View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Servicio de Difusión de la Creación Intelectual

Una integración de modelos estáticos UML y Eiffel

Resumen

Las técnicas formales y semiformales de especificación de software pueden jugar roles complementarios en el desarrollo de software orientado a objetos. Se describe en este trabajo un método de ingeniería “forward” de modelos estáticos UML basado en la integración de notaciones semiformales UML, especificaciones algebraicas y código. Eiffel fue seleccionado como lenguaje orientado a objetos. El énfasis está puesto en la descripción de una de las etapas del método: la transformación de especificaciones algebraicas a Eiffel. En particular, se analizan las transformaciones para los diferentes tipos de relaciones UML.

1. Introducción

En los últimos años UML ("Unified Modeling Language") ha emergido como un estándar de–facto para expresar modelos orientados a objetos (OMG, 1999). Es un lenguaje diseñado para especificar, visualizar, construir y documentar artefactos de sistemas de software (Booch et al., 1999).

Si bien UML no es un lenguaje de programación, sus modelos pueden conectarse a variedad de lenguajes a través de procesos de ingeniería “forward” y reversa. Existen en el mercado herramientas CASE que asisten en la generación de código a partir de modelos orientados a objetos. Pueden mencionarse entre otras, Rational Rose, Together, Argo/UML, Stp/UML y MagicDraw UML. La falta de una semántica formal para UML limita la potencia de las herramientas CASE que deben operar sobre los modelos expresados en dicho lenguaje. Una fuente de problemas en los procesos de generación de código es que los modelos UML son semánticamente más ricos que los lenguajes de programación orientados a objetos. Por ejemplo, estos últimos no poseen una sintaxis explícita para expresar diferentes tipos de asociaciones. Si bien éstas pueden ser simuladas por referencias y punteros, la estructura del sistema no es transparente y esto conduce a problemas durante los procesos de ingeniería "forward" y reversa. Asimismo toda la información registrada en los modelos UML, como por ejemplo, especificaciones en OCL, no tiene implicancias en la implementación.

Estos problemas han sido ampliamente reconocidos y han motivado el análisis de distintos enfoques para dar semántica a las notaciones UML. Una alternativa es darles precisión a partir de especificaciones formales y permitir a los diseñadores que manipulen los modelos UML que ellos han creado sin forzarlos a un cambio de estilo de especificación. Esto puede lograrse si es posible definir correspondencias entre las construcciones de los modelos UML, las especificaciones formales y el código, que permitan una generación asistida de los dos últimos. Asimismo, sólo deberían formalizarse las etapas del proceso que puedan beneficiarse de una especificación formal y un análisis riguroso.

En Favre y Clérici (2001) se presentó un método riguroso para ingeniería "forward" a partir de modelos estáticos UML. El mismo se basa en tender un puente entre los modelos estáticos UML que intervienen en estos procesos y código orientado a objetos basado en la formalización algebraica de los primeros, la especificación formal de bibliotecas de componentes reusables y la definición de un proceso transformacional guiado por reglas para traducir paso a paso construcciones UML permitiendo “traceability”. Los diseñadores pueden manipular los modelos UML que han creado y no especificaciones formales, es decir, podrían ignorar el formalismo algebraico.

Para proveer una base formal a las estructuras sintácticas y semánticas de los modelos UML y mecanismos para manipularlos e integrarlos a procesos de generación de código se definió el

lenguaje algebraico GSBL^{oo}. Para facilitar el reuso se definió el modelo *SpReIm*. Este describe jerarquías de clases orientadas a objetos en diferentes niveles de abstracción que integran especificaciones y código orientado a objetos. El proceso propuesto para la generación de código se basa en establecer correspondencias formales entre UML, GSBL^{oo} y lenguajes orientados a objetos. Las transiciones entre los diagramas UML y todas las especificaciones intermedias se realizan aplicando operadores de transformación que preservan la integridad entre especificaciones y código.

Se presenta en este trabajo un análisis de una de las etapas de este proceso, la integración de especificaciones algebraicas en GSBL^{oo} con código orientado a objetos. Eiffel (Meyer, 1997) fue seleccionado para probar la factibilidad de este enfoque. En particular se analiza la transformación de los diferentes tipos de asociaciones en código Eiffel.

En la sección 2 se describen trabajos relacionados. La sección 3 incluye las bases de este trabajo: el lenguaje GSBL^{oo}, el modelo de componentes reusables *SpReIm* y el proceso de ingeniería "forward" propuesto. En la sección 4 se describe la integración de especificaciones GSBL^{oo} y el lenguaje Eiffel. Finalmente, en la sección 5 se presentan conclusiones.

2. Trabajos relacionados

Entre las referencias clásicas vinculadas a la integración de técnicas semiformales de análisis y diseño orientado a objetos con técnicas de especificación formal pueden citarse a France et al., (1997), que describen la formalización en Z de modelos FUSION, y Bourdeau y Cheng (1995) que presentan un método para derivar especificaciones algebraicas Larch a partir de diagramas de clases.

