IDEA GROUP PUBLISHING

701 E. Chocolate Avenue, Hershey PA 17033-1117, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITP4252

Forward Engineering and UML: From UML Static Models to Eiffel Code

Liliana Favre*, Liliana Martinez and Claudia Pereira
INTIA, Facultad de Ciencias Exactas, Universidad Nacional del Centro de la Provincia de Buenos Aires, Argentina
* CIC (Comision de Investigaciones Cientificas de la Provincia de Buenos Aires, {lfavre, lmartine, cpereira}@exa.unicen.edu.ar

ABSTRACT

In a previous research work we have proposed a rigorous process to forward engineering UML static models. This approach is based on the integration of semi-formal notations in UML, algebraic specifications and object-oriented code. The GSBL^{OO} language was defined to cope with concepts of UML models. The emphasis is given to the last steps in the road from UML to code: we describe how to transform GSBL^{OO} specifications into object-oriented code. Eiffel was the language chosen to demonstrate the feasibility of our approach. In particular, we analyze the transformation of different kinds of associations and the generation of Eiffel assertions.

INTRODUCTION

Unified Modeling Language (UML) has emerged as a standard modeling language in the object-oriented analysis and design world. It is a set of graphical and textual notations for specifying, visualizing and documenting object-oriented systems (OMG, 1999; Booch 1999).

There are CASE tools which offer code automatic generation and reverse engineering from object-oriented models. They present problems due to the lack of formal semantics of UML models and these models are semantically richer than the object-oriented languages, though.

These problems have motivated the analysis of different approaches to give semantics to the UML notations. The UML formalization is an open problem still and many research groups have already achieved the formalization of parts of the language. The Precise UML Group, pUML, is created in 1997 with the goal of giving precision to UML (Evans et al. 1998). Different results give semantics to UML subsets based on different formalisms (Lano 1995; Breu et al.1997; Bruel and France, 1998; Gogolla and Ritchers, 1997; Kim and Carrington, 1999; Overgaard, 1998; Barbier et. al., 2001). Currently, there are few methods that include OCL, Catalysis is the most important reference (D'Souza and Wills, 1999). Other research works propose the integration of UML with OCL (Ritchers and Gogolla, 2000). Siau and Halpin (2001) and JDM (2000) identify some problematic aspects of UML and propose possible solutions.

A variety of advantages have been attributed to the use of formal software specifications to solve these problems. A formal specification can reveal gaps, ambiguities and inconsistencies. However, formal specifications alone do not address the need of industrial practitioners, who require an understandable and scalable semantics that can be integrated by using tools.

Favre and Clérici (2001) propose a rigorous process to forward engineering UML static models using the algebraic language GSBL^{OO}. The contribution was towards an embedding of the code generation within a rigorous process that facilitates reuse. The GSBL^{OO} language was designed to cope with concepts of the UML models. This language is relation-centric, it expresses different kinds of relations as primitives to develop specifications. The transformation of UML static models specified in OCL into GSBL^{OO} and a system of transformation rules were described. The formal model *SpReIm* for defining structured collections of reusable components that integrates algebraic specifications and object-oriented code was defined. The manipulation of *SpReIm* components by means of building operators is the basis for reusability.

The emphasis is given to the last steps in the road from UML to code. Eiffel was chosen to prove the feasibility of our approach. It is based on the "Design by Contract" principle. Contracts imply obligations and benefits for clients and contractors, and are made explicit by the use of Eiffel assertions. These features facilitate to integrate axioms of specifications with object oriented code. We describe how to transform GSBL^{OO} specification into Eiffel, analyzing the transformation of different kinds of relations and the generation of Eiffel assertions.

This paper has the following structure: -Section 2 describes the basis of a rigorous forward engineering method, -Section 3 outlines GSBL^{OO} language features, -Section 4 analyzes how to transform GSBL^{OO} specifications into Eiffel code, -Section 5 considers conclusions.

