HERRAMIENTA PARA AUTOMATIZAR LA TRANSFORMACION UML/OCL A OBJECT-Z.

Autor

Becker Valeria

Trabajo de Grado

Licenciatura en Informática
Universidad Nacional de La Plata
2006

Directora: Dra. Claudia Pons

TABLA DE CONTENIDOS

AGRADECIMIENTOS

Aprovecho este espacio para demostrar mi profundo agradecimiento a todas aquellas personas e instituciones que de alguna manera, forman parte importante de este logro, tan significativo para mí. Considero que este trabajo es fruto de un gran esfuerzo y que todas las personas con las que participé y conviví durante estos años, ayudaron a la culminación y el alcance de las metas propuestas, tanto académicas como personales.

En primer lugar quiero especialmente agradecer a Claudia Pons, directora de este trabajo, por su gran apoyo a lo largo de este tiempo, tanto en el ámbito académico como en el personal. Agradezco mucho su cuidadosa guía, su entusiasmo y la dedicación que siempre demostró durante la realización de este trabajo, pero sobre todo, le quiero decir gracias por su gran calidad humana y por todo el tiempo dedicado.

También agradezco el apoyo brindado por el Laboratorio de Investigación y Formación en Informática Avanzada, especialmente a al grupo "Eclipse" que me dio la oportunidad de investigar sobre los temas referentes a este trabajo, aprendiendo muchísimo junto a ellos. Aprecio mucho el motivador ambiente de trabajo y la libertad que me brindo para crecer día a día.

A la UNLP por brindarme la posibilidad de hacer mi carrera de grado en una de las mas prestigiosas Universidades de Argentina.

En el plano personal, me gustaría agradecer a mis padres Carlos y Camby, por su ejemplo, su constante apoyo, y quiero que sepan que estos logros son también fruto de toda su enseñanza. A mis hermanos Carina, Daniela y Pablo quisiera agradecerles el soporte que siempre me han brindado en todos los aspectos y su compañía incondicional. A mi abuela Ilda, quien ha sido mi ejemplo más importante a seguir, que siempre confió en mí y que tanto espero este momento.

También me gustaría dar las gracias a todos mis familiares, tíos, primos, por su gran afecto.

No me puedo olvidar de todas las personas que he conocido a lo largo de este tiempo, empezando por los compañeros de la UNLP, y todas las personas que conocí en el camino, los cuales me es imposible nombrarlos a todos ya que son demasiados. Agradezco todos los amigos que he cosechado en esta etapa de mi vida, los cuales son muy importantes para mí.

Gracias…

Capítulo 1

INTRODUCCIÓN

En el proceso de construcción de software, el análisis y diseño son una tarea muy importante. UML (Unified Modeling Language, [OMG]) ha sido desarrollado para modelar sistemas Orientado a Objetos integrando lenguajes predecesores tales como la notación de Booch, OMT, etc.

Este lenguaje ha sido aceptado como un estándar por OMG (Object Management Group) en el año 1997 [OMG].

Los principales diagramas provistos por UML son: diagramas de casos de usos, diagramas de clases, diagramas de estados, diagramas de secuencias.

UML también provee un lenguaje textual, OCL (Object Constraint Language), fácil de leer y de escribir, que permite especificar características adicionales sobre los modelos en una forma similar a lógica de predicados. OCL es un lenguaje semi formal, su sintaxis está precisamente definida pero su semántica aún presenta ambigüedad, imprecisión e inconsistencia. Las expresiones OCL no tienen efectos laterales, es decir que su evaluación no puede alterar el estado del sistema correspondiente. Su evaluación solamente retorna un valor. En este trabajo presentamos una traducción de UML/OCL en lógica de predicados de primer orden. El objetivo es verificar propiedades de los diagramas UML y verificar la validez de las expresiones OCL que acompañan a dichos diagramas. Para ello se define una sintaxis y semántica para OCL. El beneficio de esta formalización es permitir la evaluación de expresiones OCL en un modelo UML. Provee a la ingeniería de software los siguientes beneficios:

- Una definición precisa de OCL evitando ambigüedades. Algunos agregados para mejorar la ortogonalidad del lenguaje.

- Se desarrolla una base para herramientas que soporten análisis, y validación de modelos UML con expresiones OCL.

- Estos aspectos contribuyen al objetivo de mejorar la calidad global de los sistemas de software.

Capítulo 2

SINTAXIS DE UN SUBCONJUNTO DE EXPRESIONES OCL

Antes de dar la gramática se verá que significa cada una de las palabras claves de la sintaxis OCL, adjuntando ejemplos para familiarizarse con las expresiones escritas en OCL. Para mayor información y detalle, ver [OMG-OCL]. El lector familiarizado con OCL puede saltear este capítulo.

Self

Cada expresión OCL es escrita en el contexto de una instancia de un tipo específico. En una expresión OCL, la palabra reservada *self* se usa para referirse a la instancia contextual.

Por ejemplo:

context Banco **inv**:

self.nroEmpleados > 72

Es este caso, *self* se refiere a una instancia de la clase Ejemplo. El contexto de una expresión OCL dentro de un modelo UML puede ser especificado a través de la declaración *context* al comienzo de la expresión.

Invariantes

Una expresión OCL puede se parte de un invariante. Una expresión OCL es un invariante de tipo y debe ser verdadero para todas las instancias de ese tipo en cualquier momento. Notar que las expresiones que expresan invariantes son expresiones lógicas, es decir que su evaluación retornan un valor de verdad.

Por ejemplo:

La siguiente expresión específica que el número de empleados deber ser siempre mayor que 72, en el contexto Banco. Este invariante vale para toda instancia de tipo Banco.

context Banco **inv *cantEmpleados*:**

self.nroEmpleados > 72

El nombre ***cantEmpleados***, podría omitirse, se utiliza para poder referenciar al invariante en otra expresión. En la mayoría de los casos, se utiliza la palabra *self*. Otra opción equivalente sería:

```
context b:Banco inv cantEmpleados:
    b.nroEmpleados > 72
```

Precondiciones y Poscondiciones

Una expresión OCL puede se parte de una Precondición o Poscondición, asociada a un Método u Operación. La declaración usa la palabra *context*, seguida del tipo y la declaración de la operación. Luego continúan los términos 'pre:' y 'post:' antes de las Precondiciones y Poscondiciones.

```
context nombreDeTipo::NombreDeOperacion(param1:Tipo1,...): TipoDeRetorno
  pre : param1 > ...
  post: result = ...
```

El nombre *self* puede ser usado en la expresión refiriéndose al objeto receptor la operación. La palabra reservada *result* denota el resultado de la operación, si es que hay uno. Los nombres de los parámetros (*param1*) pueden ser usadas en la expresión OCL.

Por ejemplo:

```
context Persona::edad(): Integer
  post: result = 45
```

El nombre de la precondición o poscondición, opcionalmente, puede ser escrito después de la palabra *pre* o *post*, permitiendo ser referenciado por el nombre.

Por ejemplo: El nombre de la precondición es *nombrePre* y el de la poscondición *nombrePos*.

```
context nombreDeTipo::NombreDeOperacion(param1 : Tipo1, ... ): TipoDeRetorno
    pre nombrePre : param1 > ...
    post nombrePos: result = ...
```

Valores previos en Poscondiciones

En una poscondición, la expresión puede referirse a dos conjuntos de valores:

- El valor de una propiedad al comenzar la operación o método.
- El valor de la propiedad luego de completar la operación o método.

Para referirnos al valor de una propiedad al comenzar la operación o método, se usa la palabra "@pre"

Ejemplo:

```
context Cuenta::depositar(cant:Integer)
  pre: cant>0
  post: saldo=self.saldo@pre + cant
```

Notar que "saldo@**pre**" se refiere al valor de la propiedad "saldo" de la instancia Cuenta, sobre la cual se ejecuta la operación "depositar", al comienzo de la operación.

Paquete

Para especificar explícitamente a cuál paquete, un invariante, precondición o poscondición pertenecen, estos pueden ser encerrados entre las sentencias *'package'* y *'endpackage'*. Las declaraciones de paquetes tienen la siguiente sintaxis:

```
package Package::SubPackage
  context X inv:
    ... Algún invariante...
  context X:: NombreDeOperacion (..)
    pre: ... alguna precondición ...
endpackage
```

En un archivo OCL puede haber varias definiciones de paquetes, permitiendo a todos los invariantes, precondiciones, y poscondiciones estar escritos y almacenados en el mismo archivo.

Tipos y valores básicos

En OCL, varios tipos básicos son predefinidos e independientes de cualquier modelo de objetos.

Ejemplo:

- **Boolean**: true, false
- **Integer**: 1, -5, 2, 34, 26524, ...
- **Real**: 1.5, 3.14, ...
- **String**: 'esto es un String...'

OCL define un número de operaciones sobre los tipos predefinidos.

Ejemplo:

- **Integer:** *, +, -, /, abs()
- **Real:** *, +, -, /, floor()
- **Boolean:** and, or, xor, not, implies, if-then-else
- **String:** toUpper(), concat()

Expresiones Let

La expresión *let* permite definir un atributo o una operación para ser usada más de una vez. Por ejemplo: Se define un atributo que dice si el cliente tiene al menos una cuenta, para luego ser usado.

```
context Cliente inv:
    let tieneCuenta : Boolean = self.cuentas-> notEmpty()
        in self.edad()<21 or self.tieneCuenta = true
```

Una expresión *let* puede ser incluida en un invariante o precondición o poscondición. 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 de la Clase puede ser usado. Veamos un ejemplo:

context Cliente **def:**

let tieneCuenta : Boolean = self.cuentas-> notEmpty()= true

Colecciones

El tipo *Collection* es predefinido en OCL, y tiene operaciones predefinidas para manipular las colecciones. Es un tipo abstracto, y los subtipos son colecciones concretas. OCL distingue tres tipos de colecciones: *Set, Sequence, y Bag*. Un tipo *Set* es el conjunto matemático, no contienen elementos repetidos. Un tipo *Bag* es como un conjunto, pero 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 delante de las llaves.

Set {1, 2, 5, 88}
Set { 'apple' , 'orange', 'strawberry' }
Sequence {1, 3, 45, 2, 3}
Sequence {'ape', 'nut'}
Bag {1, 3, 4, 3, 5}

Para definir una secuencia de valores enteros consecutivos, se utiliza dos puntos seguidos. Es decir, *entero_exp1 y entero_exp2*, separado por '..' denota todos los enteros entre los valores *entero_exp1 y entero_exp2* incluyendo los extremos también:

Sequence{ 1..(6 + 4) }
Sequence{ 1..10 }

Son equivalentes a

Sequence{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

Operaciones de Colecciones

OCL define varias operaciones sobre los tipos Colecciones. Veamos algunas de ellas

Select y Reject

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

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

El resultado de la operación **select**, son todos los elemento de la colección, para los cuales la expresión-lógica-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:

colección->select(c | expresión-lógica-con-c)

Y se puede abreviar mediante:

colección->select(expresión-lógica)

Por ejemplo, las tres formas de escribir un **select** son las siguientes:

context Banco **inv**:

self.cuentas->select(c: Cuenta| c.nroCuenta > 1250)->notEmpty() = true

context Banco **inv**:

self.cuentas->select(c| c.nroCuenta > 1250)->notEmpty() = true

context Banco **inv**:

self.cuentas->select(nroCuenta > 1250)->notEmpty() = true

El atributo self.cuentas es de tipo Set(Cuenta). El **select** toma cada cuenta cuyo número de cuenta sea mayor a 1250.

La operación **reject** puede expresarse como un **select** con la expresión booleana negada. Es decir, que las dos siguientes expresiones son idénticas:

colección-> reject (c:Tipo | expresión-lógica-con-c)

colección-> 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 la cual contienen objetos diferentes a la colección original. La sintaxis del **collect** es de alguna de estas formas:

colección->collect(e : Tipo | expresión-con-e)

colección->collect(e | expresión-con-e)

Por ejemplo: Especificar la colección con los números de cuentas del Ejemplo.

self.cuentas ->collect(c:Cuenta| c.nroCuenta) self.cuentas ->collect(c| c.nroCuenta)

El resultado del **collect** es un Bag y no un Set. La colección resultante del **collect** tiene el mismo tamaño que la original.

Otra notación para el **collect**, que hace a la expresión OCL más legible, es la siguiente:

self.cuentas.nroCuenta

En general, cuando se aplica 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)

Y si la propiedad es parametrizada, queda de la siguiente forma:

colección.nombrePropiedad(par1, par2, ...)

colección->collect(nombrePropiedad(par1, par2, ...)

ForAll - Exists

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

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

El resultado es verdadero si la expresión-lógica-con-e es verdadera para todos los elementos de la colección. Si la expresión-lógica-con-e es falsa para algún elemento, entonces la expresión completa evalúa falso. Por ejemplo:

context Banco
inv: self.cuentas->forAll(nroCuenta >2000)
inv: self.cuentas->forAll(c | c.nroCuenta >2000)
inv: self.cuentas->forAll(c:Cuenta | c.nroCuenta >2000)

Los invariantes evalúan verdadero si todas las cuentas tienen un número de cuenta mayor que 2000.

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 Banco
inv: self.cuentas->forAll(c1, c2 | c1 <> c2 implies c1.nroCuenta <> c2.nroCuenta) = true

context Banco
inv: self.cuentas->forAll(c1, c2:Cuenta | c1 <> c2 implies c1.nroCuenta <> c2.nroCuenta) = true

Esta expresión evalúa verdadero si todos los números de cuentas son diferentes para cuentas diferentes. Sintácticamente es equivalente a:

```
context Banco inv:
    self.cuentas->forAll( c1 | self.cuentas->forAll( c2 |
    c1 <> c2 implies c1.nroCuenta <> c2.nroCuenta)) = true
```

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

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

El resultado es verdadero si la expresión-lógica-con-e es verdadera para uno o más elementos de la colección. Si la expresión-lógica-con-e es falsa para todos los elementos, entonces la expresión completa evalúa falso. Por ejemplo:

```
context Banco
    inv: self.cuentas-> exists ( nroCuenta = 2000 ) = true
    inv: self.cuentas-> exists (c | c.nroCuenta = 2000 ) = true
    inv: self.cuentas-> exists (c:Cuenta | c.nroCuenta = 2000) = true
```

Los invariantes evalúan verdadero si alguna cuenta tiene un número de cuenta igual a 2000.

Size

Retorna el número de elementos de la colección.

```
colección->size() : Integer
```

Includes - Excludes

Retorna verdadero si el objeto está en la colección, falso caso contrario

```
colección ->includes(objeto : OclAny) : Boolean
```

Retorna verdadero si el objeto no está en la colección, falso caso contrario

colección->excludes(objeto : OclAny) : Boolean

IncludesAll - ExcludesAll

Retorna verdadero si todos los objetos de la colección c2 están en la colección, falso caso contrario.

colección->includesAll(c2 : Collection(T)) : Boolean

Retorna verdadero si ninguno de los objetos de la colección c2 está en la colección, falso caso contrario.

colección->excludesAll(c2 : Collection (T)) : Boolean

IsEmpty – NotEmpty

Retorna verdadero si la colección está vacía, falso caso contrario

colección->isEmpty() : Boolean

Retorna verdadero si la colección no está vacía, falso caso contrario

colección->notEmpty() : Boolean

Any

Retorna algún elemento de la colección para el cual la *expr* evalúa verdadera. Si hay más de un elemento para el cual *expr* es verdadera, uno de ellos es retornado. Si no existe el elemento, el resultado es de la operación es indefinido (Undefined).

colección->any(expr : OclExpression) : T

One

Retorna verdadero si *expr* evalúa verdadero para exactamente un elemento en la *colección*; falso caso contrario.

colección->one(expr : OclExpression) : Boolean

Gramática para expresiones OCL

Se presenta la gramática para las expresiones OCL, usando una notación BNF con las siguientes convenciones:

Los símbolos terminales son cadenas de caracteres entre comillas dobles (").

Las demás cadenas de caracteres son símbolos no terminales.

Los símbolos alternativos son separados por "|".

Un símbolo opcional está escrito entre corchetes ("[" y "]").

Los paréntesis agrupan elementos de la gramática.

Símbolo inicial: **oclFile**

oclFile:= package^{+}

package := "package" packageName **(**constraint**)* "endpackage"**.

packageName := pathName

constraint :=

"context" operationContext contextStereotype^{+} **|**

"context" classifierContext contextDefinition^{+}

operationContext:= name **"::"** signatureOp

signatureOp:= operationName **"(" [**formalParameter**] ")" [":"** returnType**]**

contextStereotype = stereotype name **":"** oclExpression

stereotype := "pre" | "post"

classifierContext := name **":"** name **|** name

contextDefinition :=

(“def” [name] “:”) letExpression^{+} |

(“inv” [name]) “:” oclExpression)

oclExpression := (letExpression)$^{+}$ “in” (expression | term) | expression

letExpression :=

“let” name [“(” [formalParameter] “)”] [“:” typeSpecifier] “=” term

expression := logicalExpression

logicalExpression :=

“(” logicalExpression “)” |

relationalExpression |

“not” logicalExpression |

logicalExpression binaryLogicalOperator logicalExpression

binaryLogicalOperator := “and” | “or” | “xor” | “implies”

relationalExpression :=

“(” relationalExpression “)” |

booleanTerm |

relationalTerm

relationalTerm :=

term relationalOperator term

relationalOperator := “=” | “>” | “<” | “>=” | “<=” | “<>”

term :=

numericTerm |

booleanTerm |

stringTerm |

collectionTerm |

enumLiteral |
"self" |
ifExpression |
term propertyCallList

numericTerm :=
"(" numericTerm ")" |
multiplicativeExpression |
numericTerm addOperator numericTerm |
unaryOperator numericTerm |
term "." numericPropertyCall

numericPropertyCall:=
("abs" | "floor"| "round") "()" |
("max"| "min" | "div" | "mod") "(" term ")" |
pathName [timeExpression] [qualifiers] ["("[actualParameter]")"]

multiplicativeExpression :=
"(" multiplicativeExpression ")" |
unaryExpression |
multiplicativeExpression multiplyOperator multiplicativeExpression

unaryExpression := number | name
number := ["0"-"9"] (["0"-"9"])*
["." ["0"-"9"] (["0"-"9"])*]
[("e" | "E") ("+" | "-")? ["0"-"9"] (["0"-"9"])*]
booleanTerm :=
"true" |
"false" |
name
stringTerm := string | term "." stringPropertyCall

stringPropertyCall :=

("size" | "toUpper" | "toLower")"()"| "concat(" term ")" |

"substring(" numericTerm "," numericTerm")"

string := " ' "

(

(~["","\\","\n","\r"]) |

(

"\\" (["n","t","b","r","f","\\","","\""] |

["0"-"7"] [["0"-"7"] [["0"-"7"]]]

)

)

)*

" ' "

collectionTerm := literalCollection | name

literalCollection:= collectionKind "{" [collectionItem ("," collectionItem)*] "}"

collectionItem :=

term |

term ".." term

collectionKind :=

"Set" |

"Bag" |

"Sequence" |

"Collection"

enumLiteral :=

name "::" name |

name "::" enumLiteral

ifExpression :=

“if” logicalExpression “then” (expression | term) “else” (expression | term) “endif”

propertyCallList := call^{+}

call:= (“.” propertyCall | “→” collectionPropertyCall)

collectionPropertyCall:=

(“forAll” | “select” | “reject” | “exists” | “any” | “one”) (blockBoolExpression) |
“collect” blockOneExpression |
(“size” | “notEmpty” | “isEmpty”) “()” |
(“includes”|”excludes”) “(“ term “)” |
(“includesAll” | “excludesAll”) “(“ collectionTerm “)”

blockBoolExpression := “(“ [declarator] logicalExpression “)”

blockExpression := “(“ [declarator] [actualParameter] “)”

blockOneExpression := “(“ [declarator] term “)”

declarator := name (“,”name)* [“:” simpleTypeSpecifier] “|”

actualParameter := term (“,”term)*

operationName:= name

formalParameter := (name “:” typeSpecifier) (“,” (name “:” typeSpecifier))*;

qualifiers := “[” actualParameter“]”

propertyCall :=

pathName [timeExpression] [qualifiers] ["("[actualParameter]")"]

timeExpression := "@" "pre"

typeSpecifier :=

simpleTypeSpecifier |

collectionType

collectionType := collectionKind **"(" simpleTypeSpecifier ")"**

simpleTypeSpecifier :=

"Integer" |

"String" |

"Boolean" |

"Real" |

pathname

pathName :=

name |

name **"::"** pathName

returnType := typeSpecifier

multiplyOperator := "*" | "/"

unaryOperator := "-"

addOperator := "+" | "-"

name := charForNameTop | charForNameTop charForName

charForName := /* Characters except inhibitedChar; the available characters shall be determined by the tool implementers ultimately.*/

charForNameTop := /* Characters except inhibitedChar and [“0”-”9”]; the available characters shall be determined by the tool implementers ultimately.*/

Capítulo 3

TRADUCCIÓN DE MODELO UML CON EXPRESIONES OCL A OBJECT-Z

Propiedades De La Traducción

Dado un diagrama de clases UML, incluyendo expresiones OCL, se usa 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 llamado Object-Z (extensión de Z), [Spivey]. Z es un lenguaje de especificación formal y provee una notación para describir las características del sistema basado en lógica de primer orden y teoría de conjuntos.

Se presenta una función de traducción que toma un modelo UML, que incluye un diagrama de clases UML enriquecido con expresiones OCL, y retorna una especificación en Object-Z como muestra la **Figura 1.**

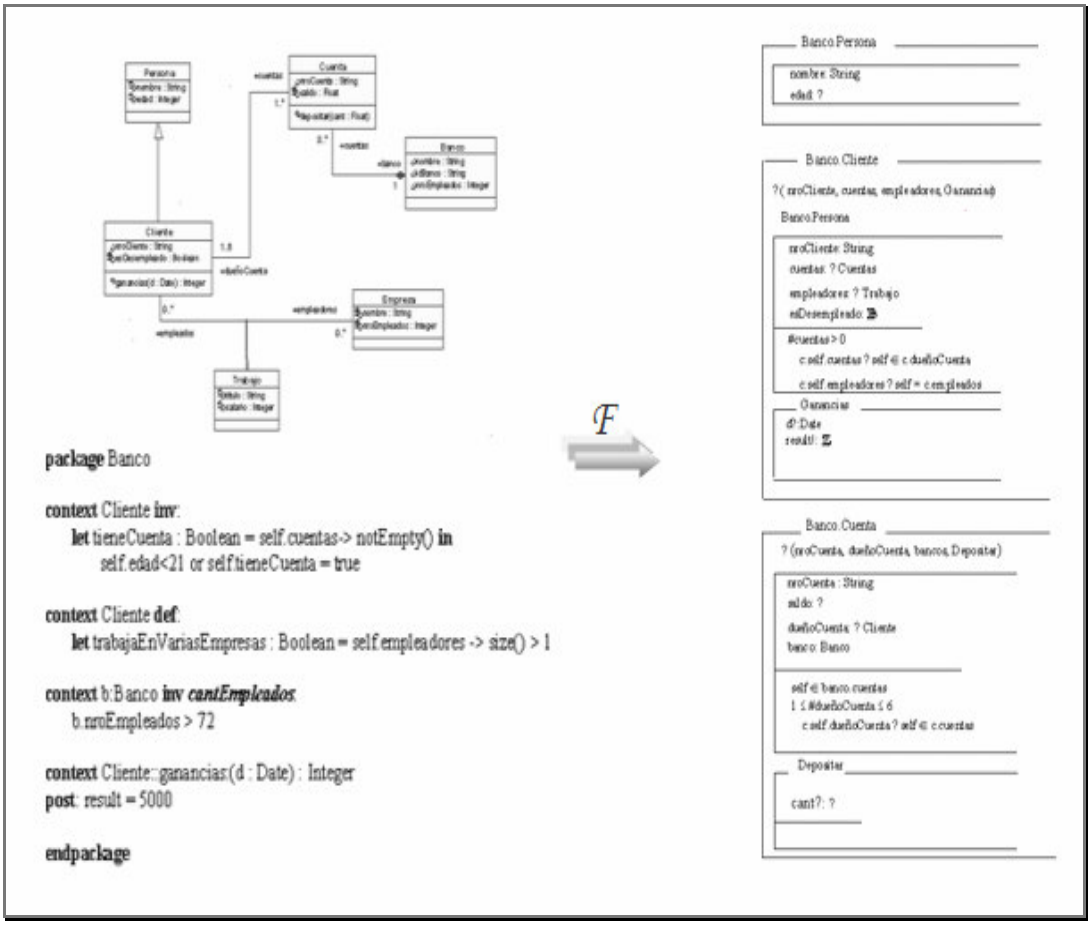


Figura 1: Función de Traducción

Función de traducción $\mathcal{F}$

Se define a continuación la función de traducción $\mathcal{F}$. Esta función tiene dos parámetros. El primero es la especificación Object-Z inicial y el segundo es el modelo UML, que incluye un diagrama de clases UML enriquecido con expresiones OCL. Luego retorna una especificación en Object-Z, que incluye todas las clases Object-Z que representan a las clases del diagrama UML. A demás la función $\mathcal{F}$ enriquece cada clase Object-Z con al traducción de las expresiones OCL. Por lo tanto la función $\mathcal{F}$ queda definida de la siguiente manera:

$$\mathcal{F}\,(S, umlModel \cup oclFile) = \textbf{let } S' = \mathcal{F}_{\text{uml}}\,(S, umlModel)$$
$$\textbf{in } \mathcal{F}_{\text{ocl}}\,(S', oclfile)$$

Primero se verá la traducción de un diagrama de clases. Luego se incorporará la traducción de las expresiones OCL. Se utilizará un ejemplo para clarificar el proceso de traducción. El ejemplo completo se puede ver en el Apéndice A.

