A Formal Mapping between UML Static Models and Algebraic Specifications

Liliana M. Favre

Intia
Universidad Nacional del Centro de la Pcia. de Buenos Aires
CIC (Comisión de Investigaciones Científicas de la Pcia. de Buenos Aires)
Argentina
lfavre@arnet.com.ar

Abstract: There are several reasons to specify UML models in a formal way The most important are to avoid inconsistencies and ambiguities and to do verification and forecasting of system properties. In this paper we propose a systematic approach to transform UML static models into algebraic specifications. We define the GSBL^{oo} algebraic language to cope with concepts of UML. Firstly, we give a formal description for UML static models using GSBL^{oo}. Then, we describe how to translate UML constructs to GSBL^{oo} constructs. In particular, we show how to transform UML relations into GSBL^{oo}. We propose a system of transformation rules to translate OCL constraints to GSBL^{oo}.

1 Introduction

In recent years, the Unified Modeling Language has emerged as a prominent 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 [Om99].

There exists a great number of UML Case tools that facilitates code generation and reverse engineering of existing software systems. Unfortunately, techniques currently available in these tools provide little support for validating models in the design stages and they are not sufficient for the complete automated code generation. Probably, this is mostly due to the lack of a precise semantics of UML and OCL. Another source of problems in these processes is that, on the one hand, UML models contain information that can not be expressed in object-oriented languages and on the other hand, the object-oriented languages express implementation characteristics that have no counterpart in the UML models. For example, languages like C++, Java and Eiffel do not allow us to express associations, their cardinality and their OCL constraints. It is the designer‘s responsibility to make good use of this information either selecting an appropriate implementation from a limited repertoire or implementing the association by himself .

A variety of advantages have been attributed to the use of formal software specification to solve these problems. It is commonly accepted that a formal specification can reveal gaps, ambiguities and inconsistencies. Any verification of UML models could take place on their corresponding specification using reasoning techniques provided for algebraic formalism before coding starts. Furthermore, a precise semantics provides the basis for automated forward engineering.

In previous work we have proposed a rigorous process to forward engineering UML static models using the algebraic language GSBL^{oo}([Fa98]; [FC01]). Our contribution was towards an embedding of the code generation within a rigorous process that facilitates reuse. We have described the formal model *SpReIm* for defining structured collections of reusable components that integrates algebraic specifications and object-oriented code. The manipulation of *SpReIm* components by means of building operators (*Rename, Hide, Combine, Extend*) is the basis for the reusability. Eiffel was chosen as the language to demonstrate the feasibility of our approach.

The emphasis in this contribution is given to the first steps in the road from UML to code. We describe how to transform UML static models that are specified in OCL, into GSBL^{oo}. We design the GSBL^{oo} language to cope with concepts of the UML models. In particular, this language is relation-centric, that is, it is possible to express different kinds of relations (dependency, association, aggregation, composition, etc) as primitives to develop specifications. We propose a system of transformation rules that allows us to automatically translate OCL specifications (preconditions, postconditions, invariants and general constraints) to GSBL^{oo}.

The paper has the following structure. We start by looking at related work in Section 2. In Section 3 we describe the GSBL^{oo} language. Section 4 outlines a rigorous process to transform UML static models into GSBL^{oo}. In Section 5 we describe how to integrate *OCL* specifications and GSBL^{oo}. This is followed by conclusions (Section 6).

2 Related Work

In the late 80s, 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++ [La90], OBJECT-Z [Ca89] or OOZE [AG91] can be mentioned. Another language with object-oriented characteristics and based on OBJ3 [GM99] is FOOPS [RS92]. Among the most recent ones the OBLOG language is being developed as the basis for a commercial product. Within the academic world there exist other developments associated with the OBLOG family: TROLL [Ju96], GNOME [SR94], LCM [FW93] and ALBERT [WD98] whose common semantic basis is the temporal logic. CASL [CO99] wants to be the central member of a language family, that includes simple languages obtained by restriction and more advanced ones obtained by extension (for example to specify reactive systems). Reflexive languages based upon the rewriting logic such as MAUDE [Cl99] and CafeOBJ [DF98] are being already designed and implemented.

A lot of work has been carried out dealing with a semantics for object oriented models. For example, [FBL97] describes the formalization of FUSION models in Z. [BC95] introduces a method to derive LARCH algebraic specifications from class diagrams.

The UML formalization is an open problem yet and many research groups have already achieved the formalization of parts of the language. 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 a different UML subset and they are based on different formalisms. The Precise UML Group, pUML, is created in 1997 with the goal of giving precision to

UML [Ev98]. [BF98] describes how to formalize UML models using Z, [La95] using Z++, [Br97] does a similar job using stream oriented algebraic specifications, [GR97] does this by transforming UML to TROLL, [Ov98] achieves it by using operational semantics. [KC99] and [MA00] integrate UML and OBJECT-Z. [FH98] describes advanced metamodeling and notation techniques that allow the enhancement of UML.

