# Facilitando el Análisis Formal de Código Java Especificado con UML+OCL

Carolina Actis
*Facultad de Informática, Universidad Nacional de La Plata Buenos Aires, Argentina*

Claudia Pons
*Facultad de Informática, UNLP*
*Universidad Abierta Interamericana, UAI*
*Comisión de Investigaciones Científicas CIC, Buenos Aires, Argentina*
*cpons@info.unlp.edu.ar*

Gabriel Baum
*Facultad de Informática, Universidad Nacional de La Plata Buenos Aires, Argentina*
*gbaum@info.unlp.edu.ar*

## Resumen

*El lenguaje UML es 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 las restricciones especificadas en el modelo son verificadas de manera ágil y simple en tiempo de ejecución, y también se habilita el análisis estático de éstas mediante el uso de probadores de teoremas implementados para JML.*

## 1. Introducción

El Desarrollo de Software Dirigido por Modelos (denominado MDD por su acrónimo en inglés, Model Driven Development) [1,2] 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) [3], es una implementación de MDD propuesta por la OMG [4]. 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 [5] 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. Adicionalmente, OCL [6] 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) [7] es un lenguaje de especificación formal que puede ser utilizado para especificar clases e interfaces Java. JML proporciona el concepto de diseño por contrato [8] 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 pos condiciones para sus métodos. Las expresiones están escritas de forma que puedan ser compiladas y e en tiempo de ejecución.

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

Este artículo se organiza de la siguiente forma; En la sección 2 se presentan los lenguajes OCL y JML y se describen sus objetivos y características claves. En la sección 3 se describen las diferentes herramientas utilizadas en el desarrollo de este proyecto. En la sección 4 se realiza una comparación entre los lenguajes OCL y JML, y se describe en detalle la función de traducción realizada. En la sección 5, se describen en detalle los módulos que componen la herramienta desarrollada. En la sección 6 Se muestra un caso de estudio, donde se aplica la herramienta para verificar una modelo UM+OCL, implementado en Java. En la sección 7 se presentan diversos trabajos relacionados con nuestra propuesta y se destacan los aportes originales de la misma. Finalmente, en la sección 8 se presentan las conclusiones finales, y se mencionan trabajos futuros que podrían realizarse a partir del presente trabajo.

## 2. Los lenguajes OCL y JML

### *El lenguaje OCL*

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) [6] 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, 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.

### *El lenguaje JML*

JML [7] es una notación para la especificación formal de clases y métodos Java [9]. 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 (DBC , Diseño-por-Contrato) [8] 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 pos condiciones 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.

JML está diseñado para soportar análisis estático, verificación formal (como la herramienta LOOP [10]),

chequeo en tiempo de ejecución [11], pruebas de unidad [12], y documentación (mediante la herramienta jmldoc). En [13] puede leerse una recopilación de herramientas para JML.

## 3. Herramientas utilizadas

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

### *Eclipse Modeling Framework (EMF)*

El proyecto EMF [14] 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 plug-ins 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) [15]. 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.

### *El Lenguaje ATL*

ATL (ATL Transformation Language) [16] 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.

### *Acceleo*

Acceleo [17] es un proyecto de código abierto que consiste en un generador de código que implementa el estándar MOF Model to Text Language (MTL) [18]. 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.

### *Plugin OCL*

Eclipse OCL es una implementación de la especificación OMG OCL 2.4 [19] para uso con Ecore y metamodelos UML. La funcionalidad principal de OCL que soporta expresiones sobre modelos se llama Essential OCL. 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.

### *OpenJML*

OpenJML [20] 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. 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. 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 [21]. OpenJML fue probado con Z3 [22], CVC4 [23] y Yices [24].

## 4. Traducción de OCL a JML

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

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 pos condiciones, 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.

En este trabajo se optó por seguir para la traducción el enfoque definido en [25], en particular su estrategia para traducir las colecciones. Dicho enfoque fue mejorado haciéndolo más expresivo y eficiente.

