DEFINICION FORMAL DE LA SEMANTICA DE UML-OCL A TRAVES DE SU TRADUCCION A OBJECT-Z

Valeria Becker y Claudia Pons

LIFIA, Facultad de Informática, Universidad Nacional de La Plata

[valeria, cpons]@sol.info.unlp.edu.ar

Resumen. En este documento presentamos una traducción de diagramas de clases UML complementados con expresiones OCL a expresiones Object-Z.

Nuestro fin es proveer una formalización de los modelos gráfico-textuales expresados mediante UML/OCL que permita aplicar técnicas clásicas de verificación y prueba de teoremas sobre los modelos.

Esta traducción está siendo implementada como parte de una herramienta CASE que permite editar y gestionar modelos. Esperamos que pueda servir como un medio que ayude promover el uso industrial de UML y OCL.

Palabras clave. OCL, UML, Object-Z, lenguajes formales, semántica.

1 INTRODUCCIÓN

La construcción de un sistema de software debe ser precedida por la construcción de un modelo, tal como ocurre en otros sistemas ingenieriles. El modelo de un sistema es una representación conceptual obtenida a partir de la identificación, clasificación y abstracción de los elementos que constituyen el problema y su posterior organización en una estructura formal. De esta forma, el modelo de un sistema actúa como una especificación de los requerimientos que el sistema debe satisfacer, proveyendo un medio de comunicación y negociación entre usuarios, analistas y desarrolladores, así como también un documento de referencia durante la corrección de errores y durante la evolución del producto.

El modelo del sistema se expresa utilizando un lenguaje de modelado (que puede variar desde lenguaje natural o diagramas hasta formulas matemáticas.

El éxito de los lenguajes gráficos de modelado, tales como el Unified Modeling Language UML [OMG2001], se basa principalmente en el uso de diagramas que transmiten un significado intuitivo. Estos lenguajes resultan atractivos para los usuarios ya que aparentemente son fáciles de entender y aplicar. Sin embargo, la falta de precisión en la definición de su semántica puede originar diversos problemas

A partir de 1997 el UML fue aceptado por el por Object Management Group (OMG) como el lenguaje estándar para modelar sistemas orientados a objetos. Como consecuencia de su estandarización surgieron activas discusiones acerca de la precisión sintáctica y semántica de sus construcciones, surgiendo varias propuestas de formalización, por ejemplo [BHH+1997], [EFLR1998], [KC1999], [PB1999], [PB2000].

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 solamente retorna un valor, sin alterar el estado del sistema correspondiente. En este trabajo

presentamos una traducción de UML/OCL en el lenguaje formal Object-Z [Smith2000]. El objetivo es proveer una formalización de los modelos gráfico-textuales expresados mediante UML/OCL que permita aplicar técnicas clásicas de verificación y prueba de teoremas sobre los modelos. Para ello se define una sintaxis y semántica para OCL. Esta formalización 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.

Este documento está dividido en las siguientes secciones: en la sección 2 damos la Sintaxis de un subconjunto de expresiones OCL. En la sección 3 exponemos la traducción de diagramas UML y fórmulas OCL en un lenguaje formal basado en lógica de primer orden. Finalmente en la sección 4 presentamos las conclusiones y describimos los principales trabajos relacionados.

Todos los ejemplos que veremos se referirán al diagrama de Clases dentro del paquete Banco ilustrado en la Figura A.