Currently there are few development methods that include OCL. The most important is Catalysis [DW99]. [Bi99] describes an approach for specifying UML interface constraints and proving the correctness of implementation relations between interfaces and classes. [MC99] examines the expressive power of OCL in terms of navigability and computability. [VJ99] proposes a tool, Alcoa, for analyzing object models that uses its own language, Alloy, based on Z. [RG00] proposes an approach for validation of UML models and OCL constraints that is based on animation. [Bo00] describes a graph-based semantics for OCL and a systematic translation of OCL constraints into expressions over graph rules. [Hu99] analyzes the integration of UML models, OCL constraints and CASL. [Pa00] proposes the first steps towards a translation of class diagrams, OCL constraints and state machine into Swinging Types.

What the latter formalizations have in common is the fact that they give semantics to UML and certainly this is also another goal in our work. However, it is not an end in itself, we want to give semantics to UML static models in order to transform design artifacts into code by means of a rigorous process that facilitates reuse, evolution and maintenance of the software.

The following differences between our approach and some of the existing ones are worth mentioning. In the first place, the GSBL^{oo} language was defined taking into account the structuring mechanisms of UML. The central innovation of this language as regards other ones is that it is relation- centric. GSBL^{oo} allows us to keep a trace of the structure of UML models in the specification structure that will make easier to maintain consistency between the various levels when the system evolve. On the other hand, a different approach is introduced for the integration of static diagrams UML specified in OCL with algebraic languages based on the transformational paradigm. Transformations are supported by a library of reusable schemes and by a system of transformation rules that allow translating OCL expressions into GSBL^{oo} step by step. All the proposed transformations can be automated, they allow traceability and can be integrated to rigorous processes of forward and reverse engineering extending the ones supported by the existing CASE tools.

3 The GSBL^{oo} Language

GSBL (Generic Specification Base Language) is a kernel language for the incremental construction and organization of specifications [CO88]. GSBL^{oo} is an object-oriented extension to GSBL designed specifically for facilitating specification of concepts of UML static models. In particular, it provides an explicit syntax for expressing UML relations. [BRJ99] distinguishes four kinds of UML relations: dependency, generalization, association and realization. A detailed description of them can be found in [Om99].

The treatment of associations in object-oriented languages as little more than pointer-value attributes has confined them to a second-class status. But associations are semantic constructions of equal weight to the classes and generalizations in the UML models. In fact, the associations allow abstracting the interaction between classes in the design of large systems and they affect the partition of the systems into modules. For full benefit, these relations should be made available in object oriented languages as primitives. Extensions of algebraic languages to support object-oriented concepts follow the same lines of thought, in general, algebraic specifications have much less structure than the original object-oriented models.

GSBL^{oo} includes a library of constructor types that captures the semantics of associations. The variety of associations may lead to the impression that it should have several language mechanisms to express them. Recognizing only A few kinds is not representative of the variety of associations. Then, we decide to provide flexible mechanisms for defining a new association just like an existing one, but with its own special properties. GSBL^{oo} helps designers produce specifications made of autonomous elements. This approach allows us to shape and grow the GSBL^{oo} to our needs and to define classes and associations as independent units, thus relieving the designer writing the specification from the burden of replicating that generic semantics for each concrete application. Following, we describe in more detail the GSBL^{oo} syntax.

3.1 Representing Object Classes

In Fig. 1 we show the syntax of a GSBL^{oo} object class specification:

```
OBJECT CLASS className [<parameterList>]
USES <usesList>
REFINES <refinestList>
RESTRICTS < restrictsList>
BASIC CONSTRUCTORS <constructorList>
DEFERRED
SORTS <sortList>
OPS <opsList>
EQS <varList> <equationList>
EFFECTIVE
SORTS <sortList>
OPS <opsList>
EQS <varList> <equationList>
END-CLASS
```

Fig. 1 : GSBL^{oo} Class Syntax

In GSBL^{oo} strictly generic components can be distinguished 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).

The USES clause expresses dependency relations. The specification of the new class is based on the imported specifications declared in *<usesList>* and their visible constituents may be used in the new specification.

The power of the inheritance comes from the fusion of a type mechanism (the definition of a new type as a special case of existing types) with module mechanisms (the definition of a module as an extension of existing modules). GSBL^{OO} distinguishes two different mechanisms: REFINES and RESTRICTS. The first one relies on the module viewpoint of classes while the second one relies on the type viewpoint. In the REFINES clause the specification of the class is built from the union of the specifications of the classes appearing in the *<refineList>*. The components of each one of them become components of the new class, and its own sorts and operations become the own sorts and operations of the new class. The RESTRICTS clause builds the specification of the new class by adding a value-constraint in the specification of the old one.