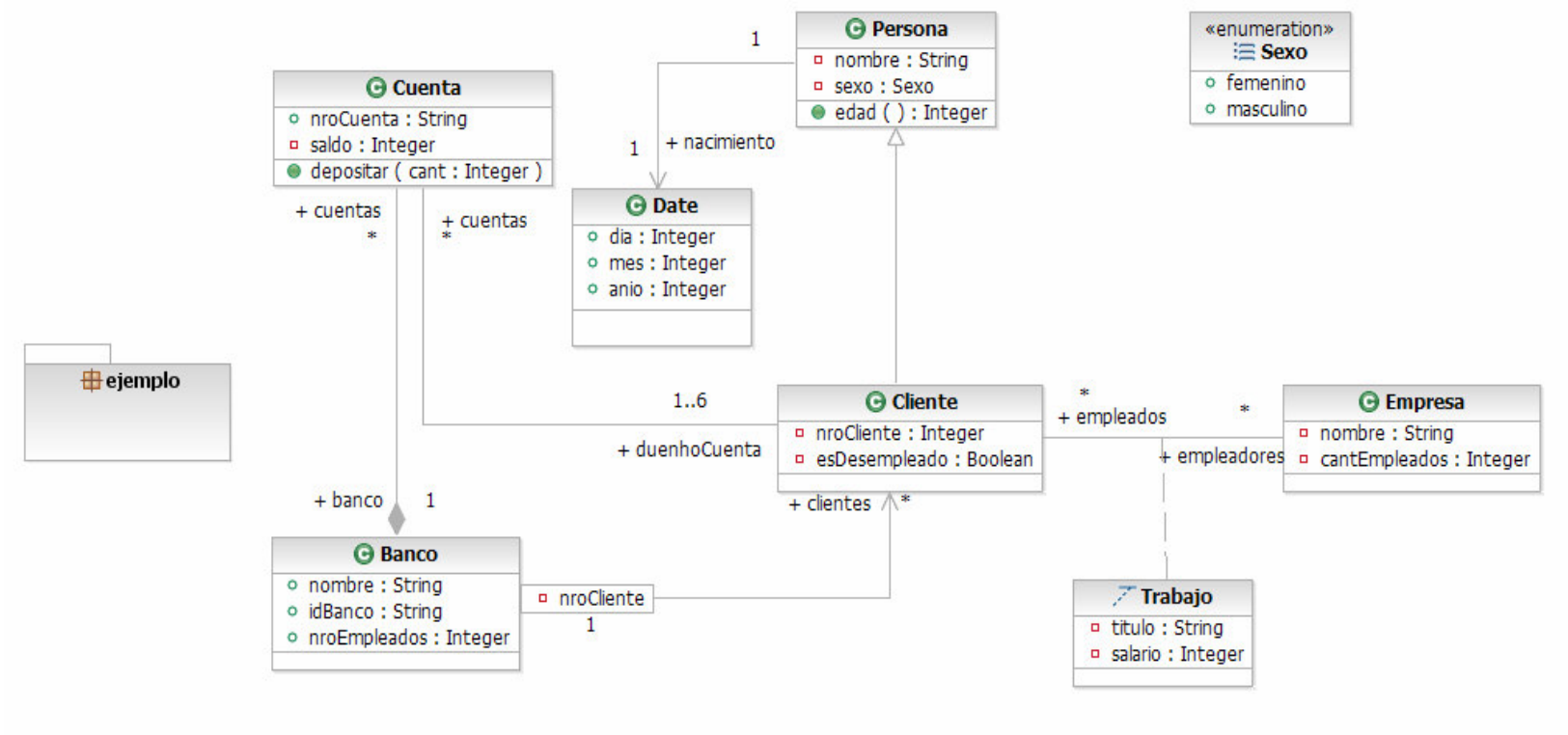
Notar que las palabras que aparecen en *Itálica*, representan variables.

Traducción De Un Diagrama De Clases - $\mathcal{F}_{uml}$

Se describirá como se traduce cada elemento de un diagrama UML a Object-Z, explicando como la función $\mathcal{F}_{uml}$ traduce un modelo UML a Object-Z.

Este trabajo, se centra en la definición formal de la función de traducción de expresiones OCL a Object-Z. Pero, para esta traducción es necesaria la traducción de un modelo UML, sobre el cual se definen las restricciones. Por este motivo, la función de traducción $\mathcal{F}_{uml}$ será descripta de una forma más informal, que la función de traducción $\mathcal{F}_{ocl}$ que es el objetivo de este trabajo.

Existen varios trabajos relacionados sobre la traducción de un diagrama de clases UML a Object-Z. Ver referencias [UML-Object-Z]



Paquete ejemplo ***Clases del paquete ejemplo***

Tipos Básicos

La transformación de los tipos básicos en Object-Z es directa. El siguiente cuadro muestra la traducción de estos tipos:

Tipos Básico	**Tipos en Object-Z**
Char	Char
String	String
Boolean	$\mathbb{B}$
Float	$\mathbb{R}$
Integer	$\mathbb{Z}$
Real	$\mathbb{R}$

El tipo Carácter es definido en Z free type como:

Char ::= 'a' | 'b' | ... | '1' | '2' | ... | ':' | '/' | '#' | ...

El tipo String es definido como una secuencia de caracteres:

String == seq Char

También se define un tipo llamado dic X Y es el conjunto de funciones parciales finitas de X a Y cuyo dominio son los valores de X.

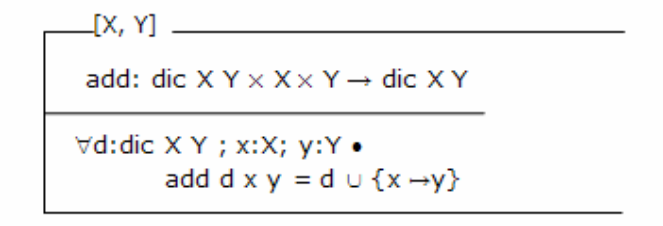
dic X Y == {f :X $\pfun$ Y}

Donde $\pfun$ son funciones parciales finitas y está definido en Z de la siguiente forma:

X $\pfun$ Y == {f:X$\fun$ Y | dom f $\in \mathbb{F}$ X }

$\fun$ Funciones finitas.

Al tipo **dic** X Y le agregamos las siguientes operaciones:



Este tipo será usado en el proceso de traducción para las asociaciones con calificadores.

Tipos enumerativos

Se crea un tipo (*Free Types*) de Object-Z, por cada tipo enumerativo de UML, con el mismo nombre que en el diagrama y se le antepone el nombre del paquete en dónde se encuentra la clase y un punto, ".". Cada valor del tipo enumerativo es declarado en notación BNF-like, es decir, de la forma:

$$enum ::= a \mid b \mid c.$$

Cada valor tipo enumerativo es una constante del tipo Object-Z definido.

Ejemplo: Traducción de l enumerativo Sexo.

Enumerativo Sexo en el paquete ejemplo del diagrama UML

Ejemplo.Sexo:= femenino | masculino

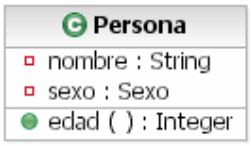
Ejemplo.Sexo en Object-Z que contiene dos constantes: femenino y masculino

Cada enumerativo traducido de un diagrama UML, es agregado a la especificación Object-Z.

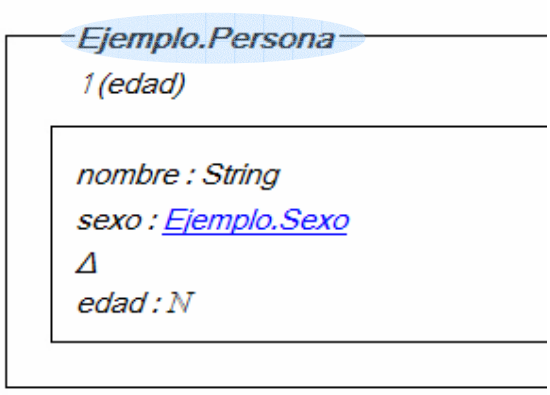
Clases

Se crea una clase en Object-Z por cada clase en el diagrama de clases y tienen el mismo nombre que en el diagrama UML y se le antepone el nombre del paquete en dónde se encuentra la clase y un punto, ".".

Ejemplo: Traducción de la clase Persona.



Clase Persona en el paquete ejemplo del diagrama UML



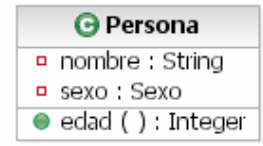
Clase Ejemplo.Persona en Object-Z

Cada clase traducida de un diagrama UML, es agregada a la especificación Object-Z, para luego obtener una especificación que represente al modelo UML en Object-Z.

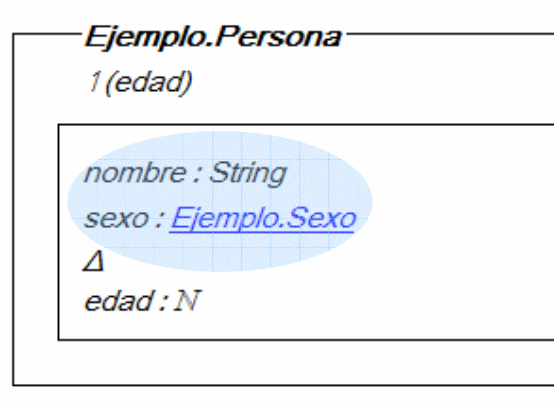
Atributos

Los atributos de una clase UML son traducidos a esquemas de estados (state schema) en la clase Object-Z correspondiente. Los atributos públicos se agregan a la lista de visibilidad. Los atributos con valores iniciales aparecerán en el esquema inicial (Init) de la clase Object-Z. Si la multiplicidad es distinta de uno, el tipo del atributo es un conjunto de las partes de los objetos de la clase opuesta, sino es del tipo de la clase opuesta. La multiplicidad debe especificarse

como un esquema de estado de la clase correspondiente. La figura se muestra como la traducción de los atributos de la clase Persona. Se puede observar que los atributos no aparecen en la lista de visibilidad porque en el diagrama aparecen como atributos privados.
Ejemplo: Traducción de los atributos de la clase Persona.



Atributos de la clase Persona en UML



Atributos de la clase Persona en Object-Z

Los atributos de la clase Persona son privados, por lo tanto no son agregados a la lista de visibilidad.

Operaciones

Las operaciones de las clases UML son divididas en dos casos:

- Operaciones de consulta, "Query", que no cambian el estado del objeto receptor.
- Operaciones de modificación, "not Query", que cambian el estado del objeto receptor.

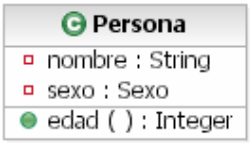
- **Operaciones De Consulta**

Este tipo de operaciones son las que aparecen un una expresión OCL, porque en una expresión OCL pueden ser usados todos los atributos, asociaciones finales y operaciones sin efectos secundarios. Un método u operación es definido libre de efectos laterales (side effect free) si es de consulta. Es decir que el atributo "isQuery" de la operación tenga el valor "true".

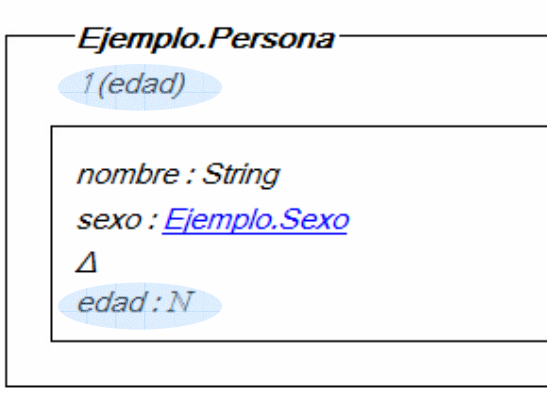
Estas operaciones son traducidas como variables secundarias de estado en la clase Object-Z correspondiente. Las operaciones de consulta siempre retornan un resultado, por lo tanto tienen un tipo de retorno. La traducción de este tipo de retorno es el que toma la variable de estado. Si la operación posee parámetros, entonces la variable de estado será una función total que recibe tantos argumentos como parámetros tenga la operación de la clase UML. Supongamos que la operación recibe un parámetro de tipo "Integer" y retorna un valor de tipo "Boolean" entonces la función de traducción creará en el esquema de estado un atributo con el mismo nombre que la operación y de tipo "$\mathbb{Z} \rightarrow \mathbb{B}$".

Las operaciones públicas, son agregadas a la lista de visibilidad, al igual que los atributos públicos.

Ejemplo: Traducción de la operación edad de la clase Persona.



Operación edad en la clase Persona en UML



La variable de estado edad en la clase Persona en Object-Z

La operación "edad():Integer" en la clase Persona, es pública, entonces es agregada a la lista de visibilidad.

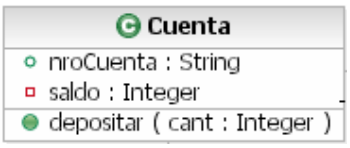
- **Operaciones de modificación**

Las operaciones de modificación traducidas a esquemas de operaciones en la clase Object-Z correspondiente y con el mismo nombre que en la clase UML. Los parámetros son traducidos como variables de entrada con igual nombre. El tipo de retorno es traducido a una variable de salida llamada "result", si es que la operación retrna algún resultado.

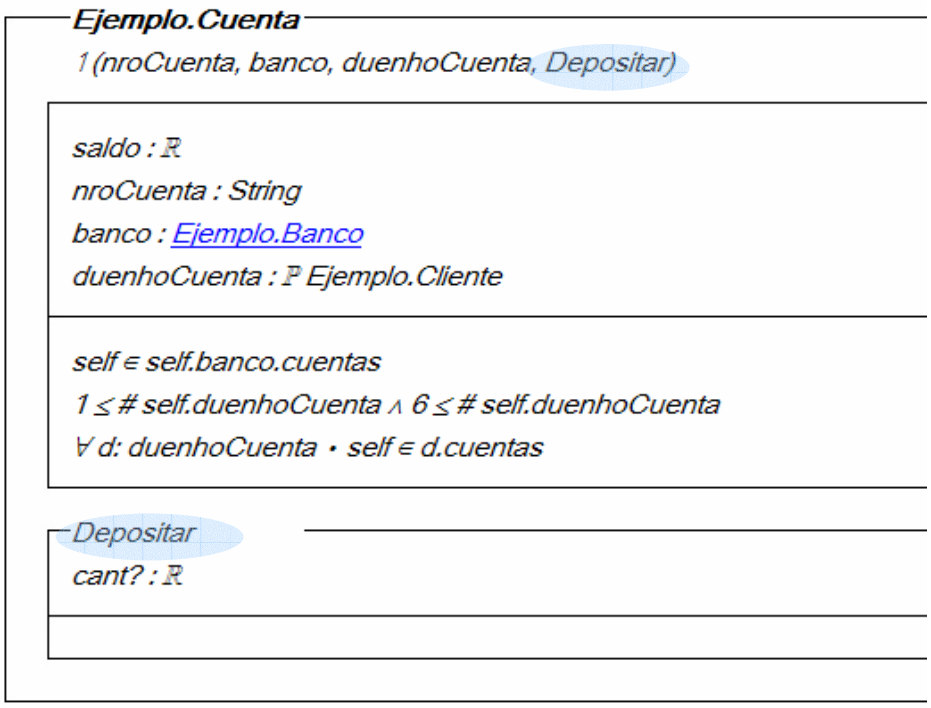
Las operaciones públicas, son agregadas a la lista de visibilidad, al igual que los atributos públicos.

En el la figura se muestra como se traduce la operación depositar de la clase Cuenta. Tiene un parámetro que es representado por la variable de entrada "cant" (cant?).

Ejemplo: Traducción de la operación depositar de la clase Cuenta.



Operación depositar en la clase Cuenta en UML



Operación Depositar en la clase Cuenta en Object-Z

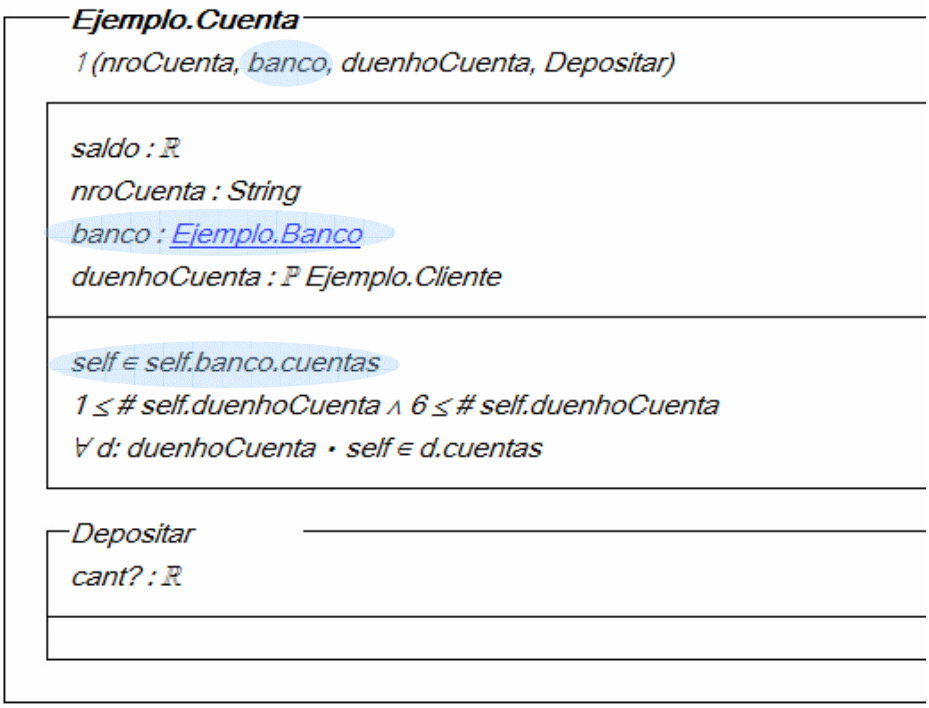
La operación "depositar(cant:Integer)" en la clase Cuenta, es pública, entonces es agregada a la lista de visibilidad.

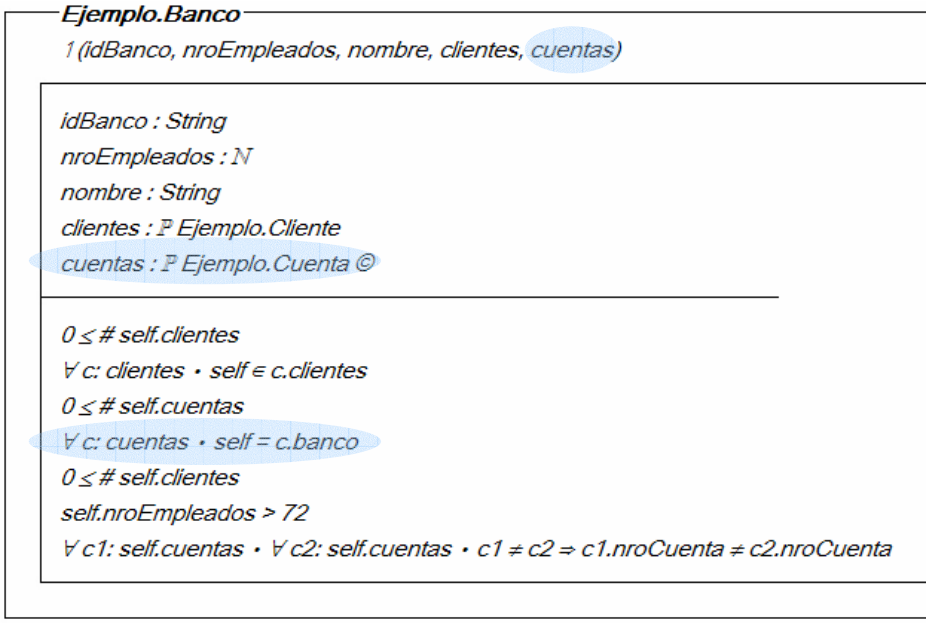
Asociaciones

Para cada asociación final de una asociación, si el extremo opuesto es navegable, se incluye la clase opuesta como atributo en una clase de Object-Z. Si la multiplicidad es distinta de uno, el tipo del atributo es un conjunto de las partes de los objetos de la clase opuesta, sino es del tipo de la clase opuesta. El nombre del atributo es el nombre del rol de la asociación final. Si el rol no fue especificado, entonces el nombre será el mismo que la clase opuesta. La multiplicidad debe especificarse como un esquema de estado de la clase correspondiente. Si los extremos de la asociación son navegables, entonces debe agregarse una restricción para especificar que la clase es un elemento de un atributo en la clase contraria. Si una clase tiene composición con otra clase, entonces se agrega © al lado del atributo que representa la otra clase para mostrar la contención del objeto no compartido en Object-Z.

Ejemplo: Traducción de una asociación entre la clase Cuenta y la clase Banco.

Asociación entre Banco y Cuenta en UML

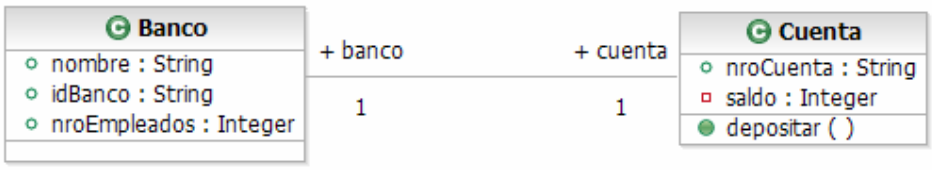




Traducción de una asociación entre Banco y Cuenta en Object-Z

Se muestra a continuación los posibles casos de la multiplicidad de una asociación y como se traduce a Object-Z. Los casos de Multiplicidad 0..1, se traduce igual que el caso de Multiplicidad *. Tomemos posibles ejemplos de asociación bi-direccional entre las clases Banco y Cuenta. Notar que se muestra a continuación los casos más generales.

Asociación uno a uno



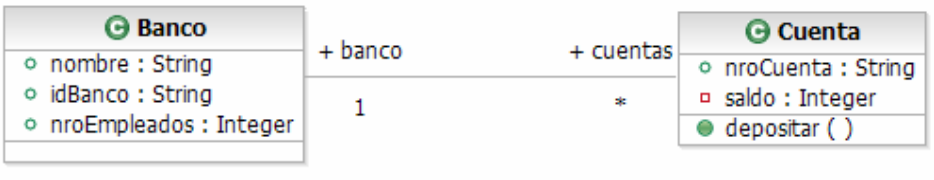
En la clase Banco se agrega el siguiente predicado en el esquema de estado:

self = self.cuenta.banco

En la clase Cuenta se agrega el siguiente predicado en el esquema de estado:

self = self.banco.cuenta

Asociación uno a muchos



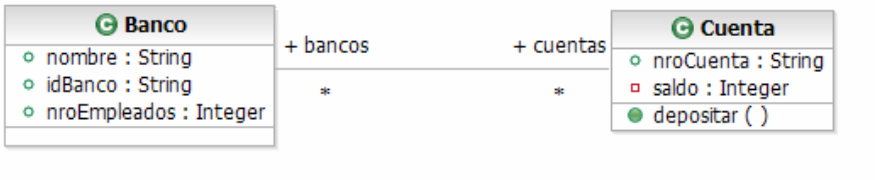
En la clase Banco se agrega el siguiente predicado en el esquema de estado:

$\forall c \in cuentas \bullet self = c.banco$

En la clase Cuenta se agrega el siguiente predicado en el esquema de estado:

$self \in self.banco.cuentas$

Asociación muchos a muchos



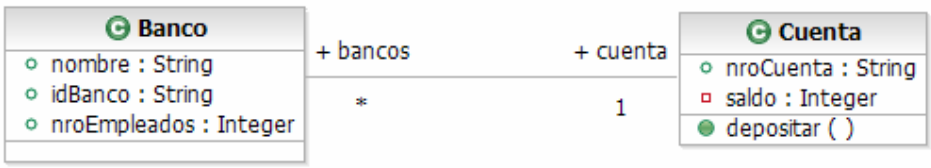
En la clase Banco se agrega el siguiente predicado en el esquema de estado:

$\forall c \in cuentas \bullet self \in c.bancos$

En la clase Cuenta se agrega el siguiente predicado en el esquema de estado:

$\forall b \in bancos \bullet self \in b.cuentas$

Asociación muchos a uno



En la clase Banco se agrega el siguiente predicado en el esquema de estado:

$self \in self.cuenta.bancos$

En la clase Cuenta se agrega el siguiente predicado en el esquema de estado:

$\forall b \in bancos \bullet self = b.cuenta$

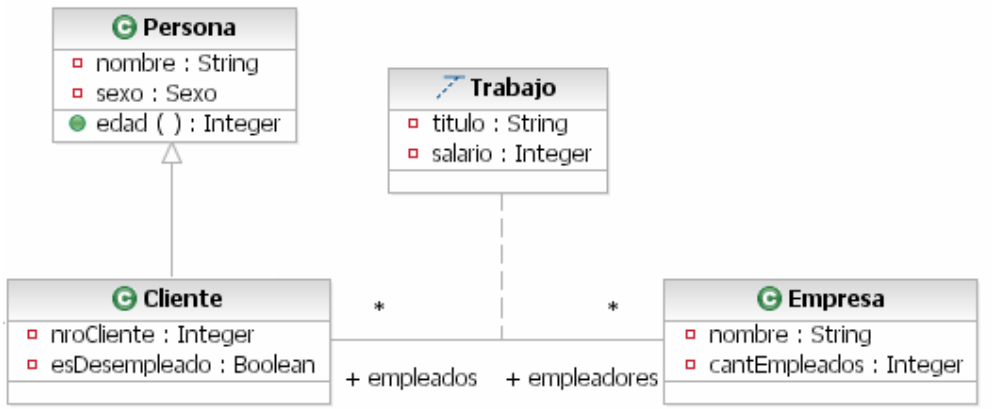
Clase de Asociación

Una clase de asociación es una asociación que tiene clases como propiedades así como propiedades de asociación. Las clases asociadas por la clase de asociación, son traducidas igual que si tuviera una asociación común, pero los atributos que serían del tipo de la clase opuesta, son del tipo de la clase de asociación.