Con la aparición de UML han surgido discusiones acerca de la precisión semántica del lenguaje. La formalización de UML es un tema abierto aún y numerosos grupos de investigación han logrado formalizar partes del lenguaje. En 1997 surge el grupo pUML (Precise UML Group) que pretende dar precisión a UML (Evans et al. (1998). Variedad de investigaciones dan semántica a subconjuntos de UML basándose en distintos formalismos: Lano (1995) usando Z++. Breu et al.(1997) hace un trabajo similar usando especificaciones algebraicas "Stream Oriented"; Bruel y France (1998) describen cómo formalizar UML usando Z, Gogolla y Ritchers (1997) analizan la transformación de UML a TROLL y Kim y Carrington (1999) usando OBJECT-Z. Overgaard (1998) presenta una semántica operacional de UML. Hussmann et al.(1999) y Padawitz (2000) especifican modelos UML en CASL y en particular presentan diferentes enfoques para especificar asociaciones. Varizi y Jackson (1999) proponen una herramienta, Alcoa, para analizar modelos orientados a objetos que usa su propio lenguaje, Alloy, basado en Z. Firesmith y Henderson-Sellers (1998) y Barbier et al (2001) describen extensiones a UML tanto en la notación como en el metamodelo.

Actualmente hay pocos métodos que incluyen OCL, la referencia más importante es Catalysis (D'Souza y Wills, 1999). Mandel y Cengarle (1999) han examinado la potencia expresiva de OCL en términos de "navegabilidad" y computabilidad. Ritchers y Gogolla (2000) proponen validar modelos UML y constraints OCL a partir de animaciones.

3. Las bases de un proceso de ingeniería "forward" de modelos UML

3.1. El lenguaje GSBL^{oo}

Los modelos conceptuales orientados a objetos están estructurados a partir de un repertorio de relaciones semánticamente más ricas que las que proveen los lenguajes de especificación. Por ejemplo, en los modelos orientados a objetos las asociaciones son relaciones de igual peso que

las de generalización. Las asociaciones permiten abstraer la interacción entre clases en el diseño de grandes sistemas y afectan la partición de un sistema en módulos. Sin embargo, han sido subordinadas a otras relaciones tanto en los lenguajes orientados a objetos como en los de especificación formal. Booch et al.(1999) distinguen relaciones de dependencia, generalización, asociaciones y realizaciones. Los diagramas de clase UML expresan, con una sintaxis diferente distintos tipos de asociaciones: asociación ordinaria, asociación calificada, clase asociación, agregación y composición. Un análisis detallado puede consultarse en (OMG, 1999), (Firesmith y Henderson, 1998) y (Barbier et al., 2001).

Las especificaciones formales deberían preservar los mecanismos de estructuración de los modelos orientados a objetos, sus construcciones y los conceptos subyacentes reforzando las interpretaciones informales.

Teniendo en cuenta lo anterior se diseñó el lenguaje GSBL^{oo}. Su diseño está ampliamente influenciado por UML. GSBL^{oo} extiende a GSBL (Clérici y Orejas, 1988) con mecanismos para especificar en forma directa modelos estáticos UML. Su diseño está centrado tanto en abstracciones de datos como en relaciones. El mismo incluye mecanismos para definir nuevas relaciones a partir de otras existentes, con sus propias propiedades que pueden utilizarse como si fuesen primitivas. Esta aproximación permite extender la jerarquía de relaciones y definir asociaciones como unidades independientes, liberando al diseñador de definir la semántica genérica para cada aplicación concreta.

GSBL^{oo} distingue dos tipos de clases para especificar clases de objetos y relaciones. La Figura 1 muestra la sintaxis de las mismas.

```
OBJECT CLASS <className>[<parameterList>]
USES <usesList>
REFINES <refinestList>
<< <relationName> >>ASSOCIATES<className>
<< <relationName> >> HAS-A SHARED <className>
<< <relationName> >>
HAS-A NON-SHARED <className>
BASIC CONSTRUCTORS <constructorList>
DEFERRED
SORTS <sortList>
OPS <opsList>
EQS <varList> <equationList>
EFFECTIVE
SORTS <sortList>
OPS <opsList>
EQS <varList> <equationList>
END-CLASS
```

```
ASSOCIATION | WHOLE-PART
 < className>
IS <constructorTypeName>
 [...:Class1;...:Class2;...:Role1;...:Role2;.
..:Mult1;...:Mult2;...:Visibility;...:Visibility2]
CONSTRAINED-BY <constraintList>
END-CLASS
```

Figura 1. Especificación de clases de objetos y relaciones

El encabezamiento de la OBJECT CLASS declara el nombre de la clase, las palabras claves OBJECT CLASS deben ser seguidas por el nombre de la clase.

Las dos primeras cláusulas (USES y REFINES) declaran subespecificaciones importadas o heredadas respectivamente.

Las agregaciones se representan en una OBJECT CLASS mediante la claúsula HAS-A. El nombre de la relación se encierra entre dobles paréntesis angulares. Las palabras claves SHARED y NON-SHARED distinguen agregaciones de composiciones. Las asociaciones se especifican mediante la cláusula ASSOCIATES. Para especificar diferentes tipos de asociaciones GSBL^{oo} provee un repertorio de tipos constructores para asociaciones. Las relaciones concretas se obtienen instanciando y adaptando esquemas de tipos constructores.

La cláusula DEFERRED declara "sorts", operaciones y ecuaciones que no están completamente definidos, es decir no hay suficientes ecuaciones para definir el comportamiento de las nuevas operaciones o no hay suficientes operaciones para generar todos los valores de un "sort".

La cláusula EFFECTIVE agrega nuevos "sorts", operaciones o ecuaciones completamente definidos o completa la definición de algún "sort" u operación definido en forma incompleta en

alguna superclase.

