos Java a partir de los elementos UML. Una transformación entre modelos puede verse como un programa que toma un modelo como entrada y produce un modelo como salida. Por lo tanto las transformaciones podrían implementarse utilizando cualquier lenguaje de programación. Para simplificar la tarea se han desarrollado lenguajes específicos del dominio de las transformaciones, tales como ATL y Acceleo, que serán descritos en detalle en el capítulo 5.

2.4 Metamodelos

Para poder realizar automáticamente las transformaciones, un modelo necesita tener un significado bien definido. En una transformación MDD, cada modelo, tanto fuente como destino, está expresado en un determinado lenguaje. Estos dos lenguajes deben estar definidos de alguna forma precisa y que ofrezca un nivel de abstracción adecuado. Se puede definir su sintaxis y su semántica mediante la construcción de un modelo del lenguaje de modelado, al cual llamamos metamodelo [**8**]. Por ejemplo, el estándar UML está escrito en UML (el metamodelo de UML [**4**]). Este contiene los elementos para describir modelos UML, como Class, Package, Operation, etc. El metamodelo de UML define también las relaciones y restricciones de estos conceptos.

Entonces, un metamodelo es un modelo que especifica los conceptos de un lenguaje, las relaciones entre ellos y las reglas estructurales que restringen los posibles elementos de los modelos válidos, así como aquellas combinaciones entre elementos que respetan las reglas semánticas del dominio. [**10**] Un metamodelo es también un modelo, por lo que debe estar escrito en un lenguaje bien definido: este lenguaje se denomina metalenguaje (un ejemplo de metalenguaje es BNF).

El metamodelo describe la sintaxis abstracta del lenguaje. Esta sintaxis es la base para el procesamiento automatizado de los modelos, mientras que la sintaxis concreta es definida mediante otros mecanismos y no es relevante para las herramientas de transformación de modelos. El metamodelo y la sintaxis concreta de un lenguaje pueden mantener una relación 1 a n, es decir que la misma sintaxis abstracta (definida por el metamodelo) puede ser visualizada a través de diferentes sintaxis concretas.

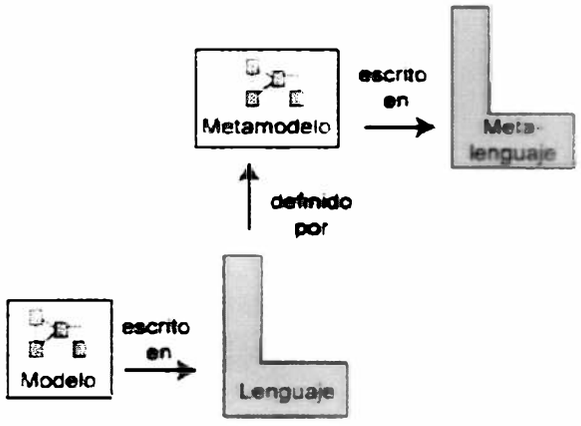


*Figura 2.2: Relación entre modelo y metamodelo [**1**]*

2.5 La Arquitectura Dirigida por Modelos (MDA)

El metamodelado es entonces un mecanismo que permite definir formalmente lenguajes de modelado. El OMG (Object Management Group) [**11**] propuso una arquitectura de cuatro capas de modelado, con el objetivo de estandarizar conceptos relacionados al modelado, desde los más abstractos a los más concretos.

Los 4 niveles definidos en esta arquitectura se denominan M3, M2, M1 y M0 (del más abstracto al más concreto), como se observa en la Figura 2.3.

Nivel M0: Instancias

En el nivel M0 se encuentran los objetos de la aplicación. En este nivel no se habla de clases, ni atributos, sino de entidades físicas que existen en el sistema.

Nivel M1: Modelo del sistema

Por encima de la capa M0 se encuentra la capa M1, que representa el modelo de un sistema de software. Los conceptos del nivel M1 representan categorías de las instancias de M0. Es decir, cada elemento de M0 es un caso de un elemento de M1.

Nivel M2: Metamodelo

Análogamente a lo que ocurre con las capas M0 y M1, los elementos del nivel M1 son a su vez instancias del nivel M2. Esta capa recibe el nombre de metamodelo.

Nivel M3: Meta-metamodelo

De la misma manera podemos ver los elementos de M2 como instancias de otra capa, la capa M3 o capa de meta-metamodelo. Un meta-metamodelo es un modelo que define el lenguaje usado para representar un metamodelo. La relación entre un meta-metamodelo y un metamodelo es análoga a la relación entre un metamodelo y un modelo.

M3 es el nivel más abstracto, que permite definir metamodelos concretos. Dentro del OMG, MOF es el lenguaje estándar de la capa M3. Esto supone entonces que todos los metamodelos de la capa M2 son instancias de MOF.

Según la propuesta de la OMG, el procedimiento recursivo de definir modelos conforme a otros modelos de mayor grado de abstracción acaba cuando se alcanza el nivel de meta-metamodelo, ya que los meta-metamodelos se dice que son conformes a ellos mismos.

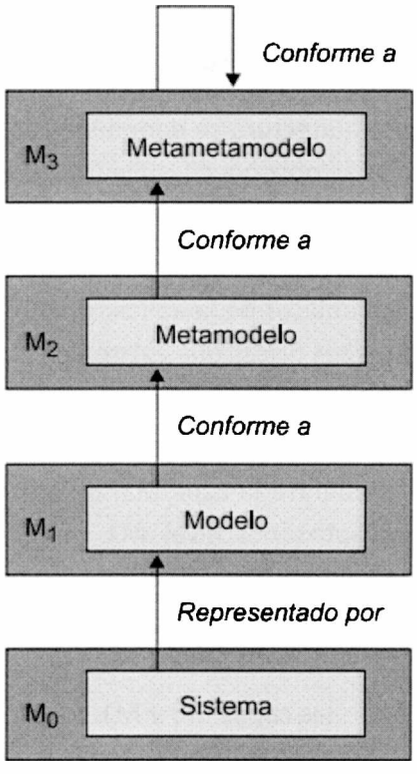


*Figura 2.3 : Arquitectura de 4 capas de modelado de la OMG [**10**]*

or último, en la Figura 2.4 se puede ver un ejemplo concreto de las cuatro capas de rquitectura de modelado, indicando las relaciones entre los elementos en iferentes capas. Se observa que en la capa M3 se encuentra el meta-metamodc 1OF, a partir del cual se pueden definir distintos metamodelos en el nivel M2, cor IML, Java, OCL, etc. Instancias de estos metamodelos serán los elementos del ni 11, como modelos UML, o modelos Java. A su vez, instancias de los elementos M erán los objetos que cobran vida en las ejecuciones del sistema.

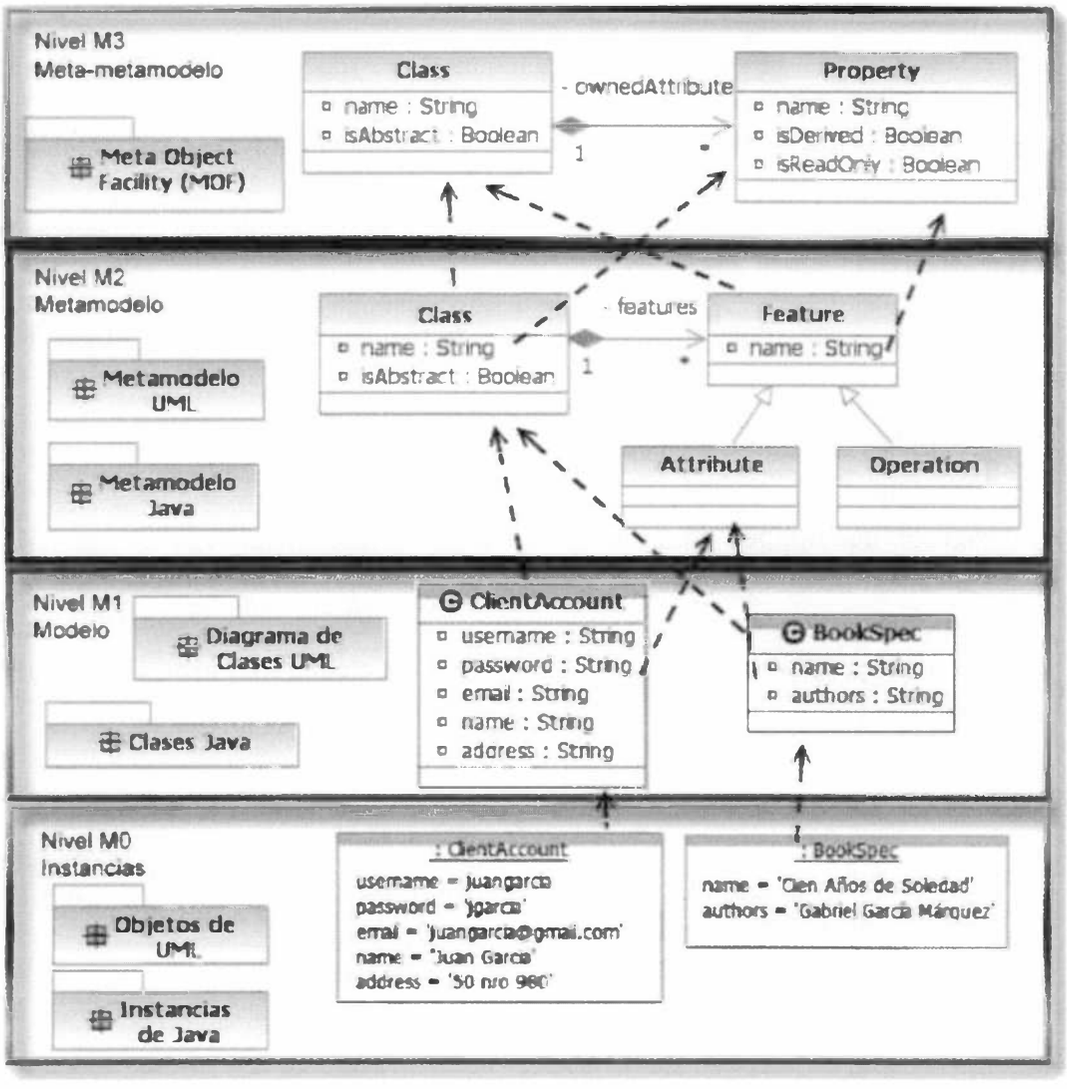


*Figura 2.4: Vista general de las relaciones entre los cuatro niveles [**1**]*

2.6 El lenguaje MOF

El lenguaje MOF (por Meta-Object Facility) [**12**] es el lenguaje de la OMG para describir metamodelos. MOF surgió con el objetivo de proveer un marco formal para la definición de UML y otros lenguajes gráficos. Como se vio anteriormente, MOF se ubica en la capa superior de la arquitectura de 4 capas, y por lo tanto provee un meta-metalenguaje que permite definir metamodelos en la capa M2. El ejemplo más conocido de un elemento en la capa M2 es el metamodelo UML, que describe al lenguaje UML.

Tal como su nombre lo indica, MOF se basa en el paradigma de Orientación a Objetos. Por este motivo usa los mismos conceptos y la misma sintaxis concreta que los diagramas de clases de UML.

2.7 Resumen

En este capítulo se describieron los elementos fundamentales que constituyen al proceso MDD. En particular se destacan:

- Modelos y sus distintos niveles de abstracción, escritos en un lenguaje estándar y bien definido.
- Lenguajes que nos permiten expresar esos modelos.
- La arquitectura de cuatro capas del modelado de OMG, en la cual cada nivel se instancia a partir del nivel inmediato superior.
- Las transformaciones que convierten a un modelo en uno más específico, y sus definiciones.

3 El lenguaje OCL

3.1 Introducción

Un modelo puede ser enriquecido mediante información adicional o restricciones sobre sus elementos. Esta información suele ser expresada mediante el lenguaje natural. Pero este es ambiguo. Para resolver este problema se han desarrollado los llamados lenguajes formales. La desventaja de los lenguajes formales tradicionales es que son difíciles de usar para personas que no tengan fuertes conocimientos matemáticos. OCL (Object Constraint Language, lenguaje de restricciones de objetos) [**13**] fue desarrollado como alternativa a estos. Es un lenguaje formal más fácil de leer y escribir; por su naturaleza orientada a objetos, resulta fácil de entender para personas con conocimientos del paradigma.

OCL es utilizado para describir expresiones en modelos UML. Típicamente estas expresiones especifican condiciones invariantes que deben valer para el sistema a modelar, o consultas sobre los objetos descritos en un modelo.

OCL es puramente un lenguaje de especificación. Por lo tanto, las expresiones OCL no tienen efectos laterales, es decir, su evaluación no puede alterar el estado del sistema. Pero puede usarse para especificar operaciones que, al ser ejecutadas, sí alteran el estado del sistema. Cuando se modela en UML se puede usar OCL para especificar restricciones específicas de la aplicación en sus modelos y para especificar operaciones del modelo UML, que son independientes del lenguaje de programación.

OCL no es un lenguaje de programación, por lo que no es posible utilizarlo para escribir lógica de programación o flujo de control. Esto se debe a que OCL es un lenguaje de modelado, por lo que sus expresiones no son ejecutables directamente.

Algunos de los usos de OCL son los siguientes:

- Como lenguaje de consultas
- Especificar invariantes en clases y tipos de un modelo de clases
- Especificar invariantes de tipo para Estereotipos
- Describir pre y post condiciones en Operaciones y Métodos
- Describir Guardias
- Especificar destinos para mensajes y acciones

- Especificar restricciones en operaciones
- Especificar reglas de derivación de atributos para cualquier expresión de un modelo UML

A continuación se describen las características principales de este lenguaje.

3.2 Restricciones en OCL

3.2.1 Invariantes

Una expresión OCL que es parte de un invariante para un elemento debe resultar verdadera para todas las instancias del elemento en todo momento. La sintaxis de la definición de un invariante es la siguiente:

```
context [NombreVariable:] NombreTipo
inv: --expresion OCL
```

El elemento sobre el cual predica la expresión OCL es parte del invariante y se escribe luego de la palabra clave *context* seguido de su nombre. La etiqueta *inv* indica que la restricción es un invariante. Opcionalmente el invariante puede llevar un nombre.

3.2.2 Pre y Post condiciones

Una expresión OCL puede también ser parte de una Precondición o Postcondición, asociada a una operación o método. Se definen agregando "pre" o "post" a su declaración según corresponda.

Las precondiciones establecen condiciones que deben cumplirse antes de ejecutar la operación, y las postcondiciones establecen condiciones que deben cumplirse después de su ejecución.

En general, la sintaxis para definir pre y post condiciones es la siguiente:

```
context Nombretipo::nombreOperacion(param1 : Tipo1, ... ): TipoRetorno
pre: expresionOcl
post: expresionOcl
```

En estas expresiones se puede acceder a los parámetros de la operación. Además, en una expresión OCL utilizada como postcondición se puede agregar el sufijo *@pre* a un elemento para hacer referencia a su valor al comienzo de la operación. La variable *result* hace referencia al valor de retorno de la operación.

3.2.3 Expresión Body

Una expresión OCL puede usarse para indicar el resultado de una operación de consulta. La sintaxis para estas definiciones es la siguiente:

```
context Nombretipo :: nombreOperacion(param1 : Tipo1, ... ) :
TipoRetorno
body: -- expresion OCL
```

Esta expresión debe ajustarse al tipo de resultado de la operación. Al igual que en las pre y postcondiciones, se pueden utilizar los parámetros en la expresión. Una expresión *body* se puede combinar con pre y postcondiciones para la misma operación.

3.2.4 Valores iniciales y derivados

Se puede usar una expresión OCL para indicar el valor inicial o derivado de un atributo o del extremo de una asociación. Esto se logra mediante la siguiente sintaxis:

```
context Nombretipo :: nombreOperacion(param1 : Tipo1, ... ) :
TipoRetorno
init: --expresion para representar el valor inicial

context Nombretipo :: nombreOperacion(param1 : Tipo1, ... ) :
TipoRetorno
derive: --expresion para representar la regla de derivacion
```

La expresión debe ajustarse al tipo de resultado del atributo. Si el contexto es un extremo de asociación, la expresión debe ajustarse al clasificador cuando la multiplicidad a lo sumo 1, o a un tipo de conjunto si esta puede ser mayor.

3.2.5 Paquetes

En caso de que no esté claro a qué paquete pertenece el clasificador al que hacen referencia las restricciones descritas arriba, esto se puede especificar explícitamente, mediante la siguiente sintaxis:

```
package Paquete::SubPaquete

context X inv:
      ...algún invariante ...
context X::nombreOperación(...)
      pre: ...alguna precondición...
endpackage
```

Un archivo OCL puede incluir varias declaraciones de paquetes, permitiendo de esta forma que todas sus restricciones se escriban en un solo archivo. Este archivo puede existir junto a un modelo UML como entidad separada.

3.3 Tipos y valores básicos

OCL es un lenguaje tipado; cada expresión OCL tiene un tipo. Para estar bien formada, una expresión OCL debe ajustarse a las reglas del tipo del lenguaje.

OCL cuenta con un conjunto de tipos básicos independientes del modelo. Estos tipos son predefinidos e incluyen un conjunto de operaciones básicas asociadas. Se los puede observar en la tabla Tabla 3.1.

Tipo	Valores	Consistente con las definiciones de implementación
OclInvalid	invalid	
OclVoid	null, invalid	
Boolean	true, false	(MOF) http://www.w3.org/TR/xmlschema-2/#boolean
Integer	1, -5, 2, 34, 26524, ...	(MOF) http://www.w3.org/TR/xmlschema-2/#integer
Real	1.5, 3.14, ...	http://www.w3.org/TR/xmlschema-2/#double
String	"Ser o no ser..."	(MOF) http://www.w3.org/TR/xmlschema-2/#string
UnlimitedNatural	0, 1, 2, 42, ..., *	http://www.w3.org/TR/xmlschema-2/#nonNegativeInteger

*Tabla 3.1: Tipos básicos de OCL [**13**]*

Algunos ejemplos de operaciones predefinidas para estos tipos pueden verse en la Tabla 3.2

Tipo	Operaciones
Integer	*, +, -, /, abs()
Real	*, +, -, /, floor()
Boolean	and, or, xor, not, implies, if-then-else
String	concat(), size(), substring()
UnlimitedNatural	*, +, /

*Tabla 3.2: Ejemplos de operaciones de los tipos predefinidos [**13**]*

Además, como una expresión OCL está escrita en el contexto de un modelo UML, todos los clasificadores de este (tipos/clases...) son también tipos utilizables en la expresión.

3.3.1 Valores Inválidos

Algunas expresiones, luego de ser evaluadas, resultan en un valor inválido. Esto sucede, por ejemplo, cuando se intentar obtener el primer elemento de una colección vacía. La validez de una expresión se puede consultar con los operadores *oclIsInvalid()* y *oclIsUndefined()*.

3.3.2 Ajuste de tipos

OCL es un lenguaje tipado y sus tipos básicos están organizados en una jerarquía de tipos, la cual determina cómo se ajustan unos a otros. No se puede, por ejemplo, comparar un Integer con un Boolean o String.

Una expresión OCL es válida si todos sus tipos se ajustan correctamente, e inválida en caso contrario. Un tipo T1 se ajusta a un tipo T2 cuando una instancia de T1 puede ser sustituida en cualquier lugar donde se espere una instancia de T2. Las reglas de ajuste de tipos son las siguientes:

- Cada tipo se ajusta a cada uno de sus supertipos.
- El ajuste de tipos es transitivo: si un tipo T1 se ajusta a un tipo T2, y T2 se ajusta a un tipo T3, entonces T1 se ajusta a T3.

Las reglas de ajuste de los tipos de la librería estándar OCL se muestran en la Tabla 3.3. Cuando se trata de dos tipos de colecciones, las cuales se verán más adelante, la relación de ajuste se mantiene solamente si son colecciones de elementos cuyos tipos se ajustan entre ellos.

Tipo	Se ajusta a/Es subtipo de	Condición
Set(T1)	Collection(T2)	Si T1 se ajusta a T2
Sequence(T1)	Collection(T2)	Si T1 se ajusta a T2
Bag(T1)	Collection(T2)	Si T1 se ajusta a T2
OrderedSet(T1)	Collection(T2)	Si T1 se ajusta a T2
Integer	Real	
UnlimitedNatural	Integer	* es un entero inválido

*Tabla 3.3: Reglas de ajuste de tipos [**13**]*

3.3.3 Expresiones Let

A veces se usa una sub-expresión más de una vez en una restricción. La expresión *let* nos permite definir una variable que se puede usar en la restricción. Por ejemplo:

```
context Persona inv:
let ingresos : Integer = self.trabajo.salario->sum() in
if esDesempleado then
ingresos < 100
else
      ingresos >= 100
endif
```

Una expresión let se puede incluir en cualquier tipo de expresión OCL y sólo se la puede acceder desde esa expresión específica. Una declaración de variable en un let debe incluir un tipo y un valor inicial.