### *Invariantes, precondiciones y pos condiciones*

Los invariantes, precondiciones y pos condiciones OCL se traducen directamente a invariantes,

precondiciones y pos condiciones JML, asociadas a sus respectivas clases y métodos.

### *Tipos simples*

Los tipos simples se traducen directamente.

### *Operadores y expresiones*

Los operadores matemáticos y lógicos de OCL se traducen casi directamente. En la Tabla 1 se muestra la traducción de las operaciones lógicas y de igualdad. Las operaciones de números y Strings se tradujeron a sus operaciones Java equivalentes de forma directa.

### *Pseudovariables y operaciones predefinidas*

Las pseudovariables y las operaciones predefinidas se traducen casi directamente, como se muestra en la Tabla 2.

**TABLA 1:** TRADUCCIÓN DE OPERACIONES BÁSICAS

| 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)<br>e1.equals(e2) (objetos) |
| e1 <> e2 | *e1 != e2* (tipos primitivos)<br>!e1.equals(e2) (objetos) |
| if e0 then e1 else e2 endif | (e0? e1 : e2) |

**TABLA 2**: TRADUCCIÓN DE PSEUDOVARIABLES Y OPERACIONES PREDEFINIDAS

| 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 |

### *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 [26] 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 OCL, y la expresión JML resultante de la traducción puede ser muy distinta sintácticamente de la expresión OCL original. Por esta razón en [25] 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 3 se muestra el mapeo de las clases OCL a las clases Java desarrolladas.

**TABLA 3**: TRADUCCIÓN DE LAS COLECCIONES

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

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: alumnos->size() > 0
```

Se genera como resultado el código de la Figura 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 de la 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 1**: Ejemplo de especificaciones JML con colecciones

**Operadores de colección:** 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 [27]. 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)
```

### *Atributos y operaciones definidos*

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.

### *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 pos condiciones 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.

### *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 la siguiente expresion:

```
  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.

### *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 = 0
       else cantMateriasAprobadas > 0 endif
```

Esta se traduce de la siguiente manera:

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

## 5. La Herramienta desarrollada

En esta sección se describe la implementación de la herramienta desarrollada y los módulos que la componen.

### *Diseño*

La herramienta diseñada consiste en un plugin Eclipse compuesto por dos módulos: Uno que realiza la transformación modelo a modelo (M2M), y otro que realiza la transformación modelo a texto(M2T). 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.

### *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 [28] 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 y en la gramática del lenguaje, definida en BNF en [29]. Además, se le agregaron meta-clases para el uso de expresiones Lambda, que serán utilizadas en la traducción.

### *Biblioteca de colecciones OCL*

Como fue mencionado antes, 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 [25]. En [30] 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, etc.

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.

### *Traducción del modelo OCL al modelo JML*

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

En la Figura 2 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. - El modelo de las clases y tipos de datos de Ecore, llamada **ECO** en el código. -La instancia del modelo Ecore, representada en el código ATL por el nombre **MOD**.

```
1 -- @path JML=/MetamodeloJML/src/JML.ecore
2 -- @nsURI OCL=http://www.eclipse.org/ocl/2015/Pivot
3
4 module ocl2jml;
5 create OUT : JML from IN : OCL, LIB : OCL, ECO: OCL, MOD : OCL
```