Una clase Object-Z es creada para representar la clase de asociación, con el mismo nombre de la clase de asociación, con los mismos atributos y métodos. Se agregan atributos extras para cada uno de las dos clases asociadas. Estos atributos deben tener el mismo nombre que el nombre del rol, si se especifica, o deben tener el nombre de la clase, con la primera letra en minúscula. El tipo del atributo será la clase que representa. Ambos atributos deben aparecer, aun cuando un extremo de la asociación no es navegable.

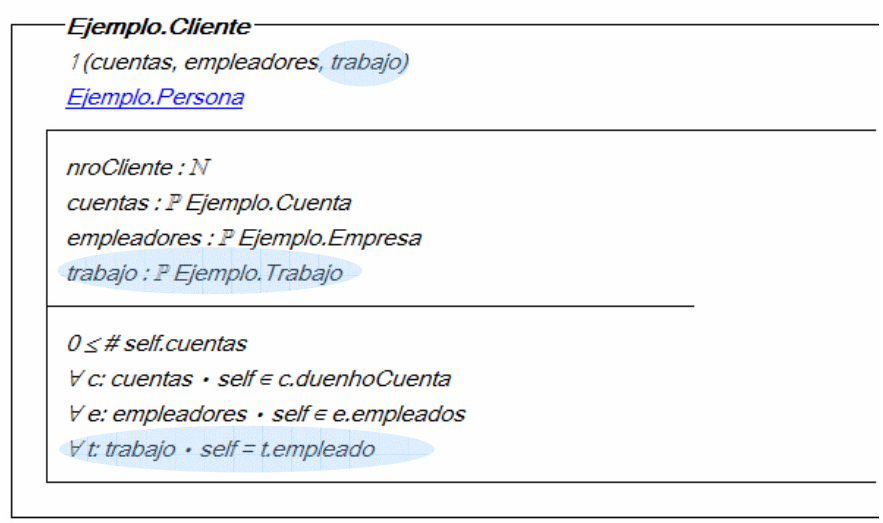
Si los extremos de la asociación son navegables, entonces debe agregarse una restricción para especificar que la clase es un elemento de un atributo en la clase contraria.

Ejemplo: Traducción de la clase de asociación Trabajo, entre la clase Cliente y la clase Empresa.

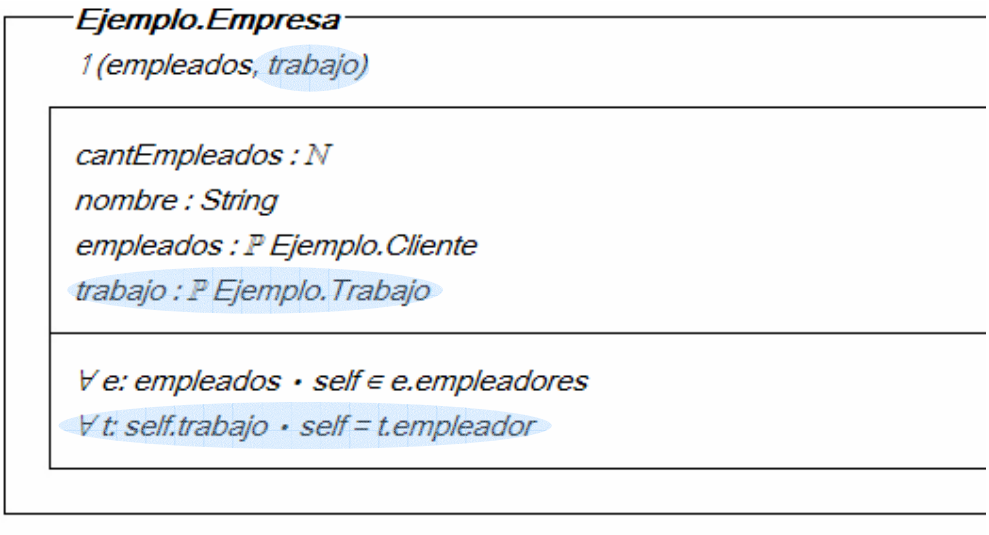


Clase de asociación Trabajo en UML

Traducción de Clase de asociación Trabajo a Object-Z



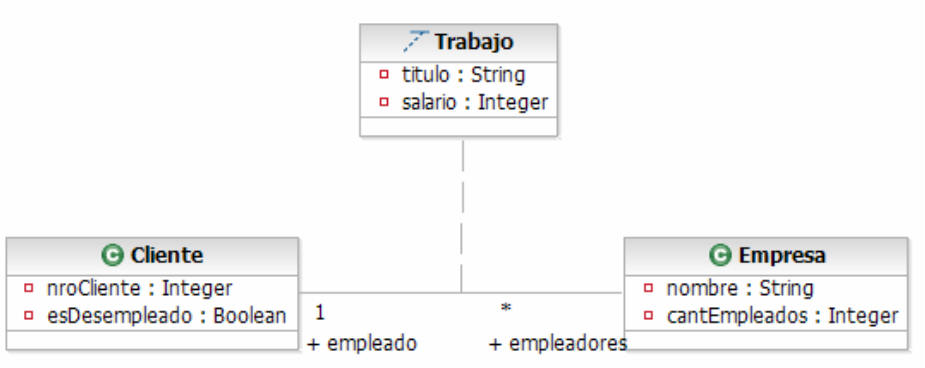
Traducción de Clase Cliente a Object-Z



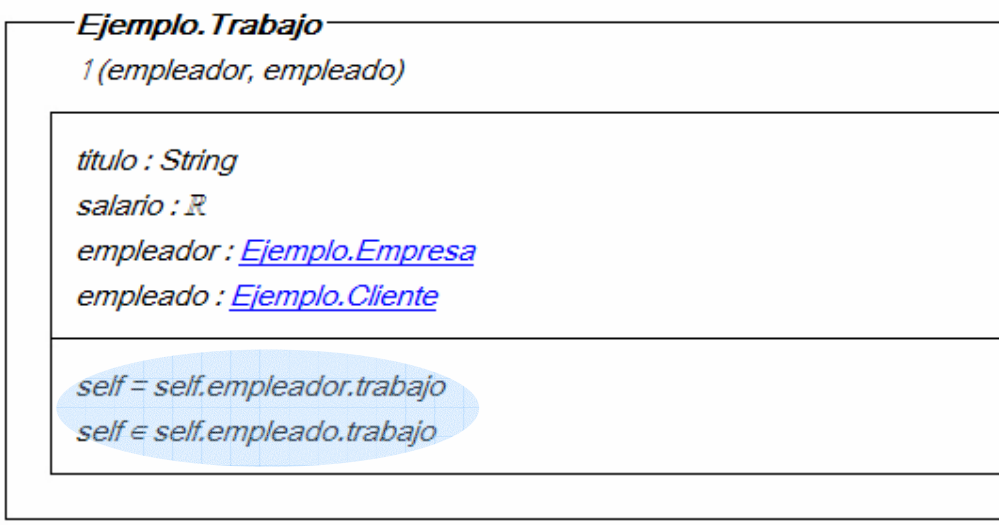
Traducción de Clase Empresa a Object-Z

Se muestra a continuación los posibles casos de la multiplicidad de una clase de asociación y como se traduce a Object-Z. Los casos de Multiplicidad 0..1, se traduce igual que el caso de Multiplicidad *.

Asociación uno a muchos

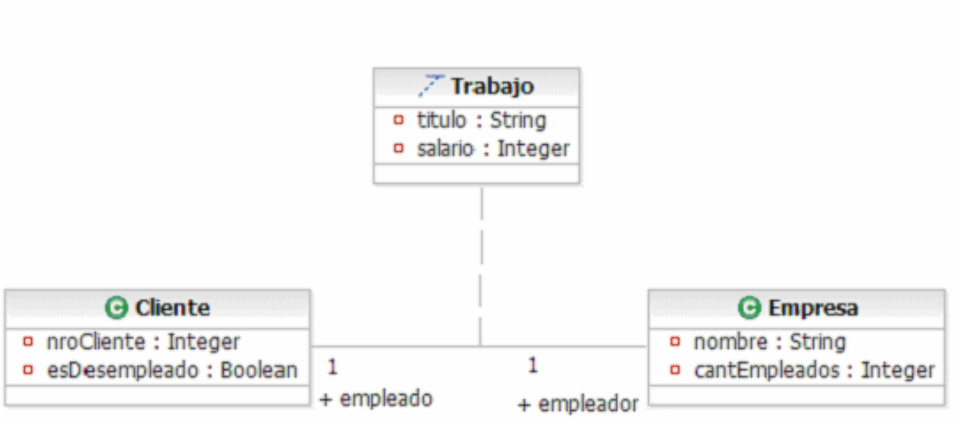


Clase de asociación Trabajo en UML

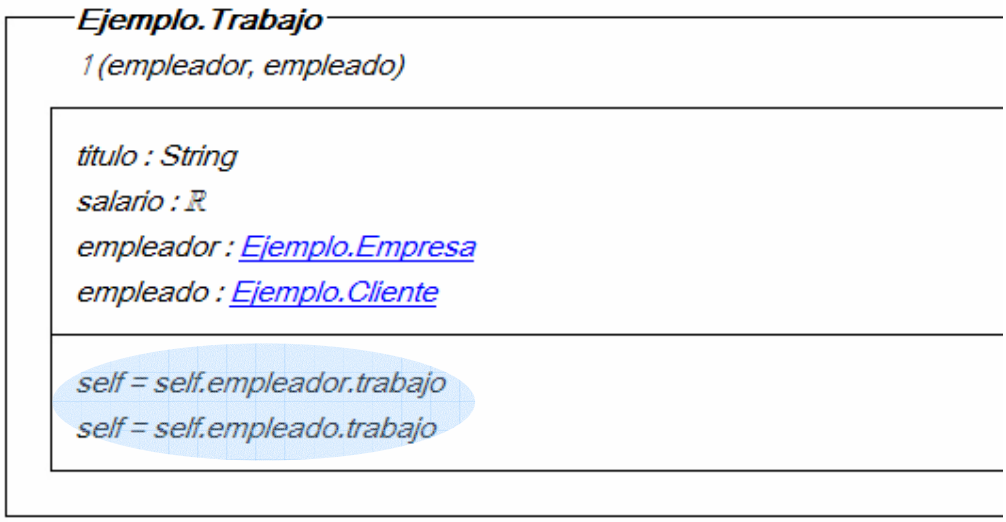


Traducción de Clase de asociación Trabajo a Object-Z

Asociación uno a uno

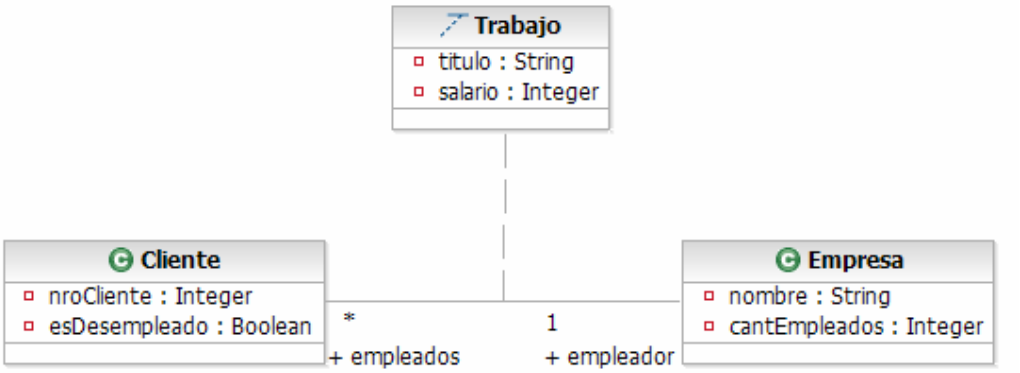


Clase de asociación Trabajo en UML

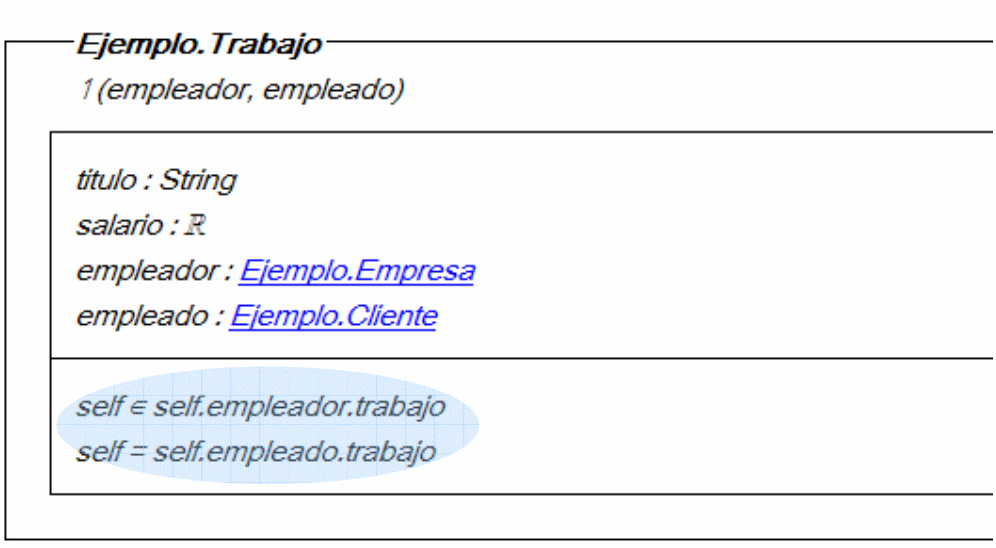


Traducción de Clase de asociación Trabajo a Object-Z

Asociación muchos a uno



Clase de asociación Trabajo en UML

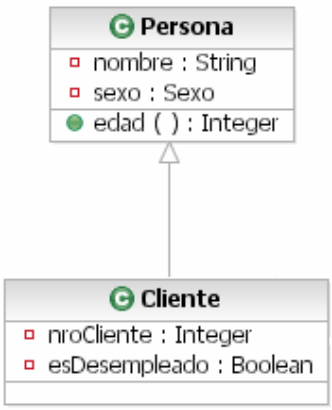


Traducción de Clase de asociación Trabajo a Object-Z

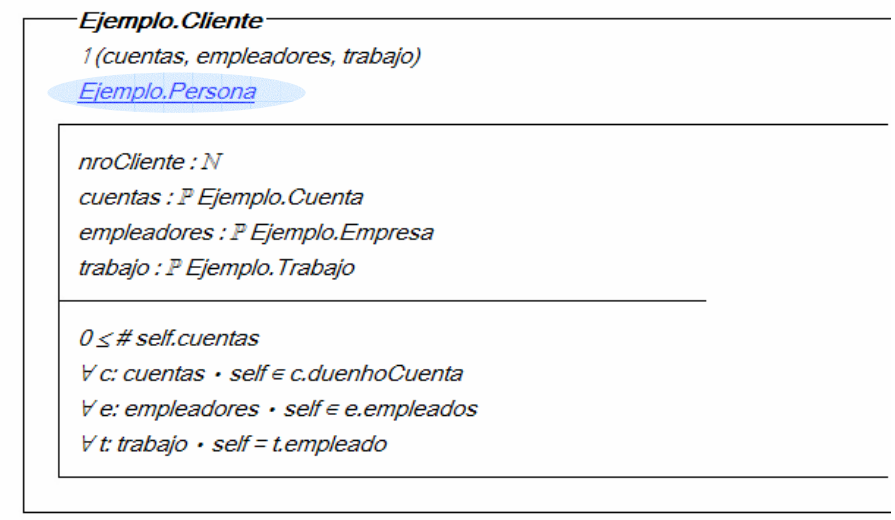
Generalización

La relación de generalización se representa en Object-Z poniendo el nombre de la clase de la que hereda en el esquema de clase, debajo de la lista de declaración.

Ejemplo: Traducción de la relación de generalización entre la clase Cliente y la clase Persona.



Generalización entre la clase Cliente y la clase Persona en UML



Traducción de la generalización de la clase Cliente en Object-Z

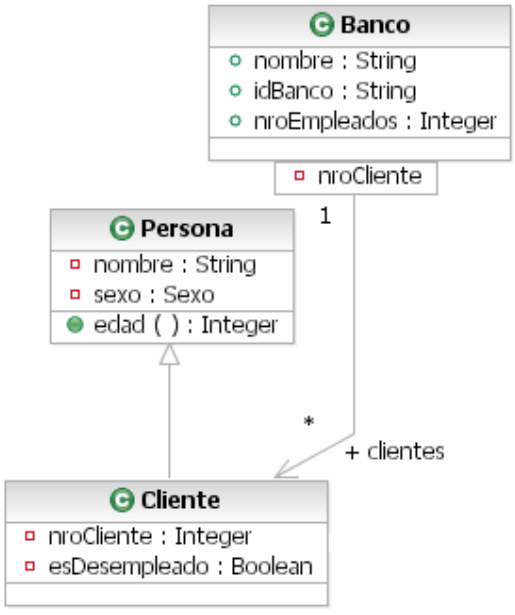
Calificador (Qualifiers)

Las asociaciones calificadas usan uno o más atributos calificadores para seleccionar los objetos al otro extremo de la asociación. Para navegarlos, podemos agregar los valores por los calificadores a la navegación. Se usa corchetes después del nombre del rol. Es permisible omitir el valor del calificador, en ese caso el resultado será todos los objetos del otro lado de la asociación.

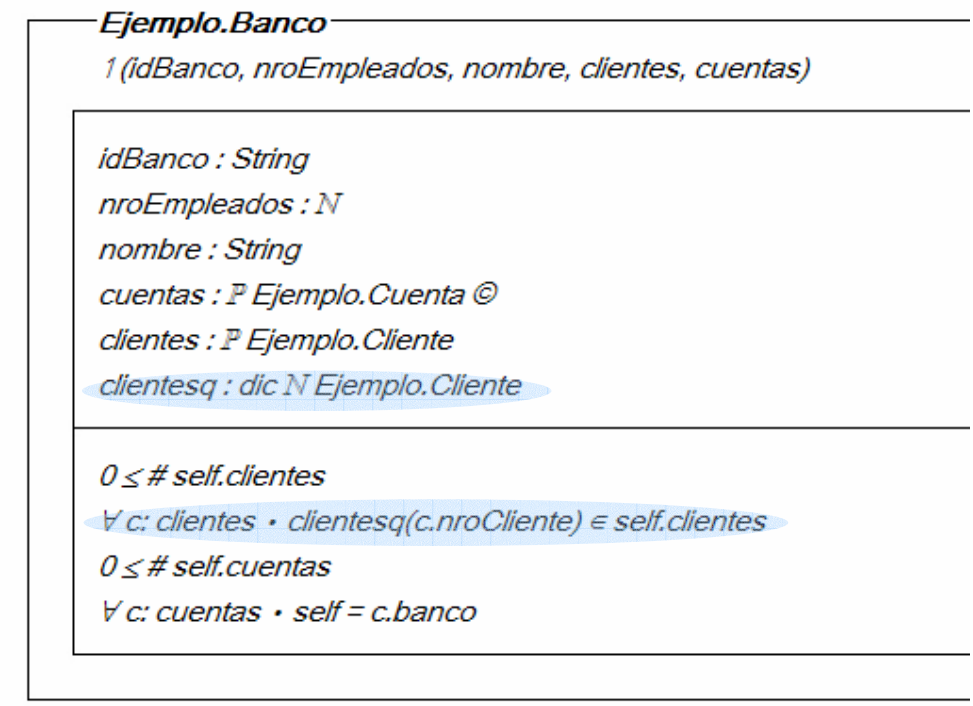
Las asociaciones calificadas agregan un atributo más a la clase de Object-Z el cual tiene tipo dic, con el mismo nombre que cla asociación y una q al final del nombre.

Por ejemplo: En la asociación clientes, entre la clase Banco y la clase Cliente, hay un calificador por el atributo "nroCliente", por lo cual se agrega a la clase Object-Z, Ejemplo.Banco, un nuevo atributo llamado "cuentasq" de tipo (dic Integer Ejemplo.Cliente) y un predicado en el esquema de estado de la clase Ejemplo.Banco, que indica que los elementos de ese atributo, son los mismos que los del atributo cuentas.

Ejemplo: Traducción del calificador de la clase Banco.



Calificador en la clase Banco del modelo UML



Traducción del calificador en Object-Z

Traducción De Una Expresión OCL - $\mathcal{F}_{ocl}$

Ahora se muestra la función de traducción $\mathcal{F}_{ocl}$ que toma una especificación Object-Z y enriquece con la traducción de las expresiones OCL que tienen el modelo UML.

Esta función tiene dos parámetros. El primero es la especificación que contiene las clases Object-Z correspondiente al modelo UML sobre el cual se escribieron las restricciones OCL. Estas restricciones son las que recibe la función en su segundo parámetro. Luego retorna la especificación Object-Z que recibió, enriquecida con la traducción a Object-Z de las restricciones OCL recibidas.

La notación para esta función, es:

$$\mathcal{F}_{ocl}(S, oclFile)=S'$$

Donde S es la especificación Object-Z inicial y S' es el resultado de enriquecer la especificación S con la traducción de las expresiones OCL escritas en un archivo OCL.

La función de traducción, $\mathcal{F}_{ocl}$, es definida recursivamente sobre la sintaxis de las expresiones OCL. Se parte de un archivo OCL, luego se va recordando la forma en que esta compuesta cada parte de un *oclFile*, según lo describe la gramática y se va detallando como se traduce cada parte específica de las expresiones.

La definición de la función comienza con el símbolo inicial de la gramática OCL. Este símbolo es oclFile:

oclFile:= package$^+$

Ahora se puede describir a **package$^+$** recursivamente. Este proceso de re-escribir símbolos no terminales en forma recursiva se va a ver a lo largo de todo el documento, porque facilitará la escritura de la función de traducción. A demás se utilizara indentación de la expesión OCL para facilitar la lectura de la traducción.

package$^+$ = package

package$^+$ = package package$^+$

Entones quedaría planteada la función de la siguiente forma:

$\mathcal{F}_{ocl}$(S, *package package*$^+$) =

let *S'* = $\mathcal{F}_{ocl}$(*S*, *package*)

in $\mathcal{F}_{ocl}$(*S'*, *package*$^+$)

La expresión **let…in** no corresponde a Object-Z se utiliza para tener una notación más clara de la definición de la función. La expresión **let...in** es útil cuando se necesita un conjunto de declaraciones locales. Un ejemplo simple es el siguiente:

let

x = $(5^3+4)/2$

in x * x

Este ejemplo define en la cláusula **let**, una expresión x que tiene el valor de la expresión $(5^3+4)/2$. La cláusula **in** retorna el valor de hacer x*x, donde x esta definida en la cláusula **let.**

Para especificar la función que falta $\mathcal{F}_{ocl}$ (*S*, *package*), se tienen en cuenta la definición de *package* en la gramática. Esto se va a realizar para cada definición más específica que vaya apareciendo en la definición de la función de traducción $\mathcal{F}_{ocl}$.

El símbolo no terminal en la gramática OCL, package, aparece definido de la siguiente forma:

package := "package" packageName (constraint)* "endpackage".

