Derivación automática de especificaciones formales ejecutables a partir de modelos UML-OCL

Alumnos
María José Dias Molina
Diego Matías Dodero Mena
Facultad de Informática, UNLP

Orientadora
Claudia Pons (UNLP-UAI)

Derivación automática de especificaciones formales ejecutables a partir de modelos UML-OCL

María José Dias Molina y Diego Matías Dodero Mena
Facultad de Informática, UNLP

Resumen

En este trabajo realizamos una integración entre los lenguajes de especificación de restricciones JML y OCL, permitiendo aprovechar los beneficios de la verificación formal y el análisis estático de código que proveen las herramientas para ambos lenguajes. Con este fin hemos implementado una herramienta de software que permite utilizar OCL para expresar propiedades sobre el modelo UML del sistema y luego derivar automáticamente el código JML para realizar la verificación de las propiedades del código en ejecución. De esta forma, las capacidades de ambos lenguajes se ven recíprocamente potenciadas.
Keywords: OCL, JML, Comparación OCL y JML, Eclipse, MOFScript, Verificación de programas, ESC/Java2.

1. Introducción

La verificación de software es una parte de la ingeniería de software cuyo objetivo es asegurar que el software satisface los requerimientos esperados. Existen dos enfoques principales, el análisis dinámico o testing y el análisis estático o verificación formal. En el análisis dinámico se realiza el análisis sobre el resultado de la ejecución de programas, es preciso pero caracteriza algunas ejecuciones. En el análisis estático, en cambio, se realiza el análisis sin ejecutar el programa, caracteriza todas las ejecuciones pero suele ser conservador.

El análisis estático tiene límites, por reducción al halting problem, se puede demostrar que encontrar todos los posibles errores en tiempo de ejecución es indecidible [1]. No hay ningún método mecánico que pueda determinar completamente si un programa produce errores en ejecución. Sin embargo se puede intentar realizar aproximaciones, siendo las técnicas más importantes:

- **Model checking**: verifica propiedades en sistemas con estados finitos o que pueden ser reducidos a estados finitos.
- **Dataflow analysis**: busca errores mecánicos difíciles de encontrar vía testing o inspecciones.
- **Bug finding**: busca patrones comunes de errores y chequea el uso de buenas prácticas.
- **Interpretación abstracta**: modela el efecto que cada sentencia tiene en el estado de una máquina abstracta. La interpretación abstracta es *consistente* pero no *completa*.
- **Métodos deductivos**: utilizan aserciones basadas en lógica de Hoare y generalmente son incompletos.

La sofisticación del análisis varía desde aquellas que consideran sólo instrucciones y declaraciones, hasta las que revisan el programa completo.

Para poner en práctica la verificación formal de un programa es necesario contar con una especificación formal de su comportamiento, expresada en algún lenguaje apropiado. Actualmente existen varios lenguajes formales estandarizados en la industria del desarrollo de software tales como Z y B [2]. Utilizando estos lenguajes formales se logra introducir rigurosidad en la etapa temprana del diseño del software, sin embargo, estos lenguajes basados en la lógica y la matemática suelen resultar

poco intuitivos para los desarrolladores y además requieren el uso de herramientas de verificación y prueba de teoremas muy alejadas del ambiente de desarrollo de software ordinario.

Por otra parte, existen lenguajes formales ligados a lenguajes de diseño o de programación específicos, tales como Java Modeling Language (JML) [3], que es utilizado en el desarrollo de aplicaciones Java y Object Constraint Language (OCL) [4] vinculado a diseños orientados a objetos. En estos casos la sintaxis del lenguaje formal se mimetiza con el lenguaje de diseño y programación, resultando mucho más amigable para los desarrolladores. Además, las herramientas de verificación están integradas directamente en el ambiente de desarrollo.

El objetivo de nuestro trabajo es realizar una integración entre JML y OCL, permitiendo aprovechar los beneficios de la verificación formal y el análisis estático de código que proveen las herramientas para ambos lenguajes. Con este fin hemos implementado una herramienta de software que permite utilizar OCL para expresar propiedades sobre el modelo UML del sistema y luego derivar automáticamente el código JML para realizar la verificación de las propiedades del código en ejecución. De esta forma, las capacidades de ambos lenguajes se ven recíprocamente potenciadas.

Este artículo está organizado de la siguiente forma. En la sección 2 describimos brevemente a los lenguajes de especificación OCL y JML, efectuando una comparación entre sus sintaxis, semánticas y capacidades expresivas. En la sección 3 analizamos las principales herramientas de verificación formal disponibles para el lenguaje JML. En la sección 4 definimos un algoritmo de traducción entre ambos lenguajes y presentamos el diseño e implementación de la herramienta que realiza la traducción automática. La sección 5 contiene un caso de estudio. En la sección 6 describimos trabajos relacionados. En la sección 7 se indican posibles líneas que pueden seguirse como continuación de este trabajo y finalmente en la sección 8 presentamos las conclusiones de este trabajo.

2. OCL y JML

2.1. El lenguaje OCL como complemento a UML

Unified Modeling Language (UML) es un poderoso lenguaje para diseñar y documentar sistemas de software. Existen muchas herramientas que asisten la creación y mantenimiento de los documentos UML. Las más avanzadas incluyen ciertas características que permiten la traducción de modelos UML a código ejecutable, y viceversa [5, 6]. OCL es un lenguaje de especificación que fue diseñado como un complemento para UML. Los diagramas UML no son lo suficientemente expresivos y no logran abarcar todos los aspectos relevantes de una especificación. En particular UML no brinda elementos para describir restricciones adicionales sobre los objetos del modelo. Estas restricciones son a veces descriptas en lenguaje natural, pero esto puede resultar en ambigüedades. Para escribir restricciones no ambiguas se han desarrollado los lenguajes formales (utilizados por personas con un gran conocimiento matemático, pero difíciles de utilizar por los desarrolladores de software ordinarios). OCL ha sido desarrollado para solucionar la deficiencia descripta.

OCL es un lenguaje funcional puro por lo que se garantiza que toda expresión es libre de efectos laterales; es decir cuando una expresión OCL es evaluada simplemente retorna un valor, sin modificar nada en el modelo. OCL no es un lenguaje de programación, no es posible escribir lógica de programas o flujo de control en OCL. OCL es un lenguaje tipado. Para ser bien formada, una expresión debe concordar con reglas de tipado del lenguaje. Cada clase definida en un modelo UML representa un tipo distinto en OCL. Además, OCL incluye un conjunto de tipos suplementarios predefinidos. El lenguaje OCL es independiente del lenguaje de programación que se utilice.

OCL puede utilizarse para diferentes propósitos: como lenguaje de consulta, para especificar invariantes en clases y tipos, en un modelo de clases para especificar invariantes de tipos, para definir estereotipos, para describir pre y post condiciones en operaciones y métodos, para describir guardas, para especificar reglas de derivación para una propiedad, para especificar restricciones en operaciones, para especificar los valores iniciales de las propiedades, etc.

2.2. JML

JML es un lenguaje que permite especificar el comportamiento e interfaz de clases y métodos Java. JML provee una semántica para describir de manera formal el comportamiento de un módulo de Java, evitando así la ambigüedad con respecto a las intenciones del diseñador del módulo. JML ha heredado ideas de Eiffel, Larch y del cálculo de refinamiento, con la meta de proveer una semántica rigurosa sin perder su accesibilidad a los programadores Java ordinarios. Hay varias herramientas que se sirven de las especificaciones de comportamiento de JML. JML utiliza pre y post condiciones e invariantes de la lógica de Hoare. Permite aplicar la metodología de diseño por contrato cuya idea principal es que las clases y sus clientes tengan un contrato entre ellos [7]. El contrato se establece de la siguiente forma, el cliente debe garantizar ciertas condiciones antes de llamar a un método definido por la clase, y la clase debe garantizar ciertas propiedades luego del llamado [3]. Una de las principales ventajas de JML es su similitud al lenguaje Java, lo que facilita su aceptación por parte de los programadores. Otra de sus ventajas es la existencia de herramientas que permiten utilizar las anotaciones JML para realizar análisis formal del código.