**Figura 2:** Encabezado del archivo de traducción ATL

En la Figura 3 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 modelo 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.

```
106 rule Model2Model {
107     from
108         mi : OCL!Model in IN
109     to
110         mo : JML!Model(
111         name <- mi.name,
112         ownedElements <- mi.ownedPackages->reject(p | p.name = '$$'),
113         ownedElements <- oclcollections,
114         ownedElements <- java,
115         compilationUnits <- OCL!Class.allInstancesFrom('IN')->select(c | c.oclIsTypeOf(OCL!Class))
116             ->collect (c | thisModule.Class2JMLCU(c)),
117         compilationUnits <- OCL!Class.allInstancesFrom('MOD')->select(c | c.oclIsTypeOf(OCL!Class))
118             -> collect (c | thisModule.Class2JavaCU(c)),
119         compilationUnits <- OCL!Enumeration.allInstancesFrom('MOD')
120             ->collect(c | thisModule.Class2EnumCU(c)),
121         orphanTypes <- OCL!DataType.allInstancesFrom('IN'),
122         orphanTypes <- object
123         ),
124         oclcollections : JML!Package (
125             name <- 'ocl2jml'
126         ),
```

**Figura 3:** Fragmento del código del módulo principal de la transformación

```
453 rule Class2SpecClass {
454     from
455         ci : OCL!Class in IN (ci.oclIsTypeOf(OCL!Class))
456     to
457         co : JML!ClassDeclaration(
458             name <- ci.name,
459             modifier <- m,
460             bodyDeclarations <- ci.modelClass().ownedProperties->
461                 select (p | not p.isImplicit)->collect(c | thisModule.Property2JMLField(c)),
462             typeSpecs <- ts,
463             package <- ci.owningPackage
464         ),
465         m : JML!Modifier (
466             visibility <- #public,
467             inheritance <- if ci.modelClass().isAbstract then #"abstract" else #"none" endif
468         ),
469         ts : JML!JMLTypeSpecs (
470             typeClauses <- ci.ownedInvariants->collect( i | if (i.isDerived()) then
471                 thisModule.Derived2Invariant(i) else  thisModule.Invariant2Invariant(i) endif),
472             typeClauses <- ci.modelClass().ownedProperties->
473                 select(p | p.type.isParamType() and not p.isImplicit)->collect (p | thisModule.Collecti
474         )
475 }
```

**Figura 4:** Código de la transformación de clase OCL a clase con especificaciones JML

```
698 rule Operation2SpecMethod {
699 from
700     o : OCL!Operation in IN (not o.isDefOperation() and o.bodyExpression.oclIsUndefined())
701 to
702         me : JML!MethodDeclaration (
703         abstractTypeDeclaration <- o.owningClass,
704         name <- o.name,
705         modifier <- m,
706         returnType <- ta,
707         parameters <- o.ownedParameters->collect(p | thisModule.Parameter2VariableDecl(p)),
708         methodSpecs <- ms
709     ),
710         m : JML!Modifier (
711         visibility <- #public
712     ),
713         ta : JML!TypeAccess (
714         type <- if o.type.isParamType() then thisModule.ParamType2ParamType(o.type) else o.type endif
715     ),
716         ms : JML!JMLMethodSpecs (
```

**Figura 5:** Código de la transformación de una operación a un método con especificaciones JML

La regla de transformación de una clase del modelo OCL a una clase con su especificación JML se muestran en la Figura 4. 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. Esto es evaluado por la función helper *isDerived()* en la línea 470.

En la Figura 5 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 pos condiciones 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 pos condición que garantice que el resultado del método sea igual a la expresión cuerpo.

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 6.

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. 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.