Antes de escribir la función de traducción se re-escribe **(constraint)*** en forma recursiva:

1. **(constraint)*: es vacío, es decir no hay ninguna ocurrencia**
2. **(constraint)*: constraint**
3. **(constraint)*: constraint (constraint)***

Luego se define la función en dos casos (caso recursivo y caso base):

Caso recursivo:

$\mathcal{F}_{ocl}$(*S*, "package" *packageName constraint constraint** "endpackage") =

let *S'* = $\mathcal{F}_{ocl}$(*S*, "package" *packageName constraint* "endpackage")

in $\mathcal{F}_{ocl}$(*S'*, "package" *packageName constraint** "endpackage")

Caso Base:

$\mathcal{F}_{ocl}$(*S*, "package" *packageName* "endpackage") = *S*

El caso es más complejo es el definido dentro de la clusula **let** que retorna la especificaci'on *S'*:

$\mathcal{F}_{ocl}$(*S*, "package" *packageName constraint* "endpackage")

Un *constraint* se divide en dos casos que se describen a continuación para definir la función de traducción en más detalle.

constraint :=

" context " operationContext contextStereotype^{+} |

" context " classifierContext contextDefinition^{+}

Como estos casos son diferentes se van a exponer cada uno por separado.

Caso 1: "context" operationContext contextStereotype+

Este tipo de declaración usa la palabra *context*, seguida del tipo y la declaración de la operación. Luego continúan los términos 'pre:' y 'post:' para luego escribir expresiones OCL que indiquen Precondiciones y Poscondiciones. Es decir, es el caso que se usa para incluir una expresión OCL como parte de una Precondición o Poscondición, asociada a una Operación.

context nombreDeTipo::NombreDeOperacion(param1 : Tipo1, ...): TipoDeRetorno
pre : param1 > ...
post: result = ...

La función de traducción para este caso se definiría de la siguiente manera:

$\mathcal{F}_{ocl}$(S, "package" *packageName*
"context" *operationContext*
contextStereotype^{+}
"endpackage")

Pero se re-escribe recursivamente, **contextStereotype**$^{+}$, símbolo no terminal de la gramática OCL:

contextStereotype^{+} **= contextStereotype**
contextStereotype^{+} **= contextStereotype contextStereotype**$^{+}$

Quedando expresado recursivamente de la siguiente forma:

$\mathcal{F}_{ocl}$(S, "package" *packageName*

"context" *operationContext*

contextStereotype contextStereotype+

"endpackage") =

let S' = $\mathcal{F}_{ocl}$(*S*, "package" *packageName*

"context" *operationContext contextStereotype*

"endpackage")

in $\mathcal{F}_{ocl}$(*S'*, "package" *packageName*

"context" *operationContext*

contextStereotype^{+}

"endpackage")

La re-escritura de los símbolos no terminales de la gramática es necesaria para escribir este caso de la función de traducción:

$\mathcal{F}_{ocl}$(S, "package" *packageName*

"context" *operationContext*

contextStereotype

"endpackage")

Re-escritura:

operationContext:= name "::" signatureOp

Donde,

signatureOp:= operationName "("[formalParameter] ")" [":" returnType]
contextStereotype = stereotype name ":" oclExpression
oclExpression := (letExpression)$^{+}$ "in" (expression | term) | expression

Se tienen en cuenta si la expresión OCL tiene una expresión "let..in". Si es este el caso de la expresión, entonces se traduce la expresión "Let" (letExpression), para agregar una propiedad más a la clase de la especificación Object-Z (variable de estado). Entonces tendremos los siguientes casos, para una expresión OCL:

oclExpression := (letExpression)$^+$ "in" expression

oclExpression := (letExpression)$^+$ "in" term (no es correcto)

oclExpression := expression

El segundo caso no es correcto para definir una precondición o poscondición, por lo cual queda omitida su traducción.

Antes de continuar con la descripción de la función de traducción, hay que tener en cuenta que en las poscondiciones puede aparecer el símbolo "@pre", que indica el estado anterior del atributo antes de ejecutar la operación. Es necesario obtener los nombres de los atributos que se hace referencia a su estado anterior, para luego poder hacer un renombre.

Ejemplo:

```
context Cuenta::depositar(cant:Integer)
  pre: cant>0
  post: saldo=self.saldo@pre + cant
```

La función de traducción debería agregar las siguientes condiciones en el esquema de operación, "Depositar", de la clase Object-Z "Ejemplo.Cuenta".

cant >0

saldo' = saldo + cant

La función queda como se muestra a continuación, utilizando las funciones auxiliares[1]:

[1] Notar que las funciones "enrich-condition", "lookingFor-pre", "rename", F_D y F_E, son funciones auxiliares se detallaran mas adelante.

Caso con letExpession (agrega definiciones de propiedades a la clase Object-Z):

$\mathcal{F}_{ocl}$(*S,* "package" *packageName*

"context" *name* "::" *signatureOp*

stereotype name1 ":" *(letExpression)*$^{+}$ "in" *expression*

"endpackage") =

let *S'* = enrich-with-properties(*S*, F_D(*packageName)*.F_D(*name*), *(letExpression)*$^{+}$)

listAttr=lookingFor-pre(*expression*)

expressionOZ= F_E(*expression*) rename(*listAttr*)

in enrich-condition(S',F_D(*packageName*).F_D(*name*),F_D(*signatureOp*),*expressionOZ*)

Caso sin letExpession:

$\mathcal{F}_{ocl}$(*S,* "package" *packageName*

"context" *name* "::" *signatureOp*

stereotype name1 ":" *expression*

"endpackage") =

let *listAttr*=lookingFor-pre(*expression*)

expressionOZ= F_E(*expression*) rename(*listAttr*)

in enrich-condition(S,F_D(*packageName*).F_D(*name*),F_D(*signatureOp*),*expressionOZ*)

Caso 2: "context" classifierContext contextDefinition+

Este caso de declaración usa la palabra *context*, seguida del nombre de una Clase. Luego continúan los términos 'inv:' y 'def:' que se utilizan para definir invariantes o nuevas definiciones a la Clase.

context nombreDeTipo
def : letExpression
inv: expression

Continuamos con la definición de la función $\mathcal{F}_{ocl}$ de la siguiente forma:

$\mathcal{F}_{ocl}$(*S*, " package " *packageName* " context " *classifierContext contextDefinition*$^+$ " endpackage ")

Reescribimos **contextDefinition**$^+$

contextDefinition$^+$ **= contextDefinition**

contextDefinition$^+$ **= contextDefinition contextDefinition**$^+$

$\mathcal{F}_{ocl}$(*S*, " package " *packageName*
" context " *classifierContext contextDefinition contextDefinition*$^+$
"endpackage") =
let
S' = $\mathcal{F}_{ocl}$(*S*, "package" *packageName*
" context " *classifierContext contextDefinition*
"endpackage ")
in $\mathcal{F}_{ocl}$(*S'*, " package " *packageName*
" context " *classifierContext contextDefinition*$^+$
" endpackage ")

Se detalla a continuación el caso definido en la clausula let que retona la especificación *S'*.

$\mathcal{F}_{ocl}$(*S*, " package " *packageName* " context " *classifierContext contextDefinition* " endpackage ")

Re-escritura:

contextDefinition :=

("def" [name] ":") letExpression^{+} |

("inv" [name]) ":" oclExpression)

Teniendo en cuenta la gramática. Se divide en dos nuevos casos. Si son definiciones o si son Invariantes.

Definiciones

Las definiciones que se agregan con OCL son traducidas a variables secundarias de estado, con un predicado en el esquema de estado que indica a que expresión es igual esa variable.

El valor de una variable secundaria es únicamente definido en términos de valores de una o más variables primarias. De esta forma, estas solo cambian para mantener su relación con las variables primarias. Estas variables aparecen luego del símbolo Δ.

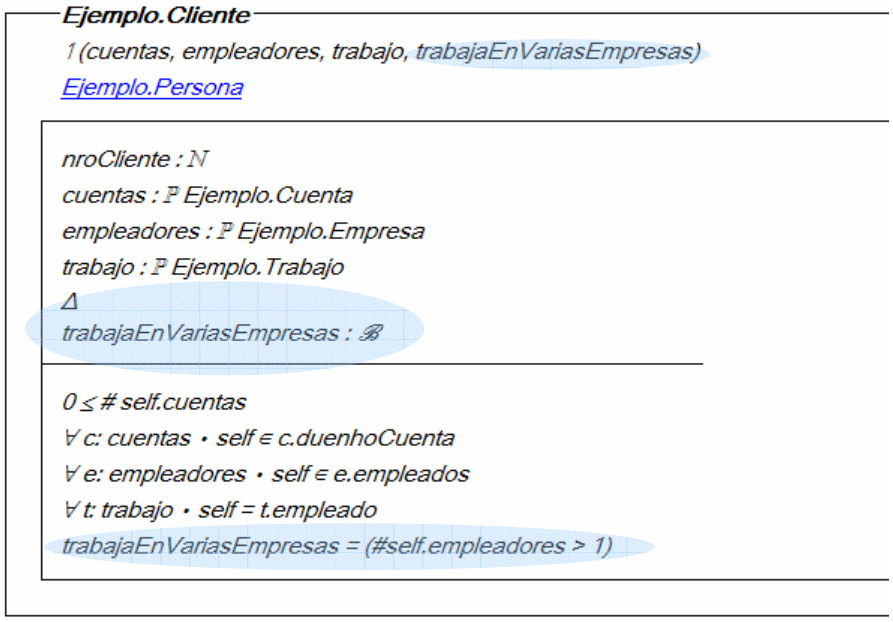
Se agrega un predicado al esquema de estado con la traducción de la expresión de la definición para definir el valor de la variable secundaria.

Si la definición contiene parámetros, el tipo de la variable de estado es una función que el dominio esta formado por la traducción de los tipos de los argumentos de la definición. A demás cada una de estas definiciones son agregadas en la lista de visibilidad de la clase correspondiente.

Ejemplo 1:

```
context Cliente def:
    let trabajaEnVariasEmpresas : Boolean = self.empleadores -> size() > 1
```

Esta definición es traducida a una variable de estado como se muestra a continuación.

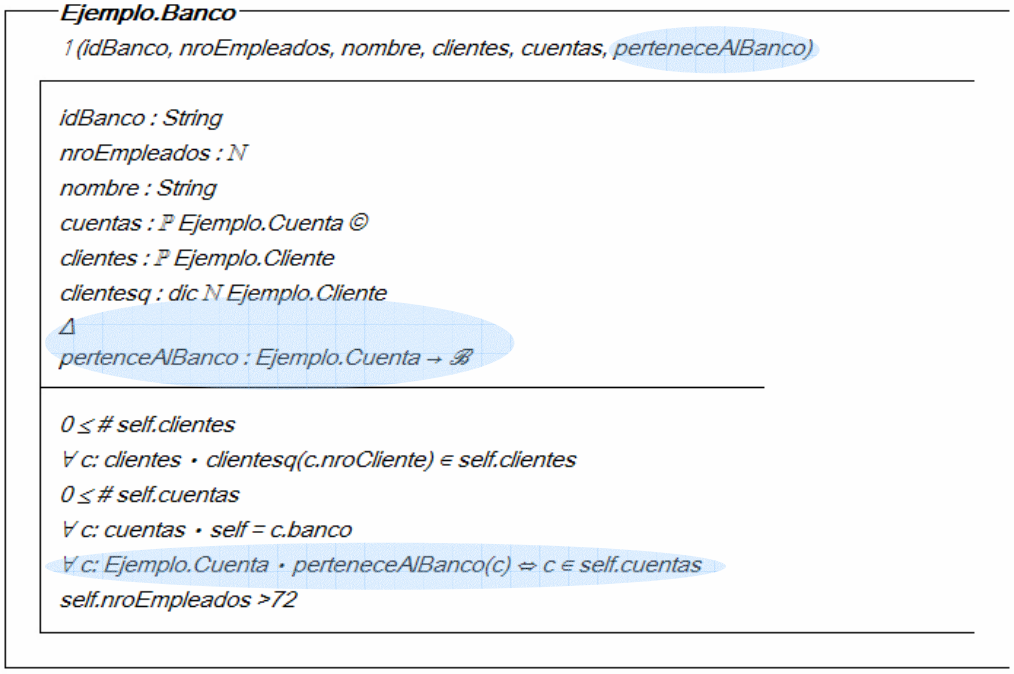


Ejemplo 1: Definición "trabajaEnVariasEmpresas" en al clase Cliente

Ejemplo 2:

context Banco **def**:

let perteneceAlBanco(c: Cuenta): Boolean = self.cuentas -> includes(c)



Ejemplo 2: Definición "pertenceAlBanco" en al clase Banco

Ejemplo de la definición "perteneceAlBanco" en al clase Banco

Con esta breve explicación se a la definición de la función.

$\mathcal{F}_{ocl}$(S, " package " *packageName*

" context " *classifierContext*

"def" *name* ":" *letExpression*$^{+}$

"endpackage") =

$\mathcal{F}_{ocl}$(S, "package" *packageName*

" context " *classifierContext*

" def " " : " *letExpression*$^{+}$

" endpackage ")

Para describir el resultado de este caso de la función se re-escribe un símbolo letExpression$^+$:

letExpression$^+$:= letExpression

letExpression$^+$:= letExpression letExpression$^+$

Este caso es diferente al caso 1 (cuando el contexto es una operación), porque en un contexto de clase se puede definir una nueva variable, que representa a las instancias de esa clase. Por ejemplo:

context b:Banco

def: …

Como muestra el ejemplo, "b", es una variable con el comportamiento de "self". Por lo tanto en la traducción, se debe agregar una reescritura del nombre de la variable. En este ejemplo, sería: **[self / b].**

Esto se debe a que la gramática de OCL permite dos construcciones para el símbolo ClassifierContext, que se detalla abajo.

Primero se muestra el caso recursivo de "letExpression$^+$" y luego se ven los dos casos, con y sin renombre de variable.

$\mathcal{F}_{ocl}$(S, " package " *packageName*

" context " *classifierContext*

"def" ":" *letExpression letExpression*$^+$

"endpackage") =

let *S'* = $\mathcal{F}_{ocl}$(*S*, "package" *packageName*

" context " *classifierContext*

" def " " : " *letExpression*

" endpackage ")

in $\mathcal{F}_{ocl}$(*S'*, "package" *packageName*

"context" *classifierContext*

"def" ":" *letExpression*$^+$

"endpackage")

La traducción de una letExpression agrega variables secundarias a la clase en la cual está definida. A demás agrega un predicado en el esquema de estado que representa el valor de la variable y es la traducción de la expresión "let", como se vio en el ejemplo de arriba. Ver *Ejemplo 1* y *Ejemplo 2*

Antes de pasar la definición de la traducción de una definición, tenemos en cuenta los dos casos que nos permite escribir la regla classifierContext:

classifierContext := name ":" name | name

Utilizando las funciones auxiliares[2], la función quedaría:

$\mathcal{F}_{ocl}$(S, "package" *packageName*

"context" *nameClass*

"def" ":" *letExpression*

"endpackage") =

enrich-with-property-visible(*S*, F_D(*packageName*).F_D(*nameClass*), F_T(*letExpression*))

$\mathcal{F}_{ocl}$(S, "package" *packageName*

"context" *nameVar* ":" *nameClass*

"def" *name* ":" *letExpression*

"endpackage") = enrich-with-property-visible(*S*, F_D(*packageName*).F_D(*nameClass*), F_T(*letExpression*) [*"self"/nameVar*])

[2] Notar que las funciones "enrich-with-property-visible" F_D y F_T, son funciones auxiliares se detallaran mas adelante.

Invariantes

Los invariantes son traducidos a un predicado en el esquema de estado de la clase correspondiente.

$\mathcal{F}_{ocl}$(S, " package " *packageName*
" context " *classifierContext*
"inv" *name* ":" *oclExpression*
"endpackage") =
$\mathcal{F}_{ocl}$(S, "package" *packageName*
" context " *classifierContext*
" inv " " : " *oclExpression*
" endpackage ")

Nota: Se elimina el nombre del invariante y se vuele a llamar a la función $\mathcal{F}_{ocl}$. El nombre del invariante no es necesario para la traducción de la expesión OCL a Object-Z.

En el siguiente caso *classifierContext* puede ser el nombre de la clase (*nameClass*) o el nombre de una variable con el nombre de la clase (*nameVar* ":" *nameClass*). Por lo cual se escribe la funcion en los siguientes casos:

$\mathcal{F}_{ocl}$(S, " package " *packageName*
" context " *nameClass*
"inv" ":" *oclExpression*
"endpackage") =
enrich-inv(S, F_D(*packageName*).F_D(*nameClass*), F_E(*oclExpression*))

$\mathcal{F}_{ocl}$(S, "package" *packageName*
"context" *nameVar* ":" *nameClass*
"inv" ":" *oclExpression*
"endpackage") =

enrich-inv(S, F_D(*packageName*).F_D(*nameClass*), F_E(*oclExpression*)[“self”/*nameVar*])

Notar: [*“self”/nameVar*] reemplaza el nombre de la variable que está en *nameVar* por “self”.

Funciones extras

Para facilitar la definición de la función de traducción $\mathcal{F}_{ocl}$, se agregan varias funciones que abstraen la traducción de los términos, las declaraciones (de variables, de tipos, de símbolos de funciones, etc.), renombre de variables, que componen una expresión OCL. Estas funciones son definidas mas abajo, pero antes se detalla una breve descripción de lo que hacen cada una de estas funciones.

Función de traducción de expresiones OCL

Una expresión OCL es una expresión que retorna un valor de verdad, es decir, una expresión lógica, una expresión relacional, un valor "true", "false", una variable o una propiedad de una clase que sea de tipo "Boolean".

- F_E: Expresión → Object-Z

La función de traducción F_E recibe una expresión OCL y retorna una expresión Object-Z.

Por ejemplo:

Sea "self.edad <21 or self.tieneCuenta" una expresión relacional, dentro de un invariante OCL, entonces la función F_E nos retorna:

F_E(self.edad<21 or self.tieneCuenta) = self.edad < 21 ∨ self.tieneCuenta

Función de traducción de términos OCL

Un término OCL es una expresión que retorna algún valor, no necesariamente un valor del tipo "Boolean". En OCL tenemos términos numéricos, términos "boolean", String, colecciones, variables, expresiones condicionales, propiedades sobre las clases, etc.

- F_T: Term → Object-Z

La función de traducción F_T recibe un término OCL y retorna un término Object-Z.

Por ejemplo:

Sea "72" un término numérico en OCL, la función de traducción me retorna el término numérico 72 de Object-Z.

F_T (72) = 72

Para el término 72 + 80 de OCL la función nos retorna el término correspondiente Object-Z, que sería:

F_T (72 + 80) = 72 + 80

Función de traducción de declaraciones OCL

OCL tiene varias declaraciones, entre las cuales se encuentran: declaración de variables, de tipos, de operadores (lógicos, relacionales, matemáticos, etc.), declaración de parámetros formales de una operación OCL, declaradores de una iteración sobre colecciones OCL.

- F_D: Declaración → Object-Z

La función de traducción F_D recibe una declaración en OCL y retorna una definición en Object-Z.

Por ejemplo:

Sean "implies", "not", "or", "and" "xor" declaraciones de operadores lógicos en OCL y *on* (cualquier símbolo en OCL), entonces la función retorna los correspondientes operadores en Object-Z.

F_D ("implies") = ⇒

F_D ("not") = ¬

F_D("or") = ∨

F_D("and") = ∧

F_D("xor") = ¬⨟⇔

F_D(*on*) = *on* (el mismo símbolo o nombre, pero en Z)

Otras funciones auxiliares que se utilizan, son:

- toUppercaseFirst: String → String

  Función que toma un String y retorna el mismo con la primera letra en mayúscula.

- names: Declaración → Object-Z

  Función que toma una declaración de parámetros formales OCL y retorna una n-tupla con los tipos de la función, basándose en los tipos del parámetro formal.

- types: Declaración → Object-Z

  Función que toma una declaración de parámetros formales OCL y retorna una los nombres de los parámetros separados por coma, basándose en los tipos del parámetro formal.

Cada una de estas funciones será detallada a continuación.

Función de traducción F_E

Función de traducción F_E que toma una expresión OCL y retorna una expresión Object-Z. Se describe a continuación la traducción de una **expression:**

expression := logicalExpression
logicalExpression :=
" (" logicalExpression ") " |
relationalExpression |
"not" logicalExpression |
logicalExpression binaryLogicalOperator logicalExpression

$F_E($" (" *logicalExpression* ") " $)$ = ($F_E($*logicalExpression*$)$)

F_E(logicalExpression binaryLogicalOperator logicalExpression) = F_E(logicalExpression) F_D(binaryLogicalOperator) F_E(logicalExpression)

F_E("not" *logicalExpression*) = F_D("not") F_E(*logicalExpression*)

relationalExpression :=

"(" relationalExpression ")" |

booleanTerm |

relationalTerm

F_E(relationalExpression) es:

F_E("(" *relationalExpression* ")") = (F_E(*relationalExpression*))

relationalTerm :=

term relationalOperator term

F_E(relationalTerm) es:

F_E(*term relationalOperator term*) = F_T(*term*) F_D(*relationalOperator*) F_T(*term*)

booleanTerm :=

"true" |

"false" |

name

F_E(booleanTerm) es :

F_E("true") = true

F_E("false") = false

F_E(*name*) = *name*

Función de traducción F_T

Función de traducción F_T que recibe un término OCL y retorna un término Object-Z.

La traducción de una letExpression genera un esquema de estado. Este será utilizado para actualizar el esquema de estado de la clase en la cual están definidos.

letExpression := "let" name ["(" [formalParameter] ")"] [":" typeSpecifier] "=" term

F_T (letExpression) lo dividimos en estos posibles casos:

Cómo no tienen parámetros, alcanza con crear una variable de estado que será que luego será agregada a la clase correspondiente. Se crea un esquema de estado que se utilizará para actualizar el esquema de estado correspondiente.

F_T ("let" name ":" typeSpecifier "=" term) =

$[F_D(name) : F_D(typeSpecifier) \mid F_D(name) = F_T(term)]$

Este caso como no tiene definido el tipo, se indica que la variable de estado de la clase es cualquier subtipo de Object.

F_T ("let" *name* "=" *term*) =

$[F_D(name) : \downarrow \text{Object} \mid F_D(name) = F_T(term)]$

Ahora se ve la traducción de una operación definida en una cláusula **Let**. La función retorna un esquema de estado con un variable que es una función si tiene parámetros de entrada, sino la variable toma el tipo de la traducción del tipo de retorno de la operación. La variable tiene el mismo nombre que el de la operación definida en la cláusula "let". Este esquema se utilizará para actualizar el esquema de estado correspondiente.

Este es el caso de una operación sin parámetros de entrada y con tipo de retorno definido.

F_T ("let" *name* "()" ":" *typeSpecifier* "=" *term*) =

$[F_D(name) : F_D(typeSpecifier) \mid F_D(name) = F_T(term)]$

Caso con parámetros de entrada y tipo de retorno definido:

F_T ("let" *name* "(" *formalParameter* ")" ":" *typeSpecifier* "=" *term*) =

[F_D(*name*) : types(*formalParameter*) → F_D(*typeSpecifier*) | ∀ F_D(*formalParameter*) • F_D(*name*)

(names(*formalParameter*)) = F_T(*term*)]

Los dos casos que se muestran a continuación son ambos sin tipo de retorno definido y el primero sin parámetros y el segundo con parámetros de entrada.

F_T ("let" *name* "()" "=" *term*) =

[F_D(*name*) : ↓Object | F_D(*name*) = F_T(*term*)]

F_T ("let" *name* "(" *formalParameter* ")" "=" *term*) =

[F_D(*name*) : types(*formalParameter*) →↓Object | ∀ F_D(*formalParameter*) • F_D(*name*)

(names(*formalParameter*)) = F_T(*term*)]

Donde "types" es una función que retorna una n-tupla con los tipos de la función, basándose en los tipos del parámetro formal y "names" es una función que retorna una los nombres de los parámetros separados por coma, basándose en los tipos del parámetro formal.
Continuamos con la traducción de un término, donde la sintaxis es:

term :=

numericTerm |
booleanTerm |
stringTerm |
collectionTerm |
enumLiteral |
"self" |
ifExpression |
term propertyCallList

Se ve a continuación la función para cada uno de estos casos:

numericTerm :=
"(" numericTerm ")" |
multiplicativeExpression |
numericTerm addOperator numericTerm |
unaryOperator numericTerm |
term "." numericPropertyCall

numericPropertyCall :=
("abs" | "floor"| "round") "()" |
("max"| "min" | "div" | "mod") "(" term ")" |
pathName [timeExpression] [qualifiers] ["("[actualParameter]")"]

F_T("(" *numericTerm* ")") = (F_T(*numericTerm*))

F_T(*numericTerm addOperator numericTerm1*) = F_T(*numericTerm*) F_D(*addOperator*) F_T(*numericTerm1*)

F_T(*unaryOperator numericTerm*) = F_D(*unaryOperator*) F_T(*numericTerm*)

F_T(*term* ".abs()") = | F_T(*term*) |

F_T(*term* ".floor()") = floor (F_T(*term*))

F_T(*term* ".round()") = round(F_T(*term*))

F_T(*term* "."max(" *term1* ")") =max ((F_T(*term*),F_T(*term1*))

F_T(*term* ".min(" *term1* ")") =min(F_T(*term*), F_T(*term1*))

F_T(*term* ".div(" *term1* ")") = F_T(*term*) div F_T(*term1*)

F_T(*term* ".mod(" *term1* ")") = F_T(*term*) mod F_T(*term1*)

multiplicativeExpression :=
"(" multiplicativeExpression ")" |
unaryExpression |
multiplicativeExpression multiplyOperator multiplicativeExpression

unaryExpression := number | name

F_T(“(“ *multiplicativeExpression* “)”) = (F_T(*multiplicativeExpression*))

F_T(*multiplicativeExpression multiplyOperator multiplicativeExpression1*) = F_T(*multiplicativeExpression*) F_D(*multiplyOperator*) F_T(*multiplicativeExpression1*)

F_T (*number*)= *number*

La traducción de un name es la nueva representación en Object-Z, lo que indica *name*, es el nombre de alguna variable o atributo de una clase.