3.3.4 Expresiones de definición

Para reutilizar variables u operaciones en múltiples expresiones OCL se puede usar una restricción de definición. Esta restricción debe estar unida a un Clasificador y sólo puede incluir definiciones de variables u operaciones. Estas se utilizan del a misma forma que cualquier atributo u operación normal. La sintaxis es la siguiente:

```
context Persona
def: ingresos : Integer = self.trabajo.salario -> sum()
def : tieneTitulo(t : String) : Boolean = self.trabajo->exists(titulo
= t)
```

3.4 Colecciones

OCL maneja colecciones, esenciales a la hora de navegar por asociaciones del modelo. El tipo abstracto Collection define una variedad de operaciones predefinidas para permitirle al modelador manipular las colecciones. Collection tiene como subtipos los siguientes tipos concretos:

- Set: Actúa como el conjunto matemático, por lo que no tiene elementos repetidos.
- OrderedSet: Es como un Set, pero sus elementos tienen un orden
- Bag: Es como un conjunto pero puede tener elementos repetidos.

- Sequence: Es como un Bag pero sus elementos tienen un orden

Las características de estos tipos de colecciones están resumidas en la Tabla 3.4.

Tipo de Colección	Ordenada	Elementos únicos
Set	No	Sí
OrderedSet	Sí	Sí
Bag	No	No
Sequence	Sí	No

Tabla 3.4: Características de las colecciones OCL

OCL permite también que los elementos de una colección sean colecciones.

3.4.1 Operaciones de Colecciones

OCL define muchas operaciones para los tipos de colecciones. A continuación se describen las más importantes.

Select y Reject

Estas operaciones se utilizan para obtener un subconjunto de una colección. La operación *select* tiene la siguiente sintaxis:

```
colección -> select ( v | expresión-lógica-con-v )
```

El parámetro de *select* tiene una sintaxis especial que nos permite especificar qué elementos de la colección seleccionar. A la variable v se la llama el iterador. Cuando se evalúa el *select*, se itera por la colección, de forma que *v* hace referencia a cada objeto de la colección. Por cada *v* se evalúa la expresión. Si da verdadera, entonces *v* estará en la colección resultante.

Se puede escribir también el parámetro omitiendo *v*, o declarando explícitamente su tipo.

La operación *reject* existe por comodidad, ya que se comporta de forma opuesta a *select*; el subconjunto resultante está formado por los elementos que hacen falsa a la expresión lógica.

Collect

Esta operación se utiliza para definir una colección derivada a partir de otra, que contiene objetos diferentes a la original (es decir, no es una sub-colección). La sintaxis es la misma que la de *select* y *reject*. El resultado de una operación *collect* es la colección de los resultados obtenidos luego de evaluar expresión-lógica-con v sobre cada v. Por ejemplo, la expresión siguiente:

```
self.empleados->collect( persona | persona.fechaDeNacimiento)
```

Retorna como resultado una colección con la fecha de nacimiento de cada empleado de self.empleados.

ForAll y Exists

La operación *forAll* en OCL nos permite especificar una expresión booleana que debe valer para todos los objetos en la colección. Su sintaxis es similar a la de las operaciones anteriores. Una expresión *forAll* da como resultado un valor Booleano, que es verdadero si la expresión lógica usada como parámetro es verdadera para todos los elementos de la colección, y falso en caso contrario.

La operación *exists* es similar a *forAll*, con la diferencia que se evalúa como verdadera si al menos un elemento cumple con la condición.

Tanto *forAll* como *exists* pueden usar más de una variable como iterador, como se muestra en el siguiente ejemplo:

```
self.empleados->forAll( e1, e2 : Persona | e1 <> e2 implies
e1.apellido <> e2.apellido)
```

Esta expresión es verdadera si no hay dos empleados con el mismo apellido.

Iterate

La operación *iterate* es más compleja que las anteriores, pero también más genérica. Todas las operaciones vistas anteriormente se pueden describir en términos de un *iterate*.

```
coleccion->iterate( elem : Type; acu : Type = <expresión> | expresión-
con-elem-y-acu )
```

La variable *elem* es el iterador, y *acu* es el acumulador. El acumulador tiene como valor inicial *<expresión>*. Cuando se evalúa el *iterate*, *elem* itera sobre la colección, se evalúa

expresión-con-elem-y-acu para cada *elem*, y su resultado se le asigna a *acu*. De esta forma, el valor de *acu* se construye durante la iteración de la colección.

3.5 Objetos y Propiedades

Las expresiones OCL pueden hacer referencia a clasificadores, es decir, tipos, clases, interfaces, asociaciones y tipos de datos. Todos los atributos, extremos de asociaciones, métodos y operaciones sin efectos laterales también pueden ser usados. En un modelo de clases, se puede definir que una operación o método es libre de efectos laterales mediante el atributo *isQuery*. Se consideran propiedades a los siguientes:

- Un atributo.
- Un extremo de asociación.
- Una operación con el atributo *isQuery* en verdadero.
- Un método con *isQuery* en verdadero.

Las propiedades se referencian con un punto seguido del nombre de la propiedad (Por ejemplo, self.esCasado).

3.5.1 Propiedades: Atributos

Por ejemplo, la edad de una Persona podría escribirse como *self.edad*. El valor de la expresión *self.edad* es el valor del atributo edad, y el tipo de la expresión es el del atributo (por ejemplo, Integer).

Los atributos pueden tener multiplicidad mayor a uno, en cuyo caso su tipo es el de una colección de valores.

3.5.2 Propiedades: Operaciones

Las operaciones pueden tener parámetros. Por ejemplo, si un objeto Persona tiene un salario expresado en función de la fecha, la operación se accedería de la siguiente forma: *unaPersona.salario(unaFecha)*.

El resultado de una operación es un valor del tipo de retorno de la operación. Si la operación tiene parámetros de salida o entrada/salida, el resultado entonces es una tupla que contiene estos parámetros y el valor de retorno.

3.5.3 Propiedades: Extremos de Asociación y Navegación

A partir de un objeto se puede navegar una asociación del diagrama de clases para acceder a otros objetos y sus propiedades. Para hacerlo se utiliza el extremo opuesto de la asociación:

```
objeto.nombreExtremoAsociacion
```

El valor de esta expresión es el conjutno de objetos del otro lado de la asociación nombreExtremoAsociacion. Si la multiplicidad del extremo de asociación tiene como valor máximo 1, el valor de esta expresión es un objeto.

3.5.4 Propiedades predefinidas en todos los objetos

Las siguientes propiedades predefinidas se aplican a todos los objetos:

- oclIsTypeOf(t : Classifier) : Boolean
- oclIsKindOf(t : Classifier) : Boolean
- oclIsInState (s : OclState) : Boolean
- oclIsNew () : Boolean
- oclAsType (t : Classifier) : instance of Classifier

La operación *oclIsTypeOf(t)* es verdadera si el tipo de *self* y *t* son el mismo, mientras que *oclIsKindOf(t)* determina si *self* es de tipo *t* o de uno de sus subtipos.

La operación *oclInState(s)* es verdadera si el objeto está en el estado *s*. Los estados posibles son los de la máquina de estados que define el comportamiento del clasificador.

La operación *oclIsNew()* es verdadera si, al usarla en una postcondición, el objeto es creado durante la ejecución de la operación (es decir, no existía durante el momento de precondición).

Y por último *oclAsType(t)* se utiliza para convertir al objeto origen en un objeto de tipo *t*, siendo este un subtipo o supertipo del objeto origen.

3.6 Resumen

En este capítulo se dio una primera visión del lenguaje OCL, en particular:

- El objetivo de lenguaje, sus características principales y sus usos.
- Los tipos de restricciones que maneja el lenguaje, incluyendo invariantes, precondiciones y postcondiciones.
- Los tipos que usa OCL y cómo se ajustan entre ellos.
- Las colecciones de OCL y sus operaciones más importantes.

4 El lenguaje JML

4.1 Introducción

JML [6] es una notación para la especificación formal de clases y métodos Java [**14**]. JML nos permite especificar tanto la interfaz sintáctica del código Java (nombres, visibilidad, chequeo de tipos...), como su comportamiento.

JML le agrega el concepto de Design-by-Contract (Diseño-por-Contrato) [**7**] a Java. DBC es un método para el desarrollo de software, cuya idea principal es que una clase y sus clientes tienen un "contrato" entre ellos. El cliente debe cumplir ciertas condiciones antes de llamar a un método, y la clase garantiza que ciertas propiedades se mantendrán luego de la invocación de este. Los contratos están escritos en el lenguaje en sí, y son compilados a código ejecutable. Entonces, se pueden detectar las violaciones a estos inmediatamente. JML implementa DBC mediante cláusulas para precondiciones y postcondiciones de métodos, e invariantes de clases.

Las especificaciones JML se pueden agregar directamente al código Java, utilizando comentarios especiales llamadas anotaciones. De esta forma se puede especificar el comportamiento de los métodos de manera independiente de su implementación.

JML sirve además como documentación formal del código, ya que no sólo especifica el comportamiento del programa, sino que además se puede comprobar de forma automática.

Una de las ventajas de JML es que, al usar notación Java en su especificación, es más fácil de aprender para los programadores que, por ejemplo, lenguajes formales que usen términos matemáticos. Como Java no tiene la expresividad necesaria para un lenguaje de especificación, JML extiende las expresiones Java con construcciones específicas, como por ejemplo cuantificadores.

A continuación se describen las características principales del lenguaje.

4.2 Características principales

Las especificaciones JML pueden escribirse en los archivos .java donde la clase esté definida, o en archivos .jml separados. La forma de incluirlas es mediante anotaciones

en comentarios, por lo que son ignoradas por el compilador de Java. Pueden agregarse entre /*@ . . . */ o luego de //@

En la Tabla 4.1 se pueden ver algunas de las extensiones de JML a las expresiones Java. La restricción principal de JML es que las expresiones usadas en sus casos de especificación no pueden tener efectos laterales. Por lo tanto, las expresiones Java de asignación (=,+=, etc) e incremento (++) y decremento (--) no pueden ser usadas. Por la misma razón, solamente se pueden invocar métodos puros desde una expresión JML. Un método es puro si no tiene efectos laterales en el estado del programa. En JML se puede especificar que un método es puro mediante la palabra clave *pure*.

Sintaxis	Significado
\result	resultado de la invocación de un método
a ==> b	a implica b
a <== b	a se deduce de b (b implica a)
a <==> b	a si y sólo si b
a <=!=> b	no (a si y sólo si b)
\old(E)	Valor de E en el pre-estado

*Tabla 4.1: Algunas de las extensiones de JML para las expresiones Java [**15**]*

4.2.1 Precondiciones y Postcondiciones

El significado de las especificaciones JML de métodos es el siguiente: Un método debe ser invocado en un estado donde su precondición sea satisfecha; de no ser así, no se garantiza nada (la invocación podría nunca terminar, o hacer cambios arbitrarios en el estado). Al estado en el que se llama un método se lo llama pre-estado. Si un método se llama en un pre-estado apropiado, entonces hay dos posibles resultados para su ejecución. El método puede terminar normalmente, o generar una excepción. Si termina normalmente sin lanzar una excepción, entonces a ese estado se lo llama un post-estado normal. Si termina lanzando una excepción que no herede de la clase *java.lang.Error*, entonces a ese estado de terminación se lo llama post-estado excepcional. En el post-estado normal, se debe satisfacer la post-condición normal del método. Asimismo, en un post-estado excepcional, la excepción lanzada debe estar permitida por la cláusula *signals_only* de la especificación, y el post-estado excepcional

debe satisfacer las postcondiciones excepcionales correspondientes (definidas en las cláusulas *signals*).

JML usa la cláusula *requires* para especificar las precondiciones, y la cláusula *ensures* para especificar las postcondiciones. En las postcondiciones además se puede usar la palabra clave *\old* para hacer referencia al valor de una expresión en el pre-estado, y *\result* para hacer referencia al valor de retorno del método.

En la Figura 4.1, se puede ver un ejemplo de código Java con anotaciones JML que calcula la raíz cuadrada de un número real. La precondición *requires x >= 0.0* especifica que para garantizar el resultado esperado, el parámetro *x* debe ser mayor o igual a 0. Mientras que la postcondición *ensures JMLDouble.approximatelyEqualTo(x, \result * \result, eps)* garantiza que el valor de retorno del método sea aproximadamente igual a la raíz cuadrada de *x* (teniendo en cuenta errores de precisión).

```
public class SqrtExample {

    public final static double eps = 0.0001;

    //@ requires x >= 0.0;
    //@ ensures JMLDouble.approximatelyEqualTo(x, \result * \result,
eps);
    public static double sqrt(double x) {
        return Math.sqrt(x);
    }
}
```

Figura 4.1: Especificación JML de un método para calcular la raíz cuadrada [16]

4.2.2 Invariantes

Un invariante es una propiedad que debe valer en todos los estados visibles al cliente. Debe ser verdadero cuando el control no está dentro de los métodos del objeto. Entonces, un invariante debe valer al final de la ejecución de cada constructor, y al principio y el final de la ejecución de todos los métodos [**17**].

En JML, una cláusula de invariante público nos permite definir los estados aceptables de un objeto que sean visibles al cliente. A estos invariantes se los suele llamar **invariantes de tipo.** También JML nos permite especificar invariantes con visibilidad más restringida; como estos no son visibles al cliente, a veces se los llama **invariantes de representación**. Los invariantes de representación se pueden usar para definir los

estados internos aceptables de un objeto; por ejemplo, que una lista enlazada es circular, o decisiones de diseño. También se pueden tener invariantes públicos sobre campos privados pero públicos para la especificación, de forma que son invariantes tanto de tipo como de representación. Para que un campo sea público para la especificación se utiliza el modificador *spec_public*.

En la Figura 4.2 se puede ver el código con especificaciones JML de una clase Persona. El invariante *public invariant !name.equals("")* especifica que el nombre de una persona no puede ser vacío. Como dicho anteriormente, esto es desde el punto de vista del cliente; no va a valer antes de la ejecución del constructor, por ejemplo.

```
public class Person {
  private /*@ spec_public non_null @*/
    String name;

  /*@ public invariant !name.equals(""); @*/

                        \
  //@ also
  //@ ensures \result != null;
  public String toString();

  /*@ also
    @ requires n != null && !n.equals("");
    @ ensures n.equals(name)
    @   && weight == 0; @*/
  public Person(String n);
}
```

Figura 4.2: Especificación de la clase Persona. La palabra clave also indica que el método hereda especificaciones de sus supertipos

4.2.3 Modelo y Fantasma

En JML, se pueden declarar varios nombres con el modificador *model*, como por ejemplo métodos, campos y tipos.

Un campo modelo es un campo sólo visible al nivel de la especificación, que funciona como abstracción del estado concreto de un estado. El valor del campo modelo se determina a partir del campo concreto desde el cual se abstrae; esta relación se especifica mediante una cláusula *represents*.

Los métodos y tipos modelo no son abstracciones de métodos o tipos concretos, sino que simplemente son parte de la especificación.

El siguiente es un ejemplo del uso de campos modelo:

```
//@ public model non_null String name;
private /*@ non_null @*/ String fullName;
//@ private represents name <- fullName;
```

El campo modelo *name* funciona como abstracción de *fullName*, para usar en la especificación. Mediante la cláusula *represents* se establece la relación entre ambos campos. De esta forma *name* tomará el valor de *fullName*.

Un campo fantasma es similar a un campo modelo ya que sólo es visible y utilizable por la especificación. La diferencia es que el valor de un campo fantasma no está determinado por una cláusla *represents*, sino que su valor se inicializa directamente. Los campos fantasmas se especifican mediante el modificador *ghost*.

4.2.4 Especificación de comportamiento excepcional

JML nos permite no sólo especificar el comportamiento de un método en el caso esperado, sino también en casos excepcionales. Se puede utilizar una anotación de *exceptional_behavior* para describir el comportamiento garantizado cuando se lanza una excepción. En la Figura 4.3 se muestra como ejemplo de los tipos de comportamiento la especificación del método *peek()* de una lista de prioridades.

```
/*@
   @ public normal_behavior
   @   requires ! isEmpty();
   @   ensures elementsInQueue.has(\result);
   @ also
   @ public exceptional_behavior
   @   requires isEmpty();
   @   signals (Exception e) e instanceof NoSuchElementException;
   @*/
/*@ pure @*/ Object peek() throws NoSuchElementException;
```

*Figura 4.3: Ejemplo de especificación del método peek() de una lista de prioridades [**18**]*

El comportamiento esperado, especificado con la palabra clave *normal_behavior*, sucede cuando la lista tiene al menos un elemento. La postcondición garantiza que el elemento retornado esté efectivamente en la lista.

El caso de que la lista esté vacía es considerado, especificado con la palabra clave *exceptional_behavior*. Es un caso excepcional debido a que no debería poder consultar un elemento de una lista vacía. La cláusula *signals* especifica las propiedades que se deben garantizar al final de la invocación del método en caso de que este termine abruptamente por el lanzamiento de una excepción. En el ejemplo, la propiedad que debe garantizarse es que la excepción lanzada sea de tipo *NoSuchElementException.*

4.2.5 Cuantificadores

JML soporta varios tipos de cuantificadores en sus especificaciones: un cuantificador universal (*\forall*), un cuantificador existencial (*\exists*), cuantificadores generalizados (*\sum*, *\product*, *\min* y *\max*) y un cuantificador numérico (*\num_of*). Por ejemplo, el predicado siguiente usa un cuantificador universal para especificar que todos los estudiantes del conjunto ingresantes tienen tutores:

```
(\forall Estudiante e;
ingresantes.contains(e);
s.getTutor() != null)
```

Un cuantificador tiene una declaración, como *Estudiante e*, de un nombre local al cuantificador. Luego le sigue un rango opcional, como *ingresantes.contains(e)* en el ejemplo, que restringe el dominio al que se aplica el cuantificador; en el ejemplo, el cuantificador sólo se aplica a estudiantes que sean ingresantes. Si el rango se omite, se aplica a todos los objetos posibles. Finalmente, el tercer predicado, el cuerpo del cuantificador, *s.getTutor() != null* en el ejemplo, debe valer para todos los objetos que satisfacen el rango del predicado.

Los cuantificadores *\max*, *\min*, *\product* y *\sum* retornan el máximo, mínimo, producto y suma respectivamente de los valores de la expresión cuerpo que son parte del rango. El cuantificador *\num_of* retorna la cantidad de valores para los que el rango y el predicado cuerpo son verdaderos.

4.2.6 Especificación de ciclos

JML también define una variedad de anotaciones que pueden ser mezcladas con sentencias Java en el cuerpo de un método, constructor o bloque de inicialización.

En JML una sentencia de ciclo puede ser anotada con uno o más invariantes de ciclo, y una o más funciones variantes.

Un invariante de ciclo es usado para probar corrección parcial de una sentencia de ciclo. La sintaxis es la siguiente:

```
//@ maintaining J;
  while (B) { S }
```

El predicado *J* debe valer al principio de cada iteración del ciclo.

Una función variante se usa para ayudar a probar la terminación de una sentencia de ciclo. Especifica una expresión de tipo *long* o *int* que no debe ser menor a 0 mientras se ejecuta el ciclo, y debe reducirse al menos en uno (1) en cada vuelta del ciclo. La sintaxis es la siguiente:

```
//@ decreasing E;
 while (B) { S }
```

En el caso de que el ciclo contenga una sentencia *continue*, el variante se chequea justo antes de cada uso de *continue*.

4.3 ¿Por qué usar JML?

JML es un lenguaje de especificación formal hecho a medida para Java. Su uso básico es la especificación del comportamiento de módulos Java. Los dos beneficios principales de usar JML son:

- La descripción precisa y no ambigua del comportamiento de módulos de programa Java (clases e interfaces) y documentación de código Java.
- La posibilidad de tener herramientas que lo soporten.

Una especificación JML puede describir el contrato de una interfaz y su comportamiento secuencial. Como es una especificación de interfaz, se pueden

especificar todos los detalles Java de dicha interfaz, como por ejemplo sus mecanismos de parámetro, si es *final*, *protected*, etc. Si no se usara un lenguaje de especificación hecho específicamente para Java, no se podría lograr una especificación a tal grado de detalle, lo cual podría causar problemas a la hora de integrar el código. Por ejemplo, en JML se puede especificar las condiciones precisas en las que se lanzan ciertas excepciones, cosa difícil de lograr en un lenguaje de especificación independiente de Java que no implemente la noción de excepción.

Además, el soporte más básico para la implementación de JML (análisis sintáctico y chequeo de tipos) es útil de por sí. Mientras que los comentarios informales en el código fácilmente pueden quedar desactualizados a medida que este cambia, una simple ejecución del chequeador de tipos de JML encuentra referencias a parámetros o campos que no existen, errores tipográficos, etc.