2.3. Comparación

Tanto OCL como JML son lenguajes de especificación, pero tienen diferencias en sus objetivos y características. Para evaluar la factibilidad de la traducción debemos analizar la correspondencia entre las construcciones de ambos lenguajes, tanto a nivel sintáctico como semántico, poniendo énfasis en el último aspecto. Podemos sintetizar a grandes rasgos las diferencias entre OCL y JML mediante la siguiente lista de ventajas y desventajas:

Ventajas de OCL:

1. Tiene un nivel más alto de abstracción. Un diagrama UML puede tener anotaciones OCL antes del desarrollo de código.
2. No está relacionado con un lenguaje determinado.
3. Está estandarizado por el OMG.

Desventajas de OCL:

1. No favorece la verificación en tiempo de ejecución.

Ventajas de JML:

1. Es muy cercano al código Java, lo que facilita su uso por parte de programadores y desarrolladores.
2. Ofrece una gran variedad de conceptos en el nivel de implementación, como manejo de excepciones, cláusulas de asignación e invariantes de ciclos.
3. Existen herramientas de verificación que utilizan anotaciones JML, tanto para análisis estático de código como comprobación en tiempo de ejecución.

Desventajas de JML:

1. Está asociado al lenguaje Java, no es general como OCL.
2. No está estandarizado, la mayoría de las herramientas actuales no implementan la semántica de JML.

2.4. Diferencias semánticas

En esta sección destacamos las principales diferencias semánticas entre OCL y JML, las cuales deben tenerse en cuenta cuidadosamente al realizar la integración de ambos lenguajes.

2.4.1. Estado previo a la ejecución

Tanto OCL como JML permiten que en las postcondiciones se pueda referir a estados previos. En OCL se utiliza el modificador @**pre**, mientras que en JML se utiliza la cláusula **\old**. La diferencia principal es que la primer construcción puede utilizarse con símbolos individuales, mientras que la segunda sólo puede aplicarse a expresiones, e.g. **a@pre.b**, **a.b@pre** son expresiones OCL válidas mientras que en JML sólo lo es **\old(a.b)**.

Un problema de la construcción **\old** ocurre cuando se utilizan arreglos. Supongamos que se quiere establecer en una postcondición de un método m que manipula un array `a[]` y una propiedad idx que el valor de $a[0]$ es igual al valor anterior del arreglo en la posición idx. La expresión **\old**$(a[idx])$ no funcionaría, ya que el valor de idx en el estado previo sería utilizado, por lo que debemos recurrir a la expresión siguiente:

```
(\forall int x; x == idx; \old(a[x]) == a[0]);
```

2.4.2. Cláusula modifies

JML provee frame conditions, a través de las cláusulas **assignable** y **modifies**. Estas cláusulas permiten especificar las variables que un método puede modificar. OCL no soporta esta construcción [8, 9].

2.4.3. Rango de cuantificadores

En JML los cuantificadores se extienden sobre todos los elementos de un tipo determinado y no sólo sobre todos los elementos creados o alocados.

2.4.4. Enteros

JML utiliza la semántica de Java para los números enteros. Los números enteros en Java tienen un rango limitado (en una implementación de 32 bits, un rango de $-2^{31}..2^{31}-1$), y los operadores sobre estos tipos obedecen las reglas de la aritmética modular. Uno de los principales problemas es que al escribir especificaciones los programadores tienen el modelo mental de los enteros matemáticos, e ignoran la finitud de los tipos numéricos, lo que ocasiona errores en las especificaciones [8]. En [10] se propone una extensión a JML llamada JMLa que incluye un tipo primitivo **\bigint** que representa enteros de precisión arbitraria, por ejemplo los enteros matemáticos.

2.4.5. Igualdad e identidad

En Java existen dos maneras de comparar objetos: por identidad (por ejemplo a nivel de referencia) y por igualdad (por ejemplo a nivel de objetos). Para tipos primitivos (por ejemplo `int`, `double`, etc.) ambas formas son equivalentes y se realiza utilizando el operador ==. Para comparar objetos, la comparación por identidad utiliza también el operador == y comprueba que la dirección en memoria de los objetos es la misma, la comparación por igualdad utiliza el método `equals(o : Object) : boolean` definido en la clase `Object`. En OCL sólo se puede comparar los objetos por igualdad [11].

2.4.6. Arreglos

OCL no ofrece una construcción primitiva para modelar los arreglos de Java. El tipo de colección `Sequence` no es útil, ya que no tiene en cuenta que los arreglos Java son objetos y declara operaciones, por ejemplo *union* o *append* que no tienen sentido en arreglos.

2.4.7. Excepciones

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

2.4.8. Niveles de visibilidad

Java permite especificar cuatro niveles de visibilidad: **public**, **package** (default), **protected** y **private**. En OCL la visibilidad no se tiene en cuenta al momento de realizar las navegaciones en las restricciones.

3. Herramientas de verificación para JML

Las principales herramientas de verificación disponibles para el lenguaje JML pueden clasificarse en:

- **Runtime checkers**: agregan chequeos en tiempo de ejecución. Ejemplo: jmlrac.
- **Static checkers**: tienen mejor cobertura, encuentran errores y violaciones de contratos. Ejemplo: ESC/Java2.
- **Buscadores de modelos**: buscan contraejemplos.
- **Verificadores de programas**: verificación total, son asistidos por computadora pero no automáticos. Ejemplos: KeY, Loop, Jack.

En 1999 se definió el lenguaje JML e inmediatamente después comenzó el desarrollo de herramientas de verificación basadas en el nuevo lenguaje [12]. Las funcionalidades básicas que debe proveer cualquier herramienta JML son el análisis sintáctico y el chequeo de tipos.

En esta sección mostramos un panorama general de las herramientas de verificación existentes para JML. De esta forma, conocemos las herramientas existentes, sus ventajas y limitaciones. Las herramientas principales desarrolladas para JML son las siguientes:

1. El compilador JML (`jmlc`) es una extensión de un compilador Java y compila programas Java con especificaciones JML en bytecodes Java. El código compilado incluye instrucciones de chequeo en tiempo de ejecución que verifican especificaciones JML tales como precondiciones, postcondiciones e invariantes.
2. La herramienta para realizar testing de unidad (`jmlunit`) combina el compilador Java con JUnit, una herramienta de testing de unidad para Java. La herramienta libera al programador de escribir código que decida si un test tiene éxito o falla, en lugar de escribir tales pruebas, `jmlunit` utiliza las especificaciones JML procesadas por `jmlc` para decidir si el código testeado funciona correctamente.
3. El generador de documentación `jmldoc` produce código HTML conteniendo tanto comentarios Javadoc como especificaciones JML.
4. La herramienta para realizar chequeo estático (`escjava2`) puede encontrar posibles errores en un programa Java. Es muy útil para encontrar potenciales referencias a **null** y acceso a índices de arreglos fuera del rango. Utiliza anotaciones JML para mejorar la detección de errores y chequea las especificaciones JML.
5. El chequeador de tipos `jml` es una herramienta para chequear las especificaciones JML, es más rápida que `jmlc` ya que no compila el código.

Las primeras herramientas desarrolladas para JML quedaron rápidamente obsoletas, debido a la evolución, tanto del lenguaje Java como de JML. Fue necesario desarrollar nuevas herramientas. En particular las siguientes:

- JML4c es un compilador de JML construido sobre la plataforma Eclipse JDT. Traduce un subconjunto significativo de especificaciones JML en chequeos en tiempo de ejecución. Utilizando JML4c, se pueden verificar clases Java 5 anotadas con especificaciones JML. Entre sus características principales podemos mencionar: velocidad de compilación mejorada, soporte para características de Java 5 (tales como tipos genéricos y ciclos for mejorados), soporte para clases internas, mensajes de error mejorados.