Las clases asociación se especifican en forma independiente, encabezadas por las palabras claves ASSOCIATION o WHOLE-PART para distinguir asociaciones ordinarias de agregaciones. La cláusula IS vincula la relación a un tipo constructor que será instanciado a partir de las clases intervinientes en la relación, multiplicidades, roles y visibilidad. La cláusula CONSTRAINED-BY permite especificar constraints asociados a la relación.

El mecanismo provisto por $GSBL^{oo}$ para agrupar clases y relaciones y controlar su visibilidad es el PACKAGE. En (Favre y Clérici, 2001) se describe en detalle este lenguaje.

3.2. El modelo *SpReIm*

El modelo *SpReIm* permite describir componentes reusables en diferentes niveles de abstracción que integran especificaciones algebraicas y código orientado a objetos (Favre y Clérici, 2001). El mismo explota la potencia del formalismo algebraico para expresar abstracciones, a la vez que respeta los principios de diseño de las taxonomías orientadas a objetos. La especificación formal de componentes evita ambigüedades e inconsistencias en sus descripciones y permite su adaptación mediante la aplicación de operadores de reuso que preservan relaciones semánticas.

El modelo *SpReIm* permite describir jerarquías de objetos en tres diferentes niveles de abstracción: *Especialización, Realización* e *Implementación.*

El nivel de especialización describe una jerarquía de especificaciones incompletas relacionadas por relaciones de especialización. En este contexto, debe verificarse que si P(x) es una propiedad acerca de objetos x de tipo T, luego P(y) debe ser verificado para cada objeto y de tipo S, donde S es una especialización de T.

El nivel de especialización tiene dos vistas, una especificada en OCL y otra en $GSBL^{oo}$. La primera permite al usuario identificar componentes especificados en un lenguaje conocido por él, OCL. Esta vista contiene clases especificadas a partir de precondiciones, postcondiciones e invariantes OCL. La vista algebraica brinda el soporte para la reusabilidad.

Cada hoja en el nivel de especialización está asociada con un subcomponente en el nivel de realización. Un subcomponente del nivel de realización es un árbol de especificaciones algebraicas: la raíz del árbol es la especificación más abstracta, los nodos internos corresponden a diferentes realizaciones de la raíz y las hojas corresponden a subcomponentes en el nivel de implementación.

Cada especificación en el nivel de realización está asociada a un subcomponente en el nivel de implementación, el cual agrupa un conjunto de esquemas de implementación asociados con una clase en un lenguaje orientado a objetos.

El componente Association

Association es un tipo especial de componente *SpReIm* (ver Figura 2). El nivel de especialización de este componente describe asociaciones a través de especificaciones incompletas clasificadas de acuerdo a su tipo, su grado y su conectividad. El nivel de realización describe una jerarquía de especificaciones completas asociadas a diferentes realizaciones. Por ejemplo, para una asociación binaria, bidireccional y muchos a muchos, pueden asociarse realizaciones diferentes mediante hashing, secuencias o árboles. El nivel de implementación asocia a cada hoja del nivel de realización diferentes implementaciones en un lenguaje orientado a objetos.

Los subcomponentes del nivel de implementación son esquemas reusables de implementaciones de asociaciones.

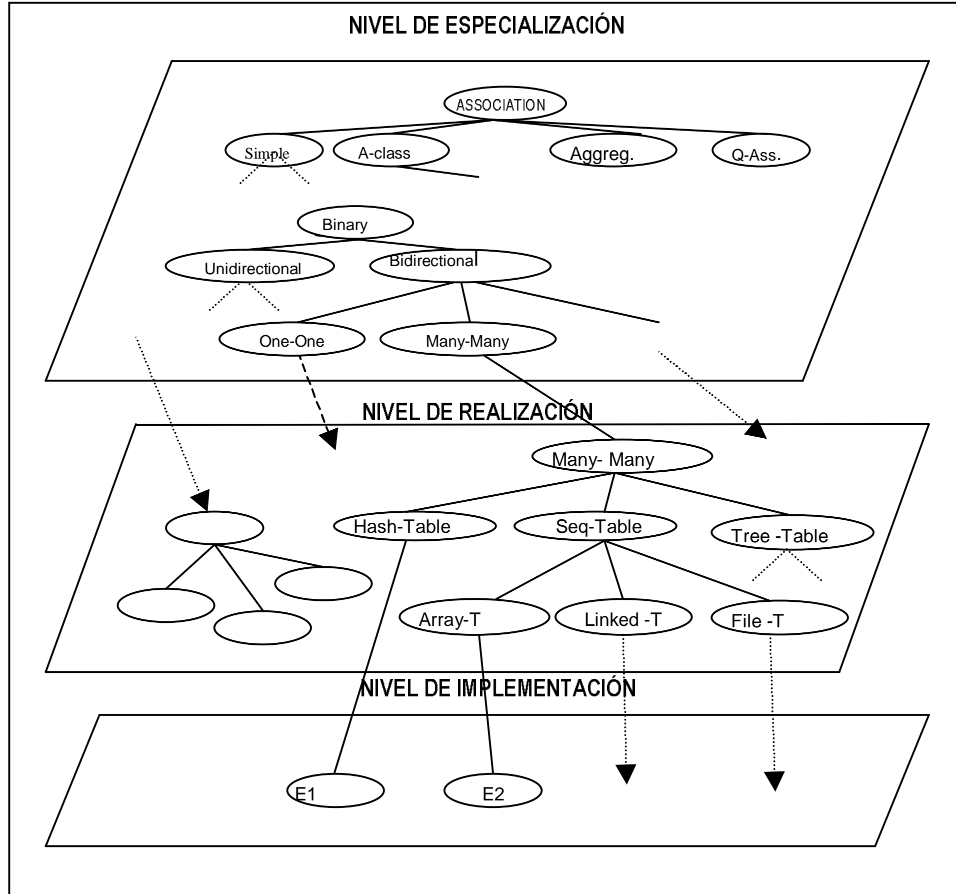


Figura 2. El componente ASSOCIATION

3.3. Un proceso riguroso de ingeniería "forward"

Se describen a continuación las etapas básicas del método de ingeniería "forward" propuesto: *Especificación, Traducción y Reuso.*