Pero el chequeo de tipos no es el único uso de JML. JML está diseñado para soportar análisis estático, verificación formal (como la herramienta LOOP [**19**]), chequeo en tiempo de ejecución [**17**], pruebas de unidad [**20**], y documentación (mediante la herramienta jmldoc). En el paper [**21**] se puede ver una recopilación de herramientas para JML.

4.4 Resumen

En este capítulo se describieron los elementos principales del lenguaje JML. Específicamente, se vieron:

- Las características principales de JML, y qué le agrega al lenguaje Java
- La semántica de sus especificaciones
- Algunos de los tipos de especificaciones, cláusulas, y expresiones específicas del lenguaje, que no pueden ser expresados en código Java
- Los usos y beneficios del lenguaje y ejemplos de herramientas desarrolladas en base a este.

5 Herramientas utilizadas

En este capítulo se provee una introducción a las herramientas utilizadas para el desarrollo del presente trabajo.

5.1 Eclipse Modeling Framework (EMF)

El proyecto EMF [**22**] es un framework para modelado y generación de código, que brinda soporte para el desarrollo de herramientas y otras aplicaciones a partir de modelos de datos estructurados. Consiste en un conjunto de plugins de Eclipse que pueden usarse para crear un modelo de datos y generar código u otro tipo de salida basada en ese modelo. EMF le permite al desarrollador crear un metamodelo mediante diferentes formas, por ejemplo, XMI, anotaciones Java, UML, etc. Dada una especificación de un modelo, EMF proporciona herramientas y soporte en tiempo de ejecución para producir un conjunto de clases Java para el modelo, junto con clases adaptadoras que permitan visualizarlo y editarlo.

En EMF los modelos se especifican usando un meta-metamodelo llamado Ecore. Ecore es una implementación de eMOF (Essential MOF). Ecore en sí es un modelo EMF y su propio metamodelo. Todas sus metaclases mantienen el nombre del elemento que implementan y agregan como prefijo la letra "E", indicando que pertenecen al metamodelo Ecore. Por ejemplo, la metaclase EClass implementa a la metaclase Class de MOF.

EMF soporta tres niveles de generación de código:

- **Modelo**: provee interfaces Java y clases de implementación para todas las clases del modelo, además de una clase Factory y una clase de implementación de paquete.
- **Adaptadores:** genera clases de implementación que adaptan las clases del modelo para la edición y visualización de estas.
- **Editor:** produce un editor estructurado que se ajusta al estilo recomendado de los editors de modelos EMF de Eclipse, y sirve como punto de partida para empezar a personalizar.

5.2 El Lenguaje ATL

ATL (ATL Transformation Language) [**23**] es en un lenguaje de transformación de modelos y un toolkit, creado por el grupo de investigación ATLAS INRIA & LINA. En el ámbito del MDD, ATL nos provee de formas de generar un conjunto de modelos destino a partir de un conjunto de modelos origen.

Es un lenguaje híbrido, que permite realizar tanto construcciones declarativas como imperativas. Aunque el estilo recomendado para las transformaciones es el declarativo, el lenguaje incluye construcciones imperativas para permitir de este modo la especificación de mapeos que sean demasiado complejos para ser manejados declarativamente.

Las transformaciones ATL son unidireccionales; parten de modelos de sólo lectura y producen modelos de sólo escritura. En la Figura 5.1 se muestra cómo es una transformación ATL: Un modelo fuente Ma, basado en el metamodelo MMa, se transforma en un modelo destino Mb, basado en el metamodelo MMb, siguiendo la definición de transformación del archivo MMa2MMb.atl, escrita en el lenguaje ATL. La definición de la transformación es a su vez un modelo basado en el metamodelo ATL. Todos los metamodelos se ajustan al meta-metamodelo (En este ejemplo, MOF).

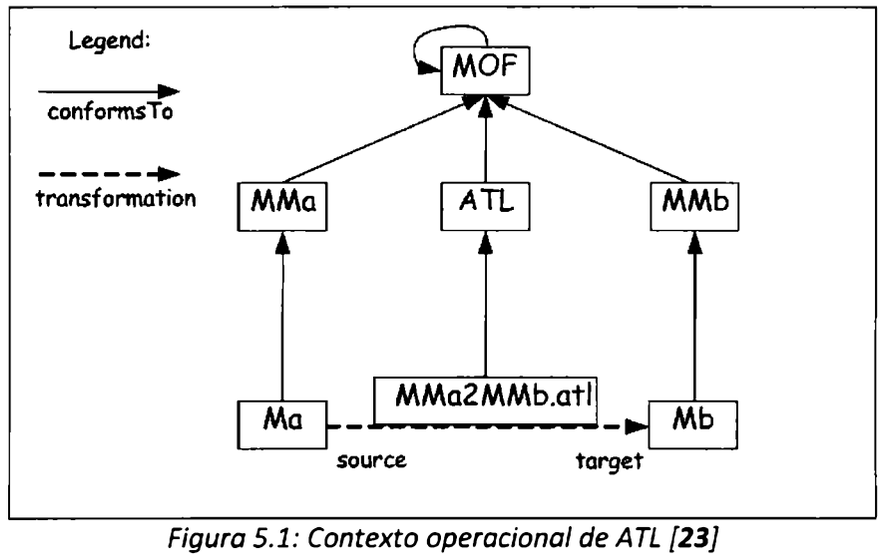


*Figura 5.1: Contexto operacional de ATL [**23**]*

5.2.1 Módulos ATL

Un módulo ATL corresponde a una transformación modelo a modelo. Este tipo de unidad ATL le permite a los desarrolladores especificar cómo producir un conjunto de modelos destino a partir de modelos origen. La cantidad de modelos de entrada y salida es fija, por lo que un módulo ATL no puede generar una cantidad desconocida de modelos destino.

Un módulo ATL está compuesto por los siguientes elementos:

- Una sección de encabezado (header) que define atributos del módulo de transformación.
- Una sección opcional para importar librerías ATL existentes.
- Un conjunto de helpers, que funcionan de forma similar a métodos Java.
- Un conjunto de reglas que definen cómo se generan los modelos destino a partir de los modelos origen.

Estos elementos se detallan a continuación.

5.2.1.1 Header

La sección header define el nombre del módulo de transformación y el nombre de las variables correspondientes a los modelos destino y fuente. La sintaxis es la siguiente:

```
module nombre_modulo;
create modelos_salida [from|refining] modelos_entrada;
```

El nombre de los modelos declarados será el usado en el código para identificarlos.

5.2.1.2 Sección Import

La sección opcional import nos permite declarar qué librerías ATL deben ser importadas. Las librerías ATL se declaran de la siguiente manera:

```
uses extensionless_library_file_name;
```

5.2.1.3 Helpers

Los helpers ATL pueden considerarse como el equivalente ATL a los métodos Java. Nos permiten modularizar el código ATL, dado que cada helper se puede invocar desde

distintos puntos de una transformación ATL. Los helpers se definen de la siguiente manera:

```
helper [context tipo_contexto] def : nombre_helper(parámetros) :
tipo_retorno = ...;
```

Cada helper se caracteriza por su contexto, su nombre, sus parámetros, y su tipo de retorno. El contexto define a qué elementos se aplica el helper, es decir, qué elementos pueden invocarlo. En caso de que no se especifique el contexto, el helper se asocia con el contexto global del módulo ATL; en ese caso la variable self se refiere al módulo en sí. El nombre y los parámetros forman parte de la firma del helper. El cuerpo del helper se define como una expresión OCL.

ATL nos permite también definir atributos. Un helper de atributo es un tipo específico de helper que no acepta parámetros, y se define en el contexto del módulo ATL o de un elemento de modelo. Estos pueden verse como constantes definidas en un contexto en particular.

5.2.1.4 Reglas ATL

Existen tres tipos de reglas en ATL:

Matched rules: Constituyen la esencia de una transformación declarativa ATL, ya que nos permiten especificar cómo se generan e inicializan los elementos destino a partir de los elementos origen.

Cada matched rule se identifica por su nombre, el cual debe ser único dentro de una transformación ATL. Está formada por dos secciones obligatorias (patrones destino y origen) y dos secciones opcionales (variables locales y la sección imperativa).

La sección de variables locales, introducida por la palabra clave *using*, nos permite declarar e inicializar variables locales cuya visibilidad está limitada a la regla actual.

El patrón origen se define luego de la palabra clave *from*, y nos permite especificar una variable de elemento del modelo que corresponde al tipo de elemento origen con los que la regla tiene que coincidir. Este tipo corresponde a una entidad de uno de los metamodelos origen de la transformación. Esto significa que la regla generará elementos destino por cada elemento del modelo origen que conforme a este tipo. Para aplicar la regla a un subconjunto de estos elementos, se pueden usar expresiones OCL dentro del patrón origen de la regla.

El patrón destino de una matched rule le sigue a la palabra clave *to*. Especifica qué elementos van a ser generados cuando el patrón origen de la regla coincida, y cómo son inicializados.

Y por último, la sección opcional imperativa, definida con la palabra clave *do*, nos permite especificar código imperativo que se ejecutará luego de la inicialización de los elementos destino generados por la regla.

Lazy rules: Son reglas declarativas que, a diferencia de las matched rules, sólo se aplican mediante la invocación explícita desde otras reglas.

Called rules: Son reglas imperativas, que tienen que ser llamadas explícitamente para ser ejecutadas y aceptan parámetros. Deben ser invocadas desde una sección de código imperativo. Las called rules pueden generar elementos destino de forma similar a las matched rules.

5.2.2 Semántica de la ejecución de un módulo

La ejecución de un módulo ATL está organizada en tres fases sucesivas.

La primera es la de inicialización del módulo. En esta fase, se inicializan los atributos definidos en el contexto del módulo de la transformación.

En la segunda, se evalúan las condiciones de las matched rules declaradas sobre los elementos de los modelos origen. Cuando la condición de una matched rule es satisfecha, el motor ATL genera los elementos de los modelos destino que correspondan a los declarados en la regla, pero estos no son inicializados todavía.

Finalmente, en la última fase, se inicializan los elementos generados en el paso anterior mediante la ejecución del código asociado al patrón de donde surge el elemento. El código imperativo opcional de una regla se ejecuta una vez que terminó la fase de inicialización de esta. Este código podría disparar la ejecución de called rules.

5.3 Acceleo

Acceleo [**24**] es un proyecto de código abierto que consiste en un generador de código que implementa el estándar MOF Model to Text Language (MTL) [**25**]. Está basado en EMF, por lo que nos permite usar modelos creados con cualquier tipo de herramienta

de EMF como entrada para la generación. El código generado puede ser cualquier tipo de texto.

Luego de crear un generador Acceleo, este se puede usar fácilmente como plugin Eclipse, o también puede ser invocado desde código Java.

Si sucediera que una operación sea imposible de resolver con código Acceleo, se pueden invocar servicios Java. En ese caso los tipos de los parámetros y valor de retorno están limitados a los tipos de los metamodelos usados en la generación o tipos primitivos.

5.3.1 Características del lenguaje

Un generador Acceleo está compuesto por varios archivos llamados módulos. Los módulos están parametrizados por las URIs de los metamodelos en los que se basan los modelos a partir de los que generar código.

La estructura más usada en Acceleo son los templates, utilizados para generar directamente el código. Los templates tienen visibilidad, nombre y parámetros, junto con sus tipos.

Dentro de un template se usan dos tipos de expresiones para generar el código: expresiones estáticas que se generan sin ninguna transformación, y expresiones Acceleo que usan elementos del modelo para computar el texto generado.

```
  [module moduleName('http://www.eclipse.org/emf/2002/Ecore')/]
[template public genMyTemplate(aParam: EClass)]
[comment @main/]
  Este fragmento del código es estático
  [aParam.name/]
  [/template]
```

Figura 5.2: Ejemplo de código Acceleo

En la Figura 5.2, se puede ver por un lado un fragmento de código estático. Este quedará igual en el código final. Y por otro lado una expresión Acceleo, que generará texto usando el nombre de la instancia de EClass pasada como argumento del template durante la ejecución del generador. El comentario *@main* marca al template como template main, es decir, como punto de entrada de la generación. Esto significa que Acceleo generará una clase Java para empezar la generación con este template.

Los templates también pueden usar bloques *file* para generar archivos. Un bloque *file* tiene tres parámetros: Una expresión que retorne un String para el nombre del archivo, un booleano que indique si deberían sobreescribirse los archivos existentes o si el texto generado debería agregarse al final del archivo existente, y la codificación del archivo (por ejemplo, UTF-8).

Las expresiones del lenguaje están basadas en un superconjunto del lenguaje OCL. Para facilitar el desarrollo de los generadores, los templates Acceleo cuentan también con estructuras *if*, *for* y *let*.

5.4 Plugin OCL

Eclipse OCL es una implementación de la especificación OMG OCL 2.4 [**26**] para uso con Ecore y metamodelos UML. La funcionalidad principal de OCL que soporta expresiones sobre modelos se llama Essential OCL. Este lenguaje es muy limitado en sí mismo, ya que no hay forma de proveerle modelos, por lo que está extendido de varias maneras para proporcionarle el contexto faltante. Complete OCL nos permite definir en un documento separado invariantes y otras características que complementan un metamodelo existente. Por otro lado OCLinEcore nos permite embeber OCL en las anotaciones de un modelo Ecore para enriquecerlo. También se incluye el lenguaje OCLstdlib para la definición de bibliotecas estándares OCL.

En este momento el proyecto Eclipse OCL está haciendo una transición a una nueva infraestructura subyacente, por lo que existen 2 proyectos OCL paralelos, el original conocido como Classic OCL, y el nuevo denominado Unified o Pivot OCL.

5.4.1 Classic OCL

El código clásico hacía priorizaba la utilidad para programadores Java. Originalmente soportaba metamodelos Ecore y evolucionó para soportar UML también. El soporte de tanto Ecore como UML complejizaba el código Java tanto para los desarrolladores OCL como los consumidores.

El código clásico se encuentra principalmente en los siguientes plugins:

- org.eclipse.ocl
- org.eclipse.ocl.ecore
- org.eclipse.ocl.uml

5.4.2 Complete OCL

El lenguaje Complete OCL nos permite complementar un metamodelo existente en un documento separado, con invariantes y otras características. Su sintaxis está definida por la especificación OMG OCL 2.4, la cual se encuentra en [**26**].

Un ejemplo detallado de Complete OCL se puede encontrar instalando en Eclipse el RoyalAndLoyal Example Project. Este es el ejemplo estándar utilizado en muchos textos y cursos OCL; fue producido inicialmente como parte de [**5**]. Debido a que incluye una amplia variedad de expresiones OCL, se eligió utilizarlo en el presente trabajo como caso de estudio sobre el que se aplicará la traducción a JML.

5.4.3 Metamodelo Unificado o Pivot

El metamodelo Pivot o Unificado es un prototipo para la resolución de algunos de los problemas fundamentales de la especificación OCL 2.4. El metamodelo Pivot es derivado de los metamodelos de UML y OCL para lograr un metamodelo unificado para UML con semántica ejecutable. En la práctica, cuando se utiliza el metamodelo Pivot para metamodelos Ecore o UML, se crea sobre la marcha una instancia del metamodelo Pivot, de manera de proporcionar una funcionalidad unificada con OCL para las instancias del metamodelo Ecore y UML.

Desde la perspectiva de la especificación, el metamodelo Pivot:

- Está alineado con UML.
- Soporta el modelado de la librería estándar OCL.
- Soporta el agregado de definiciones Complete OCL adicionales.
- Soporta representación XMI intercambiable
- Soporta un oclType completamente reflexivo.

Mientras que desde la perspectiva de Eclipse, el metamodelo Pivot:

- Oculta las diferencias de Ecore con respecto a EMOF
- Oculta las diferencias de MDT/UML2 con respecto a UML
- Permite que mucha de la semántica sea definida por un solo modelo de biblioteca
- Permite la extensión de usuario y reemplazo del modelo de biblioteca
- Permite el cumplimiento exacto de los estándares de la OMG

El código unificado es proporcionado por el plugin org.eclipse.ocl.pivot, con soporte adicional para UML en el plugin org.eclipse.ocl.pivot.uml.

5.4.3.1 Clases básicas del metamodelo

En el metamodelo Pivot, las restricciones OCL son representadas por la clase Constraint. En la Figura 5.3 se muestran las clases principales relacionadas a esta.

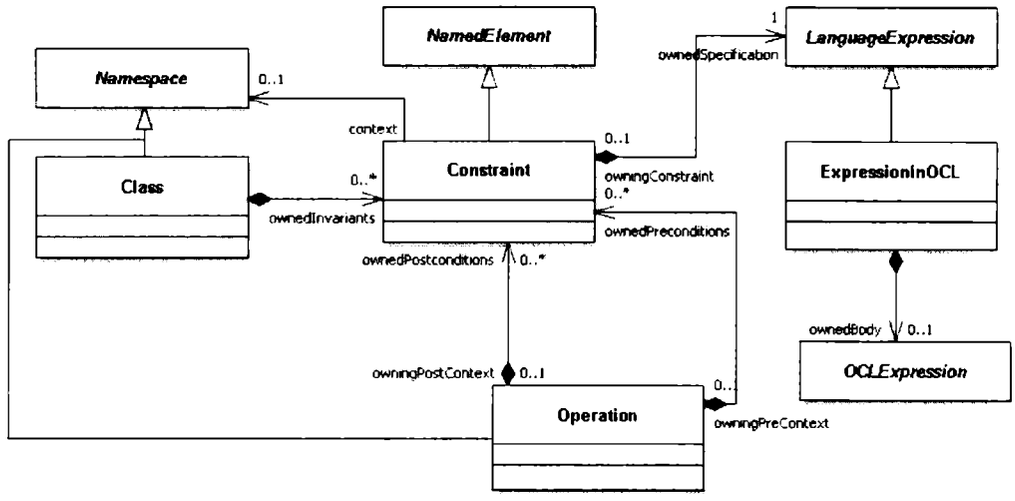


Figura 5.3: Diagrama reducido de las clases relacionadas con las restricciones OCL

Una *constraint* es un elemento nombrado, y tiene como contexto un *Namespace*. Este puede ser, por ejemplo, una clase o una operación. A su vez, una clase puede tener constraints como invariantes, y una operación puede tener constraints como pre y postcondiciones. La constraint puede tener como referencia una *ExpressionInOCL*. Esta clase tiene como cuerpo una *OCLExpression*, y, aunque no se muestra en la imagen, puede tener referencias a variables parámetro, a la variable self, y a la variable resultado.

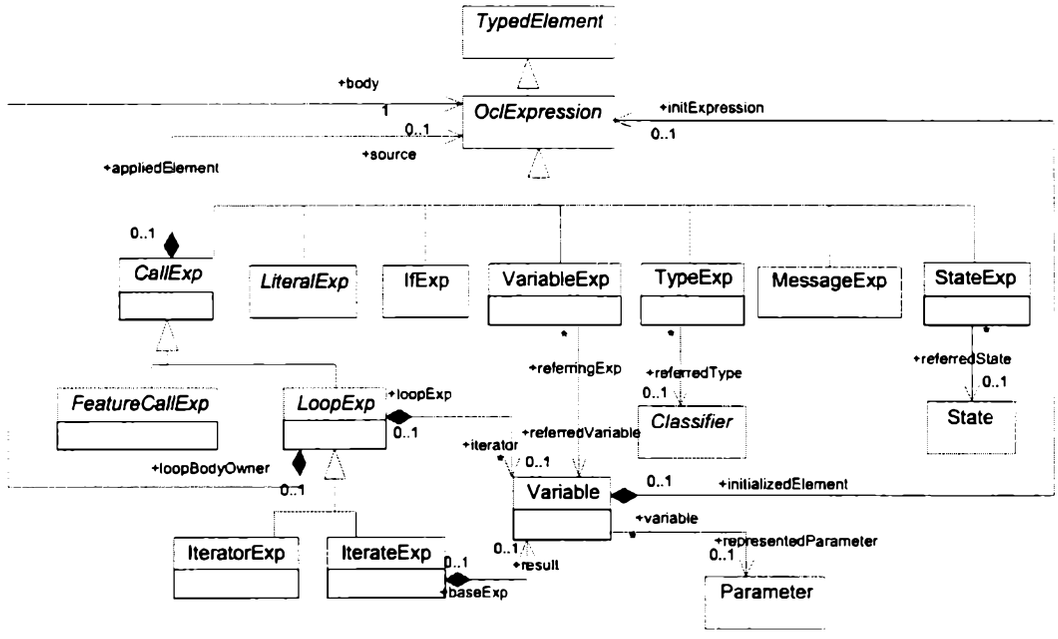


*Figura 5.4: Diagrama de clases básico de las expresiones OCL [**26**]*

La clase *OCLExpression* está definida bajo la especificación de OCL 2.4, y su estructura básica se muestra en la Figura 5.4. Cada expresión OCL tiene un tipo. Como se ve en la imagen, una expresión OCL puede ser de una variedad de formas distintas, como por ejemplo, una expresión *if,* un literal, una expresión de variable, etc. La clase *FeatureCallExp*, que representa los llamados a operaciones o referencias a propiedades, se muestra en mayor detalle en la Figura 5.5.