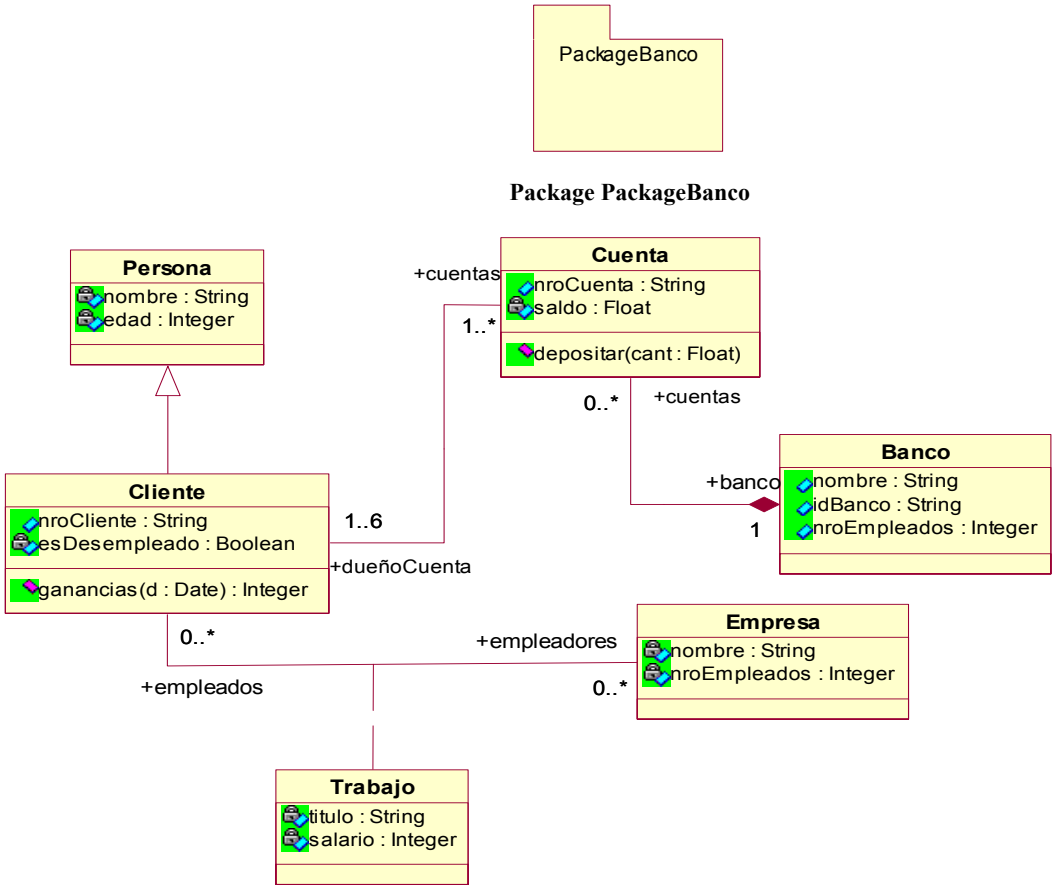


Figura A.

2 SINTAXIS DE UN SUBCONJUNTO DE EXPRESIONES OCL

Antes de dar la gramática veremos 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].

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 Banco. 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.

Por ejemplo:

El siguiente invariante especifica que el número de empleados deber ser siempre mayor que 72, en el contexto de tipo 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 Potscondiciones

Una expresión OCL puede se parte de una Precondición o Poscondición, asociada a un Método u Operación. 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 ($param_i$) pueden ser usadas en la expresión OCL.

context nombreDeTipo::NombreDeOperacion($param_1$: $Tipo_1$, ...): TipoDeRetorno

pre : alguna-expresion-ocl

post: result = alguna-expresion-ocl

Por ejemplo:

context Persona::edad(fechaActual): Integer

post: result = fechaActual – self.fechaDeNac

Valores previos en Postcondiciones

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(n)

post: saldo =saldo@**pre**+n

Paquete

Las fórmulas OCL pueden agruparse dentro de paquetes para facilitar su comprensión y manejo. Las declaraciones de paquetes tienen la siguiente sintaxis:

package Package::SubPackage

…una lista de formulas ocl…

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

- **Boolean**: true, false
- **Integer**: 1, -5, 2, 34, 26524, ...
- **Real**: 1.5, 3.14, ...
- **String**: 'esto es un String...'
- **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 derivado o una operación para ser usada luego en otras expresiones OCL.

