UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Desarrollo de una herramienta para derivación automática de especificaciones OCL a JML
Autores: María José Dias Molina, Diego Matías Dodero Mena
Directores: Dra. Claudia Pons, Prof. Ricardo Rosenfeld
Carrera: Licenciatura en Sistemas, Licenciatura en Informática

Resumen

UML es un poderoso método para diseñar y documentar sistemas de software. Existen muchas herramientas para asistir en la creación y mantenimiento de los documentos UML. Las más avanzadas incluyen ciertas características que permiten la traducción de modelos UML a código, y viceversa. UML no está lo suficientemente refinado para proveer todos los aspectos relevantes de una especificación, ya que hay ciertas restricciones que no pueden expresarse en el modelo y muchas veces son descriptas en lenguaje natural, pero esto puede resultar en ambigüedades. Para escribir restricciones no ambiguas se han desarrollado lenguajes formales. Uno de ellos es OCL, que permite incrementar la precisión de los modelos UML. Además existe JML para especifcar programas escritos en el lenguaje Java. La motivación principal de la tesina es analizar y comparar los lenguajes OCL y JML, para luego definir una traducción entre ambos lenguajes y finalmente desarrollar una herramienta que realice la traducción automáticamente.

Palabras Claves

OCL, JML, Comparación OCL y JML, Verificación de programas, OCLE, ESC/Java2, Sireum/Kiasan, JML4c, OpenJML, Key, Loop, Jack, Metamodelos, Desarrollo dirigido por modelos (MDD), Eclipse, MOFScript, Traducción OCL a JML, UML, Colecciones OCL.

Trabajos Realizados

Investigación del lenguaje OCL y JML. Investigación del lenguaje de transformación MofScript. Comparación entre los lenguajes de OCL y JML. Utilización de los plugin que UML y OCL que provee eclipse para realizar la herramienta de traducción del modelo UML y restricciones OCL al modelo Java con anotaciones JML. Búsqueda de distintas herramientas de verificación de código, para analizar el código generado por nuestra herramienta de derivación.

Conclusiones

La herramienta de traducción OCL a JML, hace que el mantenimiento de las especificaciones sea más fácil, ya que al realizar cambios en el modelo, se facilita la regeneración de las especificaciones JML, sin modificar el código generado y sus especificaciones, ya que estos se generan por separado. La existencia de herramientas de verificación de programas disponibles para JML posibilita el descubriento 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.

Trabajos Futuros

La traducción de las restricciones OCL a notaciones JML está basado en un esquema clásico de recorrido de árbol sintáctico, se podría analizar la factibilidad de la traducción utilizando una transformación M2M.

La herramienta desarrollada no provee un editor de modelos ni un editor de restricciones OCL. Se podría desarrollar un editor para edición de modelos UML y archivos de restricciones con resaltado de sintaxis y verificación de las restricciones contra el modelo.

Analizar el impacto en la traducción de las nuevas características incorporadas en Java 8, en particular las funciones anónimas.

Extensión del plugin para traducir desde OCL a otros lenguajes de especificación como Jass o Spec#.

Fecha de la presentación: Noviembre 2011

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

Tesina de licenciatura

Alumnos
María José Dias Molina
Diego Matías Dodero Mena

Directores
Dra. Claudia Pons
Prof. Ricardo Rosenfeld

2011

Facultad de Informática
Universidad Nacional de La Plata

Agradecimientos

Quiero agradecer a mi familia, por el apoyo que me dieron siempre para poder llegar a donde estoy.

A mis compañeros de trabajo, quienes también me apoyaron durante todo este año para que termine la tesina.

Y por último, pero no menos importante a Diego, mi compañero de tesina sin el cual no hubiera podido terminar este trabajo.

— María José

A mis compañeros con los que compartí la carrera y que me acompañaron en esta etapa de mi vida, mi paso por la facultad no hubiese sido lo mismo sin ellos.

A mi familia, por el apoyo y la confianza que tuvieron siempre conmigo. A mi mamá, que ya no está conmigo, pero que estaría orgullosa de que esté concluyendo esta etapa.

A Majo, mi compañera de tesina, por su responsabilidad y paciencia, sin ella no hubiese podido terminar este trabajo.

— Diego

También queremos agradecer especialmente a Claudia y Ricardo por todo el apoyo brindado durante este año y la buena predisposición de ambos para que terminemos la tesina. Agradecemos a Jerónimo Irazábal, quien nos dio la orientación inicial para el desarrollo.

— Majo y Diego

Índice general

Índice de figuras

Índice de tablas

Índice de código fuente

Capítulo 1

Introducción

En este capítulo se describe la motivación del trabajo de tesina, el problema a resolver y los resultados esperados. Finalmente se describe la estructura general de la tesina.

1.1. El problema de la corrección del software

El objetivo de la corrección de software es asegurar programas correctos y libres de errores. Las aproximaciones más importantes para alcanzar este objetivo son el testing y la verificación formal.

El testing consiste en suministrar datos de entrada a un programa y estudiar su comportamiento en esos casos. Actualmente existe una metodología llamada Test-driven development (TDD) que consiste en escribir los tests antes de empezar a desarrollar el programa y tener un conjunto de tests que puedan ejecutarse automáticamente al realizar un cambio en el programa. A pesar de esto el testing presenta limitaciones para garantizar la corrección del software, ya que el testing sirve para encontrar errores pero no garantiza su ausencia.

1.2. Verificación formal del software

La verificación formal del software plantea una metodología para probar la correctitud de un programa con respecto a una especificación. Hay dos aproximaciones a la verificación, una operacional que consiste en analizar los estados del programa y una axiomática, la cual define un método de prueba, con axiomas y reglas asociadas a las instrucciones del lenguaje de programación [50]. Uno de los principales sistemas axiomáticos fue presentado por C.A.R. Hoare en 1969, ese sistema fue extendido por muchos investigadores, incluyendo al propio Hoare.

La lógica de Hoare ha dado origen a diversos lenguajes de especificación, y junto con sus ampliaciones puede ser utilizada para la verificación de programas secuenciales, concurrentes y distribuidos.

1.3. Motivación

Hay dos áreas importantes que han alcanzado un gran desarrollo: los métodos de especificación formal de programas y las herramientas de análisis estático de código. Existe una retroalimentación entre ambos aspectos, ya que las herramientas de análisis se basan en la especificación formal de programas para demostrar propiedades sobre los programas.

Uno de las principales lenguajes para especificar, construir y documentar los sistemas es Unified Modeling Language (UML). Para formalizar los modelos realizados en UML existe Object Constraint Language (OCL), de la misma forma existe el lenguaje Java Modeling Language (JML) para especificar los programas escritos en el lenguaje Java.

Para poder utilizar las herramientas de análisis estático sobre el código derivado (al lenguaje Java en particular) del modelo UML con especificaciones, deben definirse especificaciones JML para el código derivado, sin embargo esta metodología no es escalable, ya que deben traducirse manualmente las especificaciones OCL. Esto es propenso a errores, ya que se deben mantener dos conjuntos de especificaciones, y los cambios en un conjunto deben replicarse en el otro. También las especificaciones OCL pueden quedar obsoletas a medida que se avanza en la implementación.

La motivación principal de la tesina es analizar y comparar los lenguajes OCL y JML, para luego definir una traducción entre ambos lenguajes y finalmente desarrollar una herramienta que realice la traducción automáticamente.

1.4. Objetivos de la tesina

Los objetivos planteados en la tesisa son:

- Investigar métodos formales para la especificación de programas.
- Comparar los lenguajes de especificación existentes.
- Investigar herramientas de verificación de programas.
- Integrar JML con OCL dentro del marco del desarrollo orientado a modelos.
- Investigar la factibilidad de la traducción automática de OCL a JML.

1.5. Estructura general de la tesina

La tesina se estructura de la siguiente manera:

En el Capítulo 2 explicamos el lenguaje OCL, sus objetivos, su sintaxis y semántica. También se explica su relación con UML.

En el Capítulo 3 presentamos el lenguaje JML, sus objetivos y su relación con DBC. Explicamos también la sintaxis y semántica de JML, su relación con la semántica del lenguaje Java. Mostramos las diferentes partes de una especificación JML y cómo permite construir especificaciones puras, es decir que permite especificar una clase la cual se puede implementar cumpliendo con la especificación.

En el Capítulo 4 comparamos OCL con JML, ventajas y desventajas de cada uno. Veremos que los lenguajes no son equivalentes y mostraremos las diferencias entre los mismos. Presentaremos también un función de traducción que transforma un restricción OCL en una especificación JML.

En el Capítulo 5 clasificamos las diferentes herramientas utilizadas para verificación de programas basadas en JML. En particular analizamos detalladamente herramientas de verificación estática y de chequeo en tiempo de ejecución. También presentamos el panorama de la evolución de las herramientas de JML, no sólo en funcionalidad, sino también su evolución para tener en cuenta las nuevas características del lenguaje Java.

En el Capítulo 6 introducimos conceptos básicos sobre Model driven development (MDD). Además explicamos los conceptos básicos de metamodelos. Presentamos también el lenguaje Meta Object Facility (MOF) para definir metamodelos e Ecore, una implementación de MOF.

En el Capítulo 7 describimos la plataforma Eclipse y la arquitectura de plugins. Se presenta el framework Eclipse Modeling Framework (EMF) y plugins UML2 y OCL.

En el Capítulo 8 presentamos la herramienta MOFScript para transformación de modelo a texto utilizada en la implementación.

En el Capítulo 9 se presenta el prototipo implementado como plugin. Explicamos el diseño e implementación de la herramienta. Presentamos también la implementación de la librería de colecciones OCL en Java.

En el Capítulo 10 vemos un caso de estudio basado en el ejemplo Royal and Loyal expuesto en [53].

En el Capítulo 11 se presentan diversos trabajos relacionados con nuestro tema de investigación.

En el Capítulo 12 se presentan las conclusiones finales y las contribuciones principales que aporta esta tesina al desarrollo formal, así como las líneas de investigación a futuro que pueden extenderla.

Capítulo 2

Object Constraint Language

UML es un poderoso método para diseñar y documentar sistemas de software. Existen muchas herramientas para asistir en la creación y mantenimiento de los documentos UML. Las más avanzadas incluyen ciertas características que permiten la traducción de modelos UML a código, y vice versa. [43, 53]

El OCL es un lenguaje de especificación de modelado diseñado y mantenido por el Object Management Group (OMG) el cual crea y mantiene el estándar UML. OCL está diseñado como un complemento para UML.

Un diagrama UML, como por ejemplo un diagrama de clases, no está lo suficientemente refinado para proveer todos los aspectos relevantes de una especificación. Existe, entre otras cosas, una necesidad por describir restricciones adicionales sobre objetos en el modelo. 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 las personas que modelan un sistema). OCL ha sido desarrollado para solucionar la deficiencia descripta.

OCL es un lenguaje orientado a objetos, fácil de leer y escribir, que permite incrementar la precisión de los modelos UML [48], es decir, permite expresar restricciones semánticas del sistema que no se pueden expresar a partir de una notación gráfica. De esta forma, los diagramas complementados con expresiones OCL son más precisos, su documentación es más clara, se mejora la comunicación entre desarrolladores (evitando errores producidos por malas interpretaciones) y la comprensión del sistema en etapas iniciales del desarrollo de software es mayor. Es un lenguaje puro por lo que se garantiza que toda expresión OCL es libre de efectos laterales; es decir cuando una expresión OCL es evaluada simplemente retorna un valor, sin modificar nada en el modelo.

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

OCL es un lenguaje tipado, lo que significa que cada expresión tiene un tipo. Para ser bien formada, una expresión debe concordar con los tipos de reglas del lenguaje. Cada `Classifier` [1] definido con un modelo UML representa un tipo distinto OCL. Además, OCL incluye un conjunto de tipos suplementarios predefinidos. Para mas información ver el apéndice B Librería estándar de OCL.

Como lenguaje de especificación, todas las cuestiones de implementación están fuera del alcance y no pueden ser expresadas en OCL. La evaluación de una expresión OCL es instantánea. Esto significa que el estado de los objetos en un modelo no pueden cambiar durante la evaluación.

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

Las restricciones OCL no pueden ejecutarse directamente y ser controladas en tiempo de ejecución, por lo que violaciones a las restricciones pueden no ser detectadas, provocando problemas de mantenimiento y desarrollo.

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 Estereotipos
- para describir pre y post condiciones en Operaciones y Métodos
- para describir Guardas
- para especificar reglas de derivación para una propiedad
- para especificar restriccciones en operaciones
- para especificar los valores iniciales de las propiedades

Para los ejemplos de OCL, se utilizará el siguiente diagrama de clases.

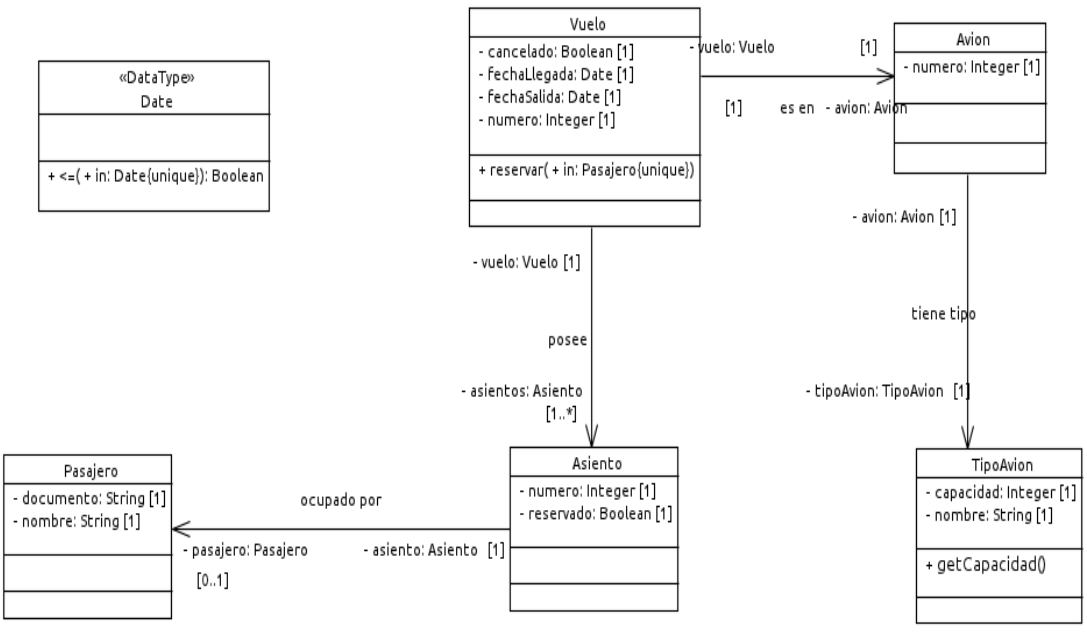

Figura 2.1: Ejemplo de vuelos

2.1. Restricciones en OCL

Las restricciones que podemos especificar con OCL son:

- Invariantes
- Definiciones
- Precondiciones
- Postcondiciones

2.1.1. Invariante

Un invariante es una restricción que define una condición que debe ser válida siempre para todas las instancias de la clase.

La restricción tiene el estereotipo *<<invariant>>*.

Su sintaxis es:

```
context [VariableName:] TypeName
  inv: <OclExpression>
```

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

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

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

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

2.1.2. Definición

Una definición es una restricción cuyo propósito es definir expresiones OCL reusables. La restricción tiene el estereotipo *<<definition>>*.

Su sintaxis es:

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

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

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

La variable capacidad es conocida en el contexto de `Vuelo`.

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

2.1.3. Precondición

Una precondición es una restricción que establece una condición que debe cumplirse antes de ejecutar la operación.

La restricción tiene el estereotipo << *precondition* >>.

Su sintaxis es:

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

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

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

La variable *capacidad* es la definida anteriormente en el contexto de `Vuelo`. En la expresión se puede utilizar la variable **self** para referenciar a un objeto del tipo que define la operación.

2.1.4. Postcondición

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

La restricción tiene el estereotipo << *postcondition* >>.

Su sintaxis es:

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

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

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

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

2.2. Expresión de valor inicial

Una expresión de valor inicial es una expresión que actúa como el valor inicial de una propiedad y debe tener el tipo de ésta. Además hay que tener en cuenta su multiplicidad, es decir si la multiplicidad es mayor que uno el tipo es un `Set` u `OrderedSet` del tipo de la propiedad.

Su sintaxis es:

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

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

```
context Vuelo :: cancelado : Boolean
  init : false
```

2.3. Expresión de valor derivado

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

Su sintaxis es:

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

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

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

2.4. Expresión de consulta

Una expresión de consulta es una expresión que se liga a una operación de consulta. Para indicar que es una operación de consulta se utiliza el atributo *isQuery*.

Su sintaxis es:

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

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

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

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

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

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

2.5. Package Context

Para especificar explícitamente a cual paquete pertenece la invariante, pre o postcondición, estas restriccciones pueden encerrarse entre las declaraciones *package* y *endpackage*. Tiene la siguiente sintaxis:

```
package Package::SubPackage
  context X inv:
  ... some invariant ...
  context X::operationName(..)
  pre: ... some precondition ...
endpackage
```

Un archivo OCL, puede contener cualquier número de declaraciones package, permitiendo a todas las invariantes, precondiciones y postcondiciones ser escritas y almacenadas en un sólo archivo. Este archivo, debe coexistir con el modelo UML como una entidad separada.

2.6. Tipos y valores básicos

En OCL existen varios tipos básicos predefinidos e independientes de cualquier modelo de objetos, con un conjunto de operaciones sobre ellos. Por Ejemplo:

Tipo	**Valores**	**Operaciones**
Boolean	true, false	and, or, xor, not, implies, if-then-else
Integer	1, -5, 2, 34, 26524, ...	*, +, -, /, abs()
Real	1.5, 3.14, ...	*, +, -, /, floor()
String	'esto es un String...'	toUpper(), concat()

Tabla 2.1: Valores, tipos y operaciones OCL

2.6.1. Expresiones Let

La expresión let permite definir un atributo derivado o una operación para ser usada luego en otras expresiones OCL. Por ejemplo: Se define un atributo indicando si un asiento está o no reservado para luego utilizarlo.

```
context Asiento inv:
let estaReservado : Boolean = self.reservado
    in self.estaReservado implies self.pasajero <> null
```

Una expresión let puede ser incluida en un invariante, precondición o postcondición. Para poder reutilizar las operaciones y/o variables del **let**, se escribe la palabra **def**. Todas las variables y operaciones definidas en el alcance **def** son conocidas en el mismo contexto donde cualquier propiedad puede ser usada. Veamos un ejemplo:

```
context Asiento def:
let estaReservado : Boolean = self.reservado
```

2.6.2. Ajuste de tipos

OCL es un lenguaje tipado y los tipos de valores básicos están organizados en una jerarquía. Esta jerarquía determina la conformidad de tipos diferentes a otros. No se puede por ejemplo, comparar un Integer con un `Boolean` o un `String`.

Una expresión OCL en la cual todos los tipos se ajustan, es una expresión válida. Una expresión en la cual no se ajustan todos los tipos, es una expresión inválida. Un `tipo1` se ajusta a un `tipo2` cuando una instancia del `tipo1` puede ser sustituida en cada lugar donde se espera una instancia del `tipo2`. Las reglas de ajuste de tipos para un diagrama de clases es simple:

- Cada tipos se ajusta a su supertipo.
- El ajuste de tipos es transitivo: si el `tipo1` se ajusta al `tipo2` y el `tipo2` a `tipo3` entonces `tipo1` se ajusta a `tipo3`.

Las reglas de ajustes de tipos se listan en la Tabla 2.2.

Tipo	**Se ajusta a/Es subtipo de**
Set(T)	Collection(T)
Sequence(T)	Collection(T)
Bag(T)	Collection(T)
Integer	Real

Tabla 2.2: Reglas de ajustes de tipos

La relación de ajuste entre colecciones se mantiene sólo si las colecciones de los tipos de los elementos se ajustan entre sí. La Tabla 2.3 muestra ejemplos de expresiones válidas e inválidas.

expresión OCL	**válida**	**explicación**
1 + 2 * 34	sí	
1 + 'motorcycle'	no	el tipo String no se ajusta al tipo Integer
23 * false	no	el tipo Boolean no se ajusta al tipo Integer
12 + 13.5	sí	

Tabla 2.3: Expresiones válidas

2.6.3. Re-tipeo o casting

En algunas circunstancias, es deseable usar una propiedad de un objeto que está definido en un subtipo del tipo actual. Como la propiedad no está definida en el tipo actual, esto resulta en un error de ajuste de tipo.

Cuando se está seguro que el tipo actual del objeto es el subtipo, el objeto puede ser retipeado usando la operación `oclAsType(OclType)`. Esta operación davuelve como resultado el mismo objeto, pero el tipo es el argumento `OclType`. Cuando hay un objeto **object** de tipo Tipo1 y Tipo2 es otro tipo, se puede escribir:

```
object.oclAsType(Type2) --- evalúa al objeto con el tipo Tipo2
```

Un objeto puede ser retipeado a uno de sus subtipos, entonces en el ejemplo, Tipo2 debe ser subtipo de Tipo1. Si el tipo actual del objeto no es subtipo del tipo al cual estamos retipeando, la expresión es indefinida.

2.6.4. Reglas de precedencia

El siguiente es el orden de precedencia para las operaciones, empezando desde la mas alta precedencia, en OCL es:

- @pre
- punto y operaciones con flecha "." y "->"
- unarias "not" y el menos unario "-"
- "*" y "/"
- "+" y menos binario "-"
- "if-then-else-endif"
- "<", ">", "<=", ">="
- "=", "<>"

- "and", "or" y "xor"
- "implies"
- Paréntesis "(" y ")" pueden ser usados para cambiar la precedencia.

2.6.5. Uso de operadores infijos

En OCL está permitido el uso de operadores infijos. Los operadores '+', '-', '*', '/', '<', '>', '<>', '<=', '>='se utilizan como operadores infijos. Si un tipo define uno de estos operadores correctamente, pueden usarse como operadores infijos. La expresión:

```
a + b
es conceptualmente igual a la expresión:
a.+(b)
```

Esto es, invocando a la operación '+'con b como parámetro de la operación. Los operadores infijos definidos para un tipo deben ser exactamente uno por parámetro. Para los operadores infijos '<', '>', '<>', '<=', '>=', 'and', 'or'y 'xor', el tipo de retorno debe ser Boolean.

2.6.6. Palabras claves

Las keywords en OCL son las palabras reservadas. Esto significa que no pueden estar en el nombre de un paquete, o expresión, tipo o propiedad. La lista de palabras claves son: **if**, **then**, **else**, **endif**, **not**, **let**, **or**, **and**, **xor**, **implies**, **package**, **endpackage**, **context**, **def**, **inv**, **pre**, **post**, **in**.

2.6.7. Comentarios

Los comentarios en OCL se escriben con dos guiones seguidos. Todo lo que siga a los dos guiones será el comentario. Por ejemplo:

```
-- esto es un comentario
```

2.6.8. Valores indefinidos

Cuando una expresión OCL está siendo evaluada, hay una posibilidad que una o más de las consultas en la expresión sean indefinidas. Si este es el caso, toda la expresión será indefinida.

Hay dos excepciones de esto, para los operadores Boolean:

- `True` `OR` cualquier otra cosa es `True`
- `False` `AND` cualquier otra cosa es `False`

Las reglas anteriores son válidas, independientemente del orden se sus argumentos y son válidas ya sea que el valor de la otra subexpresión se conozca o no.

2.7. Objetos y propiedades

Las expresiones OCL pueden referirse a `Classifiers`, por ejemplo, tipos, clases, interfaces, asociaciones (actuando como tipos), y tipos de datos. También todos los atributos, los associations-ends, métodos y operaciones sin efectos laterales que están definidas en esos tipos pueden usarse. En un modelo de clases, una operación o método está definido para ser libre de efectos laterales si el atribute `isQuery` de la operación es `true`. Una propiedad puede ser:

- un Attribute
- un AssociationEnd
- un Operation con `isQuery` en `true`
- un Method con `isQuery` en `true`

2.7.1. Propiedades

El valor de una propiedad en un objeto que está definido en un diagrama de clases se especifica por un punto seguido por el nombre de la propiedad.

```
context AType inv:
self.property
```

Si **self** es una referencia a un objeto, entonces $self.property$ es el valor de la propiedad property en **self**.

2.7.2. Propiedades: Atributos

Por ejemplo, la edad de una persona se escribe como $self.age$:

```
context Person inv:
 self.age > 0
```

El valor de la subexpresión $self.age$ es el valor del atributo age en una instancia particular de Person identificada por **self**. El tipo de la subexpresión es el tipo del atributo age, el cual es Integer.

Usando atributos y operaciones definidas en los tipos básicos, podemos expresar cálculos sobre el modelo de clases. Por ejemplo, una regla de negocio puede ser que "la edad de Person debe ser siempre mayor o igual a cero". Esto puede afirmarse, como se mostró en el ejemplo anterior.

2.7.3. Propiedades: Operaciones

Las operaciones pueden tener parámetros. Por ejemplo, un objeto `Vuelo` tiene el reservar en función de un pasajero.

Esta operación, puede accederser como sigue, por un `Vuelo` $unVuelo$ y un `Pasajero` $unPasajero$:

```
unVuelo.reservado(unPasajero)
```

La operación en sí podría definirse como una postcondición. Esta es una restricción que es estereotipada como <<postcondition>>. El objeto que es retornado por la operación es el resultado. Ejemplo:

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

Para referirse a una operación o método que no toma un parámetro, deben poner los paréntesis vacíos:

```
context TipoAvion inv:
self.getCapacidad() > 100
```

2.7.4. Propiedades: Association Ends and Navigation

Empezando desde un objeto específico, se puede navegar una asociación del diagrama de clases para referirse a otros objetos y sus propiedades. Para realizar esto, navegamos la asociación utilizando el association-end opuesto:

```
object.rolename
```

El valor de una expresión es el conjunto de objetos del otro lado de la asociación rolename. Si la multiplicidad del association-end tiene un máximo de uno ("0..1" o "1"), entonces el valor de la expresión es un objeto. Un ejemplo es:

```
context Company
inv: self.manager.isUnemployed = false
inv: self.employee->notEmpty()

context Vuelo
inv: self.avion.tipoAvion.capacidad > = false
inv: self.asientos->notEmpty()
```

En la primer invariante, *self.avion.tipoAvion* es de TipoAvion, ya que la multiplicidad de la asociación es uno. En la segunda, *self.asientos* es un Set de Asiento. Si la asociación en el diagrama de clases hubiera tenido ordered , el resultado sería un Sequence.

Las colecciones, como Sets, Bags y sequences son tipos predefinidos en OCL. Tienen muchas operaciones predefinidas y se acceden usando '->'seguido por el nombre de la propiedad. Por ejemplo:

```
 context Avion inv:
self.tipoAvion.capacidad > 30
```

Esto aplica la propiedad *size* en el conjunto *self.tipoAvion.capacidad*, y el resultado son los aviones con cierta capacidad.

```
 context Vuelo inv:
  self.asientos->notEmpty()
```

Esto aplica la propiedad *notEmpty* en el `Set` *self.asientos*. Evalúa a `true` si el set es no vacío, `false` en caso contrario.

Nombres de roles faltantes

Cuando el nombre de un rol falta en una de los ends de una asociación, se utiliza el nombre de la asociación, comenzando en minúscula.

Navegación sobre asociaciones con multiplicidad cero o uno

Como la multiplicidad de rol tipoAvion es uno, entonces *self.tipoAvion* es un objeto de tipo `TipoAvion`. Un objeto simple, también puede utilizarse como un `Set`, se comporta como si fuera un `Set` que contiene un único objeto. Por ejemplo:

```
context Avion inv:
  self.tipoAvion->size()=1
```

La subexpresión *self.tipoAvion* es utilizada como un `Set`, ya que "->" es utilizada para acceder a la propiedad *size* de un `Set`. Esta expresión evalúa a `true`.

El siguiente ejemplo muestra como puede utilizarse la propiedad de una colección.

```
context Avion inv:
  self.tipoAvion->foo
```

La subexpresión *self.tipoAvion* es utilizada como `Set`, ya que la flecha es usada para acceder a la propiedad foo de un `Set`. Pero la expresión es incorrecta, ya que foo no es una propiedad definida en `Set`.

```
context Avion inv:
  self.tipoAvion.capacidad > 30
```

La subexpresión *self.tipoAvion* es utilizada como `TipoAvion`, ya que el punto se utiliza para acceder a la propiedad capacidad de `TipoAvion`.

Para los próximos ejemplos se utilizará en parte el siguiente diagrama de clases:

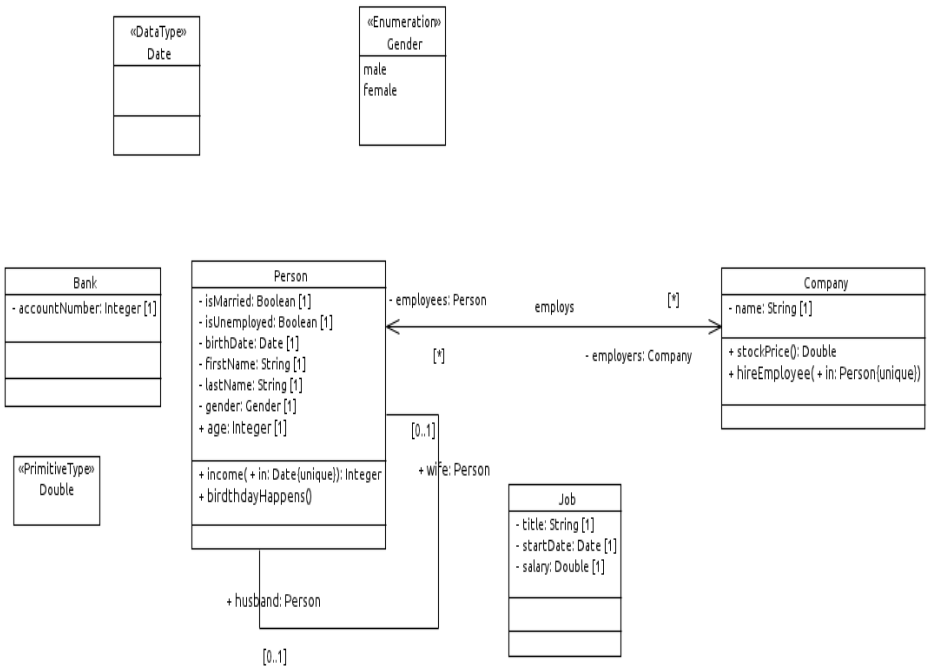

Figura 2.2: Ejemplo de Diagrama de clases

En el caso de una asociación opcional de multiplicidad 0..1, es muy útil para chequear cuando hay un objeto o no cuando se navega la asociación. En el ejemplo podemos escribir:

```
context Person inv:
 (self.gender=Sex::male and self.spouse->notEmpty())
   implies self.spouse.sex= Sex::female
```

Combinar propiedades

Las propiedades pueden combinarse para realizar expresiones mas complicadas. Una regla importante es que una expresión OCL siempre evalúa a un objeto específico de un tipo específico. Ejemplos:

```
[1] La edad de las personas casadas es >= 18
  context Person inv:
    self.wife->notEmpty() implies self.wife.age >= 18 and
    self.husband->notEmpty() implies self.husband.age >= 18
[2] una compañía tiene como mucho 50 empleados
```

```
context Company inv:
 self.employee->size() <= 50
```

2.7.5. Propiedades predefinidas en todos los objetos

Hay muchas propiedades que se aplican a todos los objetos y están definidas en OCL. Estas son:

```
oclIsTypeOf(t : OclType) : Boolean
oclIsKindOf(t : OclType) : Boolean
oclInState(s : OclState) : Boolean
oclIsNew() : Boolean
oclAsType(t : OclType) : instance of OclType
```

La operación *oclTypeOf* es verdadera si el tipo de **self** y t son el mismo. Por ejemplo:

```
context Person
inv: self.oclIsTypeOf( Person ) -- is true
inv: self.oclIsTypeOf( Company) -- is false
```

La propiedad anterior trata con el tipo directo de un objeto. La propiedad *oclIsKindOf* determina si t es tipo directo o uno de los supertipos del objeto. La operación `oclInState(s)` devuelve `true` si el objeto está en el estado s. Los valores para s son los nombres de los estados en la máquina de estado, adjunta al `Classifier` del objeto.

La operación *oclIsNew* evalúa a `true` si, usada como postcondición el objeto es creado durante la operación, es decir que no existía en el tiempo de la precondición.

2.7.6. Características de las clases

Todas las propiedades discutidas hasta ahora en OCL son propiedades de instancias de clases. Los tipos son predefinidos en OCL o definidos en el modelo de clases. Un tipo predefinido para cada tipo es `allInstances`, que devuelve un `Set` de todas las instancias del tipo en un tiempo específico cuando la expresión es evaluada. Por ejemplo si queremos asegurarnos que todas las instancias de `Person` tienen distinto nombre, puede escribirse:

```
context Person inv:
  Person.allInstances->forAll(p1, p2 |
  p1 <> p2 implies p1.name <> p2.name)
```

Person.allInstances es el conjunto de todas las personas y es del tipo `Set(Person)`.

2.8. Colecciones

El tipo `Collection` está predefinido en OCL. Es un tipo abstracto provisto de un conjunto de operaciones. Sus subtipos son colecciones concretas: `Set`, `Sequence`, y `Bag`. Un tipo `Set` es el conjunto matemático, no contienen elementos repetidos. Un tipo `Bag` es como un conjunto, con la diferencia que puede contener elementos duplicados, una o más veces. Un tipo `Sequence` es como un `Bag` en el cual los elementos están ordenados.

Los elementos de una colección son escritos separados por comas y encerrados entre llaves. El tipo de una colección es escrito antes de las llaves:

```
Set {1, 2, 5, 88}
Sequence {1, 3, 45, 2, 3}
Bag {1, 3, 4, 3, 5}
```

Para definir una secuencia de valores enteros consecutivos, se utiliza dos puntos seguidos. Es decir, *exp_entera*1 y *exp_entera*2, separado por ".." denota todos los enteros entre los valores *exp_entera*1 y *exp_entera*2 incluyendo también los extremos:

```
Sequence{ 1..(6 + 4) } y Sequence{ 1..10 } son equivalentes
a Sequence{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
```

2.9. Operaciones de Colecciones

OCL define varias operaciones sobre los tipos Colecciones. Por ejemplo:

Select y Reject La operación *select* especifica un subconjunto de una colección y su sintaxis es la siguiente:

```
collection->select(c:Tipo | expresión-lógica-con-c )
```

El resultado de la operación *select*, son todos los elementos de la colección, para los cuales la $expresi'on - l'ogica - con - c$ resultó verdadera. La variable c es llamada iterador. Cuando el *select* es evaluado, c itera sobre la colección y la $expresión - lógica - con - c$ es evaluada con cada c. El tipo de la variable iterador es opcional, con lo que nos quedaría otra forma equivalente:

```
collection->select( c | expresión-lógica-con-c )
```

Y se puede abreviar mediante:

```
collection->select(expresión-lógica)
```

Por ejemplo, las tres formas de escribir un *select* son las siguientes (el invariante especifica que en los asientos del avión, debe existir al menos uno que esté reservado):

```
context Vuelo inv:
self.asientos->select(a: Asiento | a.reservado == true)->notEmpty

context Vuelo inv:
self.asientos->select(a| a.reservado == true) ->notEmpty()

context Vuelo inv:
self.asientos->select(reservado == true) ->notEmpty()
```

El atributo $self.asientos$ es de tipo `Set(Asiento)`. El *select* toma cada asiento que esté reservado. La operación *reject* puede expresarse como un select con la expresión booleana negada. Es decir, que las dos siguientes expresiones son idénticas:

```
collection-> reject ( c:Tipo | expresión-lógica-con-c )
collection-> select( c:Tipo | not expresión-lógica-con-c )
```

Collect La operación *collect* se utiliza cuando queremos especificar una colección que deriva de otra colección, pero que contiene objetos diferentes a la colección original. El siguiente ejemplo especifica una colección con los n'meros de asientos del avión.

```
self.asientos ->collect(a| a.numero ), o simplemente:
self.asientos->collect( numero )
```

El resultado del *collect* es un `Bag` y no un `Set`. La colección resultante del `collect` tiene el mismo tamaño que la original. Una abreviatura para el *collect*, que hace a la expresión OCL más legible, es:

```
self.asientos.numero
```

En general, cuando aplicamos una propiedad a una colección, entonces, automáticamente interpretará como un *collect* sobre cada elemento de la colección con la propiedad especificada. Es decir, que las siguientes expresiones son equivalentes:

```
colección.nombrePropiedad
colección ->collect(nombrePropiedad)
```

ForAll - Exists La operación *forAll* permite especificar una expresión booleana que debe valer para todos los elementos de una colección:

```
collection->forAll ( e : Tipo | expresión-lógica-con-e )
```

El resultado es verdadero si la $expresi'on - l'ogica - con - e$ es verdadera para todos los elementos de la colección. Si la $expresi'on - l'ogica - con - e$ es `false` para algún elemento, entonces la expresión completa evalúa a `false`. Por ejemplo:

```
 context Avion inv:
self.asientos->forAll(a:Asiento | a.reservado == true)
```

Este invariante se satisface si todas los asientos del avión están reservados. También se podría usar más de un iterador. Esto es un *forAll* sobre el producto cartesiano de la colección y ella misma. Por ejemplo:

```
 context Avion inv:
self.asientos->forAll(a1,a2 | a1 <> a2 implies a1.numero <> a2.numero)
```

La operación *exists* permite especificar una expresión booleana que debe valer para al menos un objeto en la colección:

```
collection-> exists ( e : Tipo | expresión-lógica-con-e )
```

El resultado es `true` si la $expresi'on - l'ogica - con - e$ es `true` para uno o más elementos de la colección. Si la $expresi'on - l'ogica - con - e$ es `false` para todos los elementos, entonces la expresión completa evalúa a `false`. Por ejemplo:

```
 context Avion inv:
self.asientos->exists (a: Asiento | a.reservado == true)
```

Los invariantes evalúan verdadero si algún asiento del avión está reservado.

Operación Iterate La operación iterate es un poco mas complicada, pero es muy genérica. Las operaciones *reject*, *select*, *forAll*, *exists*, *collect*, pueden ser descriptas en términos de iterate. Un acumulador va construyendo un valor iterando la colección.

```
collection->iterate( elem : Type; acc : Type = <expression> |
expression-with-elem-and-acc )
```

La variable *elem* es el iterador, como es la definición de *select*, *forAll*, etc. La variable *acc* es el acumulador. El acumulador toma un valor inicial <expression>.

Cuando el iterate es evaluado, elem itera sobre la colleción y la $expression - with - elem - and - acc$ se evalúa para cada elemento. Luego de cada evaluación de $expression - with - elem - and - acc$, el valor se asigna a *acc*. De esta forma, el valor de *acc* se construye durante la iteración de la colección. La operación *collect* descripta en términos de iterate es de la forma:

```
collection->collect(x : T | x.property)
-- is identical to:
collection->iterate(x : T; acc : T2 = Bag{} |
acc->including(x.property))
```

Iteradores en Operaciones de colecciones Las operaciones de colecciones que toman una `OclExpression` como parámetro pueden tener una declaración de iterador opcional. Para cada operación llamada op, las opciones de sintaxis son:

```
collection->op( iter : Type | OclExpression )
collection->op( iter | OclExpression )
collection->op( OclExpression )
```

2.9.1. Valores previos en postcondiciones

En una postcondición, la expresión puede referirse a dos conjuntos de valores para cada propiedad de un objecto:

- el valor de una propiedad al principio de una operación o método.ro

El valor de una propiedad en una postcondición es el valor al finalizar la operación. Para referirse al valor de una propiedad al principio de una operación, debe agregarse al nombre de la propiedad la palabra clave **@pre**:

```
context Person::birthdayHappens()
  post: age = age@pre + 1
```

La propiedad age se refiere a la propiedad de una instancia de Person la cual ejecuta la operación. La propiedad **@pre** se refiere al valor de la propiedad age de Person que ejecuta la operación, al principio de la operación.

```
context Company::hireEmployee(p : Person)
  post: employees = employees@pre->including(p) and
  stockprice() = stockprice@pre() + 10
```

El ejemplo anterior puede especificarse por una postcondición y una precondición:

```
context Company::hireEmployee(p : Person)
pre : not employee->includes(p)
post: employees->includes(p) and
stockprice() = stockprice@pre() + 10
```

2.10. Tipos predefinidos en OCL

Esta sección contiene todos los tipos estándar definidos con OCL, incluyendo todas las propiedades definidas en esos tipos. Su firma y una descripción de su semántica definen cada propiedad. Con la descripción, la palabra reservada 'result'se usa para refererirse al valor que resulta de evaluar la propiedad. En varios lugares, las postcondiciones son usadas para describir propiedades del resultado. Cuando hay mas de una postcondición, todas ellas deben ser true.

2.10.1. Tipos básicos

Los tipos básicos son `Integer`, `Real`, `String`, y `Boolean`, como vimos en la sección 2.6. Se complementan con `OclExpression`, `OclType` y `OclAny`. Para ver el conjunto de operaciones ver el apendice B Librería estándar de OCL.

OclType Todos los tipos definidos en un modelo UML, o predefinidos con OCL, tienen un tipo. Este tipo es una instancia del tipo OCL llamado `OclType`. El acceso a este tipo permite al diseñador limitar el acceso a meta-nivel del modelo. Esto puede ser útil para los diseñador avanzado.

OclAny En el contexto de OCL, el tipo `OclAny` es el supertipo de todos los tipos en el modelo y los tipos básicos predefinidos de OCL. La tipos Collection predefinidos de OCL no son subtipos de `OclAny`. Las propiedades de `OclAny` están disponibles en cada objeto en todas las expresiones OCL.

Todas las clases en un modelo UML heredan todas las propiedades definidas en `OclAny`. Para evitar conflictos de nombres entre las propiedades en el modelo y las propiedades heredadas de `OclAny`, todos los nombres en las propiedades de `OclAny`, empiezan con ócl'. También puede usarse la operación *oclAsType()* para referirse explícitamente a las propiedades `OclAny`.

OclState El tipo `OclState` se usa como parámetro para la operación *oclInState*. No hay propiedades definidas para `OclState`. Se especifica un `OclState` usando el nombre del estado, como en una máquina de estados.

OclExpression Cada expresión OCL es un objeto en el contexto de OCL. El tipo de expresión es `OclExpression`. Este tipo y sus propiedades se usan para definir la semántica de las propiedades que toman una expresión como uno de sus parámetros: *select*, *collect*, *forAll*, etc. Una `OclExpression` incluye un iterador opcional y un acumulador opcional.

Real El tipo `Real` de OCL representa el concepto matemático de real. Notar que `Integer` es una subclase de `Real`, por lo que por cada parámetro de tipo `Real`, puede usarse un integer como el parámetro actual.

Integer EL tipo `Integer` de OCL representa el concepto matemático de integer.

String El tipo `String` de OCL representa las cadenas ASCII.

Boolean El tipo `Boolean` de OCL representa los valores `true`/`false`.

Enumeration El tipo `Enumeration` de OCL representa las enumeraciones definidas en un modelo UML.

Para un resumen de las operaciones de los tipos descriptos anteriormente ver el apéndice B Librería estándar de OCL.