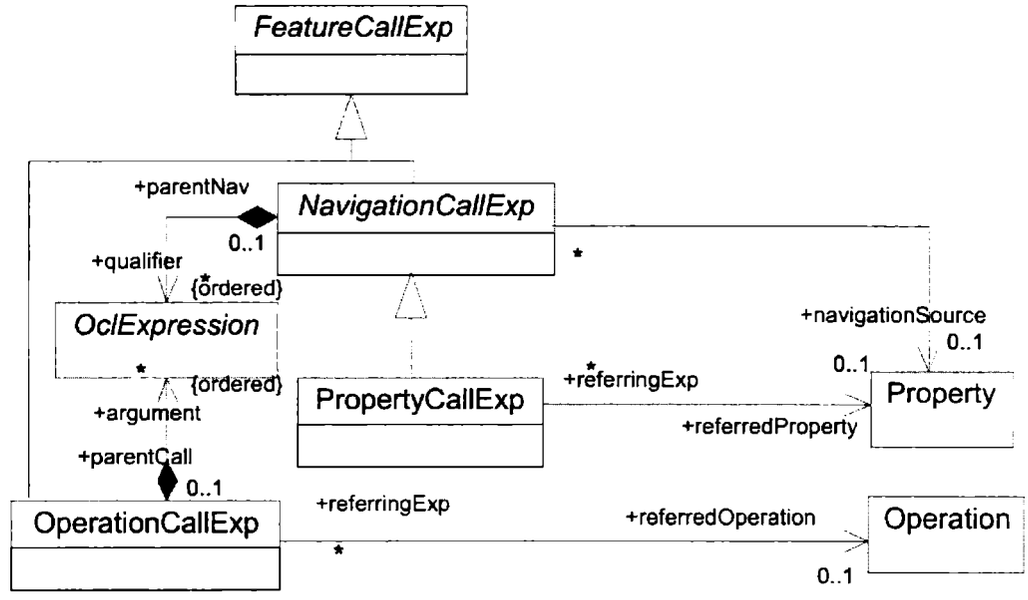


*Figura 5.5: Estructura de una FeatureCallExp [**26**]*

5.5 MoDisco

MoDisco [**27**] es un framework extensible para el desarrollo de herramientas dirigidas por modelos para la modernización de software existente. Sus objetivos son proveer:

- Aseguramiento de calidad: Verificar si un sistema existente cumple con los requerimientos esperados, mediante la detección de anti-patrones en el código existente y el cálculo de métricas.
- Documentación: Extracción de información de un sistema existente para ayudar a entender aspectos del sistema, tales como su estructura, comportamiento, persistencia, flujo de datos, etc.
- Mejora: Transformar un sistema existente para que integre mejores normas de codificación o patrones de diseño
- Migración: Transformar un sistema existente para cambiar un componente, su framework, el lenguaje, o su arquitectura.

Para lograr esto MoDisco incluye metamodelos que describen sistemas existentes, descubridores que instancian automáticamente estos metamodelos y herramientas genéricas para entender y transformar modelos complejos creados a partir de sistemas existentes.

En este trabajo se utilizaron algunas de las herramientas de MoDisco para el desarrollo de un metamodelo JML.

5.6 OpenJML

OpenJML [**28**] es un conjunto de herramientas para JML, construido sobre el framework OpenJDK para Java. Su objetivo principal es implementar una herramienta completa para JML que sea simple de usar para especificar y verificar programas Java.

OpenJML cuenta con las siguientes herramientas:

- Una herramienta de línea de comandos que nos permite realizar: análisis sintáctico y chequeo de programas Java+JML, chequeo estático de anotaciones JML utilizando solvers SMT externos, y chequeo en tiempo de ejecución, utilizando una extensión del compilador Java de OpenJDK.
- Un plugin de Eclipse que encapsula las funcionalidades de la herramienta de línea de comandos en una interfaz gráfica Eclipse con características gráficas apropiadas para las anotaciones JML, y permite explorar contraejemplos interactivamente.

OpenJML tiene como propósito reemplazar las herramientas JML2 desarrolladas en ISU (que sólo funcionaban hasta Java 1.4). OpenJML soporta el análisis de Java 1.8 y versiones previas. Es el único proyecto activo de desarrollo de herramientas para JML en este momento.

Para realizar la verificación estática de JML, OpenJML hace uso de probadores SMT. En general, OpenJML puede usar cualquier solver que cumpla con el estándar SMT-LIB versión 2.0 [**29**]. OpenJML fue probado con Z3 [**30**], CVC4 [**31**], y Yices [**32**].

Como su nombre lo dice, OpenJML es un proyecto de código abierto. Su implementación se encuentra en el repositorio GitHub del proyecto OpenJML [**33**].

5.7 Resumen

En este capítulo se describieron las distintas herramientas utilizadas en este trabajo. Las herramientas descritas son entonces las siguientes:

- EMF como base de los modelos.
- ATL para realizar transformaciones modelo a modelo.

- Acceleo para realizar transformaciones modelo a texto.
- El plugin OCL como implementación para Eclipse de OCL, y el metamodelo Pivot como metamodelo unificado de UML y OCL.
- MoDisco por sus herramientas para el modelado de código Java
- OpenJML para la ejecución y verificación del código JML.

6 Traducción de OCL a JML

En este capítulo se realiza una comparación de OCL y JML, y se explica detalladamente cómo es la función de traducción de un lenguaje a otro.

6.1 Motivación

Como se vio en el capítulo 3, OCL es una notación formal para especificar restricciones en modelos UML. Dado que estas especificaciones no pueden ser ejecutadas directamente y chequeadas en tiempo de ejecución por una implementación, las violaciones a estas podrían nunca ser detectadas, y esto podría causar eventuales problemas en el desarrollo y mantenimiento.

Podemos evaluar las restricciones OCL en tiempo de ejecución traduciéndolas a código JML. OCL no es específico de ningún lenguaje de programación, mientras que JML es específico de Java. El mayor beneficio de usar JML para la especificación de aplicaciones Java es la amplia variedad de herramientas que soportan JML. Utilizándolas se puede realizar efectivamente verificación estática y comprobación en tiempo de ejecución de las restricciones, facilitando así encontrar violaciones a estas.

6.2 Comparación de los lenguajes

Tanto OCL como JML están basados en Design-by-Contract y permiten describir propiedades de clases y métodos. Ambos lenguajes cuentan con el concepto de invariante y pre y postcondiciones, por lo que la traducción es directa; cada restricción OCL se traduce a una expresión JML. Aun así, no todas las expresiones OCL pueden expresarse en JML. Asimismo OCL no cuenta con todas las funcionalidades de JML, por lo que hay elementos de este lenguaje que no serán aprovechados en la traducción.

A continuación se presentan algunas diferencias semánticas entre los lenguajes.

6.2.1 Diferencias semánticas

Refiriéndose al pre-estado

Tanto OCL como JML permiten que en las postcondiciones se pueda referir a estados previos. En OCL se utiliza la palabra clave *@pre*, y en JML se utiliza la cláusula *\old*. La diferencia entre estas es que la primera construcción puede aplicarse a símbolos individuales, mientras que la segunda sólo se puede aplicar a expresiones completas.

Por ejemplo, dada una expresión a.b.c, a@pre.b@pre.c@pre, a.b@pre.c@pre, a.b.c@pre y a@pre.b.c@pre son todas expresiones legales en OCL, mientras que sólo \old(a.b.c) se permite en JML, la cual es equivalente a la primera de las expresiones OCL.

Rango de los cuantificadores

En JML, la cuantificación se puede aplicar a todos los elementos de un tipo y no solamente a los elementos creados o asignados en memoria. En OCL realizar una cuantificación sobre, por ejemplo, todos los enteros, no es posible; su semántica sólo permite conjuntos finitos.

Excepciones

A diferencia de JML, OCL, por su naturaleza genérica, no posee una construcción para especificar excepciones que puedan ocurrir en una operación.

Igualdad e identidad

En Java existen dos maneras de comparar objetos:

- Por identidad: Mediante el operador == se evalúa si dos referencias a objetos son iguales.
- Por igualdad: Mediante el operador *equals(o : Object)* se debería evaluar si dos objetos son iguales en cuanto a sus datos. Por defecto la implementación de *equals* en la clase Object evalúa identidad, por lo que queda a responsabilidad del programador implementar este método en las clases que desarrolle para lograr el comportamiento deseado.

En OCL, en cambio, sólo se pueden comparar objetos por igualdad.

6.3 Función de traducción

A pesar de las diferencias descritas arriba, OCL y JML son lenguajes muy similares, por lo que gran parte de la traducción es directa. En este trabajo se optó por seguir para la traducción el enfoque definido en [**34**], en particular su estrategia para traducir las colecciones.

6.3.1 Invariantes, precondiciones y postcondiciones

Los invariantes, precondiciones y postcondiciones OCL se traducen directamente a invariantes, precondiciones y postcondiciones JML, asociadas a sus respectivas clases y métodos.

6.3.2 Tipos simples

Los tipos simples se traducen directamente, como se muestra en la Tabla 6.1: Traducción de tipos de datos simples

OCL	JML
Boolean	boolean
Integer	int
Real	double
String	String
UnlimitedNatural	\bigint

Tabla 6.1: Traducción de tipos de datos simples

El tipo \bigint es específico de JML y se utiliza para representar enteros de precisión infinita.

6.3.3 Operadores y expresiones

Los operadores matemáticos y lógicos de OCL se traducen casi directamente. En la Tabla 6.2 se muestra la traducción de las operaciones lógicas y de igualdad.

OCL	JML
not e	!e
e1 and e2	e1 && e2
e1 or e2	e1 || e2
e1 xor e2	e1 ^ e2
e1 implies e2	e1 ==> e2
e1 = e2	*e1 == e2* (tipos primitivos) e1.equals(e2) (objetos)
e1 <> e2	*e1 != e2* (tipos primitivos) !e1.equals(e2) (objetos)
if e0 then e1 else e2 endif	(e0? e1 : e2)

Tabla 6.2 : Traducción de operaciones básicas

Las operaciones de números y Strings se tradujeron a sus operaciones Java equivalentes de forma directa.

6.3.4 Pseudovariables y operaciones predefinidas

Las pseudovariables y las operaciones predefinidas se traducen casi directamente, como se muestra en la Tabla 6.3.

OCL	JML
self	this
result	\result
variable@pre	\old(variable)
exp.oclIsNew()	\fresh(exp)
exp.oclIsUndefined()	exp == null
exp.oclIsTypeOf(t)	\typeof(exp) == \type(t)
exp.oclIsKindOf(t)	\typeof(exp) <: \type(t)
exp.oclAsType(t)	(t) exp

Tabla 6.3: Traducción de pseudovariables y operaciones predefinidas

6.3.5 Colecciones

Aunque OCL sólo cuenta con cuatro colecciones, en el código final de Java se puede usar una variedad de colecciones con distinto vocabulario para sus operaciones. Si se hacen especificaciones JML directamente sobre las colecciones del programa Java, ante un cambio de representación (por ejemplo, de set a array) las especificaciones podrían quedar desactualizadas.

En [**35**] se propone usar los tipos modelo definidos por JML para representar varios tipos de colecciones implementadas como clases Java. El problema es que la organización, estructura y vocabularios de estas colecciones es distinta a la de las de OCL, y la expresión JML resultante de la traducción puede ser muy distinta sintácticamente de la expresión OCL original. Por esta razón en [**34**] se propone el desarrollo de una biblioteca que implemente las colecciones OCL en Java; en este trabajo se optó por seguir ese enfoque.

Las colecciones OCL y sus métodos entonces se traducen directamente a sus respectivas clases Java y métodos del mismo nombre. Al usar en la traducción el

mismo vocabulario que en las restricciones OCL se mantiene una clara relación con la especificación original. Para que la especificación sea independiente de la elección de colección en Java, se utilizan campos modelo que las representan. En la Tabla 6.4 se muestra el mapeo de las clases OCL a las clases Java desarrolladas.

OCL	Java + JML
Set	OCLSet
Bag	OCLBag
OrderedSet	OCLOrderedSet
Sequence	OCLSequence

Tabla 6.4: Traducción de las colecciones

Para explicar el método se verá un ejemplo. Supongamos que tenemos un modelo con una clase Materia, que cuenta con un conjunto de alumnos. Dado el siguiente código OCL:

```
context Materia
inv TieneAlumnos :
      alumnos->size() > 0
```

Se genera como resultado el código de la Figura 6.1. En este caso se eligió como representación de la colección concreta un objeto de tipo LinkedHashSet de alumnos. Las especificaciones JML no se aplican a esa colección, sino a una abstracción de esta, representada por una de las clases de la biblioteca desarrollada, OCLOrderedSet. Las clases de la biblioteca implementan métodos estáticos para crear objetos de su tipo a partir de cualquier colección. Este método se utiliza entonces en la cláusula *represents* para definir la relación entre la colección del a especificación y la colección concreta. Como podemos ver, el invariante no se aplica a *alumnos* sino a *alumnos_spec*. De esta forma, si cambiara la representación de alumnos, las especificaciones JML seguirían siendo las mismas.

```
public class Materia {

      protected /*@ spec_public*/ LinkedHashSet<Alumno> alumnos;

      /*@
            protected spec_public model OCLOrderedSet<Alumno> alumnos_specs;
            represents alumnos_specs = OCLOrderedSet.convertFrom(alumnos);
            public invariant (this.alumnos_specs.size() > 0);
      */

}
```

Figura 6.1: Ejemplo de especificaciones JML con colecciones

6.3.5.1 Operadores de colección

Como se vio en el capítulo 3, OCL cuenta con operaciones para las colecciones que utilizan iteradores. Operaciones como *select* y *collect* se implementan en la biblioteca mediante el uso de expresiones Lambda [**36**]. Las expresiones Lambda tienen como tipo una interfaz funcional, es decir, una interfaz con un solo método. La biblioteca desarrollada incluye interfaces funcionales para permitir el uso de expresiones Lambda como parámetros de los métodos de las colecciones.

Por ejemplo, en el contexto de las materias y alumnos, si en OCL se tiene la expresión siguiente:

```
alumnos->collect(a | a.nombre)
```

En JML, la expresión resultante sería:

```
this.alumnos_specs.collect((a) -> a.nombre)
```

OpenJML, la herramienta utilizada en este trabajo para evaluar las expresiones JML, todavía no soporta por completo las expresiones lambda, pero como puede verse el repositorio GitHub del proyecto [**33**], se está trabajando actualmente en esto.

6.3.6 Atributos y operaciones definidos

En la sección 3.2.4 se describen los atributos y operaciones definidos en OCL. Como estos sólo se usan como ayuda para la especificación se eligió traducir los atributos definidos a campos fantasma, y las operaciones a métodos modelo. De esta manera se pueden usar en la especificación JML sin modificar el código Java.

6.3.7 Expresión de cuerpo de operación

En cuanto a la traducción las expresiones *body*, para que la especificación JML sea independiente de la implementación, se optó por traducirlas como postcondiciones sobre el resultado del método. Por ejemplo, dada una clase Materia con una operación cantidadAlumnas, la siguiente restricción OCL:

```
context Materia::cantidadAlumnas () : Integer
body : alumnos->select(a | a.sexo = Sexo::Femenino )->size()
```

Se traduce de la siguiente manera :

```
/*@
ensures \result ==
this.alumnos_specs.select((a) -> a.sexo.equals(Sexo.Femenino)).size();

*/
public int cantidadAlumnas ();
```

De esta forma se especifica el resultado del método sin modificar su implementación.

6.3.8 Valores iniciales y atributos derivados

Los valores iniciales de OCL se traducen a una cláusula *initially* que iguala el campo a su valor especificado. Por ejemplo, siguiendo con el modelo de los alumnos, para especificar que un alumno inicialmente no está suspendido, se escribe de la siguiente manera:

```
context Alumno::suspendido : Boolean
init :false
```

La cual se traduce a la siguiente expresión JML:

```
//@ public initially suspendido == false;
```

A los atributos derivados se eligió traducirlos como invariantes. Si un atributo es derivado, entonces su valor siempre va a cumplir con una restricción. Por lo tanto, en términos de especificación, es equivalente implementarlos como atributos con un invariante que restrinja sus valores.

6.3.9 Expresiones let

Para traducir las expresiones *let* se optó por definir por cada variable de la expresión un campo fantasma, cada una con el nombre y valor inicial de la variable equivalente. Luego cada referencia a esa variable es traducida a una referencia al campo. Por ejemplo, dada la siguiente expresión OCL que utiliza una expresión let:

```
context Alumno
inv : let cantMateriasAprobadas : Integer = materiasAprobadas->size()
in
      if esIngresante() then cantMateriasAprobadas =
      else cantMateriasAprobadas >    endif
```

Esta se traduce de la siguiente manera:

```
protected ghost Integer cantMateriasAprobadas =
this.materiasAprobadas_specs.size();
public invariant ((this.esIngresante()? (cantMateriasAprobadas == 0) :
(cantMateriasAprobadas > 0)));
```

6.4 Resumen

En este capítulo se describió el porqué de la traducción de OCL a JML. Se analizó comparativamente a OCL y a JML, explicando sus diferencias sintácticas y semánticas, y se describió en detalle la función de traducción realizada.

7 Herramienta desarrollada

En este capítulo se describe detalladamente la implementación de la herramienta desarrollada y los módulos que la componen.

Los casos de estudio a utilizar serán el proyecto de Eclipse Royal and Loyal previamente mencionado, y un modelo de una biblioteca.

7.1 Diseño

La herramienta diseñada consiste en un plugin Eclipse compuesto por dos módulos: Uno que realiza la transformación modelo a modelo, y otro que realiza la transformación modelo a texto. El funcionamiento general entonces es el siguiente:

El metamodelo OCL utilizado como entrada para la primera transformación es el metamodelo Pivot. La instancia del metamodelo Pivot se obtiene a partir de un archivo Complete OCL y su modelo Ecore correspondiente, utilizando la función el plugin OCL para generar la sintaxis abstracta de estos.

El metamodelo JML, utilizado como salida, fue desarrollado para este trabajo. La herramienta genera a partir de la instancia del metamodelo Pivot, una instancia de este metamodelo JML.

Luego de la transformación M2M, se transforma la instancia del metamodelo JML en texto, consiguiendo así archivos .java con el código de las clases Java, y archivos .jml con las especificaciones obtenidas a partir del código OCL. Estos archivos están listos para ser analizados y ejecutados por la herramienta OpenJML.

7.2 Metamodelo JML

Para lograr efectivamente la transformación modelo a modelo, se desarrolló un metamodelo para el lenguaje JML. Está basado en el metamodelo Java de MoDisco, que se encuentra implementado en org.eclipse.gmt.modisco.java. En [**37**] se describen algunas de las meta-clases de este metamodelo. MoDisco cuenta también con un descubridor que permite dado un código Java generar una instancia del metamodelo.

El metamodelo JML desarrollado mantiene las meta-clases principales del metamodelo Java, y le agrega meta-clases específicas de JML. La elección de las clases y sus relaciones no fue arbitraria, sino que se basa en las clases del analizador sintáctico

(parser) de OpenJML, disponible en [**33**], y en la gramática del lenguaje, definida en BNF en [**38**]. Además, se le agregaron meta-clases para el uso de expresiones Lambda, que serán utilizadas en la traducción.

Como este metamodelo está pensado para utilizarse con especificaciones definidas en archivos .jml separados, no soporta anotaciones mezcladas con el código, por lo que elementos como invariantes de ciclos no están incluidos en él.

7.2.1 Elementos específicos de JML

Los elementos agregados al metamodelo para soportar la sintaxis de JML se describen a continuación.

Cada declaración de clase tiene asociado uno o más elementos JMLTypeSpecs. Estos representan la especificación de un tipo, y a su vez cada uno está compuesto por elementos JMLTypeClause. En la Figura 7.1 se muestra un diagrama de clases simplificado de las clásulas de tipo definidas.

Un ejemplo de cláusula de tipo son los invariantes; en este metamodelo se representan mediante elementos de tipo JMLTypeClauseExpr con el atributo *kind* definido como *invariant*. El atributo *expression* representa la expresión que debe valer como invariante. Los otros dos tipos de cláusulas de expresión definidas son las cláusulas *initially*, y los axiomas.