F_T(*name*)= *name*

booleanTerm :=

“true” |

“false” |

name

F_T(*booleanTerm*) = F_E(*booleanTerm*)

stringTerm := string | term “.” stringPropertyCall

stringPropertyCall :=

(“size” | “toUpper” | “toLower”)”()”| “concat(“ term “)” | “substring(“ numericTerm “,” numericTerm”)”

String es una secuencia de caracteres.

F_T(*string*)= *string*

F_T(*term* “.size()”) = # F_T(*term*)

F_T(*term* “.toUpper()”) = toUpper(F_T(*term*))

F_T(*term* “.toLower()”) = toLower(F_T(*term*))

F_T(*term* “.concat(“ *term1* “)”) = F_T(*term*) ⌢ (F_T(*term1*))

F_T(*term* ".substring(" *numericTerm1* "," *numericTerm2*")") = substring (F_T(*term*), (F_T(*numericTerm1*),F_T(*numericTerm2*)))

collectionTerm := literalCollection | name

literalCollection:= collectionKind "{" [collectionItem ("," collectionItem)*] "}"

Recursivamente:

iteralCollection:= collectionKind "{" "}"

iteralCollection:= collectionKind "{" collectionItem"}"

iteralCollection:= collectionKind "{" collectionItem collectionItemList "}"

Donde *collectionItemList* es de la forma: collectionItem (" , " collectionItem)*

collectionItemList:= "," collectionItem

collectionItemList:= ("," collectionItem) collectionItemList

collectionItem :=

term |

term ".." term

collectionKind :=

"Set" |

"Bag" |

"Sequence" |

"Collection"

F_T("Set" "{" *term* "}") = {F_T(*term*)}

F_T("Set" "{" *term* ".." *term1* "}") = F_T(*term*) .. F_T(*term1*)

F_T("Set" "{" *collectionItem* "," *collectionItemList* "}") = F_T("Set" "{" *collectionItem*"}") ∪ F_T("Set" "{" *collectionItemList* "}")

F_T("Bag" "{" *term* "}") = ⟦F_T(*term*)⟧

F_T(“Bag” “{“ *term* “..” *term1* “}”)= items(< x: N | x (F_T(*term*) .. F_T(*term1*))>)

F_T(“Bag” “{“ *collectionItem* “,” *collectionItemList* “}”) = F_T(“Bag” “{“ *collectionItem*”}”) ⊎ F_T(“Bag “ “{“ *collectionItemList* “}”)

F_T(“Sequence” “{“ *term* “}”) = F_D(*collectionKind*)

F_T(“Sequence” “{“ *term* “..” *term1* “}”) = < x: N | x (F_T(*term*) .. F_T(*term1*))>[3]

F_T(“Sequence” “{“ *collectionItem* “,” *collectionItemList* “}”) = F_T(“Sequence” “{“ *collectionItem*”}”) ⌢ F_T(“Sequence” “{“ *collectionItemList* “}”)

El tipo **Collection** es abstracto por lo cual no podría haber un término de tipo **Collection**.

Por ejemplo:

Collection{1,2,3} no es una expresión correcta porque Collection es abstracta.

Esta es la definición de un tipo enumerativo.

enumLiteral :=

name “::” name |

name “::” enumLiteral

Resursivamente queda

enumLiteral := name “::” name

enumLiteral := name “::” enumLiteral

Un ejemplo de un tipo enumerativo es el siguiente:

Enumerativo Sexo en el paquete ejemplo del diagrama UML

[3] http://www.ecs.soton.ac.uk/publications/rj/1994/decsys/gravell/gravell.html

En esta publicación se describe una abreviación para escribir una secuencia por comprensión en Z.

context Persona **inv:**

self.sexo = Sexo::masculino

Invariante OCL en el contexto Persona con un valor enumerativo

La función de traducción ignora la cadena de nombres que se anteponen al valor del enumerativo. Por lo cual la función para este caso queda definida de esta manera:

F_T(*name* "::" *name*) = F_T(*name*)

F_T(*name* "::" *enumLiteral*) = F_T(*enumLiteral*)

F_T ("self") = self

ifExpression :=
"if" logicalExpression "then" (expression | term) "else" (expression | term) "endif"

F_T("if" *logicalExpression* "then" *expression* "else" *expression1* "endif") =
if F_E(*logicalExpression*) then F_E(*expression*)
else F_E(*expression1*)

F_T("if" *logicalExpression* "then" *term* "else" *term1* "endif") =
if F_E(*logicalExpression*) then F_T(*term*)
else F_T(*term1*)

Ahora veamos el caso de una secuencia de llamadas a propiedades. Las propiedades pueden ser llamadas a operaciones o llamadas a atributos.

Como se explicó anteriormente las operaciones que aparecen en una expresión OCL son operaciones de consultas por lo cual en la traducción del diagrama de clases se crearon variables de estado por cada atributo y variables secundarias por cada operación de consulta. Por lo cual podremos traducir a Object-Z en una secuencia de llamadas a variables de estado.

El caso especial es cuando las llamadas a operaciones son sobre colecciones, ya sean Set, Bag, Sequence.

Para la traducción de este tipo de expresiones se utiliza una nueva función auxiliar. Esta función esta definida recursivamente por la cantidad de llamadas a propiedades.

Luego la traducción de este tipo de expresiones resulta de la siguiente manera:

term propertyCallList

F_T(*term propertyCallList*) = add-propertyCall(F_T(*term*), *propertyCallList*)

actualParameter := term (“,” term)*

F_T(*term* “,” *term1*) = F_T(*term*) , F_T(*term1*)

Donde *term1* es de la forma term (“,” term)*

F_T(*term*) se detalló anteriormente.

Función de traducción F_D

Función de traducción F_D que toma una definición en OCL y retorna una definición en Object-Z.

F_D(binaryLogicalOperator)

binaryLogicalOperator := “and” | “or” | “xor” | “implies”

F_D(relationalOperator)

relationalOperator := “=“ | “>“ | “<“ | “>=“ | “<=“ | “<>“

Como operationName esta definido

operationName:= name

La traducción es:

F_D(“implies”) = ⇒

F_D(“not”) = ¬

F_D(“or”) = ∨

F_D(“and”) = ∧

F_D(“xor”) = ¬⨾⇔

F_D(*on*) = *on* (el mismo símbolo o nombre pero en Z)

El símbolo formalParameter esta determinado por

formalParameter := (name “:” typeSpecifier) (“,” (name “:” typeSpecifier))*;

Recursivamente:

formalParameter := (name “:” typeSpecifier)

formalParameter := (name “:” typeSpecifier) “,” formalParameter;

Entonces:

F_D(formalParameter) es:

F_D(*name* " : " *typeSpecifier*) = F_D(*name*) : F_D(*typeSpecifier*)

F_D((*name* ":" *typeSpecifier*) ","*formalParameter*) =

F_D(*name*) : F_D(*typeSpecifier*)

F_D(*formalParameter*)

F_D(pathname "@pre") = F_D(pathname) "@pre"

La traducción de typeSpecifier depende si es un simpleTypeSpecifier o un collectionType, por ello la traducción que quedaría:

typeSpecifier :=

simpleTypeSpecifier |

collectionType

simpleTypeSpecifier :=

"Integer" |

"String" |

"Boolean" |

"Real" |

pathname

F_D(simpleTypeSpecifier) es:

F_D("Integer") =$\mathbb{Z}$

F_D("String") = String

F_D("Boolean") = $\mathcal{B}$

F_D("Real") = $\mathbb{R}$

packageName := pathName

pathName := name | name "::" pathName

F_D(*name*)= *name*

F_D(*name* "::" *pathName*)= F_D(*name*).F_D(*pathName*)

F_D(collectionType) es:

collectionType := collectionKind "(" simpleTypeSpecifier ")"

collectionKind := "Set" | "Bag" | "Sequence" | "Collection"

F_D(*collectionKind* "(" *simpleTypeSpecifier* ")") = F_D(*collectionKind*) F_D(*simpleTypeSpecifier*)

F_D("Set") = $\mathbb{P}$

F_D("Bag") = bag

F_D("Sequence") = seq

F_D("Collection") = seq

returnType := typeSpecifier

multiplyOperator := "*" | "/"

F_D("*") = *

F_D("/") = ÷

unaryOperator := "-"

F_D("-") = -

addOperator := "+" | "-"

F_D("+") = +

F_D("-") = -

La traducción de declarator es:

declarator := name (“,” name)* [“:” simpleTypeSpecifier]”|”

Se renombre esta expresión, para escribirla recursivamente:

nameList:= name (“,” name)

Recursivamente:

nameList:= name

nameList:= name “,” nameList

Entonces el símbolo no terminal declarador queda definido así:

declarator := name [“:” simpleTypeSpecifier]”|”

declarator := name “,” nameList [“:” simpleTypeSpecifier]”|”

Los posibles casos que recibe la función F_D son cuatro:

F_D(*name*”|”) = *name*

F_D(*name*”:” *simpleTypeSpecifier* “|”) = *name*

F_D(*name* “,” *nameList* “|”) = *name*, F_D(*nameList* “|”)

F_D(*name* “,” *nameList* “:” *simpleTypeSpecifier* “|”) = *name* , F_D(*nameList* “|”)

qualifiers := “[“ actualParameter “]”

F_D(“[“*actualParameter* “]”) = [F_T(*actualParameter*)]

propertyCall :=

pathName [timeExpression] [qualifiers] [“(“[actualParameter]”)”]

timeExpression := “@” “pre”

Este es el caso de un atributo que referencia al estado anterior de aplicar una operación. Este caso ya se había detallado arriba:

F_D (*pathName* "@" "pre")= F_D (*pathname*) "@pre"

Es un elemento de una colección, calificado por uno o varios atributos

F_D (*pathName qualifiers*)= qualifier(F_D (*pathname*)) **(**F_E(*qualifiers*)**)**

Donde la función qualifier retorna el nombre con una "q" al final, porque esta es la variable de estado de Object-Z que se utilizan para las asociaciones calificadas:

qualifier (*name*) = F_D (*name*)+"q"

F_D (*pathName* "()") = F_D (*pathname*)

F_D (pathName "("*actualParameter*")") = F_D (*pathname*) **(**F_T(*actualParameter*)**)**

Estos tres casos no son correctos, porque el "@pre" solo aparece en nombre de atributos, por lo cual la función no esta definida:

pathName timeExpression qualifiers

pathName timeExpression "()"

pathName timeExpression "("actualParameter")"

Ahora se muestra como, la función F_D hace la traducción de:

signatureOp:= **operationName "("[formalParameter] ")" [":" returnType]**

Este resultado se utiliza para buscar en la clase Object-Z la operación con esa signatura, por lo cual se devuelve la signatura en Object-Z:

F_D (*operationName* "()")= toUppercaseFirst (F_D (*operationName*))

F_D (*operationName* "()" ":" *returnType*) = toUppercaseFirst (F_D (*operationName*)) : (F_D (*returnType*))

F_D (operationName "("formalParameter ")") = toUppercaseFirst (F_D (operationName)) (F_D (formalParameter))

F_D (*operationName* "("*formalParameter* ")" ":" *returnType*) = toUppercaseFirst (F_D (*operationName*))) (F_D (*formalParameter*)) : (F_D (*returnType*))

Función names

Función que retorna una n-tupla con los tipos de la función, basándose en los tipos del parámetro formal

Función types

Función que retorna una los nombres de los parámetros separados por coma, basándose en los tipos del parámetro formal.

Función lookingFor-pre

Esta función toma una expresión OCL y retorna la lista de atributos que aparecen con el símbolo "@pre". Por ejemplo, si tenemos la siguiente expresión:

saldo=saldo@pre +cant

La función retorna la siguiente lista: {saldo}

Función rename

La siguiente función toma una lista de atributos y retorna la lista de renombres. Continuando con el ejemplo anterior, quedaría

rename({saldo})=[saldo/saldo', saldo@pre/saldo]

Función add-propertyCall(termOZ, propertyCallList)

Se muestra a continuación la definición de la función add-propertyCall, que recibe un termino Object-Z, una lista de llamadas a propiedades del modelo UML/OCL y retorna un termino Object-Z. Recordamos como es la sintaxis del propertyCallList

propertyCallList := call
propertyCallList := call propertyCallList
call:= (“.” propertyCall | “→” collectionPropertyCall)

Se puede observar que hay tres casos, el recursivo y dos casos base, es decir que:
term”.” propertyCall
term”→” collectionPropertyCall
term calls propertyCallList

A continuación se muestra el caso recursivo:

term call propertyCallList

add-propertyCall(termOZ, calls propertyCallList)=
let term=add-propertyCall(*termOZ*, *calls*)
in add-propertyCall(*term*, *propertyCallList*)

Casos no recursivos

add-propertyCall(*termOZ*, “.” *propertyCall*)

Un término Object-Z correcto para aplicarle una propiedad, puede ser una variable, un atributo o una operación. Entonces, la función queda definida de la siguiente manera:

add-propertyCall(*termOZ*, “.” *propertyCall*)= *termOZ*. F_D (*propertyCall*)

add-propertyCall(terminoOZ, “->“ collectionPropertyCall)

Se muestra a continuación la traducción de una llamada a una operación sobre colecciones. Recordemos la sintaxis de ***collectionPropertyCall***

collectionPropertyCall:=
(“forAll” | “select” | “reject” | “exists” | “any” | “one”) (blockBoolExpression) |
collect blockOneExpression |
(“size” | “notEmpty” | “isEmpty”) “()” |
(“includes”|”excludes”) “(“ term “)” |
(“includesAll” | “excludesAll”) “(“ collectionTerm “)”

blockBoolExpression := “(“ [declarator] logicalExpression “)”

blockExpression := “(“ [declarator] [actualParameter] “)”

blockOneExpression := “(“ [declarator] term “)”

declarator := listEnumerate_of name) [“:” simpleTypeSpecifier]”|”
name (“,” name)* [“:” simpleTypeSpecifier]”|”

Por el momento consideramos que los términos son de la clase Collection por lo cual será traducido como un Set. El problema está que estáticamente no se puede saber si es un Set, Bag o Secuencia.

add-propertyCall (*termOZ,* “→” “forAll” “(“ *declarator logicalExpression* “)”) = $\forall$ F_D(*declarator*) $\in$ *termOZ* • F_E(*logicalExpresion*)

add-propertyCall (*termOZ,* “→” “select” “(“ *declarator logicalExpression* “)”) = {F_D(*declarator*) $\in$ *termOZ* | F_E(*logicalExpresion*)}

add-propertyCall (*termOZ,* “→” “reject” “(“ *declarator logicalExpression* “)”) = {F_D(*declarator*) $\in$ *termOZ* | $\neg F_E$(*logicalExpresion*)}

add-propertyCall (*termOZ,* “→” “exists” “(“ *declarator logicalExpression* “)”) = $\exists$ F_D(*declarator*) $\in$ *term*OZ • F_E(*logicalExpresion*)

add-propertyCall (*termOZ,* “→” “any” “(“ *declarator logicalExpression* “)”) =

$\# \{ F_D(declarator) \in termOZ \mid F_E(logicalExpresion) \} > 0$

add-propertyCall (*termOZ,* “→” “one” “(“ *declarator logicalExpression* “)”) $= 1 = \#\{F_D(declarator) \in termOZ \mid F_E(logicalExpresion)\}$

add-propertyCall (*termOZ,* “→” “collect” “(“ *declarator term1* “)) = $[\![F_T(term1) \mid F_D(declarator) \in termOZ]\!]$

add-propertyCall (*termOZ*, “→” “size” “()”) = #*termOZ*

add-propertyCall (*termOZ*, “→” “notEmpty” “()”) = #*termOZ* > 0

add-propertyCall (*termOZ*, “→” “isEmpty” “()”) = # *termOZ* = 0

add-propertyCall (*termOZ*, “→” “includes” “(“ *term1* “)”) = F_T (*term1*) ∈ *termOZ*

add-propertyCall (*termOZ,* “→” “excludes” “(“ *term1* “)”) =F_T(*term1*) ∉ *termOZ*

add-propertyCall (*termOZ*, “→” “includesAll” “(“ *collectionTerm* “)”) = F_T (*collectionTerm*) ⊆ *termOZ*

add-propertyCall (*termOZ*, “→” “excludesAll” “(“ *collectionTerm* “)”) = (F_T (*collectionTerm*) ∩ *termOZ*)= ∅

Función enrich-condition

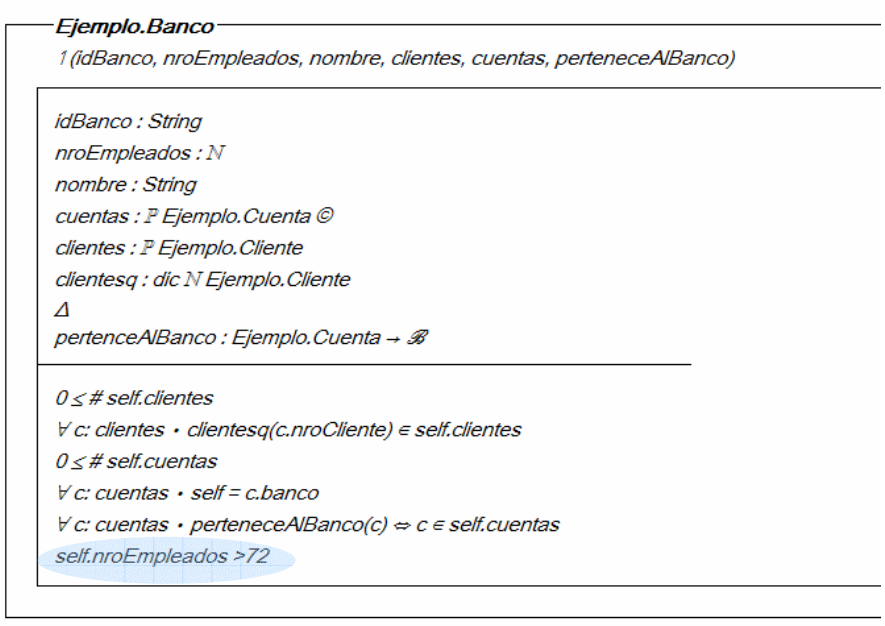
Se definió la función "enrich-condition" con cuatro parámetros. El primero es una especificación Object-Z, el segundo, el nombre de una clase Object-Z, el tercero es la signatura de una operación Object-Z, y el último, una expresión. Esta función retorna una nueva especificación Object-Z. Esta especificación el la modificación de la que recibió, a la cual se le agrega un predicado en el esquema de estado de la operación recibida.
Lo que realiza esta función es buscar la clase y la operación en la especificación recibida para agregarle la expresión recibida como un predicado más en el esquema de estado de la operación.

Función enrich-inv

Se definió la función "enrich-inv" con tres parámetros. El primero es una especificación Object-Z, el segundo, el nombre de una clase Object-Z, y el último, una expresión. Esta función retorna una nueva especificación Object-Z. Esta especificación el la modificación de la que recibió, a la cual se le agrega un predicado en el esquema de estado de la clase recibida.
Lo que realiza esta función es buscar la clase en la especificación recibida y agregarle la expresión recibida como un predicado más en el esquema de estado de la clase.
Ejemplo:
Se tienen el siguiente invariante en OCL

```
package ejemplo
context Banco inv cantEmpleados:
    self.nroEmpleados > 72
endpackage
```

Luego de la traducción de expresión "b.nroEmpleados > 72" a Object-Z, la función enrich-inv agrega ese resultado a la clase ejemplo.Banco



Invariante CantEmpleados agregado en la clase Object-Z correspondiente

Función enrich-with-properties

Se definió la función "enrich-with-properties" con tres parámetros. El primero es una especificación Object-Z, el segundo, el nombre de una clase Object-Z, y el último, una expresión **(letExpression)**$^+$. Esta función retorna una nueva especificación Object-Z. Esta especificación el la modificación de la que recibió, a la cual se le agregan las propiedades de traducir cada "**letExpression"**.

Re-escritura recursiva del símbolo letExpression$^+$:

(letExpression)$^+$**= letExpression**

(letExpression)$^+$**= letExpression (letExpression)**$^+$

enrich-with-properties(S, className, letExpression (letExpression)$^+$*)=*

let S'= enrich-with-property(*S, className,* F_T(*letExpression*))=

in *enrich-with-properties(S', className, (letExpression)*$^+$*)*

Función enrich-with-property y enrich-with-property-visible

Se definieron las funciones "enrich-with-property" y "enrich-with-property-visible", ambas con tres parámetros y una especificación Object-Z de retorno. El primer parámetro es una especificación Object-Z, el segundo, el nombre de una clase Object-Z, y el último, un esquema de estado. El esquema de estado que reciben, es un esquema especial. Este tiene una variable y un predicado, por lo cual estas funciones modifican la especificación que recibieron, agregandolé una nueva variable secundaria y un predicado (obtenido del esquema recibido) en el esquema de estado de la clase, también recibida como segundo parámetro. En otras palabras toman el esquema de estado, buscan la clase en la especificación recibida y actualizan el esquema de estado de la clase hallada.

La diferencia entre "enrich-with-property" y "enrich-with-property-visible" es que la segunda agrega a las lista de visibilidad el nombre de la variable.

Capítulo 4

IMPLEMENTACIÓN

En este capítulo se detalla la arquitectura de la herramienta desarrollada y algunas decisiones de diseño.

La herramienta fue implementada en la plataforma Eclipse. La elección de esta plataforma fue por la posibilidad de generar plug-ins que enriquezcan el ambiente con una funcionalidad adicional hace posible la combinación por parte del usuario de distintos plug-ins que le permitan formar el ambiente de desarrollo que mejor se ajuste a los requerimientos de cada proyecto en particular. Esto es posible por el ambiente de desarrollo integrado (IDE) como es eclipse.

Eclipse

Eclipse es un ambiente de desarrollo integrado (IDE) de código abierto de propósito general, construida sobre un mecanismo para descubrir, integrar, y ejecutar módulos llamados plug-in. En la plataforma Eclipse, un plug-in es la unidad más pequeña de una función que puede ser desarrollada y entregada de manera separada. El modelo de interconexión es simple: un plug-in declara uno o más puntos de extensión, y a su vez puede contribuir con extensiones a uno o más puntos de extensión de otros plug-ins. Excepto por un pequeño núcleo llamado Platform Runtime, toda la funcionalidad de la plataforma Eclipse esta situada en plug-ins. Ver Figura 1. Estas características hacen de Eclipse el ambiente ideal para el desarrollo de herramientas altamente integradas.

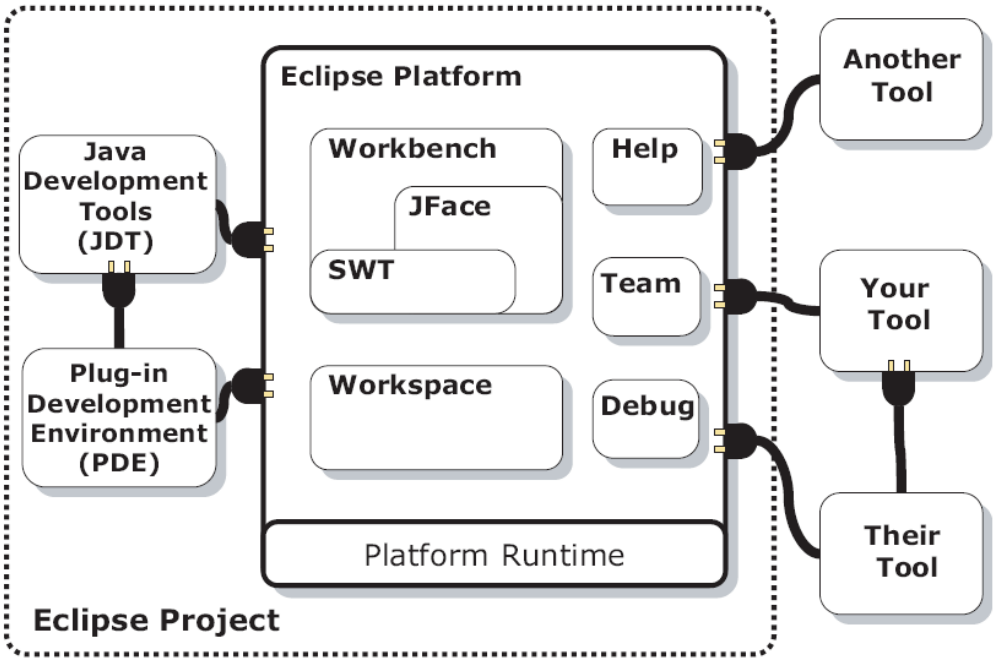


Figura 1: Arquitectura de Eclipse

Arquitectura del Editor

La arquitectura del editor se basa en una serie de plug-ins con funcionalidades diferentes. Desde un nivel abstracto, puede ser visto como una herramienta que extiende la plataforma eclipse y que tiene como requisitos a los plug-ins EMF y UML2, externos a la herramienta y a la plataforma eclipse. Ver Figura 2.

