JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering **©JOT, 2005**

Vol. 4, No. 1, January-February 2005

Foundations for MDA-based Forward Engineering

Liliana Favre,
Universidad Nacional del Centro de la Provincia de Buenos Aires, CIC (Comisión de Investigaciones Científicas de la Provincia de Buenos Aires), Argentina

Abstract

Model Driven Architecture (MDA) is an emerging technology that is supposed to provide a technical framework for information integration and tools interoperation; many UML tools claim to be compliant with it. Model-to-model transformations are essential in MDA. This article describes foundations for UML-based transformation tools. We introduce the NEREUS language to cope with concepts of UML metamodel. A transformational system to translate OCL to NEREUS was defined. In this framework, we describe the NEREUS process to forward engineering UML static models to object-oriented code. Eiffel was the language of choice in which to show the feasibility of our approach. Transformations are supported by a library of reusable components and by a system of transformation rules that allow translating UML/OCL constructions to NEREUS specifications and Eiffel step-by-step.

1 INTRODUCTION

The standardization of UML regarded as notation leads to improvements in CASE (*Computer Aided Software Engineering*) tools, methods and standard modeling libraries. UML is used in many ways and different domains for expressing different types of concepts such as language independent software specification, high-level architecture, website structure, workflow specification and business modeling. It has been applied successfully to build systems for different types of applications running on any type and combination of hardware, operating system, programming language and network [OMG03].

In the marketplace there are numerous UML CASE tools that differ widely in functionality, usability, performance and platforms [Case03]. They are having a significant impact on the software development industry. However, the support that they provide has numerous gaps. Many tools can generate some code from a model, but that usually goes no further than the generation of some template code. Reasoning about

Cite this article as follows: Liliana Favre: "Foundations for MDA-based Forward Engineering", in *Journal of Object Technology*, vol. 4, no. 1, January-February 2005, pp. 129-153. http://www.jot.fm/issues/issue_2005_01/article4

models of systems is well supported by automated theorem prover and model checkers, however these tools are not integrated into UML-based environments. Also, these tools provide limited facilities for refactoring and reverse engineering.

The OMG is promoting the MDA that is supposed to provide a technical framework for information integration and tools inter-operation based on the separation of platform specific models (PSM) from platform independent models (PIM). Many tools claim to compliant with MDA. It is still evolving and some problems have been detected in the transformation processes that require flexible code generation mechanisms [Kleppe03].

Formal and semi-formal techniques can play complementary roles in MDA-based software development processes. We consider this integration beneficial for both semi-formal and formal specification techniques. On the one hand, semi-formal techniques have the ability to visualize language constructions allowing a great difference in the productivity of the specification process, especially when the graphical view is supported by means of good tools. On the other hand, formal specifications allow us to produce a precise and analyzable software specification and automate model-to-model transformations. The combination of UML and formal specifications offers the best of both worlds to software developer.

In this article we describe foundations for MDA-based forward engineering. Metamodeling is one key of the MDA. In this direction, we define the NEREUS language to cope with concepts of UML metamodel. In particular this language is relation-centric, that is it expresses different kinds of relations (dependency, association, aggregation, composition) as primitives to develop specifications. Much more information can be included in the specification metamodel using the combination of UML and OCL (Object Constraint Language) [Warmer03]. A transformational system to translate OCL to NEREUS was defined. NEREUS can be viewed as a communication bridge between UML and other algebraic languages and between UML and object oriented languages.

The UML/OCL is used to generate high-level specifications which are independent of any implementation technology. These specifications are tailored to specify realizations that fit a specific technology, which in turn are used to generate the code. Eiffel was the language of choice in which to show the feasibility of our approach. The process is based on the adaptation of reusable components that are defined in a framework that fits MDA. All of the proposed transformations can be automated; they can be integrated into iterative and incremental software development processes supported by the UML-based tools. Following this approach we can use the transformations of the forward engineering process and apply them backwad to reverse engineer code to a UML diagram.

The structure of the rest of this article is as follows. Section 2 discusses related work. Section 3 gives a brief description of the NEREUS language. Section 4 describes the NEREUS process to forward engineering UML models. Section 5 analyses a mapping from UML/OCL to NEREUS. Section 6 describes how to transform NEREUS specifications into Eiffel. Finally, Section 7 concludes and discusses further work.

2 RELATED WORK

Object orientation and formal languages

In the early 1980s, new specification languages or extensions of formal languages to support object-oriented concepts began to develop. Among them the different extensions of the Z language, for example Z++ [Lano91], OBJECT-Z [Smith00] or OOZE [Alencar91] can be mentioned. Another language with object-oriented characteristics is FOOPS [Rappanotti92].

Larch/Smalltalk was the first language with subtype and inheritance specification [Cheon94] . Larch/C++ is another language with similar characteristics [Leavens96].

CASL-LTL, an extension of CASL [Astesiano02], has been provided to deal with reactivity [Reggio99].

BON is an object-oriented method possessing graphical and textual languages for specifying classes, their relations and assertions, written in first-order predicate logic [Paige02].

Among the most recent languages, JML is a behavioral interface specification language for formally specifying the behavior and interfaces of Java classes and functions [Leavens02]. GSBL^{oo} is an extension of GSBL with constructions that allow the expression of different kinds of UML relations [Favre01].

Semi-formal and formal modeling techniques

Various works analyzed the integration of semiformal techniques and object-oriented designs with formal techniques. [Bordeau95] introduces a method to derive Larch specifications from class diagrams. [France97] describes the formalization of FUSION models in Z.

A lot of work has been carried out dealing with the semantics for UML models. The PreciseUML Group, pUML, was created in 1997 with the goal of giving precision to UML [Evans98]. It is difficult to compare the existing results and to see how to integrate them in order to define a standard semantics since they specify different UML subsets and they are based on different formalisms.

[Bruel98] describes how to formalize UML models using Z, and [Breu97] does a similar job using stream-oriented algebraic specifications. Additionally, [Gogolla97] does this by transforming UML to TROLL and [Overgaard98] achieves it by using operational semantics. U2B [Snook00] transforms UML models to B [Abrial96]. [Kim00, Kim02] formalize UML by using OBJECT-Z. [Reggio01] presents a general framework of the semantics of UML, where the different kinds of diagrams within a UML model are given individual semantics and then such semantics are composed to get the semantics on the overall model. [MCumber01] propose a general framework for formalizing UML diagrams in terms of different formal languages using a mapping from UML metamodels and formal languages metamodels.

Other works describe advanced metamodeling techniques that allow the enhancement of UML. [Gogolla02] analyzes the UML metamodel part dealing with stereotypes, and make various suggestions for improving the definition and use of stereotypes. [Barbier03] introduces a formal definition for the semantics of the Whole-Part relation that can be incorporated into version 2.0 of UML.

UML-based tools

In the marketplace, there are about 100 UML CASE tools that differ widely in functionality, usability, performance and platforms [Case03]. Current UML tools can help with the mechanics of drawing and exporting UML diagrams, eliminating syntactic errors and consistency errors between diagrams and supporting code generation and reverse engineering.

A number of tools claiming to support OCL have emerged. For example, the main task of USE tool [Ziemann03] is to validate and verify specifications consisting of UML/OCL class diagrams. Key [Ahrendt02] is a tool based on Together [Case03] enhanced with functionality for formal specification and deductive verification.