En la etapa de Especificación, el diseñador completa el diagrama de clases UML. La construcción de diagramas de clases se realiza en forma iterativa e incremental. El diseñador debe homogeneizar los modelos estáticos que surgieron en etapas previas con otros modelos UML. Esto lo hace valiéndose de las facilidades provistas por la herramienta CASE como así también analizando él mismo la semántica de las clases involucradas a fin de reestructurar clases y sus relaciones manteniendo cohesión y detectando "similaridades" entre clases que pueden ser combinadas, partidas en múltiples clases o eliminadas del modelo.

En esta etapa el diseñador debe interactuar con la biblioteca *SpReIm* a fin de fomentar la construcción reusando componentes existentes. Por ejemplo, puede identificar ciertas clases que pueden obtenerse a partir de la adaptación de otras existentes en componentes *SpReIm*. El diseñador dispone de la vista OCL del nivel de especialización de la biblioteca, a partir de la cual puede identificar componentes cuando construye el diagrama de clases.

Además, el diseñador debe completar la especificación de los diagramas mediante la inclusión de aserciones OCL (precondiciones, postcondiciones e invariantes) que permitan a posteriori una integración más potente de estos modelos con el código generado. Como resultado de esta fase se obtiene entonces un diagrama o "packages" que agrupan diagramas, especificados en OCL y relacionados a componentes.

En la fase de *Traducción*, las especificaciones UML anotadas en OCL son traducidas a una

especificación GSBL^{oo} incompleta. Esta especificación puede ser obtenida automáticamente a partir de la instanciación de esquemas reusables y de la biblioteca de componentes *SpReIm*.

Un aspecto a destacar en este proceso es la transformación de especificaciones OCL a axiomas GSBL^{oo}. Las especificaciones OCL (precondiciones, postcondiciones, invariantes de clases y constraints de asociaciones) se traducen a axiomas en las especificaciones asociadas. Estas transformaciones son soportadas por un sistema de reglas de transformación que traducen especificaciones OCL a GSBL^{oo} (Favre et al., 2000).

La especificación obtenida permite realizar un análisis del comportamiento modelado, así como crear modelos más informativos y precisos que realimentan el proceso.

En la etapa de *Reuso* se construye una especificación algebraica lo "más completa posible" que se integra con implementaciones en Eiffel. El proceso está basado en componentes genéricos *SpReIm* que pueden ser manipulados formalmente a fin de adaptarlas a nuevas aplicaciones a partir de operadores formales de reuso (*Extend, Rename, Combine y Hide*) aplicados tanto a especificaciones formales como a implementaciones en Eiffel.

4. Transformando especificaciones GSBL^{oo} en código Eiffel

Las especificaciones resultantes de transformar modelos estáticos UML en GSBL^{oo} deben integrarse con código orientado a objetos. Cada PACKAGE GSBL^{oo} contiene clases y relaciones que deben transformarse a código. Se describen a continuación estas transformaciones y se ejemplifican para las clases y relaciones expresados en el diagrama UML de la Figura 3.

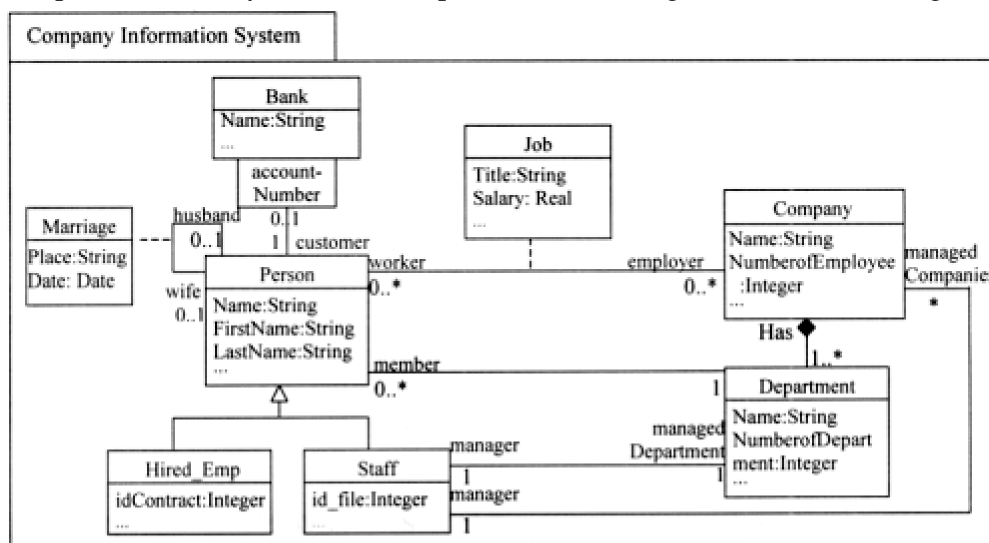


Figura 3. Diagrama de clases UML

4.1. Transformación de clases

El encabezamiento de una OBJECT CLASS en GSBL^{oo} declara el nombre de la clase y puede estar seguido de una lista de parámetros. Los parámetros en GSBL^{oo} pueden restringirse a una clase o sus subclases. En Eiffel, también pueden instanciarse restringiéndolos a clases o sus descendientes. Luego esta traducción es directa.