Si bien existen posibles variantes en cuanto a esta arquitectura, se decidió que este esquema permitiría poder separar las funcionalidades específicas en plug-ins que luego pueden ser usados en otros contextos. Por ejemplo este plug-in podría se utilizado en alguna herramienta que diagrame modelos UML y utilice UML2 como metamodelo.

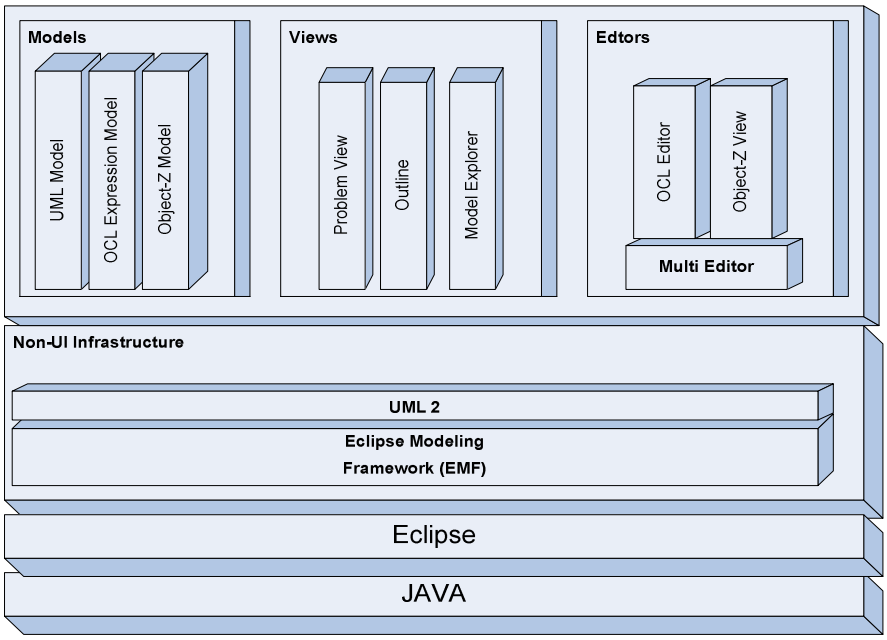


Figura 2: Arquitectura de OCL Editor

Otro beneficio de esta arquitectura es que este conjunto de plug-ins podrá ser incluido en la herramienta llamada ePlatero. El objetivo general de este proyecto es contribuir al mejoramiento de los procesos de desarrollo de software, a través del análisis del paradigma de desarrollo conocido como MDD (Model Driven Development) integrado con la aplicación de métodos formales y soportados por herramientas automáticas.

Para este trabajo se realizo una aplicación para usuario final creada con la API Rich-Client Platform (RCP) de Eclipse. Se trata de un framework a partir del cual se puede construir aplicaciones sobre la base de Eclipse. Es decir, proporcionan un Eclipse "Vacío" que se puede modificar para crear aplicaciones.

Plug-ins

ar.edu.unlp.info.sol.eplatero.oclEditor

Necesario para unir todos los plug-ins que forman el editor de OCL

ar.edu.unlp.info.sol.eplatero.oclEditor.core

Contiene el metamodelo UML y OCL, para representar el modelo UML y las expresiones OCL como objeto. Es el plug-in que provee la implementación de la función de traducción.

ar.edu.unlp.info.sol.eplatero.oclEditor.coreui

Provee las clases que implementan el editor OCL.

ar.edu.unlp.info.sol.eplatero.oclEditor.explorer

Implementación del explorador.

ar.edu.unlp.info.sol.eplatero.core

Plugins encargado de mantener las instancias de los editores sincronizados.

ar.edu.unlp.info.sol.eplatero.oclEditor.model

Modelo utilizado en representar los archivos que se encuentran en el espacio de trabajo.

ar.edu.unlp.info.sol.eplatero.oclEditor.model.edit

Provee las clases necesarias para editar el metamodelo, por ejemplo "comandos" y también clases utilizadas en el explorador y en la vistas de propiedades.

ar.edu.unlp.info.sol.eplatero.oclEditor.rcp

Configuración de RCP framework para obtener el editor de OCL final.

Este plug-in no es necesario se el editor de OCL se quiere utilizar para agregar funcionalidad a otra aplicación de modelado realizada en el ambiente de Eclipse.

En la figura siguiente se puede observar las dependencias de estos plug-ins. Las dependencias son muy importantes cuando se quieren reutilizar los plug-ins en otro contexto u otra aplicación.

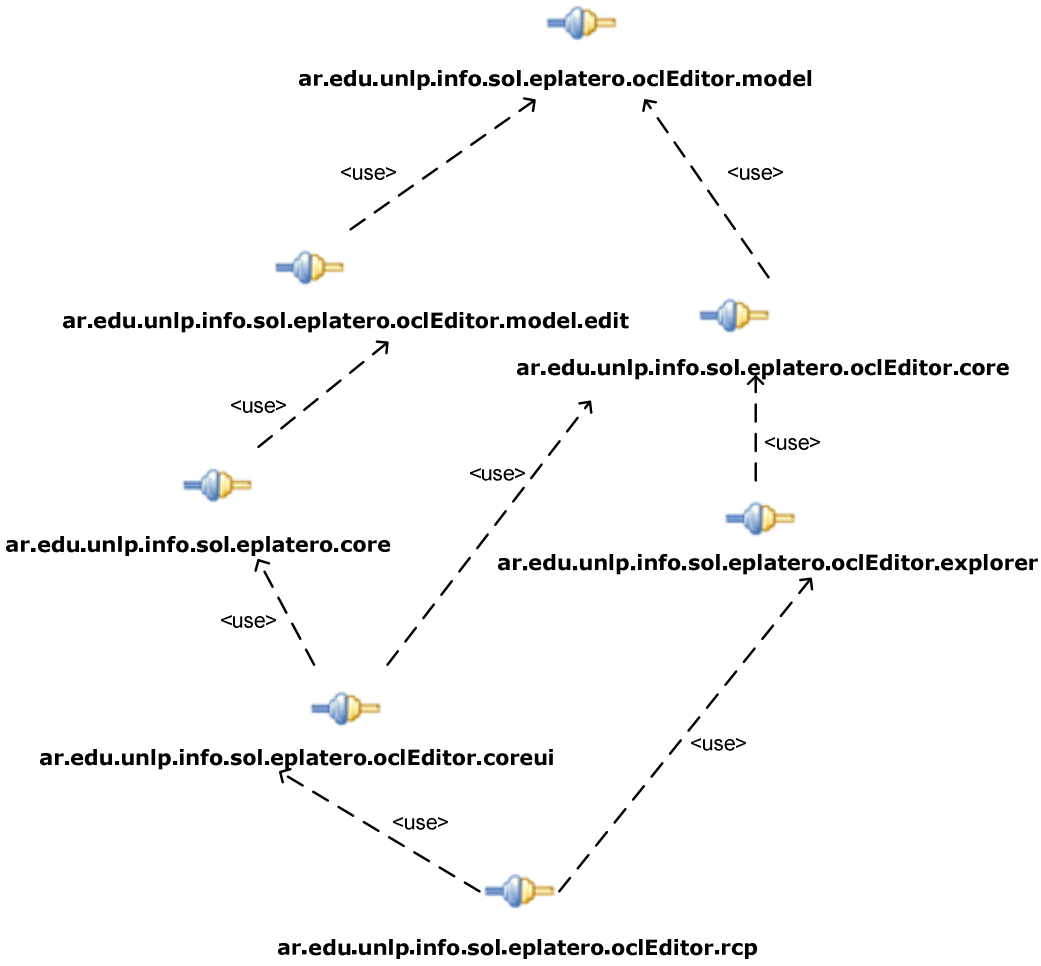


Figura 3: Dependencias

Diseño

En esta sección se describirá el diseño realizado para implementar la función de traducción detalladamente en el capítulo III. Esta sección quiere dejar constancia que la función de traducción de UML-OCL a Object-Z puede ser implementada fácilmente siguiendo los detalles de la definición de $\mathcal{F}$.

El plug-in que contiene la implementación de la función de traducción $\mathcal{F}$ es, **ar.edu.unlp.info.sol.eplatero.oclEditor.core**. Este es el plug-in mas importante a la hora de describir el diseño e implementación del este trabajo.

Hay varios elementos principales en este diseño:

- ✓ Metamodelo UML
- ✓ Metamodelo OCL
- ✓ Especificación Object Z
- ✓ Función de traducción $\mathcal{F}$.

Como Metamodelo UML se utilizo el metamodelo de UML1.5 que es usado en la definición de las expresiones OCL que este editor soporta.
Se definió un modelo para representar las expresiones OCL y un modelo para representar las expresiones y la especificación Object Z. Ver siguiente figura 1.

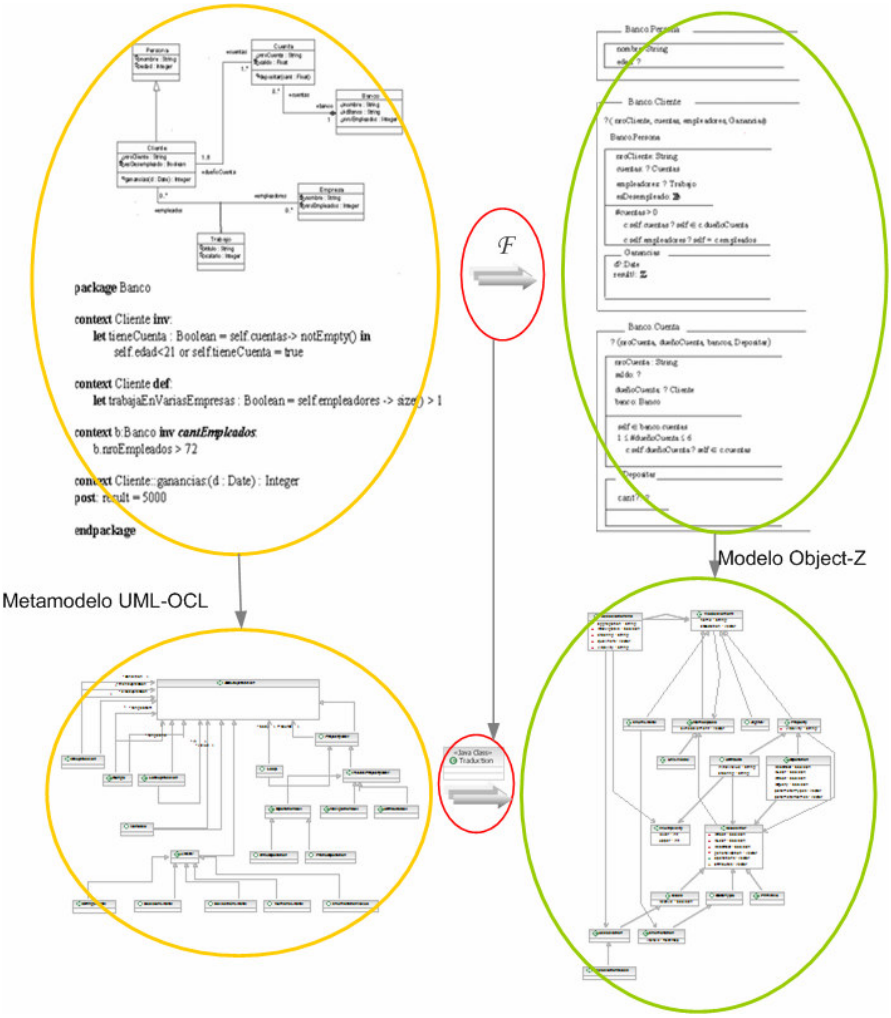


Figura 4: Representación de los modelos

Metamodelo UML

Como se menciono anteriormente se utilizo el metamodelo de UML versión 1.5 para esta implementación. Como esta herramienta no soporta un editor grafico para diagramas de clase UML, se decidió importar este desde un herramienta case externa. Se eligió usar UML2 por las siguientes razones.

El proyecto UML2 (un Eclipse Tools sub-project) es una implementación basada en EMF (Eclipse Modeling Framework) del metamodelo UML™ 2.x para la plataforma Eclipse. Este proyecto provee la implementación del metamodelo UML y tiene toda la capacidad de crear modelos UML y persistirlos. Herramientas tales como RSA (Rational Software Architect) or

RSM (Rational Software Modeler) permiten exportar sus modelos UML a este formato. UML 2.0 soporta todos los elementos de modelado de UML versión 1.5 y nuevos elementos. Esta implementación solo se toman los elementos que aparecen el la versión 1.5 y los demás serán ignorados. Esto permite que en un trabajo futuro esta herramienta sea actualizada rápidamente para soportar modelos de UML basados en la versión 2.0.

Metamodelo OCL

Se creo un metamodelo para representar las expresiones OCL como objetos, de esta menera facilita la traducción a Object Z.

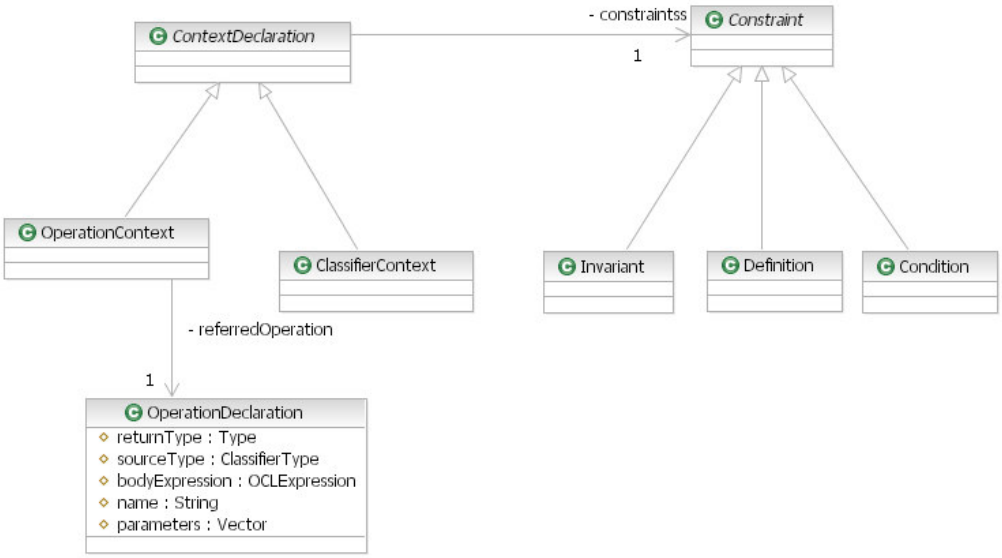


Figura 5: OCL Metamodelo – Representación del contexto

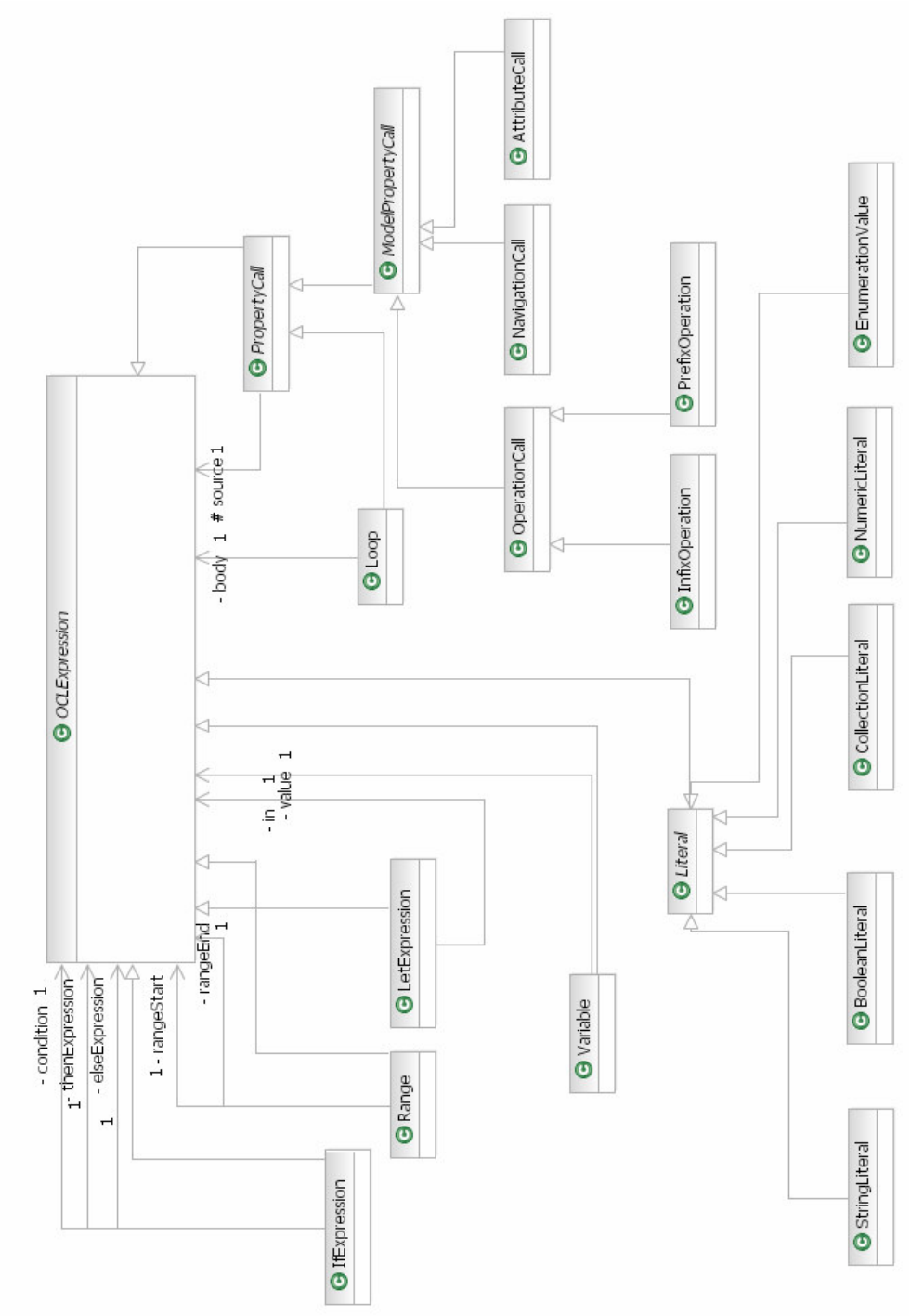


Figura 6: OCL Metamodelo – Representación de las expresiones OCL

Estos objetos son creados a partir de un archivo con expresiones OCL. El parser toma el archivo como entrada y genera una colección de objetos que representan las expresiones escritas con OCL. Estas son las que recibirá la función de traducción junto al modelo UML y las transformara a una especificación Object Z.

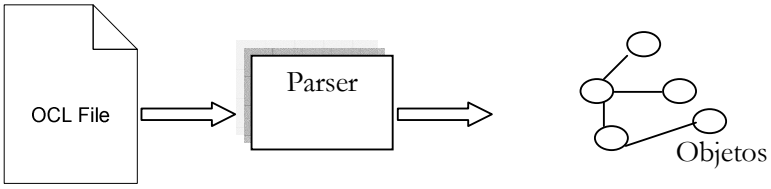


Figura 7: Funcionamiento del Parser

Para desarrollar el parser se utilizó Java Compiler Compiler (JavaCC) [JAVACC]. Javacc es un generador de parser que se puede utilizar en las aplicaciones Java. Se describe la gramática que el parser soportará en archivo ".jj". Este archivo se emplea para generar el parser. La gramática fue descripta en el capítulo 2. El parser de OCL generado por javacc transforma el texto de entrada en el modelo a instancias del metamodelo de OCL detallado anteriormente.

Modelo Object Z

Se creo un modelo para representar las expresiones Object Z.

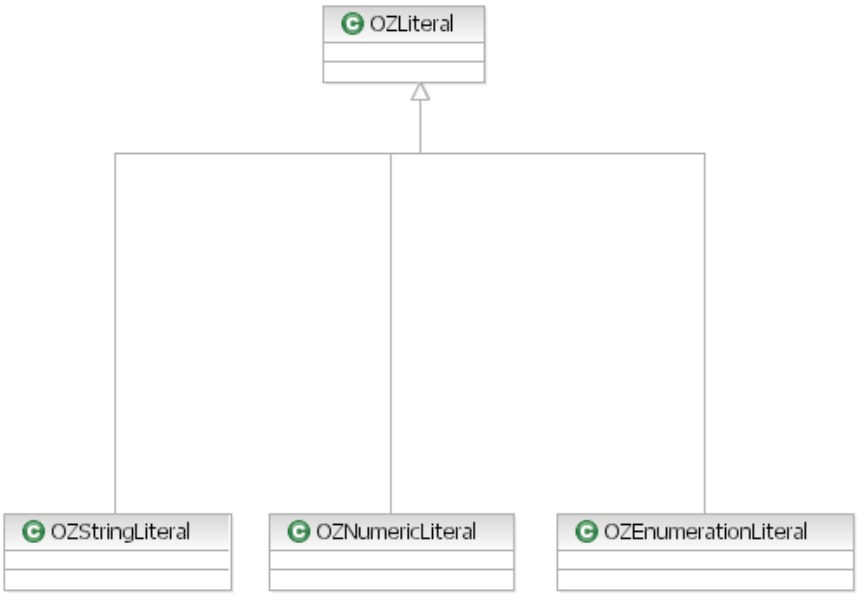


Figura 8: Representación de un literal Object-Z

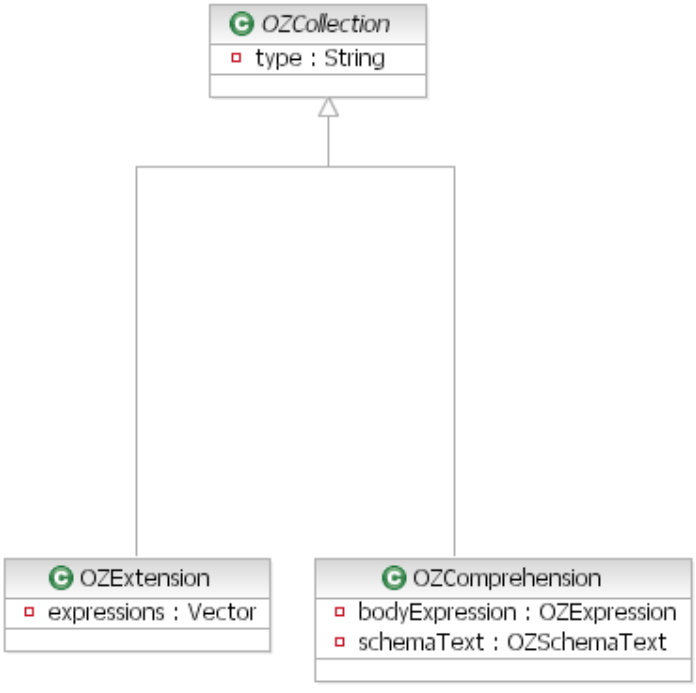


Figura 9: Representación de colecciones Object-Z (Set, Bag, Seq)