MOTIVATION

There are CASE tools for code generation starting from UML models. Unfortunately, the current techniques available in these tools are not enough for the complete automated generation of source code. As an example, Rational Rose™ will be analyzed (Quatrani, 1998). This allows generating databases definitions, class interfaces and relations in which the class participates. The current modeling languages available in Rational Rose™ (Booch, OMT, UML) do not have a precisely defined semantics. This hinders the code generation processes. Another source of problems in these processes is that, on one hand, the UML models contain information that can not be explicited in object-oriented languages and on the other, the object-oriented languages express implementation characteristics that have no counterpart in the UML models. For instance, languages like C++, Java and Eiffel do not allow explicit associations, their cardinality and their constraints. It is the responsibility of the designer to make good use of this information, selecting from a limited repertoire of implementations or implementing the association by himself. The forward and reverse processes in Rational Rose™ are facilitated by means of the insertion of annotations in the generated code. These annotations are the link between the model elements and the language, they should be kept intact and not changed. It is the programmer's responsibility to know what he can modify and what he can not.

The programmer, to solve implementation problems, can modify the code by adding or removing classes, modifying class attributes or operations, changing operation signatures, etc. These code modifications require the programmer's ability to keep the integration between model and language.

Moreover, the existing tools do not exploit all the information contained in the UML models, for instance, cardinality and constraints of associations, preconditions, postconditions and class invariants in OCL are only translated as annotations. Assertions in OCL could be translated to assertions in an object-oriented language that supports them, like Eiffel. Furthermore, reuse is based on object-oriented language libraries and not on specifications that describe relations between classes and their operations free from implementation details.

To overcome these problems, a rigorous process to forward engineering UML static models using the algebraic language GSBL^{OO} was proposed (Favre and Clérici, 2001).

Starting from UML class diagrams, an incomplete algebraic specification can be automatically built by instantiating reusable schemes.

This paper appears in *Issues and Trends of Information Technology Management in Contemporary Organizations,* the proceedings of the Information Resources Management Association International Conference. Copyright © 2002, Idea Group Inc.

This specification contains the highest information obtained by translating the UML constructions and OCL constraints to GSBL^{OO}.

Preconditions, postconditions and invariants in OCL will be translated to GSBL^{OO}. The process of transformation is supported by a transformation rules system (Favre et. al, 2000).

Transformations are based on a reusable component library defined by the *SpReIm* model. (Favre and Clérici, 2001). This model allows defining structured collections of reusable components that integrate algebraic specifications and object-oriented code. It takes advantage of the algebraic formalism power to describe behavior in an abstract way integrating them with concrete implementations. It consists of three abstraction levels:

- *Specialization*: it describes a hierarchy of incomplete specifications related by specialization relationships through two views: one based on OCL specifications and the other on GSBL^{OO}.
- *Realization*: it describes a hierarchy of complete specifications related by realization relationships.
- *Implementation*: it relates hierarchy of concrete classes schemes in an object-oriented language.

The reuse of components is based on the application of operators. Reuse operators were defined on the three levels of the *SpReIm* model.

Thus, an algebraic specification can be semi-automatically built. It can be used to detect inconsistencies in the class diagrams. The GSBL^{OO} specifications must be integrated with object-oriented code. The emphasis in this paper is given to the transformation of GSBL^{OO} to Eiffel.

FORMALIZING UML STATIC MODELS IN THE GSBL^{OO} LANGUAGE

The existing algebraic specification languages do not possess specific constructions to express all kinds of relations in UML (dependency, association, generalization and realization). These are generally buried in client and inheritance relations. However, associations are semantics constructions of equal weight to the classes and generalizations in the UML models, and should not be treated just as implementation constructions. In fact, the associations allow abstracting the interaction between classes in the design on large systems and they affect the partition of the systems in modules. Since an integrated method requires common structuring mechanisms for object-oriented models and algebraic specifications, the GSBL^{OO} language has been defined. It enriches GSBL (Clérici, 1988). The ability to specify deferred and effective parts, inheritance relations among classes and mechanisms such as implicit parameterization to support reuse of specifications and their incremental development are among the most important features of GSBL.

GSBL^{OO} extends GSBL with constructions that allow expressing different kinds of UML relations. OBJECT CLASS and ASSOCIATION class specify classes and associations (ordinary, qualified, association-class) respectively.

An OBJECT CLASS distinguishes incomplete and complete parts. The DEFERRED clause declares new sorts, operations or equations incompletely defined. The EFFECTIVE clause either declares new sorts, operations or equations completely defined, or completes the definition of some inherited sort or operation. Sorts and operations are declared in the SORTS and OPS clauses. In GSBL^{OO}, it is possible to specify partial functions restricting operations by preconditions.