2.11. Resumen

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

- El por qué del lenguaje. Como un diagrama UML no da la suficiente información necesaria para una especificación, se necesitan escribir restricciones adicionales sobre los modelos. A veces se hacen en un lenguaje natural, pero puede resultar en ambigüedades. El lenguaje OCL ha sido desarrollado para completar la deficiencia descripta, ya que permite incrementar la precisión de los modelos UML, es decir permite expresar restricciones semánticas del sistema que no se pueden expresar a partir de una notación gráfica.
- Se vieron las restricciones OCL, invariantes, definiciones, precondiciones y postcondiciones.
- Los tipos OCL, `Boolean`, `Integer`, `Real` y `String`.
- Los objetos y propiedades, que pueden ser un `Attribute`, un `AssociationEnd`, un `Operation` con *isQuery* en `true`, o un `Method` con *isQuery* en `true`.
- Las colecciones en OCL son un tipo abstracto provisto de un conjunto de operaciones. Sus subtipos son colecciones concretas: `Set`, `Sequence`, y `Bag`. También se vieron las operaciones de las colecciones: *select*, *reject*, *collect*, *forAll*, *exists*, *iterate*.

Capítulo 3

JML

3.1. Introducción

JML es un lenguaje que especifica el comportamiento e interfaz de clases y métodos Java, que comenzó en Iowa State University por el grupo de Gary Leavens. JML provee una semántica para describir de forma 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 formal rigurosa a la vez que se hace accesible a todo programador de Java. Hay varias herramientas que se sirven de las especificaciones de comportamiento del JML.

JML utiliza pre y postcondiciones e invariantes de la lógica de Hoare. Permite aplicar la metodología de diseño por contrato (Design by Contract (DBC)). La idea principal de DBC es que las clases y sus clientes tengan un contrato entre ellas [42]. 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 [38].

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 del código basadas en la lógica de Hoare.

3.2. Sintaxis

Las especificaciones JML pueden estar en los archivos *.java* o en archivos *.jml* separados. Estas especificaciones se agregan como anotaciones en comentarios, por lo cual serán ignorados por el compilador de Java. Pueden agregarse entre `/*@ . . . @*/` o luego de `//@`.[39, 12]

JML provee las siguientes palabras clave:

1. **invariant**: Define un invariante para la clase.
2. **requires**: Define una precondición en el método.
3. **ensures**: Define una postcondición en el método.
4. **signals**: Señala excepciones.
5. **signals_only**: Señala excepciones.
6. **assignable**: Define a que campos se los puede asignar en el método siguiente.
7. **pure**: Define un método libre de efectos laterales.

8. **also**: Utilizado para combinar casos de especificación y para definir que un método hereda las especificaciones de sus supertipos.
9. **assert**: Define una aserción JML.
10. **spec_public**: Declara una variable privada o protegida como pública para especificaciones.
11. **non_null**: Declara un atributo que no puede ser **null**.
12. **nullable**: Declara un atributo que puede ser **null**.

JML también provee las siguientes expresiones:

1. **\result**: Es un identificador para el valor de retorno del método.
2. **\old(<expression>)**: Es un modificador para referenciar al valor de <expression> al momento de comenzar el método.
3. **a ==> b**: a implica b.
4. **a <== b**: b implica a.
5. **a <==> b**: a sí y solo sí b.
6. **\fresh**: Asegura que el objeto fue recién alocado y no alocado en un estado anterior.
7. **\max**: Retorna el máximo entre un conjunto de enteros.
8. **\min**: Retorna el mínimo entre un conjunto de enteros.
9. **\product**: Retorna el producto entre un conjunto de enteros.
10. **\sum**: Retorna la suma entre un conjunto de enteros.
11. **\num_of**: Retorna la cantidad de valores para cierta variable, para la cual es `true` en cierto rango.

JML también soporta cuantificadores:

1. **(\forall <dec1>; <range-exp>; <body-exp>)**: El cuantificador universal.
2. **(\exists <dec1>; <range-exp>; <body-exp>)**: El cuantificador existencial.

También soportan la sintaxis básica de Java para las operaciones básicas *and*, *or* y *not*. Las anotaciones JML también tienen acceso a los objetos Java, a los métodos de los objetos y los operadores que están dentro del alcance del método que está siendo anotado y que tienen la visibilidad apropiada. Estos se combinan para proveer una especificación formal de las propiedades de clases, campos y métodos.

Los contratos en JML están definidos por las siguientes cláusulas:

1. Las precondiciones son escritas con la cláusula **requires**.
2. Las postcondiciones son escritas con la cláusula **ensures**.
3. Las invariantes son escritas con la cláusula **invariant**.

3.3. Explicación y ejemplos

3.3.1. Invariantes, pre y postcondiciones

Las invariantes deben ser mantenidas por todos los métodos de la clase, incluso cuando se lanza una excepción.

Las invariantes son incluidas implícitamente en todas las pre y postcondiciones. Para los constructores las invariantes son incluidas sólo en las postcondiciones y no en las precondiciones. Es por eso que los constructores aseguran las invariantes pero no requieren de ellas.

Las invariantes documentan las decisiones de diseño y hacen que el código sea mas fácil de entender.

JML utiliza la cláusula **requires** para especificar la obligación del cliente y utiliza la cláusula **ensures** para especificar la obligación del programador.

Para las postcondiciones pueden usarse la palabra clave **\old** para hacer referencia al valor de un campo antes de la ejecución del método. O puede usarse la palabra clave **\result** para hacer referencia al valor de retorno del método.

Ambas palabras clave son herramientas necesarias y útiles para especificar postcondiciones. Un ejemplo se muestra en el Fragmento de código fuente 3.1.

```
1  public class Account
2  {
3    private /*@ spec_public */ int balance;
4    //@ public invariant balance >= 0;
5
6    /*@ ensures this.balance == 0; @*/
7    public Account(){
8      this.balance = 0;
9    }
10
11   /*@ requires amount >= 0;
12     @ ensures this.balance == amount; @*/
13   public Account(int amount){
14     this.balance = amount;
15   }
16
17   /*@ requires amount > 0;
18     @ ensures this.balance == \old(this.balance) + amount; @*/
19   public void deposit(int amount){
20     this.balance = this.balance + amount;
21   }
22
23   /*@ requires amount > 0 && amount <= this.balance;
24     @ ensures this.balance == \old(this.balance) - amount; @*/
25   public void withdraw(int amount){
26     this.balance = this.balance - amount;
27   }
28
29   /*@ requires amount > 0 && amount <= this.balance;
30     @ ensures this.balance == \old(this.balance) - amount &&
31       account.balance == \old(account.balance + amount); @*/
32   public void transfer(Account account, int amount){
33     this.withdraw(amount);
34     account.deposit(amount);
35   }
36 }
```

Fragmento de código fuente 3.1: Ejemplo de invariantes pre y postcondiciones

Explicación de invariantes, pre y postcondiciones

Lo primero que tenemos en el ejemplo es una invariante de clase **public invariant balance >= 0;**. Significa que el **balance** debe ser mayor o igual a cero.

Para el método `public Account()` tenemos la precondición **requires this.balance == 0;** que significa que el **balance** de la cuenta debe ser cero.

Para el método `public Account(int amount)` tenemos la precondición **requires amount >= 0;** que significa que el **amount** debe ser mayor o igual que cero y como postcondición tenemos **ensures this.balance == amount;** que significa que luego de ejecutarse el método el *balance* tendrá el mismo valor que el *amount*.

Para el método `public void deposit(int amount)` tenemos la precondición **requires amount > 0;** que significa que el *amount* debe ser mayor a cero y como postcondición tenemos **ensures this.balance == \old (this.balance) + amount;** que significa que el valor de *balance* luego de ejecutarse el método será el valor de **balance** antes de ejecutarse sumado al valor de *amount*.

Para el método `public void withdraw(int amount)` tenemos la precondición **requires amount > 0 && amount <= this.balance;** que significa que el valor de *amount* debe ser mayor a cero y además debe ser menor o igual al valor de *balance* y como postcondición tenemos **ensures this.balance == \old(this.balance) − amount;** que significa que el valor de *balance* luego de ejecutarse el método será el valor de *balance* antes de ejecutarse restado al valor de *amount*.

Para el método `public void transfer(Account account, int amount)` tenemos la precondición **requires amount > 0 && amount <= this.balance;** que significa que el valor de *amount* debe ser mayor a cero y además debe ser menor o igual al valor de *balance* y como postcondición tenemos **ensures this.balance == \old(this.balance) − amount && account.balance == \old(account.balance + amount);**. La primer parte antes del && significa que el valor de *balance* luego de ejecutarse el método será el valor de *balance* antes de ejecutarse, restado al valor de *amount*. La segunda parte de && significa que el valor de *balance* luego de ejecutarse el método será el valor de *balance* antes de ejecutarse sumado al valor de *amount*.

3.3.2. Excepciones

Además de las postcondiciones normales, JML soporta postcondiciones excepcionales. Éstas se escriben con la cláusula **signals**.

Las excepciones mencionadas en la cláusula **throws** son permitidas por defecto. Por ejemplo la cláusula `signals` por defecto es `signals (Exception) true;`. Para descartarla se puede añadir explícitamente `signals (Exception) false;` o utilizar `normal_behavior`. Un ejemplo se muestra en Fragmento de código fuente 3.2.

```
1 /*@ requires amount >= 0;
2 @ ensures balance <= \old(balance);
3 @ signals (BankException) balance == \old(balance);
4 @*/
5 public debit(int amount) throws BankException {
6    if (amount > balance) {
7      throw (new BankException("No way"));}
8    balance = balance - amount;
9 }
```

Fragmento de código fuente 3.2: Ejemplo de excepciones

En el ejemplo se agrega la expresión `signals` para especificar la excepción. Notar que cuando se levante una excepción, el valor de balance debería seguir siendo el que tenía anteriormente. Un ejemplo se muestra en el Fragmento de código fuente 3.3.

```
1 /*@ normal_behavior
2 @ requires amount >= 0 && amount <= balance;
3 @ ensures balance <= \old(balance);
4 @*/
5 public debit(int amount) throws BankException{
6   if (amount > balance) {
7     throw (new BankException("No way"));}
8    balance = balance - amount;
9 }
```

Fragmento de código fuente 3.3: Ejemplo de normal behavior

En el ejemplo no se utiliza `signals` sino que se utiliza `normal_behavior` para especificar el comportamiento normal del método.

3.3.3. Assignable

Para los métodos que no son puros, las propiedades pueden especificarse utilizando la cláusula **assignable**. Este cláusula dice los campos que están permitidos modificar. Un ejemplo se muestra en el Fragmento de código fuente 3.4.

```
1 /*@
2 @ requires amount >= 0;
3 @ assignable balance;
4 @ ensures balance == \old(balance) - amount;
5 @*/
6 public void debit(){}
```

Fragmento de código fuente 3.4: Ejemplo de Assignable

3.3.4. Pure

Los métodos **pure**, son aquellos que terminan sin efectos colaterales. Éstos pueden usarse en las anotaciones JML. [40] Un ejemplo se muestra en el Fragmento de código fuente 3.5.

```
1  /*@ pure @*/ int getBalance (){
2    return balance;
3  };
4
5 //@ requires amount < getBalance();
6  public debit (int amount){...}
```

Fragmento de código fuente 3.5: Ejemplo de Pure

3.3.5. Also

La palabra clave **also** se utiliza para las especificaciones de métodos que son heredados en subclases. Un ejemplo se muestra en el Fragmento de código fuente 3.6.

```
1  class Parent {
2  //@ requires i >=0;
3  //@ ensures \result >= i;
4    int m(int i) {...}
5  }
6  class Child extends Parent {
7  //@ also
8  //@ requires i <= 0;
9  //@ ensures \result <= i;
10    int m(int i) {...}
11 }
```

Fragmento de código fuente 3.6: Ejemplo de Also

3.3.6. Assert

Una aserción o anotación es una cláusula lógica insertada en medio de un programa. El objetivo de una aserción es representar una condición que debe ser cierta en un punto del programa. Esta condición puede ser una condición típica de Java, escrita usando los operadores lógicos && y | |. La cláusula de JML assert sirve para indicar estas condiciones o aserciones en cualquier lugar de un programa. Un ejemplo se muestra en el Fragmento de código fuente 3.7.

```
1 public int max(){
2 if (x >= y)
3   z = x;
4 else
5   z = y;
6 return z;
7 /*@ assert z >= x && z >= y && (z == x || z == y);@*/
```

Fragmento de código fuente 3.7: Ejemplo de Assert

En el ejemplo puede verse que la condición que debe ser cierta se indica directamente a continuación de la cláusula **assert**, y debe estar terminada con el carácter ";". Justo delante de la cláusula aparecen los caracteres **//@** que indican al compilador que la línea contiene una cláusula de JML.

Si compilamos este segmento de código con el compilador de JML, se generará un programa que además de ejecutar las instrucciones escritas en Java, también comprobará si la condición indicada con **assert** es cierta. Si esta condición no fuese cierta (por ejemplo, por modificación del código para realizar alguna otra función diferente) entonces el programa se detendría y se visualizaría un error en tiempo de ejecución indicando que dicha aserción no se cumple.

3.3.7. Spec_public

La visibilidad de los campos puede suavizarse utilizando la palabra clave **spec_public**. Un ejemplo se muestra en le Fragmento de código fuente 3.8

3.3.8. Non_null y nullable

Muchas de las invariantes y las precondiciones son sobre referencias que no deben ser null, es por eso que JML lo adopta como por defecto y sólo los campos que pueden ser **null** deben anotarse. Un ejemplo se muestra en el Fragmento de código fuente 3.9.

```
1 public class ePurse{
2   private /*@ spec_public @*/ int balance;
3
4   //@ ensures balance <= \old(balance);
5   public debit(int amount){}
6 }
```

Fragmento de código fuente 3.8: Ejemplo de Spec_Public

```
1 int[] a; //@ invariant a != null;
2
3 /*@ non_null @*/ int[] a;
4
5 /*@ nullable @*/ int[] b;
```

Fragmento de código fuente 3.9: Ejemplo de Non_null y Nullable

3.3.9. Fresh

El operador **\fresh** asegura que un objeto no estaba alocado en el pre estado. Por ejemplo **\fresh(x)** asegura que x no es `null` y que los objetos vinculados a estos identificadores no estaban alocados antes. Se utiliza sólo en postcondiciones. No puede utilizarse en la especificación de un constructor, ya que el operador **new** de Java aloca el almacenamiento para el objeto, y el único trabajo del constructor es inicializar ese almacenamiento.

3.3.10. typeof

El operador **\typeof** retorna el tipo dinámico más específico del valor de una expresión [41]. El significado de **\typeof**(E) es indefinido si E es **null**. Si E tiene un tipo estático que es un tipo por referencia (no es un tipo primitivo) entonces **\typeof**(E) es lo mismo que `E.getClass()`. Por ejemplo, si c es una variable del tipo estático `Collection` que apunta a un objeto de clase `HashSet`, entonces **\typeof**(c) es `HashSet.class` que es lo mismo que decir **\type**(`HashSet`). Si E tiene un tipo que no es un tipo por referencia, entonces **\typeof**(E) representa la instancia de `java.lang.Class` que representa su tipo estático. Por ejemplo, **\typeof**(**true**) is `Boolean.TYPE` que es lo mismo que **\type**(`boolean`). Así una expresión de la forma **\typeof**(E) tiene tipo **\TYPE**, que JML considera del mismo tipo que `java.lang.Class`.

3.3.11. type

El operador **\type** puede ser utilizado para introducir literales de tipo **\TYPE** en expresiones. Una expresión de la forma **\type**(`T`) donde `T` es un nombre de tipo tiene el tipo **\TYPE**. Como en JML **\TYPE** es lo mismo que `java.lang.Class`, una expresión de la forma **\type**(`T`) significa lo mismo que `T.class`, si `T` es un tipo por referencia. Si `T` es un tipo primitivo, entonces **\type**(`T`) es equivalente al valor del campo **TYPE** del correspondiente tipo por referencia. De esta forma, **\type**(`boolean`) es equivalente a `Boolean.TYPE`.

3.3.12. Subtipo

El operador relacional <: compara dos tipos por referencia y retorna verdadero cuando el tipo del lado izquierdo es un subtipo del tipo del lado derecho [41]. Aunque la notación puede sugerir otra cosa, el operador es también reflexivo. Un tipo puede compararse con <: contra sí mismo. En una expresión de la forma E_1 <: E_2, tanto E_1 como E_2 deben tener tipo **\TYPE**, como en JML **\TYPE** es lo mismo que `java.lang.Class` la expresión E_1 <: E_2 significa lo mismo que E_1`.isAssignableFrom(`E_2`)`. Como resultado, los tipos primitivos no son subtipos de `java.lang.Object`, ni entre ellos, aunque lo son de ellos mismos, e.g. `Integer.TYPE` <: `Integer.TYPE` es **true**.

3.3.13. Cuantificadores Máximo, Mínimo, Producto y Suma

Los cuantificadores **\max**, **\min**, **\product**, **\sum**, son cuantificadores generalizados que retornan el máximo, mínimo, producto o suma de los valores de las expresiones dadas, donde las variables se encuentran en el rango dado. La expresión en el cuerpo debe ser un tipo numérico primitivo, como `int` o `double`; Ejemplos:

1. **(\max int i; 0 <= i && i < 5; i)** retorna como resultado 4.
2. **(\min int i; 0 <= i && i < 5; i-1)** retorna como resultado -1.
3. **(\product int i; 0 < i && i < 5; i)** retorna como resultado 24 (1 * 2 * 3 * 4)
4. **(\sum int i; 0 <= i && i < 5; i)** retorna como resultado 10 (0 + 1 + 2 + 3 + 4)

3.3.14. num_of

El cuantificador **\num_of** retorna el número de valores que son verdaderos en el rango de la expresión. El cuerpo debe ser de tipo boolean, y toda la expresión cuantificada es de tipo long. Por ejemplo, si tenemos (\num_of int i; 0 <= i && i < 5; i*2 < 6) retorna como resultado 3 (Los valores 0, 1 y 2 son los que cumplen la condición)

3.3.15. Cuantificadores

Los cuantificadores **\forall** y **\exists**, son los cuantificadores universal y existencial. Por ejemplo: (\forall int i,j; 0 <= i && i < j && j < 10; a[i] < a[j]), dice que los valores a[0] ... a[9] están ordenados.

El cuerpo de los cuantificadores universal y existencial deben ser del tipo boolean. El tipo de una expresión universal o existencial es de tipo boolean. Cuando el rango del predicado no es satisfacible, el valor del **\forall** es `true` y el valor del **\exists** es `false`. Por ejemplo:

1. (\forall int i; 0 < i && i < 0; 0 < i) retorna `true`.
2. (\exists int i; 0 < i && i < 0; 0 < i) retorna `false`.

3.4. Model fields y cláusula represents

JML provee model fields para permitir a la especificación abstraerse de los detalles de implementación. Los model fields en JML desempeñan el rol que los valores abstractos desempeñan para los tipos de datos abstractos tradicionales. La idea es que los clientes de una clase puedan usar una especificación abstracta en términos de model fields, y que solo en la implementación de la clase debemos conocer como los model fields abstractos se relacionan con la representación concreta actual. [23, 7]

Se muestra un ejemplo simple de una especificación JML con un model field para la Java/JML Interface: `MyIntInterface` en el siguiente Fragmento de código fuente 3.10

Declara un model field *myint* que representa un valor entero entre 0 y 512. Un model field es un campo imaginario que sólo puede ser usado para propósitos de especificación. En nuestro ejemplo, el model field *myint* es usado en la especificación del método `succ`. La especificación consta de varias cláusulas:

1. La cláusula **requires** especifica la precondición.
2. La claúsula **assignable** lista los campos o model fields que pueden ser modificados.
3. La cláusula **ensures** especifica la postcondición que debe satisfacerse cuando succ termine normalmente.
4. La claúsula **invariant** especifica la invariante de clase, que se incluye implícitamente en todas las pre y post condiciones.

```
1  interface MyIntInterface {
2   /*@ model int myint;
3    invariant 0 <= myint && myint < 512 @*/
4   /*@ requires true;
5    assignable myint;
6    ensures
7     myint == (\old(myint) + 1) \% 512; @*/
8
9    void succ();
10 }
```

Fragmento de código fuente 3.10: Java/JML Interface: MyIntInterface

Para un cliente (alguien que está utilizando objetos del tipo `MyIntInterface`) que está tratando con model fields, es diferente a alguien que está implementando una clase que implementa `MyIntInterface`:

1. Esencialmente, los clientes pueden tratar a un model field como un campo Java, y olvidarse que no en un campo real, pero hay cierta representación escondida detrás de un model field. Desafortunadamente esto es cierto si la representación está propiamente encapsulada, y no hay forma en la cual pueda alterarse la representación del model field *myint*.

2. Los implementadores deben especificar como el model field abstracto se representa en la implementación concreta. En JML, esto se realiza con las claúsulas **represents** y **depends**. Éstas determinan cuales son las obligaciones seguidas por la especificación abstracta. Hay dos tipos de cláusula **represents**:

   a) El primero (que utiliza la notación <-) da una definición explícita del model field en términos de fields reales. En el ejemplo del Fragmento de código fuente 3.11 se muestra la utilización de la notación **<-**.

```
1  class MyIntImpl1 implements MyIntInterface {
2   /*@ represents myint <- (f ? 256 : 0) + (b & 0xFF);
3     @ depends myint <- b,f; @*/
4    byte b;
5    boolean f;
6
7    public void succ() {
8      f = (b == -2) ^ f;
9      b = (byte)(b + 1);
10   }
11 }
```

Fragmento de código fuente 3.11: Una implementación usando <-: MyIntImpl1

   b) El segundo (usando la notación
   such_that) es de tipo mas general sólo especifica una relación de representación entre el model field y el campo real. En el ejemplo del Fragmento de código fuente 3.12 se muestra la utilización de **such_that**.

Model fields con representación de funciones (usando <-)

En JML una definición explícita de un model field en términos del campo real puede ser dada usando una claúsula **represents** de la siguiente forma:

```
//@ represents myint <- ...
```

```
1  class MyIntImpl2 implements MyIntInterface {
2 /*@ represents myint \such_that (!f ==> myint == (b & 0xFF)) &&
3  @ ( f ==> myint == (b & 0xFF) + 256);
4  @ depends myint <- b,f;
5  @*/
6    byte b;
7    boolean f;
8    void succ() {
9    ...
10   }
11 }
```

Fragmento de código fuente 3.12: Una implementación usando such_that : MyIntImpl2

La expresión a la derecha del <-, en este caso, una expresión intera, da la representación de la función para el model field. Por ejemplo en el Fragmento de código fuente 3.11 y `boolean` y un `byte` son usados para implementar *myint*. Juntos, un `boolean` y un `byte` pueden representar un valor numérico de 9 bits, por ejemplo un entero entre 0 y 512. El `boolean` f en el Fragmento de código fuente 3.11, es el bit mas significante. La claúsula **represents** en el mismo ejemplo da una definición explícita del model field *myint* en términos de la representación concreta en el campo b y f.

Model fields como macros

Para verificar que la implementación concreta de `MyIntImpl1` dada en el Fragmento de código fuente 3.11 conoce a la especificación abstracta de `MyIntInterface` dada en el Fragmento de código fuente 3.11 es muy sencillo. Básicamente, podemos tratar al model field *myint* en la especificación abstracta de `MyIntInterface` como un macro y expandirlo para obtener la obligación concreta para `MyIntImpl1`. Más específicamente:

1. Se reemplazan todas las ocurrencias de *myint* en claúsulas asignables por la claúsula **depends** especificada en `MyIntImpl1`, por ejemplo por "b,f".

2. Reemplazamos todas las demás ocurrencias de *myint* por la claúsula **represents** especificada en `MyIntImpl1`, por ejemplo por "(f ? 256 : 0) + (b & 0xFF)"

Model fields con representación de relación (usando such_that)

El Fragmento de código fuente 3.12 muestra la clase `MyIntImpl2`, la cual tiene la misma implementación que la clase `MyIntImpl1`, pero una claúsula **represents** diferente. La claúsula **represents** en el Fragmento de código fuente 3.12 usa una cláusula
such_that para especificar una relación entre el model field x y el campo real f y b. En este ejemplo particular, la relación es uno a uno: por cualquier valor dado de b y f existe exactamente un valor de *myint* en la relación de representación. De hecho, la representación de relación es simplemente una representación de función especificada en el Fragmento de código fuente 3.13, escrita de forma diferente.

En general una representación de relación no necesita ser una función. Por ejemplo, la representación de relación del Fragmento de código fuente 3.13 es una relación parcial y es de muchos a uno: dados los valores de los campos concretos j y k, puede haber muchos valores, o ninguno, para el model field x que hay en la representación de la relación.

Tratar con estas representaciones de relaciones, que no son uno a uno, es mas difícil que tratar con las funcionales. Pueden distinguirse tres tipos de claúsulas **rep**:

1. funciones **rep**, que son escritas utilizando la notación <-.

```
1  class A {
2  int j,k;
3  //@ model int x;
4  //@ represents x \such_that (j <= x && x <= k);
5  /*@ requires true;
6  ensures false; */
7    public void m() {
8      j = 1;
9      k = 0;
10     // there is no x such that j <= x <= k !
11   }
12   public void n() {
13     j = 10;
14     k = 12;
15     //@ assert x == 10;
16   }
17 }
```

Fragmento de código fuente 3.13: Ejemplo clase A

2. relaciones **rep** funcionales, que son escritas utilizando **such_that**, que especifican una función uno a uno.

3. relaciones **rep** correctas, que son escritas usando **such_that** como relaciones correctas.

Una razón para utilizar el sugundo tipo de claúsula **rep** en lugar de la primera es por conveniencia, algunas personas encuentran más fácil leer la claúsula **rep** en Fragmento del código fuente 3.12 antes que la del Fragmento del código fuente 3.11.

3.5. Comprensión de conjuntos

La sintaxis de las expresiones de comprensión de conjuntos es como sigue:

```
set-comprehension ::= { [ bound-var-modifiers ] type-spec
  quantified-var-declarator '|' postfix-expr && predicate }
```

La notación de comprensión de conjuntos, puede utilizarse para definir en forma breve, conjuntos. El significado de una *new-expr* con un sufijo de comprensión de conjuntos, como `new ST { T x | s.has(x) && P (x) }`, es un subconjunto de tipo `ST`, del conjunto s, el cual contiene sólo los elementos x que están en s y que $p(x)$ es `true`.

El ejemplo siguiente es el `JMLObjectSet` que es el subconjunto de los objetos `Integer` non-null encontrados en el conjunto $myIntSet$ cuyos valores están entre 0 y 10 inclusive.

```
new JMLObjectSet {Integer i | myIntSet.has(i) &&
  i != null && 0 <= i.getInteger() && i.getInteger() <= 10 }
```

La sintaxis de JML limita la comprensión de conjuntos por lo que la barra vertical (|) es siempre una invocación de método de algún conjunto sobre la variable declarada. En la práctica, o se empieza con algún conjunto que se tiene a mano, o se empieza desde conjuntos que contienen objetos de tipos primitivos que se encuentran en `org.jmlspecs.models.JMLObjectSet` y `org.jmlspecs.models.JMLValueSet`. El tipo de la expresión es el tipo debe ser `JMLObjectSet` o `JMLValueSet`.

La variable ligada, cuyo alcance es el comprensión de conjunto, puede no entrar en conflicto con las variables locales existentes, pero puede ocultar campos estáticos y de instancia. El tipo de la variable ligada es utilizado para restringir los objetos que forman parte del conjunto resultado; si el conjunto llamado en *postfix-expr* contiene objetos que no son asignables a la variable ligada, ellos son están en el conjunto resultado. Las dos siguientes expresiones de comprensión de conjuntos resultan en conjuntos idénticos:

```
new JMLObjectSet {Integer i | s.contains(i) && 0 < i.intValue() }
new JMLObjectSet {Object i | s.contains(i) && i instanceof Integer &&
0 < ((Integer) i).intValue() }
```

3.6. Anotaciones de sentencias

JML define anotaciones para instrucciones que pueden ser agregadas entre instrucciones Java en el cuerpo de un método, constructor o bloque de inicialización. En JML un ciclo (loop) puede anotarse con uno o más invariantes de ciclo, y uno o más funciones variantes. En 3.14 hay un ejemplo en el interior del método *sumArray*. En el ejemplo hay un ciclo **while** con dos invariantes expresados con la palabra clave **maintaining**, y una única función variante, expresada con la palabra clave **decreasing**. Los invariantes y la función variante se escriben antes del ciclo. El primer invariante de ciclo describe el rango que la variable i puede tomar, y el segundo relaciona i y el valor de *sum*. Al finalizar la ejecución del ciclo, valen la negación de la expresión de testeo del ciclo y los invariantes. Esto se muestra con las aserciones luego del ciclo.

3.6.1. Invariantes de ciclos

Un ciclo puede especificar uno o más invariantes, utilizando las palabras clave **loop_invariant**, **loop_invariant_redundantly**, **maintaining** y **maintaining_redundantly**. Una expresión de invariante es utilizada para ayudar a probar la corrección parcial de un ciclo.

El significado de un ciclo, que no utiliza la instrucción **break** que termina la ejecución del ciclo, tal como:

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

es el siguiente

```
while (true) {
  //@ assert J;
  if (!(B)) { break; }
  S
}
```

Por lo tanto, el invariante es válido al comienzo de cada iteración del ciclo.

La regla para deducir que es verdadero luego de la ejecución del ciclo puede ser expresada simplemente si el ciclo no contiene ninguna instrucción **break**, y si la condición del ciclo B, es tanto una expresión Java y una expresión de especificación JML (esto significa que B no tiene efectos laterales). Para tales ciclos, la regla es que, luego de un ciclo con condición B e invariante J, es válida la negación de la condición ($!B$) junto con el invariante J. Esto se muestra en el siguiente esquema de programa:

```
//@ maintaining J;
while (B) {  // suponiendo que B no tiene efectos laterales
  S
}
// assert !(B) && J;
```

```
1  package org.jmlspecs.samples.jmlrefman;
2
3  /** An example of some simple loops with loop invariants
4   *  and variant functions specified.
5   */
6  public abstract class SumArrayLoop {
7
8      /** Return the sum of the argument array. */
9      /*@   old \bigint sum =
10       @   (\sum int j; 0 <= j && j < a.length; (\bigint)a[j]);
11       @   requires Long.MIN_VALUE <= sum && sum <= Long.MAX_VALUE;
12       @   assignable \nothing;
13       @   ensures \result == sum;
14       @*/
15     public static long sumArray(int [] a) {
16         long sum = 0;
17         int i = a.length;
18
19         /*@ maintaining -1 <= i && i <= a.length;
20           @ maintaining sum
21           @             == (\sum int j;
22           @                     i <= j && 0 <= j && j < a.length;
23           @                     (\bigint)a[j]);
24           @ decreasing i; @*/
25         while (--i >= 0) {
26             sum += a[i];
27         }
28
29         //@ assert i < 0 && -1 <= i && i <= a.length;
30         //@ hence_by (i < 0 && -1 <= i) ==> i == -1;
31         //@ assert i == -1 && i <= a.length;
32         //@ assert sum ==
33         //@     (\sum int j; 0 <= j && j < a.length; (\bigint)a[j]);
34         return sum;
35     }
36 }
```

Fragmento de código fuente 3.14: Ejemplo de anotaciones de ciclos

Si el ciclo contiene una instrucción **break** que termina la ejecución, se necesita pensar en detalle si es necesario establecer lo que será verdadero luego del ciclo. La condición deseada que debería ser verdadera luego del ciclo, que es terminado con una instrucción **break**, puede ser anotada en el código utlizando una instrucción **assert**, por ejemplo, si el ciclo tiene la forma:

```
//@ maintaining J;
while (true) {
  S1
  if (C) {
     S2
     //@ assert Q;
     break;
  }
  S3
}
```

entonces, luego del ciclo la condición afirmada Q, debería ser válida, asumiendo que no hay otras instrucciones **break** que terminen la ejecución del ciclo.

3.6.2. Funciones variantes de ciclos

Un ciclo puede también especificar una o más funciones variantes, utilizando las palabras clave **decreasing**, **decreasing_redundantly**, **decreases** y **decreases_redundantly**. Una función variante es utilizada para ayudar a probar la terminación de un ciclo. Especifica una expresión de tipo `long` o `int` que debe ser no menor a 0 cuando el ciclo se está ejecutando, y debe decrementarse por al menos uno (1) cada vez que se ejecuta el ciclo.

El significado de un ciclo tal como el siguiente

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

en el que S no utiliza la instrucción **continue** es la siguiente:

```
while (true) {
  long vf = E;  // suponiendo que vf es un nombre de variable nuevo
  if (!(B)) { break; }
  S
  //@ assert 0 <= vf;
  //@ assert E < vf;
}
```

Si el ciclo contiene una instrucción **continue**, entonces el variante del ciclo se verifica antes de cada uso de la instrucción **continue**. Por ejemplo, si el ciclo tiene la forma:

```
//@ decreasing E;
while (B) {
  S1
  if (C) {
    S2
    continue;
  } S3
}
```

entonces el significado es el siguiente:

```
while (true) {
  long vf = E; // suponiendo que vf es un nombre de variable nuevo
  if (!(B)) { break; }
  S1
  if (C) {
    S2
    //@ assert 0 <= vf;
    //@ assert E < vf;
    continue;
  }
  S3
  //@ assert 0 <= vf;
  //@ assert E < vf;
}
```

3.7. Resumen

En este capítulo vimos los elementos principales del lenguaje JML, entre ellos:

- JML es un lenguaje que especifica el comportamiento e interfaz de clases y métodos Java. JML provee una semántica para describir de forma formal el comportamiento de un módulo de Java, evitando así la ambigüedad con respecto a las intenciones del diseñador del módulo.

- JML utiliza pre y postcondiciones e invariantes y permite aplicar la metodología de diseño por contrato (DBC), cuya idea principal es que las clases y sus clientes tengan un contrato entre ellas [42]. 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 [38].
- Las palabras claves utilizadas en JML: **invariant**, **requires**, **ensures**, **signals**, **signals_only**, **assignable**, **pure**, **also**, **assert**, **spec_public**, **non_null**, **nullable**.
- Las expresiones que provee: **\result**, **a ==> b**, **a <== b**, **a <==> b**, **\fresh**, **\typeof**, **\type**, **subtipo**, **\max**, **\min**, **\product**, **\sum**, **\num_of**.
- Los cuantificadores que soporta: **\forall**, **\exists**
- También se dedicó una sección a las anotaciones de sentencias e invariantes de ciclos.

Capítulo 4

Análisis comparativo entre OCL y JML

4.1. Introducción

En los capítulos 2 y 3 se presentaron los lenguajes OCL y JML. Los dos lenguajes son lenguajes de especificación, pero tienen diferencias en sus objetivos y características. Para evaluar la factibilidad de la traducción de una especificación OCL a una especificación JML, debemos analizar la correspondencia entre las construcciones de ambos lenguajes, tanto a nivel sintáctico como a nivel semántico, con enfásis en la última. Mostraremos que aunque OCL y JML son lenguajes muy similares, no son equivalentes, es decir, hay construcciones en OCL que no tienen correspondencia en JML y viceversa, por lo tanto reconocer estas diferencias es fundamental para desarrollar una traducción adecuada. Cuando hablemos de la función de traducción, distinguiremos entre Java y JML en los casos que sea necesario, pero en general hablaremos de Java.

4.2. Diferencias entre OCL y JML

4.2.1. Ventajas y desventajas de OCL y JML

Podemos sintetizar a grandes rasgos las diferencias entre OCL y JML [22]

OCL

Ventajas

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

1. OCL no puede verificarse fácilmente en tiempo de ejecución.

JML

Ventajas

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

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

4.2.2. Diferencias semánticas

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]);
```

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 [22, 35].

Rango de cuantificadores

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

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, por lo que son válidas las siguientes igualdades [25].

```
Integer.MIN_VALUE == Integer.MAX_VALUE + 1
Math.abs(Integer.MIN_VALUE) == Integer.MIN_VALUE
Integer.MIN_VALUE * Integer.MIN_VALUE == 0
(Integer.MIN_VALUE + 1) * (Integer.MIN_VALUE + 1) == 1
Integer.MAX_VALUE * Integer.MAX_VALUE == 1
```

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 [22]. En [25] se propone una extensión a JML llamada JMLa que incluye un tipo primitivo **\bigint** que representa enteros de precisión arbitraria, i.e. los enteros matemáticos.

Igualdad e identidad

En Java existen dos maneras de comparar objetos: por identidad (i.e. nivel de referencia) y por igualdad (i.e. nivel de objetos). Para tipos primitivos (e.g. `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 la figura 4.1 vemos un ejemplo.

```
public class Punto {
  private int x;
  private int y;

  public Punto(int x, int y){
    this.x = x;
    this.y = y;
  }

  @Override
  public boolean equals(Object o){
    if (o instanceof Punto){
      Punto p = (Punto) o;
      return this.x == p.x && this.y == p.y;
    }
    return false;
  }

  public static void main(String[] args) {
    Punto p1 = new Punto(2, 3);
    Punto p2 = new Punto(2, 3);
    // Imprime false p1 y p2 no son el mismo objeto
    System.out.println(p1 == p2);
    // Imprime true p1 y p2 son iguales segun la definicion
    System.out.println(p1.equals(p2));
  }

}
```

Fragmento de código fuente 4.1: Ejemplo de igualdad e identidad en Java

En OCL sólo se puede comparar los objetos por igualdad [30].

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, e.g. *union* o *append* que no tienen sentido en arreglos.

Excepciones

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

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. JML provee el calificador **spec_public** para la especificación pública de atributos privados.

4.3. Función de traducción / Mapeos

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 [34, 36] y la de colecciones en [19].

4.3.1. 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`

Tabla 4.1: Traducción de tipos de datos simples

4.3.2. 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.

OCL	**JML**
not e	$!e$
e_1 *and* e_2	e_1 && e_2
e_1 *or* e_2	$e_1 \mid\mid e_2$
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*	$(e_0 ? e_1 : e_2)$

Tabla 4.2: Traducción de operaciones básicas

4.3.3. Pseudovariables

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

4.3.4. Cuantificadores

Los cuantificadores son mapeados directamente a JML. Debemos tener en cuenta las consideraciones señaladas en la sección 4.2.2, en JML el rango de los cuantificadores son todos los objetos del tipo.

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`)

Tabla 4.3: Traducción de pseudovariables

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)

Tabla 4.4: Traducción de cuantificadores

4.3.5. Colecciones

Una parte importante de OCL son las colecciones y las operaciones sobre éstas. Las operaciones sobre colecciones es uno de los aspectos más difíciles de mapear a JML, una de las principales razones es que las operaciones OCL están basadas en el framework de colecciones de Smalltalk [53]. Smalltalk, a diferencia de Java, permite pasar bloques de código como parámetros. Java no posee esta funcionalidad, no tiene funciones de alto orden, lo que dificulta la implementación de las operaciones *select*, *reject*, *collect* e *iterate*.

Uno de los problemas principales mencionados en [19] es que no siempre puede ser posible encontrar un mapeo apropiado entre los modelos OCL y las representaciones Java. Puede no haber un vocabulario Java correspondiente para las expresiones OCL utilizadas en las restricciones. En general, las restricciones OCL tiene que ser redefinidas en el vocabulario definido por una particular elección de una representación, e.g. ,conjuntos, listas o arreglos. Tales especificaciones tienden a ser largas, difíciles de leer y entender, y difíciles de ser relacionadas a las restricciones OCL originales.

Uno de los enfoques considerados para implementar las operaciones de colecciones OCL es el descripto en [35, 36], podemos observar como principal ventaja de este enfoque la fácil implementación y mapeo a JML, pero presenta dos problemas principales: la organización, estructura y vocabulario de las colecciones JML son distintos a las de OCL y la expresión JML resultante de la traducción puede ser muy distinta sintácticamente de la expresión OCL original. El otro enfoque considerado es el que se describe en [19], donde se plantea el desarrollo de una librería que implemente las colecciones OCL en el lenguaje Java.

Las colecciones OCL son mapeadas en clases Java, según la tabla 4.5. La introducción de una librería de colecciones del tipo OCL nos permite mapear las restricciones OCL en especificaciones JML de una forma unívoca preservando las estructuras originales y usando casi el mismo vocabulario. Las clases Java que implementan la librería de clases OCL son inmutables, i.e. no hay métodos que puedan cambiar los valores de estas clases. Se implementan las operaciones *select* y *reject* (y *collect*) adaptando el esquema definido en [31], las que se mapean de la siguiente forma, sea la siguiente expresión OCL:

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

La expresión OCL indica que la cantidad de asientos reservados del vuelo debe ser menor o igual a la capacidad del avión. Al no disponer en Java de funciones de primer orden, debemos extender la clase

`OCLFilterExpression`, la que define una función. Podemos comparar con el enfoque planteado en

```
public class Vuelo
{
  /*@ private model OCLFilterExpression<Asiento> filter0(){
    @ return new OCLFilterExpression<Asiento>(){
    @   public boolean filter(Asiento a){
    @     return a.getReservado();
    @   }
    @ };
    @}
  */
    private /*@ spec_public non_null */  OCLSet<Asiento> asientos;
  /*@ public invariant this.asientos.select(filter0).size()
    @ <= this.capacidad;
  */
  /*  */
}
```

Fragmento de código fuente 4.2: Traducción a JML

[35], con lo cual la expresión nos quedaría de la siguiente forma:

```
//@ public invariant !(new JMLObjectSet{Asiento a |
    this.asientos.has(a) && a.reservado}.size() <= this.capacidad)
```

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

Tabla 4.5: Traducción de colecciones

Una de las operaciones más difíciles es la operación *sum* definida en la clase `Collection`, que retorna la suma de los elementos de la colección. El estándar OCL ordena que cada elemento de la colección debe pertenecer a un tipo que soporte la adición binaria (+), y el tipo de retorno debe ser del tipo dado como parámetro. El lenguaje Java no posee polimorfismo paramétrico y clases de tipo (como Haskell), por lo que la solución es definir el tipo numérico más general (`Double`) como el tipo de retorno y revisar en tiempo de ejecución el tipo de cada elemento. De acuerdo a los tipos de los elementos la suma es retornada como `Integer` o `Double`, si al menos un elemento no soporta la adición, entonces se genera una excepción `RuntimeException`.

4.3.6. Atributos derivados

En la subsección 2.3 se explica el valor derivado. OCL permite definir valores derivados y definidos. Existen dos formas principales de mapear estas construcciones, dependiendo de si los atributos definidos y derivados se utilizan sólo en especificaciones o forman parte del modelo de implementación. En el primer caso, se mapean los atributos definidos o derivados en OCL, como ghost fields en JML, i.e. atributos que sólo pueden utilizarse en la especificación cuyo valor se determina a partir de otros atributos. En el segundo caso, para cada atributo definido o derivado se agrega al modelo UML una operación que retorna el valor del atributo definido o derivado. Hemos optado por la segunda opción, para permitir la utilización de los atributos derivados. El mapeo puede realizarse de las siguientes formas:

1. Definir cada atributo derivado como ghost field en JML cuyo valor es del atributo derivado.
2. Agregar para cada atributo derivado un model method en JML que retorna el valor del atributo derivado o definido.
3. Agregar para cada atributo derivado una operación en el modelo UML que retorne el valor del atributo.

4.3.7. Construcción let

Un caso especial en la traducción de expresiones es la construcción **let var = exp in exp2**. Esta construcción puede mapearse de tres formas, dependiendo como en el caso de los atributos derivados, si sólo se utiliza en especificaciones o puede formar parte del modelo de implementación. El mapeo es similar al realizado para los atributos derivados o definidos.

4.3.8. Traducción de contratos / Precondiciones, postcondiciones e invariantes

OCL	JML
inv: condition	//@ public invariant condition;
pre: condition	//@ requires condition;
post: condition	//@ ensures condition;

Tabla 4.6: Traducción de precondiciones, postcondiciones e invariantes

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

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

4.4. Resumen

En este capítulo se vio a grandes rasgos una comparación entre los lenguajes OCL y JML. Vimos las ventajas y desventajas de cada uno, entre ellas pueden enumerarse:

4.4.1. Ventajas de OCL sobre JML

1. OCL tiene la ventaja que no está relacionado con ningún lenguaje determinado, mientras que JML está relacionado al lenguaje Java.
2. OCL tiene un nivel más alto de abstracción. Un diagrama UML puede tener anotaciones OCL antes del desarrollo de código. En cambio JML se incluye dentro del código.
3. Está estandarizado por el OMG. En cambio JML no está estandarizado, la mayoría de las herramientas actuales no implementan la semántica de JML.