Our work describes foundations for MDA-based forward engineering. The following differences between our approach and some of the existing ones are worth mentioning. In the first place, NEREUS is more expressive than other algebraic languages and more suitable for representing certain aspects of the UML metamodel. As GSBL^{oo} is relation-centric: it expresses different kinds of relations as primitives to develop specifications [Favre01]. The characteristics that distinguishes NEREUS from GSBL^{oo} is its neutrality language. NEREUS can be viewed as an intermediate notation open to many algebraic languages such as CASL or Larch. In particular, we define the semantics of NEREUS in terms of the CASL language. On the other hand, a system of transformation rules to translate OCL to NEREUS is introduced.

We define a framework for reuse that fits MDA very closely. Component models are defined in three different levels of abstraction: Platform Independent Component Model (PICM), Platform Specific Component Model (PSCM) and Implementation Component Model (IMC). A transformational approach for the integration of UML/OCL with NEREUS is introduced. We propose to define PIM and PSMs by integrating UML/OCL and NEREUS specifications. Also, we define how to transform PIMs into PSMs. Transformations are supported by reusable components and by a system of transformation rules that allow translating NEREUS specifications to object-oriented code step-by-step.

3 FORMALIZATION OF THE UML STATIC VIEW

The concept of transformation of models is central to the realization of the benefits of MDA [OMG03]. To enable automatic transformation of a model, we need the UML metamodel that is written in a well-defined language.

The strongest point in UML metamodel is the modeling of class diagrams and well-formed rules in OCL. In this direction, we propose the NEREUS language to cope with the UML metamodel. NEREUS is suitable to build specifications in which the structural aspects are important. NEREUS is relation-centric, that is it expresses different kinds of relations (dependency, association, aggregation, composition) as primitives to develop specifications.

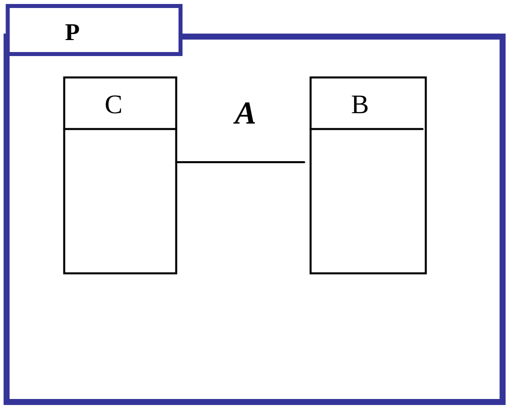
NEREUS is an intermediate notation open to many other formal languages. In particular, we define its semantics by giving a precise formal meaning to each of the construction of the NEREUS in terms of the CASL language, due to it is a unifier of proven algebraic languages [Astesiano02].

NEREUS allows us to develop PIM and PSMs that are full of information about systems to be implemented.

The NEREUS Language

NEREUS consists of several constructions to express classes, associations and packages. Fig 1 shows the relation between UML static models and NEREUS.

UML



NEREUS

```
PACKAGE P
   CLASS C
    …
    END-CLASS
   CLASS B
   …
   END-CLASS
  ASSOCIATION A
   …
  END-ASSOCIATION
  …
END-PACKAGE
```

Fig. 1: UML versus NEREUS

The syntax of a basic specification is shown in Fig. 2. NEREUS distinguishes variable parts in a specification by means of explicit parameterization. The elements of *<parameterList>* are pairs C1-> C2 where C1 is the formal generic parameter constrained by an existing class C2 (only subclasses of C2 will be a valid actual parameter).

CLASS className [<parameterList>] **IMPORTS** <importList> **INHERITS** <inheritsList> **ASSOCIATES** <associatesList> **DEFERRED** **FUNCTIONS** <functionList> **EFFECTIVE** **TYPE** <sortList> **FUNCTIONS** <functionList> **AXIOMS** <varList> <axiomList> **END-CLASS**	**ASSOCIATION** <relationName> **IS** <constructorTypeName> [...: Class1; ...:Class2; ...:Role1; ...:Role2; ...:mult1; ...:mult2; ...:visibility1; ...: visibility2] **CONSTRAINED BY** <constraintList> **END** **PACKAGE** <packageName> **IMPORTS** <importsList> **INHERITS** <inheritsList> <elements> **END-PACKAGE**

Fig. 2: NEREUS Syntax

The IMPORTS clause expresses dependency relations. The specification of the new class is based on the imported specifications declared in *<importList>* and their public operations may be used in the new specification.

NEREUS distinguishes subclassing from subsorting. Subsorting is like inheritance of behavior, while subclassing relies on the module viewpoint of classes. Subclassing is expressed in the INHERITS clause, the specification of the class is built from the union of the specifications of the classes appearing in the *<inheritsList>*. Subsortings are declared by the following syntax s1, s2, s3,...,s_n < s. Operations declared on some sort are automatically inherit by its subsorts.

NEREUS allows us to define local instances of a class in the IMPORTS and INHERITS clauses by the syntax *className [<bindingList>]* where the elements of *<bindingList>* can be pairs of sorts *s1: s2,* and/or pairs of operations *o1:o2* with *o2* and *s2* belonging to the own part of *ClassName*. References to parameterized specifications always instantiate the parameters. The sort of interest of a class (if any) is also implicitly renamed each time the class is substituted or renamed.

NEREUS distinguishes deferred and effective parts. The DEFERRED clause declares new sorts or operations that are incompletely defined. The EFFECTIVE clause either declares new sorts or operations that are completely defined, or completes the definition of some inherited sort or operation.

Operations are declared in FUNCTIONS clause. In NEREUS it is possible to specify any of the three levels of visibility for operations: public, protected and private. These are expressed by prefixing the symbols : + (public), # (protected), and - (private).

NEREUS supports higher-order operations (a function *f* is higher-order if functional sorts appear in a parameter sort or the result sort of *f*). In the context of OCL Collection formalization, second-order operations are required. It is possible to limit the scope of the declarations of auxiliary symbols by using local definitions.

NEREUS provides a taxonomy of constructor types that classifies binary associations according to kind (aggregation, composition, association, association class, qualified association), degree (unary, binary), navigability (unidirectional, bidirectional), connectivity (one-to one, one-to-many, many-to-many). New associations can be defined by the syntax shown in Fig. 2. 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.

The package is the mechanism provided by NEREUS for grouping classes and associations and controls its visibility (Fig. 2). <*importsList*> lists the imported packages; <*inheritList*> lists the inherited packages and <*elements*> are classes, associations and packages.

Several useful predefined types are offered in NEREUS, for example *Collection*, *Set*, *Sequence, Bag*, *Boolean*, *String*, *Nat* and enumerated types.

A detailed description may be found in [Favre03b]. In the next sections we give several examples that illustrate NEREUS specifications.

4 MDA-BASED FORWARD ENGINEERING OF UML STATIC MODELS

The MDA is a framework for software development that is driven by models in different abstraction levels. Model-to-model transformations that can be automated are crucial in MDA. The MDA process is divided into three main steps [Kleppe03]:

- Construct a model with a high level of abstraction that is called Platform Independent model (PIM).
- Transform the PIM into one or more Platform Specific Models (PSM), each one suited for different technologies.
- Transform the PSMs to code.

The PIM, PSMs and code describe a system in different levels of abstraction. The MDA also defines the relation between them.

We define a MDA-based forward engineering of UML static models. We propose to define PIMs and PSMs by integrating UML/OCL and NEREUS specifications. A PSM is tailored to specify UML static models in terms of realizations that are available in one specific technology. For example, an Eiffel PSM is a model in terms of Eiffel libraries. The construction of a PSM and code is based on a number of reusable components that can be manipulated in order to adapt them to new applications. The final step is the transformation of the PSM to code. Eiffel was the language of choice in which to experiment. Fig. 3 shows the main steps of the proposed process.