Por ejemplo: Se define un atributo indicando si el cliente tiene al menos una cuenta, para luego ser usado.

context Cliente **inv**:

let tieneCuenta : Boolean = self.cuentas-> notEmpty() **in** self.tieneCuenta implies self.edad()<21

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 del Classifier puede ser usado. Veamos un ejemplo:

context Cliente **def**:

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

Colecciones

El tipo Collection está predefinido en OCL. Es un tipo abstracto provisto de un conjunto de operaciones. Sus subtipos son colecciones concretas: Set, Sequence, y Bag. Un tipo Set es el conjunto matemático, no contienen elementos repetidos. Un tipo Bag es como un conjunto, 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 antes de las llaves:

Set {1, 2, 5, 88}

Sequence {1, 3, 45, 2, 3}

Bag {1, 3, 4, 3, 5}

Para definir una secuencia de valores enteros consecutivos, se utiliza dos puntos seguidos. Es decir, *exp_ entera1 y exp_ entera2*, separado por '..' denota todos los enteros entre los valores *exp_ entera1 y exp_ entera2* incluyendo los extremos también:

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

1 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 (el invariante especifica que en el banco debe existir al menos una cuenta cuyo saldo sea mayor que cero):

context Banco **inv**:

self.cuentas->select(c: Cuenta| c.saldo > 0) ->notEmpty()

context Banco **inv**:

self.cuentas->select(c| c.saldo > 0) ->notEmpty()

context Banco **inv**:

self.cuentas->select(saldo > 0) ->notEmpty()

El atributo *self.cuentas es de tipo* Set(Cuenta). El *select* toma cada cuenta cuyo salde sea mayor a 0.

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)

2 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. Por ejemplo: la siguiente expresión especifica la colección con los números de cuentas del banco.

self.cuentas ->collect(c| c.nroCuenta), o simplemente:

self.cuentas ->collect(nroCuenta)

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

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

colección.nombrePropiedad

colección ->collect(nombrePropiedad)

3 *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)

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(c:Cuenta | c.nroCuenta >2000)

Este invariante se satisface 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)

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)

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 (c:Cuenta | c.nroCuenta = 2000)

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

4 Otras operaciones que provee OCL para las colecciones son: *Size, Includes, Excludes, IncludesAll, ExcludesAll, IsEmpty, NotEmpty y Any*. Para mayor información y detalle, ver [OMG-OCL].

3 TRADUCCIÓN

Dado un diagrama de clases UML, incluyendo expresiones OCL, usamos 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. Utilizaremos para la representación del sistema el Lenguaje de Especificación de Sistemas Object-Z [Smith2000] que es una extensión del lenguaje Z [Spivey1989].

3.1 PROPIEDADES DE LA TRADUCCIÓN

Presentamos una función de traducción que toma un diagrama de clases UML, incluyendo expresiones OCL, y retorna una especificación en Object-Z como muestra la Figura B.

La función de traducción F utilizará varias funciones auxiliares para lograr la traducción de una expresión OCL a una expresión Object-Z. Estas funciones auxiliares no serán presentadas en este documento debido a las limitaciones de espacio, pero pueden leerse en [BP2003].

Primero describiremos la traducción de un diagrama de clase y luego se especificará como se agregan las restricciones OCL a la especificación Object-Z obtenida del diagrama de clases.

