From Object-Oriented Design Towards Object-Oriented Programming

Naoufel Kraïem*, Faïez Gargouri**, Faouzi Boufarès***

* Laboratoire CRI, Université de Paris I, 17 rue Tolbiac, 75013 Paris - naoufel@masi.ibp.fr
** Laboratoire CEDRIC-IIE, 18 Allée J. Rostand, 91025 Evry - gargouri@rubis.iie.cnam.fr
*** IUT Villetaneuse, Université de Paris XIII, Av. J.B. Clément, 93430 Villetaneuse

Abstract : *The emergence of the object philosophy in the new software development techniques gave birth to many object models. The object-oriented approach enables the improvement of software quality, the reduction of future maintenance requirements, the reuse and the adaptation of specification and developments. However the difficulty lies in the transition between the conceptual specification and the implementation because of the disparity of the formalism proper to each level. To resolve the problem, we propose an object oriented interface supported by a software tool and based on a pivot model and a set of mapping rules.*

1 Introduction

The Object-Oriented approach emerges in certain number of data processing domains, such as programming, software engineering, data base, DBMS, analysis and design of data base and information system. The paradigm underlying the computational object-oriented are stabilised enough to consider that they are providing a unifying approach for information system development.
However the development of object-oriented applications remains problematic.
Object-oriented design methodologies are focusing on system design as a later stage of the application life cycle, implying that the earliest stage leading to requirements specification and conceptual design, have been performed.
Object-oriented analysis methodologies are still under investigation. Three main approaches are being proposed:

- the functional approach uses traditional DFD based techniques to derive object specification
- the data driven approaches are influenced by E/R modelling to define objects
- the object based approaches recommend the use of the object concept right from the beginning of the system life cycle. The concept of object is then the basic element the system relies on.

The claim of these approaches is that enhancements and extensions of the computational object concept are required to make it relevant to conceptual modelling.
O* [6], MCO [7], (OOD, GOOD) [3,4] HOOD [11] and OFM [1] are examples of approaches to support conceptual modelling in an object-oriented way.
Our aim is to bridge the gap between object-oriented conceptual modelling and object-oriented implementation. To do so, we propose an interface supported by a software tool and based on a pivot model and a set of mapping rules.

This article is organized as follows. Section 2 presents a very brief summary of the O* and MCO models. The third section, describes the object-oriented interface (O2I) and the mapping rules from object-oriented design towards object-oriented implementation and some concluding remarks are presented in section4.

2 Overview of MCO and O* Models

2.1 Main MCO Concepts

MCO [8] is an object-oriented methodology for the design of information systems. It allows designers to use the object-oriented paradigm from the early steps of design to modelize the universe of discourse of any system or organization (business or scientific application). Here after, we present the main concepts of MCO model.

An agent is a concrete or abstract entity having a set of acquaintances, which provides or requires services. We distinguish two kinds of agents: actors and objects.

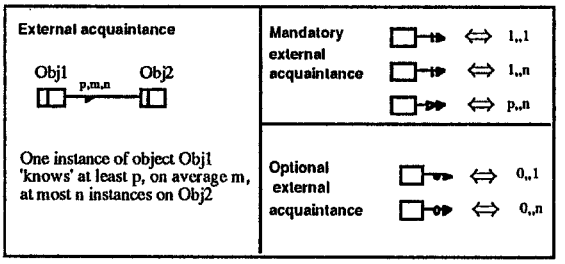
An actor is an intelligent agent because it can change its state without any external request from another agent.

An object is a non-intelligent agent with encapsulated acquaintances and services. An object is said to be non-intelligent because its type acquaintances and its behavior can not be modified without an external request.

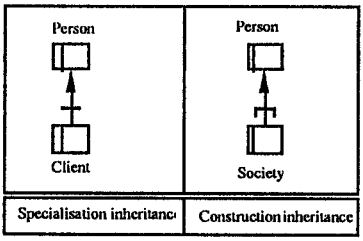
An abstract object is used to factorize the common acquaintances of some objects. No instances can be created for an abstract object.

An *acquaintance* of an object is one of its characteristics. Each object can use, modify and return its acquaintances. Two kinds of acquaintances are used in MCO: external and internal. External acquaintances of an object Obj1 represent all the objects Obji (i=1..n), to which it is linked. Internal acquaintances of an object are used to define its set of instances.