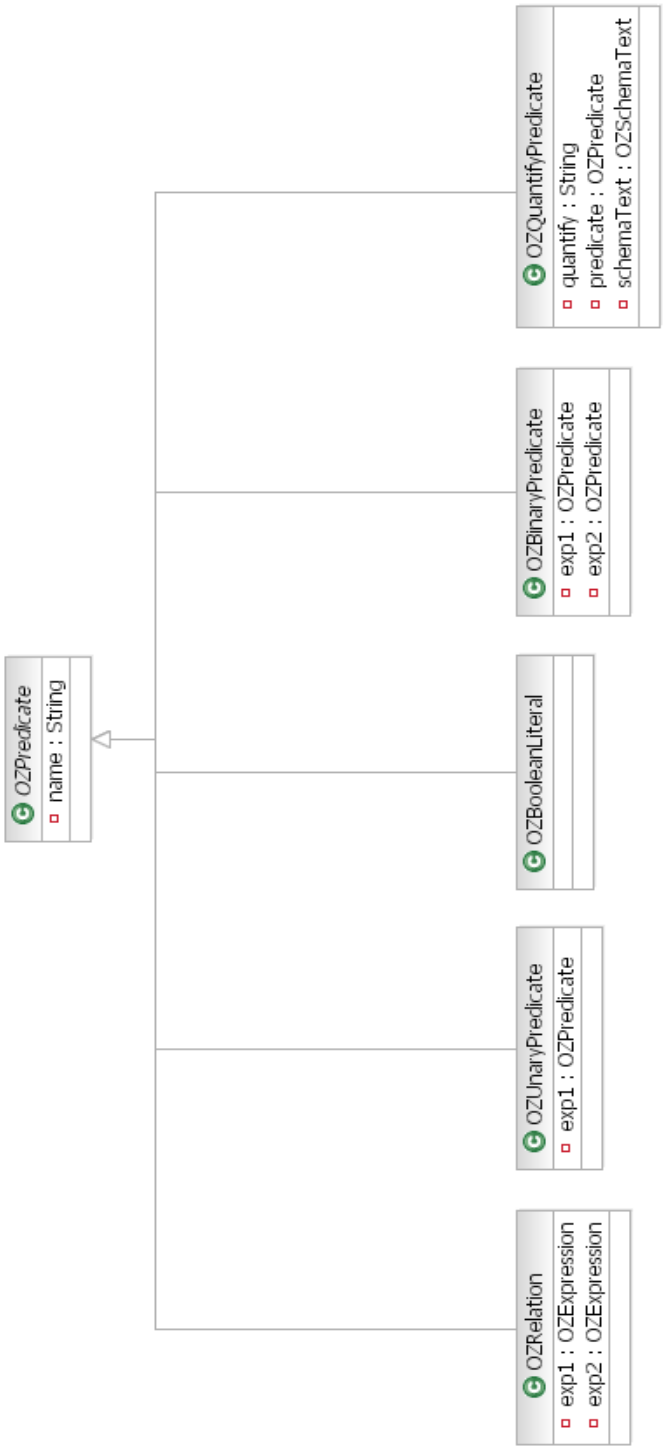


Figura 10: Representación de predicados

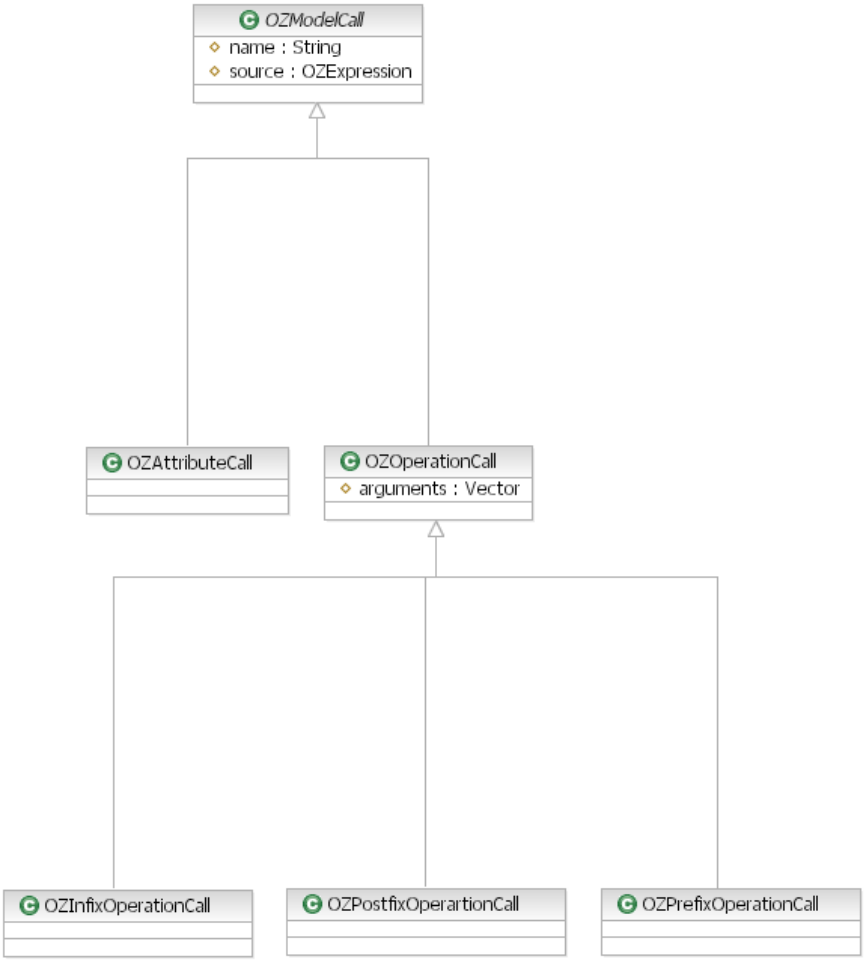


Figura 11: Representación de invocación a Operaciones o atributos de una clase Object-Z

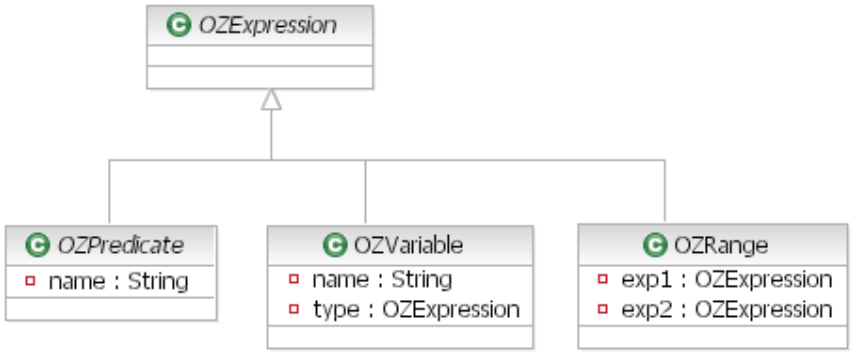


Figura 12: Representación de una Expresión Object-Z

Implementación de la función de traducción $\mathcal{F}$.

Se creo una clase en Java, llamada "Tanslation" con un método llamado "translateOZ" que implementa la función F. Por casa función auxiliar se definió un método en la misma clase.

La clase "Traduction" tiene un el método "translateOZ" que recibe el modelo UML y las expresiones OCL y retorna una especificación Object- Z.

Se invoca de la siguiente manera:

OZSpecification specification= traduction.translateOZ(model, oclFile);

Recordando la definición del capitulo anterior, la función $\mathcal{F}$ se había definido así:

$$\mathcal{F}\,(S, umlModel \cup oclFile) = \textbf{let } S' = \mathcal{F}_{\text{uml}}\,(S, umlModel)$$
$$\textbf{in } \mathcal{F}_{\text{ocl}}\,(S', oclfile)$$

El método translateOZ(model, oclFile) esta definido de la siguiente manera:

```
public OZSpecification translateOZ( UMLModel model, OCLFile oclFile)
 {
//Traduce el modelo UML a una especificacion Object-Z
OZSpecification specification = translateOZ(model);
// Toma la especificación Object-Z y la enriquece con la traducción de las expresiones OCL.
return tranlateOZ(specification,oclFile);
}
```

Visualización de la especificación Object-Z

Para la visualización de la especificación Object-Z generada por el proceso de traducción, se utilizó XML. Esta técnica esta basada sobre el trabajo "Web Display Formal Specification in ZML" [Object-Z Web, Object-Z Web II], el cual define la sintaxis de la estructura XML para escribir especificaciones formales, en particular, Object-Z.

Dado que el proceso de traducción, es decir, la implementación de la función $\mathcal{F}$ retorna un conjunto de objeto que representa la especificación Object-Z, se utilizó el patrón "Visitor" para recorrer la estructura y generar el archivo que visualiza la especificación.

Usar este patrón nos permite una fácil y rápida extensión de la herramienta para generar otra representación del modelo Object-Z, por ejemplo, generar un archivo Latex u algún otro tipo de archivo que sea necesario para exportarlo a otra herramienta.

Patrón "Visitor"

La idea básica es que se tiene un conjunto de clases elemento que conforman la estructura de un objeto. Cada una de estas clases elemento tiene un método aceptar (accept()) que recibe al objeto visitador (visitor) como argumento. El visitador es una interfaz que tiene un método visitor diferente para cada clase elemento; por tanto habrá implementaciones de la interfaz

visitor de la forma: visitorClase1, visitorClase2... visitorClaseN. El método accept de una clase elemento llama al método visit de su clase. Clases concretas de un visitador pueden entonces ser escritas para hacer una operación en particular.

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 envío doble (en inglés éste término se conoce como Double-Dispatch) en un lenguaje convencional orientado a objetos de envío único (Single-Dispatch), como es Java.

El patrón visitor también especifica cómo sucede la interacción en la estructura del objeto. En su versión más sencilla, donde cada algoritmo necesita iterar de la misma forma, el método accept de un elemento contenedor, además de una llamada al método visitor, también pasa el objeto visitor al método accept de todos sus elementos hijos.

Este patrón es ampliamente utilizado en intérpretes, compiladores y procesadores de lenguajes, en general.

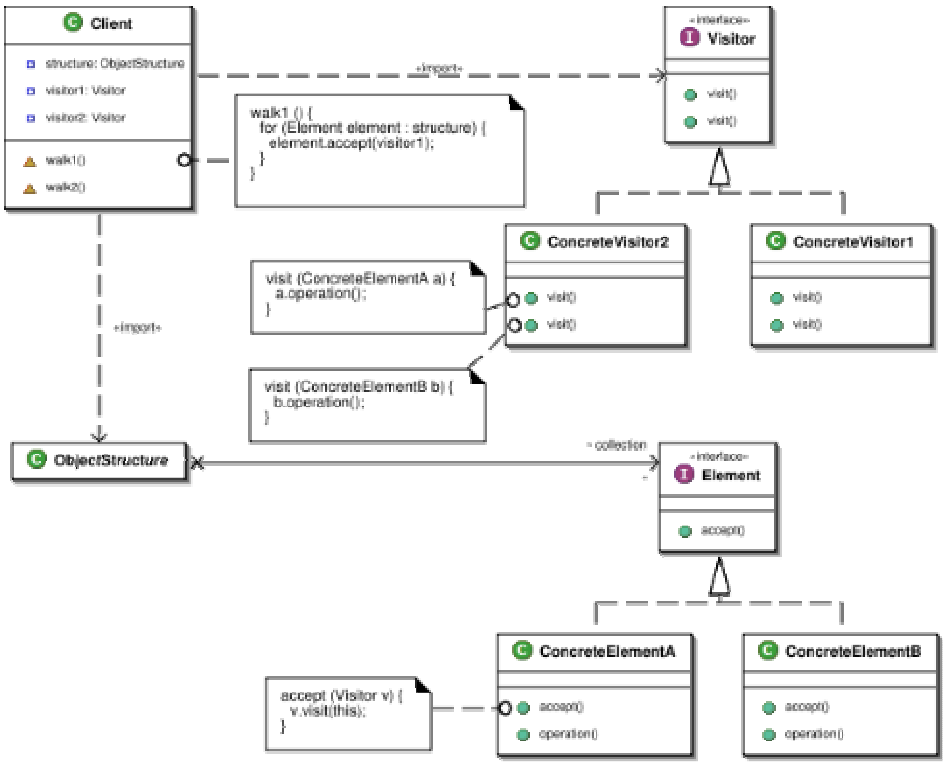


Figura 13: Patrón "Visitor"

Este trabajo tenía como primera intención utilizar Z-Eves para evaluar especificaciones escritas en Object-Z. En unas de las etapas de estudio de este trabajo se detectó que aún no se pueden evaluar especificaciones escritas en Object-Z en esta Z-Eves.

Usar el patrón Visitor, nos garantiza porder generar un archivo Z-Eves cuando este permita evaluar Object-Z o generar otro archivo para algún otro evaluador, fácilmente. Esto se podría hacer agregando una nueva subclase de la clase "Visitor" que genere el archivo necesario, tal como ahora se genera el archivo XML para visualizar la especificación en la herramienta.

Editor OCL

Para finalizar éste capítulo se describe las partes de esta herramienta.

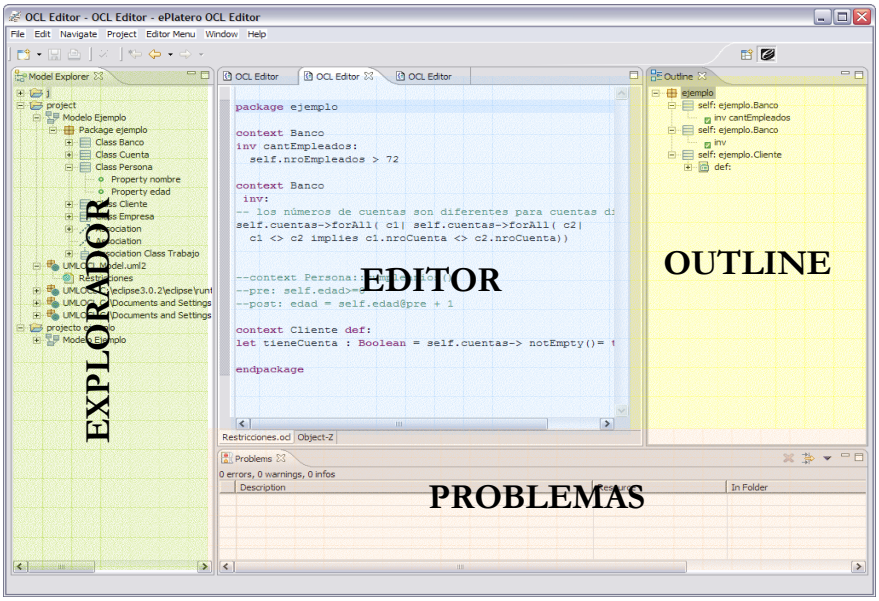


Figura 14: OCL Editor

En la perspectiva de esta herramienta se puede observar 4 partes:

- Explorador

Esta vista muestra el modelo UML sobre el cual se escriben las expresiones OCL y el archivo que contiene tales expresiones.

Esta vista fue creada porque esta herramienta no permite crear diagramas UML y era necesario poder visualizar cual era el modelo sobre el que se escribirán las restricciones OCL.

También visualiza todos los proyectos que se encuentran en el ambiente de trabajo, comúnmente llamado "workspace".

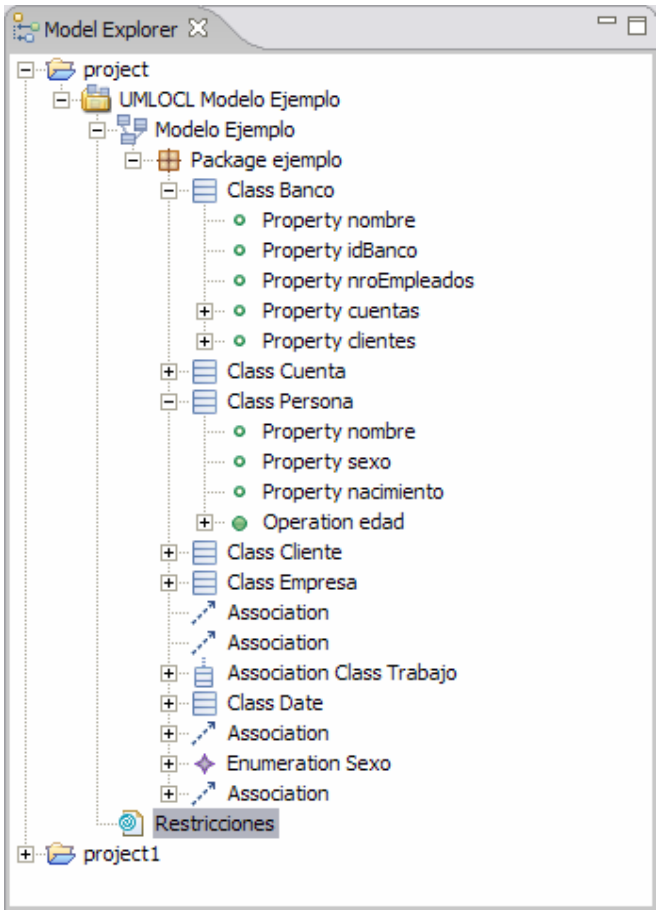


Figura 15: Explorador

- Editor

En esta vista se puede ver el editor de expresiones OCL y el visualizador de Object-Z

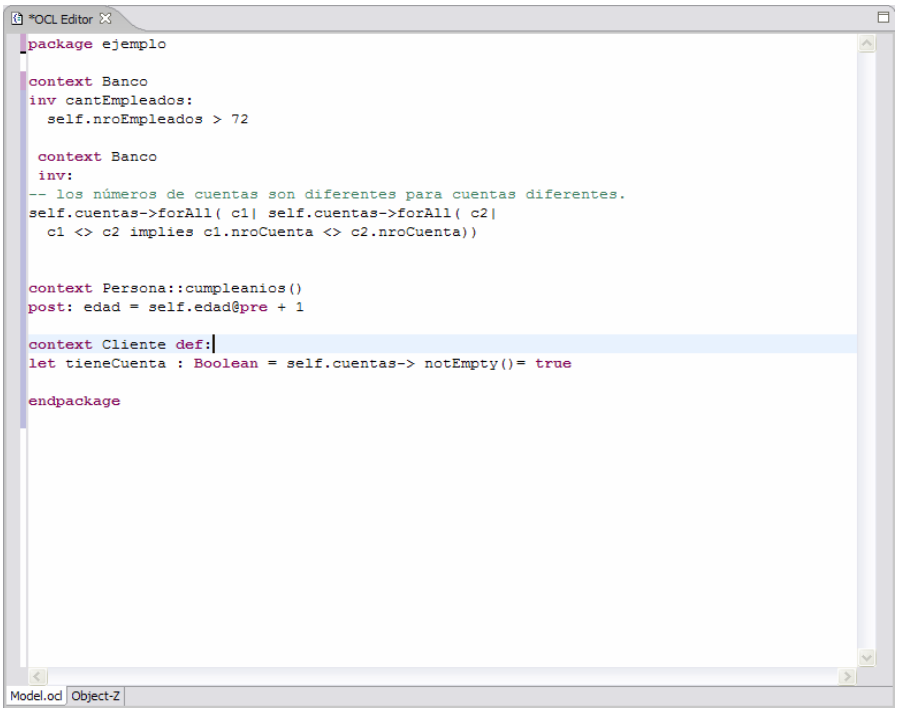


Figura 16: OCL Editor

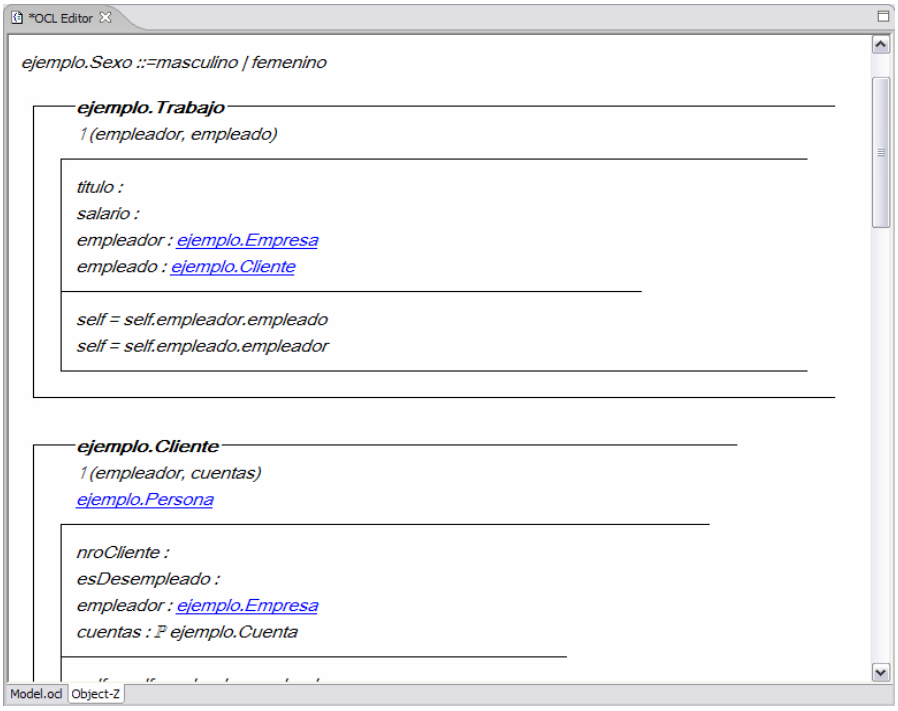


Figura 17: Visualizar Object-Z

- Outline

Esta vista es muy utilizada para ver una abstracción del documento que esta siendo editado, permitiendo encontrar expresiones rápidamente en archivos extensos.

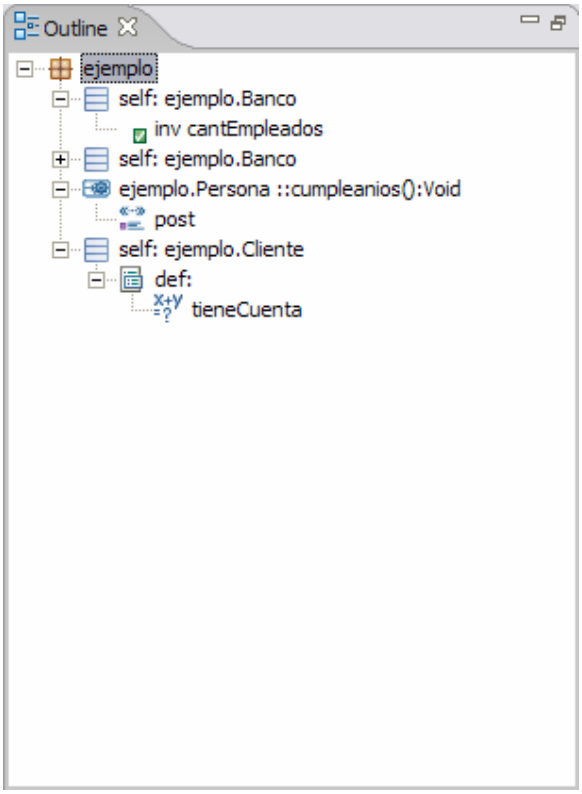


Figura 18: Vista Outline

- Problemas

Esta vista se utiliza para mostrar los errores que ocurren en el momento de edición del archivo. Por ejemplo si ocurre algún tipo de error sintáctico de las expresiones OCL.

Los errores son mostrados con una marca en el editor, en la linea donde ocurre el error y una descripción en la parte inferior de la herramienta (Problems View).

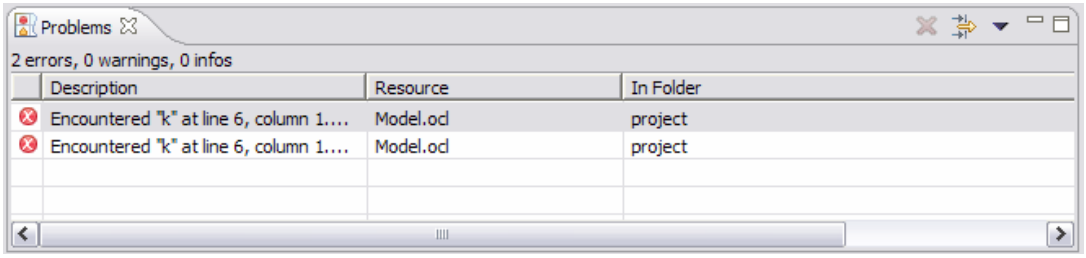


Figura 19: Vista de Problemas

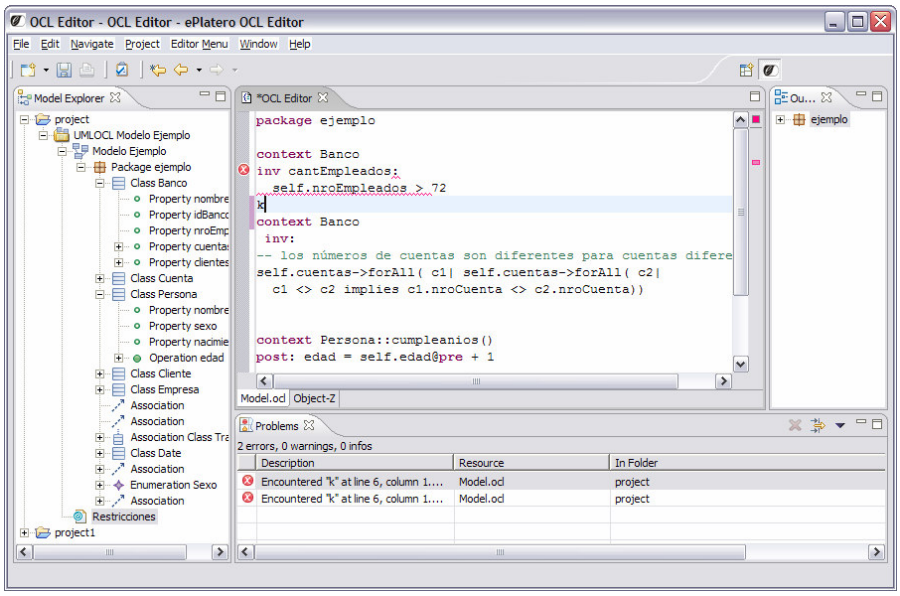


Figura 20: Vista de Problemas y marca de error en el archivo.

Capítulo 5

CONCLUSIONES Y TRABAJOS FUTUROS

Hemos presentado la traducción de la información lógica contenida un diagrama de clase UML y en las expresiones OCL en una especificación Object-Z.