GSBL^{oo} allows us to define local instances of a class in the USES and REFINES clauses by the following syntax: *ClassName*[*<bindingList>*] where the elements of *<bindingList>* can be pairs of class names C1:C2, being C2 a component of *ClassName*; pairs of sorts s1:s2, and/or pairs of operations o1: o2 with o2 and s2 belonging to the own part of *ClassName*.

The sort of interest of a class (if any) is also implicitly renamed each time the class is substituted or renamed. Instances of parameterized classes can be defined with the usual syntax *ClassName[<actualParameterList>]* when no additional renaming or substitution is needed.

The syntax of a complete class can include the BASIC CONSTRUCTORS clause that refers to generator operations.

GSBL^{oo} distinguishes incomplete and complete parts. The DEFERRED clause declares new sorts, operations or equations that are incompletely defined. The EFFECTIVE clause either declares new sorts, operations or equations, that are 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 any of the three levels of visibility for operations public, protected and private. They are expressed by prefixing the symbols: +(public), #(protected) and -(private). If we do not decorate an operation with a symbol of visibility it can be assumed that it is public.

As an example, in Fig. 2 we show a GSBL^{oo} specification for an Object Class *Collection*. It must be taken into account that some operations are second-order operations (*forAll, exists and iterate*).

3.2 Representing Associations

Associations are defined as standard elements in GSBL^{oo}. ASSOCIATION is a taxonomy of constructor types that classifies associations according to:

- Its kind (aggregation, composition, association, etc).
- Its degree (unary, binary, ternary and in general as n-ary).
- Its navigability, for example a binary association can be unidirectional or bi-directional.
- Its connectivity (one-to-one, one-to-many, many-to-many, etc).

```
OBJECT CLASS Collection [Elem:ANY]
USES Boolean, Nat
BASIC CONSTRUCTORS create, add
DEFERRED
SORT Collection
OPS
create: → Collection
add: Collection x Elem → Collection
size: Collection → Nat
count: Collection x Elem → Nat
EFFECTIVE
OPS
 isEmpty: Collection → Boolean
 includes: Collection x Elem → Boolean
 includesAll:
 Collection x Collection → Boolean
 forAll :
 Collection x (Elem->Boolean)→ Boolean
 exists:
 Collection x (Elem->Boolean)→ Boolean
iterate:
Collection x (Elem x Acc) x (->Acc) -> Acc
EQS{c,c1:Collection;e:Elem;
f: Elem->Boolean; base : ->Acc}
isEmpty (create) = True
isempty(add (c,e) ) = False
includes(create,e)= False
includes(add(c,e),e1)= if e=e1 then True
                         else includes (c,e1)
includesAll.(c,create) = True
includesAll (c,add (c1,e))=
includes(c,e) and includesAll (c,c1)
forAll (create,f)= True
forAll (add (c,e),f )= f (e) and forAll (c,f)
exists (create,f) = False
exists (add(c,e),f ) = f (e) or exists (c,f)
iterate(create, f, base) = base
iterate(add(c,e),g, base)= g (e, iterate (c, g ,base))
END-CLASS
```

Fig 2 : OBJECT CLASS Collection

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

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

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

The IS clause expresses the instantiation of <*relationName*> 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 relation name is enclosed in quotation marks, before the keywords ASSOCIATES or HAS-A. These keywords 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, i.e operations that do not belong to any of the associated classes, but rather to the association itself. Such an association is called an ASSOCIATION CLASS.

The mechanism provided by GSBL^{OO} for grouping classes and relations is the package. It is possible to specify families of packages by generalization. A specialized package can be used anywhere a more general package can be used. Elements made available to another package by generalization have the same visibility in the heir as they have in the owning package.

3. 3 An Example

Fig. 3 shows a simple class diagram that introduces two classes (*Person* and *Meeting*) and a bidirectional association between them. We have meetings in which persons may participate. Participants know about the meetings they are involved with, and meetings know their participants . This example was analyzed in [Hu99] and [Pa00]. We propose a different approach based on GSBL^{oo}. This language helps us to built specifications made of autonomous units (classes and relations) connected by a simple structure.