Las relaciones USES de GSBL^{oo} expresan relaciones cliente-servidor en Eiffel y no tienen traducción explícita. Las clases relacionadas en la cláusula REFINES se expresan a través de la herencia en Eiffel. Éste provee mecanismos para realizar modificaciones sobre las clases heredadas: *rename, redefine, new_exports, undefine y select* que permitirán adaptar clases en base a las relaciones especificadas.

OBJECT CLASS Person **USES** Date, Integer, String, Sex, Boolean ... **EFFECTIVE** **SORT** Person **OPS** inic: Boolean x Boolean x Date x Integer x String x String x Sex x Integer→Person get_Age: Person → Integer get_FirstName: Person → String get_LastName: Person → String ... **EQS** {p:Person; d:Date; b,b1:Boolean;s1,s2: String; i,j:Integer; t-sex:Sex} get_Age(inic(b1,b ,d,i,s1,s2,t-sex,j))=i get_firstName(inic(b1,b ,d,i,s1,s2,t-sex,j))= s1 get_LastName(inic(b1,b ,d,i,s1,s2,t-sex,j))= s2 ... **END-CLASS** **OBJECT CLASS** Staff_Emp **REFINES** Person ... **EFFECTIVE** **SORT** Staff_Emp **OPS** inic: Integer x Boolean x Boolean x Date x Integer x String x String x Sex x Integer→ Staff_Emp get_idFile: Staff_Emp → Integer **EQS** { d:Date; b,b1:Boolean;s1,s2: String; i,j,id:Integer; t-sex:Sex} get_idFile (inic(id,b1,b ,d,i,s1,s2,t-sex,j))= id **END-CLASS**	**class PERSON** creation make feature -- data members for class attributes FirstName, LastName: STRING; Age: INTEGER; .... feature -- operations for class attributes get_Age: INTEGER is do Result:= Age end; --get_Age set_Age(e: INTEGER) is do Age:=e end; --set_Age get_FirstName: STRING is... get_LastName: STRING is... feature{NONE} make (first, last: STRING...) is do FirstName:= first; LastName:= last; ... end -- class PERSON **class STAFF_EMP** inherit PERSON feature -- data members for class attributes idFile: INTEGER; -- operations for class attributes get_idFile: INTEGER is do Result:= idFile end; -- get_idFile set_idFile (number:INTEGER) is do idFile:=number end; -- set_idFile ... end -- class STAFF_EMP

Figura 4. Transformación de OBJECT CLASS

Las claúsulas DEFERRED y EFFECTIVE en GSBL^{oo} declaran “sorts” (géneros) y operaciones de la clase con las ecuaciones que definen su comportamiento. Si una OBJECT CLASS es incompleta, es decir contiene “sorts”, operaciones, en la cláusula DEFERRED, la palabra clave *class* en Eiffel es precedida por la palabra clave *deferred*. Los “sorts” no requieren traducción explícita.

A partir de la signatura de las operaciones se genera la interfaz para los métodos propios de la clase Eiffel (denominados *feature* en Eiffel). La traducción de cada operación tiene un tratamiento diferente según el tipo de *feature* al que haga referencia. Éstos pueden ser funciones, procedimientos, variables o constantes. Para todos los casos se antepone la palabra clave *feature*. También se debe tener en cuenta que de todos los dominios de una operación, el primero que coincida con el “sort” de la clase especificada es el objeto *Current* en Eiffel. El mismo debe ser eliminado de la lista de parámetros del *feature* resultante, teniendo en cuenta la sintaxis de Eiffel.

Las funciones y procedimientos pueden presentar parámetros. Una vez obtenidos los nombres de cada uno de ellos mediante un requerimiento explícito al usuario o extrayéndolo de la especificación, se arma la lista de parámetros de cada *feature*. Tanto las funciones como los procedimientos requieren de un cuerpo delimitado por las palabras claves *do end*, el cual será completado por el usuario en una etapa posterior. Antes de este cuerpo se pueden agregar el comentario y la precondición.

Las precondiciones de la especificación se traducen a precondiciones de los métodos en Eiffel, y los axiomas a postcondiciones e invariantes Eiffel. En Favre (1998) se presentó un análisis de estas transformaciones. En la Figura 4 se muestra la traducción de la OBJECT CLASS *Person* .

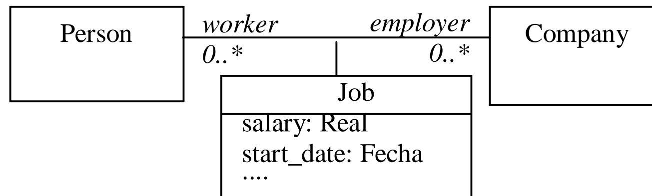
4.2. Transformación de asociaciones

La transformación de asociaciones se realiza automáticamente a partir de esquemas existentes en el nivel de implementación del componente *Association.* Se describirán a continuación la transformación de clases asociación, asociación ordinaria, asociación calificada y composición. Para cada una de ellas se mostrará a partir de un ejemplo (ver Figura 3), la especificación GSBL^{oo} y la traducción a Eiffel obtenida.