GSBL^{OO} expresses dependencies in the context of classes by means of the USES clause. The mechanisms for defining inheritance are the RESTRICT and REFINES clauses. The REFINES clause relies on the module viewpoint of classes and the RESTRICTS clause on the type viewpoint of a class. Both of them reflect IS-A relations between abstractions in the external model of any software system. Associations are defined as standard elements in GSBL^{OO}.

Generic relations can be used in the definition of concrete relations by instantiation. New associations and whole-part relations (aggregation and composition) can be defined by means of the following syntax:

```
ASSOCIATION <relationName>
IS <constructorTypeName>[...:Class1;..:Class2; ...:Role1;...:Role2; ...: mult1;...:mult2;
                        ..:visibility1;...:visibility2 ]
CONSTRAINED BY <constraintList>
END
```

```
WHOLE-PART <relationName>
IS <constructorTypeName> [...: Whole;...: Part; ;...:Role1;...:Role2;...:  mult1;...:mult2;
                          ...:visibility1;...:visibility2]
CONSTRAINED BY <constraintList>
END
```

The IS clause expresses the instantiation of *<constructorTypeName>* with classes, roles, visibility and multiplicity. The CONSTRAINED-BY clause allows the specification of static constraints in first order logic. Relations are defined in an Object Class by means of the following syntax:

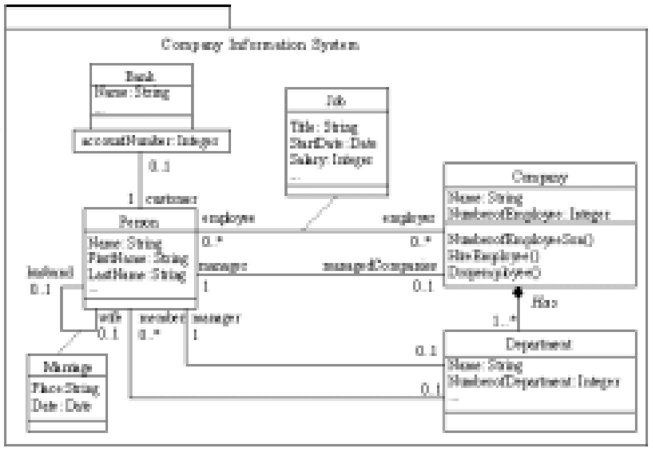
```
OBJECT CLASS C...
        "<relationName>"ASSOCIATES <className>
        "<relationName>" HAS-A SHARED <className>
        "<relationName>" HAS-A NON-SHARED <className>
           ...
END-CLASS
```

The keywords ASSOCIATES and HAS-A identify ordinary association or aggregation respectively. The keywords SHARED and NON-SHARED refer to simple aggregation and composition respectively. An association may be refined to have its own set of operations and properties. Such an association is called an Association Class.

The PACKAGE is the mechanism provided by GSBL^{OO} for grouping classes, and controls its visibility. It matches the UML semantics. Classes and their relations from a system design into a series of packages might be separated using the GSBL^{OO} import dependences to control access among these packages.

Figure 1 shows an information system for a company. There is an association between *Person* and *Company,* specifying that managers (instances of *Person*) manage companies. Every manager may manage only one department and every company may have only one manager. There is a qualified association between *Person* and *Bank.* In the context of *Bank*, an *accountNumber* (qualifier) identifies a particular customer. In an *employer/employee* relation between *Company* and *Person*, there is a *Job* that represents the properties of that relation, which applies to exactly one pairing of *Person* and *Company*. Figure 2 partially shows the GSBL^{OO} specification of Figure 1.

Figure 1: Company information system. A UML class diagram.