- OpenJML es una re-implementación, basada en OpenJDK, de herramientas tales como las herramientas JML2 para JML sobre Java 1.4 y ESC/Java y ESC/Java2 para verificación estática. OpenJML extiende OpenJDK para crear herramientas JML para Java 7.
- KeY es una herramienta que provee facilidades para especificación formal y verificación de programas [13]. Originalmente fue diseñada para utilizar OCL como lenguaje de programación, pero luego fue modificada para utilizar JML [14]. El proyecto utiliza su propio parser tanto para Java como para JML, creando obligaciones de prueba (proof obligations), estas condiciones de verificación (verification conditions) son luego demostradas interactivamente utilizando los sistemas de prueba Coq y Why.
- JACK es una herramienta desarrollada actualmente en el INRIA. El objetivo de JACK es proveer un entorno para la verificación de programas Java y JavaCard. JACK implementa un cálculo de precondiciones más débiles totalmente automatizado para generar obligaciones de prueba código Java con anotaciones JML.
- El proyecto LOOP comenzó en la universidad de Nijmegen como una exploración de la semántica de los lenguajes orientados a objetos en general y Java en particular. Luego evolucionó para investigar la verificación de código Java con anotaciones JML [15].

En nuestro trabajo hemos utilizado en particular OpenJML y ESC/Java2, porque realizan la verificación sobre las especificaciones con mínima intervención del usuario.

4. Traducción automática entre lenguajes

OCL y JML son muy similares sintácticamente, una gran parte del mapeo de OCL a JML es casi directo, en particular los operadores y expresiones. Los aspectos más difíciles de mapear son las colecciones y las operaciones sobre colecciones. El enfoque utilizado para la traducción de componentes básicos está basada en [16, 17] y la de colecciones en [18].

4.1. Tipos simples

El mapeo de tipos simples es directo, con la salvedad de que en Java los tipos numéricos de punto flotante no conforman un superconjunto de los enteros.

OCL	**JML**
`Boolean`	`boolean`
`Integer`	`int`
`Real`	`double`
`Char`	`char`
`String`	`String`

Cuadro 1: Traducción de tipos de datos simples

4.2. Operadores y expresiones

Los operadores matemáticos se traducen directamente, los operadores lógicos se traducen casi directamente. El operador **implies** no tiene equivalencia en el lenguaje Java, pero fue implementado en JML. JML (no Java) posee un operador bicondicional <==> que no existe en OCL.

4.3. Pseudovariables

Las pseudovariables y las referencias al estado de las variables son mapeadas casi directamente a JML.

OCL	**JML**
not e	!e
e_1 *and* e_2	e_1 && e_2
e_1 *or* e_2	e_1 || e_2
e_1 *implies* e_2	e_1 ==> e_2 e_2 <== e_1
	e_1 <==> e_2
e_1 = e_2	e_1 == e_2 (tipos primitivos) $e_1.equals(e_2)$ (objetos)
e_1 <> e_2	e_1 != e_2
if e_0 *then* e_1 *else* e_2 *endif*	(e_0? e_1 : e_2)

Cuadro 2: Traducción de operaciones básicas

OCL	**JML**
self	*this*
result	\result
variable.OCLIsNew()	\fresh(variable)
variable@pre	\old(variable)
variable.OCLIsUndefined()	variable == null
variable.OCLIsTypeOf(`Class`)	\typeof(variable) == \type(`Class`)
variable.OCLIsKindOf(`Class`)	\typeof(variable) <: \type(`Class`)

Cuadro 3: Traducción de pseudovariables

4.4. Cuantificadores

Los cuantificadores son mapeados directamente a JML. Debemos tener en cuenta las consideraciones señaladas en la sección 2.4.3, en JML el rango de los cuantificadores son todos los objetos del tipo.

OCL	**JML**
coleccion->forAll(v : Tipo | expresion con v)	(\forall Tipo v; coleccion.includes(v); expresion con v)
coleccion->exists(v : Tipo | expresion con v)	(\exists Tipo v; coleccion.includes(v); expresion con v)

Cuadro 4: Traducción de cuantificadores

4.5. Construcciones no traducidas

Ciertas construcciones de OCL no fueron traducidas para la realización de este trabajo

- Iteradores *collect*, *iterate*, *sortedBy*, *any*: Esos métodos requieren expresiones como parámetro.
- Método *allInstances*: Java no permite como Smalltalk obtener fácilmente las instancias de una clase. En Java el usuario debe programar éstos métodos para poder recuperarlas.Los diseñadores de OCL de todas formas recomiendan no utilizar la operación.
- Tuplas: En Java se pueden implementar como un Map de String a Object, pero su utilización se vuelve engorrosa perdiéndose la facilidad de uso de tuplas.
- Expresión de invocación de mensaje en una postcondición.

4.6. La herramienta de traducción

La herramienta tiene cuatro funcionalidades principales:

1. **Parse OCL:** realiza un análisis sintáctico y semántico sobre un archivo con restricciones OCL.
2. **Translate UML model:** Dado un modelo UML genera el código Java anotado con especificaciones JML.
3. **Create JML specification:** Dado un modelo UML crea sólo la especificación JML.
4. **Create Java model:** Dado un modelo UML crea sólo el código Java.

4.7. Estructura general

La herramienta está formada por tres componentes principales que se describen a continuación.

- **Analizador del modelo:** Dado un modelo UML en formato XMI y un archivo con restricciones OCL, realiza el análisis sintáctico del archivo y el análisis semántico contra el modelo UML. Si no existen errores en el análisis, su salida es la lista de restricciones OCL.
- **Traductor JML:** Dada una lista de restricciones OCL genera un modelo equivalente al modelo de entrada anotado con especificaciones JML.
- **Transformador M2Text:** Dado el modelo anotado con especificaciones JML genera el modelo Java anotado con especificaciones JML.

4.8. Plataforma de implementación de la herramienta

Eclipse es una plataforma de software de código abierto independiente de una plataforma para desarrollar lo que el proyecto llama "Aplicaciones de Cliente Enriquecido", opuesto a las aplicaciones "Cliente liviano" basadas en navegadores. Esta plataforma, típicamente ha sido usada para desarrollar entornos integrados de desarrollo (del inglés IDE, Integrated development environment). [19]

Eclipse fue diseñada para desarrollar herramientas con el fin de construir aplicaciones web y de escritorio. La plataforma no provee funcionalidad por sí misma, sino que permite el rápido desarrollo de features integradas basadas en un modelo de plugins. Esto significa que cuando la plataforma es cargada, al usuario se le presenta un ambiente de desarrollo integrado, compuesto por los plugins que hay habilitados.

La Plataforma Eclipse está diseñada y construida para cumplir con los siguientes requerimientos:

- Soportar la construcción de una variedad de herramientas para el desarrollo de aplicaciones.
- Soportar un irrestricto conjunto de proveedores de herramientas, incluyendo vendedores de software independientes.
- Soportar herramientas para manipular tipos de contenido arbitrario (por ejemplo HTML, Java, C, JSP, EJB, XML, UML, GIF, etc).
- Permitir una fácil integración de las herramientas entre sí, a través de los diferentes tipos de contenidos y sus proveedores.
- Soportar el desarrollo de aplicaciones basadas o no, en interfaz gráfica de usuario (en inglés Graphical User Interface, GUI y non-GUI-based).
- Correr en una gran cantidad de sistemas operativos.

El principal objetivo es el de proveer facilidades a los proveedores de herramientas para desarrollar las mismas con mecanismos de uso y reglas para seguir. Estos mecanismos son provistos por una API de interfaces bien definida, clases y métodos.