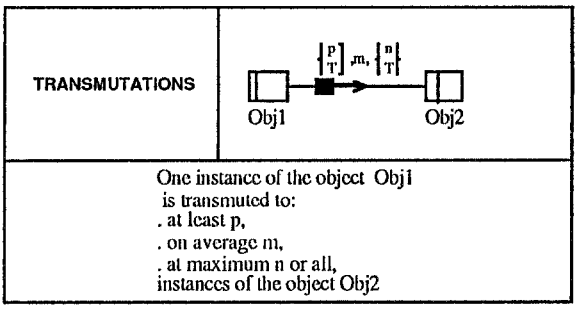
The graphical conventions used to represent external acquaintances are as follows:



In the MCO paradigm, we distinguish two kinds of inheritance: the classical 'is-a' inheritance link and the construction inheritance. The second kind doesn't have a semantic sense. It enables to reuse some characteristics of one object in an other without having the classical hierarchy of type/subtype. For instance :



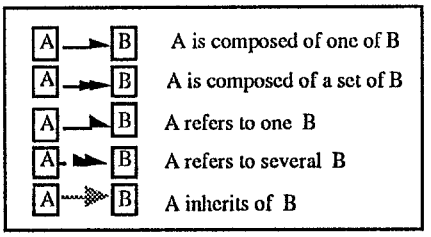
Client and society both inherit from the same class Person. The meaning of inheritance is not the same in the two cases. In the first one, the client *is a* person. In the second case, society inherits some of the person characteristics to define its own type, but it's not a person.
An object *transmutation* is defined by one or more departure objects, several arrival objects and the set of conditions necessary to its starting. Graphically a transmutation is represented as follows:



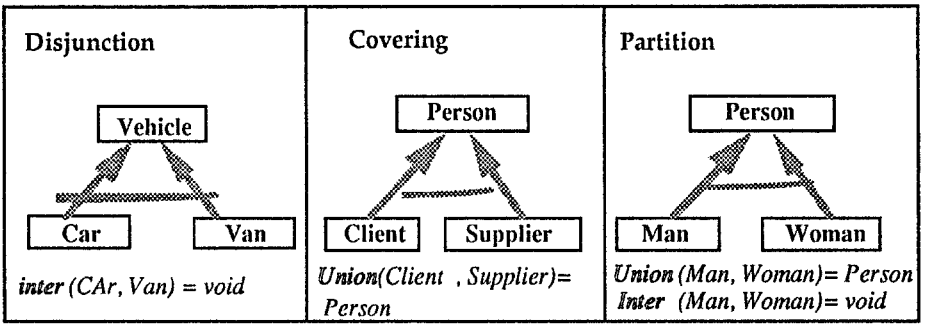
An object can request or ensure a *service* to another. This service consists, in general, in giving or demanding information about the state or the value of one characteristic of an object.
The object characteristics are: its Internal and external acquaintances, its services and its transmutations.
An illustration of the use of the main MCO concepts is given Fig 1.

2.2 Main O* Concepts

O* is an analysis method which recommends the use of the object-oriented paradigm as soon as the early development stages. It is supported by a conceptual model centered on the concept of object.
The objects perceived in the real world are classified into classes. A *object class* has an extension - the set of its instances -, and characterises the structure (static) and the behavior (dynamic) of the objects.
From a static point of view, an object is characterised by properties and references.
A property is valued either in a domain or in an object class. In the last case, a *composition link* joins two object classes, namely composed and component. The underlying semantics is that a component object is strongly dependent on, and belongs exclusively to its composed object. Composition reflects the fact that the composed object and its component(s) have life cycles of a similar duration.
A *referring link* expresses a transient link between objects of two object classes, called respectively referring and referred objects. The semantic is that a referred object can be shared by several referring objects. It also expresses an existential constraint on the referred object. The life cycle of a referring object is smaller than that of the corresponding referred object.
Composition and referring links are either simple or multiple.
An *actor class* allows to identify the agents of the external environment with which the information system communicates. This allows to define precisely the relation between the information system and the outside, to distinguish different competence zones and their interface to the information system.
The graphic representation allows a vision of several relationships between object classes and applications. From a static point of view (fig 1), each object class is associated with others by one, or more, composition links, referring links or inheritance links. Each object class is represented by a box. Links between object classes have to be read in the arrow direction:



Structural constraints specifying the invariant features of the objects can be specified upon properties and references.
The O* *inheritance* mechanism is defined between O* classes, named specialized and generalized O* classes. The first one inherits all the characteristics of the second, and has its particular ones [6].
The Inheritance constraint restricts the possibilities of existence of the objects of several specialized classes, for each object of a generalized class.
An inheritance constraint is specified within the generalized class. There are three types of inheritance constraints: disjunction, covering and partition :



From a dynamic point of view, an object is characterized by *operations* and *events*. An operation affects an object while an event ascertains a significant state change of the object, for which operations must be triggered. Other events, activated by actors of the organization or depending on a date (temporal), are not placed in an object class. They notice the arrival of messages from outside to the system. The event mechanism is similar to the one defined in the Remora method [17].
Static and dynamic graphs give a global view of the interrelations and interactions between objects. The object class description provides a local view of the objects by listing sets of properties, references, constraints, operations, events and inheritance links. The first three items characterize the local aspects of an object, the next two specify the static and dynamic relationships between objects.
The *state transition graphs* are used to express conditions on the sequence of an object operations. Transitions describe the change from a state to another (of an object) by the operation execution. The nodes of the graph are states and the arcs are state transitions [2].

A same example is described here after using the two models, O* and MCO (see Fig 1).
In this O* example, we note a covering constraint, each person must be a client or a supplier (or both). In the MCO example, this constraint is represented by an abstract class Person (non instanciable) whereas Client and Supplier are persistent classes. Lower level classes may be

created to represent clients who are suppliers at the same time. Discussion about creation of such classes can be found in [15].
To represent static links between objects, in O* one way arrow is used where as in MCO double way is requiered.

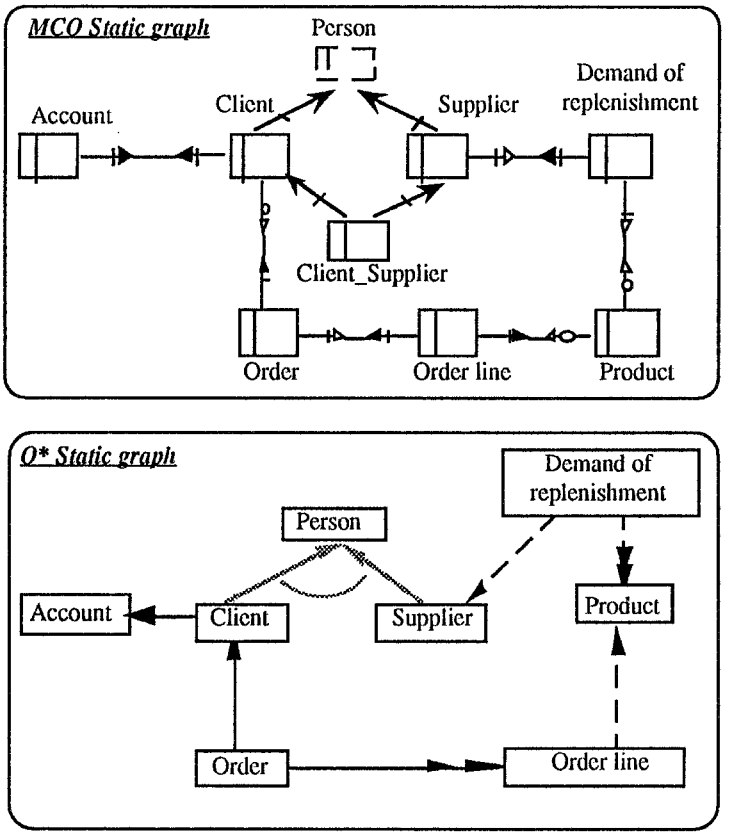


Fig 1: An MCO and O graphical descriptions of the static relationships between classes*

2.3 Equivalence Between MCO and O* Models

We propose a comparative table between the two models O* and MCO, described above. The comparison criteria we chose deal with the nature of the concepts used by the models mentioned before. These criteria are listed according to their types and will be used for the translation from the conceptual to the implementation level. A framework for evaluating and comparing current object-oriented analysis and design research was developed in [14].