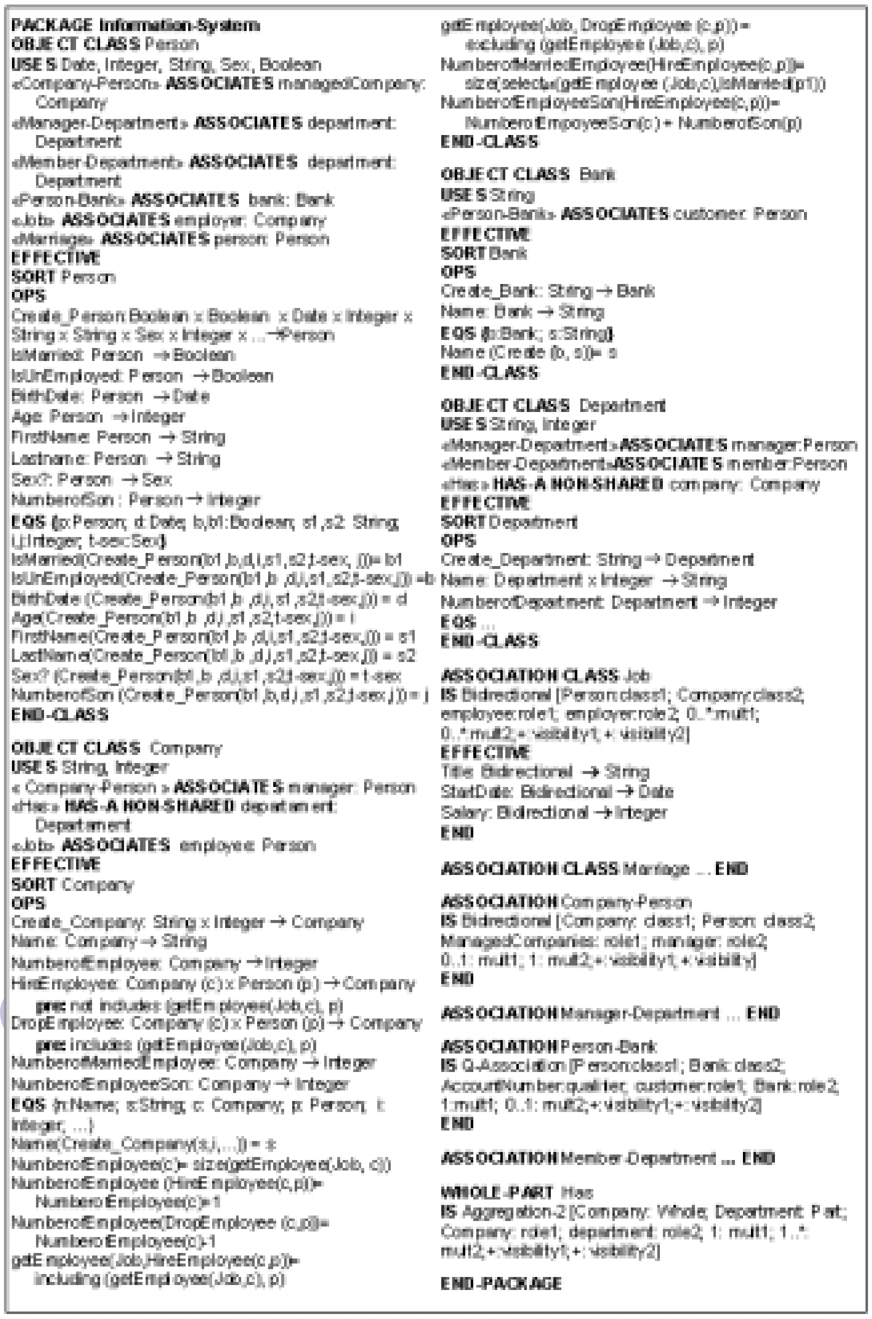


Figure 2: Information system company. A GSBL^{oo} specification

```
PACKAGE Information-System
OBJECT CLASS Person
USES Date, Integer, String, Sex, Boolean
«Company-Person» ASSOCIATES managedCompany:
    Company
«Manager-Department» ASSOCIATES department:
    Department
«Member-Department» ASSOCIATES department:
    Department
«Person-Bank» ASSOCIATES bank: Bank
«Job» ASSOCIATES employer: Company
«Marriage» ASSOCIATES person: Person
EFFECTIVE
SORT Person
OPS
Create_Person: Boolean x Boolean x Date x Integer x
String x String x Sex x Integer x ...→Person
IsMarried: Person →Boolean
IsUnEmployed: Person →Boolean
BirthDate: Person →Date
Age: Person →Integer
FirstName: Person → String
Lastname: Person → String
Sex?: Person → Sex
NumberofSon : Person → Integer
EQS {p:Person; d:Date; b,b1:Boolean; s1,s2: String;
i,j:Integer; t-sex:Sex}
IsMarried(Create_Person(b1,b,d,i,s1,s2,t-sex, j))= b1
IsUnEmployed(Create_Person(b1,b ,d,i,s1,s2,t-sex,j)) =b
BirthDate (Create_Person(b1,b ,d,i,s1,s2,t-sex,j)) = d
Age(Create_Person(b1,b ,d,i,s1,s2,t-sex,j)) = i
FirstName(Create_Person(b1,b ,d,i,s1,s2,t-sex,j)) = s1
LastName(Create_Person(b1,b ,d,i,s1,s2,t-sex,j)) = s2
Sex? (Create_Person(b1,b ,d,i,s1,s2,t-sex,j)) = t-sex
NumberofSon (Create_Person(b1,b,d,i,s1,s2,t-sex,j)) = j
END-CLASS

OBJECT CLASS Company
USES String, Integer
« Company-Person » ASSOCIATES manager: Person
«Has» HAS-A NON-SHARED department:
    Department
«Job» ASSOCIATES employee: Person
EFFECTIVE
SORT Company
OPS
Create_Company: String x Integer → Company
Name: Company → String
NumberofEmployee: Company → Integer
HireEmployee: Company (c) x Person (p) → Company
    pre: not includes (getEmployee(Job,c), p)
DropEmployee: Company (c) x Person (p) → Company
    pre: includes (getEmployee(Job,c), p)
NumberofMarriedEmployee: Company → Integer
NumberofEmployeeSon: Company → Integer
EQS {n:Name; s:String; c: Company; p: Person; i:
Integer; ...}
Name(Create_Company(s,i,...)) = s
NumberofEmployee(c)= size(getEmployee(Job, c))
NumberofEmployee (HireEmployee(c,p))=
    NumberofEmployee(c)+1
NumberofEmployee(DropEmployee (c,p))=