4.9. Tecnología utilizada en la traducción

MOFScript es un lenguaje de transformación de modelo a texto presentado por la OMG. Este lenguaje presta particular atención a la manipulación de strings y de texto y al control e impresión de salida de archivos. Está construido por reglas y una transformación consiste en un conjunto de reglas. [20]

No sólo es un estándar, sino que además se basa en otros estándares de la OMG: es QVT compatible y Meta Object Facility (MOF) compatible con el modelo de entrada (el target siempre es texto). Para las reglas de transformación, MOFScript define un metamodelo de entrada para el source y sobre el cual operarán las reglas. El target es generalmente un archivo de texto (o salida de texto por pantalla).

Las transformaciones son siempre unidireccionales y no es posible definir pre y post condiciones para ellas. La separación sintáctica resulta clara por la misma definición de reglas, lo que hace a la legibilidad del lenguaje. No provee estructuras intermedias, pero sí parametrización necesaria para la invocación de reglas.

En la figura 1 se muestra la arquitectura de MOFScript.

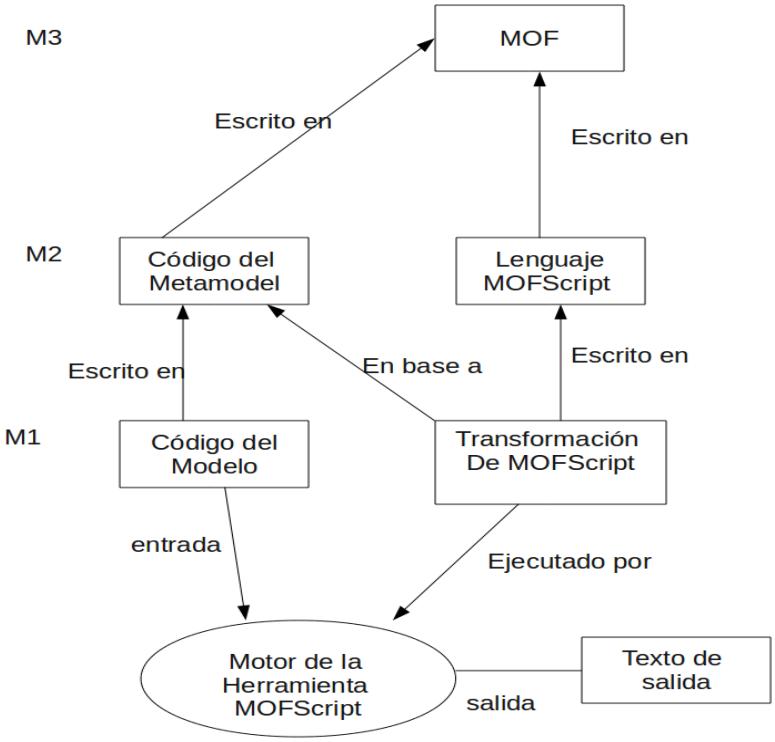

Figura 1: Arquitectura de cuatro capas de MOFScript

5. Caso de estudio

Para mostrar el funcionamiento de las herramientas de verificación, se utilizará el ejemplo de Cuenta Bancaria extraído de [16], ya que estas herramientas no soportan tipos complejos. La Figura

2 muestra el ejemplo a utilizar.

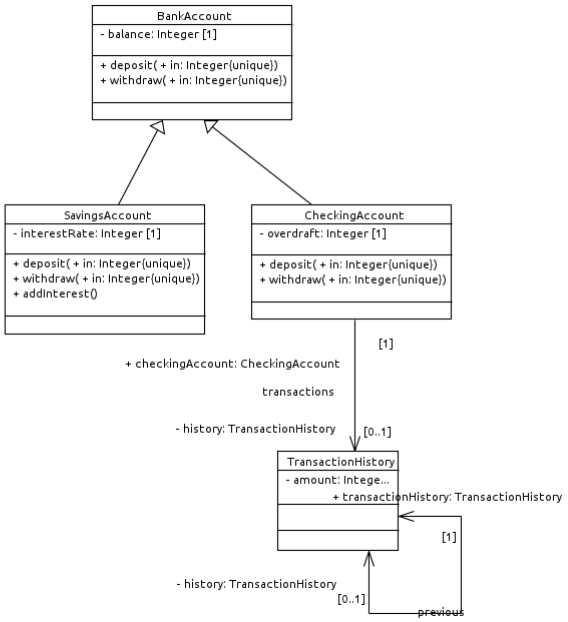

Figura 2: Ejemplo de cuenta bancaria

Para el desarrollo de este ejemplo, vamos a suponer que ya tenemos creado un workspace **jmltest** y hemos generado dentro del package **model** los archivos *.java* del modelo y sus especificaciones *.jml*. También debemos tener el modelo .uml y el archivo de restricciones .ocl que se muestra en la Figura 3

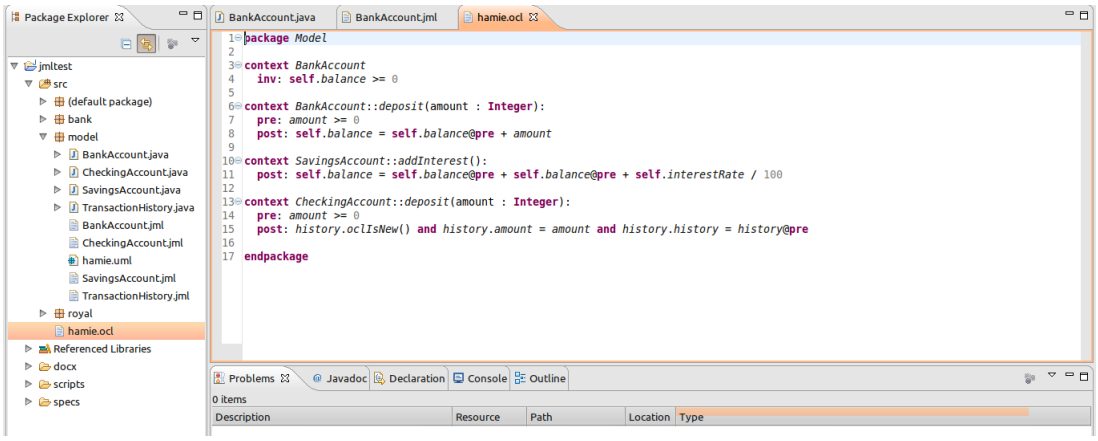

Figura 3: OCL utilizado

La clase sobre la que vamos a trabajar es BankAccount, mostrada en la Figura 4.

Y su especificación JML es la mostrada en la Figura 5

5.1. OpenJML

Para poder correr el programa **OpenJML** debemos tener un script para facilitar su uso, con los paths que se encuentran dentro de él, bien configurados. El script es el siguiente:

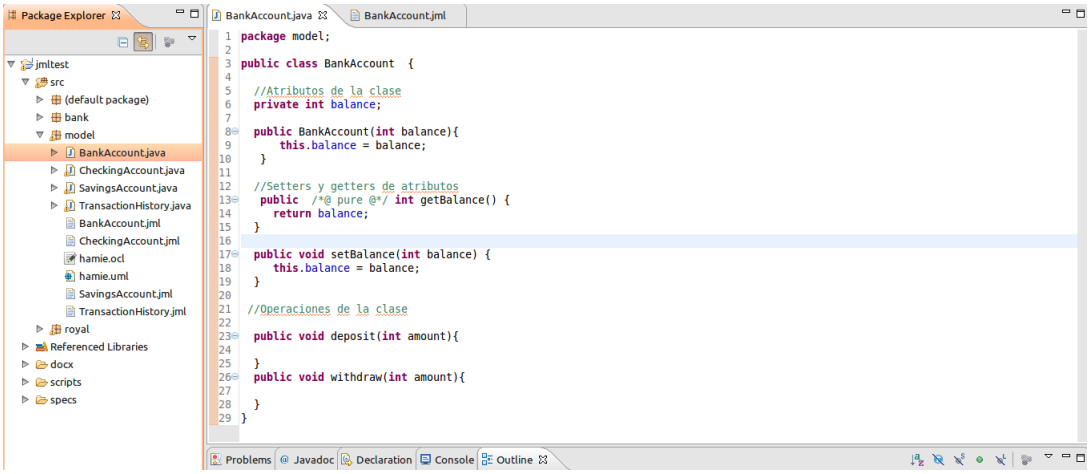

Figura 4: Clase Bank Account Generada

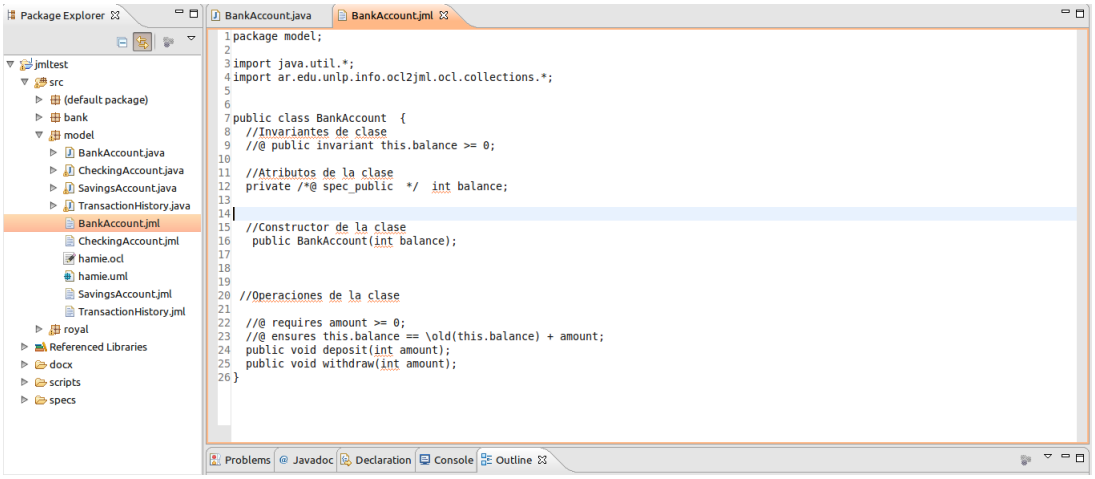

Figura 5: Especificación JML de la clase Bank Account Generada

```
OPEN_JML_PATH={path al ejecutable de OpenJML}
OCL_PATH={path donde se encuentra ubicado el archivo ocl2jmlcollections.jar}
SRC_PATH={path al código fuente}

if [ $# -lt 3 ]
then
        echo "Usage ./openjml.sh java_version java_class option"
        exit
fi

case "$1" in
  java6) java -Xbootclasspath/p:$OPEN_JML_PATH/openjmlboot.jar::$OPEN_JML_PATH/jSMTLIB.jar
-jar $OPEN_JML_PATH/openjml.jar
-cp $OCL_PATH/ocl2jmlcollections.jar:$OPEN_JML_PATH/jmlruntime.jar -specspath $SRC_PATH
-sourcepath $SRC_PATH -showNotImplemented -progress -counterexample -trace -command $3 $2
  ;;
  java7) java -cp $OPEN_JML_PATH/openjml.jar:$OPEN_JML_PATH/jSMTLIB.jar org.jmlspecs.openjml.Main
-cp $OCL_PATH/ocl2jmlcollections.jar:$SRC_PATH:$OCL_PATH/guava-10.0.1.jar
-specspath $SRC_PATH:$OCL_PATH/ocl2jmlcollections.jar:$OCL_PATH/guava-10.0.1.jar
-sourcepath $SRC_PATH -showNotImplemented -progress -command $3 $2
  ;;
esac

$SRC_PATH -rac $1
```

Como **OpenJML** corre a través de Java 7, debemos setear nuestra versión de la máquina virtual de Java a Java 7, si no la tuviéramos. Esto se realiza a través del siguiente comando por consola:

```
sudo update-alternatives --config java
```

Presionamos enter y se nos presentará en pantalla lo siguiente:

```
Existen 2 opciones para la alternativa java (que provee /usr/bin/java).

  Selecci\'on   Ruta                                          Prioridad  Estado
------------------------------------------------------------
* 0            /usr/lib/jvm/java-6-openjdk/jre/bin/java   1061      modo autom\'atico
  1            /usr/lib/jvm/java-6-openjdk/jre/bin/java   1061      modo manual
  2            /usr/lib/jvm/jdk1.7.0/jre/bin/java         3         modo manual

Pulse <Intro> para mantener el valor por omisión [*] o pulse un número de selección:
```

Para lo cual debemos elegir la opción 2.

Corriendo el script Desde una consola acceder al directorio donde tenemos el script a ejecutar y luego escribir:

```
./openjml.sh java7 ../src/model/BankAccount.java esc
```

En pantalla se nos mostrará una salida como la Figura 6.

Los warnings que se encontraron fueron:

1. En el método constructor de la clase, no se puede establecer la invariante declarada para la clase.
2. En el método *setBalance*, no se puede establecer la invariante declarada para la clase.
3. En el método *deposit*, el prover no puede establecer la aserción de postcondición.

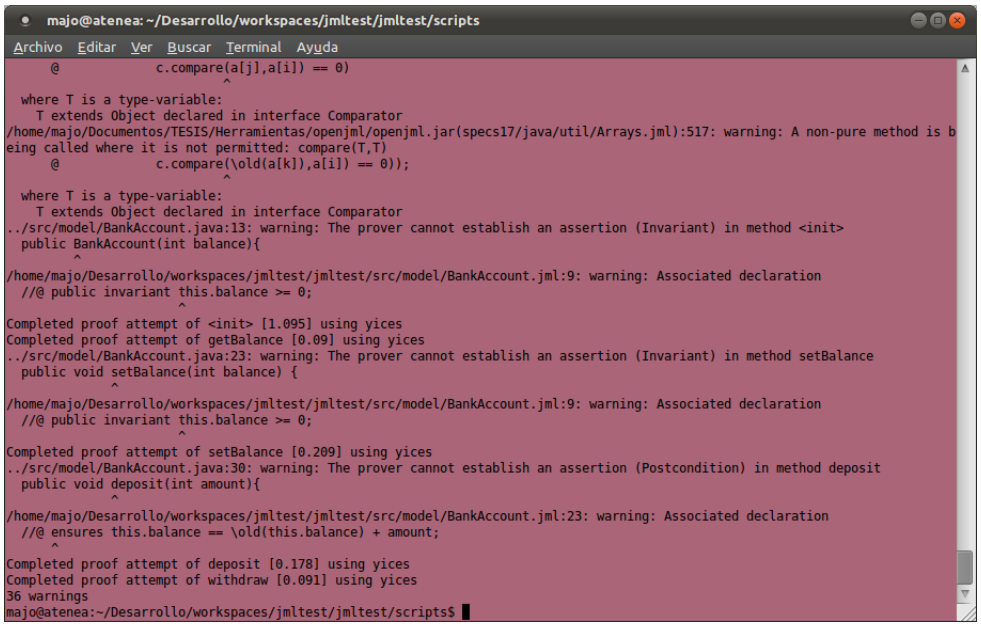

Figura 6: Primer salida de la consola

Lo que debemos hacer ahora es modificar ciertos métodos y volver a ejecutar el script de **OpenJML**.

Debemos borrar los métodos *getBalance* y *setBalance*, ya que sólo se puede modificar el saldo de la cuenta a través de los métodos *deposit* y *withdraw*, los cuales debemos implementar.