3.2 TRADUCCIÓN DE 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	$\mathbb{R}$

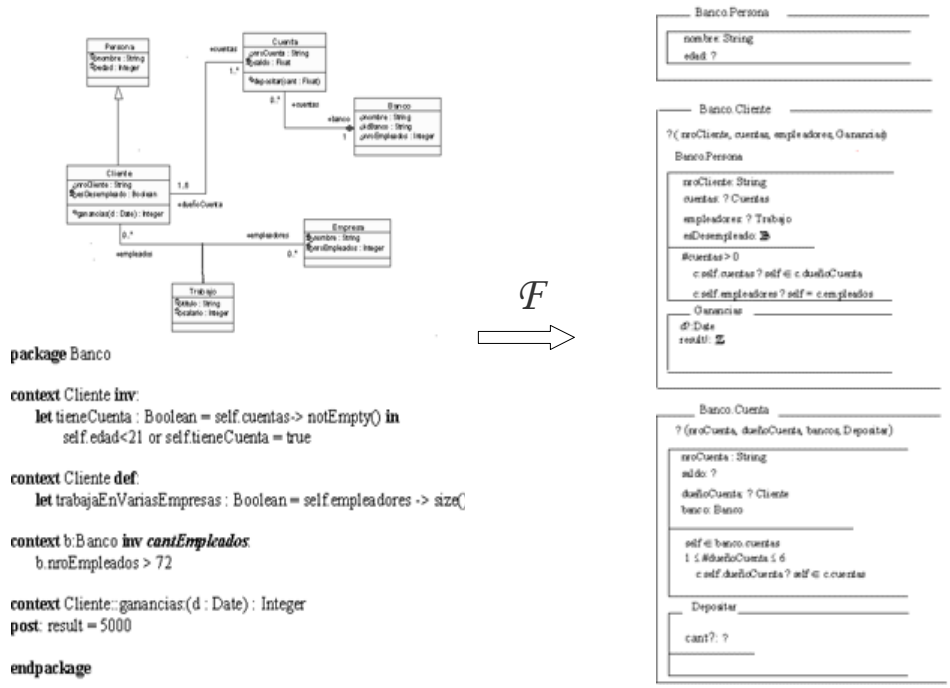


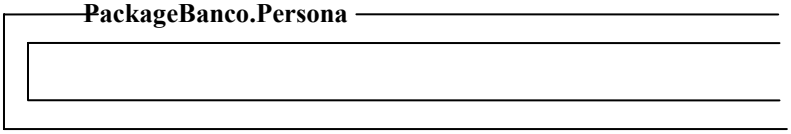
Figura B.

3.3 TRADUCCIÓN DE UN DIAGRAMA DE CLASES

A continuación detallaremos como se traduce cada elemento de un diagrama de clases UML en una especificación Object-Z. Comenzamos la traducción de las clases, atributos, operaciones, asociaciones, clases de asociaciones y por último damos la traducción de las generalizaciones que aparecen en un diagrama de clases. El diagrama UML de la figura A se usará para ejemplificar el proceso de traducción.

- **Clases**

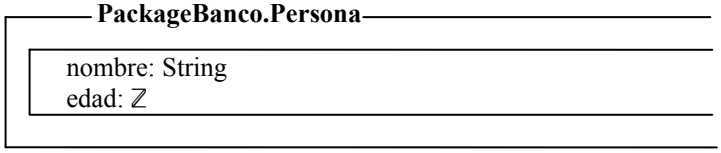
Se crea una clase en Object-Z por cada clase en el diagrama de clases. Cada clase tiene el mismo nombre que en el diagrama UML y se le antepone el nombre del paquete en dónde se encuentra el diagrama y un punto ".". Un ejemplo de traducción de una clase es la siguiente:



Clase PackageBanco.Persona en Object-Z

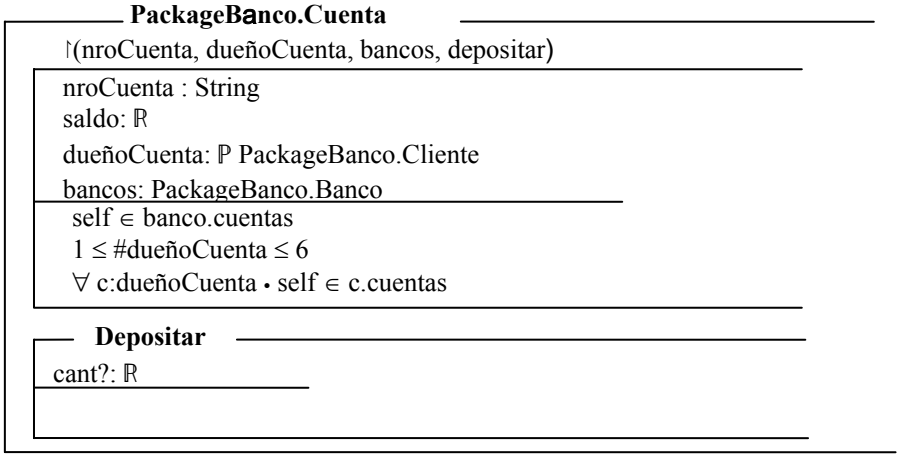
- **Atributos**

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.



- **Operaciones**

Las operaciones de las clases UML son traducidas a esquemas de operaciones en la clase Object-Z correspondiente y con el mismo nombre que en la clase UML. Los parámetros son implementados como entradas con igual nombre. El tipo de retorna es traducido a una salida. Las operaciones públicas, son agregadas a la lista de visibilidad, al igual que los atributos públicos. Por ejemplo:



- **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 en el esquema de estado de la clase correspondiente. Si los extremos de la asociación son navegables, entonces debe agregarse una restricción para especificar la propiedad de simetría de la asociación, por ejemplo: ∀ c:dueñoCuenta • self ∈ c.cuentas.

Si una clase tiene composición con otra clase, entonces se agrega el símbolo © al lado del atributo que representa la otra clase para mostrar la contención del objeto no compartido en Object-Z.

PackageBanco.Cuenta

↾(nroCuenta, dueñoCuenta, bancos, depositar)

nroCuenta : String
saldo: ℝ
dueñoCuenta: ℙ PackageBanco.Cliente
bancos: PackageBanco.Banco

self ∈ banco.cuentas
1 ≤ #dueñoCuenta ≤ 6
∀ c:dueñoCuenta • self ∈ c.cuentas

Depositar

cant?: ℝ

PackageBanco.Banco

↾(nombre, idBanco, nroEmpleados, cuentas)

nombre: String
idBanco: String
nroEmpleados: ℤ
cuentas : ℙ PackageBanco.Cuenta©

∀ c:cuentas • self = c.banco

- **Clase de Asociación**

Una clase de asociación es una asociación que tiene propiedades de clase como así también 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 la simetría de la asociación. Por ejemplo:

PackageBanco.Trabajo

↾(empleados, empleadores)

titulo: String
salario: ℤ
empleados : PackageBanco.Cliente
empleadores: PackageBanco.Empresa

self ∈ empleados.empleadores
self ∈ empleadores.empleados

- **Generalización**

La relación de generalización se representa en Object-Z especificando el nombre de la clase padre en el esquema de clase, debajo de la lista de declaración. Por ejemplo:

PackageBanco.Cliente

↾(nroCliente, cuentas, empleadores, Ganancias)
PackageBanco.Persona

nroCliente: String
cuentas: ℙ PackageBanco.Cuentas
empleadores: ℙ PackegeBanco.Trabajo
esDesempleado: 𝔹

#cuentas > 0
∀ c:cuentas • self ∈ c.dueñoCuenta
∀ c:empleadores • self ∈ c.empleados

Ganancias

d?:Date
result!: ℤ

3.4 TRADUCCIÓN DE UNA EXPRESIÓN OCL.

Para la traducción de una expresión OCL se tiene en cuenta que las clases a las que se hace referencia están especificadas en un modelo UML y que fueron traducidas a una especificación Object-Z, que la llamamos *S*. La función de traducción toma la especificación *S* y la enriquece con nuevas propiedades obtenidas de un oclFile y retorna un zFile, que contiene expresiones Object-Z. En este documento no vamos a presentar cómo está definida la traducción de una expresión OCL pero si mostraremos un ejemplo.