Type	O* Model	MCO Model
A	Class	Class
B	Inheritance	Inheritance
C	Property Simple Multiple Enumereted Interval Aggregat	Internal acquaintance Atomic Multiple Enumereted Interval Composed
D	Constraint Attribute Uniqueness	Constraint Attribute Uniqueness
E	Composition link Simple Multiple Refering link Simple Multiple	Cardinality (1,,1), (1,,1) (1,,N), (1,,1) Cardinality (1,,1), (0,,N) (1,,N), (0,,N)
F	Inheritance Constraint Disjunction Couvering Partition	Inheritance type Simple Multiple Construction
G	Operation	Service
H	Event External Temporal Internal	Event model object Event model object Service
I	State transition graph	States and behavior graph of objects
J	Actor class	Actor agent
K	-	Transmutation

We can easily see that the object-oriented paradigm concepts (class, object, inheritance, encapsulation ...) are used in the two models. However, some differences exist, such: the specification of links between objects, the expression of inheritance constraints and dynamic aspect. For example, in MCO model the single acquaintance link includes the O* static links (composition and referring links). They are differenciated using MCO cardinalities. The O* event concept has not the same definition as in MCO model. But it can be simulated, when it is external or temporal, using the Event model object concept and using service when it's internal.

3 Object-Oriented Interface (O2I)

In order to automate the translation from object-oriented design towards object-oriented programming, we propose an interface called Object-Oriented Interface (O2I).
O2I is a generic interface which guides automatically the mapping from conceptual modelling to a logical specification. In order to be generic, the interface leans on an object-oriented pivot language. It's considered as a super-set of object-oriented implementation languages. The interface uses some mapping rules to:

a- transform the conceptual modelling to a pivot specification
b- transform the pivot specification to target environment

The interface contains a several collection of rules related to different mappings . For instance, from O* into ONTOS/C++ and Eiffel, from MCO into ONTOS/C++ and Eiffel.
An interface user has two alternative situations :

a- either the conceptual model and the target language take part of the interface and then the mapping will be automatically done

b- or one of them, or both, are unknown and then Rc and Ri rules (see 3.3 and 3.4) should be defined and integrated to the interface.

Defining an intermediate model such as a pivot model has the advantage of avoiding the study and definition of a new transformation technique between the conceptual model and target implementation models newly envisaged, by just adopting the pivot model while shifting the target one.

O2I is then an intermediate step downstream object-oriented design and upstream object-oriented programming.

3.1 General Overview

O2I aims at assuring a mapping, guided by a software tool, from a conceptual specification towards an implementation. The interface consists of an object-oriented pivot language (O2IL), Rc rules for the mapping from the conceptual specifications to O2IL and Ri rules from the O2IL to object-oriented implementation (language and persistence).

The figure below illustrates the O2I:

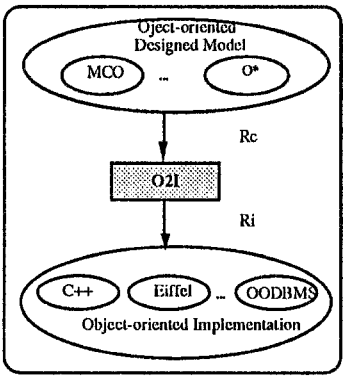


Fig 2: O2I : Object-oriented Interface

Two steps form the translation process. First, Rc rules are run to transform the user's conceptual scheme into O2I. Then Ri rules are applied to lead to an object-oriented programming environment. Rc and Ri rules are detailed further.

3.2 O2IL Syntax

The O2I language syntax is presented bellow. Notations used are inspired from Backus' ones:

[A] : A is optional
{A}+ : A exists at least once
A | B : A or B
A , B : A and B

Let:
G be a generalized class (abstract or persistent) and S_i a specialized class i (a subclass of G).
<G>: the set of G instances.
<S>$_i$: the set of S_i instances.