```
public class Vuelo
{
  /*@ private model OCLFilterExpression<Asiento> filter0(){
    @ return new OCLFilterExpression<Asiento>(){
    @   public boolean filter(Asiento a){
    @     return a.getReservado();
    @   }
    @ };
    @}
  */
    private /*@ spec_public non_null */  OCLSet<Asiento> asientos;
  /*@ public invariant this.asientos.select(filter0).size()
    @ <= this.capacidad;
  */
  /*  */

  /*@ requires ! (this.cancelado and
    @   this.asientos.select(filter0).size() < this.capacidad);
  */
  /*@ ensures (\exists Asiento a; this.asientos.includes(a);
    @   a.reservado && a.getPasajero().equals(pasajero));
  */
  public Boolean reservar(Pasajero pasajero);
}
```

Fragmento de código fuente 4.3: Traducción a JML

4.4.2. Ventajas de JML sobre OCL

1. Es que 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. En cambio OCL no puede verificarse en tiempo de ejecución (ver OCLE en la Sección 5.4).

Y luego vimos la diferencia entre ellos con respecto a: diferencias semánticas, tipos simples, operaciones y expresiones, pseudovariables, cuantificadores, colecciones (de la cual se abordó un poco mas en profundidad el tema, ya que no todas las operaciones con colecciones OCL tienen su correspondencia en funciones JML), atributos derivados, let y por último, pre y postcondiciones e invariantes.

Capítulo 5

Herramientas de verificación de programas

5.1. 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 [50]. 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. Generalmente son incompletos.

El análisis estático de programas es el análisis de software que es realizado sin ejecutar realmente el programa. En la mayoría de los casos, el análisis es llevado a cabo por una herramienta automática. La sofisticación del análisis varía desde aquellas que consideran sólo instrucciones y declaraciones, hasta las que revisan el programa completo.

5.2. Herramientas de verificación

Existen diversas herramientas de verificación que se pueden clasificar en:

- **Runtime checkers**: agregan chequeos en tiempo de ejecución.
  - jmlrac
- **Static checkers**: tienen mejor cobertura, encuentran errores y violaciones de contratos.

  - ESC/Java2

- **Buscadores de modelos**: buscan contraejemplos.
- **Verificadores de programas**: verificación total, son asistidos por computadora pero no automáticos.
  - KeY
  - Loop
  - Jack

5.2.1. Estructura general

5.3. Primeras herramientas JML

En 1999 se definió el lenguaje JML e inmediatamente después comenzó el desarrollo de herramientas de verificación basadas en el nuevo lenguaje [37]. Las funcionalidades básicas que debe proveer cualquier herramienta JML son el análisis sintáctico y el chequeo de tipos. Las herramientas principales desarrolladas para JML son:

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.

5.4. OCLE

Object Constraint Language Environment (OCLE) es una herramienta CASE UML que ofrece soporte completo para OCL, tanto al nivel modelo como metamodelo de UML, desarrollada en la universidad de Cluj-Napoca entre 2003 y 2005 [8]. OCLE provee generación de código Java tanto del modelo UML como de las especificaciones OCL, el código generado incluye el sistema de tipos OCL y verificación automática de las restricciones OCL.

La principal ventaja de OCLE es que permite centralizar el desarrollo del modelo y la definición de restricciones en la misma herramienta, como principales desventajas podemos nombrar que el código generado es confuso ya que mezcla el código de implementación con la verificación de restricciones y que sólo soporta UML hasta la versión 1.5.

5.5. ESC/Java2

Extended static checker for Java version 2 (ESC/Java2) es una herramienta de programación que intenta encontrar errores comunes en tiempo de ejecución en programas Java con anotaciones JML, realizando un análisis estático del código del programa y de las anotaciones. El enfoque subyacente utilizado en ESC/Java2 es el análisis estático de código. Esta técnica fue utilizada en ESC/Java (y su predecesor ESC/Modula-3) [27]. El chequeo estático de código generalmente involucra el uso de un demostrador automático de teoremas, y en ESC/Java2 se utiliza usualmente Simplify.

ESC/Java2 no es ni consistente (*sound*) ni completo. Esto es intencional y la idea es reducir el número de errores y/o warnings que se reportan al programador, lo que hace a la herramienta más útil en la práctica.

ESC/Java fue desarrollado originalmente en Compaq en 1997, luego del trabajo realizado en ESC/Modula-3 que finalizó en 1996. En 2002 se liberó el código fuente de ESC/Java y herramientas relacionadas. ESC/Java2 fue liberado en 2004 (por la Universidad de Nijmegen's). Desde 2004, el desarrollo de ESC/Java2 ha sido administrado por el KindSoftware Research Group en University College Dublin. A través de los años, ESC/Java2 ha incorporado nuevas características, como la habilidad de utilizar múltiples demostradores de teoremas y la integración con Eclipse [24].

5.5.1. Estructura de ESC/Java2

En la Figura 5.1 se puede ver la estructura general de ESC/Java2. ESC/Java2 traduce un programa anotado con especificaciones JML a condiciones de verificación (Verification condition). ESC/Java parsea el código Java y lo compila a una forma intermedia, a la que agrega las especificaciones JML. Luego emplea técnicas de análisis estático para la ejecución simbólica del código que está entre las especificaciones JML. Los resultados de las especificaciones JML y de la ejecución simbólica son utilizados para construir fórmulas de lógica de primer orden, las que son ingresadas a un demostrador de teoremas automático externo. El demostrador intenta encontrar contraejemplos para las fórmulas, si falla significa que las condiciones son válidas.

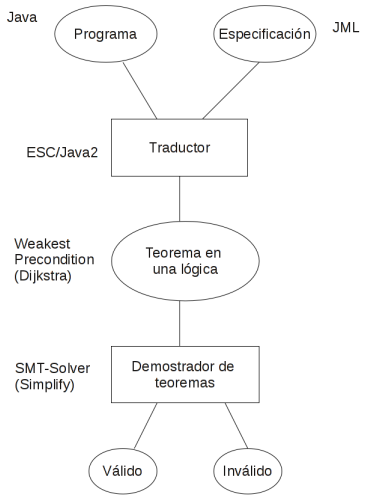

Figura 5.1: Estructura general de ESC/Java2

```
1  public class A
2  {
3    byte[] b;
4    public void n(){
5      b = new byte[20];
6    }
7
8    public void m(){
9      n();
10     b[0] = 2;
11   }
12 }
```

Fragmento de código fuente 5.1: Ejemplo de uso ESC/Java2

5.5.2. Ejemplo de uso

ESC/Java2 realiza un razonamiento modular sobre el programa, es decir, razona sobre cada método por separado. Si ejecutamos ESC/Java2 sobre el programa de Código fuente 5.1, ESC/Java2 nos advierte (Código fuente 5.2) que en la línea 10 puede haber una referencia a **null** aunque sepamos que eso no es posible. Para corregir los errores, modificamos el programa (Código fuente 5.3) y volvemos a ejecutar la herramienta (Código fuente 5.4) que no advierte de ningún error.

```
1  ESC/Java version ESCJava-2.0.5
2      [0.062 s 17739392 bytes]
3
4  A ...
5    Prover started:0.013 s 22154240 bytes
6      [0.148 s 22777112 bytes]
7
8  A: n() ...
9      [0.023 s 22777112 bytes]  passed
10
11 A: m() ...
12 ------------------------------------------------------------------------
13 ../A.java:10: Warning: Possible null dereference (Null)
14     b[0] = 2;
15      ^
16 ------------------------------------------------------------------------
17 ../A.java:10: Warning: Array index possibly too large (IndexTooBig)
18     b[0] = 2;
19      ^
20 ------------------------------------------------------------------------
21     [0.043 s 23400056 bytes]  failed
22
23 A: A() ...
24     [0.01 s 24022920 bytes]  passed
25   [0.225 s 24022920 bytes total]
26 2 warnings
```

Fragmento de código fuente 5.2: Resultado de la ejecución de ESC/Java2

```
1  public class A
2  {
3    byte[] b;
4
5    //@ensures b != null && b.length == 20;
6    public void n(){
7      b = new byte[20];
8    }
9
10   public void m(){
11     n();
12     b[0] = 2;
13   }
14 }
```

Fragmento de código fuente 5.3: Ejemplo de uso ESC/Java2

```
1  ESC/Java version ESCJava-2.0.5
2      [0.048 s 17747192 bytes]
3
4  A ...
5    Prover started:0.013 s 22269000 bytes
6      [0.135 s 22269000 bytes]
7
8  A: n() ...
9      [0.031 s 22896256 bytes]  passed
10
11 A: m() ...
12     [0.017 s 23519184 bytes]  passed
13
14 A: A() ...
15     [0.022 s 23519184 bytes]  passed
16   [0.206 s 23519184 bytes total]
```

Fragmento de código fuente 5.4: Resultado de la ejecución de ESC/Java2

5.5.3. Consistencia y completitud de ESC/Java2

Un sistema lógico es consistente (*sound*) si cuando una aserción {P}S{Q} se puede demostrar, entonces es válida en la semántica. Un sistema de verificación es completo si una aserción válida en la semántica se puede demostrar. ESC/Java2 no es consistente ni completo, ESC/Java2 puede no producir advertencias para todos los errores, y puede producir mensajes de error superfluos (espurios). La no consistencia se debe, principalmente, a que los ciclos son analizados mediante unrolling (no se utilizan invariantes de ciclos), al modelo aritmético y que los invariantes de objetos no son verificados en todos los objetos existentes. La incompletitud se debe principalmente a la incompletitud de Simplify [29], el cual tiene capacidades de razonamiento limitadas sobre los cuantificadores y la aritmética.

La no consistencia y no completitud es una decisión de diseño deliberada [24]. El objetivo es incrementar la efectividad de la herramienta. Pueden requerirse un gran número de anotaciones JML para convencer al verificador de la ausencia de un error. En [41] hay una lista de todos los casos de inconsistencia e incompletitud en ESC/Java. El Código fuente 5.5 muestra un ejemplo de incompletitud de ESC/Java2.

```
1  public class Fermat
2  {
3    /*@ requires 0 < n;
4      @ ensures \result ==
5      @      (\exists int x,y,z;
6      @        Math.pow(x,n) + Math.pow(y,n) == Math.pow(z,n));
7      @*/
8    public static boolean fermat(double n) {
9      return (n == 2);
10   }
11 }
```

Fragmento de código fuente 5.5: Ejemplo de incompletitud en ESC/Java2

Al ejecutar el ejemplo anterior, ESC/Java2 advierte que la postcondición posiblemente no sea satisfecha. Generalmente, el demostrador automático tardó demasiado.

```
1  public class Positive{
2    private int n = 1;
3
4    //@ invariant n > 0;
5    public void increase(){
6      n++;
7    }
8  }
```

Fragmento de código fuente 5.6: Ejemplo de inconsistencia en ESC/Java2

En el ejemplo de Código fuente 5.6, ESC/Java2 no produce ninguna advertencia, pero *increase* puede romper el invariante si $n = 2^{31} - 1$. ESC/Java2 no contempla representación concreta, este problema puede ser solucionado mejorando el modelo aritmético de Java.

5.5.4. Conclusiones

El análisis estático es un método rápido pero poderoso para encontrar potenciales problemas. Permite asegurar que los casos más raros serán también tratados correctamente, y es útil para

determinar dónde pueden ocurrir los errores en tiempo de ejecución. Sin embargo, debemos tener en cuenta que ESC/Java2 es inconsistente e incompleto.

Ventajas

1. Automática, no requiere intervención humana.
2. Se obtiene feedback sin especificaciones.
3. Detecta la mayoría de los errores comunes.
4. Fuerza al programador a escribir los contratos de los métodos.

Desventajas

1. No es completa ni consistente (tira falsos positivos y falsos negativos).
2. Requiere bastantes anotaciones.
3. Se requiere cierta experiencia para interpretar los resultados.
4. Sólo procesa código fuente Java hasta la versión 1.4 (i.e. no hay genéricos) [28].
5. Los mensajes de error pueden ser crípticos.
6. No está bien documentada.

5.6. Sireum/Kiasan

Sireum/Kiasan es una herramienta de verificación automática basada en contratos JML y generación de casos de prueba, desarrollada por la Universidad de Kansas y la Universidad de Pennsylvania [51]. Kiasan no necesita parámetros de entrada para chequeo de unidades. Sin aserciones o contratos, Kiasan detecta posibles excepciones en tiempo de ejecución como `NullPointerException`, `ArithmeticException`, etc. Con aserciones embebidas en el código, Kiasan determina automáticamente si las aserciones pueden ser violadas. Cuando se agregan contratos basados en JML, Kiasan puede utilizarlos para filtrar parámetros de entrada que no satisfagan los contratos, mientras asegura que los estados luego de la ejecución satisfacen los contratos. A diferencia de otras herramientas, Kiasan genera un número bajo de alarmas. Además Kiasan genera feedback mostrando los estados de programa y casos de prueba JUnit útiles para mostrar violaciones de propiedades.

5.6.1. Ejemplo

Si ejecutamos Kiasan sobre el programa de Código fuente 5.7 se obtiene el resultado del Código fuente 5.8.

Kiasan indica que encontró un ejemplo en el cual la postcondición del método *transfer* no se cumple. En la Figura 5.2 podemos ver que la postcondición no se cumple cuando existe un alias entre *this* y *account*. En el segundo contraejemplo (Figura 5.3) podemos ver que ocurre un error en tiempo de ejecución (`NullPointerException`) cuando *account* es **null**.

5.7. JML4c

JML4c es un compilador de JML construido sobre la plataforma Eclipse JDT. Traduce un subconjunto significante 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:

1. Velocidad de compilación mejorada.

```
1  public class Account
2  {
3    private /*@ spec_public */ int balance;
4    //@ public invariant balance >= 0;
5
6    /*@ ensures this.balance == 0; @*/
7    public Account(){
8      this.balance = 0;
9    }
10
11   /*@ requires amount >= 0;
12     @ ensures this.balance == amount; @*/
13   public Account(int amount){
14     this.balance = amount;
15   }
16
17   /*@ requires amount > 0;
18     @ ensures this.balance == \old(this.balance) + amount; @*/
19   public void deposit(int amount){
20     this.balance = this.balance + amount;
21   }
22
23   /*@ requires amount > 0 && amount <= this.balance;
24     @ ensures this.balance == \old(this.balance) - amount; @*/
25   public void withdraw(int amount){
26     this.balance = this.balance - amount;
27   }
28
29   /*@ requires amount > 0 && amount <= this.balance;
30     @ ensures this.balance == \old(this.balance) - amount &&
31       account.balance == \old(account.balance + amount); @*/
32   public void transfer(Account account, int amount){
33     this.withdraw(amount);
34     account.deposit(amount);
35   }
36 }
```

Fragmento de código fuente 5.7: Ejemplo de uso Sireum/Kiasam

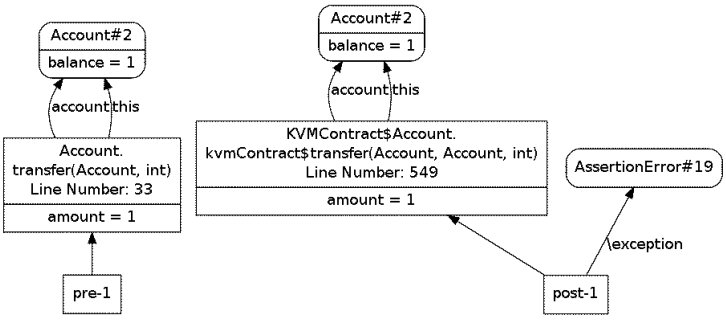

Figura 5.2: Primer contraejemplo encontrado por Kiasan

```
1  Executing Account.<init> ...
2
3  Generating report ...
4  Done!
5
6  Executing Account.<init> ...
7
8  Generating report ...
9  Done!
10
11 Executing Account.deposit ...
12
13 Generating report ...
14 Done!
15
16 Executing Account.withdraw ...
17
18 Generating report ...
19 Done!
20
21 Executing Account.transfer ...
22
23 Ensures clauses, ((this.balance == ((\old this.balance) - amount))
24  && (account.balance == (\old (account.balance + amount)))),
25   was violated
26
27 Generating report ...
28 Done!
29
30 Rendering reports...
31 Done!
```

Fragmento de código fuente 5.8: Resultado de la ejecución de Sireum/Kiasan

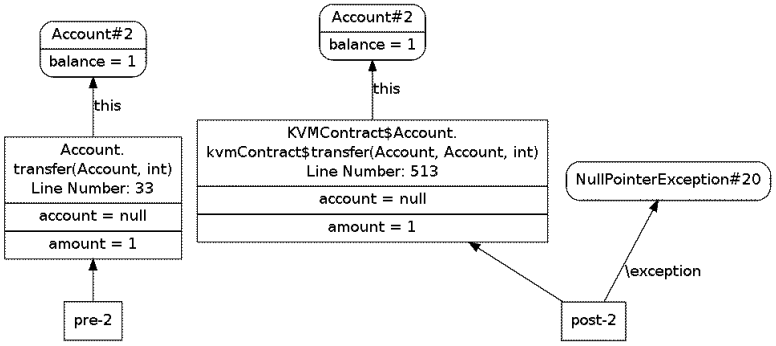

Figura 5.3: Segundo contraejemplo encontrado por Kiasan

2. Soporte para características de Java 5, tales como genéricos y ciclos **for** mejorados.

3. Soporte para clases internas.

4. Mensajes de error mejorados.

5.8. OpenJML

OpenJML es una reimplementación, 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.

Las herramientas anteriores a OpenJML estaban basadas en compiladores implementados manualmente (hand-crafted), y el esfuerzo de mantener esos compiladores Java sobrepasó a los voluntarios y académicos al evolucionar el lenguaje Java [26]. Para poder facilitar la actualización de las herramientas JML a la evolución del lenguaje Java, la comunidad JML decidió implementar JML sobre OpenJDK. OpenJDK era una alternativa, los integrantes del proyecto OpenJML descubrieron que, si bien OpenJDK no tenía Integrated Development Environment (IDE), era mucho más extensible que Eclipse JDT. Rápidamente, se desarrollaron herramientas de línea de comandos para parseo y chequeo de tipos de construcciones JML y código Java, también se realizaron experimentos con SMT solvers (Yices, CVC3 y Simplify) para verificación.

Uno de los principales problemas (temas a resolver) de esta implementación es que al integrar OpenJML a Eclipse, el compilador Eclipse es utilizado para compilar código Java y el compilador OpenJML/OpenJDK es utilizado como backend para procesar anotaciones JML y tareas de verificación.

5.9. Otras herramientas

5.9.1. KeY

KeY es una herramienta que provee facilidades para especificación formal y verificación de programas [17]. Originalmente fue diseñada para utilizar OCL como lenguaje de programación, pero luego fue modificada para utilizar JML [26]. 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.

5.9.2. JacK

JACK es una herramienta desarrollada inicialmente en el laboratorio de investigación de Gemplus, un fabricante de tarjetas inteligentes. Actualmente el desarrollo se continúa en 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.

5.9.3. LOOP

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 [24].

5.10. Resumen

En este capítulo se explica la definición de verificación de programas, así como distintas herramientas de verificación. Éstas se clasifican en: Runtime checkers (Por ejemplo: jmlrac), Static checkers (Por ejemplo: ESC/Java2), Buscadores de modelos y Verificadores de programas (Por ejemplo: Key, Loop, Jack). También se muestran ejemplos de entrada y salida sobre ellas.

Además, algunas de ellas serán utilizadas para verificar la salida de nuestra implementación explicada en los Capítulos 9 y 10.

Capítulo 6

Desarrollo de software dirigido por modelos

6.1. Introducción

El desarrollo de software dirigido por modelos (MDD) y en concreto la propuesta Model driven architecture (MDA) de OMG constituyen una aproximación para el desarrollo de sistemas de software que consiste en separar la especificación de la funcionalidad del sistema, de la especificación de la implementación de esa funcionalidad en una Plataforma tecnológica específica [45].

6.2. Arquitectura dirigida por modelos

MDA es una de las iniciativas más conocidas y extendidas dentro del ámbito de MDD. Fue presentada por el consorcio OMG en Noviembre de 2000, con el objetivo de abordar los desafíos de integración de aplicaciones y los continuos cambios tecnológicos.

MDA propone el uso de un conjunto de estándares como MOF, UML o XML Metadata Interchange (XMI). Su objetivo es separar la especificación de la funcionalidad del sistema de implementación sobre una plataforma concreta, por lo que se hace una distinción entre modelos Platform Independent Model (PIM) y modelos Platform Specific Models (PSMs).

6.2.1. Conceptos básicos

MDA está estructurado alrededor de un conjunto de conceptos básicos [46], entre los que podemos mencionar:

1. **Sistema**: concepto principal de MDA que puede incluir: un programa, un único sistema de computadora, alguna combinación de partes de diferentes sistemas, personas, una empresa, etc.

2. **Modelo**: un modelo de un sistema es una descripción de una especificación del sistema y de su entorno. Generalmente es presentado como una combinación de texto y gráficos.

3. **Dirigido por modelos**: MDA es dirigido por modelos porque provee una forma de usar los modelos para dirigir el análisis, diseño, construcción, despliegue, operación, mantenimiento y modificación de un sistema.

4. **Arquitectura**: la arquitectura de un sistema es una especificación de las partes y conectores del sistema y de las reglas para la interacción de las partes utilizando los conectores.

5. **Punto de vista**: Un punto de vista de un sistema es técnica para la abstracción utilizando un conjunto elegido de conceptos arquitecturales y reglas de estructuración, para poder enfocarse en aspectos particulares del sistema.

6. **Vista**: una vista o modelo de punto de vista de un sistema es una representación de ese sistema desde la perspectiva de un punto de vista.

7. **Plataforma**: una plataforma es un conjunto de subsistemas y tecnologías que proveen un conjunto coherente de funcionalidad a través de interfaces y de patrones de uso especificados.

8. **Aplicación**: una aplicación es una funcionalidad que es desarrollada. Un sistema es descripto en términos de una o más aplicaciones soportadas por una o más plataformas.

9. **Independencia de la plataforma**: la independencia de la plataforma es una cualidad que un modelo puede exhibir, que indica cuán independiente es un modelo de las características de una plataforma de cualquier tipo particular.

6.3. Modelos y transformaciones

6.3.1. Introducción

Un modelo es una representación conceptual o física a escala de un proceso o sistema, con el fin de analizar su naturaleza, desarrollar o comprobar hipótesis o supuestos y permitir una mejor comprensión del fenómeno real al cual el modelo representa y permitir así perfeccionar los diseños, antes de iniciar la construcción de los objetos reales.

6.3.2. Tipos de modelos

Tipos de modelos según MDD

Algunos modelos describen el sistema de manera independiente de los conceptos técnicos que involucra su implementación en una plataforma de software, mientras que otros modelos tienen como finalidad primaria definir tales conceptos técnicos. Los tipos de modelos que identifica MDD son:

- **Modelo independiente de la computación (CIM)**. Un CIM es una vista del sistema desde un punto de vista independiente de la computación que no muestra detalles de la estructura del sistema. Usualmente se lo llama modelo del dominio y en su construcción se utiliza un vocabulario familiar para los expertos del dominio.

- **Modelo independiente de la plataforma (PIM)**. Un PIM es un modelo con un alto nivel de abstracción que es independiente de cualquier tecnología o lenguaje de implementación. Dentro del PIM, el sistema se modela desde el punto de vista de cómo se soporta mejor al negocio, sin tener en cuenta como va a ser implementado.

- **El modelo específico de la plataforma (PSM)**. Como siguiente paso un PIM se transforma en uno o más PSMs. Cada PSM representa la proyección del PIM en una plataforma específica.

- **El modelo de implementación (Código)**. El paso final en el desarrollo es transformación de cada PSM a código fuente. La transformación suele ser bastante directa ya que el PSM está orientado al dominio tecnológico específico.

- **Modelo de plataforma**. Un modelo de plataforma provee un conjunto de conceptos técnicos, que representan los diferentes tipos de partes que forman una plataforma. También provee, para utilizar en un PSM, los conceptos que representan los diferentes tipos de elementos a ser usados para especificar el uso de la plataforma por una aplicación.

6.3.3. Cualidades de los modelos

El modelo de un problema es esencial para describir y entender el problema. El modelo constituye la base fundamental de información sobre la que interactúan los expertos en el dominio del problema y los desarrolladores de software, por lo que es fundamental que exprese el problema en forma clara y sencilla. Podemos nombrar las cualidades que esperamos encontrar en los modelos:

- **Comprensibilidad**. El modelo debe estar en un lenguaje entendible por todos los usuarios.
- **Precisión**. El modelo debe ser una fiel representación del objeto o sistema modelado. El lenguaje de modelado debe poseer una semántica precisa que permita la interpretación unívoca de los modelos.
- **Consistencia**. El modelo no debe contener información contradictoria.
- **Completitud**. El modelo debe capturar todos los requisitos necesarios. Sin embargo, dado que generalmente no se cuenta con un modelo completo al inicio, se debe poder expandir un modelo incompleto de acuerdo al conocimiento de dominio que se adquiere.
- **Flexibilidad**. El modelo debe poder ser fácilmente adaptado para reflejar las modificaciones en el dominio del problema.
- **Reusabilidad**. El modelo debe proveer las bases para el reuso de conceptos y construcciones que se presentan en forma recurrente en una amplia variedad de problemas.
- **Corrección**. Hay dos aspectos a considerar para la corrección de un modelo. En primer lugar, el modelo debe ser analizado para asegurar que cumpla con las expectativas del usuario, esto se denomina *validación* del modelo. Luego, asumiendo que el modelo es correcto, puede utilizarse como referencia para analizar la corrección de la implementación del sistema, esto se denomina *verificación* del software.

6.3.4. Transformaciones

La transformación de modelos consiste en obtener un nuevo modelo mediante la transformación de un modelo existente. En dicho contexto, los modelos son elementos de primer orden, apareciendo entonces la noción de metamodelos y transformaciones.

Un modelo debe ser definido de acuerdo a la semántica de su metamodelo: un modelo se dice que conforma a su metamodelo. De la misma manera, un metamodelo debe conformar a su metametamodelo. En esta arquitectura de tres capas (modelos, metamodelos y metametamodelos), el metametamodelo usualmente conforma a su propia semántica, es decir, éste puede ser definido utilizando sus propios conceptos.

El considerar los modelos como entidades de primer orden requiere contar con un conjunto de herramientas para definir transformaciones sobre los mismos. Una transformación de modelos provee facilidades para generar un modelo Mb, conforme a un metamodelo MMb, desde un modelo Ma conforme a un metamodelo Mb.

6.4. Metamodelado y lenguajes de modelado

6.4.1. La arquitectura de cuatro capas de modelado del OMG

La arquitectura de cuatro capas de modelado es la propuesta del OMG orientada a estandarizar conceptos relacionados al modelado, desde los más abstractos hasta los más concretos. Los cuatro niveles definidos en esta arquitectura se denominan: M3, M2, M1 y M0.

Nivel M0: Instancias

En el nivel M0 se encuentran todas las instancias "reales" del sistema, es decir, los objetos de la aplicación.

Nivel M1: Modelo del sistema

Por encima de la capa M0 se sitúa la capa M1, que representa el modelo de un sistema de software. Los conceptos del nivel M1 representan categorías de las instancias de M0. Es decir, cada elemento de M0 es una instancia de un elemento de M1. Sus elementos son modelos de los datos.

Nivel M2: Metamodelo

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

Nivel M3: Meta-metamodelo

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

M3 es el nivel más abstracto, que permite definir metamodelos concretos. Dentro del OMG, MOF es el lenguaje estándar de la capa M3. Esto supone que todos los metamodelos de la capa M2, son instancias de MOF. No existe otro nivel por encima de MOF. Básicamente, el MOF se define a sí mismo.

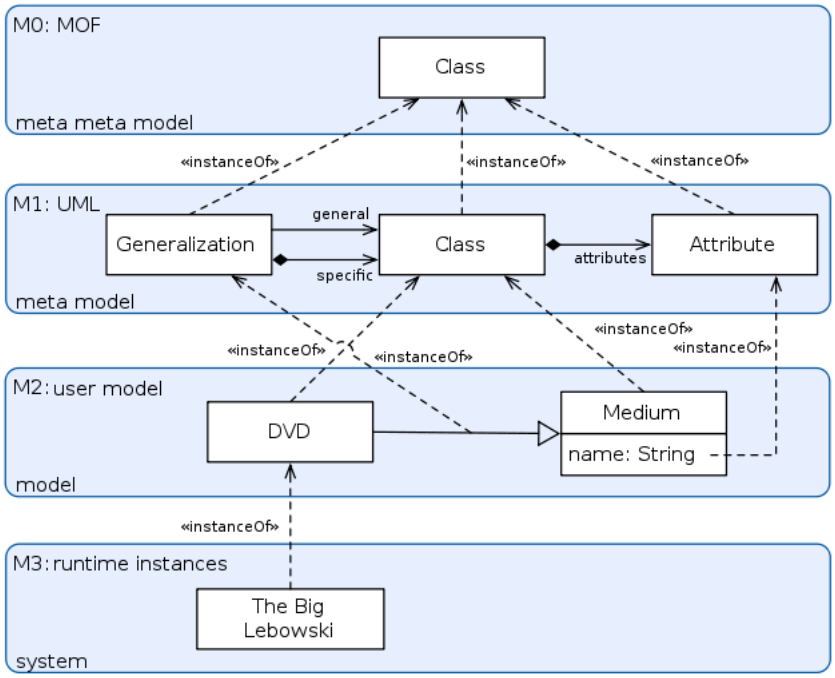

Figura 6.1: Jerarquía de metamodelos MOF

6.4.2. Importancia del metamodelado en MDD

Las razones por las que el metamodelado es importante en MDD son las siguientes:

- Necesitamos contar con un mecanismo para definir lenguajes de modelado sin ambigüedades, y permitir que una herramienta de transformación pueda leer, escribir y entender/procesar los modelos.
- Las reglas de una transformación, que constituyen una definición de una transformación, utilizan los metamodelos de los lenguajes fuente y destino para definir una transformación.
- La sintaxis de los lenguajes en los cuales se expresan las reglas de transformación debe estar formalmente definida, para permitir su automatización. Para especificar dicha sintaxis se utiliza la técnica de metamodelado.

6.4.3. MOF

Introducción

El lenguaje MOF es un estándar del OMG para el desarrollo dirigido por modelos. MOF se encuentra en la capa superior de la arquitectura de cuatro capas. Provee un meta-metalenguaje que permite definir modelos en la capa M2. Un ejemplo de un elemento en la capa M2 es el metamodelo M2, que describe al lenguaje UML.

MOF es una arquitectura de metamodelado cerrada y estricta. Es cerrada porque el metamodelo MOF se define en términos de sí mismo. Y es estricta porque cada elemento de un modelo en cualquiera de las capas tiene una correspondencia estricta con un elemento del modelo de la capa superior.

La definición de MOF está separada en dos partes fundamentales, Essential MOF (EMOF) y Complete MOF (CMOF), y se espera que en el futuro se agregue Semantic MOF (SMOF).

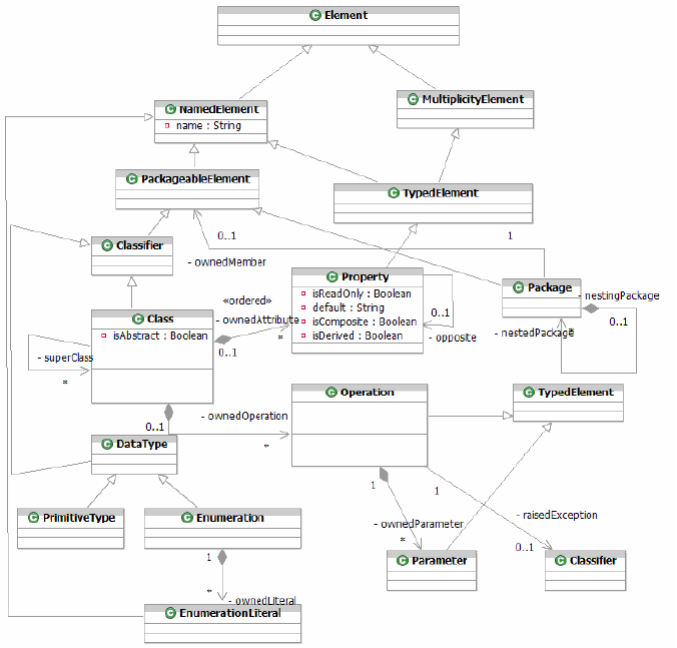

Figura 6.2: Sintaxis abstracta de MOF

Implementación de MOF Ecore

El metamodelo MOF está implementado mediante un plugin de Eclipse llamado Ecore. Este plugin respeta las metaclases definidas por MOF. Todas las metaclases mantienen el nombre del elemento que implementan y agregan como prefijo la letra “E”, indicando que pertenecen al metamodelo Ecore.

La primera implementación de Ecore fue terminada en Junio de 2002. Esta primera versión usaba como meta-metamodelo la definición estándar de MOF (v1.4). Gracias a las sucesivas implementaciones de Ecore y con la experiencia ganada con el uso de esta implementación en varias herramientas, el metalenguaje evolucionó.

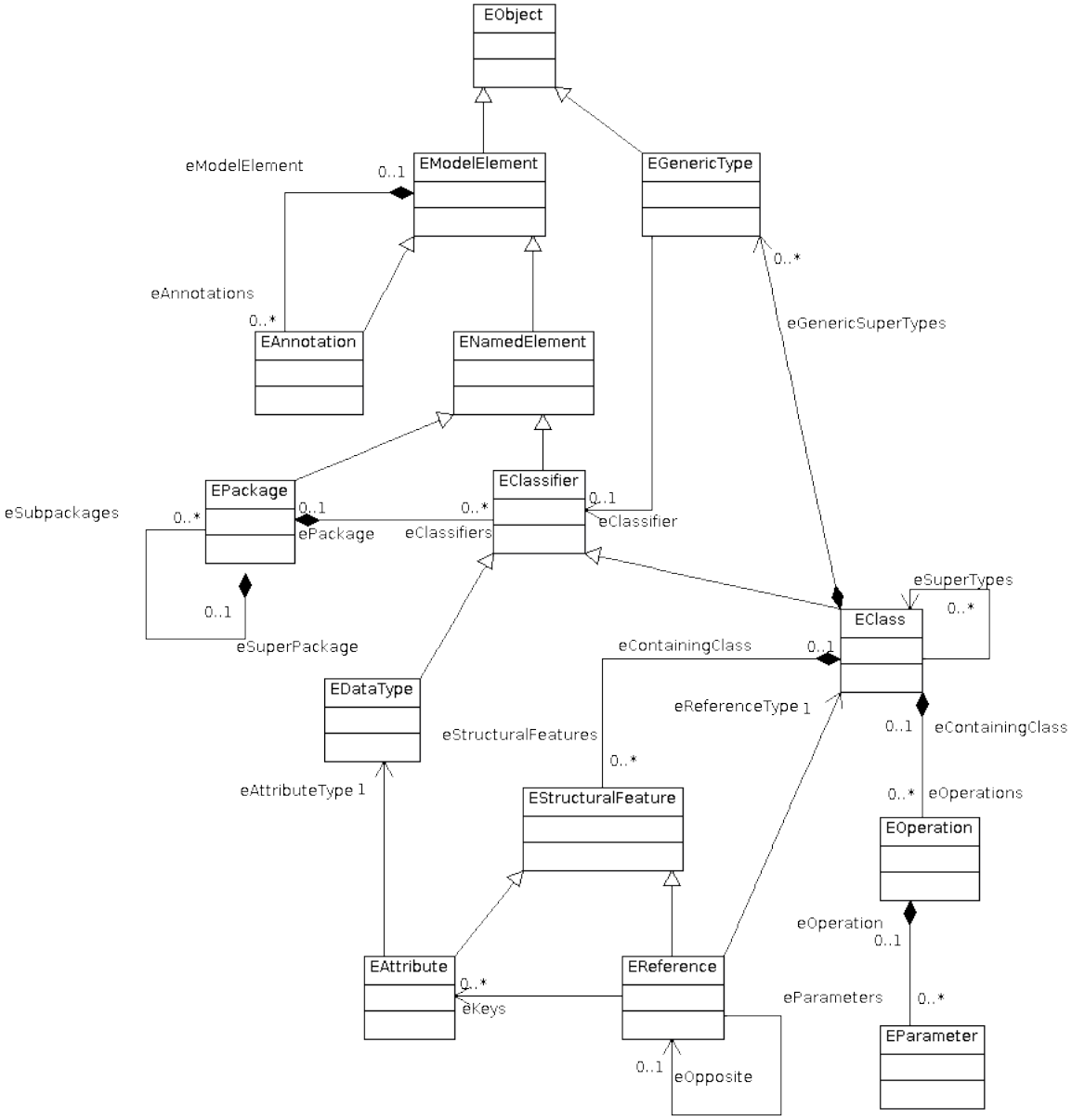

Figura 6.3: Metamodelo simplificado de Ecore

Ejemplos de metamodelos

6.4.4. El rol de OCL en el metamodelado

En el Capítulo 2 presentamos y analizamos el lenguaje OCL. Vimos que OCL es útil para definir restricciones sobre los elementos de un modelo. OCL puede utilizarse para expresar cuáles son las condiciones estructurales que las construcciones del lenguaje deben satisfacer para estar bien formadas. Estas reglas pueden definirse en cada uno de los niveles de la arquitectura de cuatro capas.

6.5. Transformaciones modelo a texto

Las transformaciones modelo a modelo crean su modelo destino como una instancia de un metamodelo específico, mientras que el destino de una transformación modelo a texto (M2T) es simplemente un documento en formato texto, generalmente de tipo String. En las subsecciones siguientes presentamos la motivación para este tipo de lenguajes y enumeramos los requisitos que los lenguajes de este tipo deberían satisfacer. Luego introducimos el lenguaje estándar Mof2Text especificado por el OMG para transformaciones M2T [49].

6.5.1. Motivación

La existencia de lenguajes para transformaciones modelo a texto nos permite:

- Elevar el nivel de abstracción
- Automatizar el proceso de desarrollo de software en las etapas finales
- Generar automáticamente nuevos artefactos desde nuestros modelos

6.5.2. Requisitos para lenguajes M2T

En abril de 2004 el OMG publicó un Request for Proposal (RFP) para transformaciones [47]. El RFP definía requisitos específicos que las propuestas debían satisfacer.

Requisitos obligatorios

1. Generación de texto a partir de modelos MOF 2.0.
2. Las transformaciones deberán estar definidas en el metanivel del modelo fuente.
3. Soporte para conversión a String de los datos del modelo.
4. Manipulación de strings.
5. Combinación de los datos del modelo con textos de salida.
6. Soporte para transformaciones complejas.
7. Varios modelos MOF de entrada (modelos fuente múltiples).

Requisitos opcionales

1. Ingeniería *round-trip* (de ida y vuelta).
2. Detección y protección de cambios manuales, para permitir regeneración.
3. Rastreo de elementos desde texto, si es necesario para poder soportar los últimos dos requisitos.

Como propuesta inicial al RFP modelo a texto del OMG, surge el lenguaje MOFScript, que cuenta con una herramienta del mismo nombre [44].

El estándar Mof2Text para expresar transformaciones modelo a texto

El lenguaje MOF Model to Text Transformation Language (Mof2Text) es el estándar especificado por el OMG para definir transformaciones modelo a texto. Sus metaclases principales son extensiones de OCL, de QVT y de MOF.

6.6. Resumen

En este capítulo se vio una introducción a MDD, particular de los siguientes puntos:

- Los diferentes tipos de abstracción de los modelos, escritos en lenguajes estándar. Las cualidades de los mismos, que deben ser consistentes, completos, precisos, comprensibles, reusables, flexibles y correctos.
- Debe proveerse un lenguaje en el cual describir la definición de las transformaciones.
- Los mecanismos para definir la sintaxis de los lenguajes de modelado. Entra en juego el "metamodelado" que describe la sintaxis abstracta del lenguaje.
- La arquitectura de capas del modelado de OMG. En la cual cada nivel se instancia a partir del nivel inmediato superior. La arquitectura de cuatro capas de modelado es la propuesta del OMG orientada a estandarizar conceptos relacionados al modelado, desde los más abstractos a los más concretos. Los cuatro niveles definidos en esta arquitectura se denominan: M3, M2, M1, M0. concreta.
- El lenguaje MOF es un estándar del OMG para el desarrollo dirigido por modelos. MOF se encuentra en la capa superior de la arquitectura de cuatro capas. Provee un meta-metalenguaje que permite definir modelos en la capa M2.
- También se vio la necesidad de contar con un lenguaje formal como OCL, que permite agregar mayor precisión en los modelos y metamodelos.
- Por último se vio una introducción a las transformaciones de modelo a texto, junto con los requisitos que debe cumplir.

Capítulo 7

Plataforma Eclipse

7.1. Introducción

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). [11]

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 ej. HTML, Java, C, JSP, EJB, Extensible Markup Language (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 y 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 Application programming interface (API) de interfaces bien definida, clases y métodos.

7.2. Estructura de la plataforma

La plataforma Eclipse está estructurada como subsistemas que están implementados como uno o más plugins. En la Figura 7.1 se muestra una vista simplificada.

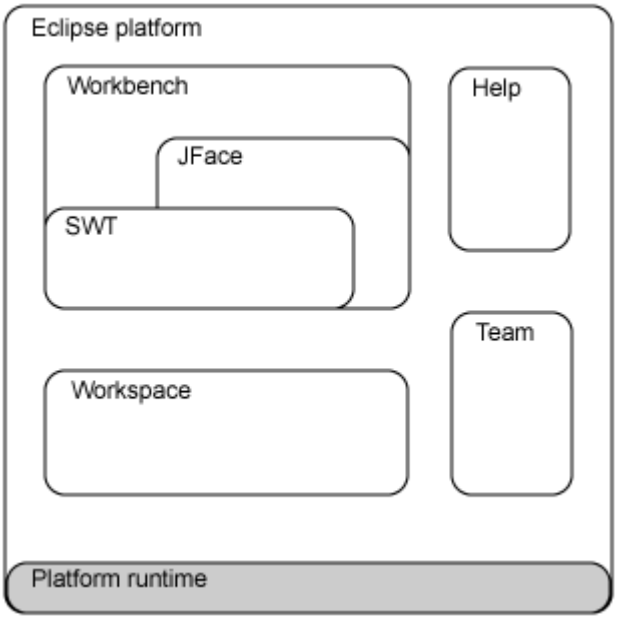

Figura 7.1: Vista simplificada de la plataforma Eclipse

Los plugins que forman un subsistema definen puntos de extensión para agregar comportamiento a la plataforma. En la Tabla 7.1 se describen los componentes de la plataforma que están implementados como plugins.

7.3. Arquitectura de la plataforma plugins

La plataforma Eclipse está estructurada alrededor del concepto de plugins. Los plugins son paquetes (bundles) estructurados de código que proveen funcionalidad al sistema. Las funcionalidades se pueden contribuir en forma de librerías de código (clases Java con API pública), extensiones de la plataforma e incluso documentación. Los plugins pueden definir puntos de extensión, que son partes bien definidas donde otros plugins pueden agregar funcionalidad. En la figura 7.2 se muestra la arquitectura de plugins de Eclipse junto con los puntos de extensión.

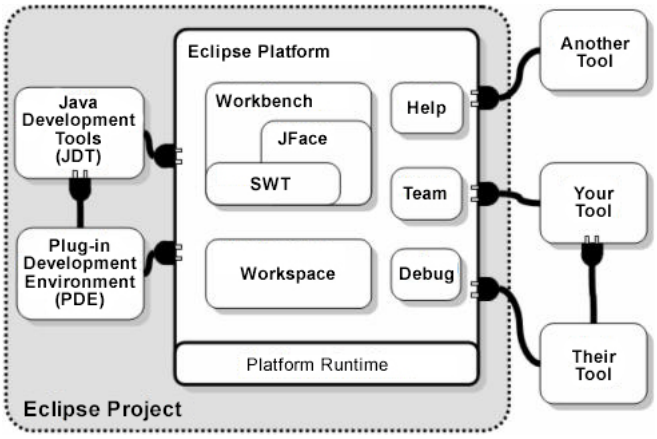

Figura 7.2: Arquitectura de plugins de plataforma

Platform runtime (Núcleo)	Define los puntos de extensión y el modelo de plugins. Descubre plugins dinámicamente y mantiene información sobre los plugins y sus puntos de extensión en un registro de la plataforma llamado manifest. Es implementado utilizando el framework OSGi.
Resource management (workspace o entorno de trabajo)	Define una API para crear y administrar recursos producidos por las herramientas y mantenidos en el sistema de archivos.
Workbench UI (Interfaz de usuario)	Implementa la interfaz de usuario para navegar por la plataforma. Define puntos de extensión para agregar componentes de UI tales como vistas o menús de acciones. Proporciona herramientas adicionales (JFace y SWT) para construir interfaces de usuario.
Help system (Sistema de ayuda)	Define puntos de extensión para que los plugins provean ayuda u otra documentación.
Debug support	Define un modelo de depuración independiente del lenguaje u clases de interfaz de usuario para construir debuggers y launchers.
Other utilities	Otros plugins utilitarios que proveen funcionalidad tales como búsqueda y comparación de recursos.

Tabla 7.1: Componentes de la plataforma Eclipse

7.4. Eclipse Modeling Framework

EMF es un framework de modelado para Eclipse y generador de código utilizado para el desarrollo de herramientas y aplicaciones. A partir de una especificación de un modelo descripto en XMI, EMF provee herramientas y soporte en tiempo de ejecución para producir un conjunto de clases Java para el modelo, junto con un conjunto de clases adaptadoras que permiten la visualización y edición basada en comandos del modelo y un editor básico. [10]

Los modelos pueden ser especificados usando notación Java, documentos XML, o herramientas de modelado como Rational Rose, y después ser exportados a EMF. Lo más importante de todo es que EMF suministra las bases para la interoperabilidad con otras herramientas y aplicaciones basadas en EMF.

El EMF incluye XML Schema Infoset Model (XSD, un nuevo componente del proyecto Model Development Tools (MDT), y una implementación basada en EMF de Service Data Objects (SDO). XML Schema Definition (XSD) provee un modelo y una API para manipular componentes de un esquema XML, con acceso a representación Document Object Model (DOM) del documento del esquema.

EMF consiste de tres piezas fundamentales:

- **EMF-Core**: El core del Framework EMF incluye un meta modelo (ECore) para describir modelos y soporte runtime para los modelos incluyendo notificaciones de cambio, soporte de persistencia con serialización XMI (XML Metadata Interchange) por defecto, y una muy eficiente API para manipular objetos EMF genéricamente.
- **EMF-Edit**: El Framework EMF.Edit incluye clases reutilizables genéricas para construir editores para modelos EMF. Esto provee:
  - Clases proveedoras de contenido y etiquetas, y otras clases que permiten a los modelos EMF ser mostrados usando visualizadores de escritorio estándar.

    - Un Framework de comandos, incluyendo un conjunto de clases de implementación de comandos genéricos para editores de construcción que soporten completamente el "rehacer" y "deshacer" automáticos.

- **EMF-Codegen**: La generación de código EMF es capaz de generar todo lo necesario para construir un editor completo para un modelo EMF. Esto incluye un GUI desde el cual pueden ser especificadas las opciones de generación, y los generadores a ser invocados. La generación se basa en Java Development Tooling (JDT), un componente de Eclipse. Hay tres niveles de generación de código que son soportadas:

    - **Modelo**: Proporciona clases Java de implementación e interfaz para todas las clases del modelo, además una clase de implementación de factory y package (meta data).
    - **Adaptadores**: Genera clases de implementación (llamadas `ItemsProviders`) que adaptan las clases del modelo para ser editadas y mostradas.
    - **Editor**: Produce un editor estructurado adecuadamente que se ajusta al estilo recomendado por los editores de modelos EMF Eclipse y sirve como punto de partida al comienzo de la personalización.

7.5. Java

Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems a principios de los años 90. El lenguaje en sí mismo toma mucha de su sintaxis de C y C++, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como por ejemplo la manipulación directa de punteros o memoria.

Las aplicaciones Java están típicamente compiladas en un bytecode, aunque la compilación en código máquina nativo también es posible. En el tiempo de ejecución, el bytecode es normalmente interpretado o compilado a código nativo para la ejecución, aunque la ejecución directa por hardware del bytecode por un procesador Java también es posible.

El lenguaje Java se creó con cinco objetivos principales:

1. Debería usar la metodología de la programación orientada a objetos.
2. Debería permitir la ejecución de un mismo programa en múltiples sistemas operativos.
3. Debería incluir por defecto soporte para trabajo en red.
4. Debería diseñarse para ejecutar código en sistemas remotos de forma segura.
5. Debería ser fácil de usar y tomar lo mejor de otros lenguajes orientados a objetos, como C++.

7.6. UML2 Plugin

UML2 es una implementación basada en EMF del metamodelo OMG de UML 2 para Eclipse. Los objetivos del plugin UML2 son:

- Proveer una implementación del metamodelo UML utilizable para permitir el desarrollo de herramientas de modelado.
- Proveer un esquema común XMI para facilitar intercambio de modelos semánticos.
- Proveer casos de prueba para validar la especificación.
- Proveer reglas de validación como forma de definir y hacer cumplir los niveles de obediencia.

UML2 permite el desarrollo de modelos UML tanto programáticamente como gráficamente (utilizando el editor de modelos UML). Aunque UML2 provee el metamodelo, no provee herramientas de modelado.

7.7. OCL Plugin

Eclipse OCL es una implementación del estándar OMG de OCL 2.2 para modelos basados en EMF. Los componentes principales proveen las siguientes funcionalidades para proveer integración OCL:

- Define APIs para análisis sintáctico y evaluación de restricciones OCL y consultas sobre modelos Ecore o UML.
- Define implementaciones OCL y Ecore de la sintaxis abstracta OCL, incluyendo soporte para la serialización de expresiones OCL.
- Provee una API que implementa el patrón Visitor para analizar y transformar el árbol sintáctico abstracto de las expresiones OCL.
- Provee una API de extensión para que los clientes puedan personalizar (customizar) los entornos de análisis y evaluación utilizados por el parser.

La implementación de OCL está definida en plugins para un fácil despliegue en Eclipse, pero al igual que EMF puede usarse solo. Los plugins se dividen de la siguiente manera:

- **org.eclipse.ocl**: El núcleo de los servicios de análisis sintáctico, evaluación y autocompletado. Define el modelo de sintaxis abstracta de OCL y la API Environment. Estas APIs son genéricas, independientes de cualquier metamodelo particular (aunque utiliza Ecore/EMF como metamodelo).
- **org.eclipse.ocl.ecore**: Es una implementación del entorno del metamodelo Ecore que une la clase `Environment` genérica y las APIs del Abstract Syntax Tree (AST) con el lenguaje Ecore. Provee soporte para trabajar con restricciones OCL y consultas que utilizan modelos Ecore.
- **org.eclipse.ocl.uml**: Es una implementación del entorno del metamodelo UML que une la clase `Environment` genérica y las APIs del AST con el lenguaje UML. Provee soporte para trabajar con restricciones OCL y consultas que utilizan modelos UML.

7.8. Resumen

En este capítulo se vio una presentación de la plataforma y algunos de sus plugins, en particular:

- Eclipse: es una plataforma de software de código abierto independiente de una plataforma para desarrollar aplicaciones. 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 con un modelo de plugins.
- Framework EMF: es un framework de modelado para Eclipse y generador de código utilizado para el desarrollo de herramientas y aplicaciones. A partir de una especificación de un modelo descripto en XMI, EMF provee herramientas y soporte en tiempo de ejecución para producir un conjunto de clases Java para el modelo, junto con un conjunto de clases adaptadoras que permiten la visualización y edición basada en comandos del modelo y un editor básico.
- Y por último se mencionó el lenguaje Java y los plugins de OCL y UML.

Capítulo 8

MOFScript

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. [44]

No sólo es un estándar, sino que además se basa en otros estándares de la OMG: es Query/-View/Transformation (QVT) compatible y 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.

8.1. Arquitectura de cuatro capas de MOFScript

En la figura 8.1 se muestra la arquitectura de MOFScript.

8.2. Constructores de MOFScript

1. **Texttransformation**

   Texttransformation define el nombre del módulo, que puede ser cualquier nombre, independientemente del nombre del archivo. Puede usarse *texttransformation* o *textmodule*.

   Define el input del metamodelo en términos de un parámetro