There is a need for reusable and adaptable transformation components. Reusable components that will be used in a process based on MDA have also to be described in different abstraction levels. We define a framework for reuse that fits MDA very closely.

Component models are defined in three different levels of abstraction: Platform Independent Component Model (PICM), Platform Specific Component Model (PSCM) and Implementation Component Model (IMC). The PICM level defines component model with a high-level of abstraction, which is independent of any implementation technology. A PICM is related to more than one PSCM, each suited for different technologies.

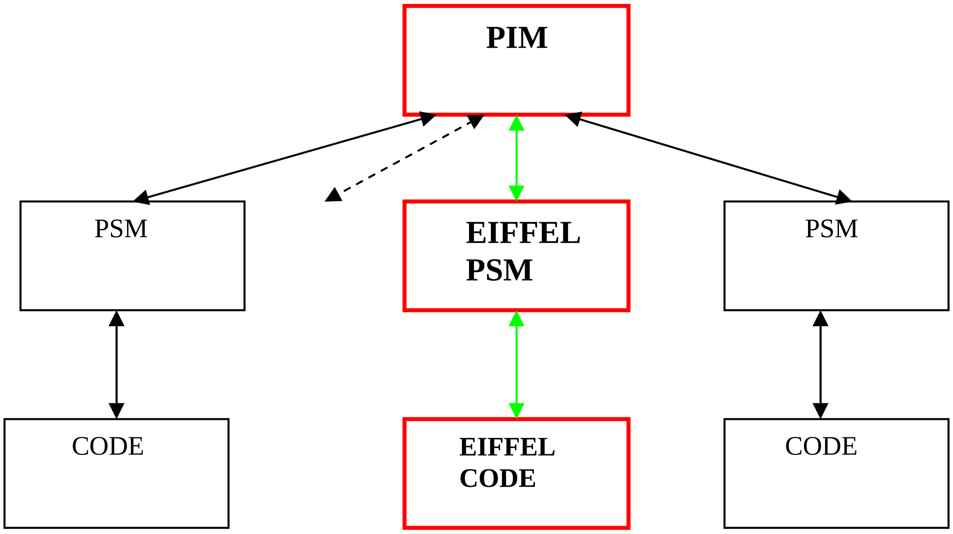


Fig. 3: The MDA-based process

The PSCM level defines a component model that is tailored to specify realizations of the PICM components which are code in a specific language.

We define specific reusable components for associations, OCL Collections and design patterns [Gamma95].

A component is defined in three levels of abstraction that integrate NEREUS incomplete algebraic specifications, complete algebraic specifications and Eiffel code. Fig. 4 depicts a specific Association component. It describes a taxonomy of associations classified according to kind, degree, navigability and multiplicity. The first level describes a hierarchy of incomplete specifications of associations using NEREUS and OCL. Every leaf in this level corresponds to sub-components at the second level. A realization sub-component is a tree of algebraic specifications: the root is the most abstract definition, the internal nodes correspond to different realizations of the root. For example, for a “binary, bi-directional and many-to-many” association, different realizations through hashing, sequences, or trees could be associated. These sub-components specify realizations starting from algebraic specifications of Eiffel libraries.

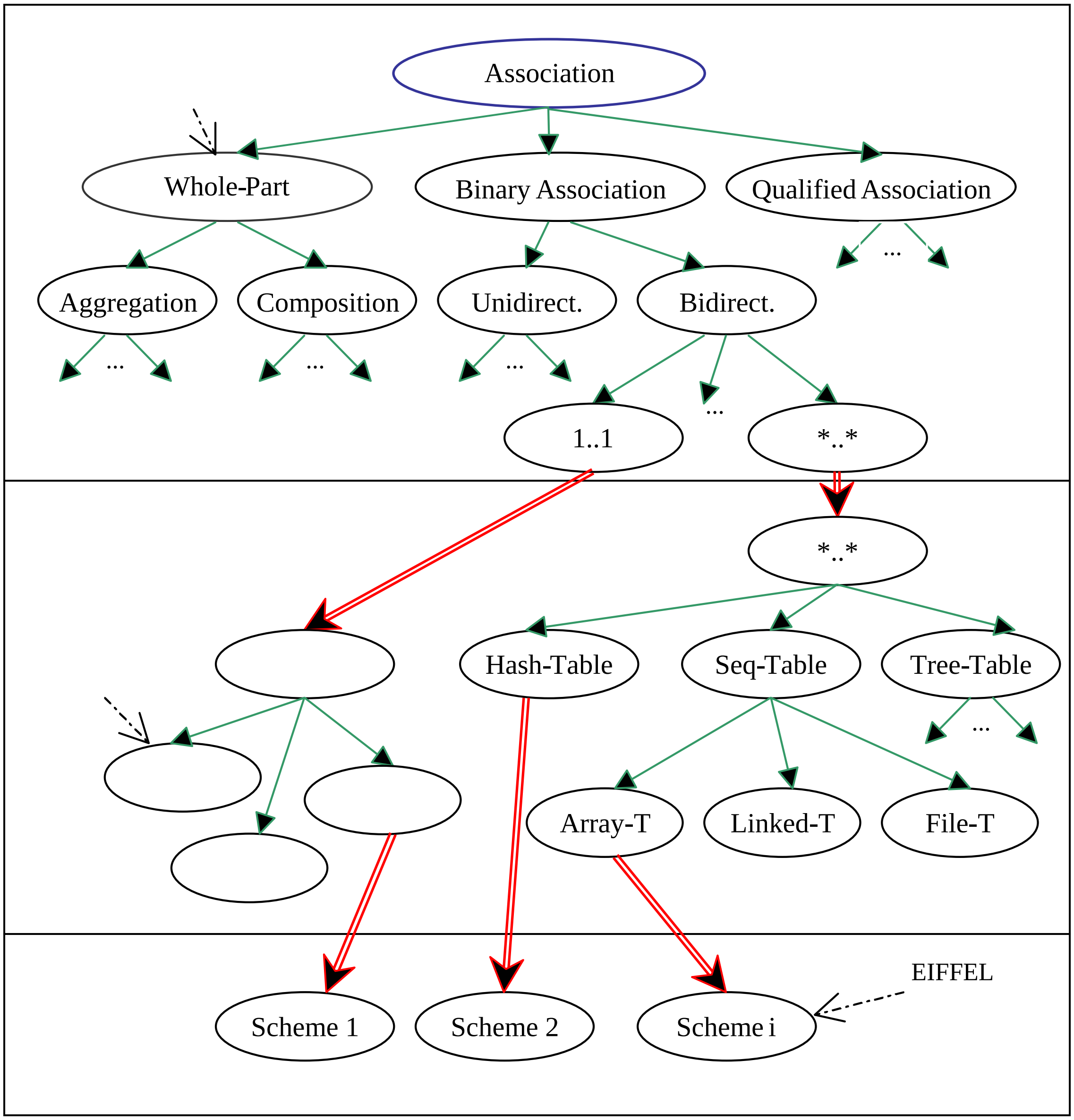


Figure 4. The component Association

The implementation level associates each leaf of the realization level with different implementations in Eiffel. 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 contrary, 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.

The component reuse is based on the application of reuse operators: *Rename*, *Hide*, *Extend* and *Combine*. These operators were defined on the three levels of components. A formal description of them and examples are included in [Favre98].

The central part of the MDA is to automate the generation of a target model from a source model. In the following sections we describe the vital transformations in this process: how a UML/OCL static model is transformed into a NEREUS specification, and how this specification is transformed into Eiffel.