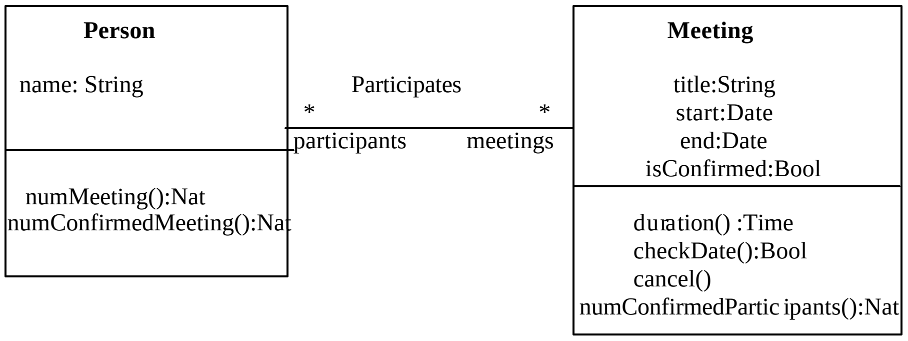

Fig 3: Class Diagram Person&Meeting

Fig. 4 shows the OCL constraints that describe the effect of some operations. Fig. 5 depicts the GSBL^{oo} specification of the class diagram and Fig. 6 shows the GSBL^{oo} specification for the *Participates* Association.

context Person:: numMeeting ( ): Nat
post: result = self.meetings -> size
context Person :: numMeetingConfirmed ( ) : Nat
post: result= self.meetings -> select (isConfirmed) -> size
context Meeting :: isConfirmed (): Bool
post: result= self.checkdate() and self.numConfirmedParticipants > min
context Meeting :: duration ( ) : Time
post: result = timeDifference (self.end, self.start)
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)))

Fig. 4 : OCL Constraints

PACKAGE Person&Meeting
OBJECT CLASS Person
USES String, Nat
<<Participates>> **ASSOCIATES** Meeting
BASIC CONSTRUCTOR create_Person
EFFECTIVE
SORT Person
OPS create_Person: String -> Person
name: Person -> String
numMeetings: Person -> Nat
numMeetingsConfirmed: Person -> Nat
EQS{ p:Person; m:Meeting; s: String}
name(create_Person(s))= s
numMeetingsConfirmed (p) =
size(select$_m$ (getMeetings(Participates,p),
isConfirmed (m))
numMeetings (p)=
size (getMeetings(Participates, p))
END-CLASS
OBJECT CLASS Meeting
USES String, Date, Boolean, Time
<<Participates>> **ASSOCIATES** Person
BASIC CONSTRUCTORS
create_Meeting
EFFECTIVE
SORT Meeting
OPS create_Meeting: String x Date x Date x Boolean -> Meeting
title: Meeting -> String
start: Meeting -> Date
end : Meeting -> Date
isConfirmed: Meeting -> Boolean
duration: Meeting-> Time
checkDate: Meeting -> Boolean
consistent: Meeting x Meeting -> Boolean ...
EQS{m,m1:Meeting; t:Time; s:String; d,d1:Date; b:Boolean; p: Person}
title (create_Meeting(s,d,d1,b)) = s
start (create_Meeting(s,d,d1,b)) = d
end (create_Meeting(s,d,d1,b)) = d1
duration (m)=
timeDifference (end(m),start(m))
isConfirmed (cancel(m))= False
isConfirmed (m) = checkDate(m) and
NumParticipantsConfirmed(m) > min
checkDate(m) = forAll$_{m1}$ (collect$_p$
(getParticipants(Participates,m),
getMeetings(Participates, p)),
consistent (m,m1))
consistent(m,m1)= not (isConfirmed(m1))
or (end(m) < start(m1) or end(m1) <
start(m))...
END-CLASS
ASSOCIATION Participates
IS Bidirectional-Set [Person: Class1; Meeting: Class2; participants:role1; meetings: role2; *:mult1; *: mult2; +: visibility1; +: visibility2]
END
END-PACKAGE