```
texttransformation testAnnotations (in
   uml:"http://www.eclipse.org/uml2/1.0.0/UML"))
```

   Un texttransformation puede tener muchos modelos como parámetros de entrada, los cuales deben estar separados por coma.

```
texttransformation TransformationWithSeveralMetaModels (in
    uml:"http://www.eclipse.org/uml2/1.0.0/UML",
    in ec:"http://www.eclipse.org/emf/2002/Ecore") {
      //transformation
  }
```

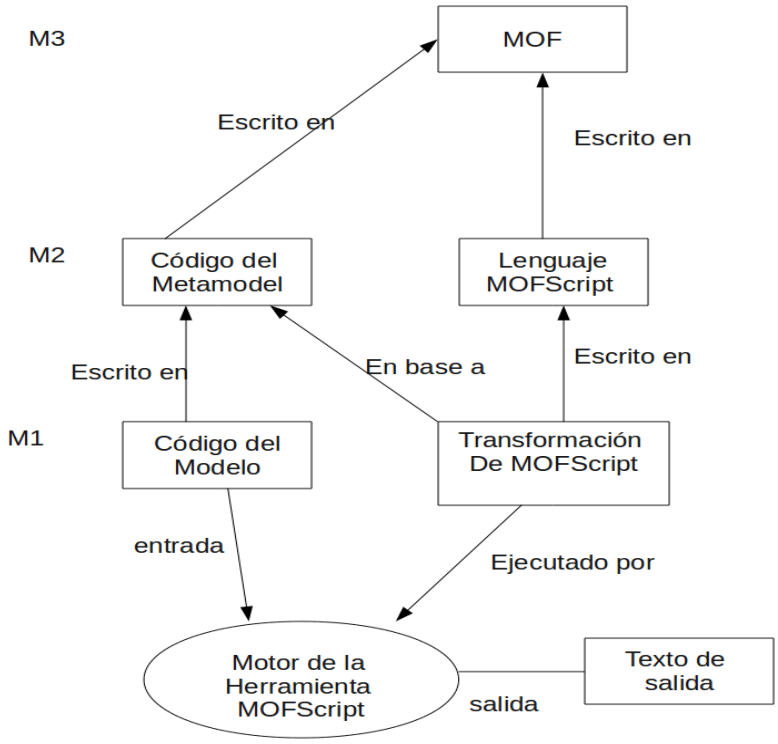

Figura 8.1: Arquitectura de cuatro capas de MOFScript

2. **Imports**

Un módulo de transformación, puede importar otras transformaciones. Las transformaciones importadas son obtenidas por el parser, primero mirando en el *directorio actual* y después en el especificado por el *import path*.

```
import "/dir/transf.m2t"
```

3. **Reglas de punto de entrada**

Las reglas de punto de entrada definen donde la transformación comienza su ejecución. Es similar al Java main. El punto de entrada debe tener un contexto (en el ejemplo es uml.Model), que define que tipo de elemento del metamodelo que será el punto de comienzo para la ejecución. Su cuerpo contendrá declaraciones.

```
uml.Model::main () {
  self.ownedMember->forEach(p:uml.Package)
  {
    p.mapPackage()
  }
}
```

Un punto de entrada puede tener un tipo de contexto con varias instancias. En ese caso, el punto de entrada puede ser ejecutado por cada instancia de ese tipo. A continuación se muestra un ejemplo donde el contexto punto de entrada *uml.Class*

```
uml.Class::main () {
  'class: ' self.name
}
```

4. **Reglas**

Las reglas son básicamente como funciones. Pueden tener un tipo de contexto, el cual es de un tipo del metamodelo. Pueden también tener un tipo de contexto, el cual puede ser un tipo primitivo o un tipo de modelo. El cuerpo de la regla, contiene un conjunto de declaraciones.

```
uml.Package::mapPackage () {
  self.ownedMember->forEach(c:uml.Class)
  c.mapClass()
}
uml.Class::mapClass(){
  file (package_dir + self.name + ext)
  self.classPackage()
  self.est\'andarClassImport ()
  self.standardClassHeaderComment ()
  'public class ' self.name ' extends Serializable { '
    self.classConstructor()
  '
  /*
   * Attributes
  */
  '
  self.ownedAttribute->forEach(p : uml.Property) {
    p.classPrivateAttribute()
  }
  newline(2)
  '}'
}
```

a) **Valores de retorno**

Una regla también puede devolver un valor, el cual puede ser reusado en expresiones en otras reglas. Par retornar un valor se utiliza la declaración **result**.

```
uml.Package::getFullName (): String {
  if (self.owner != null){
    result = self.owner.getFullName() + "."
  } else if (self.ownerPackage != null) {
    result = self.ownerPackage.getFullName() + "."
    result += self.name.toLower().replace(" ", "_");
  }
}
```

O puede usarse cono una declaración 'return', que termina inmediatamente la ejecución de la regla y retorna el valor.

```
uml.Package::getFullName (): String {
  result self.owner.getFullName();
}
```

b) **Parámetros**

Una regla puede tener cualquier número de parámetros. Un parámetro puede ser un tipo primitivo o un tipo de metamodelo.

```
uml.Model::testParameters2 (s1:String, i1:Integer) {
   stdout.println("testParameters2: " + s1 + ", " + i1)
```

```
}

uml.Model::testParameters3 (s1:String, r2:Real, b1:Boolean,
   package:uml.Package) {
     stdout.println("Package:" + package.name)
     stdout.println ("testParameters3: " + s1 + ", " + r2 +
   ", " + b1 + " " + package.name)
}
```

5. **Propiedades y variables**

Las propiedades y variables pueden ser definidas global o localmente sin una regla o un bloque. Una propiedad es una constante que se asigna al valor en la declaración. El tipo puede ser cualquiera de los tipos primitivos, un tipo de modelo, o puede estar sin tipo en la declaración, y determinarse cuando se le asigne un valor.

Una variable puede cambiar su valor en las asignaciones durante tiempo de ejecución. Una variable puede tener el tipo de cualquiera de los tipos primitivos. También puede estar sin tipo en la declaración, y determinarse cuando se le asigne un valor. Si no se le asigna ningún valor, el tipo será 'String'

```
property packageName:String = "org.mypackage"
var myInteger = 7
```

6. **Tipos primitivos**

Los tipos primitivos en MOFScript se resumen a continuación:

a) `String`: El tipo `string`, que representa los valores de texto.

b) `Integer`: El tipo `integer`.

c) `Real`: El tipo `real`.

d) `Boolean`: El tipo `boolean`.

e) `Hashtable`: El tipo `Hashtable`.

f) `List`: El tipo `List`.

g) `PropertyMap`: Es similar a la clase de java `Properties`. Puede ser usado para cargar y almacenar propiedades al estilo java.

h) `Object`: El tipo `object` puede representar cualquier tipo.

7. **Archivos**

Las declaraciones de archivos son declaraciones de un dispositivo de salida para texto. Se usa con la palabra clave **file**. El nombre del archivo y su extensión se pasan como parámetro. Se puede pasar como parámetro el parth relativo o absoluto de la salida.

```
file (c.name + ".java")
file ("c:\tmp\" + c.name + ".java")
file f2 ("test.java")
f2.println ("\t\t output to file f2")
```

8. **Declaraciones de impresión**

Proveen una salida a un dispositivo de salida, que puede ser un archivo o una salida estándar.

```
println ("public class" + c.name);
```

Si no hay declarado un archivo, la salida estándar se utiliza como salida. Para forzar la salida standar debe escribirse el prefijo **stdout**.

```
stdout.println ("public class" + c.name);
```

9. **Iteradores**

Los iteradores en MOFScript se utilizan para iterar colleciones de elementos del modelo desde un modelo fuente. La declaración **forEach** define un iterador sobre una colección, ya sea un elemento de la colección de un modelo, una lista o hashtable,o un String/Integer.

Una declaración del **forEach** puede ser restringida por un tipo de constraint ($collection -> forEach(c : someType)$), donde el tipo puede ser un tipo del metamodelo o un tipo primitivo. Si no se da el tipo constraint, se aplica a todos los elementos de la colección. Una declaración **forEach**, puede tener también una guarda, que es básicamente un tipo de expresión booleana. Un constraint se describe luego del tipo utilizando la barra vertical ('|') (collection -> **forEach** (a:**String** | a.**size**() = 2).

```
-- applies to all objects in the collection of type Operation
c.ownedOperation->forEach(o:uml.Operation) {
  -- statements.
}
-- applies to all objects in the collection
-- of type Operation that has a name that starts with 'a'
 c.ownedOperation->forEach(o:uml.Operation | o.name.startsWith(\a"))
 {
  /* statements */
 }
// applies to all operation elements in the collection that
// has more than zero parameters and a return type
c.ownedOperation->forEach(o:uml.Operation | o.ownedParameter.size()
  > 0 and o.returnResult.size() > 0) {
 /* statements */
}
```

a) Iteradores para variables List y Hashtable

Los iteradores se pueden utilizar para variables de tipo List/Hashtable como sigue:

```
var list1:List
list1.add("E1")
list1.add("E2")
list1.add(4)
list1->forEach(e){
  stdout.println (e)
}
```

b) Iteradores para Strings

Los iteradores para `String` definen loops sobre el contenido de los caracteres del string.

```
var myVar: String = "Jon Oldevik"
myVar->forEach(c)
stdout.print (c + " ")
```

c) Iteradores para Integers

Los iteradores para integers definen loops basados en el tamaño del contexto del integer. Por ejemplo integer '3' producirá un loop que se ejecutará 3 veces.

```
property aNumber:Integer = 34
aNumber->forEach(n)
stdout.print(" " + n)
```

d) Iteradores para String e Integer literales

Los iteradores pueden definirse usando `String` o `Integer` literales. Funciona de la misma forma que los iteradores basados en propiedades/variables de `String` e `Integer`.

```
"MODELWare, the MDA(tm) project"->forEach(s){
  stdout.print ("" + s)
  5->forEach(n){
    stdout.println (" " + n)
  }
}
```

10. **Declaraciones condicionales**

Las declaraciones condicionales son las declaraciones del **if**. Están definidas por una simple rama **if**, seguida de un conjunto **else-if** y una posible rama **else**. Los argumentos de las ramas son expresiones booleanas.

```
if (c.hasStereoType (\entity")) {
  // statements
} else if (c.hasStereoType(\service")) {
  // statements
} else {
  // statements
}
if (c.ownedOperations.size() > 0 and c.name.startsWith(\C")) {
  // statements
} else {
  // statements
}
```

11. **Declaraciones de while**

Las declaraciones de **while** funcionan de la misma manera que lo hacen en Java. La palabra clave está seguida por un constraint de cualquier tipo booleano. Por ejemplo:

```
 var i : Integer = 10
 while (i > 0){
  //Statements
   i -=1
}
```

Este ejemplo itera nueve veces antes de terminar su ejecución.

12. **Expresiones selectivas**

Una expresión select realiza una query sobre una colección y retorna una lista que contiene el resultado de esa query. La sintaxis es similar al **forEach**. Toma un tipo como parámetro y puede tener un constraint.

```
var xList:List = self.eClassifiers->select(c:ecore.EClass)
‘Number of classes: ’ xList.size()
xList->forEach(clazz:ecore.EClass) {
  ‘\n \t Class: ’ clazz.name
}
var yList:List = self.eClassifiers->select(c:ecore.EClass |
  c.name.startsWith("MOF"))
```

13. **Expresiones lógicas**

Las expresiones lógicas sin expresiones que evalúan a true o false y son usadas en las declaraciones de iteradores y condicionales.

```
self.ownedAttribute->forEach(p : uml.Property |
  p.association != null){
  // statements
}
 if (self.name = "Car" or self.name = "Person") {

}
```

14. **Operaciones primitivas**

Obtienen el EResource asociado de un Eobject

a) **Operaciones de strings**

Se detallan a continuación las operaciones:

1) `substring(lower: int, upper: int): String`.
   Retorna el substring desde el índice *lower* hasta el índice *upper*.
2) `subStringBefore(beforeString: String): String`.
   Retorna el substring del string que se encuentra antes de $beforeString$.
3) `subStringAfter(afterString: String): String`.
   Retorna el substring del string que se encuentra luego de $afterString$.
4) `toLower(): String`.
   Convierte el string a minúsculas.
5) `toUpper(): String`.
   Convierte el string a mayúsculas.
6) `firstToUpper(): String`.
   Convierte la primer letra del string a mayúsculas.
7) `firstToLower(): String`.
   Convierte la primer letra del string a minúsculas.
8) `size(): int`.
   Retorna el tamaño del string.
9) `indexOf(indexStr: String) : int`.
   Retorna el índice de la primer ocurrencia de *indexStr* o -1 si no existe.

10) `endsWith(str: String) : Boolean`.
Retorna `true` si el string termina con *str*.
11) `startsWith(str: String) : Boolean`.
Retorna `true` si el string comienza con *str*.
12) `trim(): String`.
Elimina todos los espacios en blanco del principio y final del string.
13) `normalizeSpace(): String`.
Realiza un trim del strinf y reemplaza todas las secuencias de espacios en blanco por un sólo carácter de espacio en blanco.
14) `replace(replaceWhat:String, withWhat:String): String`.
Reemplaza todas las ocurrencias que coinciden con la expresión regular *replaceWhat* con el string *withWhat*.
15) `equals(str: String): Boolean`.
Retorna `true` si el string es igual a *str*.
16) `equalsIgnoreCase(str: String): Boolean`.
Retorna `true` si el string es igual a *str* ignorando mayúsculas y minúsculas.
17) `isUpperCase(int index): Boolean`.
Retorna `true` si el carácter en la posición *index* está en mayúsculas. Si no se pasa un índice, se toma la primer posición.
18) `isLowerCase(int index): Boolean`.
Retorna `true` si el carácter en la posición *index* está en minúsculas. Si no se pasa un índice, se toma la primer posición.
19) `charAt(int index): String`.
Retorna el carácter de la posición *index*.
20) `forEach()`.
Es una operación de iteración para Strings, itera sobre cada carácter del string.
21) `matches(regexp): Boolean`.
Chequea si el string coincide con la expresión regular *regexp*.
22) `first(int i): String`.
Retorna los primeros i caracteres del string.
23) `strcmp(String other): Integer`.
Compara dos strings.
24) `last(int i): String`.
Retorna los últimos i caracteres del string.
25) `isAlpha(): Boolean`.
Chequea si el string tiene solo letras.
26) `isAlphaNum(): Boolean`.
Chequea si el string tiene letras y números.

Ejemplo

```
"myString".toLower()
c.name.size()
c.name.endsWith("Fa")
```

b) **Operaciones de Integer**

Las operaciones de Integer son:

1) Las aritméticas estándar: +, -, * , /.
2) `forEach()`: Operación de iteración para enteros. Itera sobre el tamaño del integer (desde 0 hasta su tamaño).

c) **Operaciones de List** Las operaciones de lista son:

1) `add (e:Object) : Boolean` . Agrega un objeto a la lista.
2) `remove (e:Object) : Boolean` . Elimina un objeto de la lista.
3) `size () : Integer` . Retorna el tamaño de la lista.

4) `clear () : void` . Vacía la lista.
5) `first () : Object` . Devuelve el primer elemento de la lista.
6) `last () : Object` . Devuelve el último elemento de la lista.
7) `isEmpty () : Boolean` . Retorna `true` si la lista está vacía.
8) `forEach () [iterator operation]`. Mecanismo de iteración aplicado a listas.
9) `select (x : type | condition)` . Selecciona un subconjunto de objetos de la colección.
10) `addAll (list)` . Agrega los objetos en la lista de *list*, al final de la misma.
11) `addAllFirst (list)` . Agrega los objetos en la lista de *list* al principio de la misma.
12) `addBefore (item, list)` . Agrega los objetos en *list* antes de *item* que ya está en la lista.
13) `addAfter (item, list)` . Agrega los objetos en *list* después de *item* que ya está en la lista.
14) `indexOf (element) : Integer` . Retorna el índice del elemento dado.

d) **Operaciones de Hashtable**

Las operaciones de hashtable son:

1) `put (key: Object, value: Object)` . Pone el elemento *value* con el valor *key* en la hashtable.
2) `get (key: Object)` . Obtiene el valor asociado con el parámetro *key*.
3) `size () : Integer` . Retorna el tamaño de la hashtable.
4) `clear () : void` . Vacía la hashtable.
5) `keys () : List` . Retorna una lista de los *keys* de la hashtable.
6) `values () : List` . Retorna una lista de los valores de la hashtable.
7) `first () : Object` . Retorna el primer elemento de la hashtable.
8) `last () : Object` . Retorna el último elemento de la hashtable.
9) `isEmpty () : Boolean` . Retorna `true` si la lista está vacía.
10) `forEach () [iterator operation]` . Mecanismo de iteración aplicado a hashtable.

e) **Operaciones de PropertyMap**

Un PropertyMap es una especialización de Hashtable. Además ofrece las siguientes operaciones para cargar y almacenar propiedades.

1) load ("prop-file")
2) loadXML("xml-prop-file")
3) store ("prop-file")
4) storeXML("xml-prop-file")

f) **Operaciones de colecciones del modelo**

1) `size () : int` . Retorna el tamaño de la colección.
2) `first () : Object` . Retorna el primero objeto de la colección.
3) `isEmpty (): Boolean` . Chequea si la colección está vacía.
4) `forEach () [ Iterator operation]` . Mecanismo de iteración aplicado de colecciones del modelo.
5) `select (x : type | condition)` . Selecciona un subconjunto de los objetos de una colección.

Ejemplo

```
if (c.attributes.size() = 0) {
  stdout.println (\Size is 0")
}
c.attributes->forEach (p:uml.Property | p = c.attributes.first()) {
  stdout.println ("First attribute")
}
```

g) **Operaciones del modelo**

1) `objectsOfType (type) : Collection` . Retorna todas las instancias del tipo `type`.
2) `store (file uri)` . Almacena el modelo dado en el archivo úri'dado.

Ejemplo:

```
ec.objectsOfType (ec.EClass)->forEach (eccl) {
  ...
}
```

h) **Operaciones OCL**

1) `oclIsTypeOf (type: typeRef): Boolean` . Retorna `true` si el tipo es exactamente el mismo que el tipo de entrada.
2) `oclIsKindOf (type: typeRef) : Boolean` . Retorna `true` si el tipo es el mismo o un subtipo del tipo de entrada.
3) `oclGetType () : String` . Retorna el nombre del tipo.

8.3. Resumen

En este capítulo abordamos el tema de MOFScript y en particular:

- MOFScript es un lenguaje de transformación de modelo a texto.
- Los contructores de MOFScript Texttransformation, los imports, las reglas de punto de entrada, propiedades y variables, tipos primitivos, archivos, declaraciones de impresión, iteradores, declaraciones condicionales y de while, expresiones selectivas y lógicas, operaciones primitivas. Todos los constructores de MOFScript se vieron con una gran variedad de ejemplos. Así como las operaciones disponibles del lenguaje para los tipos.

Capítulo 9

Diseño e implementación de la herramienta

9.1. Introducción

En este capítulo describiremos el diseño e implementación de la herramienta. Explicaremos los componentes y etapas de la herramienta y la implementación de la librería de colecciones OCL.

9.2. Funcionalidad

La herramienta tiene cuatro funcionalidades principales, a saber:

1. **Parse OCL:** realiza un análisis sintáctico y semántico sobre un archivo con restricciones OCL.
2. **Translate UML model:** Dado un modelo UML genera 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.

9.3. Paquetes del plugin

El plugin está formado por los siguientes paquetes:

- `ar.edu.unlp.info.ocl2jml.handler`: Contiene las clases que implementan las acciones del plugin.
- `ar.edu.unlp.info.ocl2jml.jml.ast.basic`: Contiene las clases que implementan el árbol sintáctico abstracto de JML.
- `ar.edu.unlp.info.ocl2jml.jml.ast.extended`: Contiene las clases que implementan los métodos modelo del árbol sintáctico abstracto de JML
- `ar.edu.unlp.info.ocl2jml.jml.ast.operators`: Contiene las clases que implementan los operadores del árbol sintáctico abstracto de JML.
- `ar.edu.unlp.info.ocl2jml.ocl.collections`: Contiene las clases que implementan la librería de colecciones OCL.
- `ar.edu.unlp.info.ocl2jml.translator`: Contiene las clases que realizan la traducción de OCL a JML.
- `ar.edu.unlp.info.ocl2jml.translator.symbols`: Contiene las clases que implementan el manejo de la tabla de símbolos para el traductor.

9.4. Estructura general

La herramienta está formada por tres componentes principales que se describen a continuación y que se muestra en la Figura 9.1:

- **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.

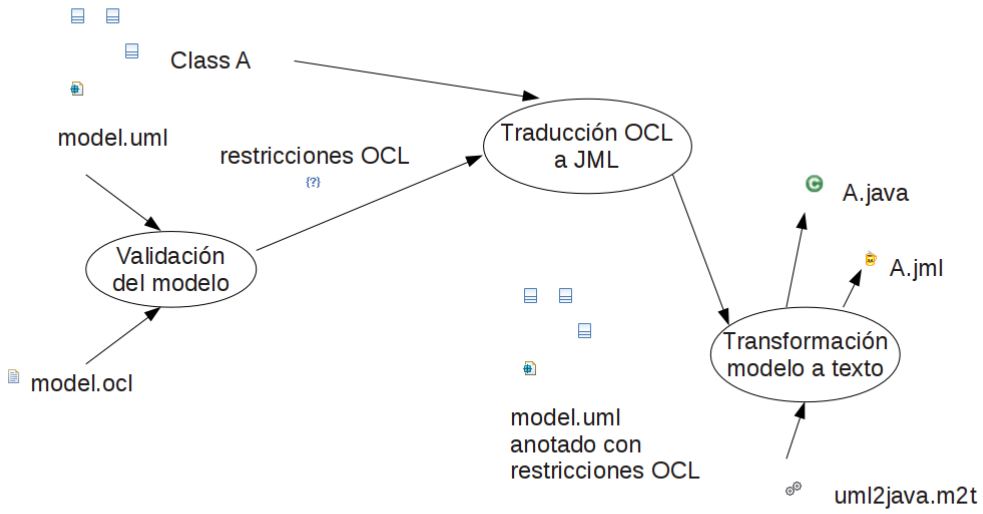

Figura 9.1: Estructura de la herramienta

9.5. Etapas de la derivación

9.5.1. Análisis del modelo

En esta etapa se recibe un modelo UML (creado con una herramienta del estilo Papyrus) y un archivo de texto con extensión *.ocl* que contiene las restricciones del modelo UML. Como resultado del análisis devuelve una lista de restricciones según el modelo de la Figura 9.2, es decir, devuelve un árbol sintáctico abstracto basado en el metamodelo de OCL. Esta tarea es delegada completamente al plugin de OCL debido a que es una de las funcionalidades de dicho plugin, en el apéndice C se describen las clases del plugin que implementan el metamodelo OCL.

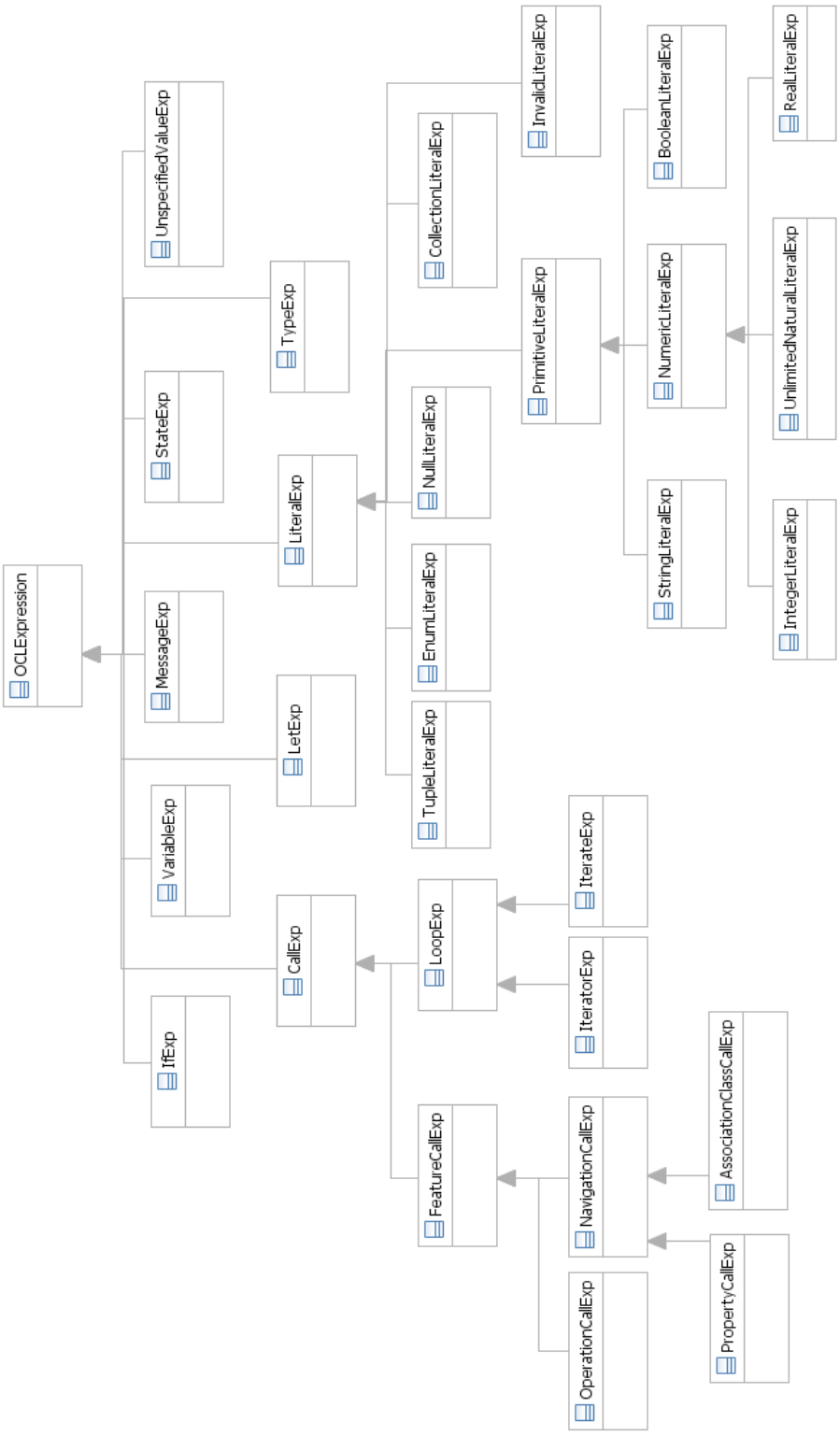

Figura 9.2: Implementación del metamodelo

9.5.2. Traducción de las restricciones a OCL

Esta es la etapa principal. Toma como entrada el modelo UML verificado en la etapa anterior y la lista de restricciones OCL obtenidas desde el archivo. Para cada una de las restricciones la transforma en su equivalente en JML, y las agrega a una copia del modelo de entrada.

La estructura del componente para la traducción de una restricción OCL utiliza la clase `OCLTranslator` que para cada restricción del archivo ocl realiza la traducción a JML.

OCL2JMLVisitor

La clase principal que implementa la traducción es `OCL2JMLVisitor`(Figura 9.3). Esta clase extiende la clase `AbstractVisitor` definida en el plugin OCL [33]. La clase `AbstractVisitor` implementa el patrón `Visitor` [32] y se encarga de recorrer la expresión de tipo `ExpressionInOCL`. Como resultado del recorrido del árbol sintáctico se obtiene una expresión de tipo `JMLSimpleExpression`.

AbstractVisitor

La clase `AbstractVisitor` está definida en el plugin de OCL de Eclipse, implementa el patron Visitor y se encarga de recorrer una expresión del árbol sintáctico glsocl. Existen dos tipos de nodos en una expresión OCL: los nodos hoja y los nodos internos. Una clase que extienda `AbstractVisitor` sólo necesita redefinir selectivamante los métodos *handleXxxx* para los nodos internos del árbol y los métodos *visitXxxx* para los nodos hoja. En el apéndice C se describen los métodos definidos en la clase.

OCL2JMLTranslator

La clase `OCL2JMLTranslator` se encarga de realizar la traducción de operaciones (invocación a métodos) y de operadores. Realiza también la traducción de las colecciones OCL a las colecciones de la librería OCL.

Namespace

La clase `Namespace` implementa una tabla de símbolos para registrar las variables de la traducción. Se encarga de almacenar el nombre, el tipo de datos, el tipo de variable y el ámbito.

Scope

La clase `Scope` implementa un ámbito de la tabla de símbolos, posee la lista de variables del ámbito y el tipo de ámbitos.

9.5.3. Transformación del modelo a texto

Esta etapa tiene como objetivo la derivación de código fuente. Toma como entrada el modelo UML intermedio anotado con anotaciones JML y utilizando el engine de MOFScript y un archivo de transformación *.m2t*, genera los archivos Java.

9.6. Implementación del metamodelo JML

Para representar las anotaciones JML traducidas desde OCL, se definió el árbol sintáctico del lenguaje. La mayoría de las clases son la contraparte en JML de expresiones en OCL más ciertas construcciones que sólo aparecen en JML. Todas las clases deben implementar el método *toString* que retorna la representación de la expresión JML.

9.6.1. Expresiones

En el diagrama de clases de la Figura 9.4 se muestran los dos tipos de expresiones del metamodelo JML.

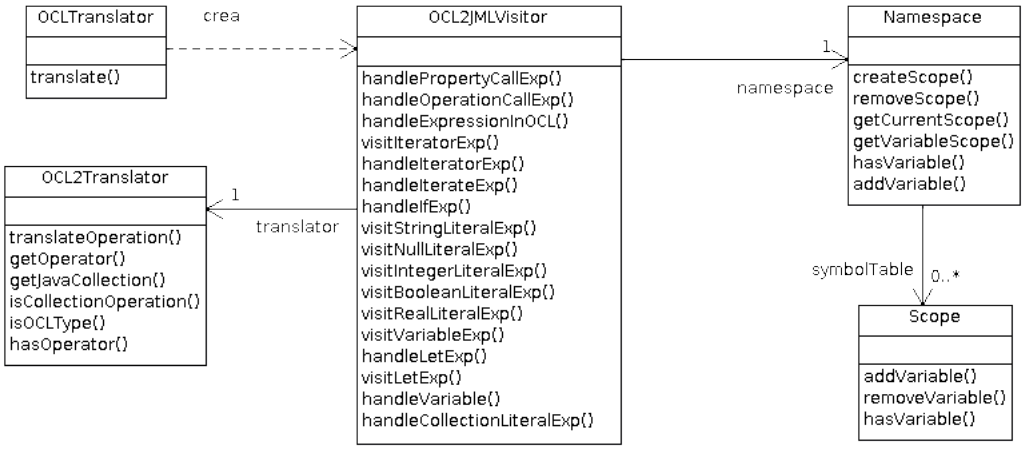

Figura 9.3: Estructura de clases del traductor

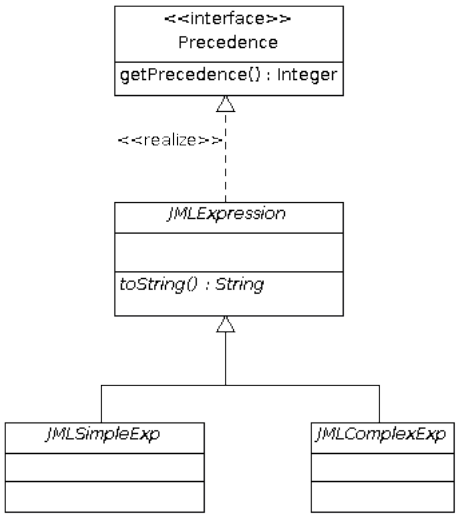

Figura 9.4: Estructura de expresiones JML

9.6.2. Operadores

En el diagrama de clases de las Figuras 9.5, 9.6, 9.7 se muestran los operadores definidos para JML. Los operadores definidos son los mismos de OCL, a los que se agrega el operador bicondicional (si y sólo si) y el operador de subtipo.

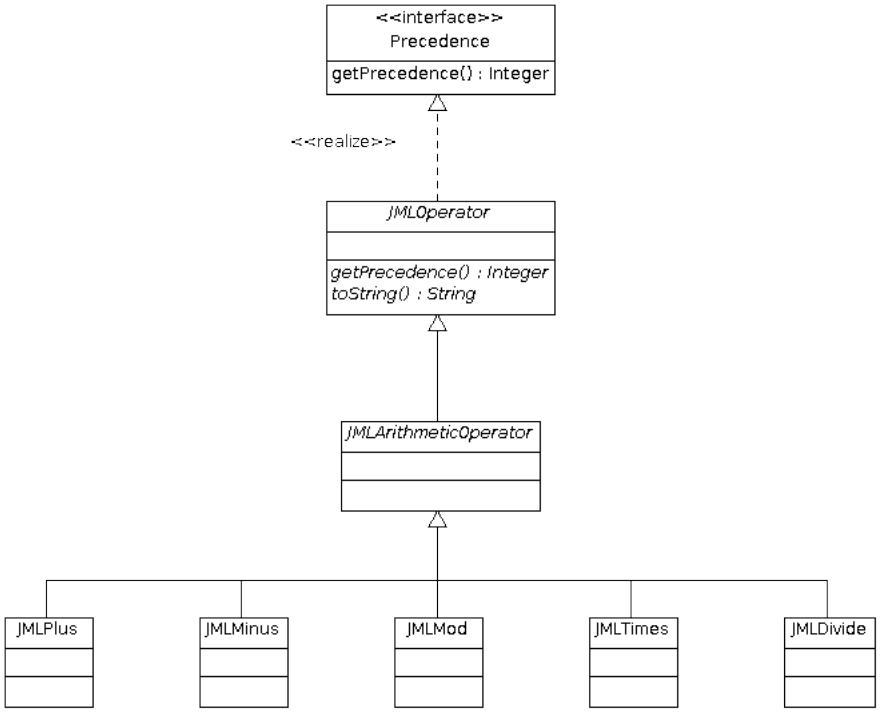

Figura 9.5: Metamodelo JML operadores aritméticos

9.6.3. Expresiones simples

En el diagrama de clases de las Figuras 9.8, 9.9, 9.10, 9.11 y 9.12 se muestran las expresiones simples que se pueden definir en JML, entre ellas, las expresiones literales, de llamada a propiedad, de llamada a operación, de valor previo en una postcondición, de objeto nuevo creado, expresiones aritmético-lógicas, de declaración y utilización de variables, y de declaración y utilización de tipos.

9.6.4. Expresiones complejas

En el diagrama de clases de la Figura 9.13 se muestran las expresiones complejas que se pueden definir en JML, entre ellas, la definición de invariante, de precondición y de postcondición.

9.7. Implementación de la librería de colecciones OCL

Una parte importante de la traducción de OCL a JML es la derivación de las colecciones y sus operaciones. Como se explicó en el Capítulo 4, la idea principal es implementar en Java la librería de Colecciones OCL. Esto permite un mapeo casi uno a uno entre ambas especificaciones. En la Figura 9.14 se muestra el modelo de clases de la implementación.

Las clases de colecciones implementan la mayoría de las operaciones de colecciones de OCL, quedando sin implementar ciertas operaciones que requieren una expresión como parámetro, entre

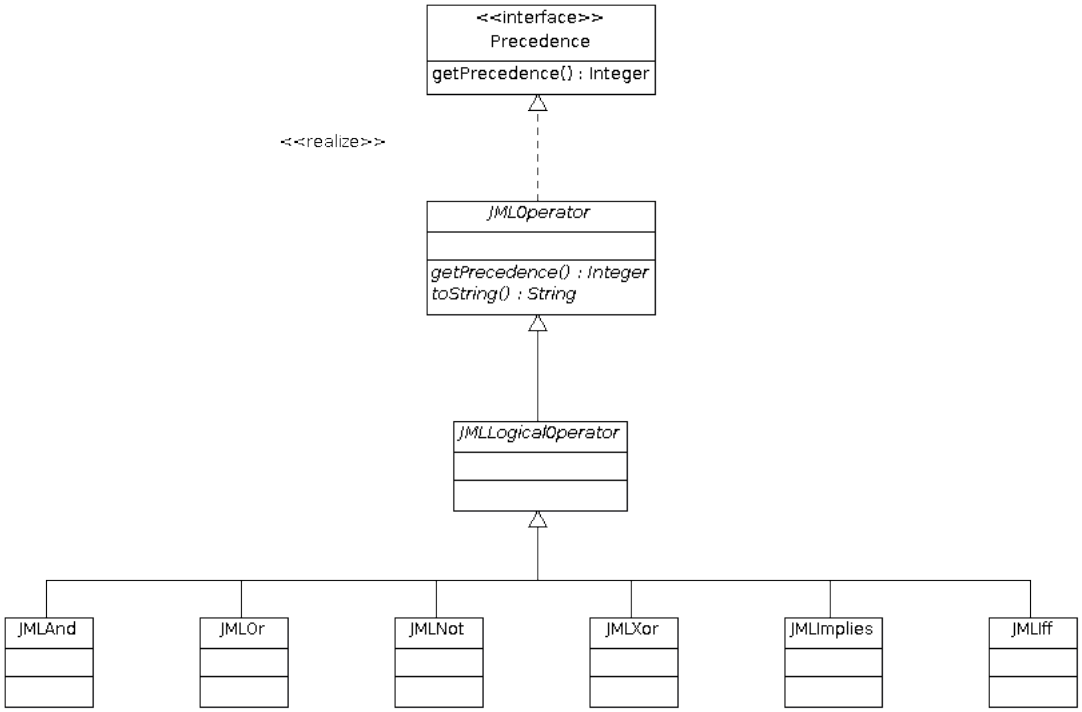

Figura 9.6: Metamodelo JML operadores lógicos

Figura 9.7: Metamodelo JML operadores relacionales

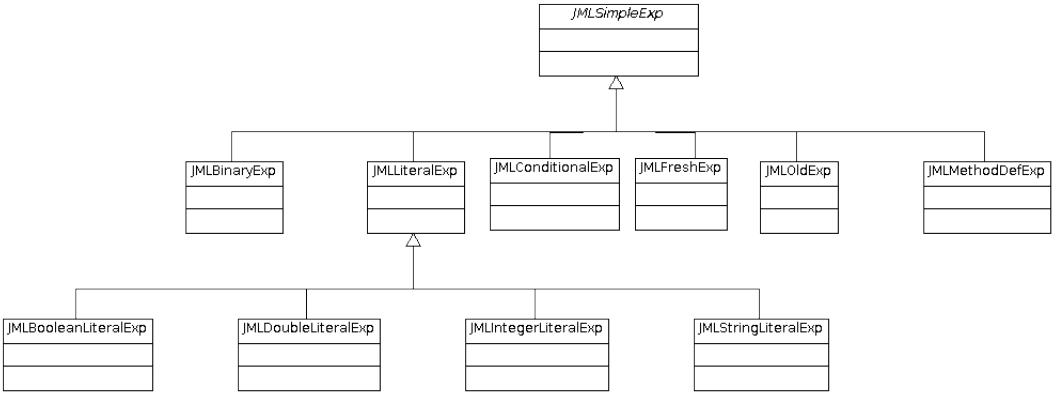

Figura 9.8: Metamodelo JML para operaciones simples. Parte 1

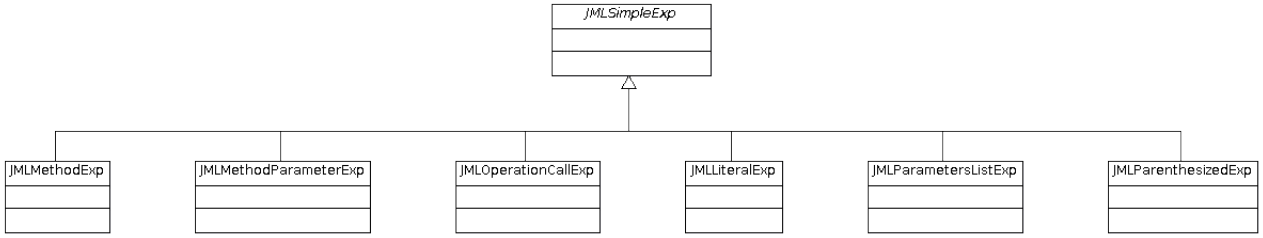

Figura 9.9: Metamodelo JML para operaciones simples. Parte 2

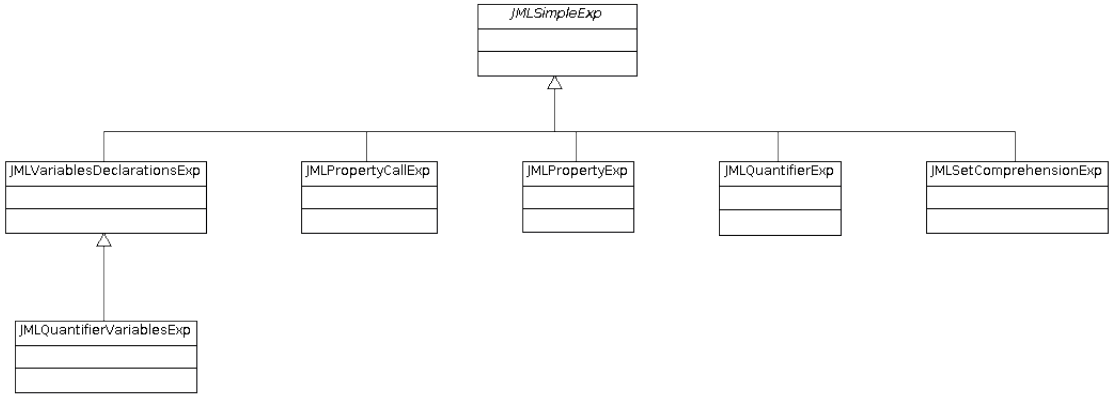

Figura 9.10: Metamodelo JML para operaciones simples. Parte 3

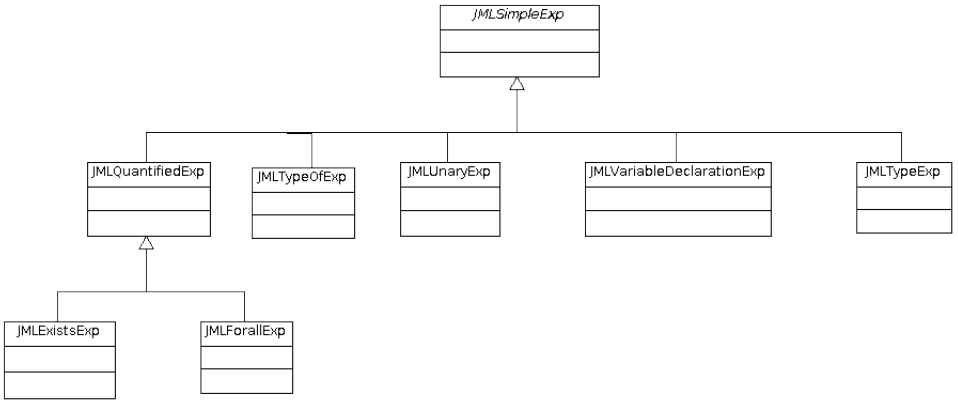

Figura 9.11: Metamodelo JML para operaciones simples. Parte 4

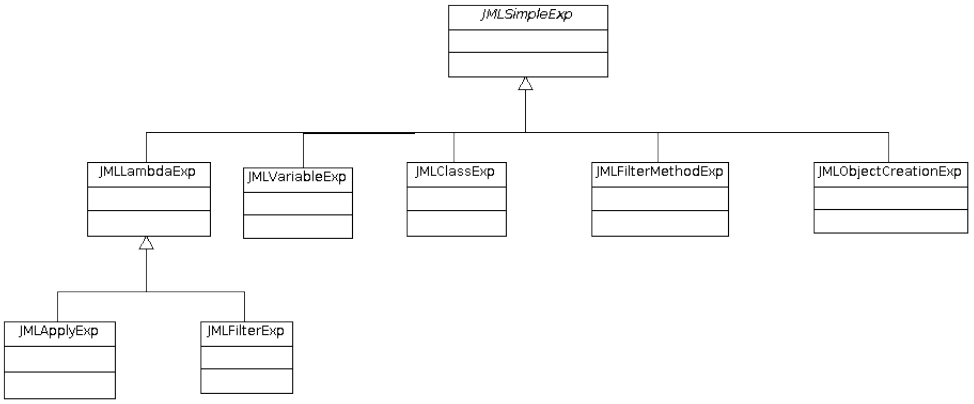

Figura 9.12: Metamodelo JML para operaciones simple. Parte 5

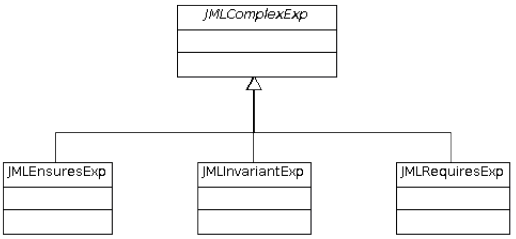

Figura 9.13: Metamodelo JML para operaciones complejas

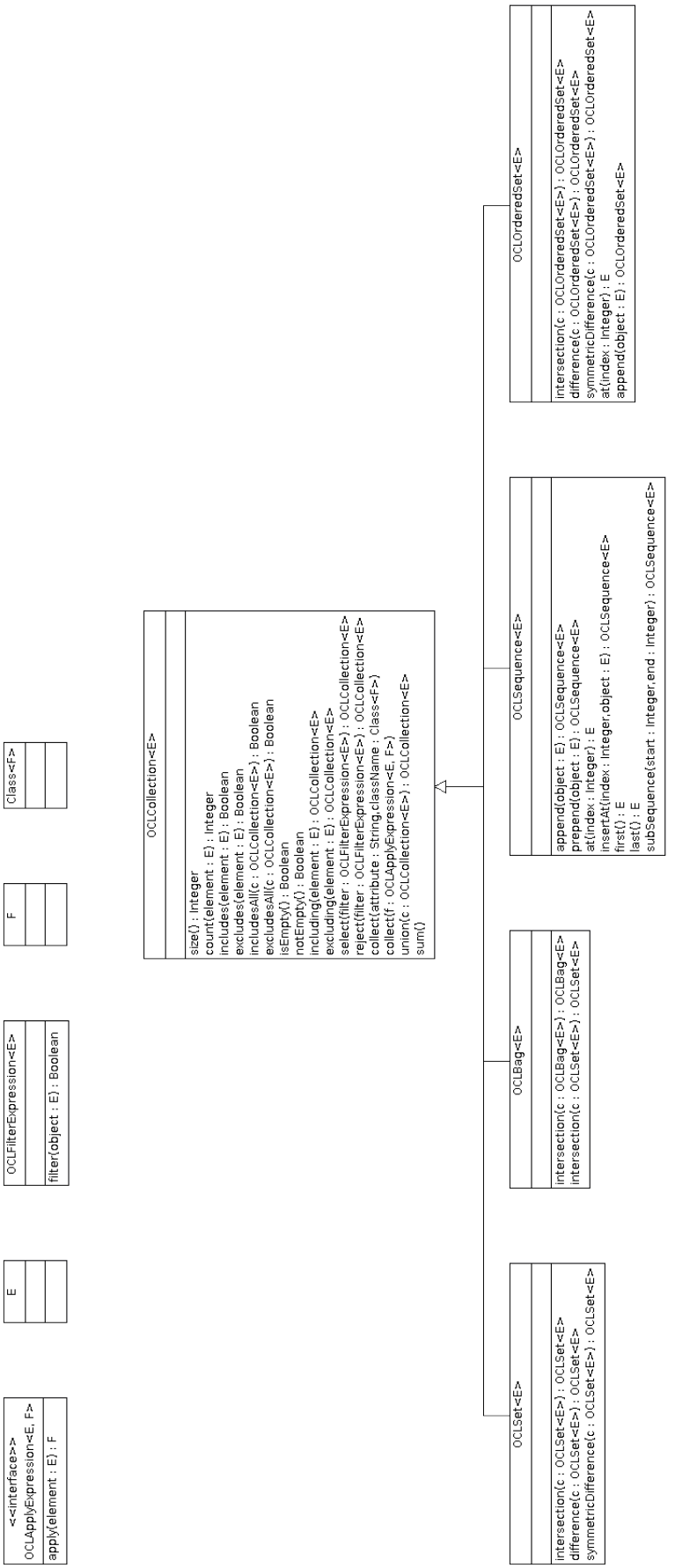

Figura 9.14: Implementación de la librería de colecciones OCL

ellas: *iterate*, *sortedBy*, *isUnique*, *collectNested*. No hay relación de generalización entre la librería de colecciones de Java y la librería de colecciones OCL, por dos razones principales: la librería de colecciones Java define operaciones adicionales y las colecciones Java no son inmutables, es decir, se pueden agregar o eliminar elementos. La librería de colecciones OCL desarrollada, sin embargo, utiliza las colecciones de la librería Java como implementación subyacente. Para la implementación de la clase `OCLBag` se utilizó la colección `MultiSet` provista por la librería de colecciones Guava (ex Google Collections), ya que brinda una implementación de la interface `Collection` de la librería de Java [6]. En la Tabla 9.1 se muestra las colecciones Java utilizadas por las colecciones OCL.

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

Tabla 9.1: Colecciones Java subyacentes

Las asociaciones definidas en el modelo se mapean a las clases de la librería teniendo en cuenta las reglas de navegación de OCL y los atributos *isOrdered* e *isUnique*, de acuerdo a la Tabla 9.2

	ordered	not ordered
unique	`OCLOrderedSet`	`OCLSet`
not unique	`OCLSequence`	`OCLBag`

Tabla 9.2: Mapeo de navegaciones y asociaciones OCL a Java

9.8. Resumen

En este capítulo vimos la descripción de las etapas de derivación de la herramienta desarrollada. La funcionalidad incluye un parser OCL, una traducción de un modelo UML, la creación de la especificación JML y la creación de un modelo Java.

La derivación que se realiza es: se toma un modelo UML con restricciones OCL, y se devuelve el UML verificado y una lista de restricciones (Etapa 1, Análisis del modelo), luego se toma el resultado de la lista de restricciones y se transforman a su equivalente JML (Etapa 2, Traducción de las restricciones OCL) y por último se toma el modelo UML con las anotaciones JML y utilizando MOFScript, se generan los archivos *.java* (Etapa 3, Transformación del modelo a texto).

Capítulo 10

Caso de estudio

En este capítulo veremos la funcionalidad implementada para la traducción de OCL a JML. Para ello se mostraremos un caso de estudio y describiremos paso a paso el proceso para dicha traducción. Para realizar el caso suponemos que tenemos todo el ambiente preparado, es decir, el plugin integrado con el eclipse, el modelo *.uml* y el archivo *.ocl* con las restricciones. Para mas información puede mirarse el Apéndice D.

10.1. Modelo Royal and Loyal

El diagrama de clases mostrado en la Figura 10.1, muestra las clases que usaremos para el caso de estudio.

10.2. Archivo OCL

Para disminuir la ambigüedad del modelo *.uml* se utilizará un archivo *.ocl* con un conjunto de completo de restricciones. El archivo completo puede verse en el Apéndice E. En la figura 10.2 se muestra como se ve nuestro archivo, en el eclipse.

10.3. Ejecución del Plugin

A continuación se mostrarán las distintas etapas de transformaciones de nuestro plugin.

10.3.1. Ejecutar el Parse OCL

Como se dijo en el Capítulo 9, el parse OCL realiza un análisis sintáctico y semántico del archivo con restricciones OCL. Para ejecutar el **Parse OCL**, debemos sobre nuestro archivo *.ocl* hacer Click derecho -> OCL Translator -> **Parse OCL**. En la Figura 10.3 se muestra una imagen de la opción descripta anteriormente.

Si el parseo del archivo tuvo éxito, se nos presentará un diálogo de información avisando que todo fue correcto. Un ejemplo se muestra en la Figura 10.4.

En caso que el parseo falle, se nos presentará un diálogo de información avisando que hubo un error, y cuál fue el primer error encontrado, en este caso el problema es que no reconoce la variable *ages*. Un ejemplo se muestra en la Figura 10.5.

Además de que hubo un error e informarlo en en diálogo, también lo hace en la pestaña de problemas, nos dirá cual fue el primer problema encontrado. En la Figura 10.6 se muestra el ejemplo.

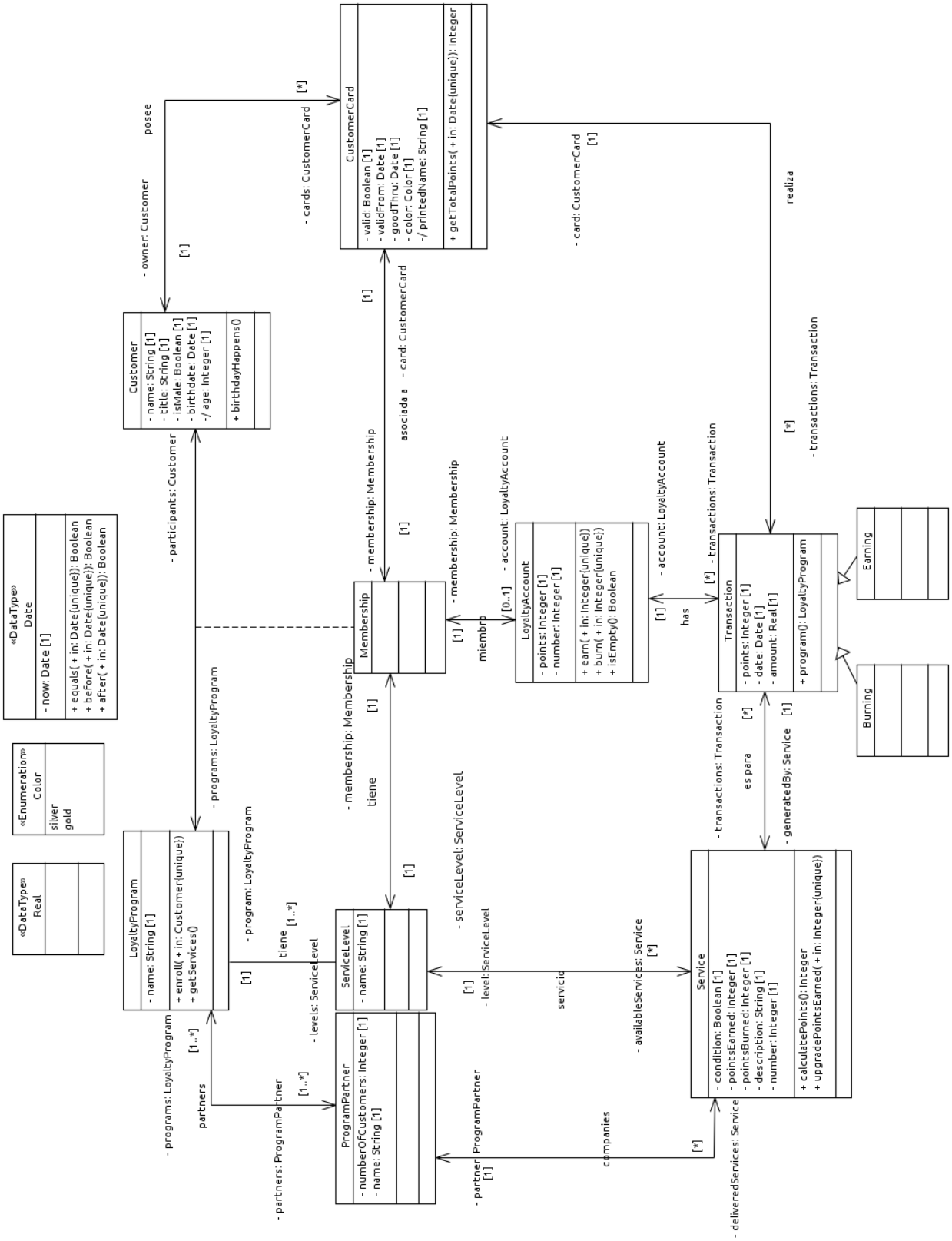

Figura 10.1: Modelo de clases Royal and Loyal

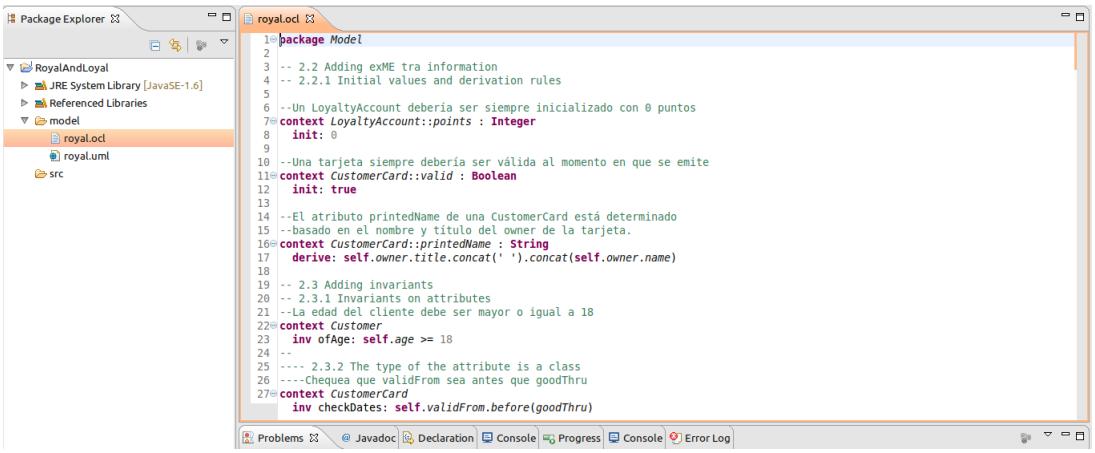

Figura 10.2: Restricciones OCL

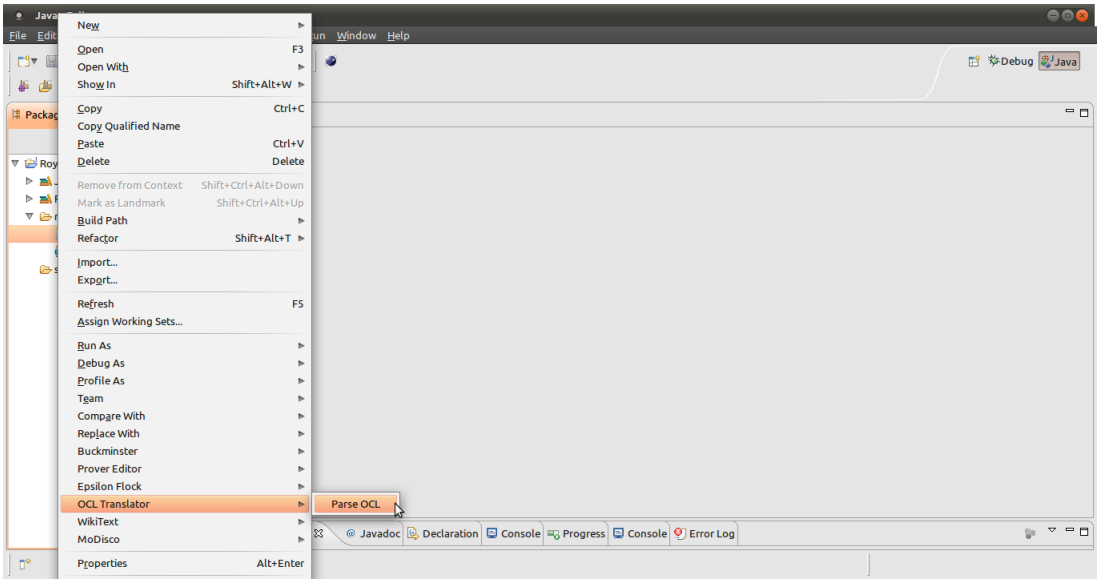

Figura 10.3: OCL Parse

Figura 10.4: Succesful Parsing

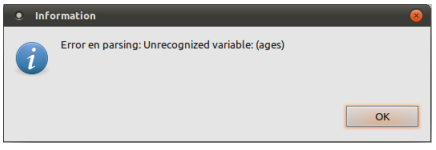

Figura 10.5: Error Parsing

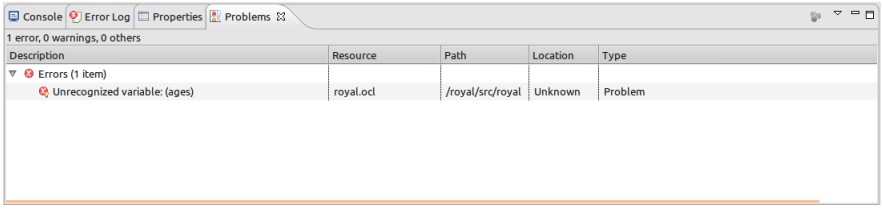

Figura 10.6: Problemas en consola

Para parsear el archivo *.ocl*, se hace en base al modelo *.uml*, como no encuentra la variable *ages* en el modelo, nos avisa del error. Si vemos el modelo de nuestro caso de estudio, podemos ver que la clase `Customer` tiene como variable *age* y no *ages*. Lo que debemos hacer es modificar la variable en nuestro archivo *.ocl* por la correcta y ejecutar nuevamente el parseo del archivo.

10.3.2. Ejecutar las distintas transformaciones

Translate UML model

Como se dijo en el Capítulo 9, el **Translate UML model**, toma un modelo UML y genera las clases java anotadas con las especificaciones JML. Para ejecutar el **Translate UML model**, debemos sobre nuestro archivo *.uml* hacer Click derecho -> OCL Translator -> Translate UML model. En la Figura 10.7 se muestra una imagen de la opción descripta anteriormente.

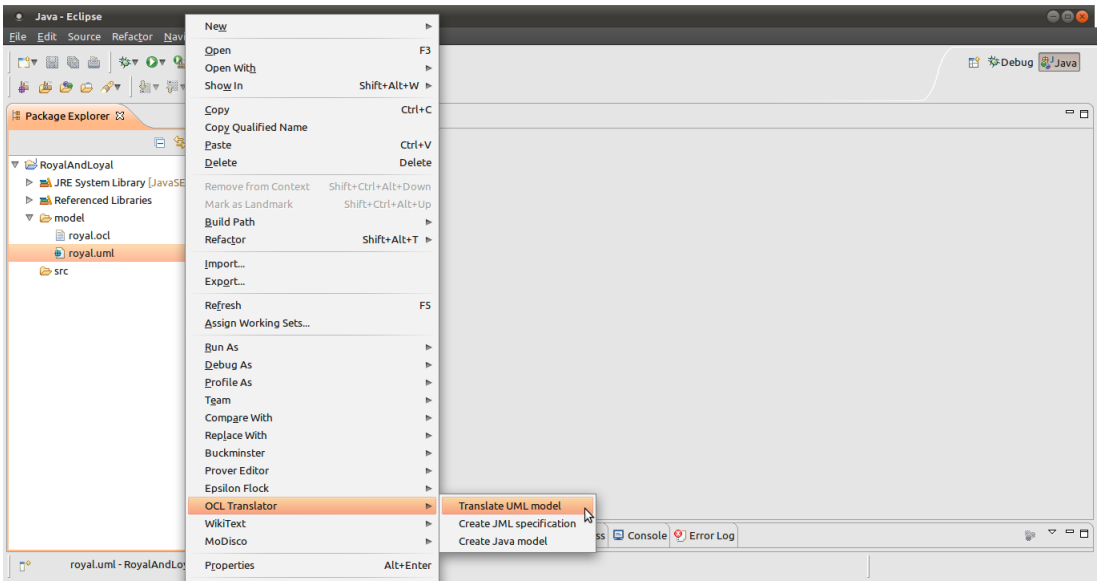

Figura 10.7: Translate UML model

Si dicha transformación tuvo éxito, se nos presentará un diálogo de información avisando que todo fue correcto. El diálogo de información es similar al presentado en la Figura 10.4.

Un ejemplo del código generado se muestra en el siguiente código, para la clase `LoyaltyProgram`:

```
package model;

import java.util.*;
import ar.edu.unlp.info.ocl2jml.ocl.collections.*;

public class LoyaltyProgram  {
```

```
 //Invariantes de clase
  /*@ public invariant
   @    this.partners.
   @        collect("getDeliveredservices", Service.class).
   @             size() >= 1;
   @*/
 //@ public invariant this.participants.size() < 10000;
 /*@ public invariant
   @        (\forall Customer temp1; this.participants.includes(temp1);
   @              temp1.getAge() <= 70);
   @*/
 /*@ public invariant
   @           (\forall Customer c1,  c2; this.participants.includes(c1) &&
   @                this.participants.includes(c2); ! c1.equals(c2) ==>
   @                     ! c1.getName().equals(c2.getName()));
   @*/

 //Atributos de la clase
 private /*@ spec_public non_null */  String name;
 private /*@ spec_public non_null */  OCLOrderedSet<ServiceLevel> levels;
 private /*@ spec_public non_null */  OCLSet<Customer> participants;
 private /*@ spec_public non_null */  OCLSet<ProgramPartner> partners;
 public /*@ spec_public non_null */  OCLSequence<Membership> membership;

 //Constructor de la clase
 public LoyaltyProgram(String name){
   this.name = name;
   this.levels = new OCLOrderedSet<ServiceLevel>();
   this.participants = new OCLSet<Customer>();
   this.partners = new OCLSet<ProgramPartner>();
   this.membership = new OCLSequence<Membership>();
 }

 //Setters y getters de atributos
 public  String getName() {
   return name;
 }

public void setName(String name) {
   this.name = name;
 }

 public  OCLOrderedSet<ServiceLevel>getLevels () {
   return levels;
 }

public void setLevels( OCLOrderedSet<ServiceLevel> levels) {
   this.levels = levels;
 }

 public  OCLSet<Customer>getParticipants () {
   return participants;
 }

public void setParticipants( OCLSet<Customer> participants) {
```

```
    this.participants = participants;
  }

  public  OCLSet<ProgramPartner>getPartners () {
    return partners;
  }

 public void setPartners( OCLSet<ProgramPartner> partners) {
    this.partners = partners;
  }

  public OCLSequence<Membership> getMembership () {
    return membership;
  }

  public void setMembership(OCLSequence< Membership> membership) {
    this.membership = membership;
  }

  //Operaciones de la clase

  //@ requires ! c.getName().equals("");
  //@ requires ! this.participants.includes(c);
  //@ ensures this.participants.equals(\old(this.participants).including(c));
  public void enroll(Customer c){

  }

  public void getServices(){

  }
}
```

Como puede verse, el código generado, genera tanto la clase java con su constructor, setters y getter, y operaciones, como las anotaciones JML.

En caso que la transformación falle, se nos presentará un diálogo de información avisando que hubo un error, y cuál fue el error encontrado. Pero en este caso, no se mostrará el error en la pestaña de problemas.

Create JML specification

Como se dijo en el Capítulo 9, el **Create JML specification**, toma un modelo UML y genera las especificaciones JML. Para ejecutar el **Create JML specification**, debemos sobre nuestro archivo *.uml* hacer Click derecho -> OCL Translator -> **Create JML specification**. En la Figura 10.8 se muestra una imagen de la opción descripta anteriormente.

Si dicha transformación tuvo éxito, se nos presentará un diálogo de información avisando que todo fue correcto. El diálogo de información es similar al presentado en la Figura 10.4.

Un ejemplo del código generado se muestra en el siguiente código, para la clase `LoyaltyProgram`:

```
package model;

import java.util.*;
import ar.edu.unlp.info.ocl2jml.ocl.collections.*;
```

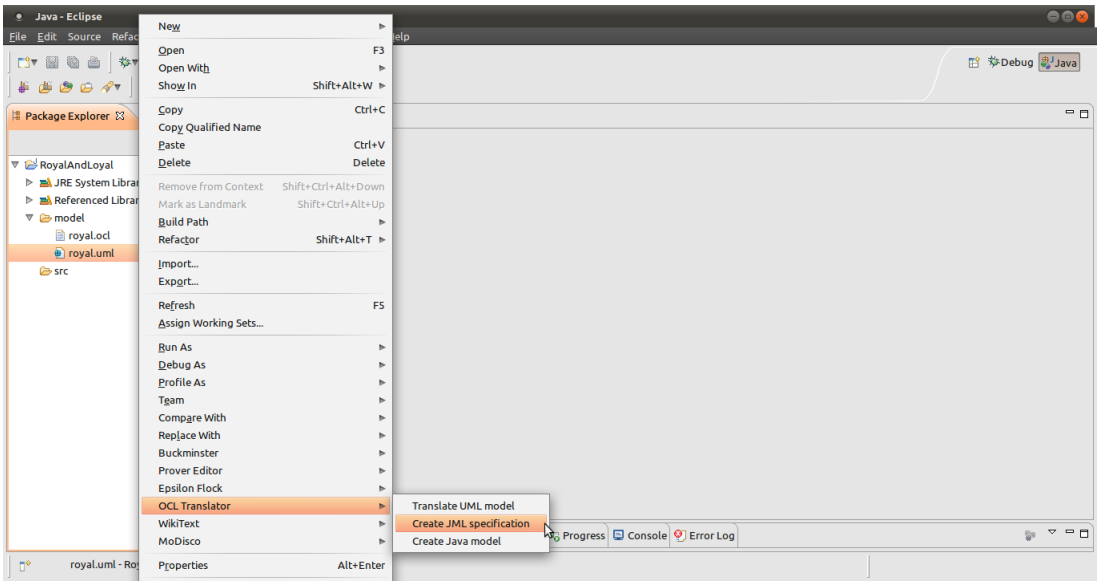

Figura 10.8: Create JML specification

```
public class LoyaltyProgram  {
  //Invariantes de clase
  /*@ public invariant
    @    this.partners.
    @        collect("getDeliveredservices", Service.class).
    @              size() >= 1;
    @*/
  //@ public invariant this.participants.size() < 10000;
  /*@ public invariant
    @        (\forall Customer temp1; this.participants.includes(temp1);
    @              temp1.getAge() <= 70);
    @*/
  /*@ public invariant
    @           (\forall Customer c1,  c2; this.participants.includes(c1) &&
    @                this.participants.includes(c2); ! c1.equals(c2) ==>
    @                     ! c1.getName().equals(c2.getName()));
    @*/

  //Atributos de la clase
  private /*@ spec_public non_null */  String name;

  private /*@ spec_public non_null */  OCLOrderedSet<ServiceLevel> levels;
  private /*@ spec_public non_null */  OCLSet<Customer> participants;
  private /*@ spec_public non_null */  OCLSet<ProgramPartner> partners;
  public /*@ spec_public non_null */  OCLSequence<Membership> membership;

  //Constructor de la clase
  public LoyaltyProgram(String name);

  //Operaciones de la clase
  //@ requires ! c.getName().equals("");
  //@ requires ! this.participants.includes(c);
  //@ ensures this.participants.equals(\old(this.participants).including(c));
```

```
  public void enroll(Customer c);
  public void getServices();
}
```

En este caso, lo que se genera son las especificaciones JML y sólo se generan los atributos de la clase y los encabezados de los métodos, con sus anotaciones.

En caso que la transformación falle, se nos presentará un diálogo de información avisando que hubo un error, y cuál fue el error encontrado. Pero en este caso, no se mostrará el error en la pestaña de problemas.

Create Java model

Como se dijo en el Capítulo 9, el **Create Java model**, toma un modelo UML y genera las clases Java sin las especificaciones JML. Para ejecutar el **Create Java model**, debemos sobre nuestro archivo *.uml* hacer Click derecho -> OCL Translator -> **Create Java model**. En la Figura 10.9 se muestra una imagen de la opción descripta anteriormente.

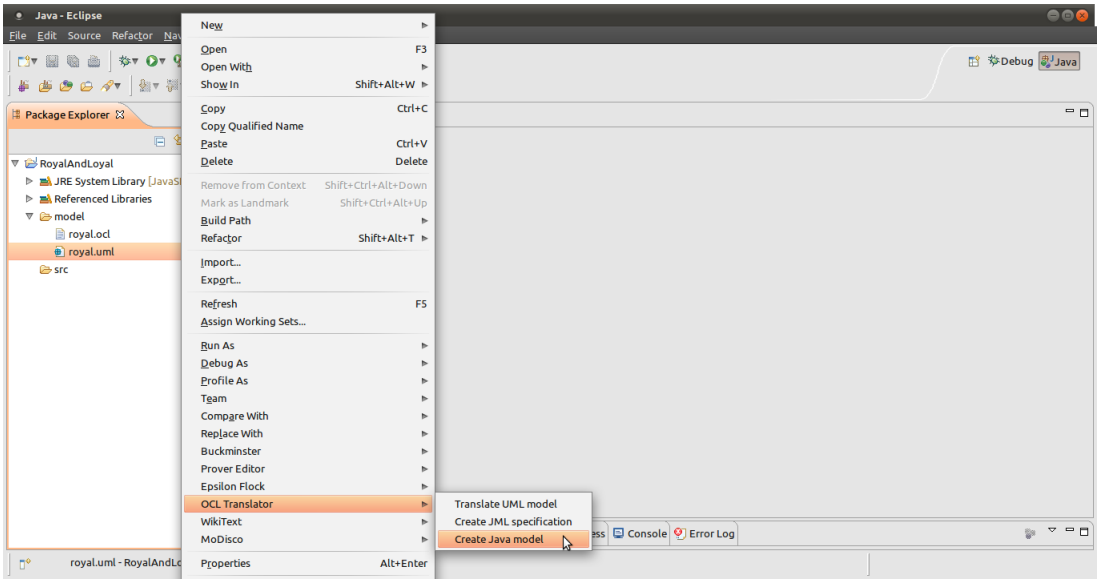

Figura 10.9: Create Java model

Si dicha transformación tuvo éxito, se nos presentará un diálogo de información avisando que todo fue correcto. El diálogo de información es similar al presentado en la Figura 10.4.

Un ejemplo del código generado se muestra en el siguiente código, para la clase `LoyaltyProgram`:

```
 package model;

import java.util.*;
import ar.edu.unlp.info.ocl2jml.ocl.collections.*;

public class LoyaltyProgram  {

  //Atributos de la clase
  private String name;
  private  OCLOrderedSet<ServiceLevel> levels;
  private  OCLSet<Customer> participants;
  private  OCLSet<ProgramPartner> partners;
```

```
public /*@ spec_public non_null */  OCLSequence<Membership> membership;

//Constructor de la clase
 public LoyaltyProgram(String name){
     this.name = name;
     this.levels = new OCLOrderedSet<ServiceLevel>();
     this.participants = new OCLSet<Customer>();
     this.partners = new OCLSet<ProgramPartner>();
     this.membership = new OCLSequence<Membership>();
}

//Setters y getters de atributos
 public  /*@ pure @*/ String getName() {
   return name;
}

public void setName(String name) {
   this.name = name;
}

public  OCLOrderedSet<ServiceLevel> getLevels () {
   return levels;
}

public void setLevels( OCLOrderedSet<ServiceLevel> levels ) {
   this.levels = levels;
}

public  OCLSet<Customer> getParticipants () {
   return participants;
}

public void setParticipants( OCLSet<Customer> participants ) {
   this.participants = participants;
}

public  OCLSet<ProgramPartner> getPartners () {
   return partners;
}

public void setPartners( OCLSet<ProgramPartner> partners ) {
   this.partners = partners;
}

public OCLSequence<Membership> getMembership () {
   return membership;
}

public void setMembership(OCLSequence< Membership> membership) {
   this.membership = membership;
}

//Operaciones de la clase
public void enroll(Customer c){

}
```

```
  public void getServices(){

  }
}
```

En este caso, lo que se genera es la clase Java, con sus atributos, setters y getters y operaciones, sin las especificaciones JML.

En caso que la transformación falle, se nos presentará un diálogo de información avisando que hubo un error, y cuál fue el error encontrado. Pero en este caso, no se mostrará el error en la pestaña de problemas.

10.4. Herramientas de verificación

10.4.1. Nuevo Ejemplo

Para mostrar el funcionamiento de las herramientas de verificación, se utilizará el ejemplo de Cuenta Bancaria extraído de [34], ya que estas herramientas no soportan tipos complejos, por lo que no podemos mostrarlas con las clases de nuestro ejemplo anterior. La Figura 10.10 muestra el ejemplo a utilizar.

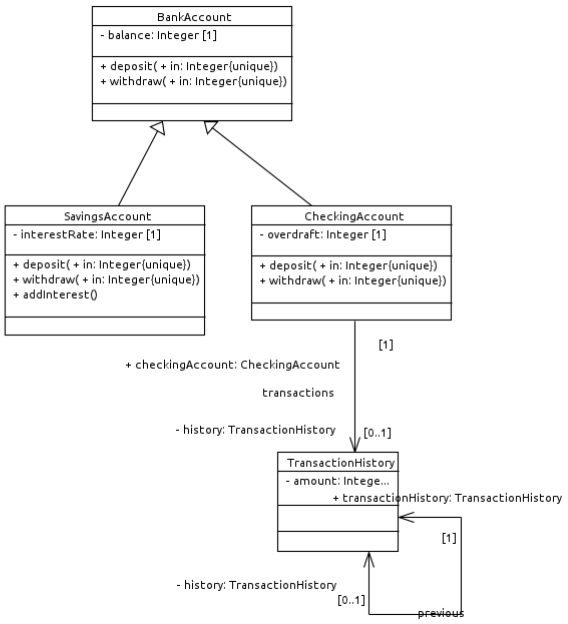

Figura 10.10: Ejemplo de cuenta bancaria

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 10.11

La clase sobre la que vamos a trabajar es BankAccount, mostrada en la Figura 10.12.

Y su especificación JML es la mostrada en la Figura 10.13

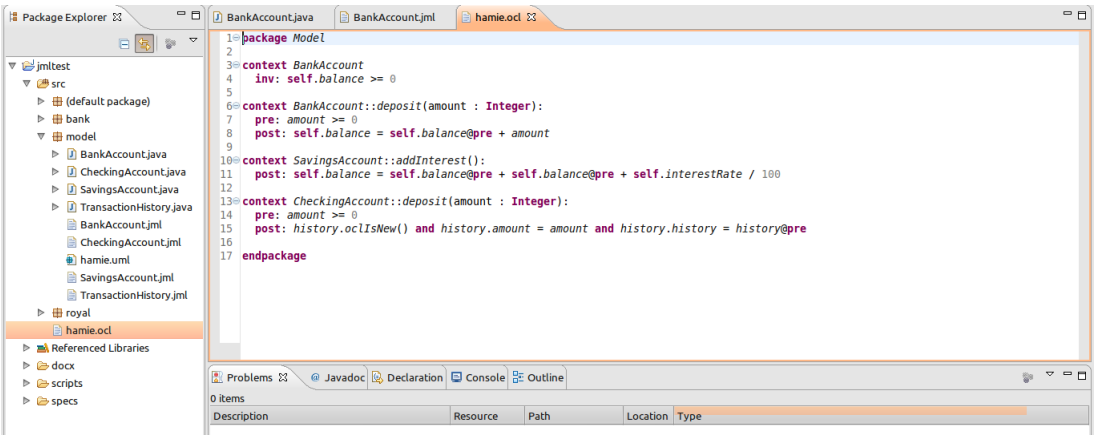

Figura 10.11: OCL utilizado

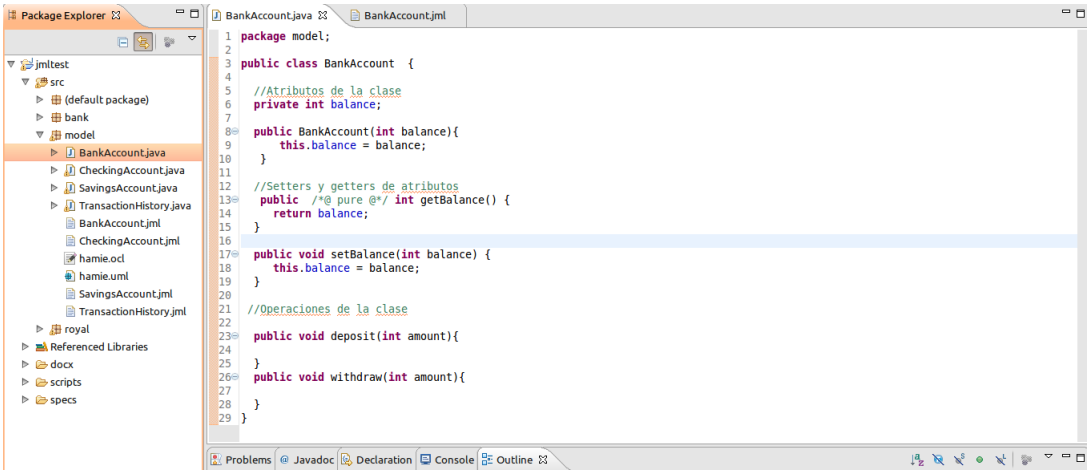

Figura 10.12: Clase Bank Account Generada

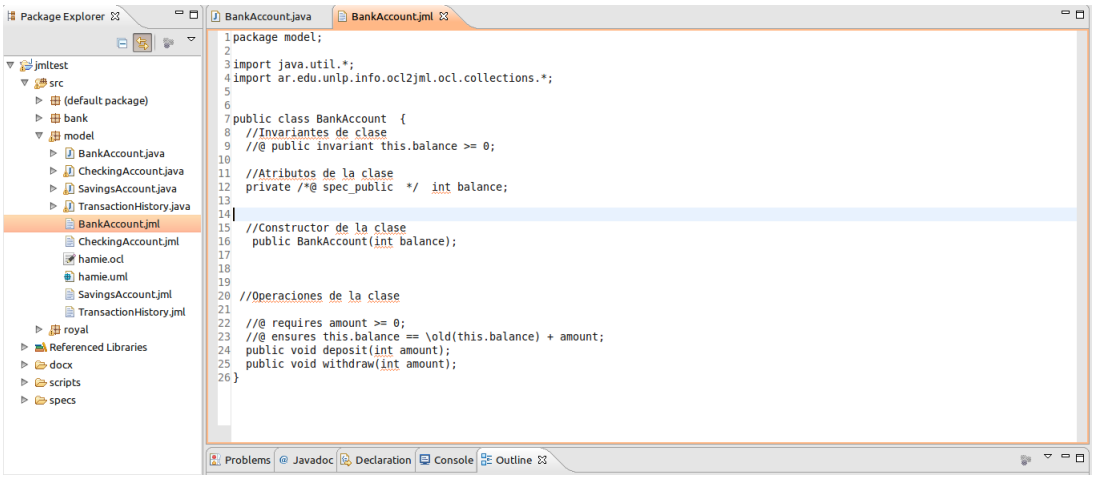

Figura 10.13: Especificación JML de la clase Bank Account Generada

10.4.2. 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:

```
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).