En la especificación Object-Z al nombre de las clases de un paquete le precede el nombre del paquete seguido de un punto. Por ejemplo: **PackageBanco.Persona**

La función de traducción enriquece el esquema de estado de una clase por cada "**inv"** definido en una expresión OCL. El ejemplo de abajo muestra un invariante en la clase Cliente del paquete PackageBanco que será traducido a un predicado Object-Z y agregado al esquema de estado de la clase Cliente.

Por cada "**def"** que aparezca en una expresión OCL, se agrega una operación con el mismo nombre, los parámetros son variables de entrada en la clase Object-Z correspondiente. Además, si la operación retorna un valor se traduce como una variable de salida llamada resul! que contendrá el valor de retorno de la operación. La operación es agregada a la lista de visibilidad.

En ejemplo que se muestra a continuación, se especifica un "**def"** en el contexto Banco, que agrega una nueva operación a la clase Object-Z Banco. En este caso, también se puede ver la traducción de una expresión OCL donde interviene la operación *select* sobre una variable de tipo Set (es el caso de: "self.cuentas→select(c | c.saldo >1000)"). La función de traducción transcribe la expresión OCL a una conjunto en Object-Z expresado por comprensión, es decir, que seleccionamos del conjunto (self.cuentas) aquellos elementos que satisfacen la propiedad descripta dentro del cuerpo del select y lo escribimos de la siguiente forma: {c:cuentas | c.saldo >1000}

Para el caso de las expresiones que contienen una subexpresión "**let nomb=… in…**" la función de traducción agrega a la clase en dónde esta definida esta expresión una operación, en este caso, con el nombre "Nomb", pero no es agregada a la lista de visibilidad. El ejemplo de abajo muestra este caso en la definición del invariante de la clase Cliente, es decir que la función de traducción agrega un esquema de operación llamado "TienenCuenta" en la clase PackageBanco.Cliente de Object-Z, pero no la agrega a la lista de visibilidad.

La última restricción en el ejemplo, muestra una poscondición en la operación "ganancias:(d: Date)" de la clase Cliente. La función de traducción enriquece el esquema de operación de la clase Object-Z correspondiente agregando una línea más al predicado de este esquema.

Veamos un ejemplo de cómo la función de traducción agrega las restricciones OCL a una especificación Object-Z obtenida del diagrama de Clases UML. Sean las siguientes restricciones escritas en un archivo OCL:

package PackageBanco

context Cliente **inv**:

let tieneCuenta : Boolean = self.cuentas-> notEmpty() **in** self.tieneCuenta implies self.edad>21