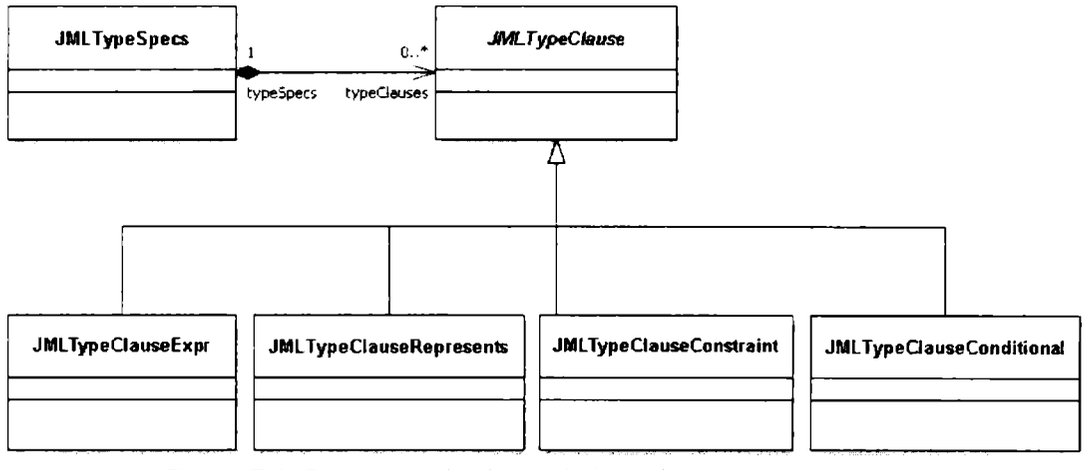


Figura 7.1: Diagrama de clases de las cláusulas de tipo de JML

Por otro lado, cada método tiene asociado un elemento de tipo JMLMethodSpecs. Un JMLMethodSpecs está compuesto por múltiples elementos de tipo JMLSpecificationCase. Un JMLSpecificationCase tiene un conjunto de elementos de tipo JMLMethodClause, un comportamiento (normal o excepcional), un conjunto de modificadores, y un booleano que define si debe llevar la palabra clave *also o* no. En la Figura 7.2 se puede ver un diagrama de clases simplificado de las cláusulas de método definidas.

Un ejemplo de cláusula de método son las precondiciones y postcondiciones; en este metamodelo se representan mediante elementos de tipo JMLMethodClauseExpr, con el atributo *methodclausekind* con valor *requires* o *ensures*. El atributo *expression* representa la expresión que debe valer como pre o postcondición.

Además, se le agregaron al metamodelo expresiones específicas de JML, como invocaciones a métodos de JML (por ejemplo, *\old(E)*), operadores, comprensión de conjuntos, y expresiones cuantificadas. En la Figura 7.3 se muestra un diagrama de clases simplificado de las

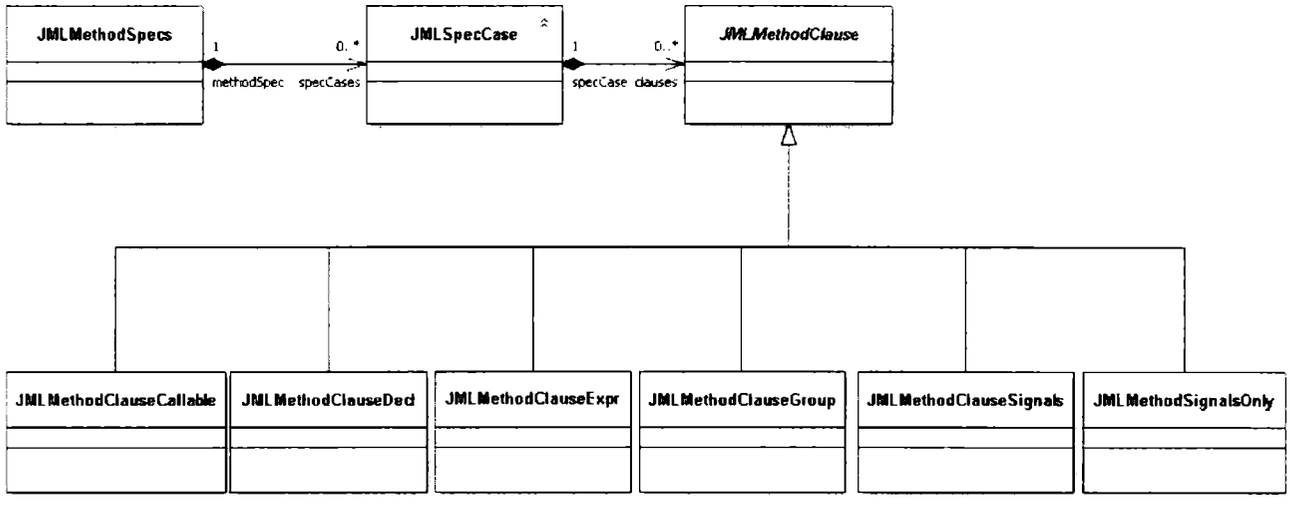


Figura 7.2: Diagrama de clases de las cláusulas de método de JML

expresiones JML. Estas son subclases de la clase abstracta Expression, que representa todas las expresiones del lenguaje. Por ejemplo, la expresión *\result* es representada por un objeto de JMLSingleton con el atributo *kind* con valor *result*.

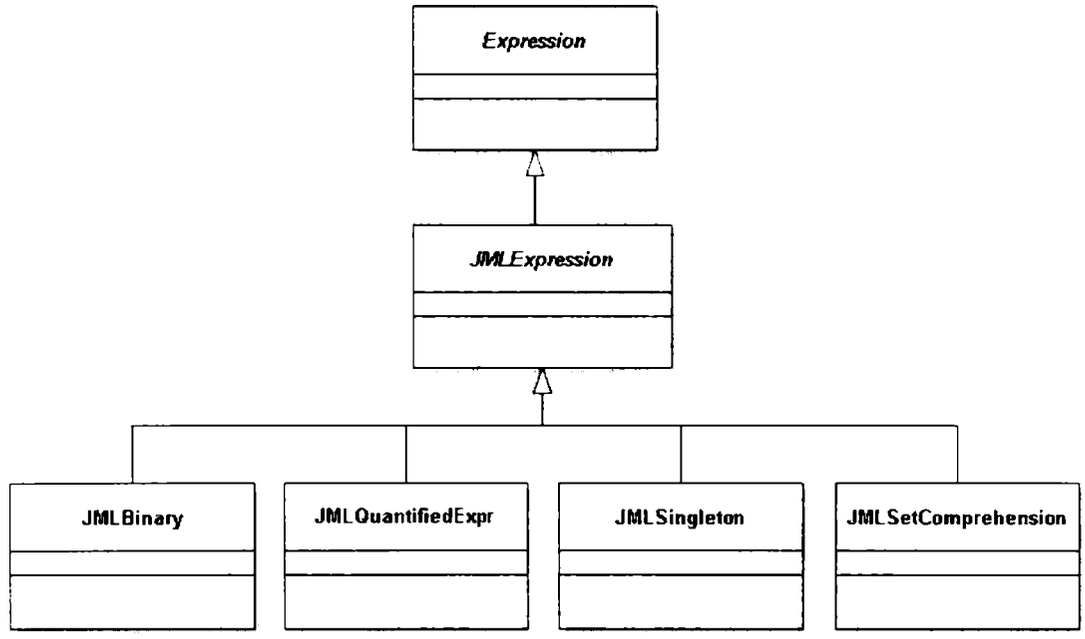


Figura 7.3: Diagrama de clases de las expresiones JML

7.3 Biblioteca de colecciones OCL

Como fue mencionado en el capítulo 6, para la traducción de colecciones se optó por utilizar una biblioteca Java que implemente las clases de las colecciones de OCL. Las clases están basadas en la propuesta de [**34**]. En [**39**] se implementó una biblioteca Java siguiendo ese enfoque. Para este trabajo se tomó esa biblioteca y se la refactorizó y se le agregó funcionalidad. Algunos de los cambios que se hicieron son los siguientes:

- Se hizo que las clases de colecciones implementen la interfaz Iterable, para facilitar el manejo de estas.
- Se agregaron métodos nuevos equivalentes a los de las colecciones OCL que no estaban implementados.
- Se implementaron métodos estáticos para la creación de objetos del tipo de las colecciones dada una colección o arreglo como parámetro.

Para implementar los métodos que reciben expresiones como parámetros se utilizan interfaces funcionales. Al definir el parámetro de un método como interfaz funcional, este se puede invocar usando expresiones lambda como argumento. Las expresiones lambda son semánticamente equivalentes a la creación de clases anónimas que

implementen interfaces funcionales. Se eligió utilizarlas por su practicidad y mayor legibilidad.

No todos los métodos pudieron ser implementados debido a la falta construcciones Java para permitir la evaluación de bloques, como es el caso de *sortBy* e *iterate*.

En la Figura 7.4 se muestra un diagrama de clases de las colecciones con los métodos públicos de estas.

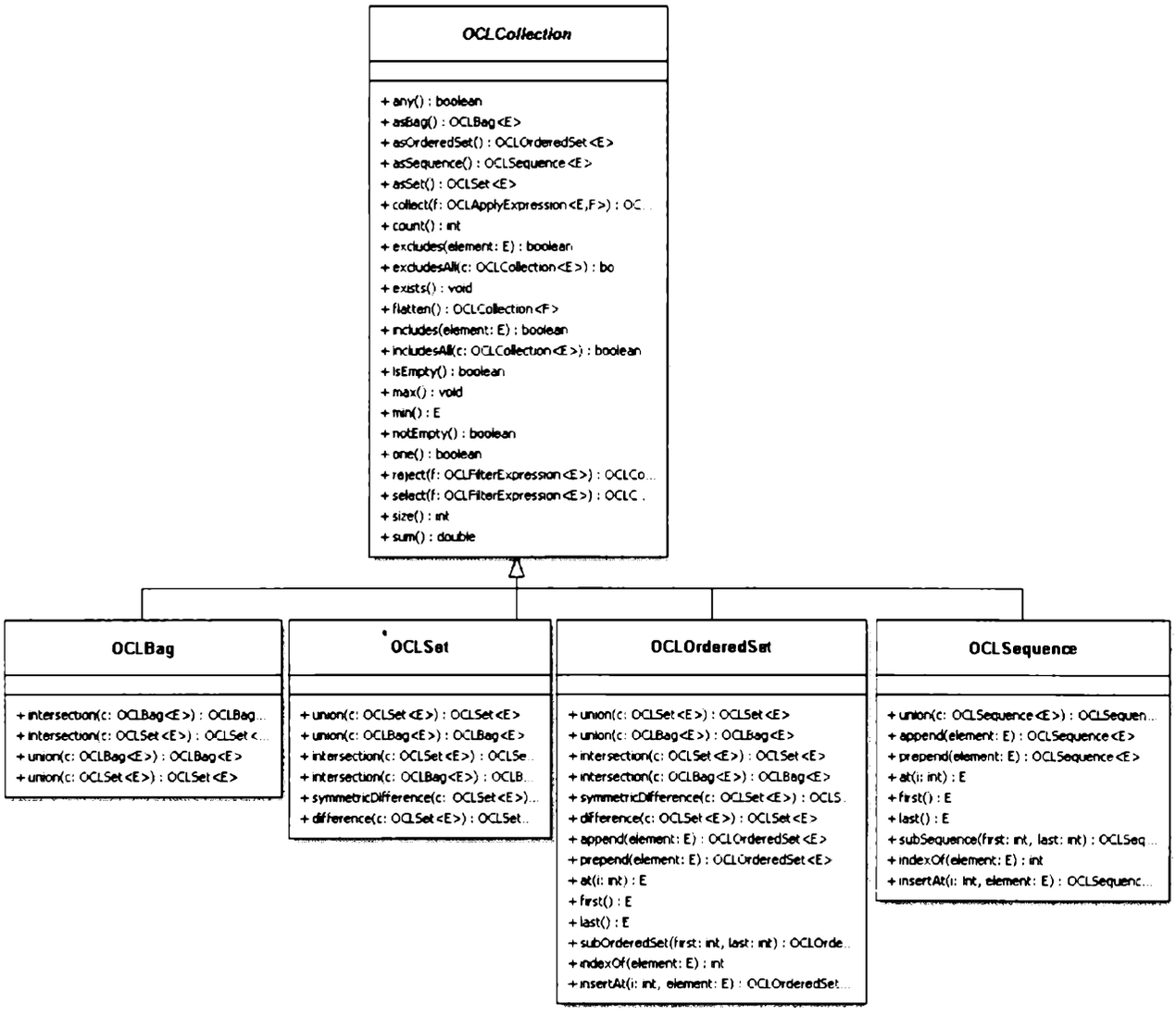


Figura 7.4 Diagrama de clases de las colecciones de la biblioteca desarrollad

Para la implementación de las colecciones se utilizaron clases Java subyacentes. Las clases utilizadas se muestran en la Tabla 7.1. MultiSet, la clase usada para la implementación de OCLBag, es una clase de la biblioteca de colecciones de Guava [**40**] que implementa la interfaz de colecciones de Java.

Esta elección se basa en si las colecciones tienen orden o no, y si tienen elementos únicos o no.

Biblioteca OCL	**Clases Java**
OCLSet	HashSet
OCLBag	MultiSet
OCLOrderedSet	LinkedHashSet
OCLSequence	List

Tabla 7.1: Colecciones Java subyacentes

7.4 Casos de estudio

7.4.1 Modelo Royal and Royal

El proyecto de ejemplo de Eclipse Royal and Loyal es un ejemplo completo de un documento Complete OCL que complementa un metamodelo Ecore independiente. Cuenta con una amplia variedad de restricciones OCL.

El proyecto se puede instalar en Eclipse Seleccionando la opción **File** -> **New** -> **Examples...**, y luego eligiendo la opción **Royal and Loyal Example**, en la sección **OCL (Object Constraint Language) plugins**.

En la Figura 7.5 se muestra el diagrama de clases del modelo. Este consiste en un sistema de fidelidad de clientes a una empresa, y los servicios que se le ofrecen a estos.

Además del modelo Ecore, el proyecto incluye un extenso archivo con restricciones OCL sobre el modelo. En la Figura 7.6 se muestran algunas de las restricciones del documento.

En este trabajo se aplicará la traducción a todas las restricciones del modelo, con la excepción de las que utilizan operadores como *iterate*, que no son soportados por la herramienta.

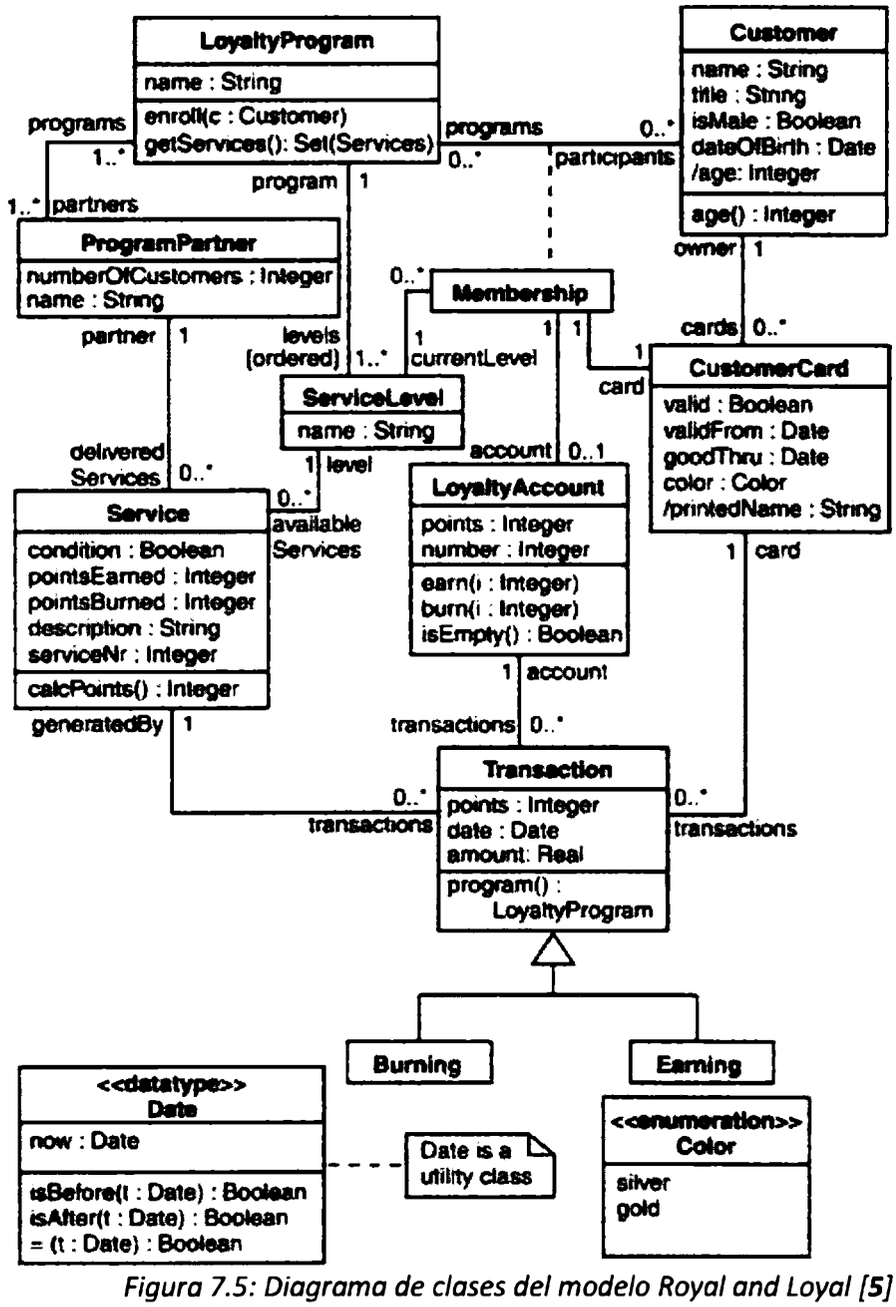


Figura 7.5: Diagrama de clases del modelo Royal and Loyal [5]

```
context Membership
def  :
    getCurrentLevelName() : String = self.currentLevel.name
inv invariant_Membership1 :
    (self.account?.points >= 0) or self.account->asSet()->isEmpty()
inv invariant_Membership2 :
    self.participants.cards->collect( i_CustomerCard : CustomerCard |
        i_CustomerCard.Membership)->includes(self)
inv invariant_noEarnings :
    programs.partners.deliveredServices->forAll(pointsEarned = 0)
    implies account->isEmpty()
inv invariant_correctCard :
    self.participants.cards->includes(self.card)
inv invariant_Membership3 :
    self.programs.levels->includes(self.currentLevel)
```

Figura 7.6: Ejemplo de restricciones OCL del modelo Royal and Loyal

7.4.2 Biblioteca

El proyecto Royal and Loyal es un buen ejemplo de restricciones OCL variadas y completas, pero no necesariamente estas restricciones tienen sentido en el modelo. Para mostrar la utilidad de las herramientas de verificación JML, se tomó como caso de estudio el metamodelo de una biblioteca desarrollado en [**41**]. El modelo se puede ver en la Figura 7.7.

En este trabajo, a este modelo se le agregaron operaciones y restricciones coherentes para poder ejecutar las herramientas de verificación JML sobre un sistema concreto con funcionamiento conocido, como lo es una biblioteca.

En la Figura 7.8 se pueden ver algunas de las restricciones OCL agregadas al modelo.