Clase Asociación

Cuando dos clases se asocian a través de una *clase asociación*, se agrega al código Eiffel de cada una de ellas un atributo "referencia" a la *clase asociación*, las operaciones *get* y *set* correspondientes a dicho atributo y los invariantes que surgen de las multiplicidades de la asociación.

Por ejemplo la clase *Company* está asociada a *Person* a través de la clase asociación *Job*. En ambas clases se seleccionó el esquema que contiene una lista vinculada de *Job*, debido a que su multiplicidad es muchos a muchos.



CLASS Company……. «Job» **ASSOCIATES** worker: Person …… **OBJECT CLASS** Person... «Job» **ASSOCIATES** employer: Company ...	**ASSOCIATION CLASS** Job **IS** *Association Class* [Person: class1; Company: class2; worker: role1; employer: role2; 0..*: mult1; 0..*: mult2;...] ... **END-CLASS**
class COMPANY ……. worker: LINKED_LIST [JOB] .. get_worker : LINKED_LIST [JOB] is do...end set_worker (j: LINKED_LIST [JOB]) is do...end ... --invariants for association class JOB worker.count >= 0 ... end -- class COMPANY **class PERSON** ... employer: LINKED_LIST[JOB]; ... get_employer : LINKED_LIST [JOB] is do...end set_ employer (j: LINKED_LIST [JOB]) is do...end ... --invariants for association class JOB employer.count >= 0 ... end -- class PERSON	**class JOB** ... worker: PERSON employer: COMPANY ... get_worker: PERSON is do Result:= worker end; get_employer: COMPANY is do Result:= employer end; set_worker (p: PERSON) is do worker:= p end; set_employer (c: COMPANY) is do employer:= c end; get_frozen_employer : BOOLEAN is do Result:= false end; add_only_employer : BOOLEAN is do Result:= false end; changeable_employer : BOOLEAN is do Result:= true end; ...end --class JOB

Asociación simple

Cuando dos clases se asocian a través de una *asociación simple*, se agrega al código Eiffel de cada una de ellas un atributo “referencia” a la otra y las operaciones e invariantes correspondientes a la asociación que surgen de instanciar el esquema correspondiente del componente Association (según su grado, multiplicidad y direccionalidad).

```
OBJECT CLASS  Company ...
«CompanyManager»ASSOCIATES
                    manager:Staff_Emp
...
OBJECT CLASS  Staff_Emp...
« CompanyManager  » ASSOCIATES
                    managedCompanies:Company
...
```

```
ASSOCIATION CompanyManager
IS A_Simple_B_B_1aM [Company: class1;
Staff_Emp: class2; managedCompanies: role1;
manager: role2; 0..*: mult1; 1: mult2]
END-CLASS
```

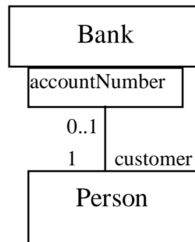
```
class COMPANY creation
        make
feature {NONE}   ...
-- data members for association CompanyManager
        manager: STAFF_EMP;
        mult_manager: MULTIPLICITY;
feature   ..
-- operations for association CompanyManager
get_mult_manager : MULTIPLICITY is ...
get_frozen_manager : BOOLEAN is ...
add_only_manager : BOOLEAN is ...
changeable_manager : BOOLEAN is ...
set_manager (p: STAFF_EMP) is ...
do      manager:= p
end;
get_manager : STAFF_EMP is
do      Result:= manager
end;
...
end -- class COMPANY

class STAFF_EMP inherit PERSON ...
feature {NONE}   ...
-- data members for association CompanyManager
managedCompanies: LINKED_LIST [COMPANY];
mult_managedCompanies: MULTIPLICITY;
..
feature
-- operations for association CompanyManager
get_mult_managedCompanies : MULTIPLICITY is ...
get_frozen_managedCompanies : BOOLEAN is ...
add_only_managedCompanies : BOOLEAN is ...
changeable_managedCompanies : BOOLEAN is ...
cardinality_managedCompanies : INTEGER is...
```

```
set_managedCompanies
            (c:LINKED_LIST[COMPANY])is
require
get_mult_managedCompanies.get_upper >= c.count
do      managedCompanies:= c
end;
get_managedCompanies:LINKED_LIST[COMPANY] is
require cardinality_managedCompanies> 0
do      Result:= managedCompanies
end;
remove_managedCompanies (e: COMPANY) is
require
        is_related_managedCompanies (e) and
        not get_frozen_managedCompanies and
        not add_only_managedCompanies
do      managedCompanies.prune (e);
end;
add_managedCompanies (e: COMPANY) is
require
        is_related_managedCompanies (e) and
        cardinality_managedCompanies <
        get_mult_managedCompanies.get_upper and
        not get_frozen_managedCompanies
do      managedCompanies.put (e);
end;
is_related_managedCompanies(e:COMPANY):
                BOOLEAN is
do      Result:=managedCompanies.has (e);
end;
modify_managedCompanies (e: COMPANY;
            e1: COMPANY) is ...
--invariants  for association Company-StaffEmp
mult_managedCompanies.get_lower = 0;
mult_managedCompanies.get_upper >= 0;
managedCompanies.count >= 0
end -- class STAFF_EMP
```

Asociación Calificada

En el ejemplo hay una asociación calificada entre las clases *Bank* y *Person*. El calificador establece que en conexión con un banco, sólo puede haber un cliente por cada instancia de número de cuenta.