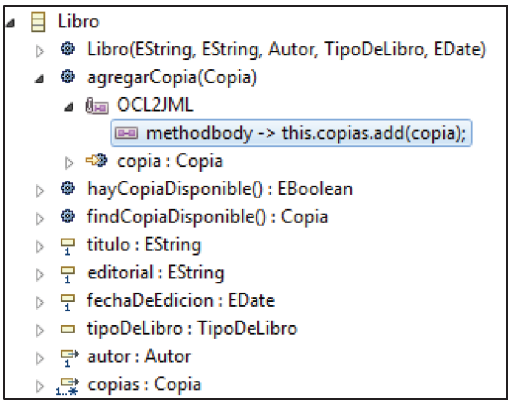


**Figura 6:** Ejemplo de anotación para la generación de código de operaciones

### *Traducción de modelo JML a código Java+JML*

La traducción de modelo Java+JML a código fue realizada utilizando la herramienta Acceleo. El template principal del módulo, a partir del cual comienza la ejecución de este, se puede ver en la Figura 7.

```
6  [template public generateElement(aCU : CompilationUni
7  [comment @main/]
8  [let path : String = getPackagePath(aCU._package)]
9  [file (path+'/'+ aCU.name, false, 'UTF-8')]
10 package [generatePackageName(aCU._package)/];
11
12 [for (imp : ImportDeclaration | aCU.imports) ]
13 import [generateImport(imp)/];
14 [/for]
15
16 [for (t : AbstractTypeDeclaration | aCU.types) ]
17     [if (t.oclIsTypeOf(ClassDeclaration))]
18 [generateClass(t.oclAsType(ClassDeclaration))/]
19     [elseif (t.oclIsTypeOf(EnumDeclaration))]
20 [generateEnum(t.oclAsType(EnumDeclaration))/]
21     [/if]
22 [/for]
23 [/file]
24 [/let]
25 [/template]
```

**Figura 7**: Template principal del módulo Acceleo

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.

## 6. Caso de estudio

### *El modelo del Sistema de Biblioteca*

Para mostrar la utilidad de las herramientas de verificación JML, se tomó como caso de estudio el modelo de una biblioteca, que puede verse en la Figura 8. Adicionalmente en la Figura 9 pueden verse algunas de las restricciones OCL definidas en el modelo.

### *Ejecución de la herramienta sobre el modelo*

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.. Debemos cliquearla y escribir el nombre del archivo a generar.

Luego, haciendo clic derecho en este archivo vemos la opción *Translate to JML*. 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.

Se puede ver cómo se generan clases por cada clase, métodos por cada operación y expresiones de invariante por cada invariante.