5 FROM UML/OCL TO NEREUS

In this section we describe how to transform specifications consisting of UML class diagrams together with OCL invariants and pre- and postconditions into a NEREUS specification. The text of the NEREUS specification is completed gradually. First, the signature and axioms are obtaining by instantiating the reusable scheme BOX_ . Next, associations are transformed by instantiating reusable schemes that exist in the component *Association*. Finally, OCL specifications are transformed using a set of transformation rules. Then, a specification that reflects all the information of UML diagram is constructed. Fig. 5 shows the main steps of this phase.

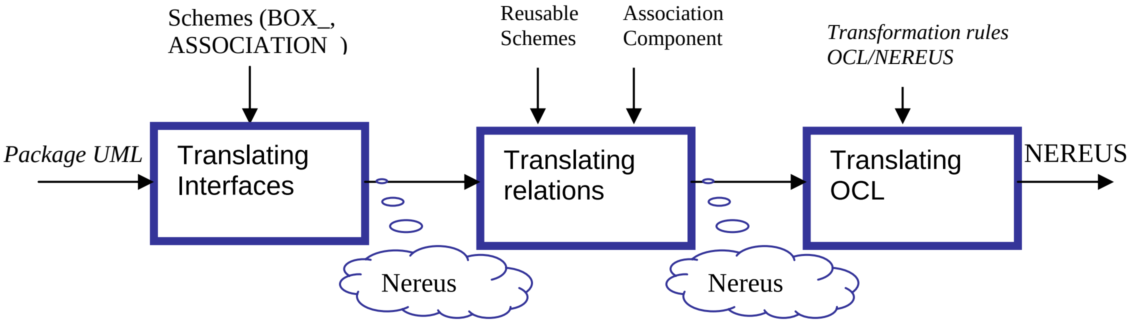


Fig. 5: From UML/OCL to NEREUS

Fig. 6 shows the BOX_ scheme. The attribute mapping requires two operations: an access operation and a modifier. The access operation takes no arguments and returns the object to which the receiver is mapped to. The modifier takes one argument and changes the mapping of the receiver to that argument. In NEREUS no standard convention exists, but frequently we use names such as *get_* and *set_* for them. Association specification is constructed by instantiating the scheme ASSOCIATION_ (Fig. 7).

CLASS *Name*
IMPORTS $TP_1,\ldots, TP_m, T\text{-}attr_1, T\text{-}attr_2,\ldots, Tattr_n$
INHERITS *B1,B2,..., Bm*
ASSOCIATES
$<<Aggregation\text{-}E_1>>,\ldots,<<Aggregation\text{-}E_m>>,$
$<< Composition\text{-}C_1>>,\ldots,<<CompositionC_k>>,$
$<< Association\text{-}D_1>>,\ldots,<<Association\text{-}D_k>>$
EFFECTIVE
TYPE *Name*
FUNCTIONS
createName : $T\text{-}attr_1$ x ... x $T\text{-}attr_n$ -> *Name*
set_i : *Name* x $T\text{-}attr_i$ -> *Name*
get_i: *Name* -> $T\text{-}attr_i$ 1<=i<=n

DEFERRED
FUNCTIONS
meth_1: *Name* x TPi_1 x TPi_2 x TPi_n -> TPi_j
...
meth_r : *Name* x TPr_1 x TPr_2 ... x TPi_n -> TPi_j
AXIOMS
$\{ t_1,t_1'$: $T\text{-}attr_1$; t_2,t_2':$T\text{-}attr_2$;...; t_n,t_n':$T\text{-}attr_n\}$
$\text{get}_i(\text{create}(t_1,t_2,\ldots,t_n)) = t_i$ $1 \le i \le n$

set_i (create $(t_1,t_2,\ldots,t_n)$, $t_{i'}$) = create $(t_1,t_2,\ldots t_{i'},\ldots,t_n)$

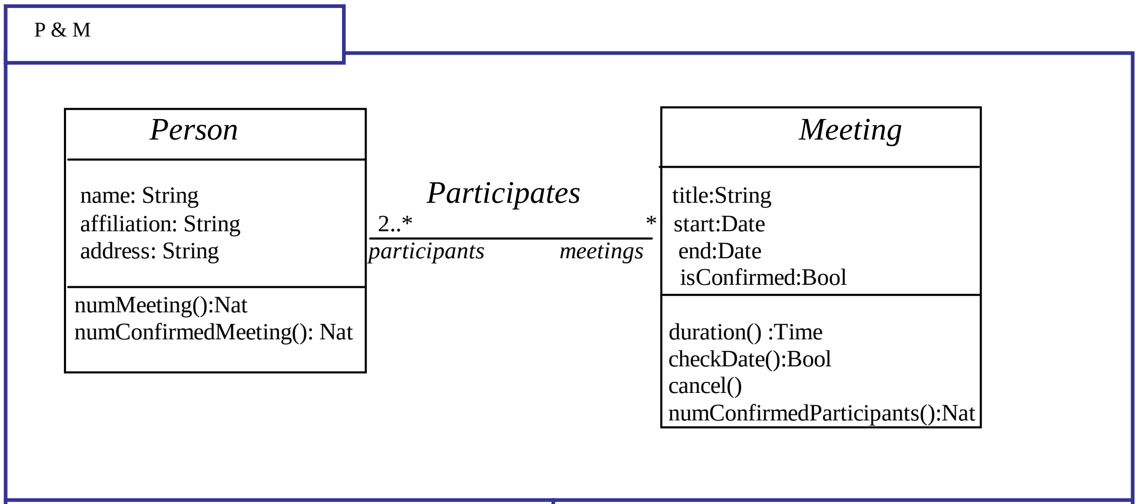
END-CLASS

Fig. 6: The BOX_ Scheme

ASSOCIATION ___
IS __ [__: Class1; __:Class2; __: Role1;__:Role2;
__:Mult1; __:Mult2; __:Visibility1; __:Visibility2]
CONSTRAINED BY __
END

Fig. 7: The ASSOCIATION_ Scheme

Fig. 8 shows a simple class diagram P&M in UML and NEREUS. P&M introduces two classes (*Person* and *Meeting*) and a bidirectional association between them. We have meetings in which persons may participate. The NEREUS specification is built by instantiating the scheme BOX_ and the scheme ASSOCIATION_ .



PACKAGE P&M
CLASS Person
IMPORTS String, Nat
ASSOCIATES <<Participates>>
EFFECTIVE
TYPE Person
GENERATED-BY Create_Person
FUNCTIONS
createPerson: String x String x String -> Person
name: Person -> String
affiliation: Person -> String
address: Person -> String
set-name: Person x String -> Person
set-affiliation : Person x String -> Person
set-address: Person x String -> Person
AXIOMS {p:Person; m: Meeting; s, s1, s2, s3: String; pa: Participates}
name(createPerson(s1,s2, s3)) = s1
affiliation (createPerson (s1, s2, s3)) = s2
address (createPerson (s1, s2, s3)) = s3
set-name (createPerson (s1, s2, s3), s) = createPerson (s,s2,s3))
set-affiliation (createPerson(s1,s2, s3), s) = createPerson (s1, s, s3))
…
END-CLASS
CLASS Meeting
IMPORTS String, Date, Boolean, Time
ASSOCIATES <<Participates>>
EFFECTIVE
TYPE Meeting
GENERATED-BY createMeeting
FUNCTIONS
createMeeting:
String x Date x Date x Boolean -> Meeting
tittle: Meeting -> String
start : Meeting -> Date
end : Meeting -> Date
isConfirmed : Meeting -> Boolean
set-tittle: Meeting x String -> Meeting
set-start : Meeting x Date -> Meeting
set-end: Meeting x Date -> Meeting
set-isConfirmed: Meeting x Boolean -> Boolean
AXIOMS {s: String; d, d1,: Date; b:Boolean;…}
title(createMeeting (s, d, d1, b)) = s
start (createMeeting (s, d, d1, b)) = d
end (createMeeting (s, d, d1, b)) = d1
isConfirmed (createMeeting (s, d, d1, b)) = b
...
END-CLASS
ASSOCIATION Participates
IS Bidirectional-Set [Person: Class1; Meeting: Class2; participates: Role1; meetings: Role2; *: Mult1; * : Mult2; + : Visibility1; +: Visibility2]
END
END_PACKAGE