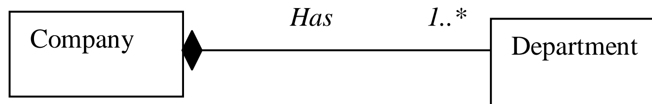
OBJECT CLASS Person... «Person-Bank» **ASSOCIATES** bank: Bank ... **OBJECT CLASS** Bank**...** «Person-Bank» **ASSOCIATES** customer:Person ...	**ASSOCIATION** Person-Bank **IS** Qualified_1a1[Bank:class1; Person:class2; AccountNumber: qualifier; customer: role1; bank:role2; 1:mult1; 0..1: mult2;...] **END-CLASS**

```
class PERSON
...
feature {NONE}
-- data members for association Person_Bank
 bank: BANK;
mult_bank: MULTIPLICITY;
feature
-- operations for association Person_Bank
get_mult_bank: MULTIPLICITY is ...
get_frozen_bank : BOOLEAN is ...
add_only_bank : BOOLEAN is ...
changeable_bank : BOOLEAN is ...
get_bank : BANK is
do          Result:=bank
end;
set_bank (b:BANK )is
do          bank:=b
end;
...
invariant
--invariants  for association Person-Bank
mult_bank.get_lim_inf =1 ORmult_bank.get_lower= 0;
mult_bank.get_upper=1ORmult_bank.get_upper=0
end – class PERSON

class BANK
...
feature{NONE}
-- data members for association Person-Bank
customer:  HASH_TABLE [PERSON,INTEGER];
mult_customer: MULTIPLICITY
feature
...
-- operations for association Person_Bank
get_mult_customer: MULTIPLICITY is ...
get_frozen_customer : BOOLEAN is ...
add_only_customer : BOOLEAN is ...
changeable_customer : BOOLEAN is ...
cardinality_customer : INTEGER is ...
get_customer (id: INTEGER) : PERSON is
do          Result:=customer.item(id)
end;
set_customer (id: INTEGER; p: PERSON) is
do          customer.put(p,id)
end;
get_customer1: HASH_TABLE [PERSON,INTEGER] is
do          Result:=customer
end;
set_customer1(c:HASH_TABLE[PERSON,INTEGER] is
require
            get_mult_customer.get_upper >= c.count
do          customer:=c
end;
remove_customer (id: INTEGER; p: PERSON) is
require     is_related_customer (p) and
            not get_frozen_customer and
            not add_only_customer
do          customer.remove (id);
end;
add_customer (id: INTEGER; p: PERSON) is
require
            is_related_customer (p) and
            not get_frozen_customer
do          customer.put (p,id);
end;
is_related_customer (p: PERSON): BOOLEAN is ...
modify_customer (id: INTEGER; p: PERSON) is ...
...
invariant
--invariants  for association  Person-Bank
mult_customer.get_lower = 1
mult_customer.get_upper = 1
end -- class BANK
```

Composición

Cuando dos clases están vinculadas a través de una composición como por ejemplo las clases *Company* y *Department* , en cada clase existirá un atributo "referencia" a la otra y un atributo multiplicidad de la asociación. Se instancia el esquema correspondiente a la composición el cual contiene todas las operaciones correspondientes a la asociación.



OBJECT CLASS Company.... «Has» **HAS-A NON-SHARED** department: Department ... **OBJECT CLASS** Department... «Has»**HAS-A NON-SHARED** company: Company ...	**WHOLE-PART** Has **IS** Composición_B_B_1aM [Company: Whole; Department: Part; company: role1; department: role2; 1: mult1; 1..*: mult2] **END-CLASS**

class COMPANY ...
feature {NONE}
-- data members for association Has
department: LINKED_LIST [DEPARTMENT];
mult_department: MULTIPLICITY;
...feature
-- operations for association Has
get_mult_department : MULTIPLICITY is ...
get_frozen_department : BOOLEAN is ...
cardinality_department : INTEGER is ...
remove_department (d: DEPARTMENT) is ...
add_department (d: DEPARTMENT) is ...
modify_department (d: DEPARTMENT; d1: DEPARTMENT) is ...
set_department(d: LINKED_LIST[DEPARTMENT])is...
get_department : LINKED_LIST [DEPARTMENT]is ...
... invariant
... –invariants for association Has
mult_department.get_lower = 1;
mult_department.get_upper >= 1;
department.count >= 1
end – class COMPANY
	class DEPARTMENT ...
	feature {NONE}
	-- data members for association Has
	company: COMPANY;
	mult_company: MULTIPLICITY;
	feature
	-- operations for association Has
	get_mult_company : MULTIPLICITY is ...
	get_frozen_company : BOOLEAN is ...
	add_only_company : BOOLEAN is ...
	changeable_company : BOOLEAN is ...
	set_company (c: COMPANY) is
	do company:= c
	end;
	get_company : COMPANY is
	do Result:= company
	end;
	invariant
	--invariants for association Has
	mult_company.get_lower = 1;
	mult_company.get_upper =1;
	end -- class DEPARTMENT

5. Conclusiones

Se describieron en este trabajo las bases de un proceso riguroso para la generación sistemática de código a partir de modelos UML y se analizó en detalle una de las etapas de este proceso: la generación de código Eiffel a partir de especificaciones algebraicas. En particular, se describió la transformación automática de diferentes tipos de asociaciones. Toda la información contenida en los modelos UML (clases, asociaciones, cardinalidad, OCL constraints, etc) es volcada en especificaciones y tendrá implicancias de implementación. Si un diagrama de clases se especificó a partir de componentes *SpReIm*, también las implementaciones se construirán reusando subcomponentes *SpReIm* del nivel de implementación. Las transiciones entre los diagramas UML y todas las versiones intermedias se realizan mediante la aplicación de transformaciones que preservan la integridad entre especificaciones y código.