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

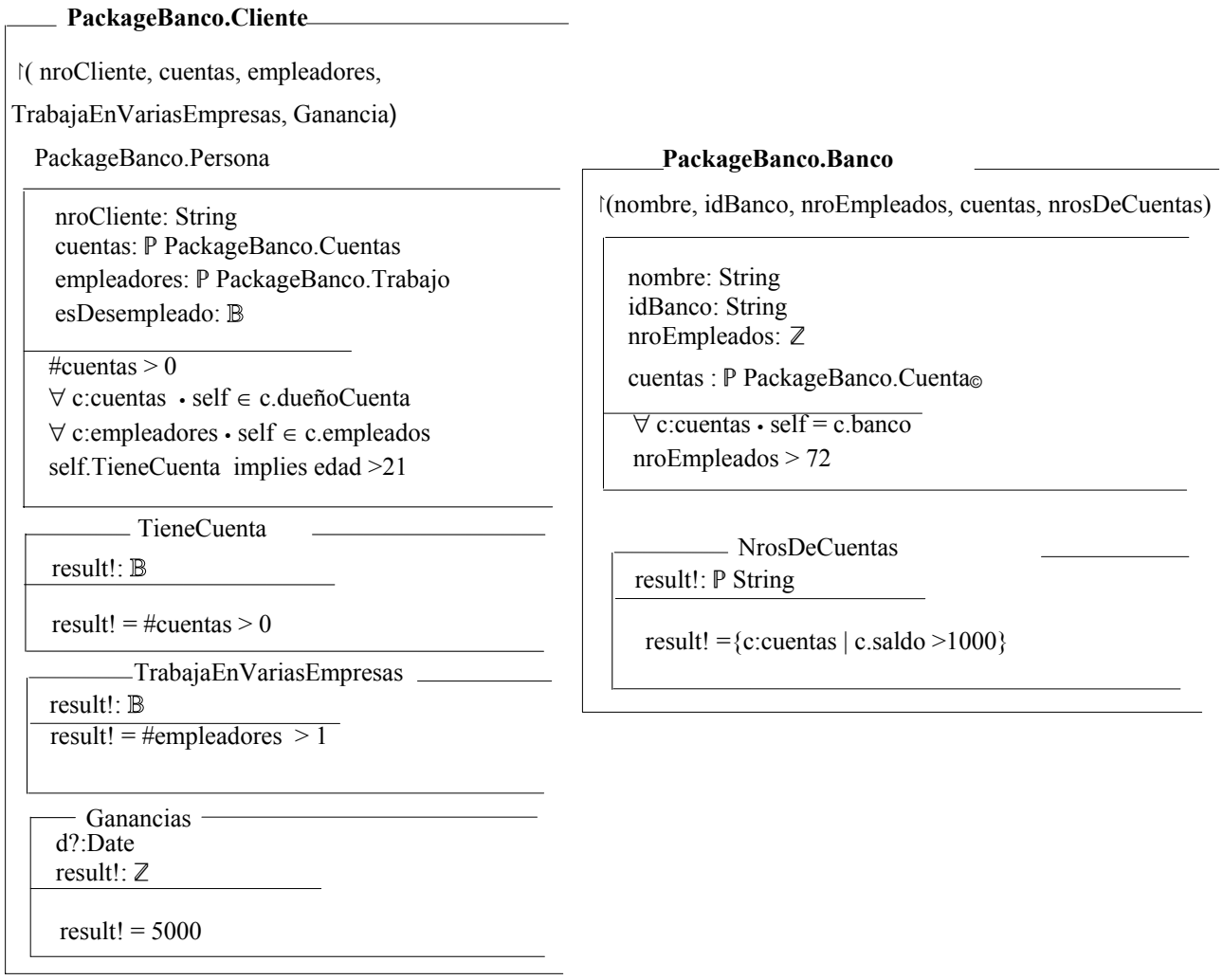
context Banco **def**: **let** nrosDeCuentas : Set(String) = self.cuentas→ select(c | c.saldo >1000)

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

context Cliente::ganancias:(d:Date) : Integer **post**: result = 5000

endpackage

Las siguientes son las clases de la especificación Object-Z obtenida del diagrama de clases de la Figura A que fueron modificadas en el proceso de traducción de las restricciones OCL:



4 CONCLUSIONES Y TRABAJOS RELACIONADOS

En este artículo hemos reportado una formalización de modelos gráfico-textuales expresados mediante UML y OCL. Esta formalización se logró mediante un proceso de traducción hacia el lenguaje formal Object-Z, permitiendo de esta forma la aplicación de técnicas clásicas de verificación y prueba de teoremas sobre los modelos.

Esta traducción está siendo implementada como parte de una herramienta CASE, desarrollada en el marco del proyecto Lifia-Eclipse [LIFIA2003] como un plugin a la plataforma universal Eclipse [Eclipse 2003].

Respecto a trabajos similares, el grupo de investigación liderado por Martin Gogolla y Mark Richters ha conducido en los últimos años las líneas de investigación tendientes a la formalización de OCL, sus trabajos mas relevantes son los siguientes:

- en [RG1999] proponen un meta modelo para OCL. El beneficio de un meta modelo para OCL es que define la sintaxis de todos los conceptos de OCL (como los tipos, expresiones, y valores) precisamente y de una manera abstracta y por medio del mismo UML. Así, todas las expresiones de OCL legales pueden sistemáticamente derivarse e instanciarse del metamodelo. También muestran que ese meta modelo se 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.