Fig. 8:Translating interfaces and relations

Fig. 9 shows an instantiation of Bidirectional-Set scheme:

```
RELATION CLASS Bidirectional-Set
-- Bidirectional /* to */ as Set
INHERITS BinaryAssociation  [Class1 ->Person, Class2->Meeting]
IMPORTS Set_Person: Set [Person], Set_Meeting: Set[Meeting]
EFFECTIVE
OPS  name,  frozen  , changeable  ,  addOnly  , getRole1, getRole2, getMult1,getMult2,
getVisibility1, getVisibility2, isRelated, isEmpty, rightCardinality, leftCardinality
create: Typename->Participates
addLink:Participates(b) x Person(p) x Meeting(m)-> Participates
   pre:  not isRelated(a,p,m)
isRightLinked: Participates x Person -> Boolean
isLeftLinked: Participates x Meeting -> Boolean
getMeetings: Participates(a) x Person(p) -> Set_Meeting
   pre: isRightLinked(a,p)
getParticipants: Participates(a) x Meeting(m)->  Set_Person
   pre: isLeftLinked(a,m)
remove: Participates (a)  x Person (p)  x Meeting (m) ->  Participates
   pre: isRelated(a,p,m)
∀a:Participates; p,p1: Person; m,m1:Meeting; t:TypeName
name(create(t))= t
name(add(a,p,m)) = name(a)
isEmpty (create(t))= True
isEmpty(addLink(a,p,m))= False
frozen  (a) = False     changeable  (a)= True  addOnly  (a) = False
getRole1(a) = " participants"  getRole2 (a)  = "meetings"
getMult1(a) = *                    getMult2(a) = *
getVisibility1(a) = +            getVisibility2(a) = +
isRelated (create(t),p,m) = False
isRelated(addLink(a,p,m),p1,m1) = (p=p1 and m=m1) or  isRelated (a,p1,m1)
isRightLinked (create(t),p) = False
isRightLinked (addLink (a,p,m),p1)= if p=p1 then True  else isRightLinked(a,p1)
isLeftLinked(create(t),m)= False
isLeftLinked(addLink(a,p,m),m1)= if m=m1 then True else isLeftLinked(a,m1)
rightCardinality(create(t),p)= 0
rightCardinality(addLink(a,p,m),p1) =
 if p=p1 then 1 + rightCardinality(a,p1) else rightCardinality(a,p1)
leftCardinality(create(t),m) = 0
leftCardinality(addLink(a,p,m),m1)=
if m=m1 then 1+ leftCardinality(a,m1)  else leftCardinality(a,m1)
getMeetings(addLink(a,p,m),p1)=
if p=p1 then  including (getMeetings(a,p1), m) else getMeetings(a,p1)
getParticipants (addLink (a,p,m),m1) =
 if m=m1 then including (getParticipants(a,m1) , m) else getParticipants(a,m1)
remove(addLink(a,p,m),p1,m1) =   if (p=p1 and m=m1) then a else remove(a,p1,m1)
END-RELATION
```

Fig. 9: The association *Participates*

From OCL to NEREUS

The Object Constraint Language (OCL) is a query and expression language for UML. Recently, a new version of OCL, version 2.0, has been defined [OMG03].

Analyzing OCL specifications we can derive axioms that will be included in the NEREUS specifications. Preconditions written in OCL are used to generate preconditions in NEREUS. Postconditions and invariants allow us to generate axioms in NEREUS.

An operation can be specified in OCL by means of preconditions and postconditions by the following syntax:

Typename :: OperationName (*parameter1*:Type1,...): *ReturnType*
pre:_ some expression of *self* and *parameter1*
post: *Result* = _ some function of *self* and *parameter1*

self can be used in the expression to refer to the object on which an operation was called, and the name *Result* is the name of the returned object, if there is any. The names of the parameter *(Parameter1,...)* can also be used in the expression.

The value of a property in a postcondition is the value upon completion of the operation. To refer to the value of a property at the start of the operation, the property name has to postfix with "@" followed by the keyword "pre". Fig. 10 shows the OCL specifications linked to the package P&M (Fig. 8).This example was analyzed in [Hussmann99] and [Padawitz00].

context Meeting:: checkDate():Bool **post:** result = self.participants->collect(meetings) ->forAll(m | m<> self and m.isConfirmed implies (after(self.end,m.start) or after(m.end,self.start)))
context Meeting::isConfirmed () **post**: result= self.checkdate() and self.numConfirmedParticipants > 2
context Meeting :: duration ( ) : Time **post**: result = timeDifference (self.end, self.start)
context Person:: numMeeting ( ): Nat **post**: result = self.meetings -> size
context Person :: numConfirmedMeeting ( ) : Nat **post:** result= self.meetings -> select (isConfirmed) -> size

Fig. 10: OCL Specifications of P&M

The transformation process of OCL specifications to NEREUS is supported by a system of transformation rules. Fig. 11 shows how to map some OCL expressions onto NEREUS.

	OCL	**NEREUS**
M-1	v (variable)	v (variable)
M-2	Type -> OperationName(parameter1:Type1,...): Rtype	OperationName:TypexType1x...-> Rtype
M-3	v. operation(v’)	Operation (v,v’)
M-4	v->operation (v’)	Operation(v,v’)
M-5	v.attribute	attribute (v)
M-6	context A object.rolename	get_ (A, object)
M-7	e.op	op (Translate $_{NEREUS}$ (e)) *Let Translate$_{NEREUS}$ be functions that translate logical expressions of OCL into first-order formalae in NEREUS.*
M-8	collection-> op (v:Elem/ |boolean-expr-with-v) op ::=select| forAll| reject| exists	**LOCAL** **OPS** f: Elem -> Boolean **∀ v : Elem** f (v)= Translate $_{NEREUS}$ (boolean-expr-with-v) **WITHIN** op (collection, f) **END-LOCAL** ----------------------------------- op$_v$ (collection, [f(v)]) *Equivalent concise notation*
M-9	collection-> collect (v:Elem | expr-with-v) expr-with-v : S	**LOCAL** **OPS** f: Elem -> S **∀ v : Elem** f (v)= Translate $_{NEREUS}$ (expr-with-v) **WITHIN** collect (collection, f) **END-LOCAL** collect$_v$ (collection, [f(v)])
M-10	c->iterate (v:Elem; acc:Type = exp | expr-with-v-and-acc)	**LOCAL** **OPS** f: Elem x Type -> Type base: -> Type **∀ v : Elem; acc: Type** f(v,acc)=Translate$_{NEREUS}$(expr-with-v-and-acc) base = Translate $_{NEREUS}$ (exp) **WITHIN** iterate (collection, f, base) **END-LOCAL**

Fig. 11: Mapping basic expressions OCL

The following rules (Fig. 12) are used to generate the axioms for the *Person* and *Meeting* classes (Fig. 13).