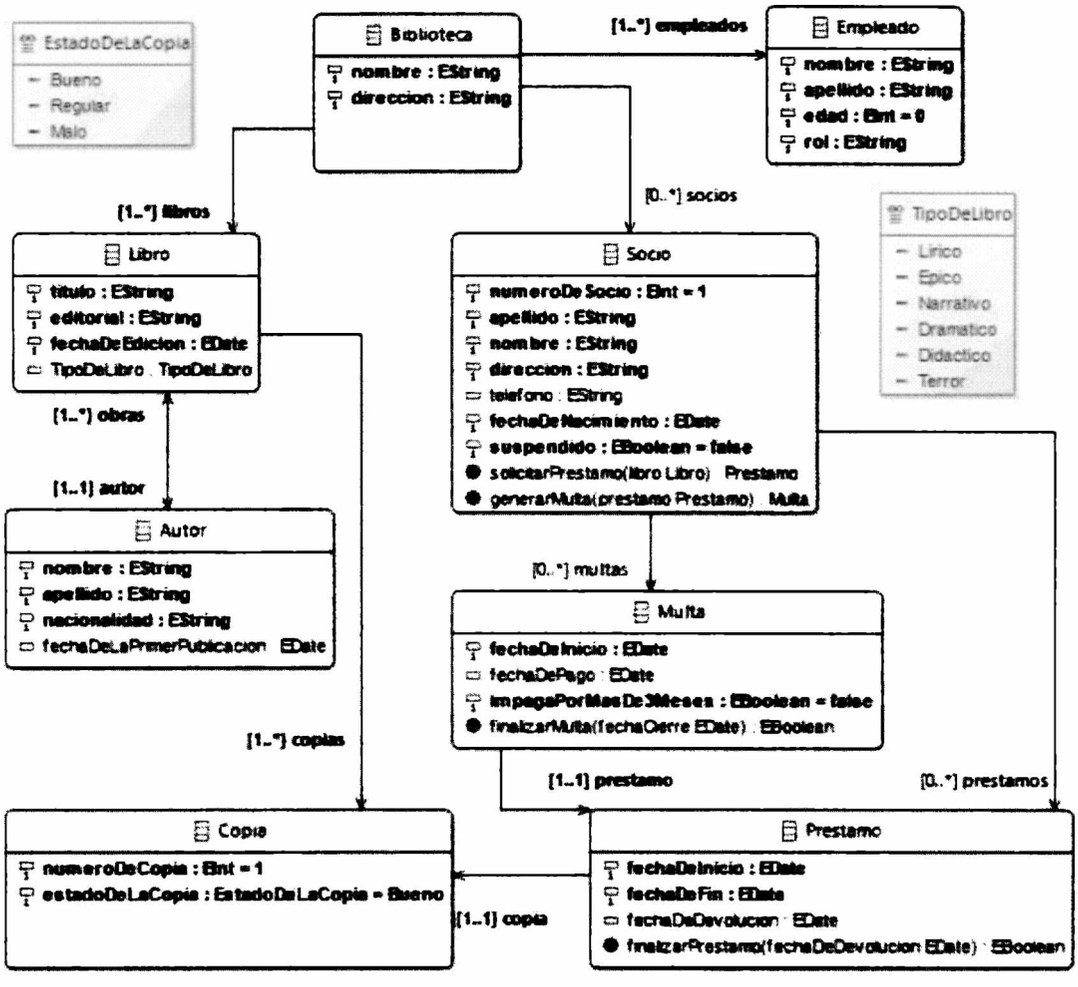


*Figura 7.7: Metamodelo de biblioteca [**41**]*

```
17      context Libro::agregarCopia( copia : Copia) : OclVoid
18          post : copias->includes(copia)
19
20      context Libro::findCopiaDisponible() : Copia
21          post : hayCopiaDisponible() implies not result.oclIsUndefined()
22
23      context Libro::hayCopiaDisponible() : Boolean
24          post : copias->size() = 0 implies result = false
25
26      context Socio
27
28      inv nombre: nombre <> '' and apellido <> ''
29      inv nMultas: self.multas->size() <= self.prestamos->size()
30
31      context Socio::suspendido : Boolean
32      init: false
```

Figura 7.8: Ejemplo de restricciones OCL para el modelo de la biblioteca

7.5 Traducción de modelo OCL a modelo JML

Para la traducción de un metamodelo a otro se utilizó el lenguaje ATL, introducido en la sección 5.2. Se utilizó un solo módulo, que será descrito a continuación, junto con las reglas esenciales para la traducción.

En la Figura 7.9 se muestra el encabezado del módulo. En los comentarios se definen las rutas de los metamodelos utilizados y cómo referirse a ellos en el código. En la sección *create* se indican los modelos de entrada y salida, y en qué metamodelos se basan. El modelo de salida es único; es una instancia del metamodelo desarrollado, e incluye las representaciones de tanto los archivos .java con las clases como los archivos .jml con su especificación.

La herramienta desarrollada utiliza como entrada archivos .oclas, es decir archivos de sintaxis abstracta de OCL. Estos son instancias del metamodelo unificado Pivot. Si se tiene instalado el plugin OCL en Eclipse, al hacer clic derecho en un archivo con extensión .ocl, en la sección **OCL** se presenta la opción **Save Abstract Syntax**, que nos permite crear un archivo .ocl.oclas con su sintaxis abstracta. Aunque se genera un archivo solo, representado en el encabezado por la palabra IN, este hace referencia a clases de 3 otros modelos que instancian el metamodelo Pivot:

- La biblioteca estándar OCL, denominada en el código como LIB. Define los tipos y sus operaciones. Su ubicación es http://www.eclipse.org/ocl/2015/Library.oclas.

- El modelo de las clases y tipos de datos de Ecore, llamada ECO en el código. Está ubicado en http://www.eclipse.org/emf/2002/Ecore.oclas.
- La instancia del modelo Ecore, representada en el código ATL por el nombre MOD.

En la Figura 7.10 se muestra el archivo generado con la sintaxis abstracta del OCL del proyecto Royal and Loyal, visualizado con el editor de Ecore. Se pueden ver los cuatro modelos y sus paquetes.

```
-- @path JML=/MetamodeloJML/src/JML.ecore
-- @nsURI OCL=http://www.eclipse.org/ocl/2015/Pivot

module ocl2jml;
create OUT : JML from IN : OCL, LIB : OCL, ECO: OCL, MOD : OCL;
```

Figura 7.9 : Encabezado del archivo de traducción ATL

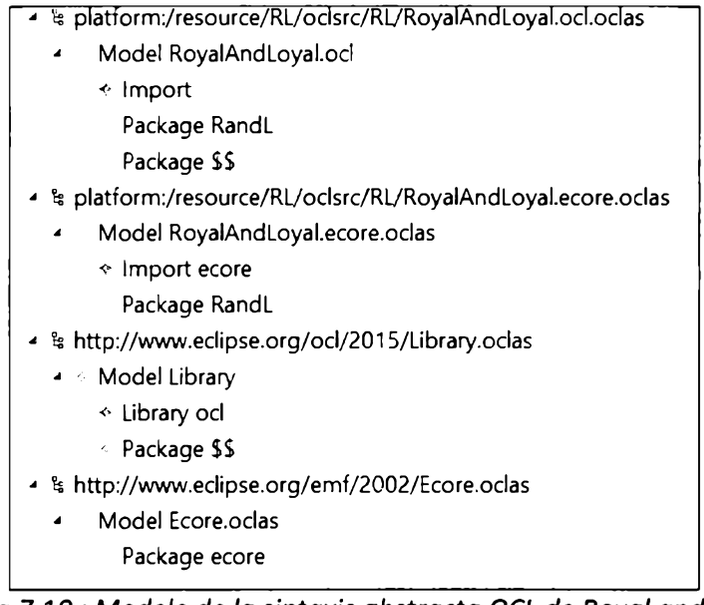


Figura 7.10 : Modelo de la sintaxis abstracta OCL de Royal and Loyal

En la Figura 7.11 se muestra parte del código del módulo ATL que crea el objeto Model del modelo JML a partir del objeto Model del modelo OCL. La clase Model del modeo JML contiene referencias a los paquetes (*ownedElements*), las unidades de compilación, es decir, los archivos con las clases (.java) y especificaciones (.jml), y los tipos de datos básicos, como tipos primitivos, arreglos, etc.

En esta regla se generan también las clases de la librería de colecciones, clases básicas Java como String y los paquetes que las contienen.

```
rule Model2Model {
    from
        mi : OCL!Model in IN
    to
        mo : JML!Model(
        name <- mi.name,
        ownedElements <- mi.ownedPackages->reject(p | p.name = 'ss'),
        ownedElements <- oclcollections,
        ownedElements <- java,
        compilationUnits <- OCL!Class.allInstancesFrom('IN')->select(c   c.oclIsTypeOf(OCL!Class))
            ->collect (c | thisModule.Class2JMLCU(c)),
        compilationUnits <- OCL!Class.allInstancesFrom('MOD')->select(c | c.oclIsTypeOf(OCL!Class))
            -> collect (c | thisModule.Class2JavaCU(c)),
        compilationUnits <- OCL!Enumeration.allInstancesFrom('MOD')
            ->collect(c | thisModule.Class2EnumCU(c)),
        orphanTypes <- OCL!DataType.allInstancesFrom('IN'),
        orphanTypes <- object
        ),
        oclcollections : JML!Package (
            name <- 'ocl2jml'
        ),
```

Figura 7.11 : Fragmento del código del módulo principal de la transformación

La regla de transformación de una clase del modelo OCL a una clase con su especificación JML se muestran en la Figura 7.12.

```
rule Class2SpecClass {
    from
        ci : OCL!Class in IN (ci.oclIsTypeOf(OCL!Class))
    to
        co : JML!ClassDeclaration(
            name <- ci.name,
            modifier <- m,
            bodyDeclarations <- ci.modelClass().ownedProperties->
                select (p | not p.isImplicit)->collect(c | thisModule.Property2JMLField(c)),
            typeSpecs <- ts,
            package <- ci.owningPackage
        ),
        m : JML!Modifier (
            visibility <- #public,
            inheritance <- if ci.modelClass().isAbstract then #"abstract" else #"none" endif
        ),
        ts : JML!JMLTypeSpecs (
            typeClauses <- ci.ownedInvariants->collect( i | if (i.isDerived()) then
                thisModule.Derived2Invariant(i) else  thisModule.Invariant2Invariant(i) endif),
            typeClauses <- ci.modelClass().ownedProperties->
                select(p | p.type.isParamType() and not p.isImplicit)->collect (p | thisModule.Collection2Represer
        )
}
```

Figura 7.12: Código de la transformación de clase OCL a clase con especificaciones JML

Esta regla especifica que, dada una clase del metamodelo OCL, se crea una declaración de clase Java, un modificador para esta, y una especificación de tipos. Las cláusulas de

tipo se generan a partir de los invariantes, y además se crean cláusulas *represents* por cada atributo que sea una colección.

Para los invariantes hay dos posibilidades: O son invariantes normales de OCL, o son atributos derivados. En el metamodelo Pivot los atributos derivados se representan como invariantes. Como no hay forma de diferenciarlos en el modelo, la herramienta toma invariantes llamados con el nombre de un atributo como atributos derivados. Evaluar esto es la función del helper *isDerived()* en la línea 470.

En la Figura 7.13 se muestra la regla de transformación de una operación OCL a un método con especificaciones JML. Se crea un solo caso de especificación, ya que OCL no soporta comportamiento normal y excepcional. Las pre y postcondiciones se generan directamente a partir de las de la operación OCL. Esta regla tiene como condición que la operación no sea definida en OCL, y que no tenga un cuerpo, ya que en esos casos la traducción se realiza de forma distinta. En caso de ser una operación definida, se crea un método modelo; y en caso de ser una operación no definida en OCL pero con cuerpo, se le agrega al método una postcondición que garantice que el resultado del método sea igual a la expresión cuerpo.

```
rule Operation2SpecMethod {
from
    o : OCL!Operation in IN (not o.isDefOperation() and o.bodyExpression.oclIsUndefined())
to
        me : JML!MethodDeclaration (
        abstractTypeDeclaration <- o.owningClass,
        name <- o.name,
        modifier <- m,
        returnType <- ta,
        parameters <- o.ownedParameters->collect(p   thisModule.Parameter2VariableDecl(p)),
        methodSpecs <- ms
    ),
        m : JML!Modifier (
        visibility <- #public
    ),
        ta : JML!TypeAccess (
        type <- if o.type.isParamType() then thisModule.ParamType2ParamType(o.type) else o.type endif
    ),
        ms : JML!JMLMethodSpecs (
        specCases <- sc
    ),
        sc : JML!JMLSpecificationCase (
        modifier <- mod$,
        clauses <- o.ownedPreconditions->collect(p   thisModule.Precondition2Requires(p)),
        clauses <- o.ownedPostconditions->collect(p   thisModule.Postcondition2Ensures(p))
    ),
        mod$ : JML!Modifier (
        visibility <- #public
    )
}
```

Figura 7.13: Código de la transformación de una operación a un método con especificaciones JML

Para tener la posibilidad de obtener como resultado de la traducción un sistema completo, se agregó una forma de definir el código del cuerpo de las operaciones directamente en lenguaje Java, mediante el uso de anotaciones. Se puede definir el código de una operación agregando una anotación con fuente **OCL2JML**, con una entrada con clave **methodbody** y valor el código que define su comportamiento, como se ilustra en la Figura 7.14.

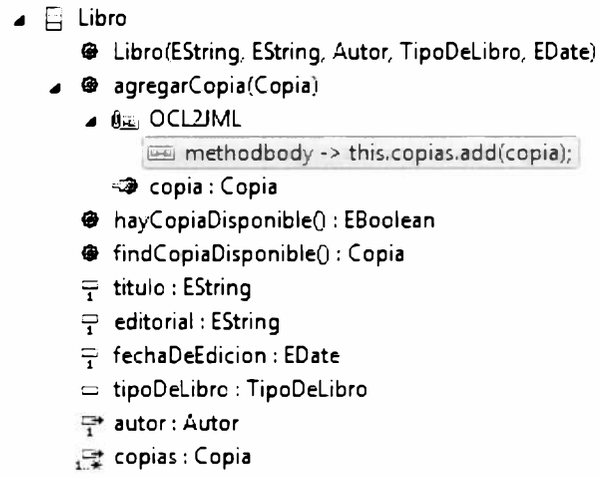


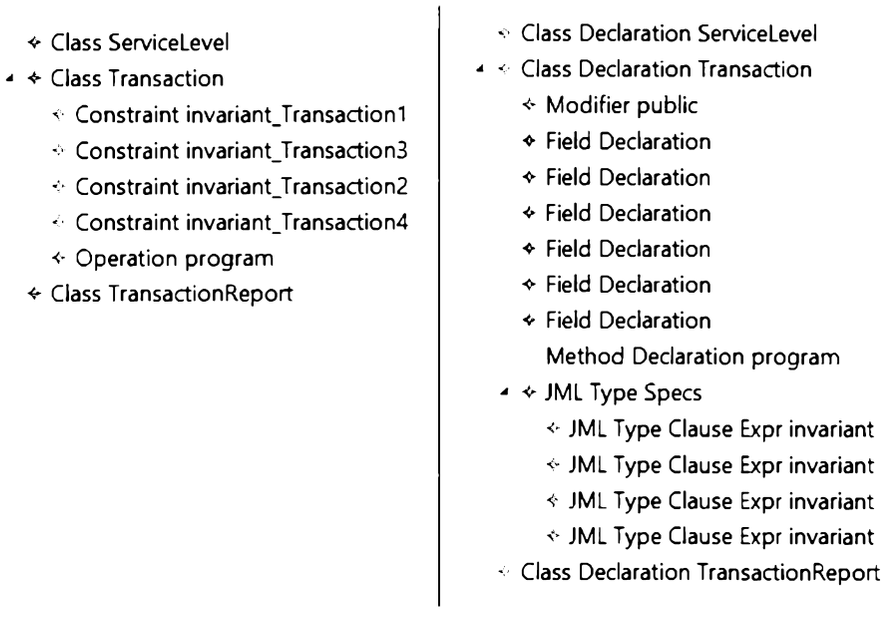
Figura 7.14: Ejemplo de anotación para la generación de código de operaciones

En este caso el método *agregarCopia(Copia)* de la clase Libro, está definido en lenguaje Java mediante la sentencia *this.copias.add(copia);*.

El módulo ATL, al encontrar una operación con una anotación OCL2JML con entrada methodbody, genera un statement especial del metamodelo JML que incluye todo el código del valor de esta entrada. Este statement luego será traducido a código directamente. De esta forma se permite crear a partir del modelo métodos Java con definiciones completas.

Una vez que se ejecuta el módulo ATL sobre el archivo .oclas, se obtiene como resultado un archivo .xmi con una instancia del metamodelo JML. En la Figura 7.15 se muestra una comparación de parte de los modelos entrada y salida de la transformación, siendo el modelo de entrada el modelo Royal And Loyal. Se puede ver cómo se generan clases por cada clase, métodos por cada operación y expresiones de invariante por cada invariante. Los atributos no son parte del modelo mostrado,

porque el modelo de las restricciones OCL no los incluye, sino que están en el modelo del modelo Ecore, mencionado anteriormente.



Figuro 7.15: Parte de modelo entrada (izquierda) y salida (derecha) de la transformación ATL

El modelo resultante de la traducción ATL luego se usa como entrada para el módulo de traducción de modelo a texto, que será descrito a continuación.

7.6 Traducción de modelo JML a código Java+JML

La traducción de modelo Java+JML a código fue realizada utilizando la herramienta Acceleo, introducida en la sección 5.3.

El template principal del módulo, a partir del cual comienza la ejecución de este, se puede ver en la Figura 7.16. Por cada unidad de compilación del metamodelo JML, se crea un archivo, además de carpetas por los paquetes que lo incluyan. En este template se invocan los templates *generateClass* y *generateEnum*, que generarán el contenido de los archivos, con sus sentencias y expresiones.

En la Figura 7.17 se muestra el template que genera las clases, mediante la invocación de otros templates. Primero genera las declaraciones de enumerativos, de haberlas, y los campos. Entre símbolos /*@ ... */, genera las especificaciones JML, si es que están definidas. Y finalmente se generan las declaraciones de métodos y clases internas.

En la traducción realizada en este trabajo, se genera por cada clase Ecore del modelo original un archivo .java y un archivo .jml con sus especificaciones.

Esta es una traducción completa del metamodelo JML desarrollado, por lo que incluye muchas expresiones y tipos de cláusulas que no son utilizados en la herramienta desarrollada en sí. Debido a esto el metamodelo JML y su módulo Acceleo pueden ser utilizados independientemente de la traducción de OCL.

```
[template public generateElement(aCU : CompilationUnit)]
[comment @main/]
[let path : String = getPackagePath(aCU._package)]
[file (path+'/'+ aCU.name, false, 'UTF-8')]
package [generatePackageName(aCU._package)/];

[for (imp : ImportDeclaration | aCU.imports) ]
import [generateImport(imp)/];
[/for]

[for (t : AbstractTypeDeclaration | aCU.types) ]
    [if (t.oclIsTypeOf(ClassDeclaration))]
[generateClass(t.oclAsType(ClassDeclaration))/]
    [elseif (t.oclIsTypeOf(EnumDeclaration))]
[generateEnum(t.oclAsType(EnumDeclaration))/]
    [/if]
[/for]
[/file]
[/let]
[/template]
```

Figura 7.16: Template principal del módulo Acceleo

```
[template public generateClass(c : ClassDeclaration)]
[generateModifiers(c.modifier,false)/] class [c.name/][if (c.typeParameters->size()>0)][for (tp : TypeParameter
    [for (el : EnumDeclaration | c.bodyDeclarations->selectByType(EnumDeclaration))]
    [generateEnum(el)/]
    [/for]

    [for(f : FieldDeclaration | c.bodyDeclarations->selectByType(FieldDeclaration))]
    [generateField(f,false)/]
    [if (not f.fieldSpecs.oclIsUndefined())]
    /*@
    [generateFieldSpecs(f.fieldSpecs)/]
    */[/if]
    [/for]

    [if (not c.typeSpecs.oclIsUndefined() or c.specDeclarations->size()>0)]
    /*@
        [for(f : FieldDeclaration | c.specDeclarations->selectByType(FieldDeclaration))]
        [generateField(f,true)/]
        [/for]
        [for(tc : JML_TypeClause | c.typeSpecs.typeClauses->selectByType(JML_TypeClauseRepresents))]
        [generateTypeClause(tc)/]
        [/for]
        [for(m: AbstractMethodDeclaration | c.specDeclarations->selectByKind(AbstractMethodDeclaration))]
        [generateMethod(m,true)/]
        [/for]
        [for (tc : JML_TypeClause | c.typeSpecs.typeClauses->reject(t | t.oclIsTypeOf(JML_TypeClauseRepresents)))]
        [generateTypeClause(tc)/]
        [/for]
    */
    [/if]

    [for (m : AbstractMethodDeclaration | c.bodyDeclarations->selectByKind(AbstractMethodDeclaration))]
    [generateMethod(m,false)/]
    [/for]
    [for (cl : ClassDeclaration | c.bodyDeclarations->selectByType(ClassDeclaration))]
    [generateClass(cl)/]
    [/for]
}
[/template]
```

Figura 7.17: Parte del template que genera las clases

7.7 Ejecución de la herramienta

Previo a la instalación de la herramienta en Eclipse, es necesario tener instalados los plugins de Acceleo, ATL y OCL, ya que son utilizados en su ejecución. La herramienta desarrollada se instala como plugin de Eclipse, que a su vez se basa en 3 plugins más que implementan su funcionalidad: el del metamodelo JML, el de la transformación ATL, y el de la generación de código Acceleo. Luego de la instalación de los 4 plugins, se puede utilizar la funcionalidad de la herramienta en el entorno Eclipse.