    NumberofEmployee(c)-1
getEmployee(Job,HireEmployee(c,p))=
    including (getEmployee(Job,c), p)
getEmployee(Job, DropEmployee (c,p))=
    excluding (getEmployee (Job,c), p)
NumberofMarriedEmployee(HireEmployee(c,p))=
    size(select(p1|getEmployee (Job,c)).IsMarried(p1))
NumberofEmployeeSon(HireEmployee(c,p))=
    NumberofEmpoyeeSon(c) + NumberofSon(p)
END-CLASS

OBJECT CLASS Bank
USES String
«Person-Bank» ASSOCIATES customer: Person
EFFECTIVE
SORT Bank
OPS
Create_Bank: String → Bank
Name: Bank → String
EQS {b:Bank; s:String}
Name (Create (b, s))= s
END-CLASS

OBJECT CLASS Department
USES String, Integer
«Manager-Department» ASSOCIATES manager:Person
«Member-Department» ASSOCIATES member:Person
«Has» HAS-A NON-SHARED company: Company
EFFECTIVE
SORT Department
OPS
Create_Department: String → Department
Name: Department x Integer → String
NumberofDepartment: Department → Integer
EQS ...
END-CLASS

ASSOCIATION CLASS Job
IS Bidirectional [Person:class1; Company:class2;
employee:role1; employer:role2; 0..*:mult1;
0..*:mult2;+:visibility1; +: visibility2]
EFFECTIVE
Title: Bidirectional → String
StartDate: Bidirectional → Date
Salary: Bidirectional → Integer
END

ASSOCIATION CLASS Marriage ... END

ASSOCIATION Company-Person
IS Bidirectional [Company: class1; Person: class2;
ManagedCompanies: role1; manager: role2;
0..1: mult1; 1: mult2;+:visibility1; +:visibility]
END

ASSOCIATION Manager-Department ... END

ASSOCIATION Person-Bank
IS Q-Association [Person:class1; Bank: class2;
AccountNumber:qualifier; customer:role1; Bank:role2;
1:mult1; 0..1: mult2;+:visibility1;+: visibility2]
END

ASSOCIATION Member-Department ... END

WHOLE-PART Has
IS Aggregation-2 [Company: Whole; Department: Part;
Company: role1; department: role2; 1: mult1; 1..*:
mult2;+:visibility1;+: visibility2]

END-PACKAGE
```

TRANSFORMING GSBL^{OO} SPECIFICATIONS IN EIFFEL CODE

The resulting specifications of transforming UML static models in GSBL^{OO} must be integrated with an object-oriented code. These transformations are described below and they are exemplified for the classes and relationships expressed in the UML diagram in Figure 1.