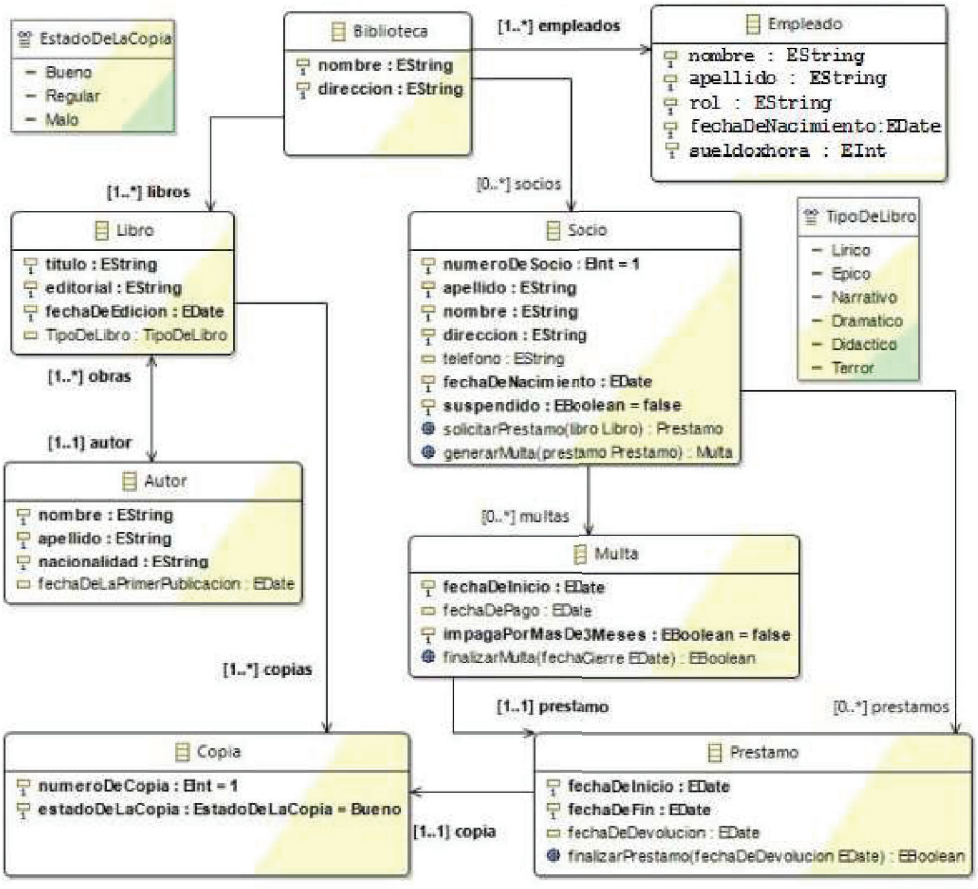


**Figura 8:** Modelo de la biblioteca

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

context Libro::
findCopiaDisponible() : Copia
post: HayCopiaDisponible() implies not result.oclIsUndefined()

context Socio::suspendido : Boolean
init: false

context Socio
inv: nombre<>´´ and apellido<>´´

context Empleado inv: sueldoxhora > 0

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

**Figura 9:** Ejemplo de restricciones OCL para el modelo de la biblioteca

### *Verificación estática*

La versión de OpenJML utilizada es la 0.8.24, que es la más reciente a la fecha. OpenJML nos permite analizar el código estáticamente, es decir, sin ejecutarlo. Esto se realiza mediante el uso de probadores de teoremas. 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 9 se muestran las restricciones OCL definidas para la clase. 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 pos condición 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 que se muestra (parcialmente) en la figura 10.

Al ejecutar OpenJML con la opción ESC (Extended Static Checking) sobre los archivos generados, la herramienta muestra 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 pos condición 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:

```
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);
}
```

**Figura 10:** Restricciones OCL para el modelo de la biblioteca

```
context Empleado::
aumentarSueldo(aumento:ecore::EInt):OclVo
id
pre : aumento >0
```

De esta forma, se deberían garantizar el invariante y las pos condiciones de la operación. Volvemos a ejecutar la herramienta de traducción, y se genera su respectiva precondición JML. Luego de realizar otra vez la traducción, 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.

### *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. Como se puede ver en el código, además de pos condiciones 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 contiene el código exhibido en la figura 11.

Para realizar la verificación en tiempo de ejecución, necesitamos crear, en alguna de las clases Java generadas en la traducción, un método main que cree algunas instancias de prueba y ejecute los métodos necesarios. En este ejemplo se le agregó un método main a la clase Biblioteca, como se ve en la Figura 12.

Luego debemos compilar las clases necesarias para su ejecución. Esto se hace mediante la opción RAC (runtime assertion checking). 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.

```
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 ();
}
```

**Figura 11:** archivo de la clase Socio con especificación JML

```
package Biblioteca;

import java.util.*;

public class Biblioteca {

 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 Informa
  Autor a = new Autor("Carolina","Actis","argentina",new Date(
  Libro l = new Libro("OCL2JML","UNLP",a,TipoDeLibro.Tecnico,n
  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 12.** main de prueba para la clase Biblioteca

Luego de compilar los archivos, se ejecuta el método main. 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 modificamos 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 13.

Volvemos a compilar las clases, y las ejecutamos una vez más. 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.

```
public static void main (String[] args){
 Biblioteca b = new Biblioteca("50 y 120","Biblioteca Inf
 Autor a = new Autor("Carolina","Actis","argentina",new D
 Libro l = new Libro("OCL2JML","UNLP",a,TipoDeLibro.Tecni
 Socio s = new Socio("Juan","Perez","2215678910",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 13**: Método main modificado de la clase Biblioteca

## 7. Trabajos relacionados

En esta sección se presentan distintos trabajos relacionados a la presente propuesta, algunos de los cuales sirvieron como base para esta.