Para empezar, primero debemos contar con un modelo Ecore y un documento Complete OCL con restricciones aplicadas al mismo. Teniendo el plugin de OCL instalado, al hacer clic derecho sobre este documento surge la opción **OCL**, y dentro de esta, **Save Abstract Syntax**, como se ve en la Figura 7.18. Debemos cliquearla y escribir el nombre del archivo a generar.

Figura 7.18: Opción del menú para generar la sintaxis abstracta de OCL

Luego, haciendo clic derecho en este archivo vemos la opción **Translate to JML**, como se muestra en la Figura 7.19. Esta opción sólo está disponible para archivos con extensión **.oclas**. Al cliquearla se realiza la traducción y se crea un archivo .xmi con el modelo JML, y una carpeta con las clases Java y sus especificaciones.

Aunque en este ejemplo los archivos .ecore, .ocl y .oclas tienen el mismo nombre, no es necesario que sea así, sino que pueden tener nombres diferentes.

A continuación se mostrará la ejecución de la herramienta sobre los casos de estudio descritos.

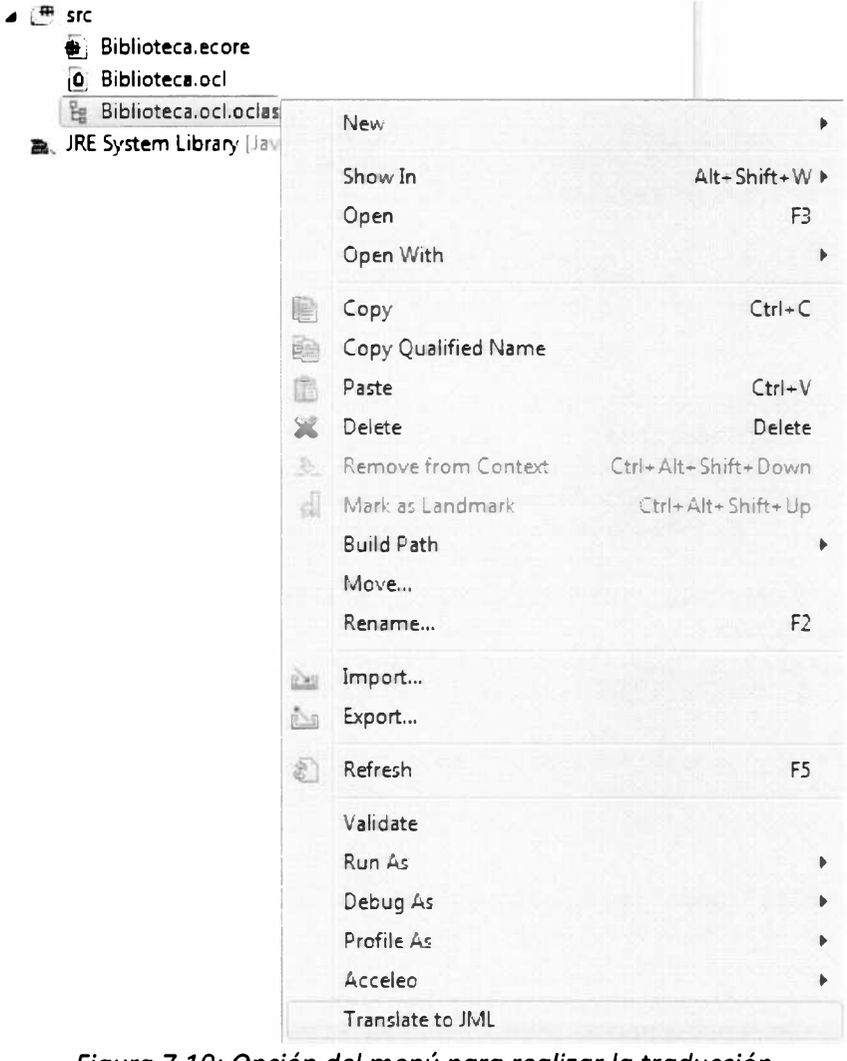


Figura 7.19: Opción del menú para realizar la traducción

7.7.1 Modelo Royal and Loyal

Al ejecutar la herramienta sobre el modelo Royal and Loyal, se crea un archivo .java y un archivo .jml por cada clase del modelo, y un archivo .java por cada enumerativo.

Como ejemplo, dadas las restricciones OCL de la clase LoyaltyAccount, mostradas a continuación:

```
context LoyaltyAccount
inv invariant_points :
      (self.points > 0) implies self.transactions->exists( t :
Transaction | t.points >    )
inv invariant_transactions :
      self.transactions->collect( i_Transaction : Transaction |
i_Transaction.points )->exists( p : Integer | p = 300 )
inv invariant_oneOwner :
      (self.transactions->collect( i_Transaction : Transaction |
i_Transaction.card )->collect( i_CustomerCard : CustomerCard |
i_CustomerCard.owner )->asSet()->size()) =

context LoyaltyAccount::points : Integer
init :


context LoyaltyAccount::totalPointsEarned : Integer
derive totalPointsEarned:
      self.transactions->select( i_Transaction : Transaction |
i_Transaction.oclIsTypeOf(Earning) )->collect( i_Transaction :
Transaction | i_Transaction.points )->sum()

context LoyaltyAccount::usedServices : Set(Service)
derive usedServices:
      self.transactions->collect( i_Transaction : Transaction |
i_Transaction.generatedBy )->asSet()

context LoyaltyAccount::transactions : Set(Transaction)
init :
      Set{}

context LoyaltyAccount::getCustomerName() : String
body: self.Membership.card.owner.name

context LoyaltyAccount::isEmpty() : Boolean
post testPostSuggestedName:   result = self.points = 0
pre testPreSuggestedName:     true
```

Al realizar la traducción, el archivo .jml generado tiene el siguiente código:

```
package RandL;

import ocl2jml.collections.*;
import java.util.*;
```


```
public class LoyaltyAccount {

      protected /*@ spec_public*/ java.lang.Integer points;
      protected /*@ spec_public*/ java.lang.Integer totalPointsEarned;
      protected /*@ spec_public*/ java.lang.Integer number;
      protected /*@ spec_public*/ HashSet<Service> usedServices;
      protected /*@ spec_public*/ Membership Membership;
      protected /*@ spec_public*/ HashSet<Transaction> transactions;

      /*@
            public model OCLSet<Service> usedServices_specs;
            public model OCLSet<Transaction> transactions_specs;
            represents usedServices_specs =
OCLSet.convertFrom(usedServices);
            represents transactions_specs =
OCLSet.convertFrom(transactions);
            public invariant ((this.points > 0) ==>
this.transactions_specs.exists((t) -> (t.points > 0)));
            public invariant
this.transactions_specs.collect((i_Transaction) ->
i_Transaction.points).exists((p) -> (p == 500));
            public invariant
(this.transactions_specs.collect((i_Transaction) ->
i_Transaction.card).collect((i_CustomerCard) ->
i_CustomerCard.owner).asSet().size() == 1);
            public invariant totalPointsEarned ==
this.transactions_specs.select((i_Transaction) ->
(\typeof(i_Transaction) == \type(Earning))).collect((i_Transaction) ->
i_Transaction.points).sum();
            public invariant usedServices ==
this.transactions_specs.collect((i_Transaction) ->
i_Transaction.generatedBy).asSet();
            public initially points == 0;
            public initially transactions == null;
      */

      /*@
            requires true;
            ensures ((\result == this.points) == 0);

      */
      public java.lang.Boolean isEmpty ();

      /*@
            ensures \result == this.Membership.card.owner.name;

      */
```

```
    public String getCustomerName ();

}
```

Se puede observar cómo por cada invariante, pre y postcondición OCL se genera su equivalente en JML. Gracias al uso de la biblioteca de colecciones, la traducción es fácilmente trazable a su código OCL original. Además, las restricciones aplicadas a las colecciones en el OCL, son aplicadas a sus abstracciones en el código JML resultante, logrando una mayor independencia de la implementación.

Como el modelo Royal and Loyal no está pensado para ser ejecutado, sino que funciona como un ejemplo completo de OCL, y además no todos los elementos del JML resultante están todavía implementados en OpenJML, se mostrará la verificación de las restricciones con el modelo de la biblioteca descrito en la sección 7.4.2.

7.7.2 Modelo de la biblioteca

La ejecución de OpenJML será por medio de la línea de comandos. Para simplificar los comandos usados, se definieron las siguientes variables de entorno:

- OJ: La ruta del directorio del ejecutable OpenJML
- OCLCOL: La ruta del directorio donde se encuentra la biblioteca de colecciones desarrollada en este trabajo
- SOURCEPATH: La ruta del directorio donde se encuentra el código fuente
- Z3: La ruta del directorio del ejecutable del probador de teoremas Z3, utilizado en este trabajo para realizar la verificación estática

La versión de OpenJML utilizada es la 0.8.24, que es la más reciente a la fecha. A continuación se mostrará la ejecución de OpenJML sobre el modelo descrito de la biblioteca.

7.7.2.1 Verificación estática

OpenJML nos permite analizar el código estáticamente, es decir, sin ejecutarlo. Esto se realiza mediante el uso de probadores de teoremas. Como la herramienta de verificación estática de OpenJML no soporta tipos complejos, para mostrar su ejecución se utilizará la clase Empleado del modelo de la biblioteca. Es una clase simple, que cuenta con los atributos *nombre, apellido, rol, fechaDeNacimiento* y

sueldoxhora. Además, tiene una operación que permite aumentar su sueldo. En la Figura 7.20 se muestran las restricciones OCL definidas para la clase.

```
context Empleado::Empleado (nombre : String, apellido : String, rol : String,
    fechaDeNacimiento : ecore::EDate, sueldoxhora : ecore::EInt
): OclVoid
pre : sueldoxhora >

context Empleado::aumentarSueldo (aumento : ecore::EInt) : OclVoid
post mayor: sueldoxhora > sueldoxhora@pre
post : sueldoxhora = aumento + sueldoxhora@pre
```

Figura 7.20: Restricciones OCL de un empleado

Se definió como invariante que el sueldo de un empleado debe ser siempre mayor a 0. La operación para aumentar el sueldo, definida en el modelo como una simple suma, tiene como postcondición que el sueldo sea mayor al anterior, y que sea igual a la suma del sueldo anterior y su aumento. Luego de ejecutar la herramienta, el archivo .jml generado contiene el código siguiente:

```
package Biblioteca;

import ocl2jml.collections.*;
import java.util.*;

public class Empleado {

      protected /*@ spec_public*/ String rol;
      protected /*@ spec_public*/ String apellido;
      protected /*@ spec_public*/ String nombre;
      protected /*@ spec_public*/ java.util.Date fechaDeNacimiento;
      protected /*@ spec_public*/ int horasSemanales;
      protected /*@ spec_public*/ int sueldoxhora;

      /*@
            public invariant (this.sueldoxhora > 0);
      */

      /*@
            requires (sueldoxhora > 0);

      */
      public Empleado (String nombre, String apellido, String rol,
java.util.Date fechaDeNacimiento, int sueldoxhora);

      /*@
```

```
            ensures (this.sueldoxhora > \old(this.sueldoxhora));
            ensures (this.sueldoxhora == (aumento +
\old(this.sueldoxhora)));

      */
      public void aumentarSueldo (int aumento);

}
```

Al ejecutar OpenJML con la opción ESC (Extended Static Checking) sobre los archivos generados, obtenemos el resultado mostrado en la Figura 7.21.

```
caro@caro-UX305FA:~$ java -jar $OJ/openjml.jar -cp $OCLCOL/ocl2jmlcollections.ja
r -esc -prover z3_4_4 -exec $Z3/z3 -sourcepath $SOURCEPATH/ -specspath $SOURCEPA
TH /home/caro/PruebasTesina/src/Biblioteca/Empleado.java
/home/caro/PruebasTesina/src/Biblioteca/Empleado.java:22: warning: The prover ca
nnot establish an assertion (InvariantExit: /home/caro/PruebasTesina/src/Bibliot
eca/Empleado.jml:16: ) in method aumentarSueldo
        public void aumentarSueldo (int aumento){
                    ^
/home/caro/PruebasTesina/src/Biblioteca/Empleado.jml:16: warning: Associated dec
laration: /home/caro/PruebasTesina/src/Biblioteca/Empleado.java:22:
                public invariant (this.sueldoxhora > 0);
                                  ^
/home/caro/PruebasTesina/src/Biblioteca/Empleado.java:22: warning: The prover ca
nnot establish an assertion (Postcondition: /home/caro/PruebasTesina/src/Bibliot
eca/Empleado.jml:27: ) in method aumentarSueldo
        public void aumentarSueldo (int aumento){
                    ^
/home/caro/PruebasTesina/src/Biblioteca/Empleado.jml:27: warning: Associated dec
laration: /home/caro/PruebasTesina/src/Biblioteca/Empleado.java:22:
                ensures (this.sueldoxhora > \old(this.sueldoxhora));
                ^
Note: /home/caro/PruebasTesina/openjml/openjml.jar(specs18/java/util/Random.jml)
 uses internal proprietary API that may be removed in a future release.
Note: Recompile with -Xlint:sunapi for details.
4 warnings
caro@caro-UX305FA:~$
```

Figura 7.21: Primera ejecución de ESC

La herramienta genera las siguientes advertencias:

- En el método *aumentarSueldo*, no se puede garantizar el invariante que dice que *sueldoxhora* debe ser siempre mayor a 0.
- El mismo método tampoco puede garantizar que valga la postcondición que asegura que el nuevo sueldo sea mayor que el anterior.

Estas advertencias se deben a que *aumentarSueldo* toma como parámetro un entero, y no hay garantía de que este sea mayor a 0. De esta forma no se garantiza ni que el sueldo sea siempre positivo, ni que aumentar el sueldo efectivamente lo haga ser

mayor. Para solucionar esto, le agregaremos a la operación *aumentarSueldo* la siguiente precondición:

```
pre : aumento >0
```

De esta forma, se deberían garantizar el invariante y las postcondiciones de la operación. Volvemos a ejecutar la herramienta de traducción, y se genera su respectiva precondición JML. Luego de realizar otra vez la traducción, se obtiene el resultado mostrado en la Figura 7.22.

```
caro@caro-UX305FA:~$ java -jar $OJ/openjml.jar -cp $OCLCOL/ocl2jmlcollections.ja
r -esc -prover z3_4_4 -exec $Z3/z3 -sourcepath $SOURCEPATH/ -specspath $SOURCEPA
TH /home/caro/PruebasTesina/src/Biblioteca/Empleado.java
Note: /home/caro/PruebasTesina/openjml/openjml.jar(specs18/java/util/Random.jml)
 uses internal proprietary API that may be removed in a future release.
Note: Recompile with -Xlint:sunapi for details.
caro@caro-UX305FA:~$
```

Figura 7.22: Segunda ejecución de ESC

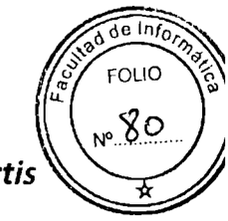
Como se puede ver, esta vez la herramienta no generó ninguna advertencia, lo cual significa que el código satisface su especificación JML. Además, la falta de advertencias significa que el código JML no tiene errores de tipo.

7.7.2.2 Verificación en tiempo de ejecución

OpenJML también nos permite realizar verificación del programa en tiempo de ejecución. Es decir, si durante la ejecución del programa se viola alguna de las especificaciones JML, se produce una advertencia. Se mostrará la ejecución de la verificación en tiempo de ejecución aplicada al modelo entero de la biblioteca. A continuación se muestran las restricciones OCL enteras del modelo.

```
import 'Biblioteca.ecore'
package Biblioteca

      context Biblioteca::agregarLibro( libro : Libro) : OclVoid
            post : libros->includes(libro)
      context Biblioteca::agregarSocio( socio : Socio) : OclVoid
            post : socios->includes(socio)
      context Biblioteca::agregarEmpleado( empleado : Empleado) :
OclVoid
            post : empleados->includes(empleado)

      context Libro
```

```
        inv property_distinto: self.titulo <> ''

    context Libro::agregarCopia( copia : Copia) : OclVoid
        post : copias->includes(copia)

    context Libro::hayCopiaDisponible() : Boolean
        post : (copias->size() = 0) implies result = false

    context Socio
        inv nombre: nombre <> '' and apellido <> ''
        inv nMultas: self.multas->size() <= self.prestamos->size()

    context Socio::suspendido : Boolean
        init: false

    context Socio::solicitarPrestamo(libro : Libro) : Prestamo
        pre : libro.hayCopiaDisponible()
        pre susp : suspendido = false
        post : prestamos->includes(result)

    context Socio::generarMulta(prestamo : Prestamo) : Multa
        pre : prestamos->includes(prestamo)
        post : multas->includes(result)

    context Socio::suspender() : OclVoid
        pre : multas->size()>0

    context Prestamo::finalizar(fechaDeDevolucion : ecore::EDate) :
OclVoid
        post: self.fechaDeDevolucion = fechaDeDevolucion
        post copia: self.copia.estaDisponible()

    context Copia::devolver() : OclVoid
    post : self.estaDisponible()

    context Copia::reservar() : OclVoid
    post : not self.estaDisponible()

    context Empleado
        inv : sueldoxhora > 0

    context Empleado::Empleado (nombre : String, apellido : String,
rol : String,
        fechaDeNacimiento : ecore::EDate, sueldoxhora :
ecore::EInt
```




```
      ): OclVoid
      pre : sueldoxhora >

      context Empleado::aumentarSueldo (aumento : ecore::EInt) :
OclVoid
            pre : aumento >
            post mayor: sueldoxhora > sueldoxhora@pre
            post : sueldoxhora = aumento + sueldoxhora@pre

      context Autor
            inv nombre: nombre <> '' and apellido <> ''

endpackage
```

Como se puede ver en el código, además de postcondiciones que definen el comportamiento esperado de las operaciones, e invariantes que garantizan que algunos atributos no sean vacíos, se definen, entre otras, las siguientes restricciones:

- Para que un socio pueda solicitar un préstamo, debe haber una copia disponible del libro, y el socio no debe estar suspendido.
- No se le puede generar una multa a un socio de un préstamo que no haya solicitado.
- No se puede suspender a un socio si no tiene ninguna multa.