The pivot model is considered to be a super-set of existing implementation models. It is a class model which generic structure is defined as follows :

```
DEFINE CLASS <class-name>
[INHERITS FROM : {< superclass-name>]
      [REDEFINITION :
            {[<attribute-name>: <attribute-type>]+]
      [RENAME :
            [[<attribute-name>      WITH      <attribute-name>]+]|
            [[<method-name>         WITH      <method-name>]+]] |
            [[<attribute-name>      WITH      <attribute-name>]+ ,
                  [<method-name>    WITH      <method-name>]+]]+]
[INSTANCE VARIABLES :
   [<attribute-name> : <attribute-type> PUBLIC | PRIVATE | PROTECTED]]+

[CONSTRAINT :
      [UNIQUENESS :
         ([<attribute-name>]+)]
      [ATTRIBUTE :
         [<expression>]+]
      [CARDINALITY :
         [<attribute-name>: (CardMin,CardMax)]+]]]
[INHERITANCE CONSTRAINTS :
   [ ⋂ (i=1..n) <S>i = Ø]
   [ ⋂ (i=1..n) <S>i ≠ Ø ]
   [ ⋃ (i=1..n) <S>i = <G>]
   [ ⋃ (i=1..n) <S>i ≠ <G>] ]
INSTANCE METHODS :
      METHOD <method-name>: PUBLIC |PRIVATE|VIRTUAL
            [( VAR : [<input-parameter>]+)]
            [PRECONDITION : [<predicate>]+]
            BODY
                  ---Algorithm
                  [[CALL ( <object-name>  <operation-name>,[,<factor>)]]+]
            END Body
            [POSTCONDITION : [<predicate>]+]
      END METHOD
END CLASS
```

With :

<attribute-type> :: <basic_domain>| <collection_domain> | <aggregate_domain> | <enumerated_domain> | <referred-domain> | <domaine-intervale>
<basic_domain> :: integer | real | date |string | boolean ...
<collection_domain> :: SET OF (<class-name>)
<enumerated_domain> :: ENUMERATED ([value]+)
<interval_domain> :: [min..max]
<aggregate_domain> :: <aggregate-name-class>
<referred_domain> :: <refrered-name-class>
<simple_expression> :: <term> <comparaison_operator> <term> | <term>
<term> :: <attribute-name> | OLD.<attribute-name> | NEW.<attribute-name> <method-name> | <constant-name>
<composed_expression> :: <simple_expression> <logical_operator> <expression>
<expression> :: <simple_expression> | <composed_expression>
<predicate > :: <expression>
<logical_operator> :: OR | NOT | AND | IN
<comparaison_operator> :: = | > | < | ≥ | ≤ |≠

A pivot language class is composed of variables, constraints and methods. Those concepts constitute the intentional definition of its instances and precise their static and dynamic aspects: structure and behavior.
Instance variables are data containers, like variables in a procedural programming languages. They are defined by a set of attributes and constraints. Each attribute has its own type. Constraints are used to precise the class external identifier, invariant and the cardinalities expressing the structural links between classes.
A multiple inheritance means that a specialized class, or subclass, may have more than one generalized classes, or superclasses. The major problem to solve with multiple inheritance is how to handle name clashes between methods or variables of multiple superclass. In our model, we use the renaming principle for attributes and methods.
To express inheritance constraints, union and intersection operators are used. Those operators are applied to the specialized class instances. Thus, inheritance constraints give the type of the class: abstract or persistent. Four kinds of inheritance constraints are defined within our model. They cover all possibilities of existence of the objects of several specialized classes, for each object of the generalized class.
In some object-oriented languages, a method may be public, private or virtual [19]. We define the set of methods for a class as the union of the set of private, public and virtual methods. A public method is a part of the interface of the class. We define the class interface as a set of methods which can be used by other classes. A private method is local to a class and not accessible by other classes. A virtual method is known in the class where it is defined, but implemented in descendants only. We extend the definition of methods as follows. Each method is defined by its input parameters, preconditions and postconditions.
The event concept is defined by a method which includes how operations are managed and synchronized.

3.3 Rc Mapping Rules

To transform the user's object-oriented conceptual scheme, the following Rc rules must be used:

Rc 1: All concepts of type A are translated into O2I class. Each class will be public, private or virtual.

Rc 2: All concepts of type B are translated into INHERITS FROM with rename and/or redefinition possibilities.

Rc 3: All concepts of type C are translated into INSTANCE VARIABLES according to there types (predefined, SET OF, ENUMERATED, aggregate ...).

Rc 4: All concepts of type D are translated into CONSTRAINT of INSTANCE VARIABLES according to their nature (uniqueness or attribute).

Rc 5: All concepts of type E are translated with aggregated attributes and cardinality constraints in the CONSTRAINT part. In the case of a strongly dependency, cardinalities must be defined in two classes, the caller and called. However, in the case of a weak dependency, the cardinalites is expressed only in the caller objects.