En [25] se propone traducir restricciones OCL a código JML, de forma que pueda ser ejecutado y evaluado en tiempo de ejecución. Ya en [26] 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 [30] se desarrolló un plugin de Eclipse que implementa la traducción de OCL a JML, basándose lo propuesto en [25]. Se implementó una biblioteca de colecciones OCL como la planteada en dicho trabajo. La traducción en sí se realizó utilizando la herramienta MOFScript [31]. Se utilizó en este trabajo también OpenJML para realizar verificación estática de los resultados de la traducción.

En [32] se avanza sobre el trabajo hecho en [25], 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 [33]. 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.

Por otro lado, en [34] 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.

### *Aportes de nuestra propuesta respecto a las descritas*

En este trabajo se implementó como plugin de Eclipse la traducción propuesta en [25] y [26], sumándole decisiones propias de implementación para las diferentes expresiones OCL, logrando mayor expresividad y eficiencia. 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 nuevo metamodelo JML y su traducción completa a código, ya que no existían versiones completas del mismo. 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 [30]. Todo el código implementado en este proyecto es abierto y de uso libre.

## 8. Conclusiones y Trabajos Futuros

Se desarrolló una herramienta basada en Eclipse que permite, dado un modelo UML incluyendo 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 varios casos de estudio, que cubren una amplia variedad de expresiones OCL. Se usó la herramienta OpenJML para realizar la verificación estática y en tiempo de ejecución de las mismas. El código completo de la herramienta, así como la descripción detallada de los experimentos están contenidos en la tesis de la autora [35].

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 JML, para poder verificar en tiempo de ejecución que las restricciones JML sean cumplidas.

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

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 [32] y [33]. 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 [34]. Se podría también hacer que la herramienta integre a OpenJML de forma más directa, para que la verificación de las especificaciones traducidas resulte más sencilla.

## Referencias

[1] M. Brambilla, J. Cabot and M. Wimmer. “Model-Driven Software Engineering in Practice”. Morgan&Claypool Publishers ISBN: 9781608458820. 2012.