Fig. 5 : PACKAGE Person&Meeting

```
RELATION CLASS Bidirectional-Set
  -- Bidirectional /* to */ as Set
REFINES BinaryAssociation    [Person:Class1;
Meeting:Class2]
USES Set_Person: Set [Person],
Set_Meeting: Set[Meeting]
BASIC CONSTRUCTORS create,  addLink
EFFECTIVE
name, frozen , changeable , addOnly ,
getRole1, getRole2, getMult1,getMult2,
getVisibility1, getVisibility2, isRelated,
isEmpty, rightCardinality, leftCardinality
create: Typename→ Bidirectional-Set
addLink:
Bidirectional-Set(b) x Person(p) x Meeting(m)→
Bidirectional-Set
pre: not isRelated(a,p,m)
isRightLinked:
Bidirectional-Set x Person → Boolean
isLeftLinked:
Bidirectional-Set x Meeting → Boolean
getMeetings: Bidirectional-Set(a) x Person(p) →
Set_Meeting
pre: isRightLinked(a,p)
getParticipants:
Bidirectional-Set(a) x Meeting(m) →
Set_Person
pre: isLeftLinked(a,m)
remove:
Bidirectional-Set (a) x Person (p) x Meeting(m)
→ Bidirectional-Set
pre: isRelated(a,p,m)
EQS{ a:Bidirectional-Set; p,p1: Person;
c2,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. 6 : RELATION CLASS Bidirectional-Set

4 From UML Class Diagram to GSBL^{00} Specifications

Starting from UML static diagrams, an incomplete algebraic specification can be automatically built. It contains the highest information that can be extracted from the UML class diagram and is obtained by translating the UML constructions and OCL constraints to GSBL^{oo}.

4.1 Mapping Classes and Associations

Given a basic UML diagram with OCL annotations a PACKAGE, whose components will be OBJECT CLASS, ASSOCIATION CLASS and relation definitions, is automatically generated. For each class shown in the diagram an OBJECT CLASS is built and for each association (ordinary, qualified or class-association) a new association is defined

These specifications are obtained by instantiating reusable schemes and classes already existing in a GSBL^{00}'s predefined library. Some of them are shown in Figure 7. *Box* specifies the class interface (attributes and methods). It is a refinement of *Cartes-Prod* that allows us to specify cartesian product of a different arity.

OBJECT CLASS Box
USES TP1:ANY,..,TPm:ANY
REFINES Cartes-Prod [T-attr$_1$:T1;
T- attr$_2$:T2;..; get-1: select-1 get-2:
select-2,...,set-1: modif-1,...,
set-n: modif-n]
DEFERRED
OPS
meth$_1$:Box x TPi$_1$ x TPi$_2$x.....TPi$_n$$\rightarrow$TPi$_j$
....
meth$_r$: Box x TPr$_1$ xTPr$_2$.....x TPr$_p$ -> TPr$_k$
END-CLASS

OBJECT CLASS Cartes-Prod
[T1:ANY,..,Tn:ANY]
EFFECTIVE
create: T1 x ... x Tn$\rightarrow$ Cartes-Prod
modif-i: Cartes-Prod x Ti $\rightarrow$ Cartes-Prod
select-i:Cartes-Prod$\rightarrow$Ti **1£ i£ n**
EQS{cp:Cartes-Prod;t1:T1;ti,
ti´:Ti...tn:Tn }
select-i (Create(t1,t2,...,ti,...,tn)) = ti
modif-i(create(t1,t2,...,ti,...,tn),ti´) =
create(t1,t2,..,ti´,..tn)
END-CLASS

Fig. 7 : *Box* and *Cartes-Prod* Schemes

Each OBJECT CLASS is obtained by instantiating the scheme of Fig. 8.

OBJECT CLASS A
USES U1,U2,...
REFINES Box [...:TP1;...:TPi;...:T-attr1;...:T-attri;...:meth1;..:methi,..]
<<Aggregation-i>> **HAS-A SHARED** Si
<<Composition-j>> **HAS-A NON-SHARED** Nj
<<Association-k>> **ASSOCIATES** Mk
...
END-CLASS

Fig. 8: Constructing an OBJECT CLASS

Generalization/specialization relations are expressed by means of the REFINES clause. *Aggreggation-i*, *Composition-i* and *Association-i* are new relations defined by instantiating constructor types. Fig. 9 shows an instantiation for *Person* class and the resulting class.

Preconditions, postconditions and invariants in OCL will be translated to preconditions and axioms in GSBL^{oo}. In the next section we describe in detail how to transform OCL constraints to GSBL^{oo}.

Thus, an algebraic specification can be semi-automatically built. We could use this specification to detect inconsistencies in the class diagrams. We could simulate the behavior of a system. We could start with an empty system where no objects and association links exist. As a next step, we could create objects and links for rigorous semantic analysis of the UML models.

```
OBJECT CLASS Person
REFINES Box [String:TP1; Nat:TP2;
name:get-1; setName:Set-1; String:T-attri1;
numMeeting:meth1;
numMeetingConfirmed : meth2]
<<Participates>> ASSOCIATES Meeting
END-CLASS
OBJECT CLASS Person
USES String, Nat
<<Participates>> ASSOCIATES
Meeting
BASIC CONSTRUCTOR create_Person
EFFECTIVE
```

```
SORT Person
OPS create_Person: String -> Person
name: Person -> String
setName: Person x String -> Person
numMeetings: Person -> Nat
numMeetingsConfirmed: Person -> Nat
EQS{ p:Person; m:Meeting;
 s,s: String}
name(create_Person(s))= s
setName(create_Person(s), s')=
create_Person (s') ...
END-CLASS
```

Fig. 9: Constructing the OBJECT CLASS *Person*

5 From OCL to GSBL^{oo} Specifications

Analyzing OCL constraints we can derive axioms that will be included in the GSBL^{oo} specifications. Preconditions written in OCL are used to generate preconditions in GSBL^{oo}. Postconditions and invariants allow us to generate axioms in GSBL^{oo}.

An operation can be specified in OCL by means of pre- and post-conditions:

```
Typename:: Operation name ( parameter1: Type1,...):Return Type
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 the 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, one has to postfix the property name with "@", followed by the keyword "pre".