De esta forma la clase Java `BankAccount` quedaría como lo mostrado en la Figura 7

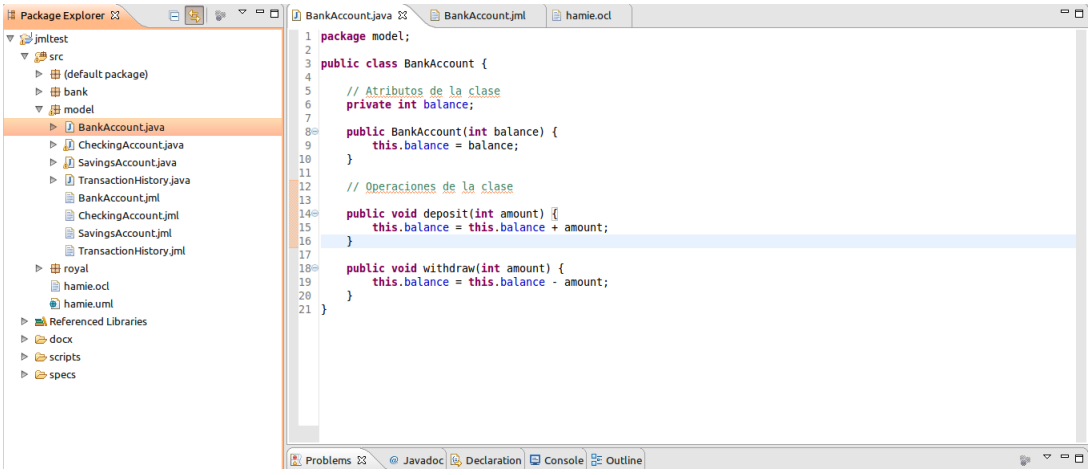

Figura 7: Clase Bank Account modificada

Volvemos a ejecutar el script de OpenJML y como resultado obtenemos lo mostrado en la Figura 8, donde nos indica que el método *withdraw* puede violar el invariante. Debemos modificar el archivo *.ocl* para agregar una precondición al método *withdraw*. El *.ocl* modificado se muestra en la Figura 9.

Nuestro siguiente paso será generar nuevamente las especificaciones *.jml* para luego correr el script. Las especificaciones regeneradas se muestran en la Figura 10.

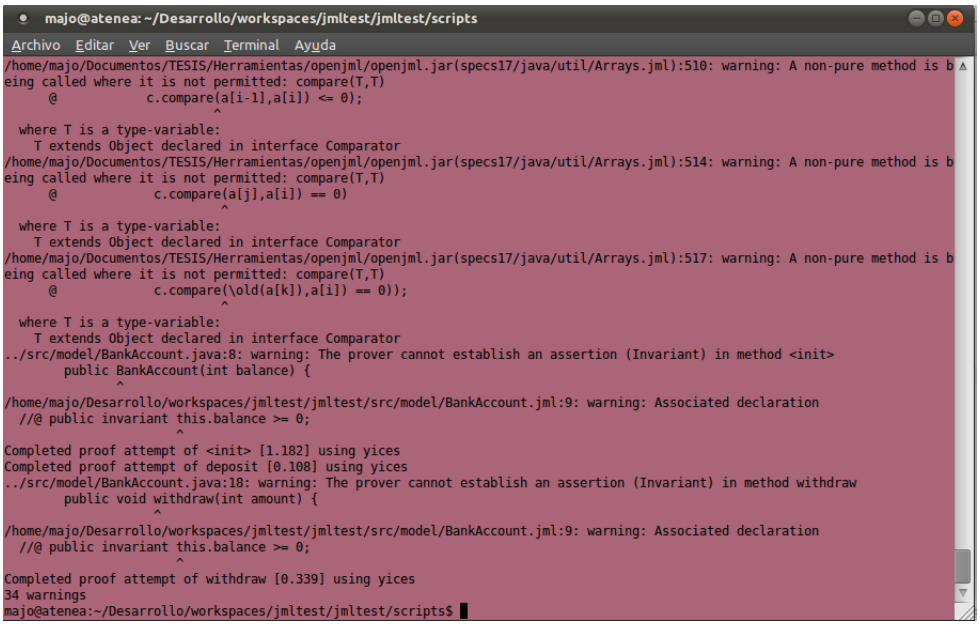

Figura 8: Segunda salida de consola

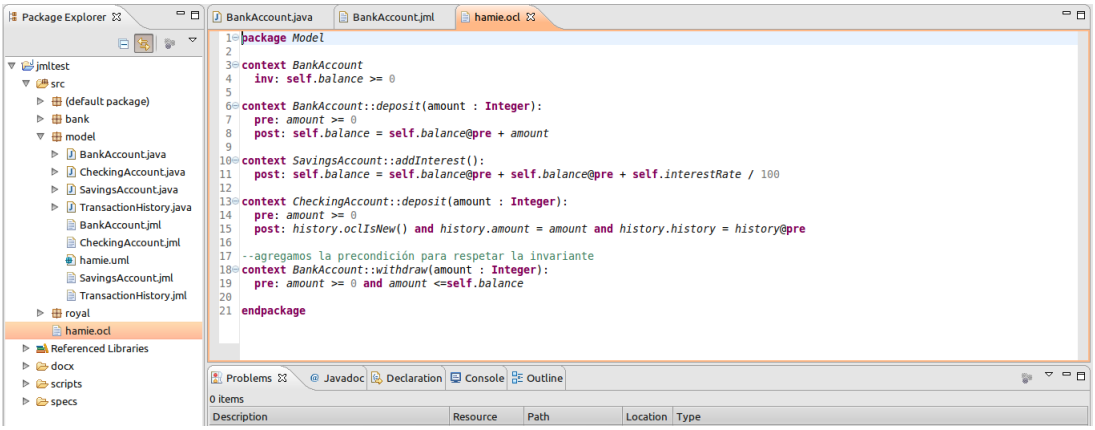

Figura 9: OCL modificado

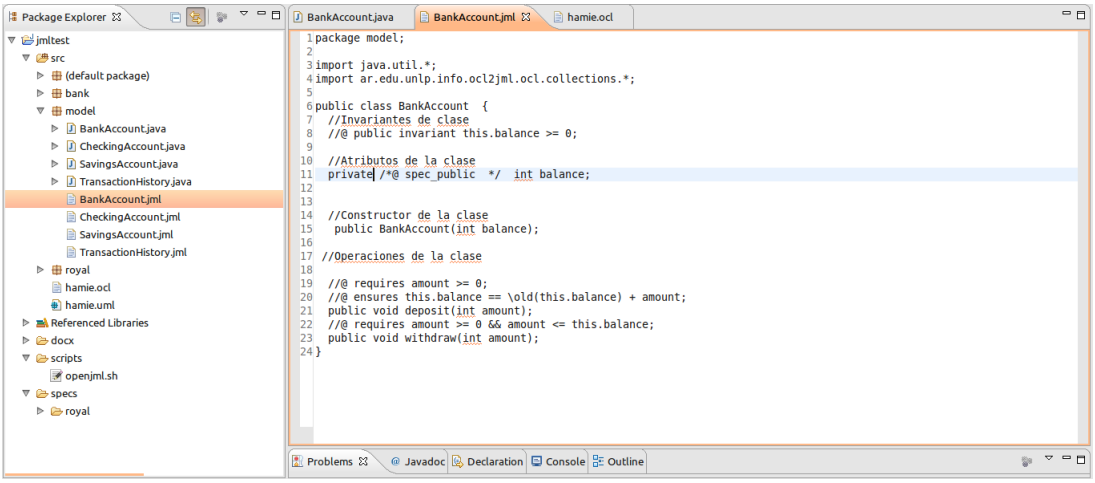

Figura 10: JML regenerado

Volvemos a ejecutar el script de **OpenJML** y como resultado obtenemos lo mostrado en la Figura 11. **OpenJML** nos informa que probó satisfactoriamente los métodos *deposit* y *withdraw*.