Rc 6: All concepts of type F are translated into INHERITANCE CONSTRAINTS according to their types (disjunction, covering, partition,...). We define four types of inheritance constraint:

a) $\bigcap_{i=1}^{n} <S>i = \emptyset$

b) $\bigcap_{i=1}^{n} <S>i \neq \emptyset$

c) $\bigcup_{i=1}^{n} <S>i = <G>$

d) $\bigcup_{i=1}^{n} <S>_i \neq <G>$

The first six rules are used to map the static aspect of an information system: mapping of attributes, domains, structural links and inheritance. For instance the translation of Person and Client classes of graphical descriptions given in figure 1 is as follows :

```
DEFINE CLASS    PERSON
INSTANCE VARIABLES :
   Nss : string (15)
   Name : string(30)
   Age : [0 .. 132]
   Adress : ADRESS
CONSTRAINT :
   UNIQUENESS :
      Nss
   CARDINALITY :
      Adress : (1, 1)
INHERITANCE CONSTRAINTS :
 CLIENT » SUPPLIER = PERSON
....
END CLASS --- Person
```

```
DEFINE CLASS    CLIENT
INHERITS FROM : PERSON
   REDEFINITION Age : [ 0..80]
   RENAME
      Adress WITH Adress_Client
.....
END CLASS    -- Client
```

Fig 3: The mapping of the static aspect of the conceptual specification

The dynamic concepts (operation, event, state transition graph, service, actor ...) are mapped using the following rules :

Rc 7: All concepts of type G are translated into INSTANCE METHODS using a precondition and a postcondition possibilities. In this case of a redefined method, the procedure will call explicitly the procedure of the same name defined in the superclass.

Some models describe operations by a text in natural language. This text specifies the operation purpose and the rules according to which attributes and states are valued or changed. In this case, the designer has to give his algorithm details using, when needed, the classical instructions (IF... THEN...ELSE...ENDIF, WHILE.... END,...). If the model uses a formal specification language [16], the translation will be automatically done into O2I language. For instance, an order creation operation is translated into O2I as follows:

```
DEFINE CLASS ORDER
INSTANCE VARIABLES:
   Ord_Line : SET OF (ORDER_LINE)
   State    : ENUMERATED (created, delivered, invoiced, paid)
   .....
CONSTRAINT :
.....
    CARDINALITY :
       Ord_Line : (1, N)
       ....
INSTANCE METHODS :
  METHOD Create_Order : PRIVATE
      (VAR : number, ord_line, Creation_date, state)
      PRECONDITION : absent order
      BODY :
        {Create_Oder_Line ()}+
        --- Create one instance of order
        state := 'created'
      END Body
      POSTCONDITION : state := 'created'
    END METHOD
END CLASS --Order
```

Fig 4: An example of operations and services mapping

Rc 8.1: All concepts of type H are translated into INSTANCE METHODS. The event predicate is translated into the method precondition or as simple condition into the method body. Trigger is translated using the procedure CALL with specifying three parameters : the object name, the operation and the factor.

For instance the product event (out of stock) can be translated into O2I as follows:

```
DEFINE CLASS PRODUCT
 .......
 METHOD Out_of_Stock : PUBLIC
  BODY :
    IF  (OLD.qte_stock >=replenishment_level) and
        (NEW.qte_stock < replenishment_level)
    THEN
       CALL  (Supplier, Demand_of_replenishment)
    ENDIF
  END Body
 END METHOD
END CLASS --Product
```

Fig 5: An example of the mapping of an Out_Of_Stock event

Rc 8.2: If an event is internal, it is translated by a private method for the object.

Rc 8.3: If an event is external or temporal, it is translated by an abstract class and a method for its execution.

Rc 9: All concepts of type I are translated with enumerated attribute (called STATE) and with INSTANCE METHODS (precondition and postcondition are mandatories).

Rc 10: All concepts of type J are translated into an abstract class (non instanciable).

Rc 11: All concepts of type K are translated into INSTANCE METHODS which allow instance migration.

3.4 Ri Mapping Rules

The second set of rules is used for the translation from an O2I specification, already established before, towards a target object-oriented implementation.

Ri 1: Each OI2 class is translated into a class within the target language.

Ri 2: Each actor class (a class without instance variable in O2IL) is translated into an abstract class (deferred in Eiffel, virtual in ONTOS/C++)

Ri 3: The INHERITS FROM concept is translated into a classical inheritance into target languages. To resolve the multiple inheritance conflict, REDEFINE and RENAME can be used.