[2] Claudia Pons, R. Giandini y G. Perez. Desarrollo de Software Dirigido por Modelos., McGraw-HIll Educación y Edulp, 2010.
[3] MDA [En línea]. Available: http://www.omg.org/mda. [Último acceso: 2017].
[4] OMG [En línea]. Available: http://www.omg.org. [Último acceso: 2017].
[5] UML. The OMG Unified Modeling Language Specification V. 2.5.1..https://www.omg.org/spec/UML/2.5.1// 2017.
[6] A. K. Jos Warmer, The Object Constraint Language: Getting Your Models Ready for MDA, 2ª ed., Addison Wesley.
[7] G.Leavens,A.Baker, C.Ruby. «The Java Modeling Language (JML),» [En línea]. Available: http://www.eecs.ucf.edu/~leavens/JML/index.shtml. [Último acceso: 2017].
[8] B.Meyer, «Applying design by contract,» de Computer, 25(10), 1992, pp. 40-51.
[9] Ken Arnold, James Gosling, David Holmes. The Java Programming Language Third Edition, Addison-Wesley, 2000.
[10] M. Huisman, «Reasoning about JAVA programs in higher order logic with PVS and Isabelle,» IPA dissertation series, nº 2001-03, 2001.
[11] Y. C. y. G. Leavens, «A runtime assertion checker for the Java Modeling Language (JML),» de Proceedings of International Conference on Software Engineering Research and Practice, 2002, pp. 322-328.
[12] Y. Cheon., G. Leavens, «A Simple and Practical Approach to Unit Testing: The JML and JUnit Way,» de ECOOP 2002 -- Object-Oriented Programming, 16th European Conference.
[13] Lilian BurdyYoonsik CheonDavid R. CokMichael D. ErnstJoseph R. KiniryGary T. Leavens K. Rustan M. Leino, An overview of JML tools and applications, International Journal on Software Tools for Technology Transfer. 2005, Volume 7, Issue 3, pp 212–232.
[14] EMF [En línea]. Available: https://eclipse.org/modeling/emf/. [Último acceso: 2017].
[15] MOF [En línea]. Available: http://www.omg.org/mof/. [Último acceso: 2017].
[16] ATL Language. [En línea]. Available: http://www.eclipse.org/atl/. [Último acceso: 2017].
[17] Acceleo [En línea]. Available: https://www.eclipse.org/acceleo/. [Último acceso: 2017].
[18] MOFM2T [En línea]. Available: http://www.omg.org/spec/MOFM2T/1.0/. [Último acceso: 2017].
[19] OCL [En línea]. Available: http://www.omg.org/spec/OCL/2.4/. [Último acceso: 2017].
[20] OPENJML [En línea]. Available: http://www.openjml.org/. [Último acceso: 2017].
[21] A. S. C. T. Clark Barrett, The SMT-LIB Standard Version 2.0, 2012.
[22] Z3 [En línea]. Available: https://github.com/Z3Prover/z3. [Último acceso: 2017].
[23] [En línea]. Available: http://cvc4.cs.stanford.edu/web/. [Último acceso: 2017].
[24] [En línea]. Available: http://yices.csl.sri.com/. [Último acceso: 2017].
[25] Carmen Avila, G. Flores. J. Y. Cheon., «A Library-Based Approach to Translating OCL Constraints to JML Assertions for Runtime Checking,» Conference: Proceedings of the 2008 International Conference on Software Engineering Research & Practice, SERP 2008, July 14-17, 2008,, USA.
[26] A. Hamie, «Translating the Object Constraint Language,» SAC '04 Proceedings of the 2004 ACM symposium on Applied computing.
[27] [En línea]. Available: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html. [Último acceso: 2017].
[28] MoDisco [En línea]. Available: https://www.eclipse.org/MoDisco/. [Último acceso: 2017].
[29] «JML Reference Manual: Grammar Summary,» [En línea]. Available: http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_22.html#SEC224. [Último acceso: 2017].
[30] María Jose Dias Molina, y Diego Dodero Mena «Desarrollo de una herramienta para derivación automática de especificaciones OCL a JML ´,»Tesis de Licenciatura en Informática, UNLP. 2011.
[31] «MOFScript Home Page,» [En línea]. Available: https://www.eclipse.org/gmt/mofscript/. [Último acceso: 2017].
[32] A. Hamie, «Pattern-based Mapping of OCL Specifications to JML Contracts,» 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), Publisher: IEEE. 2015..
[33] J. Ackermann, K. Turowski. «A Library of OCL Specification Patterns for Behavioral Specification of Software Components,» de Proceedings of the 18th international conference on Advanced Information Systems Engineering, 2006.
[34] K. Hiroaki H.Shimba, K. Hanada, K. Okano, S. Kusumoto «Bidirectional Translation between OCL and JML for Round-trip Engineering,» de 2013 20th Asia-Pacific Software Engineering Conference, IEEE Xplore. 2013.
[35] Actis, Carolina. "Verificación de modelos Independientes de la plataforma: un caso de estudio", Noviembre 2017. Facultad de Informática, UNLP. [En línea]. Available: http://sedici.unlp.edu.ar/handle/10915/67013.