Luego de ejecutar la traducción, se obtienen una vez más los archivos .java y .jml. En este caso, el archivo Socio.jml tiene el siguiente contenido:

```
package Biblioteca;

import ocl2jml.collections.*;
import java.util.*;

public class Socio {

      protected /*@ spec_public*/ int numeroDeSocio;
      protected /*@ spec_public*/ String apellido;
      protected /*@ spec_public*/ String nombre;
      protected /*@ spec_public*/ String telefono;
      protected /*@ spec_public*/ java.util.Date fechaDeNacimiento;
      protected /*@ spec_public*/ LinkedHashSet<Prestamo> prestamos;
      protected /*@ spec_public*/ LinkedHashSet<Multa> multas;
      protected /*@ spec_public*/ boolean suspendido;
```

```
	/*@
		public model OCLOrderedSet<Prestamo> prestamos_specs;
		public model OCLOrderedSet<Multa> multas_specs;
		represents prestamos_specs =
OCLOrderedSet.convertFrom(prestamos);
		represents multas_specs =
OCLOrderedSet.convertFrom(multas);
		public invariant (!this.nombre.equals("")
&& !this.apellido.equals(""));
		public invariant (this.multas_specs.size() <=
this.prestamos_specs.size());
		public initially suspendido == false;
	*/

	/*@
		requires this.prestamos_specs.includes(prestamo);
		ensures this.multas_specs.includes(\result);

	*/
	public Multa generarMulta (Prestamo prestamo);

	/*@
		requires libro.hayCopiaDisponible();
		requires (this.suspendido == false);
		ensures this.prestamos_specs.includes(\result);

	*/
	public Prestamo solicitarPrestamo (Libro libro);

	/*@
		requires (this.multas_specs.size() > 0);

	*/
	public void suspender ();

}
```

Para realizar la verificación en tiempo de ejecución, necesitamos crear, en alguna de las clases Java generadas en la traducción, un método main que ejecute los métodos necesarios. En este ejemplo se le agregó un método main a la clase Biblioteca, como se ve en la Figura 7.23.

```
package Biblioteca;

import java.util.*;

public class Biblioteca {

        protected String nombre;
        protected String direccion;
        protected LinkedHashSet<Socio> socios;
        protected LinkedHashSet<Libro> libros;
        protected LinkedHashSet<Empleado> empleados;


        public Biblioteca (String direccion, String nombre){
                socios= new LinkedHashSet<Socio>();
                libros = new LinkedHashSet<Libro>();
                empleados = new LinkedHashSet<Empleado>();
                this.nombre = nombre;
                this.direccion = direccion;
        }
        public void agregarLibro (Libro libro){
                this.libros.add(libro);
        }
        public void agregarSocio (Socio socio){
                this.socios.add(socio);
        }

        public void agregarEmpleado (Empleado empleado){
                this.empleados.add(empleado);
        }

        public static void main (String[] args){
                Biblioteca b = new Biblioteca("50 y 120","Biblioteca Informatica");
                Autor a = new Autor("Carolina","Actis","argentina",new Date());
                Libro l = new Libro("OCL2JML","UNLP",a,TipoDeLibro.Tecnico,new Date());
                Socio s = new Socio("Juan","Perez","2215678910",new Date());
                b.agregarLibro(l);
                s.solicitarPrestamo(l);
                l.agregarCopia(new Copia(1,EstadoDeLaCopia.Bueno));
                l.agregarCopia(new Copia(2,EstadoDeLaCopia.Bueno));
                s.solicitarPrestamo(l);
                s.suspender();
                s.solicitarPrestamo(l);
        }
}
```

Figura 7.23: Clase Biblioteca junto con su método main

Luego debemos compilar las clases necesarias para su ejecución. Esto se hace mediante la opción RAC (runtime assertion checking), como se muestra en la Figura 7.24. Así se generan archivos compilados .class por cada clase involucrada en la ejecución. Una vez más la falta de advertencias significa que no se encontraron errores de tipo en el código.

```
caro@caro-UX305FA:~/PruebasTesina/src$ java -jar SOJ/openjml.jar -cp SOCLCOL/ocl
2jmlcollections.jar -rac -sourcepath $SOURCEPATH/ -specspath $SOURCEPATH  /home/
caro/PruebasTesina/src/Biblioteca/Biblioteca.java -nullablebydefault
Note: /home/caro/PruebasTesina/openjml/openjml.jar(specs18/java/util/Random.jml)
 uses internal proprietary API that may be removed in a future release.
Note: Recompile with -Xlint:sunapi for details.
```

Figura 7.24: Compilación RAC del código

Luego de compilar los archivos, se ejecuta el método main mediante el comando mostrado en la Figura 7.25.

```
caro@caro-UX305FA:~/PruebasTesina/src$ java -classpath ".:$OD/jmlruntime.jar:$OC
L_COL/ocl2jmlcollections.jar" Biblioteca.Biblioteca
/home/caro/PruebasTesina/src/Biblioteca/Biblioteca.java:38: JML precondition is
false
                s.solicitarPrestamo(l);
                                    ^
/home/caro/PruebasTesina/src/Biblioteca/Socio.java:17: Associated declaration: /
home/caro/PruebasTesina/src/Biblioteca/Biblioteca.java:38:
        public Prestamo solicitarPrestamo (Libro libro){
                        ^
/home/caro/PruebasTesina/src/Biblioteca/Biblioteca.java:42: JML precondition is
false
                s.suspender();
                              ^
/home/caro/PruebasTesina/src/Biblioteca/Socio.java:44: Associated declaration: /
home/caro/PruebasTesina/src/Biblioteca/Biblioteca.java:42:
        public void suspender (){
                    ^
/home/caro/PruebasTesina/src/Biblioteca/Biblioteca.java:43: JML precondition is
false
                s.solicitarPrestamo(l);
                                    ^
/home/caro/PruebasTesina/src/Biblioteca/Socio.java:17: Associated declaration: /
home/caro/PruebasTesina/src/Biblioteca/Biblioteca.java:43:
        public Prestamo solicitarPrestamo (Libro libro){
                        ^
caro@caro-UX305FA:~/PruebasTesina/src$
```

Figura 7.25: Primera ejecución de RAC

La herramienta genera tres advertencias:

- En el primer intento de solicitar un préstamo, se viola una precondición. Como se puede ver en el código, al solicitar el préstamo el libro no tiene ninguna copia disponible.
- Cuando se intenta suspender al socio, se viola la precondición del método, que dice que no se puede suspender a un socio que no tenga multas. Se puede ver en el código que en ningún momento se generó una multa para el socio.
- A la tercera solicitud de un préstamo, se viola la precondición que dice que un socio suspendido no puede realizar préstamos. Esto sucede porque se invocó al método *suspender()*, y luego al método *solicitarPrestamo()*, con el mismo socio

Veremos qué sucede entonces si arreglamos el método main para que no se den estas situaciones indeseadas. Luego de los cambios realizados, el código del método queda como se ve en la Figura 7.26.

```
public static void main (String[] args){
        Biblioteca b = new Biblioteca("50 y 120","Biblioteca Informatica");
        Autor a = new Autor("Carolina", "Actis","argentina",new Date());
        Libro l = new Libro("OCL2JML","UNLP",a,TipoDeLibro.Tecnico,new Date());
        Socio s = new Socio("Juan","Perez","2215078910",new Date());
        b.agregarLibro(l);
        l.agregarCopia(new Copia(1,EstadoDeLaCopia.Bueno));
        l.agregarCopia(new Copia(2,EstadoDeLaCopia.Bueno));
        Prestamo p = s.solicitarPrestamo(l);
        s.generarMulta(p);
        s.suspender();
}
```

Figura 7.26: Método main arreglado de la clase Biblioteca

Volvemos a compilar las clases, y las ejecutamos una vez más, como se muestra en la Figura 7.27.

```
caro@caro-UX305FA:~/PruebasTesina/src$ java -jar $OJ/openjml.jar -cp $OCLCOL/ocl
2jmlcollections.jar -rac -sourcepath $SOURCEPATH/ -specspath $SOURCEPATH  /home/
caro/PruebasTesina/src/Biblioteca/Biblioteca.java -nullablebydefault
Note: /home/caro/PruebasTesina/openjml/openjml.jar(specs18/java/util/Random.jml)
 uses internal proprietary API that may be removed in a future release.
Note: Recompile with -Xlint:sunapi for details.
caro@caro-UX305FA:~/PruebasTesina/src$ java -classpath ".:$OJ/jmlruntime.jar:$OC
LCOL/ocl2jmlcollections.jar" Biblioteca.Biblioteca
caro@caro-UX305FA:~/PruebasTesina/src$
```

Figura 7.27: Segunda ejecución de RAC

Esta vez no se produce ninguna advertencia, lo que significa que se cumplió con toda la especificación durante la ejecución de los métodos de las clases.

7.8 Resumen

En este capítulo se describieron en detalle todos los módulos de la herramienta desarrollada, y cómo se relacionan para lograr la traducción de OCL a JML. Se mostró la ejecución de la herramienta sobre dos casos de estudio completos. Y se describió la ejecución de las herramientas de verificación sobre un caso de estudio, mostrando así el funcionamiento de JML y la utilidad de la traducción realizada.

8 Trabajos relacionados

En este capítulo se presentan distintos trabajos relacionados al tema de esta tesina, algunos de los cuales sirvieron como base para esta, y otros que tratan sobre temas similares a la misma.

8.1 A Library-Based Approach to Translating OCL Constraints to JML Assertions for Runtime Checking

En [**34**] se propone traducir restricciones OCL a código JML, de forma que pueda ser ejecutado y evaluado en tiempo de ejecución. Ya en [**35**] se había planteado una función de traducción inicial de OCL a JML. En este trabajo se agregan los siguientes aportes:

- Para la traducción utilizan una biblioteca de clases que implementan los tipos de colección definidos la biblioteca estándar OCL. De esta forma la traducción es intuitiva y fácilmente trazable al código OCL original.
- Uso de variables modelo. Las variables modelo representan una abstracción de las variables del programa, y sólo pueden ser usadas en la especificación JML, y no en el código fuente. En este trabajo se utilizan para abstraer las colecciones. De esta forma, la especificación es independiente del tipo de colección utilizada en el código fuente, ya que las expresiones JML no se aplican a la colección en sí, sino a la variable modelo que la representa.
- Se propone separar el código JML del código fuente, de forma que un cambio en el OCL no implique volver a generar el código fuente Java.

En la presente tesina, se usó como base el enfoque propuesto en este trabajo.

8.2 Desarrollo de una herramienta para derivación automática de especificaciones OCL a JML

En [**39**] se desarrolló un plugin de Eclipse que implementa la traducción de OCL a JML, basándose lo propuesto en [**34**]. Se implementó una biblioteca de colecciones OCL como la planteada en dicho trabajo. La traducción en sí se realizó utilizando la herramienta MOFScript [**42**].

Se utilizó en este trabajo también OpenJML para realizar verificación estática de los resultados de la traducción, mediante el probador de teoremas yices [**32**].

La traducción realizada utiliza la versión 7 de Java.

8.3 Pattern-based Mapping of OCL Specifications to JML Contracts

En [**43**] se avanza sobre el trabajo hecho en [**34**], y se propone un método para traducir una especificación UML/OCL a una especificación para una implementación de Java, basada en patrones de restricciones, definidos en [**44**]. Las expresiones de restricción más comunes se generalizan y capturan como patrones de restricción; estos se pueden instanciar para generar restricciones concretas.

En este trabajo, cada patrón de restricción OCL se traduce a un patrón de especificación JML equivalente. La semántica de cada patrón está descrita como una plantilla JML, es decir, expresiones JML parametrizables. Un patrón JML se puede definir como una función que mapea un conjunto de elementos de metamodelo a una restricción Java.

Los beneficios posibles de este método son mejorar la calidad de las expresiones JML al definirlas de forma más compacta, y facilitar la automatización de la traducción.

8.4 Bidirectional Translation between OCL and JML for Round-trip Engineering

Por otro lado, en [**45**] se propone una técnica de traducción bidireccional entre OCL y JML. En esta técnica el código OCL original se mantiene en los comentarios del código JML, siempre y cuando las sentencias JML generadas no sean modificadas por el usuario. Entonces, en la traducción de JML a OCL, si el JML no fue modificado, se utilizan las sentencias originales OCL. De forma similar, en la traducción de JML a OCL, las sentencias que no pueden ser traducidas son insertadas como comentarios en el OCL. Así, siempre que se pueda se mantiene el código original, y se reduce la posibilidad de que este se vuelva complejo luego de aplicar la traducción bidireccional múltiples veces.

En este trabajo se implementó la herramienta de traducción bidireccional OCL a JML mediante el uso del framework Xtext [**46**].

8.5 Lenguajes formales y derivación automática de código de pruebas a partir de modelos de software con restricciones OCL

En [**47**] se desarrolló una herramienta de Eclipse que permite, a partir de una especificación de un modelo con restricciones OCL, la generación automática de código Java, incluyendo las clases del modelo y sus respectivos Casos de Prueba. Estos casos de prueba están basados en las restricciones OCL. La generación automática de código se realiza mediante el plugin Acceleo.

La herramienta además realiza la traducción del código OCL a Alloy, permitiendo de esta forma la verificación formal de las restricciones. La traducción a Alloy se realiza utilizando AlloyMDA [**48**].

8.6 Aportes

En este trabajo se implementó como plugin de Eclipse la traducción propuesta en [**35**] y [**34**], sumándole decisiones propias de implementación para las diferentes expresiones OCL. La traducción en sí se realizó utilizando el enfoque de MDD, es decir, mediante transformaciones modelo a modelo y modelo a texto. Se desarrolló un metamodelo JML y su traducción completa a código. Además, se realizaron pruebas con OpenJML para comprobar la utilidad de la traducción. Se aprovecharon tecnologías recientes, como por ejemplo, funcionalidades de Java 8. Y se mejoró la biblioteca desarrollada en [**39**].

Todo el código implementado en esta tesina es abierto y de uso libre.

9 Conclusiones y Trabajos Futuros

9.1 Conclusiones

En esta tesina se desarrolló un plugin Eclipse que permite, dado un modelo UML y un archivo de restricciones OCL, transformarlos a código Java con especificaciones JML. La traducción se realiza siguiendo el enfoque MDD: primero, se realiza una transformación modelo a modelo, partiendo de una instancia del metamodelo unificado Pivot de UML y OCL, y convirtiéndola en una instancia de un metamodelo JML. Este metamodelo fue definido para este trabajo, basándose en el metamodelo Java de MoDisco, y en las clases de la implementación del Parser de JML de OpenJML. Luego, mediante una transformación modelo a texto, el modelo resultante de la transformación anterior es transformado a código Java y especificaciones JML.

Se probó la traducción con dos casos de estudio, que cubren una amplia variedad de expresiones OCL. Y se desarrolló un ejemplo completo para probar la utilidad de las especificaciones obtenidas en un sistema coherente. Se usó la herramienta OpenJML para realizar la verificación estática y en tiempo de ejecución de las mismas.

Como resultado del trabajo, entonces, se logró una forma simple y cómoda de convertir expresiones OCL, que por su naturaleza no son directamente verificables, en código Java con JML agregado. El código JML permite documentar el código fuente de forma precisa y comprobable, y verificar que este siga las restricciones especificadas.

Mediante la verificación estática de JML se puede comprobar con probadores de teoremas, en casos simples, que las especificaciones no tengan errores y que el código sea correcto. Esta verificación se realiza sin necesidad de ejecutar el código.

Se puede también compilar las clases Java generadas junto con el código JML, para poder verificar en tiempo de ejecución que las restricciones JML sean cumplidas.

La generación automática del código JML implementada en esta tesina nos permite entonces obtener estos beneficios directamente a partir de un documento de restricciones OCL.

9.2 Trabajos Futuros

Como trabajo futuro, se podría optimizar las traducciones para aprovechar las características de JML. Por ejemplo, en la herramienta desarrollada, una expresión OCL de la siguiente forma:

```
atributo <> null
```

Se traduce a la siguiente expresión:

```
atributo != null
```

Sin embargo, JML cuenta con el modificador *non_null*, que permite especificar la misma semántica, de manera más simple.

Una posibilidad interesante entonces sería la de considerar en la traducción los patrones de restricción descritos en [**44**] y [**43**]. De esta forma posiblemente se podrían lograr traducciones más simples y eficientes, que la simple traducción directa de las expresiones.

Otra posibilidad sería la de implementar la traducción bidireccional. Así, si el usuario modifica el código JML, esto se podría ver reflejado en el OCL original. Esto es lo que se investiga en [**45**].

Se podría también hacer que la herramienta integre OpenJML para lograr más directamente la verificación de las especificaciones traducidas.

10 Bibliografía

[1] Roxana Giandini, Gabriela Perez, Universidad Nacional de La Plata Claudia Pons, *Desarrollo de Software Dirigido por Modelos.*: McGraw-HIll Educación y Edulp, 2010.

[2] (Último acceso: 2017) MDA OMG Model Driven Architecture. [Online]. http://www.omg.org/mda

[3] (Último acceso: 2017) The Object Management Group (OMG). [Online]. http://www.omg.org

[4] Unified Modeling Language Specification 2.0: Infrastructure. OMG doc.smsc/06-02-06,.

[5] Anneke Kleppe Jos Warmer, *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd ed.: Addison Wesley.

[6] A. L. Baker, C. Ruby G.T Leavens, "Preliminary design of JML: A behavioral interface specification language for Java," in *ACM SIGSOFT Software Engineering Notes, 31(3).*, 2006, pp. 1-38.

[7] B.Meyer, "Applying design by contract," in *Computer, 25(10).*, 1992, pp. 40-51.

[8] S. J. Clark et al Mellor, "Model-driven development - Guest editor's introduction," *IEEE Softwa.* , vol. 20, no. 5, pp. 14-18.

[9] G. et al. Booch, "An MDA Manifesto," *The MDA Journal: Model Driven Architecture Straight from*, 2004.

[10] Javier Troya Castilla, Antonio Vallecillo Moreno Francisco Durán Muñoz, *Desarrollo de software dirigido por modelos.*: Universidad Abierta de Catalunya, 2013.

[11] OMG. (2006) Unified Modeling Language Specification 2.0: Infrastructure. OMG doc. smsc/06-02-06. [Online]. http://www.omg.org/spec/UML/2.0/

[12] (Último acceso: 2017) MetaObject Facility | Object Management Group. [Online].

http://www.omg.org/mof/

[13] OMG. (Último acceso: 2017) OCL. [Online]. http://www.omg.org/spec/OCL/

[14] James Gosling, and David Holmes Ken Arnold, *The Java Programming Language Third Edition.*: Addison-Wesley, 2000.

[15] Yoonsik Cheon Gary T. Leavens, *Design by Contract with JML.*, 2006.

[16] (Último acceso: 2017) The Java Modeling Language (JML). [Online]. http://www.eecs.ucf.edu/~leavens/JML/index.shtml

[17] Y. Cheon y G.T. Leavens, "A runtime assertion checker for the Java Modeling Language (JML)," in *Proceedings of International Conference on Software Engineering Research and Practice.*, 2002, pp. 322-328.

[18] (Último acceso: 2017) Getting started with JML. [Online]. https://www.ibm.com/developerworks/library/j-jml/index.html

[19] Marieke Huisman, "Reasoning about JAVA programs in higher order logic with PVS and Isabelle," *IPA dissertation series*, no. 2001-03, 2001.

[20] Yoonsik Cheon and Gary T. Leavens, "A Simple and Practical Approach to Unit Testing: The JML and JUnit Way," in *ECOOP 2002 -- Object-Oriented Programming, 16th European Conference*, pp. 231--255.

[21] Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, Erik Poll Lilian Burdy, *An overview of JML tools and applications.*, 2003.

[22] (Último acceso: 2017) Eclipse Modeling Project. [Online]. https://eclipse.org/modeling/emf/

[23] (Último acceso: 2017) ATL. [Online]. http://www.eclipse.org/atl/

[24] (Último acceso: 2017) Acceleo. [Online]. https://www.eclipse.org/acceleo/

[25] (Último acceso: 2017) MOFM2T 1.0. [Online]. http://www.omg.org/spec/MOFM2T/1.0/

[26] (Último acceso: 2017) OCL 2.4. [Online]. http://www.omg.org/spec/OCL/2.4/

[27] (Último acceso: 2017) MoDisco. [Online]. http://www.eclipse.org/MoDisco/

[28] (Último acceso: 2017) OpenJML. [Online]. http://www.openjml.org/

[29] Aaron Stump, Cesare Tinelli Clark Barrett, *The SMT-LIB Standard Version 2.0.*, 2012.

[30] (Último acceso: 2017) GitHub - Z3Prover/z3: The Z3 Theorem Prover. [Online]. https://github.com/Z3Prover/z3

[31] (Último acceso: 2017) CVC4 - the smt solver. [Online]. http://cvc4.cs.stanford.edu/web/

[32] (Último acceso: 2017) The Yices SMT Solver. [Online]. http://yices.csl.sri.com/

[33] (Último acceso: 2017) GitHub - OpenJML. [Online]. https://github.com/OpenJML/OpenJML

[34] Guillermo Flores Jr, Yoonsik Cheon Carmen Avila, "A Library-Based Approach to Translating OCL Constraints to JML Assertions for Runtime Checking," 2008.

[35] Ali Hamie, "Translating the Object Constraint Language," , 2004.

[36] (Último acceso: 2017) Lambda Expressions (The Java Tutorials > Learning the Java Language > Classes and Objects). [Online]. https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

[37] (Último acceso: 2017) [Online]. https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.modisco.java.doc%2Fmediawiki%2Fjava_metamodel%2Fuser.html

[38] (Último acceso: 2017) JML Reference Manual: Grammar Summary. [Online]. http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_22.html#SEC224

[39] Diego Matías Dodero Mena María Jose Dias Molina, "Desarrollo de una herramienta para derivación automática de especificaciones OCL a JML ´," 2011.

[40] (Último acceso: 2017) GitHub - google/guava: Google core libraries for Java. [Online]. https://github.com/google/guava

[41] Danae Claudia, Ibargüengoytia, María Amalia Lopez, *Reglas de traducción de restricciones entre OCL y LN*. Universidad Nacional de La Plata, 2017.

[42] (Último acceso: 2017) MOFScript Home Page. [Online]. https://www.eclipse.org/gmt/mofscript/

[43] Ali Hamie, "Pattern-based Mapping of OCL Specifications to JML Contracts," Computing Division, Brighton University, Brighton, U.K., 2014.

[44] J. y Turowski, K. Ackermann, "A Library of OCL Specification Patterns for Behavioral Specification of Software Components," in *Proceedings of the 18th international conference on Advanced Information Systems Engineering*, 2006, pp. 255-269.

[45] Kentrao Hanada ,Kozo Okano and Shinji Kusumoto Hiroaki Shimba, "Bidirectional Translation between OCL and JML for Round-trip Engineering," in *2013 20th Asia-Pacific Software Engineering Conference*, 2013.

[46] (Último acceso: 2017) https://eclipse.org/Xtext/.

[47] Ilan Rosenfeld, "Lenguajes formales y derivación automática de código de pruebas a partir de modelos de," UNLP, 2015.

[48] (Último acceso: 2017) AlloyMDA. [Online]. http://sourceforge.net/p/alloymda/wiki/Home/

[49] JML Reference Manual. [Online]. http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_toc.html

[50] Eclipse - The Eclipse Foundation open source community website. [Online]. http://www.eclipse.org/