Transformation of Classes

The algebraic specification obtained in the previous step must be transformed into Eiffel code. To achieve this, every clause and relation present in a GSBL^{OO} OBJECT CLASS specification was analyzed.

GSBL^{OO} and Eiffel have the same syntax for the declaration of class parameters. Then, this transformation is reduced to a trivial translation.

The relation introduced in GSBL^{OO} using the clause USES will be translated into a client relation in Eiffel. The relation expressed through the keywords REFINES and RESTRICTS in GSBL^{OO} will become an inheritance relation in Eiffel. This provides mechanisms to carry out modifications on the inherited classes that will allow adapting them.

The DEFERRED and EFFECTIVE clauses in GSBL^{OO} declare sorts and operations of the class with the equations that define their behavior. If an OBJECT CLASS is incomplete, i.e., it contains sorts and operations in the clause DEFERRED, the keyword *class* in Eiffel is preceded by the keyword *deferred*. Sorts do not require explicit translation.

From the signature of the operations, the interfaces for the methods of the Eiffel class are generated (*feature* in Eiffel). The translation of each operation has a different treatment according to the type of feature to which it makes reference (functions, procedures, variables or constants). It should also be considered that of all the domains of an operation, the first one that coincides with the sort of the specified class is the object *Current* in the object-oriented language and it should be eliminated from the list of parameters of the resulting feature.

Functions and procedures can present arguments. Once each name is obtained, either by an explicit requirement to the user or by extracting it from the specification, the list of arguments of each feature is built. Functions and procedures require a body defined by the keywords *do end*, which will be completed by the user.

Transformation of Axioms

Eiffel provides an assertion language. Assertions are boolean expressions expressing semantics properties of the classes. They "serve to express the specification of software components: indications of *what* a component does rather than *how* it does it" (Meyer 97). They can play the following roles:

- Precondition: expresses the requirements the client must satisfy to call a routine.
- Postcondition: expresses the conditions the routine guarantees on return.
- Class invariant: expresses the requirements every object of the class must satisfy after its creation.

Preconditions and axioms of a function written in GSBL^{OO} are used to generate preconditions and postconditions for routines and invariants for Eiffel classes.

It is worth clarifying for the assertion generation that a basic functionality *f: s x a1 x a2 x ......an,* where *s* is the sort of interest, is translated into Eiffel syntax as *f(a1,a2,....an)*.

A GSBL^{OO} precondition, which is a well-formed term defined over functions and constants of the global environment classes, is automatically translated to Eiffel method precondition.

Axioms in a formal specification language represent the constraints the class introduces on the operations. Axioms are translated to Eiffel postconditions and invariants.

The system can automatically derive an invariant if it can establish a correspondence between the functions in the axiom and the class features that only depend on the object state.

For instance, the axiom:
NumberofEmployee (HireEmployee(c,p))= NumberofEmployee(c)+1
it is translated to *NumberOfEmployee= old NumberOfEmployee+1*

A postcondition can automatically be generated from one axiom if a term *e(<list-of-arguments>),* associated to an operation, can be distinguished within itself in such a way that any other term of the axiom depends upon the *<list-of-arguments>* or constants. Then, the postcondition will associate itself with the method that reflects the term and will obviously depend only upon the previous state of the method execution, upon the state after its execution and upon the method arguments.

Another type of situation cannot be automatically derived. These cases do not enable the system to build an assertion without the user's interventions. The programmer can also incorporate assertions that reflect purely implementation aspects. A detailed description and examples may be found in Favre (1998).

Transformation of Associations

Associations are transformed by instantiating reusable schemes that express how to implement them. These schemes are linked to the different relation constructor types in GSBL^{OO}.

The transformation of associations is automatically derived starting from existent schemes in the level of implementation of the Association component.

The specialization level describes the different relations through incomplete specifications classified according to its kind, degree and connectivity. The realization level describes a hierarchy of specifications associated to different realizations. For instance, for an association (binary, bi-directional and many-to-many) different realizations through hashing, sequences or trees could be associated. The implementation level associates each leaf of the realization level to different implementations in an object-oriented language.