The transformation process of OCL constraints to GSBL^{oo} is supported by a system of transformation rules. Some of them are partially shown in Fig. 10.

Let *Translate* be functions that translate logical expressions of OCL into first-order formulae in GSBL^{OO}. The translation of expressions of the form e.op, where e is a complex expression of the form e1.op1, is recursively defined by *Translate*(e.op)=op(*Translate*(e)).

It is worth clarifying for the axioms generation that a basic functionality *Typename::Operation name (parameter1: Type1,...):Return Type*, is translated into GSBL^{OO} syntax as *Operation name:Typename x Type1 x...* ® *ReturnType* . In the same way the axioms terms must be translated respecting the GSBL^{OO} syntax, for example, *collection->size=0* is translated to *size(collection) = 0.* For each type T in a class diagram we associate with it a sort that conforms to the type. We define a mapping ***r***:*OCLType ->GSBLSort*. For example, in the context of the *Participates* association and a *Person* p, ***r****(meetings)=getMeeting (Participates, p)* . Fig. 11 exemplifies the transformation of OCL constraints for *numMeetingsConfirmed* and *numMeetings* operations (see Fig. 4, Fig. 5 and Fig. 9).

1. T ® Op (t1:T1,t2:T2, ..) : ReturnType **post:** result = T → **iterate** (elem: Type; acc: Return Type = <exp> | <boolean-expression-with-elem-and-acc>)	**OPS** Op:T x T1 x T2 x… → Return Type **EQS {**t:T; e: Type;t1:T1;t2:T2;… **}** Op (empty_T, t1,t2,..) = *Translate*(exp) Op (const_T(t,e), t1, t2,.....)= *Translate*(boolean-expression-with-elem-and acc) with [elem |→ e; acc: Op(t,t1,t2,...)]
2. T® Op (<list_param>) : Boolean **post:** result = T →**forAll** (elem: Type | <boolean-expression-with-elem>)	**OPS** Op:T x (Elem->Boolean) x… → Boolean **EQS {**t:T; e: Type; f: Elem->Boolean… **}** Op (empty_T, <list_param>)= TRUE Op (const_T(t,e), <list_param>)= Op(t, <list_param>) AND f(e) f(e) is true if <boolean-expression-with-elem>
3. T® Op (<list_param>) : Boolean **Post:** Result = T→ **exists** (e: Type | <boolean-expression-with-e>)	**OPS** Op:T x (Elem->Boolean) x… → Boolean **EQS {**t:T; e: Type;f:Elem->Boolean… **}** Op (empty_T, <list_param>)= FALSE Op (const_T(t,e), <list_param>)= Op(t, <list_param>) OR f(e)
4. T ® Op (<list_param>) : ReturnType **Post:** <exp_1> = <exp_2>	**OPS** Op:T x … → Return Type **EQS {… }** *Translate*(exp_1) = *Translate*(exp_2)
5. T ® Op (<list_param>) : Return Type **post**: result = <exp>	**OPS** Op:T x … → Return Type **EQS {**t:T; ... **}** Op (t, <list_param>) = *Translate*(exp)
6. T->select|exists|forAll (v:Type| <boolean-expression-with-e>)	select_v|exists_v|forAll_v (ρ(T), Translate(boolean-expression-with-e**)**

Fig. 10: From OCL to GSBL^{oo}: Transformation Rules

```
 OBJECT CLASS Person ...
 EQS{ p:Person; m:Meeting; s: String}
 -- Rule  5/Rule 6
numMeetingsConfirmed (p) =   size(select_m (getMeetings(Participates,p), isConfirmed(m))
 -- Rule 5    numMeetings (p)=  size (getMeetings(Participates, p))
 END-CLASS
```