  Selección   Ruta                                          Prioridad  Estado
------------------------------------------------------------
* 0            /usr/lib/jvm/java-6-openjdk/jre/bin/java   1061      modo automático
  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 10.14.

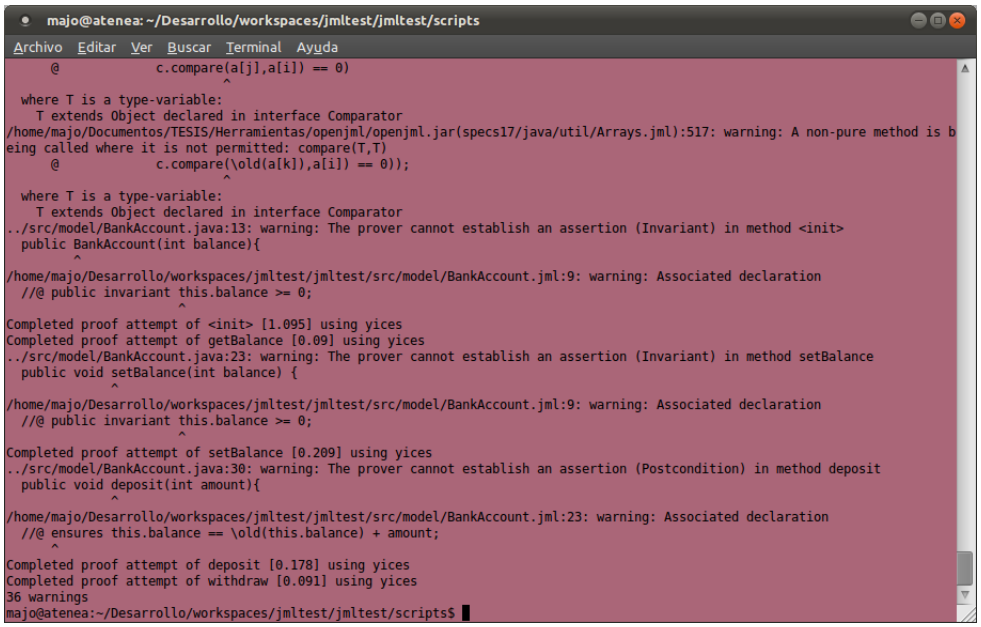

Figura 10.14: Primer salida de la consola

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.

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 10.15

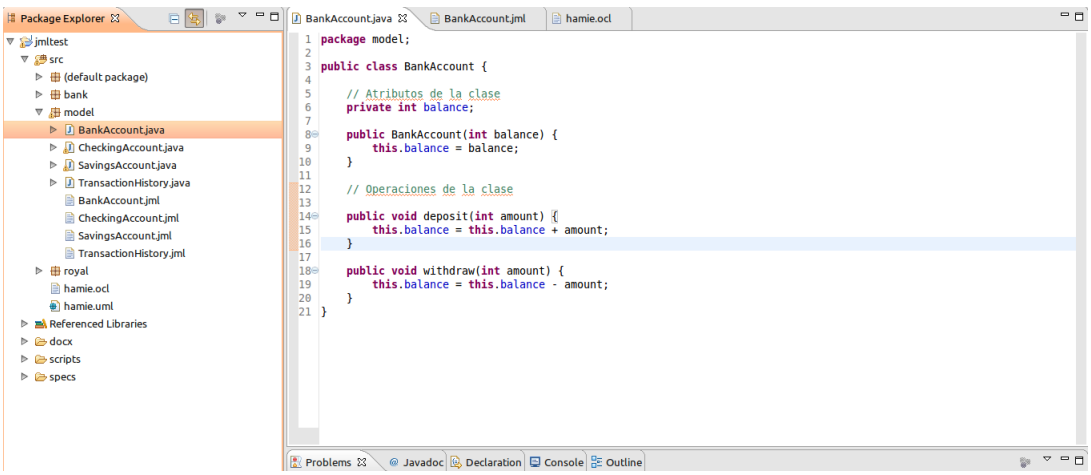

Figura 10.15: Clase Bank Account modificada

Volvemos a ejecutar el script de OpenJML y como resultado obtenemos lo mostrado en la Figura 10.16, 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 10.17.

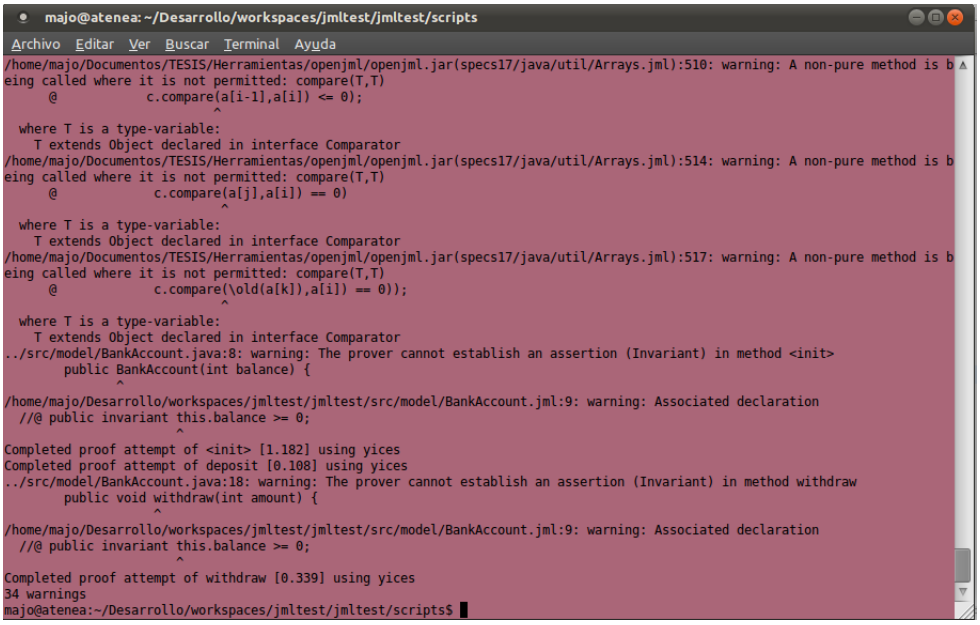

Figura 10.16: Segunda salida de consola

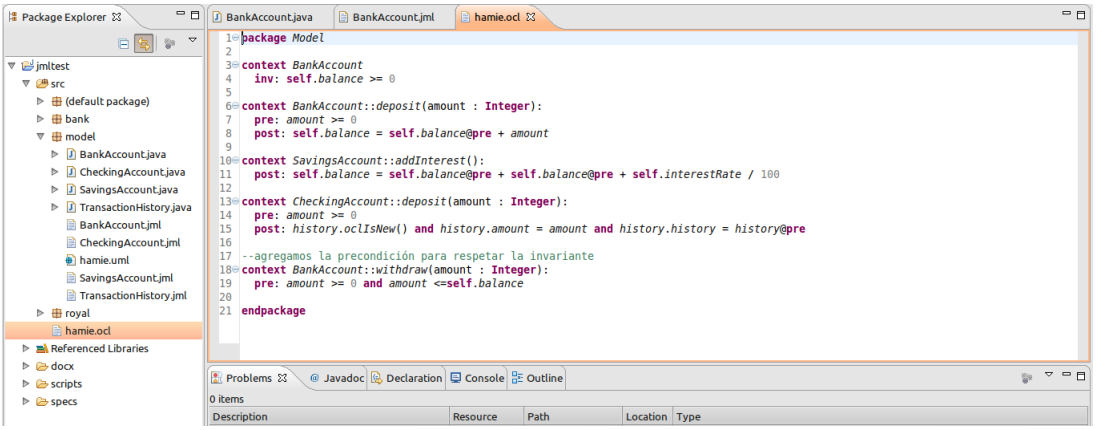

Figura 10.17: OCL modificado

Nuestro siguiente paso será generar nuevamente las especificaciones *.jml* para luego correr el script. Las especificaciones regeneradas se muestran en la Figura 10.18.

Volvemos a ejecutar el script de **OpenJML** y como resultado obtenemos lo mostrado en la Figura 10.19. **OpenJML** nos informa que probó satisfactoriamente los métodos *deposit* y *withdraw*.

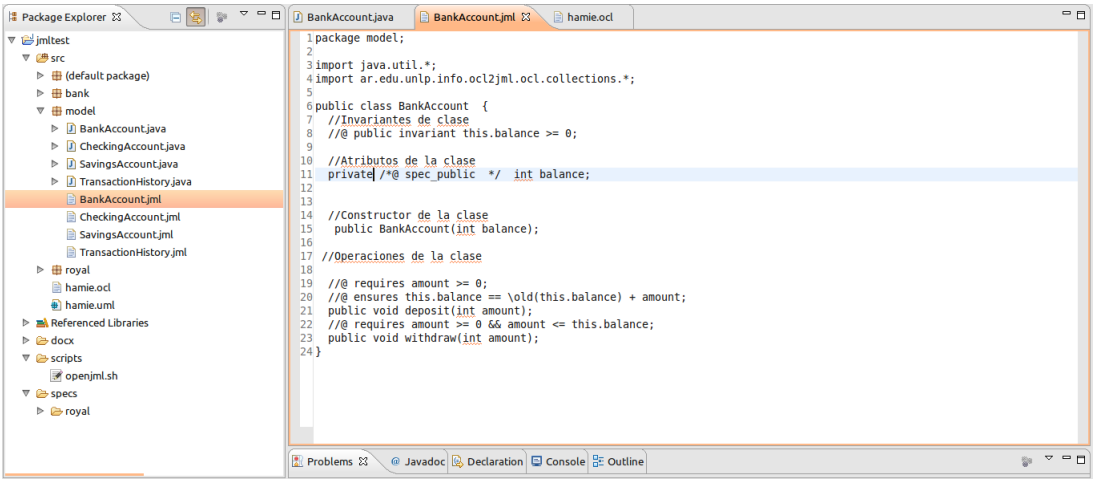

Figura 10.18: JML regenerado

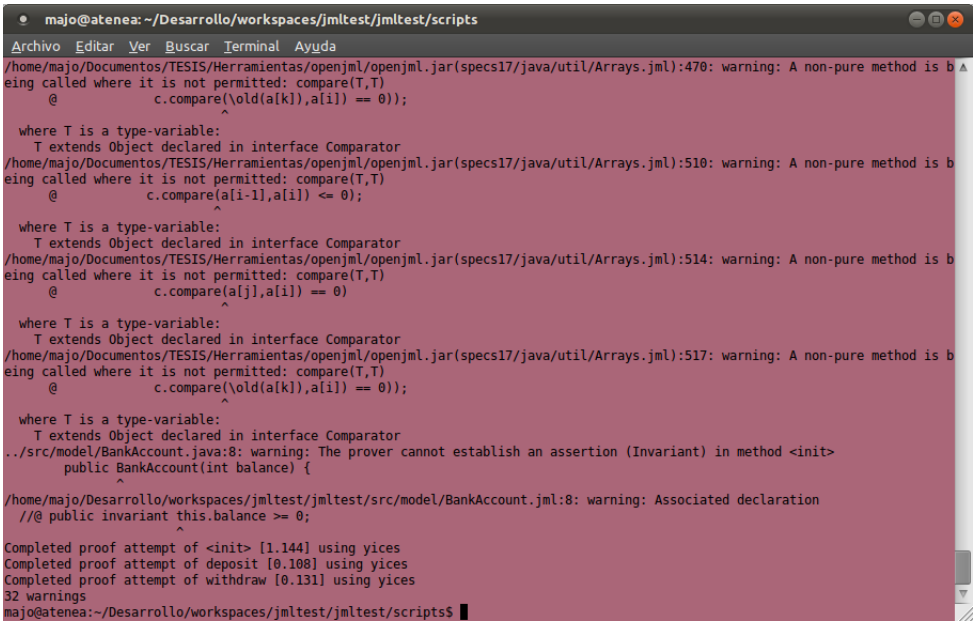

Figura 10.19: Tercera salida de consola

Capítulo 11

Trabajos relacionados

En este capítulo se presentan distintos trabajos relacionados al tema de nuestra tesis.

- En [35] 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 [36] 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 detaller 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 [19] 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 de la librería `OCL-like` a JML permite mapear las restricciones OCL a aserciones JML en uno a uno 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 abstraciones usando las clases de la librería.

- En [31] se presenta una implementación de un compilador de OCL a JML. Se explicaron los mapeos que se utilizaron 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, pero que luego de una investigación, se logró hacerlo. También se dedicó toda una sección a los mapeos de colecciones explicando que fue muy trabajoso realizar esa traducción y en otros casos imposible. También se dedicó una sección a compilador JML, diciendo que el código debía haber sido correctamente generado.

- En [34] 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 paper se propone una formalización de la relación "realizes" con respecto a la implementación escrita en Java. Esta formalizaciónse 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 cuál 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 Specification and Verification System (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 ser 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 [21] y [20] 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.

11.1. Aportes

El desarrollo de esta tesina, aporta a los trabajos relacionados mencionados en el capítulo anterior, 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 [19]. La implementación de la herramienta como plugin de eclipse permitió utilizar otros plugins como el OCL Plugin y UML Plugin.

Capítulo 12

Conclusiones y trabajos futuros

12.1. Conclusiones

Uno de los principales problemas actuales de la ingenería de software consiste en garantizar la corrección del software, es decir, que el software cumpla con la especificación (verificación) y que el software satisfaga las expectativas del usuario (validación). En este trabajo se abordó el primer punto.

Los lenguajes de especificación formales permiten especificar restricciones adicionales, no sólo para modelos gráficos, sino también para código. Podemos mencionar principales ventajas de los lenguajes de modelado gráfico, la facilidad de comprensión de los fenómenos que representan y su facilidad para ser entendidos por los desarrolladores, los que no requieren un background formal.

En esta tesina 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.

Como vimos en el caso de estudio, la existencia de herramientas de verificación de programas disponibles para JML posibilita 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.

12.2. 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 Java 8 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 podría 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 luego de generar el código. Podrían ejecutarse las herramientas de verificación desde la herramienta desarrollada, integrando las herramientas al plugin.

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. Aunque si puede tenerse en cuenta para una extensión de la herramienta.

Apéndice A

Sintaxis abstracta de OCL

La sintaxis abstracta representa los conceptos de OCL empleando MOF. Para representar el metamodelo de OCL se importan metaclases del metamodelo UML, las cuales son mostradas en el modelo con la anotación "(from <UML Package>)". [13, 18]

La sintaxis abstracta está dividida de la siguiente manera:

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

A.1. El paquete type

OCL es un lenguaje tipado. Cada expresión tiene un tipo que se declara explícitamente o derivado estáticamente. Antes de definir expresiones es necesario proveer un modelo para el concepto de tipo. En la Figura A.1 se ilustra el paquete types.

El modelo de la figura muestra los tipos OCL. El tipo básico es el UML `Classifier`, que incluye todos los subtipos de `Classifier` desde UML Superstructure.

1. `AnyType`: es la metaclase del tipo especial `OclAny`, el cual es el tipo al que todos los demás tipos se ajustan. `OclAny` es la única instancia de `AnyType`. Esta metaclase permite definir la propiedad especial de ser la generalización de todos los demás Classifiers, incluyendo `Classes`, `DataTypes` y `PrimitiveTypes`.

2. `BagType`: es una tipo de colección que puede tener elementos duplicados. Los elementos no tienen orden. Parte de un *BagType* es la declaración del tipo de sus elementos.

3. `CollectionType`: describe una lista de elementos de un tipo particular. `CollectionType` es una metaclase concreta de la cual sus instancias son de la familia de los tipos de datos abstractos `Collection(T)`. Las subclases son: `SetType`, `OrderedSetType`, `SequenceType` y `BagType`, cuyas instancias son los tipos concretos `Set(T)`, `OrderedSet(T)`, `Sequence(T)` y `Bag(T)`. No hay ninguna restricción sobre los tipos de una colección, por ejemplo podemos tener una colección cuyo tipo es otra colección.
   Asociaciones:

   - **elementType**: es el tipo de los elementos de la colección. Todos los elementos que forman parte de la colección deben cumplir con este tipo.

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

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

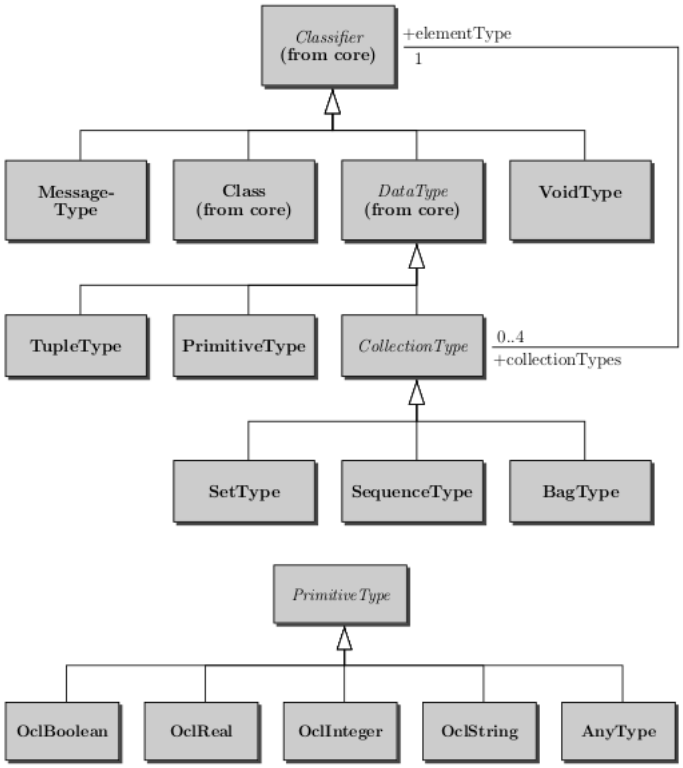

Figura A.1: Extracto del metamodelo de la sintaxis abstracta para los tipos de OCL

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

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

8. `InvalidType` representa un tipo que cumple con todos los tipos excepto el tipo `VoidType`. La única instancia de InvalidType es Invalid, la cual está definida en la librería estándar.

9. `MessageType` describe mensajes ocl. Es similar a tipo colección, `MessageType` describe un conjunto de tipos en la librería estándar. Parte de todos los `MessageType` son una referencia a la declaración del tipo.

10. `TemplateParameterType` es usado para referenciar tipos genéricos en las definiciones parametrizadas. Esto es usado en la librería estándar para representar las operaciones de colecciones parametrizadas.

11. `VoidType` es la metaclase del tipo `OclVoid` que cumple con todos los tipos excepto el `OclInvalid`. La única instancia de `VoidType` es `OclVoid`, el cual está definido en la librería estándar.

A.2. Paquete Expressions

En esta sección se define la sintaxis abstracta del paquete de las expresiones. Este paquete define la estructura que pueden tener las expresiones OCL. En la Figura A.2 se ilustra la estructura básica del paquete expressions.

1. `OclExpression` es una expresión que puede ser evaluada en un ambiente dado. Esta metaclase es la superclase de todas las expresiones en el metamodelo. Toda expresión OCL tiene un tipo que se puede determinar estáticamente analizando la expresión y su contexto. La evaluación

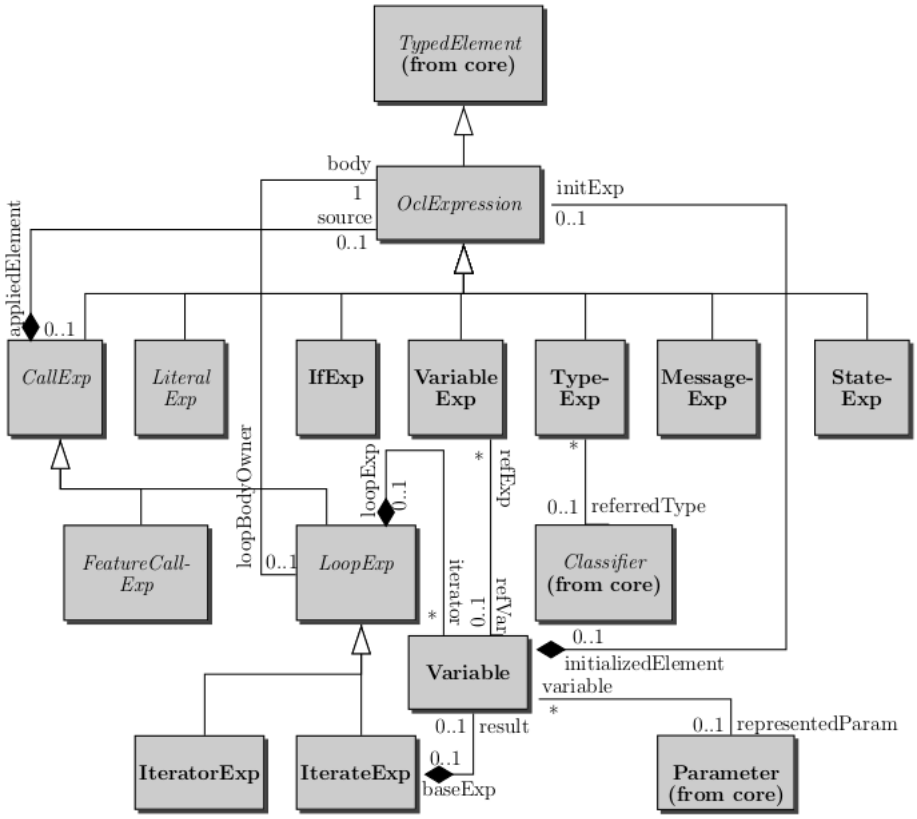

Figura A.2: Estructura básica del metamodelo de la sintaxis abstracta para expresiones

de la expresión retorna un valor. Las expresiones cuyo tipo es un boolean se pueden utilizar en las restricciones; en caso contrario, para operaciones query, valores iniciales de atributo, etc.

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

2. `Variable`. Son elementos tipados para pasar datos en las expresiones. Esta metaclase representa entre otras las variables **self** y **result**.
   Asociaciones:
   - **initExpression**: expresión OCL que representa el valor inicial de la variable.
   - **representedParameter**: parámetro de la operación actual que representa la variable. Cualquier acceso a esta representa un acceso al valor del parámetro.
3. `VariableExp` es una expresión que consiste en una referencia a una variable.
   Asociaciones:
   - referredVariable: variable a la que ésta expresión se refiere.
4. `LiteralExp` es una expresión sin argumentos que representa un valor. Algunas de sus subclases son: `IntegerLiteralExp`, `BooleanLiteralExp`, etc.
5. `PropertyCallExp` es una expresión que se refiere a un Feature (operación o propiedad) o a un iterador predefinido para colecciones. El resultado se obtiene a partir de la evaluación del

feature correspondiente. Es una metaclase abstracta.
Asociaciones:

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

6. `FeatureCallExp` es una expresión que se refiere a un Feature definido en un `Classifier` del modelo. El resultado es la evaluación de la propiedad correspondiente. La Figura A.3 muestra las subclases de `FeatureCallExp`.

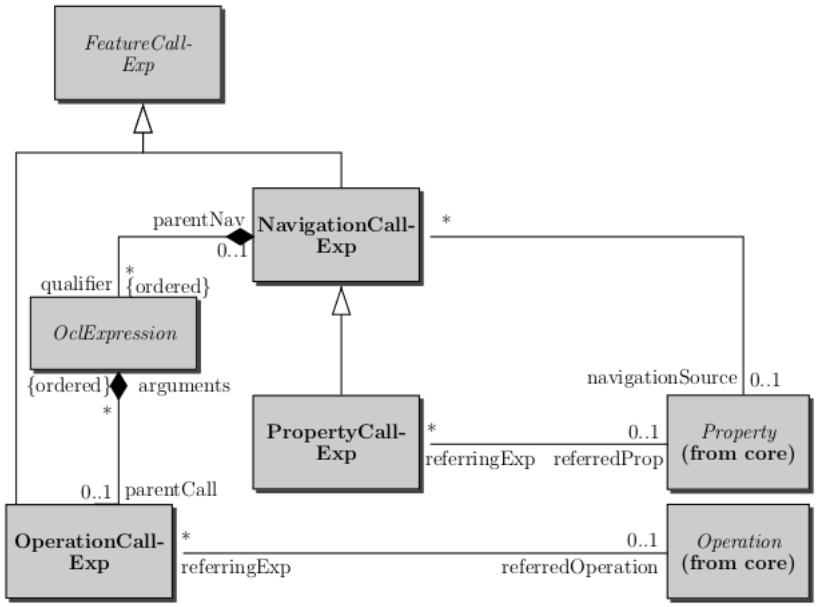

Figura A.3: Metamodelo de la sintaxis abstracta para FeatureCallExp

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

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

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

- **result**: representa la variable resultante.

9. `IteratorExp` es una expresión que evalúa la expresión indicada por la asociación body para cada elemento de la colección receptora. El resultado de la evaluación es un valor cuyo tipo depende del nombre de la expresión. En algunos casos puede ser del mismo tipo que el receptor. La metaclase `IteratorExp` representa todas las operaciones predefinidas de las colecciones que se definen a través de la expresión iterator, por ejemplo **select**, **exists**, **collect**, **forAll**, etc.

10. `IfExp` la expresión if está incompleta en el diagrama por cuestiones de legibilidad. Ésta se encuentra más detallada en la siguiente figura.

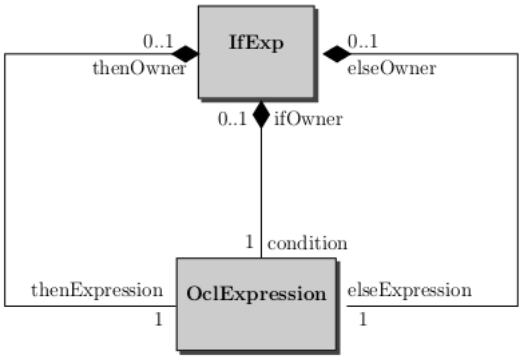

Figura A.4: Metamodelo para la expresión if

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

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

A.2.1. Metaclase ExpressionInOcl

Debido a que la metaclase `OclExpression` está definida recursivamente necesitamos una metaclase que represente el nivel superior del árbol de la sintaxis abstracta. Esta metaclase es llamada `ExpressionInOcl`, y está definida como una subclase de la metaclase `OpaqueExpression` del paquete core de UML. En la Figura A.5 se representa la metaclase `ExpressionInOcl`.

Asociaciones:

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

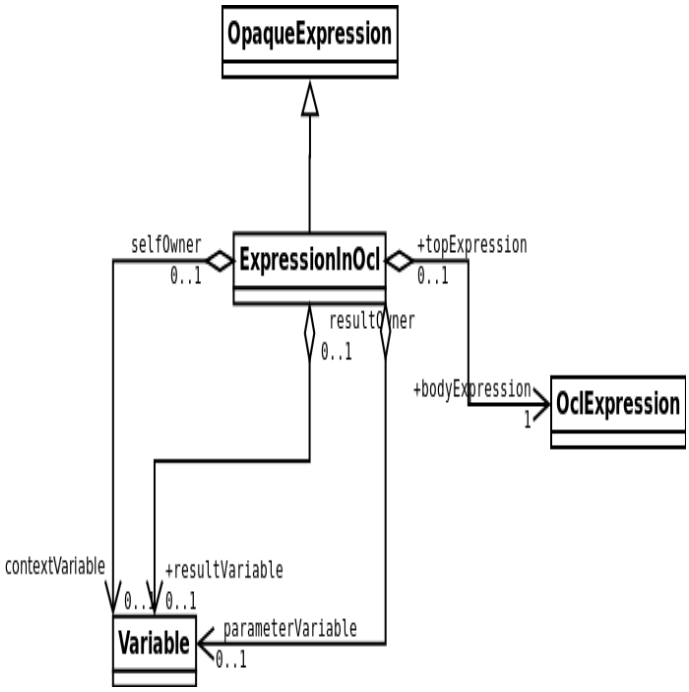

Figura A.5: Metamodelo de la sintaxis abstracta para ExpressionInOcl

Apéndice B

Librería estándar de OCL

La librería estándar de OCL define un número de tipos. Entre ellos, tipos primitivos: `Integer`, `UnlimitedNatural`, `Real`, `String` y `Boolean`. Otra parte de la librería consiste en los tipos de colecciones. Ellos son `Bag`, `Set`, `Sequence` y `Collection` donde `Collection` es un tipo abstracto. Notar que todos los tipos definidos en la librería estándar son instancias de una clase de la sintaxis abstracta. La librería estándar existe en el nivel de modelado, también referido el nivel M1, donde la sintaxis abstracta es el metanivel o nivel M2. [13]

B.1. Los tipos OclAny, OclVoid, OclInvalid, y OclMessage

B.1.1. OclAny

Todos los tipos en el modelo UML y los tipos primitivos y colecciones en la librería estándar cumplen con el tipo `OclAny`. Conceptualmente, `OclAny` se comporta como un supertipo para todos los tipos. Los Features de `OclAny` están disponibles sobre cada objeto en todas las expresiones OCL. `OclAny` es una instancia del metatipo *AnyType*.

Todas las clases en un modelo UML heredan todas las operaciones definidas sobre `OclAny`. Para evitar conflictos de nombres entre propiedades en el modelo y las propiedades heredadas desde `OclAny`, todos los nombres de las propiedades de `OclAny` empiezan con 'ocl'.

B.1.2. OclMessage

Cada tipo de mensaje ocl, es un tipo de template con un parámetro. El tipo predefinido `OclMessage` es una instancia de `MessageType`. Cada `OclMessage` está determinado por la operación o señal dada como parámetro. Notar que hay conceptualmente un número indefinido de estos tipos, ya que cada uno está determinado por una operación o señal diferente. Estos tipos no tienen un nombre. Cada tipo tiene como atributos el nombre de la operación o señal. `OclMessage` es una instancia del metatipo `MessageType`.

B.1.3. OclVoid

El tipo `OclVoid` es un tipo que cumple con todos los tipos, excepto `OclInvalid`. Tiene una sola instancia, identificada como null, que corresponde con el valor de la especificación `LiteralNull` de UML. Cualquier propiedad aplicada en **null** resulta en **invalid**, excepto para las operaciones *oclIsUndefined()*, *oclIsInvalid()*, *=(OclAny)* and *<>(OclAny)*.

B.1.4. OclInvalid

El tipo `OclInvalid` es un tipo que cumple con todos los tipos. Tiene una sóla instancia, identificada cómo inválida. Cualquier propiedad aplicada en **invalid** resulta en **invalid**, excepto para las operaciones *oclIsUndefined()* and *oclIsInvalid()*. `OclInvalid` es una instance del metatipo `InvalidType`.

B.2. Operaciones

B.2.1. OclType

- `type.name(): String`
  El nombre del *type*.

- `type.attributes(): Set(String)`
  El conjunto de nombres de los atributos de *type*, como se definieron en el modelo.

- `type.associationEnds(): Set(String)`
  El conjunto de nombres de las associationEnds navegables de *type*, como están definidas en el modelo.

- `type.operations(): Set(String)`
  El conjunto de nombres de las operaciones de *type*, como están definidas en el modelo.

- `type.supertypes(): Set(OclType)`
  El conjunto de todos los supertipos directos de *type*.

  ```
  post: type.allSupertypes()->includesAll(result)
  ```

- `type.allSupertypes(): Set(OclType)`
  El conjunto de todos los supertipos de *type*.

- `type.allInstances(): Set(type)`
  El conjunto de todas las instancias de *type* y sus subtipos, al tiempo que la expresión es evaluada.

B.2.2. OclAny

- `=(object2: OclAny): Boolean`
  Es `True` si $self$ es el mismo objeto que $object2$. Es un operador infijo.

  ```
  post: result = (self = object2)
  ```

- `<> (object2: OclAny): Boolean`
  Es `True` si $self$ es diferente a $object2$. Es un operador infijo.

  ```
  post: result = not (self = object2)
  ```

- `oclIsNew(): Boolean`
  Puede usarse sólo en una postcondición. Evalúa a `True` si $self$ se crea durante la realización de la operación (por ejemplo, no existía en el momento pre-condición).

  ```
  post: self@pre.oclIsUndefined()
  ```

- `oclIsUndefined(): Boolean`
  Evalúa a `True` si $self$ es igual a **invalid** o **null**.

  ```
  post: result = self.isTypeOf( OclVoid ) or self.isTypeOf(OclInvalid)
  ```

- `oclIsInvalid(): Boolean`
  Evalúa a `True` si $self$ es igual a `OclInvalid`.

  ```
  post: result = self.isTypeOf( OclInvalid)
  ```

- `oclAsType(type: Classifier): T`
  Evalúa a *self*, donde *self* es del tipo identificado por T. El tipo T puede ser un classifier definido en el modelo UML; si el tipo actual de *self* al tiempo de la evaluación no cumple con T, entonces la operación *oclAsType* evalúa a **invalid**. En el caso de la redefinición de un feature, cuando se convierte un objeto a un supertipo de su tipo real, no se tiene acceso a la definición de su supertipo. Sin embargo, cuando se convierte a un supertipo, cualquier feature definido por el subtipo son suprimidas.

  `post: (result = self) and result.oclIsTypeOf( t )`

- `oclIsTypeOf(type: Classifier): Boolean`
  Evalúa a `true` si *self* es del tipo t pero no un subtipo de t.

  `post: self.oclType() = type`

- `oclIsKindOf(type: Classifier): Boolean`
  Evalúa a `True` si el tipo de *self* cumple a t. Esto es, *self* es del tipo t o un subtipo de t.

  `post: self.oclType().conformsTo(type)`

- `oclIsInState(statespec: OclState): Boolean`
  Evalúa a `True` si *self* está en un estado identificado por *statespec*.

  `post: -- TBD`

- `oclType(): Classifier`
  Evalúa al tipo del cual *self* es una instancia.

  `post: self.oclIsTypeOf(result)`

- `oclLocale: String`
  Define el entorno local por omisión para las operaciones de la biblioteca, como `String:: toUpperCase ()`.

B.2.3. OclVoid

- `= (object: OclAny): Boolean`
  Redefine la operación `OclAny`, retornando `true` si el objeto es **null**.

  `post: result = object.oclIsTypeOf(OclVoid)`

B.2.4. OclMessage

- `hasReturned(): Boolean`
  Es `true` si el tipo del parámetro del template es una llamada de la operación, y la operación llamada ha retornado un valor. Esto implica el hecho que el mensaje ha sido enviado. Es `false` en los demás casos.

  `post: --`

- `result(): «The return type of the called operation»`
  Retorna el resultado de la operación llamada, si el tipo de parámetro del template es una llamada de operación, y la operación llamada ha retornado un valor. De otra forma, se retorna el valor *invalid*.

  `pre: hasReturned()`

- `isSignalSent(): Boolean`
  Retorna `true` si el `OclMessage` representa el envío de una señal UML.

- `isOperationCall(): Boolean` Retorna `true` si el `OclMessage` representa el envío de una llamada a una Operación UML.

B.3. Tipos primitivos

Los tipos primitivos definidos en la librerá estándar OCL son `UnlimitedNatural`, `Integer`, `Real`, `String`, y `Boolean`. Ellos son todas las instancias de la metaclase `Primitive` del UML core package.

- `Real`: El tipo estándar `Real` representa el concepto matemático de real. Notar que `Integer` es superclase de `UnlimitedNatural` e `Integer` es una subclase de `Real`. `Real` es una instancia del metatipo `PrimitiveType`.
- `Integer`: El tipo estándar `Integer` representa el concepto matemático de integer. Notar que `UnlimitedNatural` es una subclase de `Integer`. `Integer` es una instancia del metatipo `PrimitiveType`.
- `String`: El tipo stándar `String` representa strings, que pueden ser ASCII o Unicode. `String` es una instancia del metatipo `PrimitiveType`.
- `Boolean`: El tipo estándar `Boolean` representa los valores comunes `true/false`. `Boolean` es una instancia del metatipo `PrimitiveType`.
- `UnlimitedNatural`: El tipo estándar `UnlimitedNatural` es usado para codificar los valores no negativos de una multiplicidad en especificaciones. Esto incluye un valor especial ilimitado (*) que codifica el upper value de una multiplicidad en una especificación. `UnlimitedNatural` es una instancia del metatipo `UnlimitedNaturalType`. Notar que aunque `UnlimitedNatural` es una subclase de `Integer`, el valor ilimitado no puede ser representado como un `Integer`. Cualquier uso del valor ilimitado como integer o real es reemplazado por el valor **invalid**.

B.4. Operaciones de tipos primitivos

B.4.1. Real

- `r=(r2: Real): Boolean`
  `True` si $r$es igual a $r2$.
- `r <> (r2: Real): Boolean`
  `True` is r no es igual a $r2$. $post : result = not(r = r2)$.
- `r + (r2: Real): Real`
  El valor de sumar r y $r2$.
- `r - (r2: Real): Real`
  El valor de restar r y $r2$.
- `r * (r2: Real): Real`
  El valor de multiplicar r y $r2$.
- `- r: Real`
  El valor negativo de r.
- `r / (r2: Real): Real`
  El valor de dividir r por $r2$.
- `r.abs(): Real`
  El valor absoluto de r.

  ```
  post: if r < 0  then  result = - r  else  result = r endif
  ```

- `floor(): Integer`
  El mayor entero que es menor o igual a sí mismo.

  ```
  post: (result <= r) and (result + 1 > r)
  ```

- `round(): Integer`
  Retorna el integer mas cercano a r. Cuando hay dos integer, el mayor de ellos.

  ```
  post: ((r - result).abs() < 0.5) or
  ((r - result).abs() = 0.5 and (result > r))
  ```

- `max(r2: Real): Real`
  Retorna el máximo de r y $r2$.

  ```
  post: if self >= r2 then result = r else result = r2 endif
  ```

- `min(r2: Real): Real`
  Retorna el mínimo de r y $r2$.

  ```
  post: if r <= r2 then result = r else result = r2 endif
  ```

- `< (r2: Real): Boolean`
  Retorna `True` si r es menor que $r2$.