OCL	NEREUS
Rule 1 T → Op (<parameterList>) : ReturnType **post:** expr	**OPS** *Translate*$_{\text{NEREUS}}$ (T → Op (<parameterList>) : ReturnType) ∀ t : T, ... *Translate*$_{\text{NEREUS}}$ (exp)
Rule 2 T-> forAll|exists|select |reject (v :Type| boolean-expr-with-v)	forAll$_v$|exists$_v$|select$_v$|reject$_v$ (*Translate*$_{\text{NEREUS}}$ (T), *Translate*$_{\text{NEREUS}}$ (boolean-expr-with-v)
Rule 3 T -> collect (v :type|v.property)	collect$_v$ (*Translate* $_{\text{NEREUS}}$ (T), *Translate* $_{\text{NEREUS}}$ (v.property))
Rule 4 T ->iterate(e:Elem; acc:Type = expr | Boolean-expr-with-e)	iterate$_v$ (*Translate*$_{\text{NEREUS}}$ (T), *Translate*$_{\text{NEREUS}}$(boolean-expr-with-e), *Translate*$_{\text{NEREUS}}$(expr))

Fig. 12: Transformation Rules

CLASS Person
...
∀p:Person; s,s': String; Pa: Participates
numConfirmedMeetings (p) =
size(select$_m$ (getMeetings(Pa,p), [isConfirmed (m)]) **Rule 1, 2**
numMeetings (p) = size (getMeetings (Pa, p)) **Reglas 1**
END-CLASS

CLASS Meeting

∀m,m1:Meeting; s,s':String; d,d',d1,d1':Date; b,b':Boolean; Pa:Participates

duration (m) = timeDifference (end(m),start(m)) **Rule 1**
isConfirmed (cancel(m)) = False
isConfirmed (m)=checkDate(m) and NumConfirmedParticipants (m) > 2 **Rule 1**
checkDate(m) = **Rules 1, 2, 3**
forAll$_{me}$ (collect$_p$ (getParticipants(Pa,m), [getMeetings (Pa, p)]), [consistent (m,m$_{e)}$])
consistent(m,m1)= not (isConfirmed(m1)) or (end(m) < start(m1) or end(m1) < start(m))
NumConfirmedParticipants (m) = size (getParticipants(Pa,m))
END-CLASS

Fig. 13: Translating OCL to NEREUS

6 FROM NEREUS TO EIFFEL

This section discusses the main steps for transforming NEREUS constructions into Eiffel (Fig. 14). The Eiffel code is constructed gradually. First, associations and operation signature are translated. The transformation is supported by reusable components. From OCL and NEREUS specifications it is possible to construct contracts on Eiffel and /or feature implementations by applying heuristics.

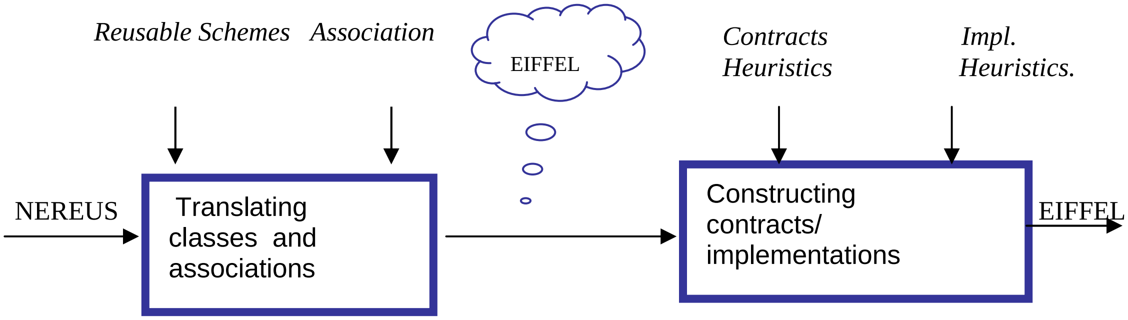


Fig. 14: The NEREUS/Eiffel phase

Mapping Classes and Associations

For generating code from some NEREUS specification we need transformation rules. For each class in NEREUS an Eiffel class is built.

If a NEREUS 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*. NEREUS and Eiffel have the same syntax for declaring class parameters. Then, this transformation is reduced to a trivial translation.

The relation introduced in NEREUS using the clause IMPORTS will be translated into a client relation in Eiffel. The relation expressed through the keyword INHERITS in NEREUS will become an inheritance relation in Eiffel. This provides the mechanism to carry out modifications on the inherited classes that will allow adaptation. Also, subsortings will become inheritance relations.

Associations are transformed by instantiating schemes that exist in the reusable component Association. 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.

The scheme shown in Fig. 15 may be used to implement the *Participates* association (Fig. 8).

```
class Class1
...
feature {NONE}
-- data members for association Association_Name
rol2: UnboundedCollectionbyReference [Class2];
mult_rol1: MULTIPLICITY;
--  operations for association Association_Name
get_mult_rol2 : MULTIPLICITY is
              do
                     Result:= mult_rol2
              end;
get_frozen_rol2 : BOOLEAN is
              do
                   Result:= result_frozen1
              end;
add_only_rol2 : BOOLEAN is
              do
                     Result :=  result_add_only1
              end;
changeable_rol2 : BOOLEAN is
              do
                     Result:= result_changeable1
              end;
cardinality_rol2 : INTEGER is
              do
                     Result:= rol2.count
              end;
set_ rol2 (
d:UnboundedCollectionbyReference [Class2]) is
require
get_mult_rol2.get_upper_bound  >=  d.count
              do
                     rol2 := d
              end;
get_ rol2 :
UnboundedCollectionbyReference[Class2] is
              do
                     Result := rol2
              end;
remove_rol2 (e: Class2) is
require
is-related_rol2 (e) and not get_frozen_rol2 and
not add-only_rol2
              do
                     rol2. prune (e)
              end;
add_rol2 (e: Class2) is
require is-related_rol2 (e) and not get_frozen_rol2
cardinality_rol2get_mult_rol2.get_upper_bound
```

```
           do
                  rol2. put (e)
           end;
add_rol2 (e:Class2) is
require
is-related_rol2 (e) and
multiplicity_rol2get_mult_rol2.get_upper_bound and
not get_frozen_rol2
            do
                  rol2. put (e)
           end;
 is_related_rol2 (e: Class2): BOOLEAN is
           do
                  Result:=rol2. has (e)
           end;
invariant
mult_ rol2.get_lower_bound = LowerBound;
mult_ rol2.get_upper_bound = Upper Bound;
rol2.count >= LowerBound;
rol2.count <= Upper Bound
end – class Class1
-----------------------------------------------------------------------
class Class2
...
feature {NONE}
-- data members for association Association_Name
rol1: UnbondedCollectionby Reference [Class1];
mult_rol1: MULTIPLICITY;
--  operations for association Association_Name
...
add_rol1( e: Class1) is
require
is-related_rol1 (e) and and not get_frozen_rol1 and
 multiplicity_rol1get_mult_rol1.get_upper_bound
                do
                  rol1. put (e)
                end;
 is_related_rol1 (e: Class2): BOOLEAN is
                 do
                  Result:=rol1. has (e)
                 end;
invariant
mult_ rol1.get_lower_bound = LowerBound;
mult_ rol1.get_upper_bound = Upper Bound;
rol1.count >= LowerBound;
rol1.count <= Upper Bound
end – class Class2
```

Fig. 15: The Bidirectional_ Set*..* Scheme

New code can be added by a textual substitution in the form

*[Class1: Person; Class2: Meeting; rol1: participants; rol2: meetings; UnboundedCollectionbyReference: UnboundedSetbyReference; result_frozen1: false; result_add_only1: false,......, LowerBound1:2; UpperBound: *; ..]*

For the association *Participates* the following will be in the code:

- For each class there is a private attribute in the opposite class
- The type of the newly created attribute is a Set and it will have corresponding get_ and set_ operations.