Fig. 11 : Constructing axioms for Person

As a special case, we have applied it to obtain GSBL^{oo} specifications of OCL types. Fig. 12 partially shows a translation of Collection specification in OCL to GSBL^{oo} specifications. For transforming preconditions and postconditions we take into account the distinction between constructor and observer operations. The programmer has to choose the adequate constructor operations. For example, in *Rule 1*, if T is instantiated with *Collection* then *const*$_T$ is instantiated with *add* and *empty*$_T$ with *create*.

```
Collection -> size :Integer
post:  result= collection->iterate(elem; acc:Boolean=False | acc +1)
Collection -> count (object:OclAny) :Integer
post: result=collection -> iterate(elem; acc:Boolean =False| if elem=e then True else acc)
Collection ● includes ( e:T ) : Boolean
post: result= Collection→iterate (elem; acc:Boolean=False | if elem=e then True else acc )
Collection ● forAll (expr:OclExpression) :Boolean
post: result= collection→iterate (elem; acc:Boolean=True | acc and exp)
Collection ● exists (expr:OclExpression) :Boolean
post: result= Collection→iterate (elem; acc:  Boolean=False  | acc or exp)
Collection ● isEmpty : Boolean
post: result= (collection->size=0) ...
```

```
OBJECT CLASS  Collection[Elem:ANY]
USES  Boolean, Nat
BASIC CONSTRUCTORS  create, add
                            -- See Figure 2
EQS{c,c1:Collection;e:Elem;f: Elem->Boolean; base : ->Acc}
size( create)= 0         size(append(s,e))= 1 + size(s)                                       Rule 1
count(create,e) =0       count (append (s,e),e1) = if e=e1 then count (s) +1 else count (s,e1)  Rule 1
isEmpty( c) =((size(c)=0 )                                                                     Rule 4
includes(create,e)=False   includes(add(c,e),e1)= if e=e1 then True else includes(c,e1)         Rule 1
forAll (create,f)= True    forAll (add(c,e),f)= f(e) and forAll(c,f)                            Rule 2
exists (create,f)= False   exists (add(c,e),f)= f(e) or exists (c,f )                           Rule 3
END-CLASS
```

Fig. 12 : Constructing axioms for Collection

6 Conclusions

In previous work we outline a rigorous process to forward engineering UML static models [FC01]. In this paper we have presented the first steps of this process. A formal mapping between UML models and GSBL^{oo} is described. Our approach is directly connected with the goal of reusability. The aim is to construct specifications by combining standard prefabricated elements. We describe a system of transformation rules to transform OCL to GSBL^{oo}. A semantics for OCL expressions together with the semantics for class diagrams has been defined in terms of GSBL^{oo}.

We believe that our approach provides several advantages. All the information contained in the UML models (associations, their cardinality, OCL constraints, etc) is translated to specifications and will have implementation implications. The transitions between the UML diagrams and algebraic specifications can be done exclusively by applying automated transformations that preserve the integrity between UML and GSBL^{oo}..

Although a tool that assists the proposed method does not exist, key phases of this one have been prototyped. To allow the automatic generation of algebraic specifications from UML models that are specified in OCL, we developed a transformation system prototype. It was built in Mathematica which allows one to use rewrite rules and to prove properties [FMP00]. The obtained results show the feasibility of our approach, however we can not make an analysis of the pragmatic implications of it. In the future we foresee the integration of our approach in the existing Case-tools environments.

References

[AG91] Alencar, A; Goguen, J.: OOZE: An Object-oriented Z Environment. In: Proc. of the European Conference on Object-Oriented Programming, ECOOP 91, Lecture Notes in Computer Science 512, Springer-Verlag, 1991; S 180-199.

[BC95] Bourdeau, R.; Cheng, B.: A Formal Semantics for Object Model Diagrams. IEEE Transactions on Software Engineering, 21 (10), 1995; S 799-821.

[BF98] Bruel, J.; France, R.: Transforming UML Models to Formal Specifications. In: Proc. of UML'98-Beyond the notation, Lecture Notes in Computer Science 1618, Springer Verlag,1998; S 78-92.

[Bi99] Bidoit, M. et. al.: Correct Realizations of Interface Constraints with OCL. In: Proc, 2nd Int. Conf.. UML'99.The Unified Modeling Language-Beyond the Standard, Lecture Notes in Computer Science 1723, Springer Verlag, 1999; S 399-415.

[Bo00] Bottoni, P. et. al.: Consistency Checking and Visualization of OCL Constraints. In: (A Evans and S.Kent) <<UML>>2000. The Unified Modeling Language. Advancing the Standard. Third International Conference, Lecture Notes in Computer Science 1939, 2000; S 294-309.

[Br97] Breu, R. et. al. : Towards a Formalization of the Unified Modeling Language. TUM-I9726 Technische Universitat Munchen.1997.

[BRJ99] Booch, G.; Rumbaugh, J.; Jacobson, I.: The Unified Modeling Language-User Guide, Addison-Wesley, 1999.

[Ca89] Carrington, D. et. al.: OBJECT-Z: An Object-Oriented extension to Z. In: Proc. of Formal Description Techniques, FORTE'89, North Holland,Amsterdam, 1989; S 281-296.

[Cl99] Clavel, M. et. al.: MAUDE as a formal meta- tool. In: Proc. of FM'99, The World Congress on Formal Methods in the Development of Computing Systems. Toulouse, France, 1999.