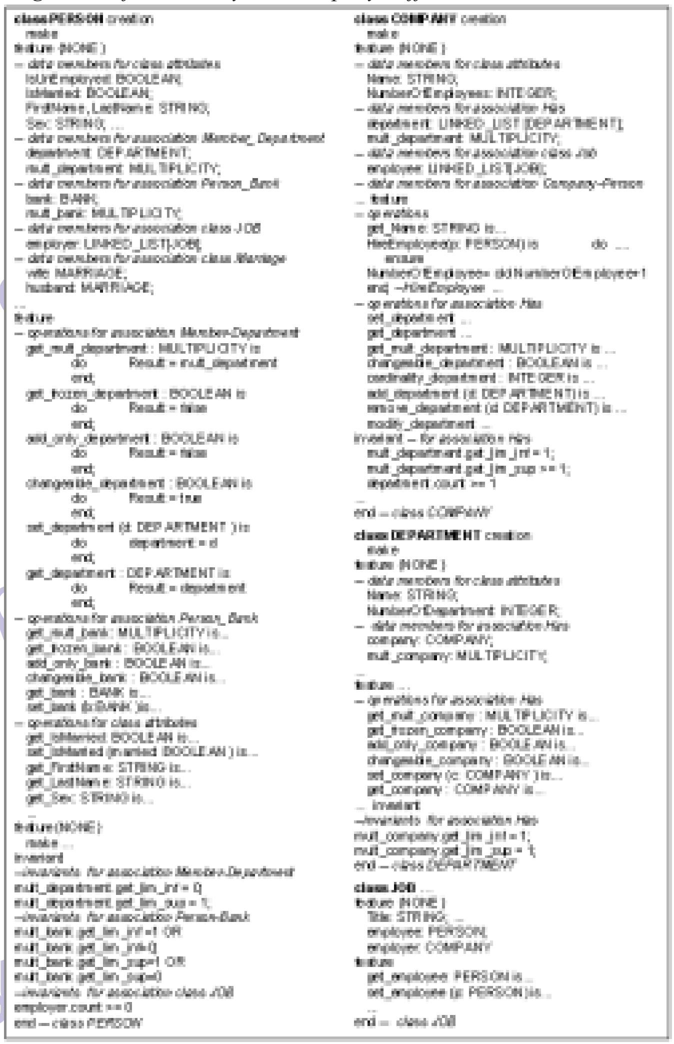
Implementation sub-components express how to implement associations and aggregations. For example, a bi-directional binary association with multiplicity "one-to-one" will be implemented as an attribute in each associated class containing a reference to the related object. On the other hand, if the association is "many-to-many" the best approach is to implement the association as a different class, in which each instance represents one link and its attributes.

For each HAS-A clause, an implementation scheme will be selected and the "aggregate" and the "part" will be instantiated. For example, if the aggregation is "one-to-many", for an attribute in the "aggregate", a reference to a sequence of pointers to the "part" will be generated.

Analogously, for every ASSOCIATES clause, a scheme in the implementation level of the Association component will be selected and instantiated. In these cases, the implementation level schemes suggest including "reference" attributes in the classes or introducing an intermediate class or container. Notice that the transformation of an association does not necessarily imply the existence of an associated class in the generated code, as an efficient implementation can suggest including "reference" attributes in the involved classes.

Figure 3 partially shows the Eiffel resulting code of transforming the GSBL^{OO} specifications of the company system.