Next, from the operation signatures, the interfaces for the features of the Eiffel class are generated. 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 Eiffel and it should be eliminated from the list of parameters of the resultant feature. Second order functionalities of collections are translated respecting the syntax of the Eiffel schemes for Collection classes.

Constructing Eiffel contracts and implementations

Eiffel provides an assertion language. Assertions are Boolean expressions of semantic properties of the classes. They can play the following roles:

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

The expression of the form *old exp* denotes the value that an attribute or expression *exp* had on routine entry. *Current* refers to the target object itself and *Result* is the name of the returned object, if there is any.

Let $\mathit{Translate}_{\mathrm{Eiffel}}$ be a function that expresses the translation of a NEREUS term to Eiffel. $\mathit{Translate}_{\mathrm{Eiffel}}$ *op(es,e2,e3,...)* (where e_s, e_2, e_3 ... are well-formed non-ground terms and e_s is a term of the sort of interest) can be given in the following inductive way:

$$\boxed{\begin{array}{l}\mathrm{Translate}_{\mathrm{Eiffel}}\ \mathrm{op}(e_s, e_2, e_3\ ...) = \\ \mathrm{Translate}_{\mathrm{Eiffel}}\ e_s.\mathrm{op}\ (\mathrm{Translate}_{\mathrm{Eiffel}}e_2, \mathrm{Translate}_{\mathrm{Eiffel}}\ e_3....)\end{array}}$$

Preconditions and axioms of a function written in NEREUS are used to generate preconditions and postconditions for routines and invariants for Eiffel classes.

A NEREUS precondition, which is a well-formed term defined over functions and constants of the global environment classes, is automatically translated to Eiffel precondition. Axioms are translated to Eiffel post-conditions, invariants and implementations. We define two heuristics to obtain postconditions and /or implementations in Eiffel:

Invariant heuristics: It is possible to derive an invariant if it can establish a correspondence between the functions in an axiom A and the class attributes that only depend on the state of the object (that is to say, all the terms of the interest sort are variables). Then, Translate$_{\text{Eiffel}}$ (A) is the Eiffel invariant.

NEREUS	EIFFEL
CLASS Bounded-Sequence [Elem] ... ∀ **s: Bounded-Sequence; e: Elem** full (s) = (capacity (s) = count (s)) empty (s) = (count(s) =0)	**class** BOUNDED-Sequence [G] ... capacity:INTEGER count: INTEGER full: BOOLEAN empty: BOOLEAN **invariant** full = (count = capacity) empty = (count = 0)
CLASS Set [Elem] .... ∀ **s: Set; e: Elem** Current has (s,e) **implies** count(extend (s, e)) =count (s) old **not** has (s,e) **implies** count(extend (s, e)) = count (s) + 1	**class** SET [G] ... extend (e : G) .... **ensure** **old** has (e) **implies** count = **old** count **not old** has (e) **implies** (count = **old** count + 1)
CLASS Set [Elem] .... ∀**s: Set; e: Elem** has (s, e) => not empty (s) Result	**class** SET[G] .... has (e : G) :BOOLEAN ... **ensure** Result **implies not** empty

CLASS Meeting
...
∀ **p: Person**
numMeetings (p)= size(getParticipates (p))
	class Meeting
	...
	numMeeting (p:PERSON)
	do
	Result := meetings.size()
	end

Fig. 16: From axioms to contracts/implementations in Eiffel

Postcondition / implementation heuristics: A postcondition can automatically be generated from one axiom if a term *e(<list-of-arguments>)* which is associated to an operation *op*, 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 feature linked to 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. If the selected term *e* is linked with a value belonging to the sort of interest, it is associated with *Current* and the sort then it is associated to *old*. If the selected term *e* is linked with a value the different sort, it is associated with *Result.* If the resulting expression is in the form *Result* =… it is possible to generate the body of the feature.The programmer can also incorporate assertions that reflect purely implementation aspects. Fig 16 shows examples of transformations.

For simple operations the body of the feature could be generated from OCL post-conditions but frequently the body of the feature must be written. In that case, generating code for the pre and post-conditions ensures that the code conforms to the specification in the UML diagrams.

In this article we show a small example of applying the NEREUS process. In more complex and realistic examples the code might be generated starting from components of design patterns[Gamma95].

7 CONCLUSIONS

In this article we describe foundations for MDA-based forward engineering. We define the NEREUS language to cope with concepts of UML metamodel and a system of transformation rules to translate OCL to NEREUS. Then, the UML metamodel can be formalized in NEREUS. We define the semantics of NEREUS by giving a precise formal meaning to each of the constructions of the NEREUS specification in terms of the CASL language. However, NEREUS is an intermediate notation open to many other formal languages. A rigorous semantics clarifies the intended meaning of the UML/OCL metamodel, ensures that no corner cases are left out, and provides a reference for implementation.

UML/OCL class diagrams are used to generate NEREUS specifications, which in turn are used to generate the code. NEREUS allows us to keep a trace of structure of UML models in the specification structure that will make easier to maintain consistency between the various levels when the system evolves. The process is based on the adaptation of reusable components that are defined in a framework that fits MDA.

All the UML model information (classes, associations and OCL specifications) are overturned in specifications having implementation implications. In particular, we show how to translate different kinds of UML associations to Eiffel. Also, we describe how to construct assertions and code from algebraic specifications.

The proposed transformations preserve the integrity between specifications and code. Modifications at specification levels must be applied again to produce a new implementation. Most of the transformations can be undone, which provides great flexibility in code generation process supported by the existing UML CASE tools.

The transformational approach has the advantage that it allows the automatic recording of the design decisions made during the code generation from the UML diagrams. Following this approach we can use the transformations and apply them backward to reverse engineer code to a UML diagram.

The transformation of algebraic specifications to Eiffel code was prototyped [Favre98]. Later works introduced an integration of the previous result with UML. The OCL/NEREUS transformation rules were prototyped [Favre00; Favre03a]. The obtained results show the feasibility of our approach.

As a perspective to this work, we foresee the integration of our results in the existing UML CASE tools. Also, we foresee to use these foundations to define rigorous round-trip processes, which support working with design patterns. UML metamodel formalization in NEREUS could be used to establish the notion of behavioral equivalence that is fundamental for refactoring.

REFERENCES

[Abrial96] J. Abrial: *The B Book: Assignning Programs to Meanings*. Cambridge University Press, 1996.

[Alencar 91] A. Alencar, J. Goguen: "OOZE: An Object-oriented Z Environment", *Proceedings of the European Conference on Object-oriented Programming, ECOOP 91, Lecture Notes in Computer Science 512,* pp. 180-199, Springer-Verlag, 1991.

[Ahrendt02] W. Ahrendt, T. Baar, B. Beckert, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, P. Schmitt: "The KeY System: Integrating Object-Oriented Design and Formal Methods", *Proceedings of FASE 2002 ETAPS 02*, Grenoble, France. Available at http://i12www.ira.uka.de/~projekt/index.html, 2002.

[Astesiano02] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. Mosses, D. Sannella; A. Tarlecki: "CASL: The Common Algebraic Specification Language", *Theoretical Computer Science*, 286(2), pp.153-196, 2002.

[Barbier03] F. Barbier, B. Henderson-Sellers, A. Le Parc-Lacayrelle, J. Bruel: "Formalization of the Whole-Part Relationship in the Unified Modeling Language", *IEEE Transactions on Software Engineering*, Vol. 29, no. 5, 2003

[Bordeau95] R. Bordeau, B. Cheng: "A Formal Semantic for Object Model Diagrams", *IEEE Transactions on Software Engineering*, Vol. 21, No 10, October,1995.