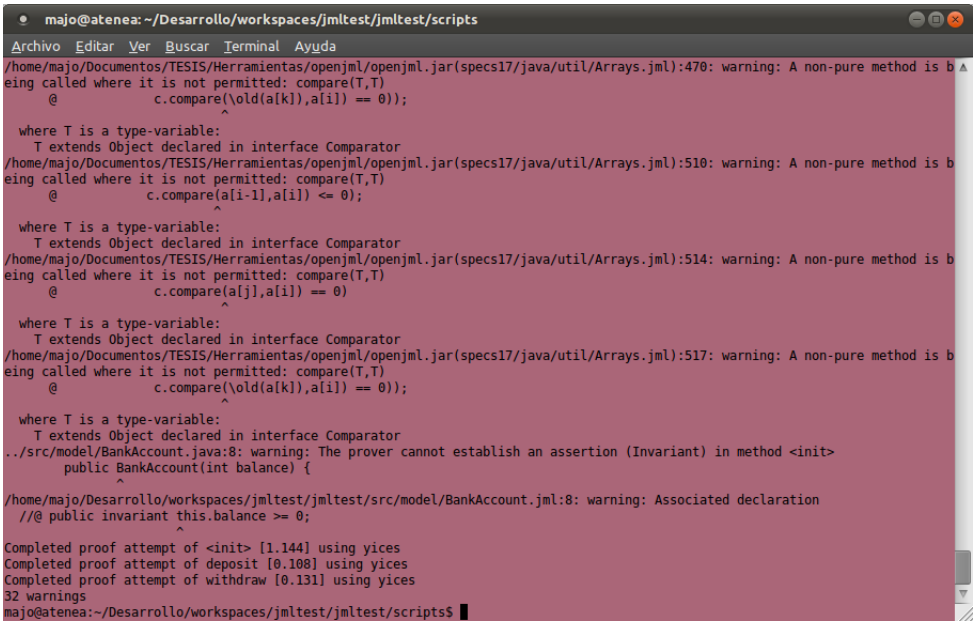

Figura 11: Tercera salida de consola

6. Trabajos relacionados

En esta sección se presentan distintos trabajos relacionados al tema de nuestro trabajo.

- En [9] se presentan y discuten diferentes estrategias para traducir algunos aspectos de los modelos de diseño UML/OCL a diseños JML/Java, es decir clases e interfaces Java anotadas con aserciones JML. Se presenta la traducción de clases, atributos, invariantes, asociaciones directas con varias multiplicidades, agregaciones y composiciones, generalizaciones y por último especificaciones de operaciones. El conjunto de defaults que se adoptó, permite ser automatizado por una herramienta que pueda tomar un modelo UML/OCL y traducirlo a un diseño JML/Java. Uno de los beneficios es que permite el uso de JML para la especificación de las restricciones del objeto en la etapa de desarrollo de una aplicación Java usando UML y OCL. Otro de los beneficios incluye el uso de un amplio rango de herramientas que soportan JML para especificación, testing y verificación de programas Java.

- En [17] se presenta un mapeo de expresiones OCL y restricciones a JML. El mapeo puede usarse para traducir modelos de diseño orientado a objetos expresado en UML y OCL a clases e interfaces Java anotadas con especificaciones JML. Como JML tiene una sintaxis parecida a Java, es mejor para especificar y detallar el diseño de un programa Java que un lenguaje genérico como lo es OCL. El uso de JML también asegura que las herramientas existentes que soportan JML pueden usarse para testear y verificar la correctitud de los programas con respecto a sus especificaciones. El mapeo puede ser también incorporado con herramientas que generan código Java a partir de diagramas UML. En este paper se presenta la traducción de tipos básicos, colecciones y sus operadores, las expresiones de iteración y por último como otros constructores son traducidos.

- En [18] se plantea el problema que las restricciones OCL no pueden ejecutarse y chequearse en tiempo de ejecución, entonces se propone el chequeo de las restricciones OCL en tiempo de ejecución traduciéndolas en aserciones JML ejecutables. Esto se hace a través de:

- Separar aserciones JML del código fuente. La ventaja de utilizar un archivo *.jml* con las aserciones, es que si se realiza alguna modificación en el código java y hay que reescribir las aserciones sólo debe modificarse el archivo *.jml*.
- El uso de model variable. Éstos se diferencian de una variable de programa Java, en dos aspectos: primero porque sólo se utilizan en la especificación y sólo puede ser referenciadas en las aserciones y no en el código del programa. Segundo el valor no se manipula directamente usando sentencias de asignación, sino que se da implícitamente como un mapeo de un estado de programa, llamado *abstraction function*. En resumen lo que se propone es para un elemento de un modelo UML, como una agregación, se introduce el model variable JML correspondiente de un tipo apropiado y se traducen a restricciones OCL escritas en términos de un elemento UML en aserciones escritas en términos del correspondiente model variable.
- Introducción de una nueva librería JML que implementa la librería estándar de OCL como los tipos de colecciones, la introducción de las clases existentes en OCL a JML permite mapear las restricciones OCL directamente a aserciones JML preservando la estructura original y usando casi el mismo vocabulario. La técnica específica es escribir las aserciones JML no en términos de estados de programa Java (como las variables de programa) sino en términos de sus abstracciones usando las clases de la librería.

- En [21] se presenta una implementación de un compilador de OCL a JML. Se exponen los mapeos utilizados para las expresiones, los operadores, tanto para los que se realiza un mapeo directo como para los que no y también nombrando los operadores que Java no soporta, como por ejemplo el *implies*, que no tienen mapeo directo. También hay una sección dedicada a los mapeos de colecciones explicando la causa de la dificultad para realizar esa traducción.

- En [16] se aplica OCL para desarrollar la comprensión Java de modelos de diseño UML donde JML es usado como lenguaje de aserciones. Esto se logra traduciendo un subconjunto de aserciones OCL en aserciones JML.

  En este trabajo se propone una formalización de la relación "realizes" con respecto a la implementación escrita en Java. Esta formalización se hace en el contexto de JML. Dado un modelo de diseño representado por un diagrama de clases con restricciones OCL, los requerimientos sintácticos y semánticos de la relación inducidos por este modelo serán definidos. Los requerimientos semánticos se dan por la especificación JML la cual expresa invariantes de clase, y pre y postcondiciones para la implementación de los métodos. Estos requerimientos pueden verificarse generando las especificaciones JML, y luego probarlas usando un demostrador de teoremas como PVS.

  La comprensión de los modelos de diseño UML por los subsistemas Java ha sido formalizado en el contexto de JML. Esto se hace mapeando expresiones y aserciones escritas en OCL a expresiones y aserciones JML. La formalización provee un enfoque para verificar la relación de comprensión donde JML es usado como lenguaje de aserción. Una forma posible de hacer la verificación es usar las herramientas desarrolladas por el proyecto LOOP el cual traduce el código Java anotado con especificaciones JML en aserciones PVS y obligaciones. Estas obligaciones son verificadas usando PVS. Esta propuesta puede usarse en el contexto de herramientas CASE las cuales generan código Java desde los diagramas UML, donde las restricciones OCL pueden ser mapeadas a aserciones JML. Esto ayuda en el debug y el testeo usando los chequeadores de aserciones en tiempo de ejecución de JML para chequear invariantes, predondiciones y postcondiciones de métodos.

- En [22, 23] se presenta una traducción de diagramas de clase UML con su complemento de restricciones OCL a expresiones Object-Z. El fin es proveer una formalización de los modelos gráfico-textuales expresados mediante UML/OCL que permita aplicar técnicas clásicas de verificación y prueba de teoremas sobre los modelos. Esa traducción fue implementada como parte de una herramienta CASE que permite editar y gestionar modelos, desarrollada como un plugin a la plataforma universal Eclipse. La traducción se realiza de la siguiente forma, dado un diagrama de clases UML, que incluye expresiones OCL, se utiliza la lógica de primer orden y teoría de conjuntos para representarlo formalmente. Luego, utilizando los mecanismos de la

lógica será posible verificar la validez de las restricciones expresadas inicialmente en OCL. Se utiliza para la representación del sistema el lenguaje de especificación de sistemas Object-Z que es una extensión del lenguaje Z.