[CO88] Clérici, S.; Orejas, F.: GSBL: An Algebraic Specification Language Based on Inheritance. In: Proc. of the European Conference on Object-oriented Programming ECOOP 88, 1988; S 78-92.

[CO99] COFI Task Group on Language Design: CASL The Common Algebraic Specification Language. In: www.bricks.dk/Projects/CoFi/Documents/CASL, 1999.

[DF98] Diaconescu, R.; Futatsugi, K.: The CAFEOBJ Report, The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification. .In: AMAST Series in Computing 6, 1998.

[DW99] D'Souza, D.; Wills, A.: Objects, Components, and Frameworks with UML. Addison Wesley, 1999.

[Ev98] Evans, A. et. al.: The UML as a Formal Modeling Notation. In: Proc. of UML'98-Beyond the Notation, Lecture Notes in Computer Science 1618. Springer, 1998.

[Fa98] 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.

[FBL97] France, R.; Bruel, J.; Larronde-Petri, M.: An Integrated Objet-Oriented and Formal Modeling Notations, JOOP, Nov/Dec, 1997; S 25- 34.

[FC01] Favre, L; Clérici, S.: 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, 2001.

[FH98] Firesmith, D.; Henderson-Sellers, B.: Clarifying specialized forms of association in UML and OML. JOOP,11(2) , 1998; S 47-50.

[FMP00] Favre, L.; Martínez, L.;Pereira, C.: From OCL to Algebraic Specifications. Technical Report. Intia. Departamento de Computación. Universidad Nacional del Centro. Argentina, 2000

[FW93] Feenstra, R.; Wieringa, R.: LCM 3.0: A Language for Describing Conceptual Models Syntax Definition. In: Rapport IR-344, Vrije Universiteit Amsterdam, 1993.

[GR97] Gogolla, M.; Ritchers, M.: On combining Semi-formal and Formal Object Specification Techniques. In: Proc. WADT97, Lecture Notes in Computer Science 1376, Springer, 1997; S 238-252.

[GM99] Goguen, J.; Malcolm, G.(Eds) Software Engineering with OBJ: Algebraic Specification in Action. Kluwer, 1999.

[Hu99] Hussmann, H. et. al.: Abstract Data Types and UML Models. Report DISI-TR-99-15, University of Genova, 1999.

[Ju96] Junclauss, R. et. al.: TROLL-A Language for Object Oriented Specification of Information Systems. In: ACM Transactions on Information Systems 14 (2), 1996; S 175-211

[KC99] Kim, S.; Carrington, D.: Formalizing the UML Class Diagram using Object-Z. In: Proc. UML 99, Lecture Notes in Computer Science 1723, 1999; S 83-98.

[La90] Lano, K.: Z++, An Object-Oriented Extension of Z. In: Proc. Z USER Workshop, Oxford, Springer-Workshops in Computing, 1990; S 151-172.

[La95] Lano, K.: Formal Object-Oriented Development. Springer-Verlag, 1995.

[MA00] Moreira, A.; Araújo, J.: Generating Object Z Specifications from Use Cases. In (J.Filipe) Enterprise Information Systems, Kluwer Academic Press, 2000; S 43-51.

[MC99] Mandel, L.; Cengarle, V.: On the Expressive Power of the Object Constraint Language OCL. Available: http://www.fast.de/projeckte/forsoft/ocl, 1999.

[Om99] OMG: Unified Modeling Language Specification, v. 1.3. Document ad/99-06-08, Object Management Group, 1999.

[Ov98] Overgaard, G.: 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, 1998.

[Pp00] Padawitz, P.: Swinging UML: How to Make Class Diagrams and State Machines Amenable to Constraint Solving and Proving.In: (Evans, A.; Kent,S.) Proc. of <<UML>> 2000-The Unified Modeling Language. Lecture Notes in Computer Science 1939. Springer, 2000; S 162-177.

[RG00] Richters, M.; Gogolla, M.: 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, 2000; S 265-277.

[RS92] Rapanotti, L.; Socorro, A.: Introducing FOOPS. Report PRG-TR-28-92, Programming Research Group, Oxford University Computing Laboratory, 1992.

[SR94] Sernadas, A.; Ramos, J.: The GNOME Language: Syntax, Semantics and Calculus. Technical Report, Instituto Superior Técnico, Lisboa, 1994.

[VJ99] Varizi, M.; Jackson, D.: Some Shortcomings of OCL, The Object Constraint Language of UML. Available: http://sdg.lcs.mit.edu/~dnj/publications.htm.

[WD98] Wieringa, R.; Dubois, E.: Integrating Semi-formal and Formal Software Specification Techniques Information System 23, 1998; S 159-178.