Ri 4.1: All object-oriented programming languages support the <basic_domain> notion.

Ri 4.2: Each <collection_domain> is translated using the generic class COLLECTION [X] into Eiffel and collection SET (X) into ONTOS/C++, where X is a type.

Ri 4.3: Each <aggregate_domain> or <referred _domain> is translated using a deferred class into Eiffel and abstract class into ONTOS/C++.

Ri 4.4: Each <enumerated_domain> or <interval _domain> is translated using the routine Eiffel concept or the ONTOS/C++ method.

Ri 5: Each uniqueness or attribute constraint is translated into an invariant or using a specific method.

Ri 6: Each cardinality constraint is translated within a specific method which verifies the minimal and maximal cardinalities (in the caller and the called).

For instance, the translation of the static aspect of the conceptual specification, giving in Fig. 3, towards Eiffel and ONTOS/C++ programming environments is as follows:

Eiffel :	*ONTOS/C++ :*

Class Person
export Nss, name,age, address
feature
Nss : STRING
name : STRING
age: INTEGER
address: expanded (ADDRESS)
set_age (new_age: INTEGER) : BOOLEAN is
do
--- verified if the the value of age is correct
if (new_value>=0 and new_value <=132)
then
result:=TRUE
age:=new_value
else
result := FALSE
end;
end ;
Set_Nss (new_value: STRING) : BOOLEEN is
do
c: COLLECTION[Person]
c.select X suchas
X.Nss :=new_value
inherit Person
if (not c.empty)
then
result:= FALSE
else
Nss:=new_value
result:=TRUE
end;
.....
end ; --- class Person
	Class Person : Public Object
	{
	Public
	char *Nss;
	char *name;
	int age, result;
	ADDRESS address
	int set_age (age)
	{
	if ((age < 0) and (age > 132))
	then
	result := 1;
	else
	result:=0;
	end;
	return (result)
	}
	int Set_Nss (new_value: STRING)
	{
	If (select * from Person X
	where X.Nss := new_value) > 0
	then
	result:=1;
	else
	{
	Nss:=new_value;
	result:=0;
	};
	return (result);
	}
	
	} ; --- class Person

where Object is a predefined ONTOS class. Each persistent object must be an instance of the *Object class* or its derived classes.

The inheritance constraints are mapped in object-oriented programming using the following set of rules:

Ri 7.1: Each inheritance constraint such that $\bigcap_{i=1}^{n} <S>_i = \emptyset$ or $\bigcup_{i=1}^{n} <S>_i \neq <G>$ is translated by a classical inheritance into the target languages (all classes are persistents).

Ri 7.2: Each inheritance constraint such that $\bigcap_{i=1}^{n} <S>_i \neq \emptyset$ is translated by a superclass and subclasses all persistents. $2^n - (n+1)$ persistent classes will be created. They represent all possible intersections between the n subclasses.

Ri 7.3: Each inheritance constraint such that $\bigcup_{i=1}^{n} <S>_i = <G>$ is translated using the Ri 7.2 rule where the superclass is abstract (deferred).

The O2I dynamic concepts are mapped using the following rules :

Ri 8: Each instance method is translated by a specific method into the target language. Pre and Post conditions are checked into the method body.

The meaning of the event concept is not the same in different conceptual models. This concept is particularly hard to implement in object oriented languages, because of the functional principles of the method calls [12]. For instance, in O*, it poses some problems such as:

- the implementation of the internal event mechanism
- the management of the dynamic transition during the execution
- the saving of the event succession.

To resolve these problems, we propose a solution based on two steps:

- when event is activated, its operations are triggered and its predicates - susceptible to be chained - are tested
- then, each event having a true predicate is activated in sequence.

To implement this mechanism, we use the two following rules :

Ri 9.1: Every event method is implemented by a routine in the target language.