6.1. Aportes

El desarrollo de este trabajo, aporta a los trabajos relacionados mencionados anteriormente, lo siguiente: al existir una herramienta de derivación automática de OCL a JML, permite separar la especificación de las aserciones JML, las especificaciones pueden ser corregidas y regeneradas fácilmente, tal como se proponía en [18]. La implementación de la herramienta como plugin de Eclipse permitió utilizar otros plugins como el OCL Plugin y UML Plugin.

7. Trabajos futuros

Para realizar la traducción de OCL a JML se recurrió a un esquema tradicional de traducción, es decir, a partir del texto de las restricciones OCL se obtiene un árbol sintáctico, el cual es recorrido para transformarlo en un árbol sintáctico de JML, el cual es traducido a texto. Una de las posibles extensiones podría ser utilizar una transformación M2M para transformar el modelo OCL a un modelo JML.

Debido a la semántica y limitaciones del lenguaje Java, ciertas operaciones de colecciones como *iterate* no pueden ser implementados fácilmente. En la versión 8 de Java se prevee la incorporación de funciones anónimas al lenguaje, lo que permitiría implementar esas operaciones. Se debería analizar que aportan las nuevas construcciones a la semántica de Java y JML y como pueden facilitar la traducción.

La herramienta desarrollada no provee un editor de modelos UML (utiliza el Papyrus) ni un editor de restricciones OCL. Se podría desarrollar editores que permitan la construcción de modelos UML y la edición de archivos *.ocl*. En el caso de OCL el editor proveería resaltado de sintaxis, resaltado de errores y verificación automática de las restricciones contra el modelo. También se podrían tener en cuenta las restricciones definidas en el modelo UML y no en un archivo aparte.

Existen otros lenguajes de especificación como Jass para Java y Spec# para C#, podría extenderse el plugin para que realice la traducción de OCL a los lenguajes mencionados.

Las herramientas de verificación estática de código se utilizan manualmente luego de generar el código. Podrían integrarse al plugin y ejecutarse automáticamente.

En la traducción, el constructor de la clase se realizó asumiendo que solo tenía una superclase y que ésta última no tenía ninguna superclase. Esta decisión se tomó ya que el realizar el constructor asumiendo mas de dos niveles, requería una mayor complejidad de desarrollo, que no hacían diferencia para el trabajo propuesto. Esta restricción puede ser eliminada en una posible extensión de la herramienta.

8. Conclusiones

En este trabajo analizamos dos lenguajes de especificación muy populares: OCL y JML. OCL es un lenguaje útil en las fases iniciales del diseño, ya que permite especificar restricciones sobre los modelos UML. Sin embargo la utilidad de OCL se vuelve menos relevante en la implementación, ya que OCL no provee construcciones para especificar implementaciones. JML por otro lado provee métodos para desarrollar especificaciones en la implementación. Ambos lenguajes son muy similares y las especificaciones OCL de la fase de diseño pueden ser traducidas a JML y utilizadas en la implementación.

La herramienta desarrollada realiza la traducción de las restricciones OCL a anotaciones JML, lo que facilita y vuelve menos propenso a errores el mantenimiento de las especificaciones al realizar cambios en el modelo. Al realizar la separación entre las especificaciones y el código generado se facilita la regeneración sin problemas de las especificaciones JML, sin necesidad de modificar el código generado y las especificaciones que el código tuviese.

La existencia de herramientas de verificación de programas disponibles para JML permite el descubrimiento de posibles errores en la especificación. Al existir derivación automática de OCL a JML, las especificaciones pueden ser corregidas y regeneradas fácilmente.

La implementación de la herramienta como plugin de Eclipse permitió reutilizar los plugins de Eclipse de UML y OCL, logrando de esa manera dirigir el esfuerzo principal a realizar la traducción.

Referencias

[1] Rosenfeld, Ricardo y Jeronimo Irazábal: *Teoría de la computación y verificación de programas.* Editorial de la UNLP, 2010.

[2] Davies, Jim y Jim Woodcock: *Using Z: Specification, Refinement and Proof.* 1996.

[3] Leavens, Gary T. y Yoonsik Cheon: *Design by Contract with JML.* Septiembre 2006.

[4] OMG: *The Object Constraint Language Specification*, Febrero 2010. `http://www.omg.org/spec/OCL/2.2`.

[5] Milley, Jonathan y Dr. Dennis K. Peters: *Software Specification and Testing Using UML and OCL.* Noviembre 2005.

[6] Warmer, Jos y Anneke Kleppe: *The Object Constraint Language: Getting Your Models Ready for MDA.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2ª edición, 2003, ISBN 0321179366.

[7] Meyer, Bertrand: *Applying design by contract.* IEEE Computer, 25:40–51, 1992.

[8] Beckert, Bernhard, Reiner Hähnle y Peter H. Schmitt (editores): *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.

[9] Hamie, Ali: *Strategies for Translating UML/OCL Design Models to JAVA/JML Designs.* Diciembre 2004.

[10] Chalin, Patrice: *JML Support for Primitive Arbitrary Precision Numeric Types: Definition and Semantics.* En *Proceedings, Workshop on Formal Techniques for Java-like Programs (FTfJP at ECOOP 2003)*, Darmstadt, Germany, Julio 2003.

[11] Dzidek, Wojciech J., Lionel C. Bri y Yvan Labiche: *Y.: Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java.* En *In: Proc. MODELS 2005 Workshops, LNCS, 3844*, páginas 10–19. Springer-Verlag, 2005.

[12] Leavens, Gary T., Albert L. Baker y Clyde Ruby: *JML: A Notation for Detailed Design.* Behavioral Specifications of Businesses and Systems, 1999.

[13] Ahrendt, Wolfgang: *The KeY Tool.* Software and system modeling, páginas 32–54, 2005.

[14] Cok, David R.: *OpenJML: JML for Java 7 by extending OpenJDK.* En *Proceedings of the Third international conference on NASA Formal methods*, NFM'11, páginas 472–479, Berlin, Heidelberg, 2011. Springer-Verlag, ISBN 978-3-642-20397-8. `http://dl.acm.org/citation.cfm?id=1986308.1986347`.

[15] Burdy, Lilian, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino y Erik Poll: *An overview of JML tools and applications.* Int. J. Softw. Tools Technol. Transf., 7:212–232, Junio 2005, ISSN 1433-2779. `http://dl.acm.org/citation.cfm?id=1070908.1070911`.

[16] Hamie, Ali: *Towards Verifying Java Realizations Of OCL-Constrained Design Models Using JML.* 2002.

[17] Hamie, Ali: *Translating the Object Constraint Language into the Java Modelling Language.* En *Proceedings of the 2004 ACM symposium on Applied computing*, SAC '04, páginas 1531–1535, New York, NY, USA, 2004. ACM, ISBN 1-58113-812-1. `http://doi.acm.org/10.1145/967900.968206`.

[18] Avila, Carmen, Guillermo Flores y Yoonsik Cheon: *A library-based approach to translating OCL constraints to JML assertions for runtime checking.* En *In International Conference on Software Engineering Research and Practice, July 14-17, 2008, Las Vegas*, páginas 403–408, Julio 2008.

[19] *Eclipse Platform Technical Overview*. Febrero 2003. `http://www.eclipse.org/whitepapers/eclipse-overview.pdf`.

[20] Oldevik, Jon: *MOFScript User Guide. Version 0.9 (MOFScript v 1.4.0)*, Febrero 2011.

[21] Farías, José Rafael, Renato Almeida y Solon Aguiar: *Projeto Compilador OCL - JML Geração de Código JML*. Universidade Federal de Campina Grande (UFCG), Junio 2011.

[22] Becker, Valeria y Claudia Pons: *Definición formal de la semántica de UML-OCL a través de su traducción a Object-Z*. En *IX Congreso Argentino de Ciencias de la Computación CACIC*, Octubre 2003.

[23] Becker, Valeria: *Herramienta para automatizar la transformación UML/OCL a Object-Z*. Tesis de Licenciatura, UNLP, Diciembre 2006.