Figure 3: Information system company. Eiffel code



```
class PERSON creation
   make
feature {NONE}
-- data members for class attributes
   IsUnEmployed: BOOLEAN;
   IsMarried: BOOLEAN;
   FirstName, LastName: STRING;
   Sex: STRING; ...
-- data members for association Member_Department
   department: DEPARTMENT;
   mult_department: MULTIPLICITY;
-- data members for association Person_Bank
   bank: BANK;
   mult_bank: MULTIPLICITY;
-- data members for association class JOB
   employer: LINKED_LIST[JOB];
-- data members for association class Marriage
   wife: MARRIAGE;
   husband: MARRIAGE;
...
feature
-- operations for association Member-Department
   get_mult_department : MULTIPLICITY is
            do        Result = mult_department
            end;
   get_frozen_department : BOOLEAN is
            do        Result = false
            end;
   add_only_department : BOOLEAN is
            do        Result = false
            end;
   changeable_department : BOOLEAN is
            do        Result = true
            end;
   set_department (d: DEPARTMENT ) is
            do        department = d
            end;
   get_department : DEPARTMENT is
            do        Result = department
            end;
-- 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...
   set_bank (b:BANK )is...
-- operations for class attributes
   get_IsMarried: BOOLEAN is...
   set_IsMarried (married: BOOLEAN ) is...
   get_FirstName: STRING is...
   get_LastName: STRING is...
   get_Sex: STRING is...
   ...
feature{NONE}
   make ...
invariant
--invariants for association Member-Department
mult_department.get_lim_inf = 0;
mult_department.get_lim_sup = 1;
--invariants for association Person-Bank
mult_bank.get_lim_inf =1 OR
mult_bank.get_lim_inf=0
mult_bank.get_lim_sup=1 OR
mult_bank.get_lim_sup=0
--invariants for association class JOB
employer.count >= 0
end -- class PERSON

class COMPANY creation
   make
feature {NONE}
-- data members for class attributes
   Name: STRING;
   NumberOfEmployees: INTEGER;
-- data members for association Has
   department: LINKED_LIST [DEPARTMENT];
   mult_department: MULTIPLICITY;
-- data members for association class Job
   employee: LINKED_LIST[JOB];
-- data members for association Company-Person
... feature
-- operations
   get_Name: STRING is...
   HireEmployee(p: PERSON) is            do ...
      ensure
   NumberOfEmployee= old NumberOfEmployee+1
   end; --HireEmployee ...
-- operations for association Has
   set_department ...
   get_department ...
   get_mult_department : MULTIPLICITY is ...
   changeable_department : BOOLEAN is ...
   cardinality_department : INTEGER is ...
   add_department (d: DEPARTMENT) is ...
   remove_department (d: DEPARTMENT) is ...
   modify_department ...
invariant -- for association Has
   mult_department.get_lim_inf = 1;
   mult_department.get_lim_sup >= 1;
   department.count >= 1
...
end -- class COMPANY

class DEPARTMENT creation
   make
feature {NONE}
-- data members for class attributes
   Name: STRING;
   NumberOfDepartment: INTEGER;
--  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...
   get_company : COMPANY is...
... invariant
--invariants for association Has
mult_company.get_lim_inf = 1;
mult_company.get_lim_sup = 1;
end -- class DEPARTMENT

class JOB ...
feature {NONE}
   Title: STRING; ...
   employee: PERSON;
   employer: COMPANY
feature
   get_employee: PERSON is...
   set_employee (p: PERSON )is...
   ...
end -- class JOB
```

CONCLUSIONS

In this research work the basis of a rigorous process for the systematic code generation starting from UML model were described. One of the stages of this process: -the Eiffel code generation starting from algebraic specifications was analyzed in detail. In particular, the automatic transformation of different types of associations was described. All the UML model information (classes, associations, cardinality, OCL constrains, etc) is overturned in specifications having implementation implications. If a class diagram is specified starting from *SpReIm* component, the implementations will also be built reusing *SpReIm* subcomponents of the implementation level. The transitions among the UML diagrams and all the intermediate versions are carried out by applying transformations that preserve the integrity between specifications and code.

Some key steps of this method were prototyped: the transformation of algebraic specifications to Eiffel code and the transformation of OCL to GSBL^{OO}. Nowadays the prototype of the method of forward engineering proposed is being implemented and its integration with the processes supported by the existent CASE tools is being analyzed.

REFERENCES

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.

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.: Object-oriented Reuse through Algebraic Specifications In: Proc. of Technology of Object-Oriented Languages and Systems, TOOLS 28, IEEE Computer Society, 1998; S 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 Computer Society. ISBN 0-765-0918-5 (170-181).

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.

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.

JDM (2000) Journal of Database Management, 11(4) Idea-Group Publishing

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.

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.

Quatrani, T. (1998) Visual Modeling with Rational Rose and UML. Addison Wesley.

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.

Siau, K.; Halpin, T. (2001) Unified Modeling Language: System Analysis, design and Development Issues. K. Siau; T. Halpin (eds) Idea-Group Publishing.