[Breu97] R. Breu, R. Grosu, F. Huber, B. Rumpe, W. Schwerin : "Towards a Precise Semantics for Object-Oriented Modeling Techniques", *Proceedings of the ECOOP'97, Lecture Notes in Computer Science 1241*, pp. 314-364, Springer-Verlag, 1997.

[Bruel98] J. Bruel, R. France: “Transforming UML Models to Formal Specifications”, *Proc. of <<UML>> 98- Beyond the notation, Lecture Notes in Computer science 1618*, Springer-Verlag, 1998.

[Case03] UML Tools: Available www.objectsbydesign.com/tools

[Evans98] A. Evans, R. France, K. Lano, B. Rumpe: “The UML as a Formal Modeling Notation”, *Computer Standards & Interfaces*, 19, 1998.

[Favre98] L. Favre: “Object-Oriented Reuse through Algebraic Specifications”, *Technology of Object-Oriented Languages and Systems, Tools 28* (C. Mingins;B. Meyer eds.), pp. 101-112, IEEE Computer Society, 1998.

[Favre00] L. Favre, L. Martínez, C. Pereira: "Transforming UML Static Models into Object-Oriented Code". *Technology of Object Oriented Languages and Systems, TOOLS 37 (eds. B. Henderson-Sellers, B. Meyer), IEEE Computer Press*, pp 170-181, Australia, 2000.

[Favre01] L. Favre: “A Formal Mapping between UML Static Models and Algebraic Specifications”. *Practical UML-Based Rigorous Development Methods- Countering or Integrating the eXtremist (A. Evans, R. France, A. Moreira, B. Rumpe eds.), Lecture Notes in Informatics (P 7) SEW*, pp. 113-127, GI Edition Konner Kollen-Verlag, Alemania, 2001.

[Favre03a] L. Favre, L. Martínez, C. Pereira: “Forward Engineering and UML: From UML Static Models to Eiffel Code”. *UML and the Unified Process* (Liliana Favre editor) Chapter IX. pp. 199-217, IRM Press, USA, 2003.

[Favre03b] L. Favre: *The Nereus Language*, Technical Report, INTIA, Universidad Nacional del Centro, Argentina, 2003.

[France97] R. France, J. Bruel, M. Larrondo-Petrie: “An Integrated Object-Oriented and Formal Modeling Environment”, *JOOP*, November-December, 1997

[Gamma95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object Oriented Software*, Addison Wesley, Reading (Mass), 1995.

[Gogolla97] M. Gogolla, M. Richters: ”On Constraints and Queries in UML”, *Proceedings UML’97 Workshop The Unified Modeling Language- Technical Aspects and Applications.* Physica-Verlag, Heidelberg, pp. 109-121, 1997.

[Gogolla02] M. Gogolla, B. Henderson-Sellers: “Formal Analysis of UML Stereotypes within the UML Metamodel”, *Proceedings of <<UML>> 2002, 5th Int. Conf. Unified Modeling Language (S. Cook; H. Hussmann; J.M. Jezequel, eds.), Lecture Notes in Computer Science*, Springer-Verlag, 2002

[Hussmann99]H. Hussmann, M. Cerioli, G. Reggio, F. Tort: *Abstract Data Types and UML Models*, Report DISI-TR-99-15, University of Genova, 1999.

[Kim99] S. Kim, D. Carrington: "Formalizing the UML Class Diagram using OBJECT-Z", *Proceedings of UML 99, Lecture Notes in Computer Science1723*, pp. 83-98, Springer-Verlag, 1999.

[Kim02] S. Kim, D. Carrington: "A Formal Model of the UML Metamodel: The UML State Machine and Its Integrity Constraints", *Lecture Notes in Computer science 2272*, pp. 497, Springer-Verlag, 2002.

[Kleppe 03] Anneke Kleppe, Jos Warmer, Wim Bast: *MDA Explained. The Model Driven Architecture: Practice and Promise*, Addison Wesley, April 2003.

[Lano 91] K. Lano: "Z++, An Object-Oriented Extension to Z", *Z User Workshop, Springer Workshops in Computing*, pp.151-172, 1991.

[Leavens96] G. Leavens: "An Overview of Larch/C++: Behavioral Specification for C++ Modules", *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, Kluwer Academic Publishers, Chapter 8, pp. 121-142, 1996.

[Leavens02] G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby: *JML Reference Manual Draft Revision 1.1*, Available at: www.cs.iastate.edu/~leavens, 2002

[McUmber01] W. McUmber, B. Cheng: "A General Framework for Formalizing UML with Formal Languages", *IEEE International Conference on Software Engineering (ICSE01)*, Canada, 2001.

[Meyer92] B. Meyer *Eiffel: The Language*. Prentice Hall, 1992.

[OMG 03] OMG (eds): *Unified Modeling Language Specification: Version 1.5*, 2003,. Available at www.omg.org

[Overgaard98] G. Overgaard: "A Formal Approach to Relationships in the Unified Modeling Language", *Proceedings of Workshop on Precise Semantic of Modeling Notations*, International Conference on Software Engineering, ICSE'98, Japan, 1998.

[Padawitz00] P. Padawitz: "Swinging UML: how to Make Class Diagrams and State Machines Amenable to constraint Solving and proving" *Proc. of <<UML>> 2000-The Unified Modeling Language . Lecture Notes in Computer Science 1939*, pp 265-277, Springer, 2000.

[Paige02] R. Paige, L. Kaminskaya, J. Ostroff: "BON-CASE: An Extensible CASE Tool for Formal Specification and Reasoning", *Journal of Object Technology (JOT)* vol 1, No 3, Special Issue: TOOLS USA 2002 Proceedings, pp. 77-96, 2002.

[Rapanotti 92] L. Rapanotti, A. Socorro. *Introducing FOOPS*. Report PRG-TR-28-92, Programming Research Group, Oxford University Computing Laboratory, 1992.

[Smith 00] G. Smith: *The Object-Z Specification Language. Advances in Formal Methods*, Kluwer Academic Publishers, 2000.

[Reggio99] G. Reggio, E. Astesiano, C. Choppy: *CASL-LTL: A CASL extension for dynamic systems*. Available at ftp://ftp.disi.unige.it/person/ReggioG, 1999.

[Reggio01] G. Reggio, M. Cerioli, E. Astesiano: "Towards a Rigorous Semantics of UML Supporting its Multiview Approach", *Proceedings of Fundamental Approaches to Software Engineering (FASE 2001). Lecture Notes in Computer Science 2029*, pp. 171, Springer-Verlag, 2001.

[Snook00] C. Snook, M. Butler: *Tool-Supported Use of UML for Constructing B Specifications*, Technical Report, Department of Electronics and Computer Science, University of Southampton, United Kingdom, 2000.

[Warmer 03] Jos Warmer, Anneke Kleppe: *The Object Constraint Language. Second Edition. Getting your Models Ready for MDA*, Addison Wesley, August 2003.

[Ziemann03] P. Ziemann, M. Gogolla: "Validating OCL Specifications with the USE Tool- An Example Based on the BART CASE Study", *Electronic Notes in Theoretical Computer Science 80,* 2003.

About the author

Liliana Favre is a professor of Computer Science at Computer Science Department in the Universidad Nacional del Centro, Argentina. She is researcher of CIC (Comisión de Investigaciones Científicas de la Provincia de Buenos Aires). Currently, she is research leader of the "Software Technology" group at Universidad Nacional del Centro. Her current research interests are focused on rigorous software and system engineering mainly on the integration of formal techniques with UML. She can be reached at lfavre@arnet.com.ar.