Algunas etapas claves de este método fueron prototipadas: la transformación de

especificaciones algebraicas a código Eiffel y la transformación de OCL a GSBL^{oo}. Actualmente se está prototipando el método de ingeniería "forward" propuesto y analizando su integración con los procesos soportados por herramientas CASE existentes, por ejemplo *Rational Rose™*.

Referencias

Barbier, F.;Henderson-Sellers, B.;Opdahl, A.; Gogolla, M. (2001) The Whole Part Relationship in the Unified Modeling Language:A New Approach. In: (K.Siau and T. Halpin), Chapter 12. Unified Modeling Language: System Analysis, Design and Development Issues, Idea Group Publishing. USA.

Booch, G.; Rumbaugh, J.; Jacobson, I.(1999) The Unified Modeling Language User Guide. Addison-Wesley, 1999.

Bourdeau, R.; Cheng, B.(1995) A Formal Semantics for Object Model Diagrams. IEEE Transactions on Software Engineering, 21 (10), 799-821.

Breu, R.; Hinkel, U.; Hofmann, C.; Klein, C.; Paech, B.; Rumpe,B.; Thurner, V.(1997) Towards a Formalization of the Unified Modeling Language. TUM-I9726 Technische Universitat Munchen.

Bruel, J.; France, R. (1998) Transforming UML Models to Formal Specifications. In : Proc. of UML'98-Beyond the notation, Lecture Notes in Computer Science 1618, Springer Verlag, 78-92.

Clérici, S.; Orejas, F.(1988) GSBL: An Algebraic Specification Language Based on Inheritance. In: Proc. of the European Conference on Object-oriented Programming ECOOP 88, 78-92.

D'Souza,D.; Wills, A.(1999) Objects, Components, and Frameworks with UML. Addison Wesley.

Evans, A.; France, R.; Lano, K.; Rumpe, B. (1998) The UML as a Formal Modeling Notation. In: Proc. of UML'98-Beyond the Notation, Lecture Notes in Computer Science 1618. Springer.

Favre, L. (1998) Object-oriented Reuse through Algebraic Specifications In: Technology of Object-Oriented Languages and Systems, TOOLS 28, IEEE Computer Society, 101-112.

Favre, L.; Martinez, L.; Pereira, C. (2000) Transforming UML Static Models to Object Oriented Code. Technology of Object-Oriented Languages and Systems, TOOLS 37, IEEE.

Favre, L; Clérici, S.(2001) A Systematic Approach to Transform UML static Models to Code. In: (K.Siau and T. Halpin), Chapter II. Unified Modeling Language: System Analysis, Design and Development Issues, Idea Group Publishing, USA.

Firesmith, D.; Henderson-Sellers, B.(1998) Clarifying specialized forms of association in UML and OML. JOOP,11(2) , 1998, 47-50.

France, R.; Bruel, J.; Larronde-Petri, M.(1997) An Integrated Objet-Oriented and Formal Modeling Notations, JOOP, Nov/Dec, 25- 34.

Gogolla, M.; Ritchers, M.(1997) On combining Semi-formal and Formal Object Specification Techniques. In: Proc. WADT97, Lecture Notes in Computer Science 1376, Springer, 238-252.

Hussmann, H.; Cerioli, M.; Reggio, G.; Tort, F.(1999) Abstract Data Types and UML Models. Report DISI-TR-99-15, University of Genova.

Kim, S.; Carrington, D.(1999) Formalizing the UML Class Diagram using Object-Z. In: Proc. UML 99, Lecture Notes in Computer Science 1723, 83-98.

Lano, K (1995) Formal Object-Oriented Development. Springer-Verlag.

Mandel, L.; Cengarle, V.(1999) On the Expressive Power of the Object Constraint Language OCL. Available: http://www.fast.de/projeckte/forsoft/ocl.

Meyer, B (1997) Object-oriented Software Construction. Prentice Hall.

OMG (1999) Unified Modeling Language Specification, v. 1.3. document ad/99-06-08, Object Management Group.

Overgaard, G.(1998) A Formal Approach to Relationships in the Unified Modeling Language. In: Proc. of Workshop on Precise Semantic of Modeling Notations, International Conference on Software Engineering. ICSE'98, Japan.

Padawitz, P.(2000) Swinging UML: How to Make Class Diagrams and State Machines Amenable to Constraint Solving and Proving.In: (Evans, A.; Kent,S.) Proc. of <<UML>> 2000-The Unified Modeling Language. Lecture Notes in Computer Science 1939. Springer 162-177.

Richters, M.; Gogolla, M.(2000) Validating UML Models and OCL Constraints. In: (Evans, A. ;Kent,S.) Proc. of <<UML>> 2000. The Unified Modeling Language, Lecture Notes in Computer Science 1939. Springer, 265-277.

Varizi, M.; Jackson, D.(1999) Some Shortcomings of OCL, The Object Constraint Language of UML. Available: http://sdg.lcs.mit.edu/~dnj/publications.htm.