- `> (r2: Real): Boolean`
  Retorna `True` si r es mayor que $r2$.

  ```
  post: result = not (r <= r3)
  ```

- `<= (r2: Real): Boolean`
  Retorna `true` si r es menor o igual que $r2$.

  ```
  post: result = ((r = r2) or (r < r2))
  ```

- `>= (r2: Real): Boolean`
  Retorna `true` si r es mayor o igual que $r2$.

  ```
  post: result = ((r = r2) or (r > r2))
  ```

B.4.2. Integer

- `i = (i2: Integer): Boolean`
  `True` si i es igual a $i2$.

- `- i: Integer`
  El valor negativo de i.

- `i + (i2: Integer): Integer`
  El valor de sumar i e $i2$.

- `i - (i2: Integer): Integer`
  El valor de restar i e $i2$.

- `i * (i2: Integer): Integer`
  El valor de multiplicar i e $i2$.

- `i / (i2: Integer): Real`
  El valor de dividir i por $i2$.

- `i.abs(): Integer`
  El valor absoluto de i.

  ```
  post: if i < 0 then result = - i else result = i endif
  ```

- `i.div( i2 : Integer) : Integer`
  El número de veces que $i2$ entra completamente en i.

```
pre : i2 <> 0
post:
   if i / i2 >= 0
     then result = (i / i2).floor()
     else result = -((-i/i2).floor())
   endif
```

- `i.mod(i2: Integer): Integer`
  El resultado es i módulo $i2$.

  ```
  post: result = i - (i.div(i2) * i2)
  ```

- `i.max(i2: Integer): Integer`
  Retorna el máximo de i y $i2$.

  ```
  post: if i >= i2 then result = i else result = i2 endif
  ```

- `i.min(i2: Integer): Integer`:
  Retorna el mínimo de i y $i2$.

  ```
  post: if i <= i2 then result = i else result = i2 endif
  ```

B.4.3. String

- `string = (string2: String): Boolean`
  `True` si el *string* y el *string*2 contienen los mismos caracteres en el mismo orden.

- `string.size(): Integer`
  El número de caracteres de *string*.

- `string.concat(string2: String): String`
  La concatenación de *string* y *string*2.

  ```
  post: result.size() = string.size() + string2.size()
  post: result.substring(1, string.size() ) = string
  post: result.substring(string.size() + 1, result.size() ) = string2
  ```

- `string.toUpper(): String`
  El valor de *string* con todos los caracteres que tenía en minúscula, pasados a mayúscula.

  ```
  post: result.size() = string.size()
  ```

- `string.toLower(): String`
  El valor de *string* con todos los caracteres que tenía en mayúscula, pasados a minúscula.

  ```
  post: result.size() = string.size()
  ```

- `string.substring(lower: Integer, upper: Integer): String`
  El sub-string de *string* que comienza en el caracter lower hasta el upper.

B.4.4. Boolean

- `b = (b2: Boolean): Boolean`
  Es igual si b es lo mismo que $b2$.

- `b or (b2: Boolean): Boolean`
  `True` si b o $b2$ son `true`.

- `b xor (b2: Boolean): Boolean`
  `True` si b o $b2$ son `true`, pero no ambos.

  `post: (b or b2) and not (b = b2)`

- `b and (b2: Boolean): Boolean`
  `True` si $b1$ y $b2$ son `true`.

- `not b: Boolean`
  `True` si b es `false`.

  `post: if b then result = false else result = true endif`

- `b implies (b2: Boolean): Boolean`
  `True` si b es `false`, o si b es `true` y $b2$ es `true`.

  `post: (not b) or (b and b2)`

- `if b then (expression1: OclExpression) else (expression2: OclExpression) endif: expression1.evaluationType()`
  Si b es `true`, el resultado es el valor de evaluar *expression*1; sino el resultado es el valor de evaluar *expression*2.

B.4.5. Enumeration

- `enumeration = (enumeration2: Boolean): Boolean`
  Igual si *enumeration* es igual a *enumeration*2.

- `enumeration <> (enumeration2: Boolean): Boolean`
  Igual si *enumeration* no es la misma que *enumeration*2.

  `post: result = not ( enumeration = enumeration2)`

Apéndice C

Clases de implementación del plugin OCL

C.1. Análisis sintáctico

La responsabilidad principal del intérprete OCL es anlizar sintácticamente las expresiones OCL [3]. Uno de los propósitos de analizar una expresión es validarla: si se puede analizar correctamente está bien formada (el parser automáticamente valida la expresión contra las reglas semánticas de buena formación).

El punto principal de entrada a la API OCL es la clase OCL, que provee un entorno de análisis sintáctico autónomo. Registra todas las restricciones que son analizadas en este entorno, incluyendo las definiciones de operaciones adicionales y atributos. el método factory (factory method) `OCL.newInstance()` es utilizado para crear un nuevo OCL con un `EnvironmentFactory` que provee el binding a un metamodelo particular (Ecore o UML). Para parsear una expresión de consulta, se utiliza el object `OCLHelper`, que provee operaciones para análisis sintáctico de consultas y restricciones (pensadas para procesar restricciones embebidas en modelos).

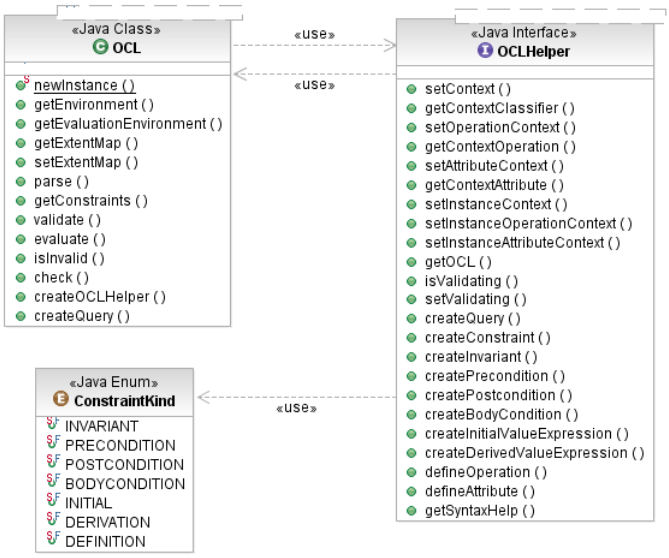

Figura C.1: OCL Plugin OCL Helper

OCL está diseñado principalmente para la especificación de restricciones. A diferencia de las consultas, hay diferentes tipos de restricciones que pueden ser utilizadas en diferentes partes de

un modelo. Esta clase de restricciones incluye, invariantes de clases, restricciones de operaciones y restricciones de derivación de atributos. La clase `OCLHelper` puede analizar sintácticamente esas restricciones.

C.2. Árbol sintáctico abstracto

El modelo de sintaxis abstracta de OCL está definido en la especificación 2.3 del lenguaje OCL. La implementación de Eclipse de OCL define ciertas extensiones a este modelo que proveen servicios adicionales. La más importante es el soporte del patrón Visitor.

C.2.1. Las interfaces Visitable y Visitor

Todas las metaclases en el modelo de sintaxis abstracta de OCL (nodos del AST) que pueden ser visitadas implementan la interface `Visitable`. Define una única operación `accept(Visitor)`, este método delega al método apropiado `visitXyz(Xyz)` del `Visitor`. Los implementadores directos de la interface `Visitable` son `OCLExpression` y las metaclases del paquete `Expressions` que no conforman con `OCLExpression`:

- `Variable`
- `CollectionLiteralPart`
- `TupleLiteralPart`
- `ExpressionInOCL`

La última clase no está definida en el paquete `Expressions` porque se refiere a la ubicación de OCL en elementos restricciones en modelos. El parser OCL, internamente, define implementaciones de visitors, incluyendo la clase `ValidationVisitor` para validar expresiones OCL y la clase `EvaluationVisitor` para evaluar expresiones OCL.

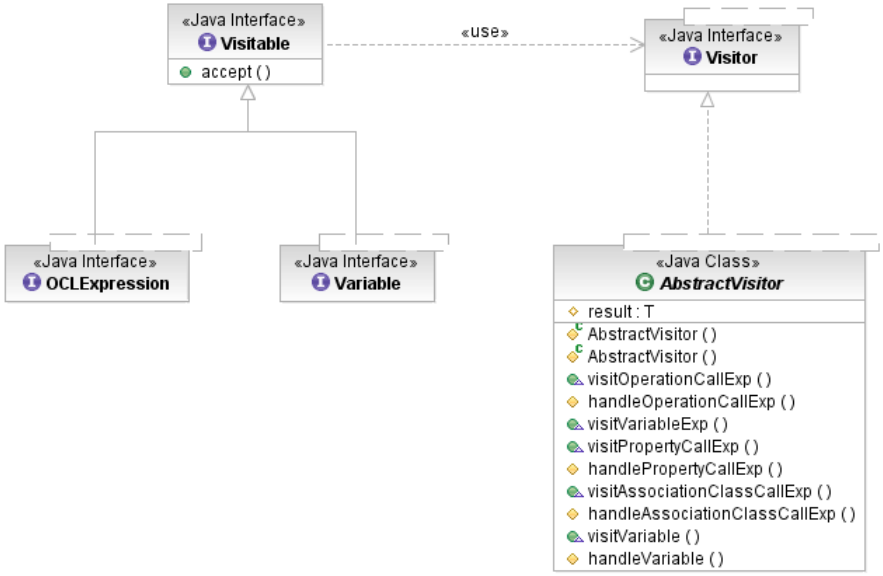

Figura C.2: OCL Plugin Visitor

C.2.2. Implementando el patrón Visitor

Para implementar un visitor se debe extender la clase `AbstractVisitor`, que provee una variable *result* del tipo genérico de parámetro de tipo `T` para almacenar el resultado calculado por el visitor y una forma de sobreescribir selectivamente métodos para procesar sólo aquellos nodos de interés.

La clase `AbstractVisitor` provee implementaciones de todos los métodos *visitXyz* de la interface que simplemente retornan el valor actual del resultado. Además, para todos los nodos internos del árbol sintáctico (e.g. `OperationCallExp` e `IfExp`), el método *visitXyz* visita recursivamente los nodos hijos y retorna los resultados de procesar esos nodos al método *handleXyz* que las subclases pueden sobreescribir para calcular un resultado desde los resultados de los nodos hijos. De esa forma, una superclase sólo necesita sobreescribir selectivamente las implementaciones por defecto de los métodos *visitXyz* para los nodos hoja del árbol y los métodos *handleXyz* para los nodos internos.

C.2.3. Clases del árbol sintáctico

El diagrama de la figura C.3 resume las metaclases del paquete `Types`

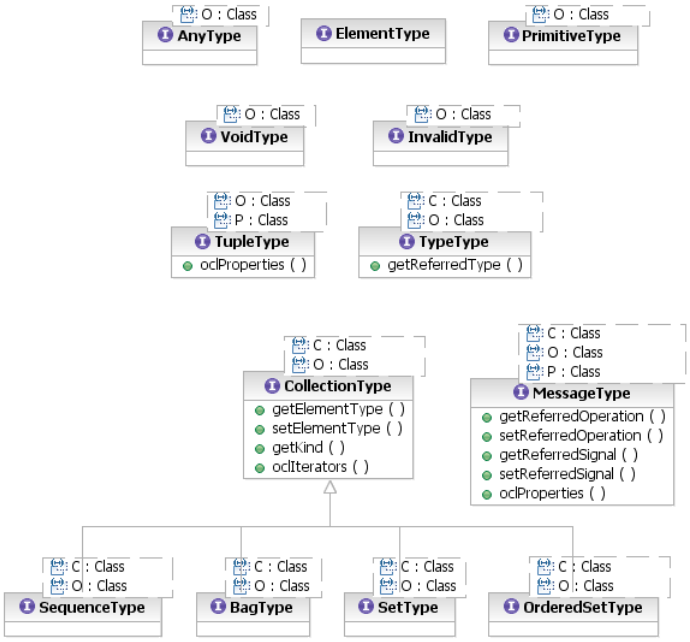

Figura C.3: OCL Plugin implementación Types

El diagrama de la figura C.4 resume las metaclases de expresiones de llamada a operación del paquete `Expressions`.

El diagrama de la figura C.5 resume las metaclases de expresiónes de literales del paquete `Expressions`.

El diagrama de la figura C.6 resume las metaclases restantes del paquete `Expressions`.

C.3. AbstractVisitor

La clase `AbstractVisitor` se encuentra definida en el plugin de OCL para Eclipse. En la figura C.2 se muestra el diagrama de clases de la clase. La definición de la clase es la siguiente:

```
public abstract class AbstractVisitor<T, C, O, P, EL, PM, S, COA, SSA, CT>
implements Visitor<T, C, O, P, EL, PM, S, COA, SSA, CT>
```

En la tabla C.1 se indica la relación entre los parámetros de tipo y el elemento correspondiente del metamodelo UML.

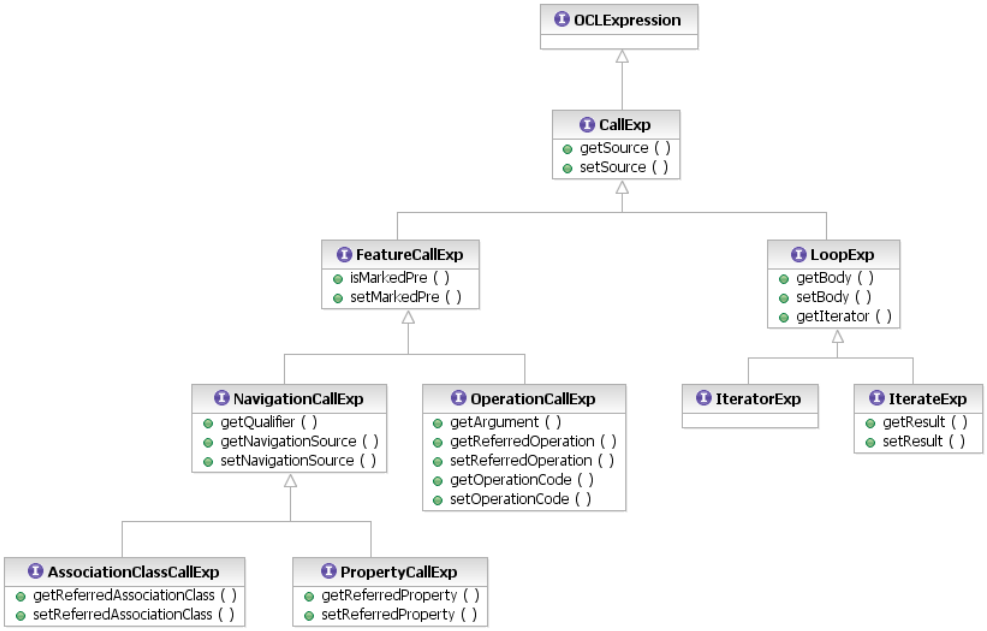

Figura C.4: OCL Plugin implementación CallExp

Parámetro de tipo	**Metamodelo UML**
T	Tipo de retorno del recorrido el árbol sintáctico
C	Corresponde a la metaclase UML `Classifier`
O	Corresponde a la metaclase UML `Operation`
P	Corresponde a la metaclase UML `Property`
EL	Corresponde a la metaclase UML `EnumerationLiteral`
PM	Corresponde a la metaclase UML `Parameter`
S	Corresponde a la metaclase UML `State`
COA	Corresponde a la metaclase UML `CallOperationAction`
SSA	Corresponde a la metaclase UML `SendSignalAction`
CT	Corresponde a la metaclase UML `Constraint`
CLS	Corresponde a la metaclase UML `Class`
E	Corresponde a la metaclase UML `Element`

Tabla C.1: Parámetros de tipo de la clase `AbstractVisitor`

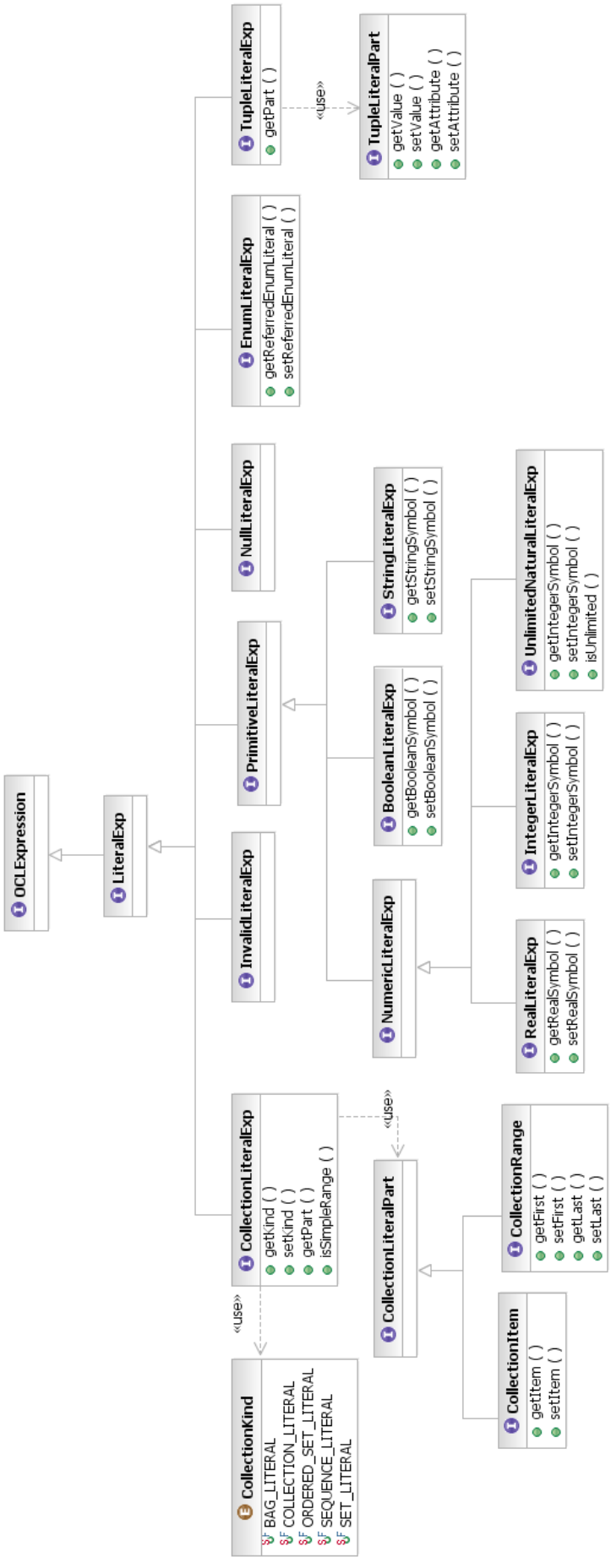

Figura C.5: OCL Plugin implementación LiteralExp

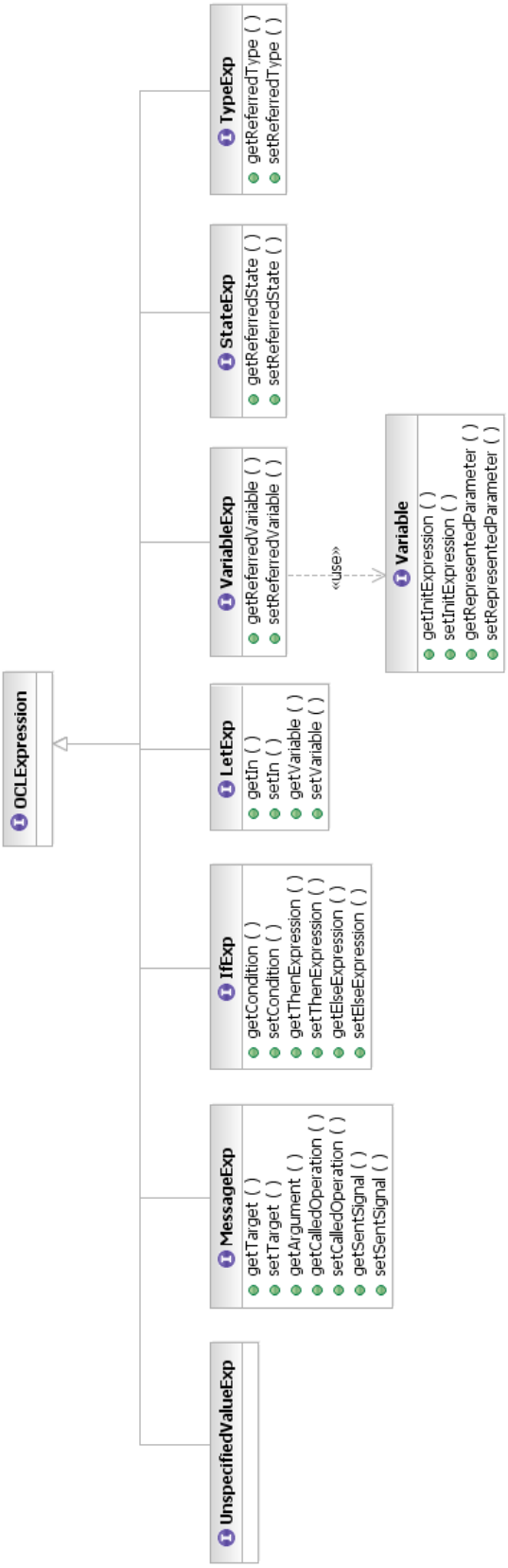

Figura C.6: OCL Plugin implementación metaclases restantes

C.3.1. Valor de retorno

La variable *result* de tipo `T` es conveniente para acumular el resultado de la visita. En la subclase que extienda `AbstractVisitor`, simplemente se debe asignar o modificar el valor de retorno en los métodos de visita sobreescritos, según sea necesario. El framework garantiza que el resultado sea retornado como el valor total de la llamada al método `Visitable#accept(Visitor)`.

C.3.2. Métodos

Existen dos tipos de nodos en una expresión OCL: los nodos hoja y los nodos internos. Una clase que extienda `AbstractVisitor` sólo necesita redefinir selectivamente los métodos *handleXxxx* para los nodos internos del árbol y los métodos *visitXxxx* para los nodos hoja. En la tabla C.2 se muestran los métodos.

Método
`public T visitVariableExp(VariableExp<C, PM>v)`
`public T visitTypeExp(TypeExp<C>t)`
`public T visitUnspecifiedValueExp(UnspecifiedValueExp<C>unspecExp)`
`public T visitStateExp(StateExp<C, S>stateExp)`
`public T visitIntegerLiteralExp(IntegerLiteralExp<C>literalExp)`
`public T visitRealLiteralExp(RealLiteralExp<C>literalExp)`
`public T visitStringLiteralExp(StringLiteralExp<C>literalExp)`
`public T visitBooleanLiteralExp(BooleanLiteralExp<C>literalExp)`
`public T visitNullLiteralExp(NullLiteralExp<C>literalExp)`
`public T visitInvalidLiteralExp(InvalidLiteralExp<C>literalExp)`
`public T visitEnumLiteralExp(EnumLiteralExp<C, EL>literalExp)`
`public T visitUnlimitedNaturalLiteralExp(UnlimitedNaturalLiteralExp literalExp)`

Tabla C.2: Métodos *visitXxxx* de la clase `AbstractVisitor`

C.4. Patrón de diseño Visitor

En programación orientada a objetos el patrón de diseño Visitor es una forma de separar un algoritmo de la estructura de objetos sobre la cual opera. Un resultado práctico de esta separación es la posibilidad/habilidad de agregar nuevas operaciones a estructuras de objetos sin modificar esas estructuras.

El patrón Visitor permite agregar nuevas funciones virtuales a una familia de clases sin modificar las clases mismas, en lugar de eso, se crea una clase Visitor que implementa todas las especializaciones apropiadas de la función virtual. La instancia del Visitor toma la referencia de la instancia como input e implementa la funcionalidad utilizando double dispatching. En la figura C.2 se muestra el diagrama de clases del patrón Visitor.

Si bien es poderoso, el patrón Visitor es más limitado que las funciones virtuales convencionales. No es posible crear visitors para objetos sin agregar un pequeño método de callback en cada clase.

C.4.1. Detalles de implementación

Un objeto usuario recibe una referencia a otro objeto que implementa un algoritmo. El primero es designado como perteneciente a una clase "elemento" y el segundo a una clase "visitor". La idea es utilizar una estructura de clases elemento, cada una de las cuales tiene un método *accept* que recibe un objeto visitor como parámetro. La clase del objeto visitor implementa un protocolo (inteface en Java) que tiene un método *visit* para cada clase elemento, es decir, habrá métodos de la forma: *visitClase1*, *visitClase2*, *visitClaseN*. El método *accept* de una clase elemento llama al

método *visit* para su clase. Las clases visitor concretas pueden ser escritas para realizar operaciones particulares, implementando estas operaciones en sus respectivos métodos *visit*.

Cada método *visit* de un visitador concreto puede ser pensado como un método que no es de una sola clase, sino de un par de clases: el visitador concreto y clase elemento particular. Así el patrón visitor simula el double dispatching en un lenguaje convencional orientado a objetos de single dispatch, como ser Java o C++.

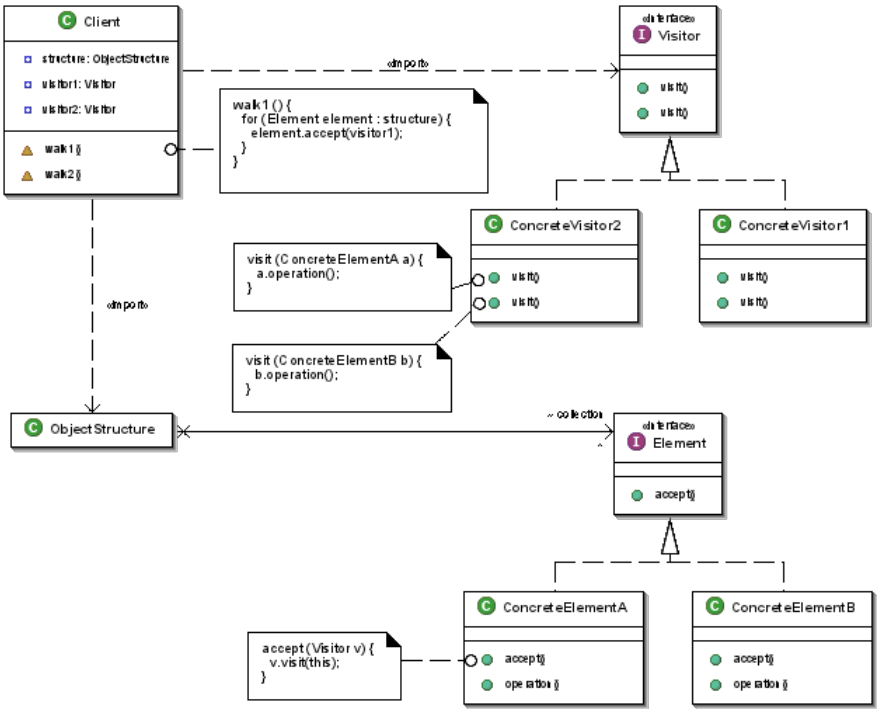

Figura C.7: Diagrama UML patrón de diseño Visitor

Apéndice D

Get Ready

D.1. Introducción

En la Sección D.2 se explicará el proceso de exportación del plugin realizado. En la Sección D.3 se explica la preparación del eclipse para poder utilizar nuestra herramienta, En la Sección D.5 se mostrará como realizar un diagrama de clases con Papyrus. En la Sección D.7 se mostrarán las restricciones a utilizar y como se ven en Eclipse.

D.2. Plugin

Lo primero que debemos hacer es generar un *.jar* de nuestro plugin, para que esté integrado con Eclipse.

Lo primero que debemos hacer es asegurarnos que todas las librerías externas se encuentren agregadas al *.xml* del plugin. En nuestro caso al haber utilizado la librería *guava-10.0.1.jar*, debemos agregarla al plugin de la siguiente manera:

- Abrir el archivo *.xml* de nuestro plugin.
- Ir a la pestaña Runtine y luego en la opción Classpath, elegir la opción Add y buscar el archivo que necesitamos, en nuestro caso el *guava-10.0.1.jar*.

Ahora si, podemos exportar nuestro plugin con la siguiente secuencia de pasos [4]:

1. Abrir el archivo *plugin.xml* de nuestro proyecto del plugin.
2. Luego ir al tab **Overview** y elegir la opción **Export Wizard** que se encuentra dentro de la sección **Exporting**. (Figura D.1)
3. Elegir el directorio donde deseamos guardarlo. (Figura D.2)
4. El *.jar* generado debemos guardarlo en el directorio “dropin” de nuestro eclipse que vamos a utilizar.

En las secciones 10.3.1 y 10.3.2, se explicará como usar el plugin.

D.3. Preparación del eclipse

D.3.1. Instalación de MOFScript

El eclipse que debemos utilizar es el Eclipse Modeling Tools que puede bajarse de http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/indigor.

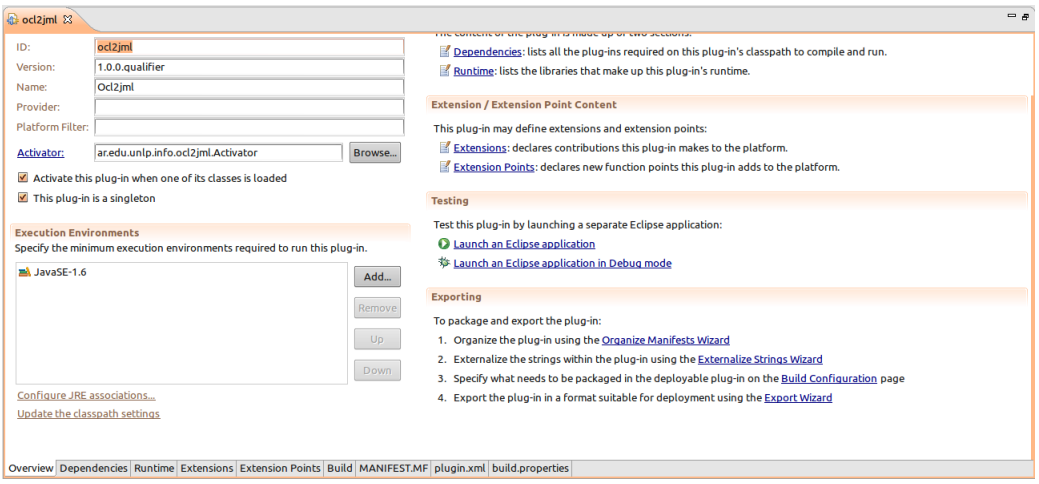

Figura D.1: Tab Overview de plugin.xml

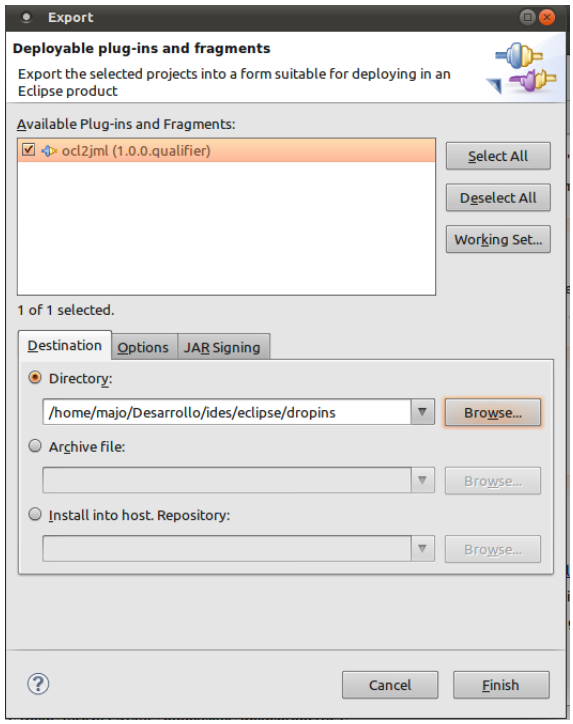

Figura D.2: Export del plugin

Una vez descomprimido el archivo y haber ingresado por primera vez, debemos bajarnos MOFScript. Esto se hace a través de Help -> Install new Software. Se nos presentará la Figura D.3.

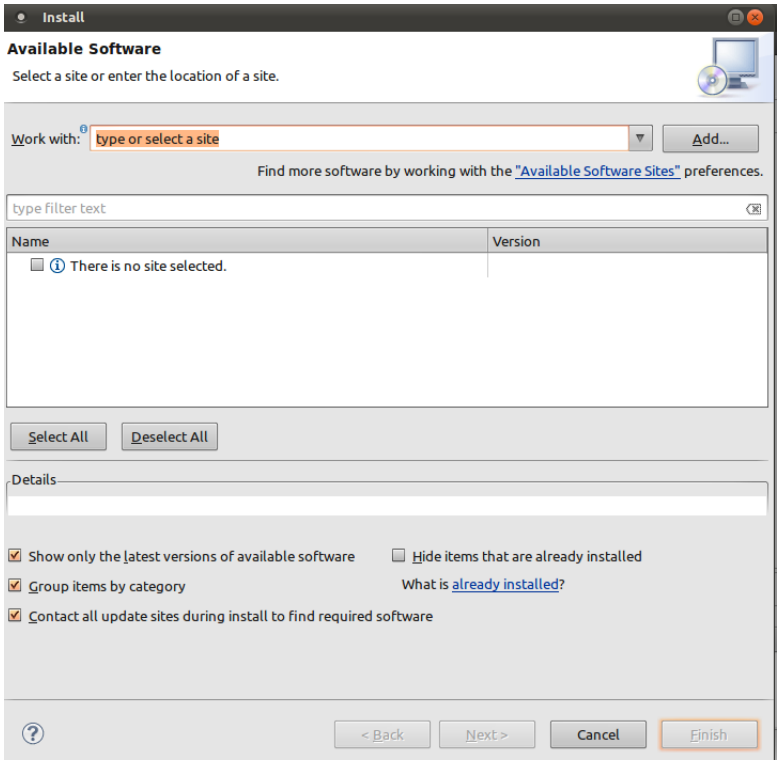

Figura D.3: Paso 1

Presionamos Add y completamos en Name: con **MOFScript** y en Location: con (como se muestra en la Figura D.4) y presionamos OK.

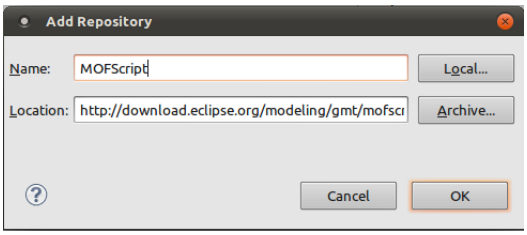

Figura D.4: Paso 2

Ahora se nos presentará la Figura D.5 donde seleccionamos MOFScript y luego Next. Se nos presentará la Figura D.6 donde debemos presionar nuevamente Next.

Aceptamos los términos de la licencia (Figura D.7) y presionamos Finish. Ahora debemos esperar a que se baje el software. Cuando se termine se nos presentará la Figura D.8 para que reiniciemos. Hacemos click en Restart now y esperamos a que vuelva a iniciar.

D.3.2. Instalación de OCLTools

Ir a Help -> Install Modelling Components. Se nos presentará una pantalla como la de la Figura D.9 donde debemos seleccionar OCL Tools y luego Finish. Se nos presentará una pantalla como

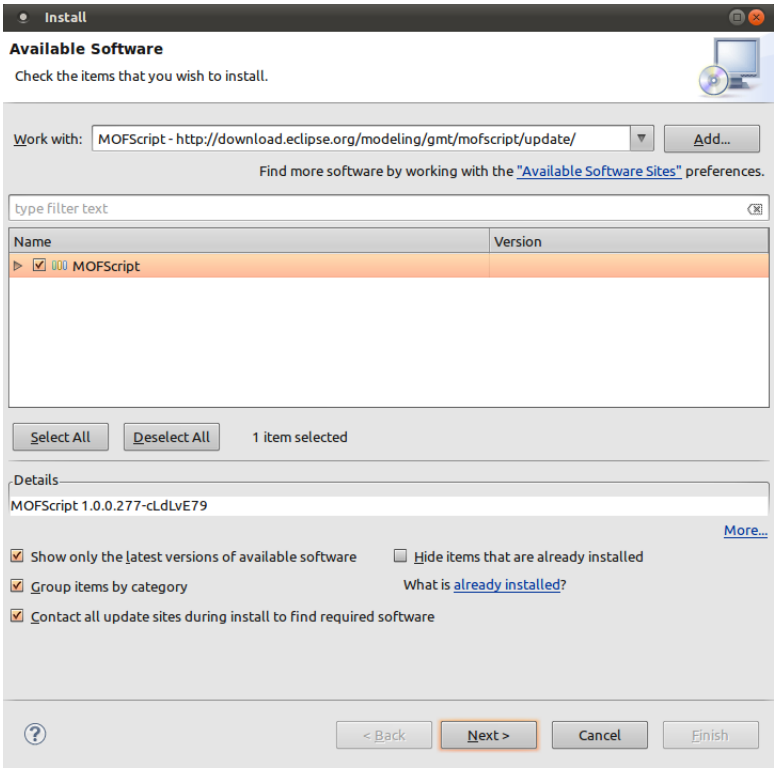

Figura D.5: Paso 3

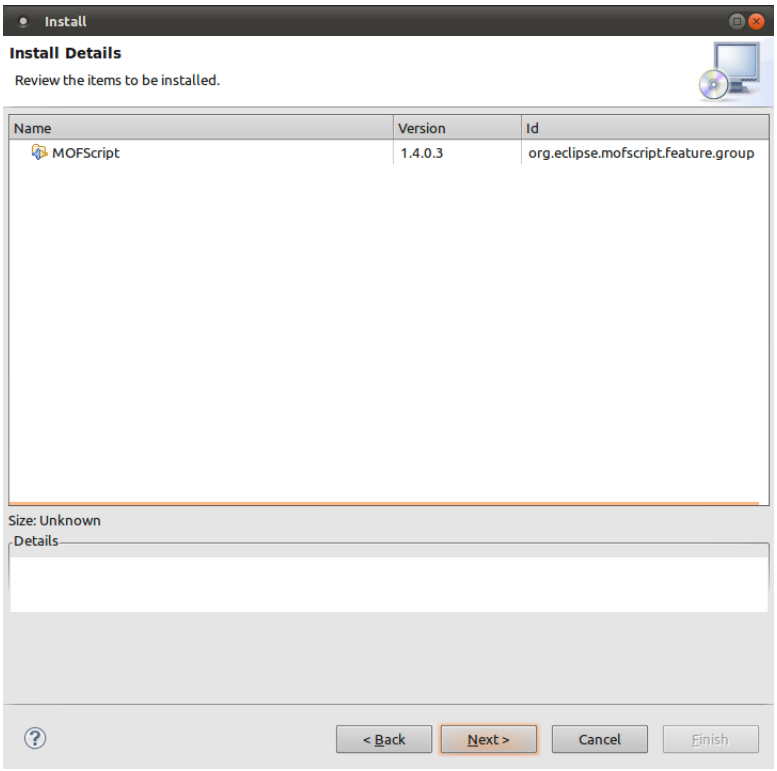

Figura D.6: Paso 4

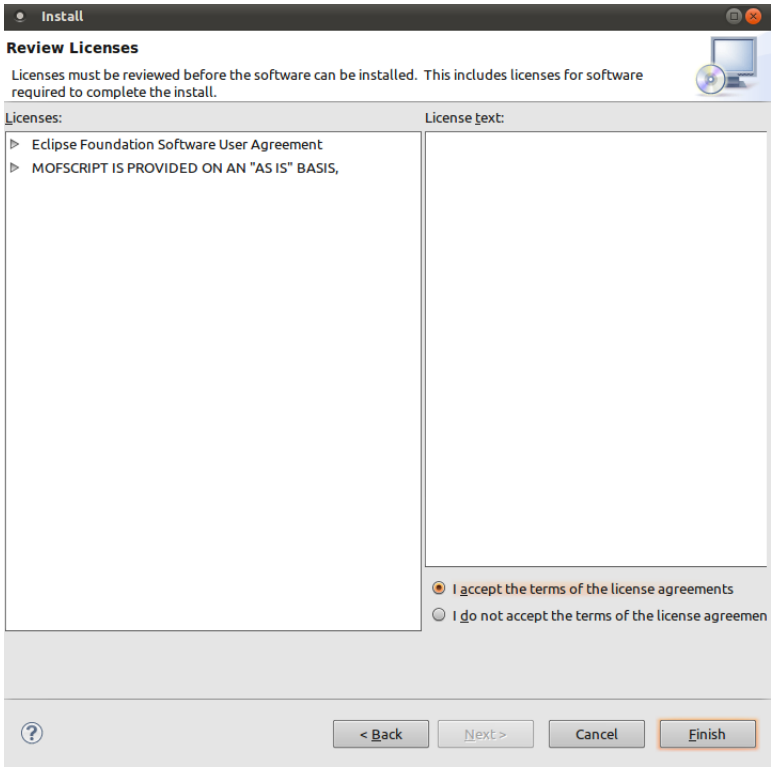

Figura D.7: Paso 5

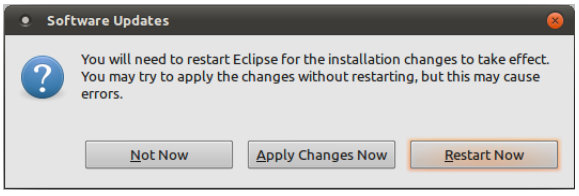

Figura D.8: Paso 6

la de la Figura D.10. Presionamos Next, aceptamos los términos de la licencia (Figura D.11) y presionamos Finish.

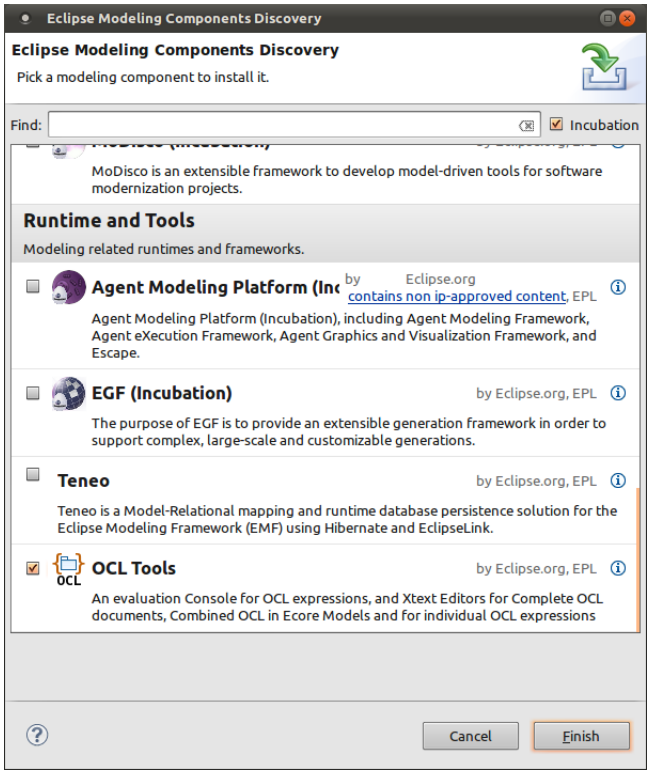

Figura D.9: Paso 7

Esperamos que que se baje el software. Se nos presentará una pantalla como la Figura D.8. Reiniciamos el eclipse. Y ya tenemos listo el eclipse para utilizarlo.

D.4. Importación del plugin

En el eclipse de trabajo, sobre el proyecto creado, debemos:

- Hacer click con el botón derecho sobre Referenced Libraries -> Build Path -> Configure Build Path. En la Figura D.12 se muestra una imagen de la opción descripta anteriormente.
- Se nos presentará un pop-up como el que se muestra en la Figura D.13
- Hacer click en Add External JARs y seleccionar el *.jar* exportado en la sección anterior.
- Se nos mostrará nuevamente el popup de las propiedades pero actualizado. En la Figura D.14 se muestra dicha actualización.
- Por último hacer click en Ok.