- en [RG2001] presentan una sintaxis formal y semántica para OCL basado en teoría de conjuntos. Luego, basándose sobre dicha formalización en [GR2002] y en [RG2000] explican la funcionalidad de USE (UML-based Specification Environment), una herramienta de especificación UML, la cual permite validar y verificar descripciones UML y OCL. USE tiene un asistente para simular modelos UML y un intérprete para chequear expresiones OCL.

El trabajo reportado en este articulo persigue el mismo objetivo que este grupo de investigación, aunque difiere en el dominio semántica elegido y la forma de definir la sintaxis y semántica de OCL. Nuestra hipótesis es que el uso del lenguaje Object-Z para dar semántica a OCL aporta beneficios adicionales tales como:

- existen herramientas maduras de verificación ya conocidas por la comunidad (en particular herramientas para Z como [Z-EVES] que pueden adaptarse para Object-Z).
- un dominio semántico orientado a objetos, tal como el provisto por Object-Z, permite esclarecer aspectos ambiguos de OCL (presentes por ejemplo en el manejo de colecciones y en el uso del polimorfismo) de una manera simple y directa.

AGRADECIMIENTOS

Este trabajo ha sido parcialmente financiado mediante el subsidio "IBM Eclipse Innovation Grant 2003" asignado al proyecto Lifia-Eclipse.

REFERENCIAS

[BHH+1997] Breu,R., Hinkel,U., Hofmann,C., Klein,C., Paech,B., Rumpe,B. y Thurner,V., Towards a formalization of the unified modeling language. ECOOP'97 procs., Lecture Notes in Computer Science vol.1241, Springer (1997).

[BUP] Bernhard Beckert, Uwe Keller, Peter H. Translating the Object Constraint Language into First-order Predicate Logic. Universität Karlsruhe - Institut für Logik, Komplexität und Deduktionssysteme. 2002.

[BP2003] Valeria Becker y Claudia Pons. Semántica de OCL. Reporte interno del proyecto Lifia-Eclipse. http://sol.info.unlp.edu.ar/~eclipse

[Eclipse 2003] The Eclipse Project. Home Page. IBM. http://www.eclipse.org

[EFLR1998] Evans,A., France,R., Lano,K. y Rumpe, B., Developing the UML as a formal modeling notation, UML'98 Beyond the notation, Muller and Bezivin editors, Lecture Notes in Computer Science 1618, Springer-Verlag (1998).

[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.

[KC1999] S.Kim and D.Carrington. formalizing the UML class diagrams using Object-Z. Proceedings of UML Conference . Lecture Notes in Computer Sciencie 1723. Nov.1999.

[LIFIA2003] Pagina del Proyecto Lifia-Eclipse. http://sol.info.unlp.edu.ar/~eclipse

[PB1999] Pons,C., Baum,G., Felder,M., Foundations of Object-oriented modeling notations in a dynamic logic framework, Fundamentals of Information Systems, Chapter 1, T.Polle,T.Ripke,K.Schewe Editors, Kluwer Academic Publisher (1999).

[PB2000] Pons C., and Baum G. Formal foundations of object-oriented modeling notations, 3rd International Conference on Formal Engineering Methods, IEEE ICFEM 2000, IEEE Computer Society Press, 4-7 September 2000, York, UK.

[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.

[OMG2001] OMG Object Management Group. Unified Modeling Language Specification, Version 1.4, September 2001. http://www.omg.org .

[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

[Smith2000] Graeme. Smith. The **Object-Z** Specification Language. Advances in Formal Methods. Kluwer Academic Publishers, 2000. ISBN 0-7923-8684-1.

[Spivey1989] 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/.