Ri 9.2: For every private event method, a specific method (TEST_PRED) is implemented in order to test predicate. A boolean parameter is used when calling the method. When the call has a factor, we must keep the predicate value for each affected object.

```
Class Product
  feature
  Pred_Out_of_stock : BOOLEAN;
  sp : Supplier;

   Out_Of_Stock (Pred_Out_of_stock) : BOOLEAN is
    do
      if (OLD.qte_stock >=replenishment_level) and
        (NEW.qte_stock < replenishment_level)
      then
        Demand_Of_Replenishment(Pred_Out_of_stock);
      end;
  end;
.....
end - - - class Product

Class Supplier
 export Demand_Of_Replenishment ...;
  feature
    .......
   Demand_Of_Replenishment(Pred_Out_of_stock): BOOLEAN is
  do
      --- operation perforning text
  end;
end; - - - class Supplier
```

Fig 6: Eiffel implementation of an event method.

4 Conclusion

In this paper we proposed an interface, supported by a software tool, to bridge the gap between object-oriented conceptual modelling and object-oriented implementation (Eiffel, ONTOS/C++). It is based on a pivot model and a set of mapping rules. The interest of our approach is to obtain a precise and non ambiguous specification easily implementable.
Our works are, at present, oriented to suggest a 'PIVOT' object-oriented design model that might allow to shift from any object-oriented conceptual specification to different environments (OO languages, OODBMS, relational DBMS...) and integrate some optimization notions using metrics.

Acknowledgements

We thank Professor C. ROLLAND, Professor X. CASTELLANI, Professor N. BIDOIT and R. LALEAU for their helpful advise and remarks and their amiability.

References

[1] E. Ardonof; "OFM : une méthode formelle pour la conception des bases de données orinetées objet", doctorat d'université Paul Sabatier, September 1992.

[2] BUSINESS CLASS project, "Analyst Workbench Tutorial" Release 2, Deliverable BC.R.TS.T34, Business class ESPRIT II P 5311, 1992.

[3] G.Booch, "Object-Oriented Development", IEEE Trans. on S.E., Vol. SE-12, N°2, Feb. 1986.

[4] G.Booch, *Software Engineering with Ada*, 2nd Edition, Benjamin/Cummings Publishing Co., Menlo Park, 1987.

[5] J. Brunet, C.Cauvet, L. Lassoudris, "Why Using Event in a Hignt Level Specification", in Proc. of the Entity / Relationship Conference, Lausanne, 1990.

[6] J. Brunet, "Modeling the world with Semantic Objects", Working conference on the object oriented approach in information systems, 1991.

[7] X. Castellani, "Le modèle de la méthode MCO d'analyse et de conception des systèmes d'objets" Congrés INFORSID, Paris 1991.

[8] X. Castellani, "MCO Méthodologie Générale d'analyse et de Conception des Systèmes d'Objets, Tome 1 : l'Ingénieurie des besions" Edition Masson, Paris 1993.

[9] P Cointe, "Metaclasses are First Class : the ObjVlisp Model",LITP, Université Paris VI ACM, OOPSLA, 1987.

[10] G. Grosz " Formalisation des connaissances réutilisables pour la conception des systèmes d'information",Thèse de doctorat de l'université Paris VI, Décembre 1991.

[11] M. Heitz, "HOOD, une méthode de conception hiérarchisée orientée objet pour le développement des gros logiciels techniques et temps réel", Déc. 1989, Journées Ada-France.

[12] N.Kraïem, J.Brunet " Mapping of Conceptual Specifications into Object-oriented Programs", SEKE'92, Proceedings of the Fourth International Conference on Software Engineering and Knowledge Engineering, IEEE, Capri, June 1992.

[13] B. Meyer , 'Conception et programmtion orientée objets", InterEdition, 1990.

[14] D.E. Monarchi, G.I. Puhr " A research typology for object-oriented analysis and design", Communications of the ACM, September 1992, Vol 35, N° 9.

[15] R. Laleau, X. Castellani, M. Jouve. "Normalized Design of the Specialization Inheritance". Indo-French workshop on Object-Oriented Systems. GOA-Inde. 2-5 Nov. 1992.

[16] R. Laleau, X. Castellani, and al. "Transformation de Spécifications informelles MCO en Spécifications formelles VDM". Journée AFCET. November 1992.

[17] C. Rolland, O. Foucaut, G. Benci, *Conception de Systèmes d'Information, la méthode Remora*, Ed. Eyrolles, Paris 1988.

[18] E. Yourdon, P. Coad "Object oriented analysis", second Edition, Yourdon press, 1991.

[19] Peter Wegner. "Concepts and Paradigms of Object-Oriented Programming,"OOPSLA-89 Keynote Talk, 1989.