D.5. Diagrama de clases

Luego que tenemos nuestro plugin, debemos crear nuestro diagrama de clases. Nosotros utilizamos la herramienta Papyrus. La serie de pasos a seguir para crear un nuevo diagrama de clases son:

1. Ir a la vista de Papyrus.
2. Creamos una carpeta con el nombre que deseamos para nuestro modelo.

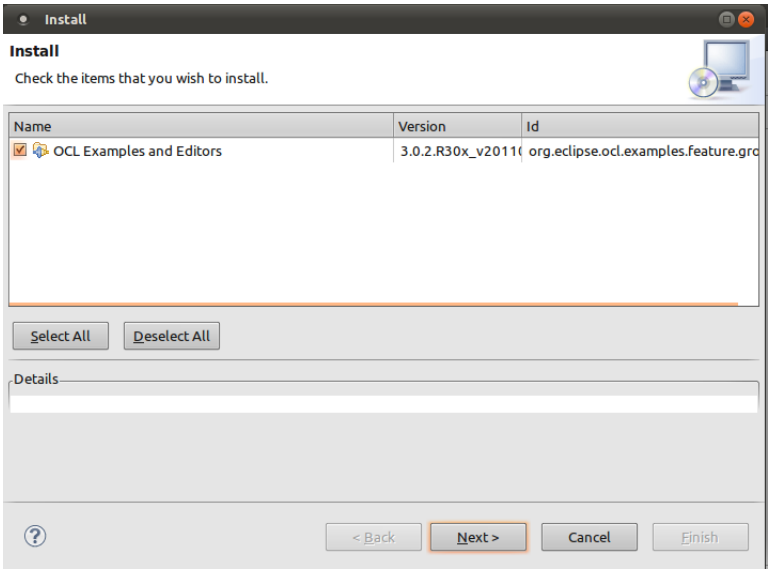

Figura D.10: Paso 8

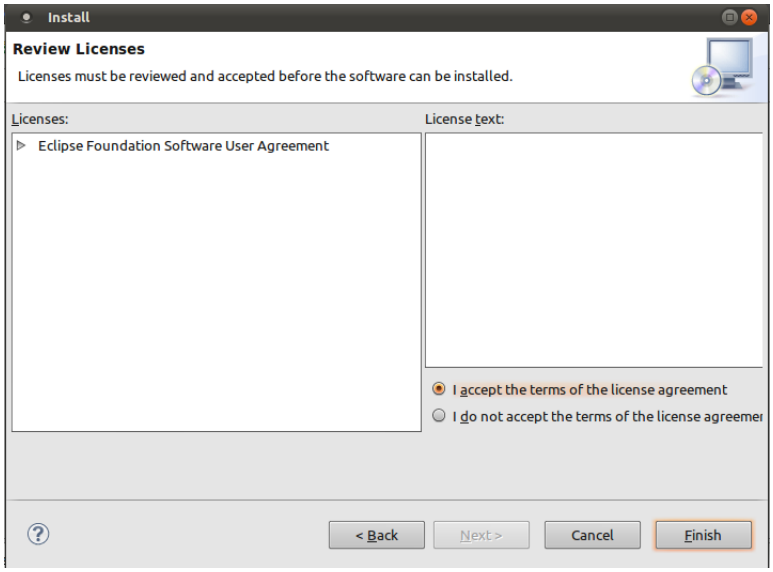

Figura D.11: Paso 9

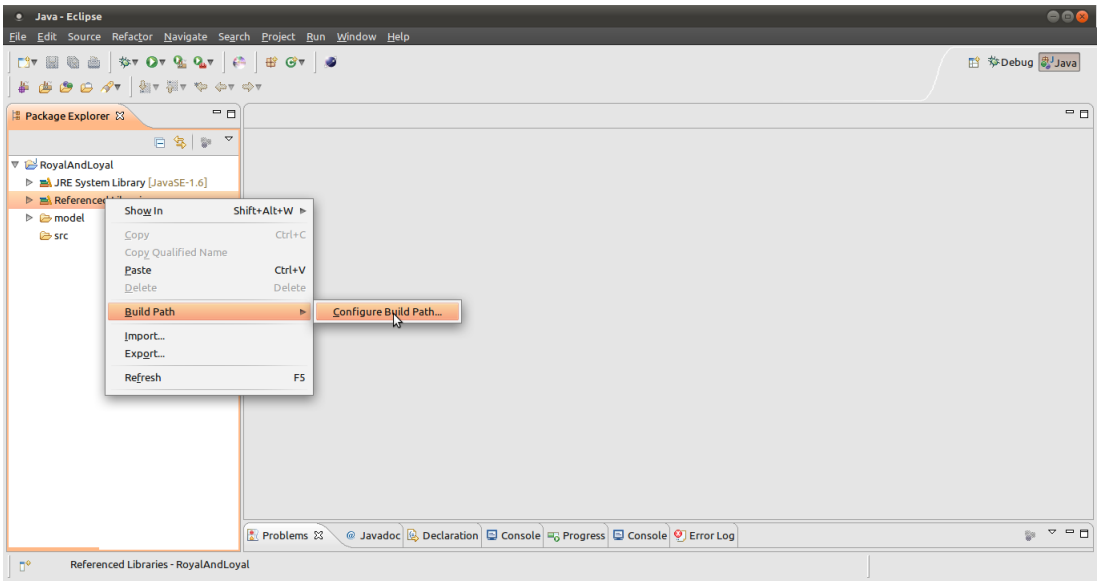

Figura D.12: Build Path

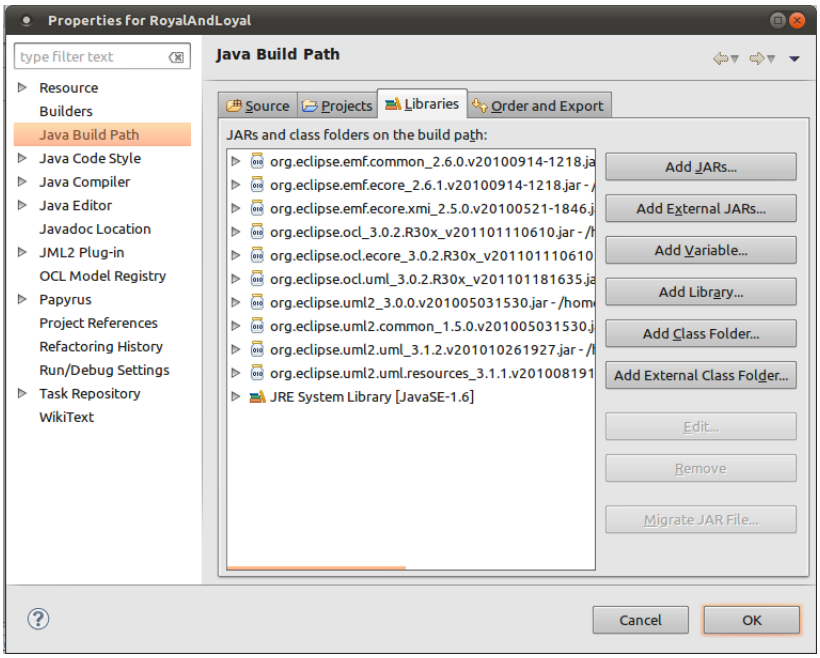

Figura D.13: Properties

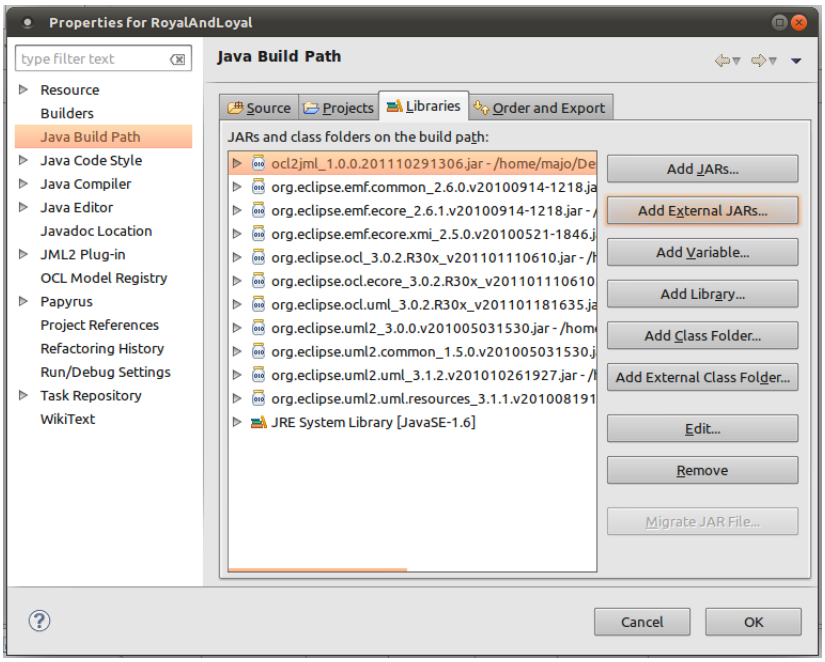

Figura D.14: Properties Actualizado

3. Sobre esa carpeta hacemos click derecho New -> Other.
4. Elegimos de la lista Papyrus Model (Figura D.15) y presionamos Next.

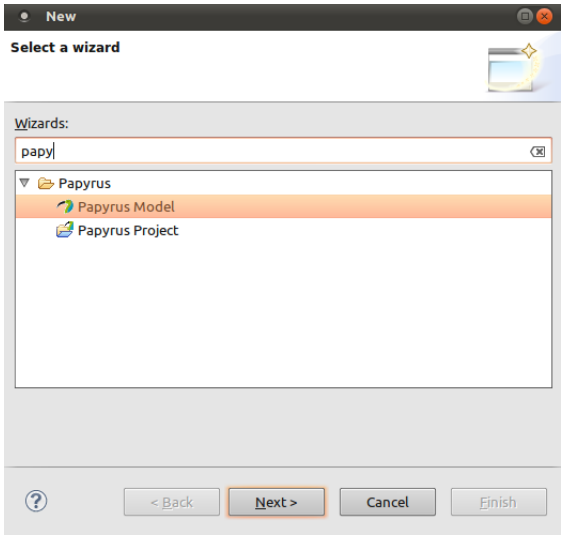

Figura D.15: Papyrus Model

5. Elegimos el nombre para nuestro modelo (Figura D.16) y presionamos Next
6. Dejamos seleccionada la opción UML (Figura D.17) y presionamos Next.
7. En esta pantalla debemos elegir el nombre de nuestro modelo y el tipo de diagrama que vamos a realizar (para nuestro caso será un diagrama de clases) (Figura D.18) y presionamos Finish.
8. Se nos abrirá un nuevo diagrama, en el cual podemos empezar a trabajar (Figura D.19).

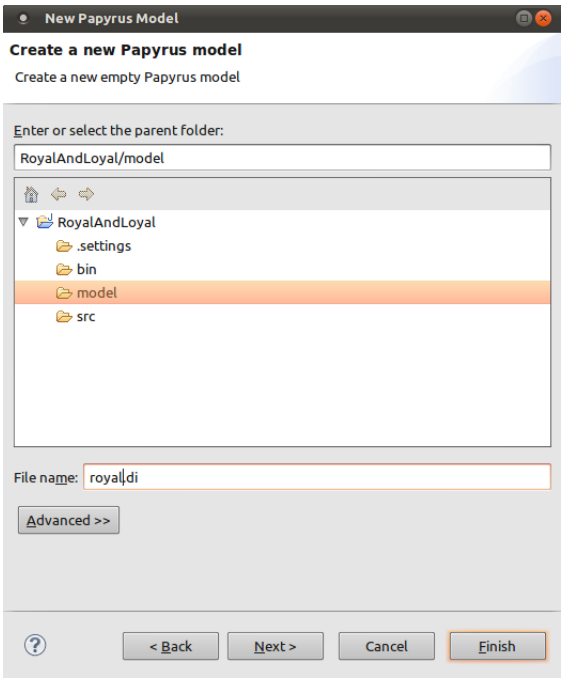

Figura D.16: Name Model

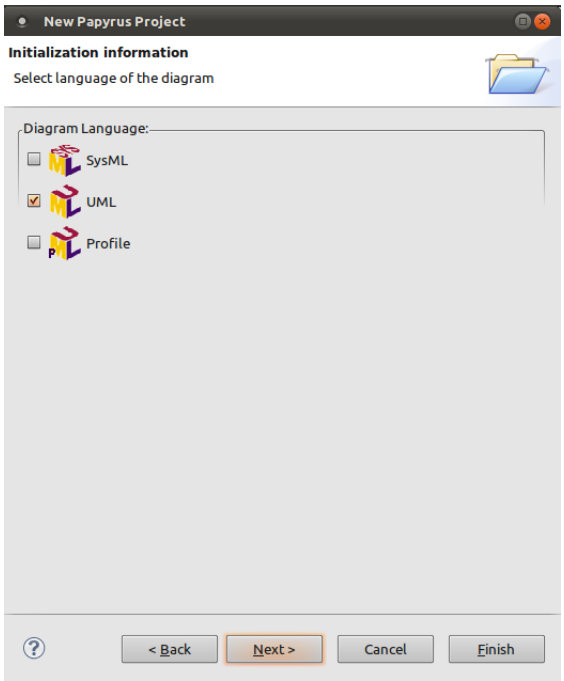

Figura D.17: Uml Check

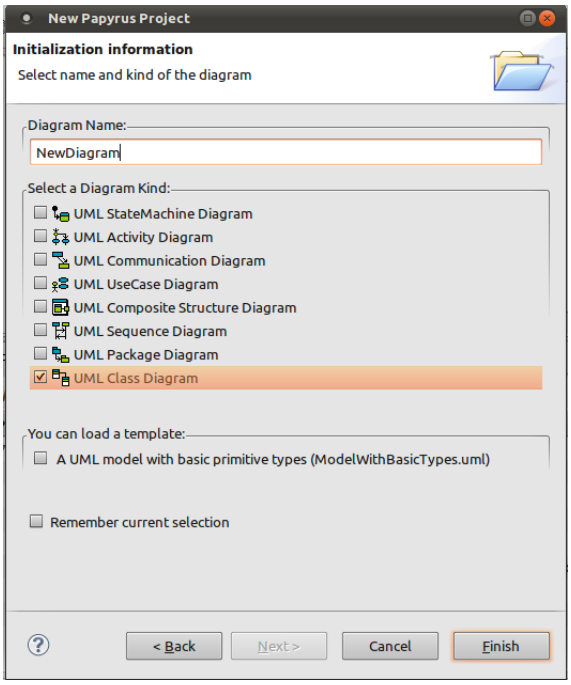

Figura D.18: Diagram Name

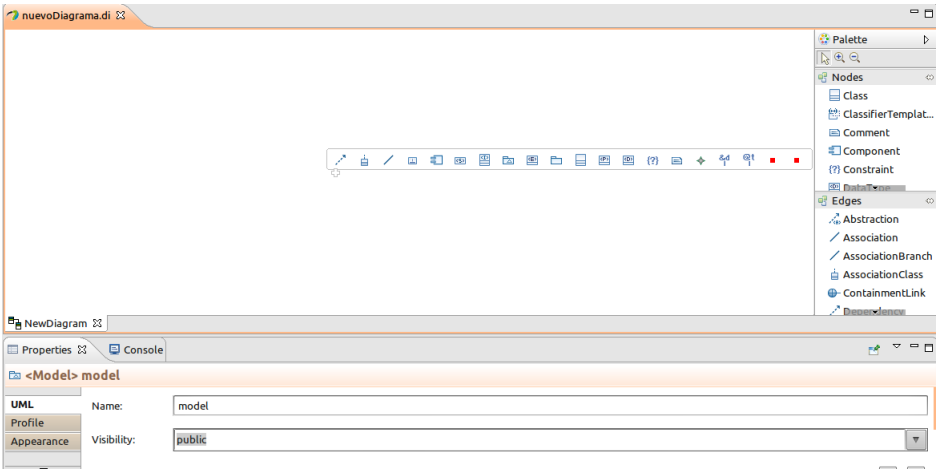

Figura D.19: New Diagram

D.6. Diagrama de clases de nuestro caso de estudio

En la Figura D.20 se muestra nuestro caso de estudio en el editor Papyrus.

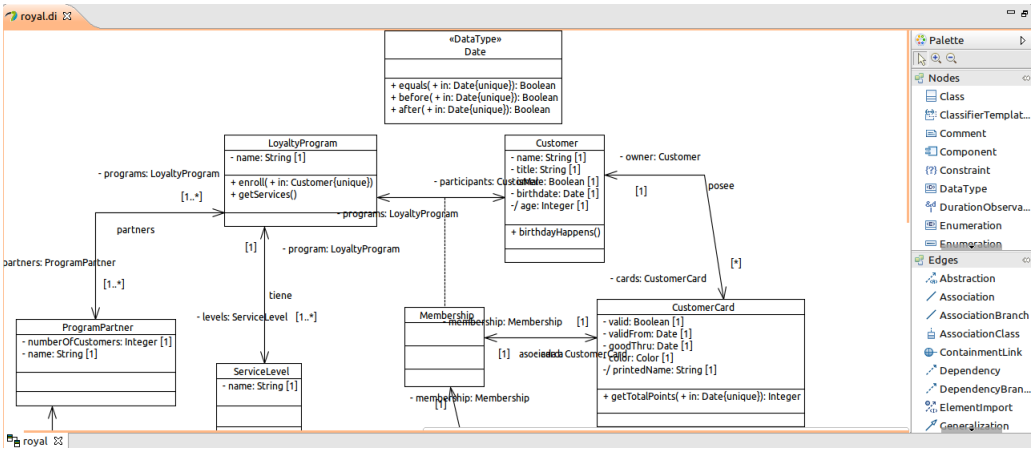

Figura D.20: Diagrama Royal and Loyal

D.7. Restricciones OCL

El siguiente paso es describir las restricciones OCL para nuestro modelo de clases. Para eso debemos agregar en nuestra carpeta anterior un nuevo archivo *.ocl*. Esto se hace con click derecho sobre la carpeta donde se encuentra nuestro modelo, y luego haciendo click en New -> File. En dicho archivo debemos crear nuestras restricciones.

D.8. Todo listo

Con todos estos pasos, ya estamos listos para utilizar la herramienta desarrollada.

Apéndice E

OCL File

E.1. Archivo de restricciones

A continuación se muestra el archivo completo de restricciones OCL:

```
 package Model

-- 2.2 Adding exME tra information
-- 2.2.1 Initial values and derivation rules

--Un LoyaltyAccount debería ser siempre inicializado con 0 puntos
context LoyaltyAccount::points : Integer
  init: 0

--Una tarjeta siempre debería ser válida al momento en que se emite
context CustomerCard::valid : Boolean
  init: true

--El atributo printedName de una CustomerCard está determinado
--basado en el nombre y título del owner de la tarjeta.
context CustomerCard::printedName : String
  derive: self.owner.title.concat(' ').concat(self.owner.name)
--
------ 2.2.2 Query operations
------ 2.2.3 Defining new attributes and operations
----
--context LoyaltyAccount
--  def: turnover : Real = self.transactions.amount->sum()
-- def: turnover : Real = (self.transactions->collect(a |a.amount))->asSet()->sum()
--
------ --Devuelve un conjunto de todos los nombres de los levels
-- Rompe en at ar.edu.unlp.info.ocl2jml.translator.OCLTranslator.createDerivationMethod(OCLTransla
--context LoyaltyProgram
--def: getServicesByLevel(levelName: String): Set(Service)
--= self.levels->select( name = levelName ).availableServices->asSet()

-- 2.3 Adding invariants
-- 2.3.1 Invariants on attributes
--La edad del cliente debe ser mayor o igual a 18
context Customer
  inv ofAge: self.age >= 18
--
---- 2.3.2 The type of the attribute is a class
```

```
----Chequea que validFrom sea antes que goodThru
context CustomerCard
  inv checkDates: self.validFrom.before(goodThru)
--
---- 2.3.3 Invariants on associated objects
----La edad del cliente dueño de la tarjeta debe ser mayor o igual a 18
context CustomerCard
inv ofAge: self.owner.age >= 18
--
---- 2.3.4 Using association classes
--context LoyaltyProgram
--  inv knownServiceLevel: self.levels->includesAll(Membership.serviceLevel)
----
------La tarjeta de un miembro debe estar incluida en las tarjetas de los miembros de esa membresí
context Membership
  inv correctCard: self.participants.cards->includes(self.card)
--
----Se define la operación getCurrentLevelName() a través del atributo nombre del servicio actual
--context Membership
--def : getCurrentLevelName() : String = self.serviceLevel.name
--
---- 2.3.5 Using Enumerations
---- Se crea una invariante que dice que si el nombre del serviceLevel de Membership es Silver, en
---- implica que el color de la tarjeta es silver. Similar para Gold
----context Membership
----  inv levelAndColor:
----self.serviceLevel.name = 'Silver' implies self.card.color = Color::silver
----     and
----    self.serviceLevel.name = 'Gold' implies self.card.color = Color::gold
--
------ 2.4 Working with collections of objects
------ 2.4.1 Using collections operations
----
------Un loyalty program debe ofrecer al menos un servicio a sus clientes
context LoyaltyProgram
  inv minServices: self.partners.deliveredServices->size() >= 1
----

------ El size de todos los programas del cliente, debe ser igual al size de las tarjetas válidas
context Customer
  inv sizesAgree: programs->size() = cards->select(valid)->size()
--

----El siguiente invariante dice que cuando el LoyaltyProgram no ofrece la posibilidad de ganar o
----puntos, los miembros del LoyaltyProgram no tienen LoyaltyAccounts; esto es que la colección de
----LoyaltyAccounts asociada a Memberships debe estar vacía.
--context LoyaltyProgram
--  inv noAccounts: self.partners.deliveredServices->forAll(
--       pointsEarned = 0 and pointsBurned = 0 )
--       implies Membership.account->isEmpty()
--
--
----2.4.2 Sets, Bags, OrderedSets, and Sequences
----Se guarda el número de clientes que participan en uno o mas loyalty programs
----ofrecidos por el programa
context ProgramPartner
```

```
  inv nrOfParticipants: numberOfCustomers = programs.participants->size()
----
------ 2.5 Adding preconditions and postconditions
------ 2.5.1 Simple preconditions and postconditions
------Si una está está vacía es porque la cantidad de puntos es 0
context LoyaltyAccount::isEmpty() : Boolean
  body: points = 0
  post: result = (points = 0)
--
--
------ 2.5.2 Previous values in postconditions
------Cuando pasa el cumpleaños de una persona su edad es la que tenía anteriormente, mas 1
context Customer::birthdayHappens():
  post: age = age@pre + 1
----
------Actualizar los puntos ganados va a ser los puntos que se tenían mas el amount
context Service::upgradePointsEarned(amount : Integer):
  post: calculatePoints() = calculatePoints@pre() + amount


--2.6 Taking Inheritance into Account
-- La suma de todos  los puntos ganados debe ser menor a 10000
context ProgramPartner
  inv totalPoints:
      self.deliveredServices.transactions.points->sum() < 10000

-- La suma de los puntos ganados debe ser menor a 10000
context ProgramPartner
inv totalPointsEarning:
  deliveredServices.transactions --all transactions
    ->select( oclIsTypeOf( Earning ) )  --select earning ones
     .points->sum() -- sum all points
    < 10000 -- sum smaller than 10,000
--
---- 2.8 Let expressions
----El siguiente ejemplo muestra que las fechas validFrom o goodThru de la tarjeta de un cliente
---- deben ajustarse cuando la tarjeta es invalidada. Una variable extra llamada correctDate se
---- define para indicar si la fecha actual está entre validFrom y goodThru dates
context CustomerCard::isCorrect(d : Date) : Boolean
  body: let correctDate : Boolean =
        self.validFrom.before(d) and
        self.goodThru.after(d)
      in
        if valid then
          correctDate = false
        else
         correctDate = true
        endif

--
---- 3.3 Completing class diagrams
---- 3.3.1 Derivation rules
----El valor del elemento derivado usedServices es definido para ser todos los servicios que son g
----sobre la cuenta
--context LoyaltyAccount::usedServices() : Set(Services)
--derive: self.transactions.service->asSet()
```

```
--
---- 3.3.2 Initial values
-- El valor inicial de las transacciones de LoyaltyAccount, debe ser un conjunto vacío
-- Seguir el código con esta, así vemos de implementar el archivo de log
context LoyaltyAccount::transactions : Set(Transaction)
  init: Set{}

-- 3.3.3 Body of query operations
--Devuelve el nombre del dueño de la tarjeta
--context LoyaltyAccount::getCustomerName() : String
--  body: Membership.card.owner.name

-- 3.3.4 Invariants
--Cada tarjeta tiene solamente un owner
context LoyaltyAccount
  inv oneOwner: transactions.card.owner->asSet()->size() = 1

-- 3.3.5 Preconditions and postconditions
--Inscribir a una persona requiere que su nombre sea distinto de vacío, y que no esté en el progra
-- y Asegura que luego de ejecutarse el método, estará en el programa
context LoyaltyProgram::enroll(c : Customer):
  pre: c.name <> ''
  pre: not participants->includes(c)
  post: participants = participants@pre->including(c)
--
--context LoyaltyProgram
--def: isSaving : Boolean =
--partners.deliveredServices->forAll(pointsEarned = 0)

-- 6.1 A combined model
-- 6.1.1 The context of an OCL expression
-- los puntos a ingresar en el método earn, deben ser mayores a cero
context LoyaltyAccount::earn(points : Integer):
  pre: points > 0

-- 6.2 Classes and another types
-- 6.2.2 Definitions of attributes or operations
--Devuelve la primer letra del nombre de la persona
context Customer
  def: initial : String = self.name.substring(1, 1)

--context CustomerCard
--  def: getTotalPoints(d : Date) : Integer =
--        self.transactions->select(date.after(d)).points->sum()

--6.3.2 Initial values

--6.4.2 Body of Query Operations
--context CustomerCard::getTransactions(from : Date, until: Date )
--: Set(Transaction)
--body: transactions->select( date.isAfter( from ) and
--date.isBefore( until ) )

--8.2 Associations and Aggregations
--context CustomerCard
--inv: self.owner.dateOfBirth.isBefore( Date::now )
```

```
context CustomerCard
inv: self.owner.programs->size() > 0

--context Membership
--inv: programs.levels->includes( currentLevel )


--9.1.3 Collections Operations
context LoyaltyProgram
inv: self.participants->size() < 10000

--9.2 Operations on Collection Types
context ServiceLevel
inv: self.program.partners->includesAll(self.availableServices.partner)

-- 9.3 Loop operations
--9.3.1 Iterator Variables
--context LoyaltyProgram
--inv: self.Membership.account->isUnique( acc | acc.number )
--context LoyaltyProgram
--inv: self.Membership.account->isUnique( acc: LoyaltyAccount | acc.number )

--9.3.3 The sortedBy Operation
--context LoyaltyProgram
--def: sortedAccounts : Sequence(LoyaltyAccount) =
--self.Membership.account->sortedBy( number )

--9.3.4 The select Operation
context CustomerCard
inv: self.transactions->select( points > 100 )->notEmpty()

--9.3.5 The reject Operation
--context Customer
--inv: self.membership.account->select( points > 0 )

--context Customer
--inv: Membership.account->reject( not (points > 0) )

--9.3.6 The any Operation
--context CustomerCard
--inv: self.Membership.account->any( number < 10000 )

-- 9.3.7 The forAll operation
--todos los participantes deben tener como mucho 70 años inclusive
context LoyaltyProgram
  inv: self.participants->forAll(age <= 70)

--No puede haber en el programa dos participantes con le mismo nombre
context LoyaltyProgram
  inv: self.participants->forAll(c1, c2 | c1 <> c2 implies c1.name <> c2.name)

-- 9.3.8 The exists operation
context LoyaltyAccount
  inv: points > 0 implies self.transactions->exists(t | t.points > 0)
```

```
-- 9.3.10 The collect operation
context LoyaltyAccount
  inv: self.transactions->collect(points)->exists(p : Integer | p < 500)

--9.3.11 Shorthand Notation for collect
context LoyaltyAccount
inv: transactions.points->exists(p : Integer | p = 500 )

--10.1.2 The result Keyword
--context Transaction::getProgram() : LoyaltyProgram
--post: result = self.card.Membership.programs
endpackage
```

Referencias bibliográficas

[1] *CVC3*. http://www.cs.nyu.edu/acsys/cvc3/.

[2] *DTD Tutorial*. http://www.w3schools.com/dtd/default.asp.

[3] *Eclipse OCL*. http://help.eclipse.org/indigo/topic/org.eclipse.ocl.doc/help/index.html.

[4] *Eclipse Plugin Development Tutorial*. http://www.vogella.de/articles/EclipsePlugIn/article.html.

[5] *The Extensible Stylesheet Language Family (XSL)*. http://www.w3.org/Style/XSL/.

[6] *Guava: Google Core Libraries for Java 1.5+*. http://code.google.com/p/guava-libraries/.

[7] *The Java Modeling language*.

[8] *Object Constraint Language Environment 2.0*. http://lci.cs.ubbcluj.ro/ocle/index.htm.

[9] *XSD (XML Schema Definition)*. http://searchsoa.techtarget.com/definition/XSD.

[10] *Eclipse Modeling Framework Project (EMF)*, 2001. http://www.eclipse.org/modeling/emf/.

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

[12] *Lecture 3: Java Modeling Language and Extended Static Checking*, 2008.

[13] *Object Constraint Language*. Diciembre 2010.

[14] *Yices: An SMT Solver*, Diciembre 2010. http://yices.csl.sri.com/.

[15] *The Coq Proof Assistant*, 2011. http://coq.inria.fr/.

[16] *The Satisfiability Modulo Theories Library*, Junio 2011. http://smtlib.org/.

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

[18] Akehurst, David y Octavian Patrascoiu: *OCL 2.0 - Implementing the Standard for Multiple Metamodels*. 2003.

[19] 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.

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

[21] 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.

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

[23] Breunesse, Cees Bart y Erik Poll: *Verifying JML specifications with model fields*. 2003.

[24] 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`.

[25] 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.

[26] 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`.

[27] Cok, David R. y Joseph R. Kiniry: *ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2*. En *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, Lecture Notes in Computer Science, páginas 108–128. Springer, Marzo 2004.

[28] Cok, David R., Joseph R. Kiniry y Dermot Cochran: *ESC/Java2 Implementation Notes*, Octubre 2008. `http://kind.ucd.ie/products/opensource/ESCJava2/ESCTools/docs/Escjava2-ImplementationNotes.pdf`.

[29] Detlefs, David, Greg Nelson y James B. Saxe: *Simplify: A theorem prover for program checking*. Informe técnico HPL-2003-148, HP Labs, Julio 2003.

[30] 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.

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

[32] Gamma, Erich, Richard Helm, Ralph Johnson y John M. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, spanishfirst edición, Noviembre 1994.

[33] Garcia, Miguel: *How to process OCL Abstract Syntax Trees, Eclipse Technical Article*, 2007. `http://www.eclipse.org/articles/article.php?file=Article-HowToProcessOCLAbstractSyntaxTrees/index.html`.

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

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

[36] 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`.

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

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

[39] Leavens, Gary T., Joseph R. Kiniry y Erik Poll: *A JML Tutorial*, 2007.

[40] Leavens, Gary T., Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok y Joseph Kiniry: *JML Reference Manual*, Julio 2011. `http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/`.

[41] Leino, K. Rustan M., Greg Nelson y James B. Saxe: *ESC/Java User's Manual*. Informe técnico, Compaq Systems Research Center, Octubre 2000.

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

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

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

[45] OMG: *Model Driven Architecture (MDA)*, Julio 2001.

[46] OMG: *MDA Guide Version 1.0.1*, Junio 2003.

[47] OMG: *MOF Model to Text Transformation Language. Request for Proposal*, Abril 2004. `http://www.omg.org/cgi-bin/doc?ad/04-04-07.pdf`.

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

[49] Pons, Claudia, Roxana Giandini y Gabriela Perez: *Desarrollo de software dirigido por modelos*. Editorial de la UNLP, 2010.

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

[51] SAnToS Laboratory, Kansas State University: *Sireum/Kiasan for Java*, Noviembre 2008.

[52] Steinberg, David, Frank Budinsky, Marcelo Paternostro y Ed Merks: *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, spanish2nd edición, Diciembre 2008, ISBN 0321331885.

[53] Warmer, Jos y Anneke Kleppe: *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, spanish2 edición, 2003, ISBN 0321179366.

Glosario

API Una API es un conjunto de reglas y especificaciones que sirven de interface entre diferentes programas y facilita su interacción. 64, 66

CIM Un Computation Independent Model (CIM), llamado modelo del dominio, es una vista del sistema desde un punto de vista independiente de la computación. . 55

Coq Coq es un sistema formal de gestión de pruebas. Provee un lenguaje formal para escribir definiciones matemáticas, algoritmos ejecutables y teoremas junto con un entorno de desarrollo semi-interactivo pruebas chequeadas por máquinas. Las aplicaciones típicas incluyen la formalización de la semántica de los lenguajes de programación [15]. 52

CVC3 CVC3 es un demostrador de teoremas automáticos de problemas SMT. Puede ser utilizado para probar la validez (o, dualmente, la satisfacibilidad) de fórmulas de primer orden en un gran número de teorías lógica y de su combinación. CVC3 es el último de una serie de populares demostradores SMT, que se originó en la Universidad de Stanford con el sistema de SVC. En particular, se basa en el código base de CVC Lite, su predecesor mas reciente [1]. 52

DBC DBC término acuñado por Bertrand Meyer en 1992 [42]. Describe un enfoque para el desarrollo de software. La idea es que se deben definir especificaciones formales, precisas y verificables, utilizando precondiciones, postcondiciones e invariantes. Estas especificaciones son llamadas "contratos". 2

DTD El propósito de un Document Type Definition (DTD) es definir la construcción de bloques legítimos de un documento XML [2]. 151

EMF El proyecto EMF es un framework de modelado y facilidades de generación de código para la creación de herramientas de instalación y otras aplicacines basadas en un modelo de datos estructurados [52]. Desde una especificación de modelo descripto en XMI, EMF ofrece herramientas y soporte en tiempo de ejecución para producir un conjunto de clases Java para el modelo, junto con un conjunto de clases adaptadoras que permiten la visualización y la edición basada en comandos, del modelo, y un editor básico [49]. 64, 65

Frame condition Frame condition es una condición que especifica las propiedades que una operación puede modificar. 36

Guava Guava es un proyecto desarrollado por Google que contiene varias librerías principales utilizadas en proyectos Java: colecciones, manejo de cachés, soporte de primitivas, librerías de concurrencia, anotaciones comunes, procesamiento de strings, entrada/salida, etc. [6]. 87

HTML HTML son las siglas de HyperText Markup Language (lenguaje de marcado de hipertexto), es el lenguaje predominante para la elaboración de páginas web. Es usado para describir la estructura y el contenido en forma de texto, así como para complementar el texto con objetos tales como imágenes. HTML se escribe en forma de "etiquetas", rodeadas por corchetes angulares (<,>). HTML también puede describir, hasta un cierto punto, la apariencia de un documento, y puede incluir un script (por ejemplo JavaScript), el cual puede afectar el comportamiento de navegadores web y otros procesadores de HTML. 44, 62

JFace JFace es un conjunto de widgets para realizar interfaces de usuario, construido sobre SWT. Fue desarrollado por IBM para facilitar la construcción del entorno de desarrollo Eclipse, pero su uso no está limitado a éste. JFace proporciona una serie de construcciones muy frecuentes a la hora de desarrollar interfaces gráficas de usuario, tales como cuadros de diálogo, evitando al programador la tediosa tarea de lidiar manualmente con los widgets de SWT.. 64

JUnit JUnit es un framework de testeo de unidad para el lenguaje Java. JUnit ha sido importante en el desarrollo de Test driven development (TDD) y forma parte de una familia de frameworks de testeo de conocidos como xUnit que se originó con SUnit. 44

OMG OMG es un consorcio establecido en 1989, originalmente dedicado a establecer estándares para sistemas orientados a objetos distribuidos, pero que en la actualidad se dedica al modelado y estándares basados en modelos. 35, 41

OpenJDK Open Java Development Kit (OpenJDK) es una implementación del lenguaje Java libre y open source. 52

OSGi OSGI son las siglas de Open Services Gateway Initiative, es un sistema de módulos y plataforma de servicios para el lenguaje de programación Java que implementa un completo y dinámico modelo de componentes. Las aplicaciones o componentes pueden instalarse remotamente, iniciarlas, terminarlas, actualizarlas y desinstalarlas sin requerir un reinicio; el manejo de packages/classes Java se especifica en gran detalle. El manejo de la aplicación del ciclo de vida (empezar, terminar, etc.) se hace vía APIs que permiten bajar las políticas de manejo remotamente. El servicio de registro, permite detectar la adición de nuevos servicios, o la eliminación de servicios y adaptarlos. 64

PIM Un PIM es un modelo con un alto nivel de abstracción que es independiente de cualquier tecnología o lenguaje de implementación. 55

Plataforma En MDA el término plataforma se refiere a los detalles tecnológicos y de implementación que son irrelevantes para la funcionalidad fundamental de un sistema de software [45]. 54

PSM Un PSM es un modelo que es específico a una plataforma. 55

QVT En MDD, QVT es el lenguaje estándar para describir transformaciones de modelos definido por el OMG. La especificación del lenguaje QVT tiene una naturaleza híbrida, declarativa/imperativa, con la parte declarativa dividida en una arquitectura de dos niveles [49]. 61

Simplify Simplify es un demostrador automático de teoremas. Utiliza demostración por refutación, i.e. intenta demostrar la validez de una fórmula determinada demostrando que la negación de la fórmula es insatisfacible. Simplify es incompleto, puede determinar que una fórmula es inválida cuando en realidad es válida; sin embargo es consistente, así que para cualquier fórmula que se determina como válida, se garantiza su validez. Simplify es utilizado como motor en ESC/Java y ESC/Java2 [29]. 45

SMT Satisfiability Modulo Theories (SMT) es un área de deducción automática que estudia métodos para chequear la satisfacibilidad de fórmulas de primer orden con respecto a determinadas teorías lógicas T. Esto difiere de la deducción automática general en que la teoría T de donde necesita no ser finita o axioma de primer orden, y los métodos de inferencia especializados son usados para cada teoría. Al ser la teoría específica y al restringir el lenguaje a ciertas clases de fórmulas (como, por lo general pero no exclusivamente, cuantificador sin fórmulas), estos métodos especializados pueden ser implementados en solucionadores que son más eficientes en la práctica que los demostradores de teoremas de propósito general. [16]. 52, 149, 151

SWT Standard Widget Toolkit (SWT), es un conjunto de componentes para construir interfaces gráficas en Java, (widgets) desarrollados por el proyecto Eclipse. Recupera la idea original de la biblioteca AWT de utilizar componentes nativos, con lo que adopta un estilo más consistente en todas las plataformas, pero evita caer en las limitaciones de ésta. La biblioteca

Swing, por otro lado, está codificada enteramente en Java. La interfaz del workbench de eclipse también depende de una capa intermedia de interfaz gráfica de usuario (GUI) llamada JFace que simplifica la construcción de aplicaciones basadas en SWT. 64

Verification condition Una condición de verificación (verification condition) es una fórmula de lógica de primer orden (no contiene construcciones de programación), que es válida si y sólo si el programa está libre de la clase de errores analizados. Un verificador transforma una terna de Hoare en una condición de verificación, si la Verification condition (condición de verificación) (VC) es válida entonces la terna es válida, sino es válida el verificador puede encontrar un contraejemplo. 45

Why Es una plataforma de verificación de software. 52

XMI XMI es un estándar del OMG para intercambio de modelos basado en XML. 64, 65, 149

XSD XSD especifica como describir formalmente los elementos en un documento XML. Esta descripción puede ser usada para verificar que cada ítem del contenido en un documento, se adhiere a la descripción del elemento en el cual se encuentra el contenido. Un esquema es una representación abstracta de las características de un objeto y las relaciones con otros objetos. Un esquema XML representa la interrelación entre atributos y elementos de un objeto XML (por ejemplo, un documento o una porción de documento). XSL tiene ventajas con respecto a los lenguajes XML, como el DTD o Simple Object XML (SOX) [9]. 64

XSL Extensible Stylesheet Language (XSL) es una familia de recomendaciones para definir la transformación y presentación de documentos XML. Consiste de tres partes. La primera XSL Transformations (XSLT): un lenguaje para transformar XML. La segunda XML Path Language (XPath): un lenguaje usado por XSLT (y otros lenguajes) para acceder o referenciar partes de un documento XML. Y la tercera XSL Formatting Objects (XSL-FO): un vocabulario XML para especificar semánticas de formato. [5]. 151

Yices Yices es un solucionador SMT eficiente eficiente que decide la satisfacibilidad de fórmulas arbitrarias que contienen símbolos no interpretados de funciones con igualdad, aritmética lineal real y entera, los tipos escalares, tipos de datos recursivos, tuplas, registros, vectores de tamaño fijo, cuantificadores, y lambda expresiones [14]. 52

Siglas

API Application programming interface. 62, 63, 66, 120, 149

AST Abstract Syntax Tree. 66, 121

CIM Computation Independent Model. 149

CMOF Complete MOF. 58

DBC Design by Contract. 20, 34, 149

DOM Document Object Model. 64

DTD Document Type Definition. 149

EMF Eclipse Modeling Framework. 2, 64–66, 149

EMOF Essential MOF. 58

ESC/Java2 Extended static checker for Java version 2. 45

IDE Integrated Development Environment. 52

JDT Java Development Tooling. 65

JML Java Modeling Language. 1, 2, 20–28, 30, 31, 33–42, 44, 45, 48, 49, 52, 77, 78, 80, 82, 87, 88, 95, 97, 103–105

MDA Model driven architecture. 54, 150

MDD Model driven development. 2, 54, 57, 150

MOF Meta Object Facility. 2, 4, 54, 57, 58, 60, 61, 67, 107

Mof2Text MOF Model to Text Transformation Language. 61

OCL Object Constraint Language. 1–7, 9–11, 13–16, 18, 19, 35–42, 60, 61, 66, 77, 78, 80, 82, 87, 88, 103, 104, 107–111, 113, 120, 122, 139

OCLE Object Constraint Language Environment. 44

OMG Object Management Group. 4, 54, 56–58, 60, 61, 66, 67, 150

OpenJDK Open Java Development Kit. 150

PIM Platform Independent Model. 54, 150

PSM Platform Specific Model. 54, 150

PVS Specification and Verification System. 104

QVT Query/View/Transformation. 67, 150

RFP Request for Proposal. 60

SMOF Semantic MOF. 58

SMT Satisfiability Modulo Theories. 150

SOX Simple Object XML. 151

SWT Standard Widget Toolkit. 150

TDD Test driven development. 150

UML Unified Modeling Language. 1, 2, 4, 9, 18, 19, 35, 41, 54, 62, 65, 66, 77, 78, 80, 87, 103, 104, 113

VC Verification condition (condición de verificación). 151

XMI XML Metadata Interchange. 54, 78, 151

XML Extensible Markup Language. 62, 64, 149, 151

XPath XML Path Language. 151

XSD XML Schema Definition. 64, 151

XSL Extensible Stylesheet Language. 151

XSL-FO XSL Formatting Objects. 151

XSLT XSL Transformations. 151