CONAIISI 2018
6to Congreso Nacional de Ingeniería Informática - Sistema de Información

# Design Sprint como marco de trabajo para la gestión de proyectos ágiles en equipos interdisciplinarios

Cristhian A. Boujon[1], Gilda R. Romero[1, 2], Sergio Lapertosa[1]
[1] *Facultad de Ingeniería y Tecnología – Universidad de la Cuenca del Plata*
*Lavalle, 50 – Corrientes, Capital.*
[2] *Facultad Regional Resistencia – Universidad Tecnológica Nacional*
*French, 414 – Resistencia, Chaco.*
*{boujoncristhianalberto_cen; romerogilda_cen; lapertosasergio_cen}@ucp.edu.ar*

## Resumen

*La evolución en la Ingeniería de Software propone varias metodologías de las llamadas ágiles para la concreción de sistemas, entre ellas se encuentra la denominada Design Sprint. Esta metodología se ha popularizado en los últimos años principalmente a que propone un enfoque de doble diamante con procesos de convergencia y divergencia de ideas en pos de solucionar un problema que tienen en el producto enunciado en un Challenge de diseño. Este trabajo presenta la implementación de la metodología Design Sprint como marco de trabajo en una experiencia entre las carreras de Ingeniería en Alimentos e Ingeniería en Sistemas de Información de la Universidad de la Cuenca del Plata. La experiencia se focalizó en desarrollar las competencias ingenieriles referidas tanto a las competencias tecnológicas, como a las sociales, políticas y actitudinales, principalmente a través de la conformación de equipos de trabajos interdisciplinarios y auto gestionados. Aquí se describe cómo se llevó a cabo la actividad, las conclusiones preliminares y las futuras acciones a desarrollar para continuar formando a los trabajadores de la Industria 4.0.*

## 1. Introducción

Las actividades significativas hacen posible que el alumno, bajo la dirección del docente, construya sus conocimientos y pueda aplicarlo a nuevas situaciones. Son un conjunto de estrategias que el docente debe aplicar a fin de lograr que los estudiantes adquieran un aprendizaje, también, significativo. Por otro lado, el efecto Pigmalión [1] trata de sobre cómo influyen las expectativas que una persona tiene sobre otra. En la pedagogía este concepto es especialmente importante ya que las expectativas positivas que el docente deposita en sus alumnos potencian su rendimiento. En suma, es menester "*organizar una experiencia educativa con la finalidad de generar situaciones para que los docentes participantes se modifiquen a partir de la interacción con ellos mismos y con otros, apropiándose de saberes nuevos, desarrollando disposiciones y construyendo capacidades en acción*" [2], para contribuir a la formación de noveles profesionales con estas nuevas capacidades que hacen al denominado Talento 4.0. Entendiendo por Talento 4.0 a la aptitud o competencia intelectual en áreas STEM (Ciencia, Tecnología, Ingeniería y Matemáticas, por su sigla en inglés) e incluye también la actitud digital, pasión por el cambio, aprendizaje autónomo y resiliencia.

A partir del año 2016, al iniciarse el dictado de la Carrera Ingeniería en Sistemas de Información (ISI) por primera vez en la Universidad, se decidió abordar el desarrollo de las cátedras con una serie de actividades [3] tendientes a dar respuesta a lo requerido en la denominada "Transformación Digital" en términos de la formación del profesional de informática, tomando como marco el Modelo Pedagógico de la UCP y las definiciones de formación por competencias del Consejo Federal de Decanos de Ingeniería (CONFEDI), plasmadas en la nueva propuesta de estándares de segunda generación para las carreras de Ingeniería [5]. Sabiendo que la Ingeniería es una profesión esencialmente creativa, donde confluyen los saberes como productos de procesos de investigación e innovación en ciencia y tecnología, evolucionando en la búsqueda de soluciones a problemas complejos dentro de las organizaciones, vale destacar la importancia de este proyecto en la ejecución de la carrera de acuerdo al contexto en el que se encuentra inserta y el perfil que se propone desarrollar en sus egresados.

Considerando estas premisas, durante el ciclo 2018 se ha llevado a cabo un trabajo interdisciplinar con una de las cátedras de la carrera de Ingeniería en Alimentos (IA). La experiencia fue desarrollada definiendo un proyecto Intercátedra desde la cátedra de "Ingeniería de Software I" de ISI y la cátedra de "Procesos Industriales II" de IA. El proyecto constaba en realizar un prototipo