OCL está actualmente en una etapa de desarrollo dentro del un proceso de estandardización dentro del OMG; este trabajo es un intento de acompañar este proceso con el propósito de solucionar algunas preguntas abiertas y explorar alternativas a este lenguaje de diseño. Además este trabajo provee una herramienta para soportar la edición de una especificación OCL y la automatización de la transformación de un modelo UML (importado desde un archivo XML) y un Archivo OCL a una especificación Object-Z

Finalmente, la traducción definida en esta tesis hace posible el uso de herramientas automáticas de verificación y validación de especificaciones Object-Z, como complemento o alternativa a las herramientas de validación de UML/OCL. La ventaja reside en que las herramientas de Object-Z son potencialmente más ricas ya que se orientan a la prueba de teoremas (theorem proving), mientras que los evaluadores UML/OCL actuales se limitan al chequeo de modelos (model checking)

Este trabajo podría ser ampliado para incluir la traducción de expresiones escritas en OCL2.0.

Se espera que la construcción de esta herramienta sobre una de las técnicas de modelado informal más popular, contribuya hacia la especificación formal en un proceso natural de refinamientos de modelos. El pasaje entre las metodologías de modelado informal y formal proporcionan los medios para entender y usar ambas tecnologías. Para los modeladores del UML o del Objeto-Z pero no de ambos, el análisis conceptual descrito en este trabajo y el uso

posible de una herramienta asociada ofrece una manera de familiarizarse con una nueva tecnología orientada al objeto.

TRABAJOS RELACIONADOS.

Slavisa Markovic and Thomas Baar propone, en [MB2006] un acercamiento para especificar la semántica de modelar idiomas de una manera gráfica. Como ejemplo, describimos la semántica de la evaluación del OCL por las reglas de la transformación escritas en el formalismo gráfico QVT.

Chiorean, Dan, Maria Bortes y Dyan Corutiu. Proponen en [CBC2005] un extendido uso de OCL.

Demuth, Birgit and Heinrich Hussmann, and Ansgar Konermann presentan en [DHK2006] un parser para OCL2.0.

Achim D. Brucker y Burkhart Wolff presentan en [BW2003] una semántica formal para embeber restricciones OCL en Isabelle/HOL.

Martin Gogolla y Mark Richters explican en [GR2002] la funcionalidad de USE, una herramienta de especificación UML, la cual permite validar y verificar descripciones UML y OCL

Mark Richters y Martin Gogolla presentan en [RG2001] una sintaxis formal y semántica para OCL basó en teoría de conjuntos incluso las expresiones, invariantes, precondiciones y poscondiciones. Dan un estudio de algunas herramientas de OCL y muestran una de las herramientas en un poco más detalle. El diseño e implementación de la herramienta que soporta la validación de modelos UML y expresiones OCL están basados en los métodos formales presentados en el artículo.

Mark Richters y Martin Gogolla presentan en [RG2000] una herramienta para validad modelos UML y expresiones OCL. La herramienta USE (UML-based Specification Environment) ayuda a los desarrolladores. Tiene un asistente para simular modelos UML y un intérprete para chequear expresiones OCL.

Mark Richters y Martin Gogolla proponen en [RG1999] un meta modelo para OCL (Object Constraint Language). El beneficio de un meta modelo para OCL es que define la sintaxis de todos los conceptos de OCL precisamente como los tipos, expresiones, y valores de una manera abstracta y por medio de los rasgos de UML. Así, todas las expresiones de OCL legales pueden sistemáticamente derivarse e instanciarse del metamodelo. También muestran que ese meta modelo integra fácilmente con el metamodelo de UML. El enfoque de su trabajo queda en la sintaxis de OCL; el meta modelo no incluye una definición de la semántica de las expresiones OCL.

Bernhard Beckert, Uwe Keller, y Peter H. Schmitt definen en [BUP] una traducción de diagramas de clases con expresiones OCL a lógica de predicado de primer orden. El objetivo del trabajo era razonar lógicamente a cerca de modelos UML. La traducción fue implementada como parte del sistema KeY.

REFERENCIAS

[MB2006] Slavisa Markovic and Thomas Baar. An OCL Semantics Specified with QVT. In O. Nierstrasz et al. (Eds.): MoDELS 2006, LNCS 4199, pp. 645-659, 2006. © Springer-Verlag Berlin Heidelberg 2006.

[CBC2005] Chiorean, Dan and Maria Bortes and Dyan Corutiu. Proposals for a Widespread Use of OCL. Workshop co-located with MoDELS'05: ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Jamaica. October 4, 2005.

[DHK2005] Demuth, Birgit and Heinrich Hussmann, and Ansgar Konermann. Generation of an OCL 2.0 Parser. Workshop co-located with MoDELS'05: ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Jamaica. October 4, 2005.

[CBC2005] Chiorean, Dan and Maria Bortes and Dyan Corutiu. Proposals for a Widespread Use of OCL. Workshop co-located with MoDELS'05: ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Jamaica. October 4, 2005.

[DHK2005] Demuth, Birgit and Heinrich Hussmann, and Ansgar Konermann. Generation of an OCL 2.0 Parser. Workshop co-located with MoDELS'05: ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Jamaica. October 4, 2005.

[BW2003] Achim D. Brucker and Burkhart Wolff. A Proposal for a Formal OCL Semantics in Isabelle/HOL. C. Munoz, S. Tahar, V. Carreno (Eds.): TPHOLs 2002, LNCS 2410, pp. 99?114, 2003. Springer-Verlag Berlin Heidelberg 2003

[GR2002] Martin Gogolla and Mark Richters. Development of UML Descriptions with USE. In A Min Tjoa, Hassan Shafazand, and K. Badie, editors, Proc. 1st Eurasian Conf. Information and Communication Technology (EURASIA'2002). Springer, Berlin, LNCS, 2002.

[RG2001] Mark Richters and Martin Gogolla. OCL - Syntax, Semantics and Tools. In Tony Clark and Jos Warmer, editors, Advances in Object Modelling with the OCL, pages 43-69. Springer, Berlin, LNCS 2263, 2001.

[RG2000] Mark Richters and Martin Gogolla. Validating UML Models and OCL Constraints. In Andy Evans and Stuart Kent, editors, Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000), pages 265-277. Springer, Berlin, LNCS 1939, 2000.

[RG1999] Mark Richters and Martin Gogolla. A Metamodel for OCL. In Robert France and Bernhard Rumpe, editors, Proc. 2nd Int. Conf. Unified Modeling Language (UML'99), pages 156-171. Springer, Berlin, LNCS 1723, 1999.

[BUP] Bernhard Beckert, Uwe Keller, y Peter H. Translating the Object Constraint Language into First-order Predicate Logic. Universität Karlsruhe - Institut für Logik, Komplexität und Deduktionssysteme

[OMG] OMG Unified Modeling Language Specification, Version 1.4, September 2001. http://www.omg.org/library/issuerpt.htm.

[OMG-OCL] OMG Unified Modeling Language Specification, Version 1.4, September 2001. http://www.omg.org/library/issuerpt.htm. Capítulo 6: Object Constraint Language Specification

[Spivey] Spivey. J.M (1989 & 1992) The Z Notation: A Reference Manual, Prentice Hall. Disponible on-line en: http:/Spivey.oriel.ox.ac.uk/~mike/zrm/.

[Object-Z Web] Object-Z Web Environment and Projections to UML. Authors: Jing Sun, Jin Song Dong, Jing Liu, Hai Wang, Department of Computer Science, School of Computing, National University of Singapore. http://www10.org/cdrom/papers/182/index.html.

[Object-Z Web II] Jing Sun, Jin Song Dong, Jing Liu and Hai Wang. Object-Z Web Environment and Projections to UML. 10th International World Wide Web Conference

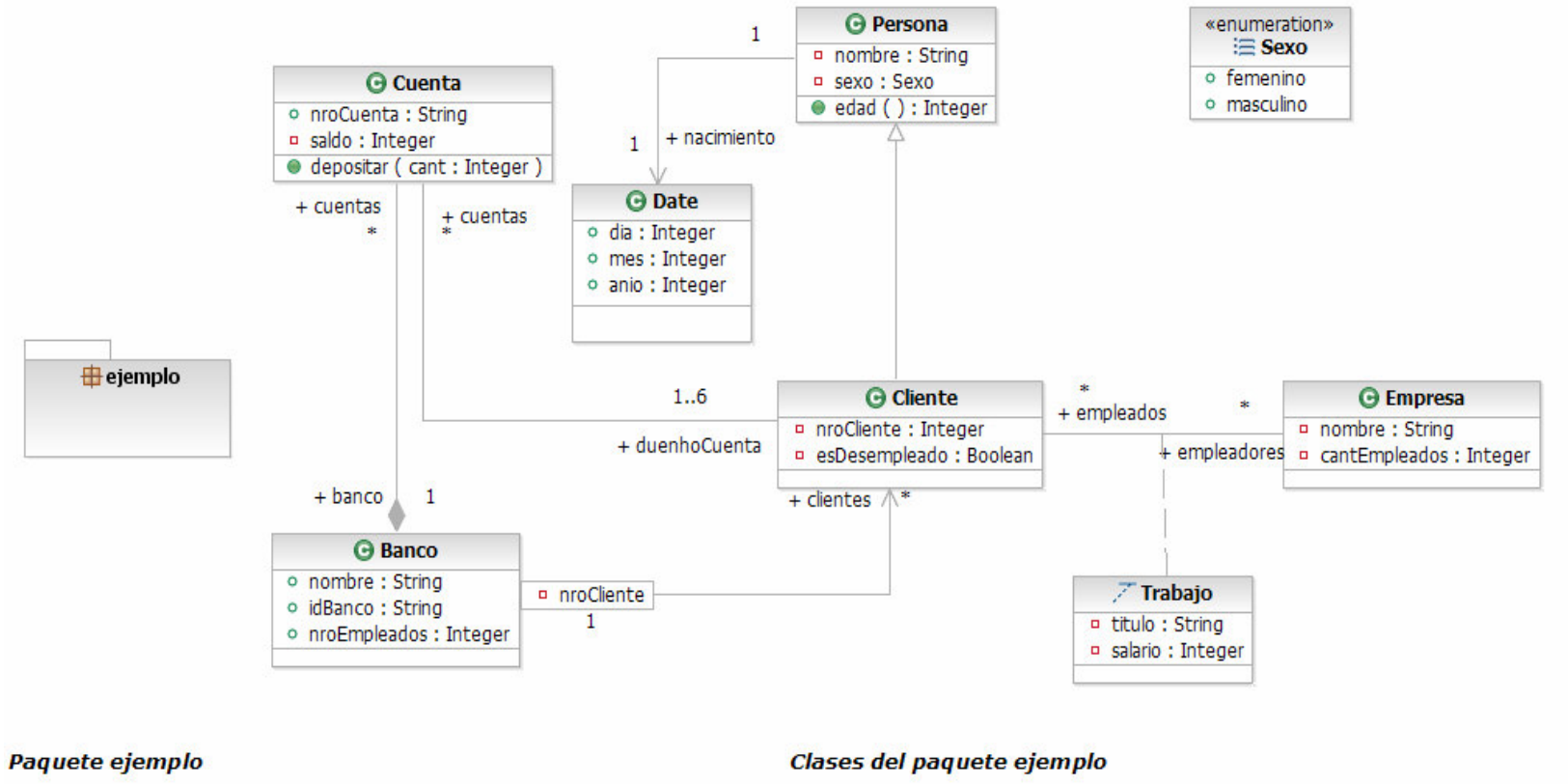
(WWW-10), refereed paper track, ACM Press, pages 725-734, Hong Kong, May 2001. http://nt-appn.comp.nus.edu.sg/fm/zml/

[UML-Object-Z]"The Development of a Tool to Transform from UML to Object-Z" By Bruce Kaines, Bachelor of Information Technology.

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

APÉNDICE A

En este apéndice se puede ver el ejemplo completo, utilizado en el documento. Primero se muestra el diagrama de clases UML, con las restricciones OCL, y luego se puede ver como quedaría la especificación Object-Z, obtenida de aplicar la función de traducción $\mathcal{F}$.



Paquete ejemplo ***Clases del paquete ejemplo***

```
package ejemplo

context Banco

inv :
-- Los números de cuentas son diferentes para cuentas diferentes.
self.cuentas->forAll( c1 |  self.cuentas->forAll( c2 |
  c1 <> c2 implies c1.nroCuenta <> c2.nroCuenta))


context Cliente def:
    let tieneCuenta :Boolean = self.cuentas->isEmpty() <> true

context Cuenta::depositar(cant:Integer)
post: saldo=self.saldo@pre + cant


endpackage
```

A continuación se puede ver la especificación Object-Z que se obtiene de aplicar la función de traducción $\mathcal{F}$ al modelo UML/OCL.

$$Char ::= A \mid B \mid C \mid D \mid E \mid F \mid G \mid H \mid I \mid J \mid K \mid L \mid M \mid N \mid O \mid P \mid Q \mid R \mid S \mid T \mid U \mid V \mid W \mid X \mid Y \mid Z \mid [\mid \backslash \mid] \mid \hat{} \mid \_ \mid ` \mid a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z$$

$$Ejemplo.Sexo ::= masculino \mid femenino$$

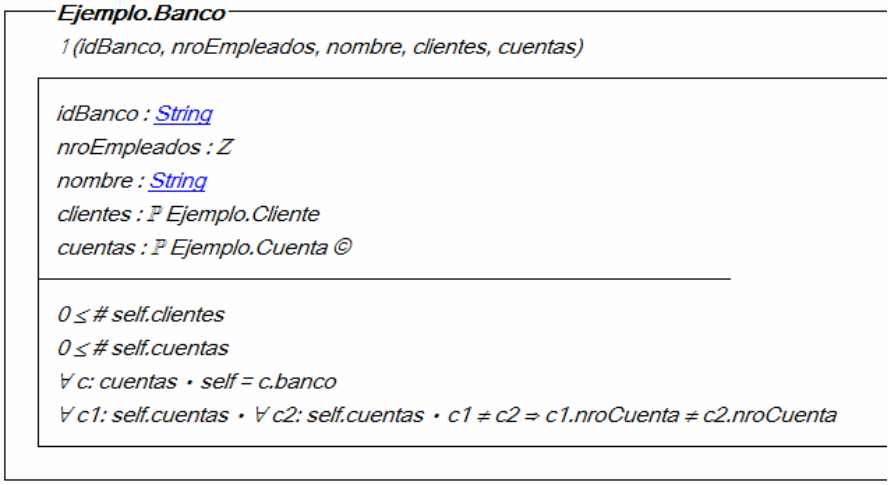
$$String == Seq\ Char$$

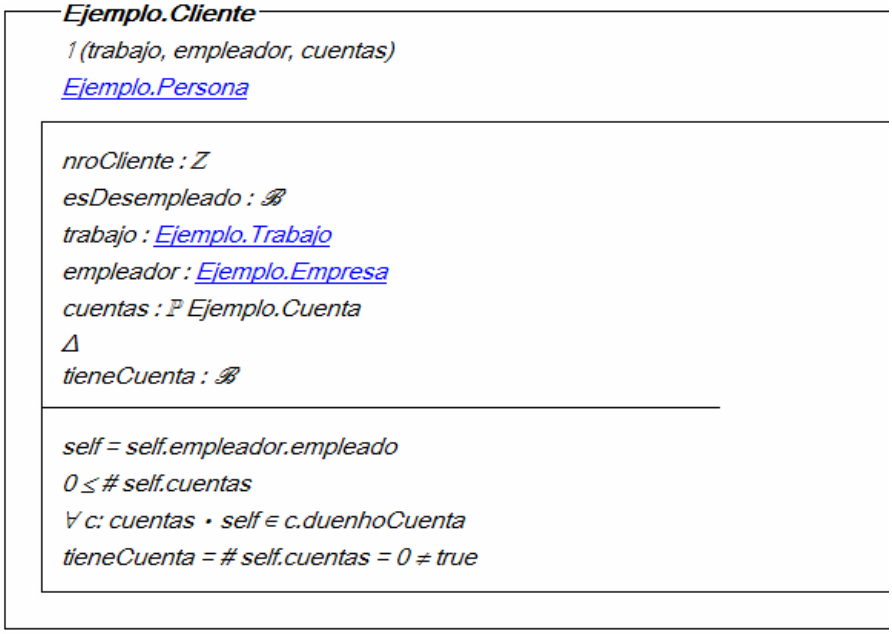
$$dic\ X\ Y == \{ f : X \pfun Y \}$$

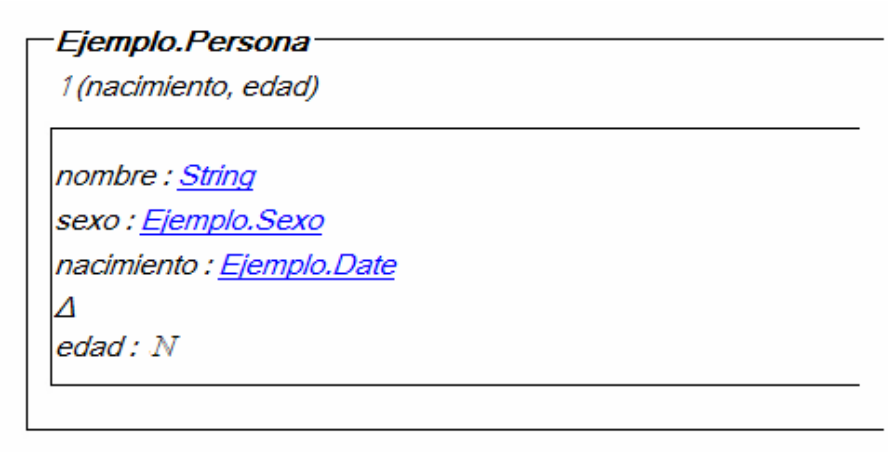
$$[X, Y]$$

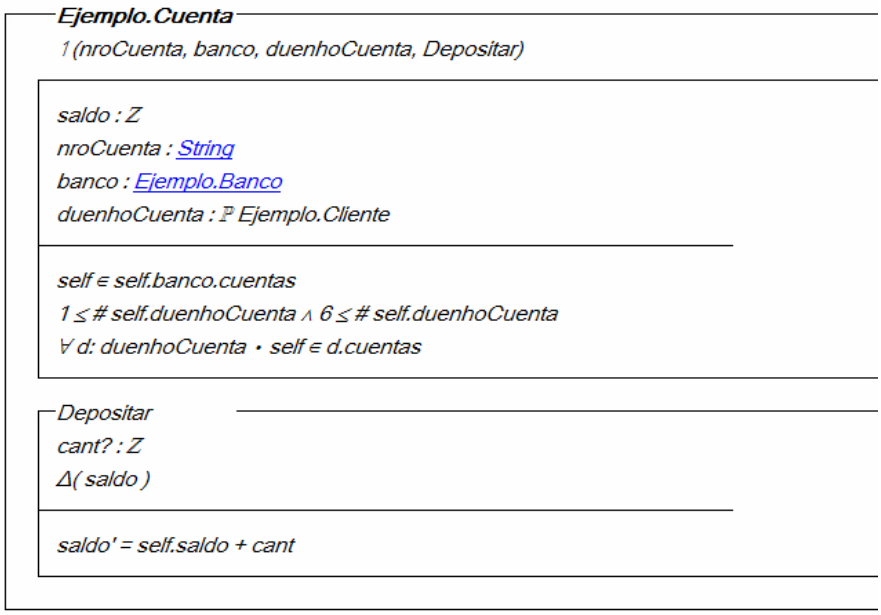
$$add : dic\ X\ Y \times X \times Y \rightarrow dic\ X\ Y$$

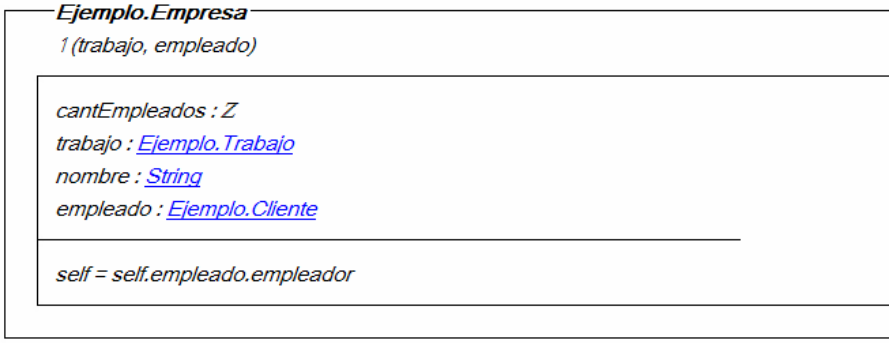
$$\forall d : dic\ X\ Y ; x : X ; y : Y \bullet add\ d\ x\ y = d \cup \{x \mapsto y\}$$

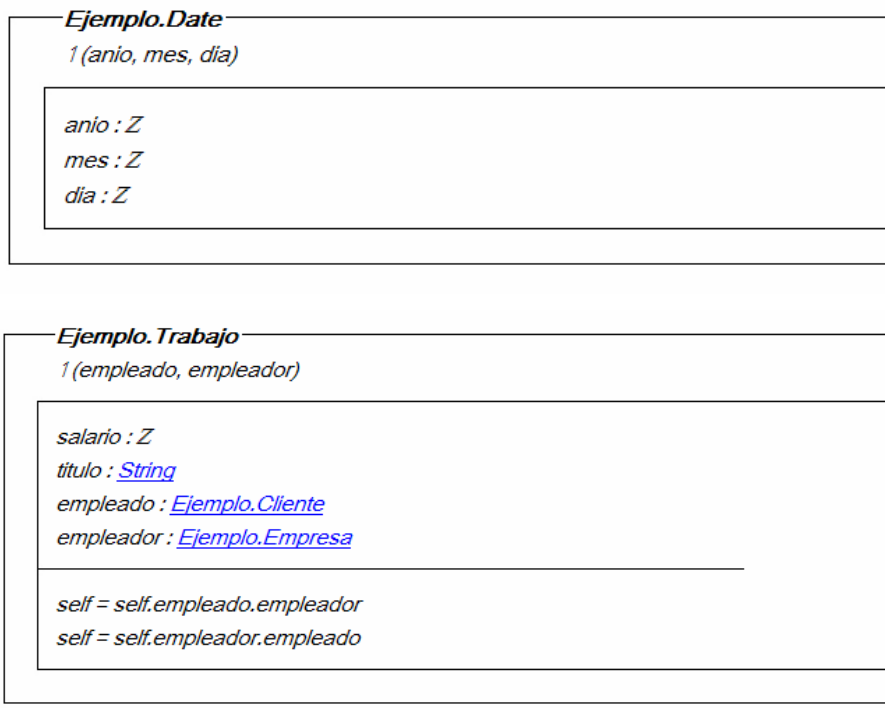

Ejemplo.Banco
1 (idBanco, nroEmpleados, nombre, clientes, cuentas)
idBanco : String
nroEmpleados : Z
nombre : String
clientes : ℙ Ejemplo.Cliente
cuentas : ℙ Ejemplo.Cuenta ©
0 ≤ # self.clientes
0 ≤ # self.cuentas
∀ c: cuentas • self = c.banco
∀ c1: self.cuentas • ∀ c2: self.cuentas • c1 ≠ c2 ⇒ c1.nroCuenta ≠ c2.nroCuenta


Ejemplo.Cliente
1 (trabajo, empleador, cuentas)
Ejemplo.Persona
nroCliente : Z
esDesempleado : 𝔹
trabajo : Ejemplo.Trabajo
empleador : Ejemplo.Empresa
cuentas : ℙ Ejemplo.Cuenta
Δ
tieneCuenta : 𝔹
self = self.empleador.empleado
0 ≤ # self.cuentas
∀ c: cuentas • self ∈ c.duenhoCuenta
tieneCuenta = # self.cuentas = 0 ≠ true


Ejemplo.Persona
1 (nacimiento, edad)
nombre : String
sexo : Ejemplo.Sexo
nacimiento : Ejemplo.Date
Δ
edad : ℕ


Ejemplo.Cuenta
1 (nroCuenta, banco, duenhoCuenta, Depositar)
saldo : ℤ
nroCuenta : String
banco : Ejemplo.Banco
duenhoCuenta : ℙ Ejemplo.Cliente
self ∈ self.banco.cuentas
1 ≤ # self.duenhoCuenta ∧ 6 ≤ # self.duenhoCuenta
∀ d: duenhoCuenta • self ∈ d.cuentas
Depositar
cant? : ℤ
Δ(saldo)
saldo' = self.saldo + cant


Ejemplo.Empresa
1 (trabajo, empleado)
cantEmpleados : ℤ
trabajo : Ejemplo.Trabajo
nombre : String
empleado : Ejemplo.Cliente
self = self.empleado.empleador


Ejemplo.Date
1 (anio, mes, dia)
anio : Z
mes : Z
dia : Z
Ejemplo.Trabajo
1 (empleado, empleador)
salario : Z
titulo : String
empleado : Ejemplo.Cliente
empleador : Ejemplo.Empresa
self = self.empleado.empleador
self = self.empleador.empleado