Generating FoCaLiZe Specifications from UML Models

Messaoud Abbas
USTHB. LSI, BP32 EL-Alia, Algiers &
College of Sciences and Technology,
El-Oued Univ. BP789 RP, Algeria.
abbasmessaoud@gmail.com

Choukri-Bey Ben-Yelles
Univ. Grenoble Alpes, LCIS,
Rue Barthélémy de Laffemas
F-26901 Valence, France.
choukri.ben-yelles@iut-valence.fr

Renaud Rioboo
CPR CEDRIC ENSIIE,
Square de la Résistance
F-91025 Evry, France.
Renaud.Rioboo@ensiie.fr

Abstract. UML is the defacto standard language to graphically describe systems in an object oriented way. Once an application has been modeled, Model Driven Architecture (MDA) techniques can be applied to generate code from such models. Because UML lacks formal basis to analyze and check model consistency, it is pertinent to choose a formal target language (in the MDA process) to enable proofs and verification techniques. To achieve this goal, we have associated to UML the FoCaLiZe language, an object-oriented development environment using a proof-based formal approach. In this paper, we propose a formal transformation of a subset of UML constructors composed of UML class diagrams annotated with OCL constraints into FoCaLiZe specification. Thanks to FoCaLiZe design and architectural features, the proposed mapping directly supports most of UML features such as multiple inheritance, function redefinition, late-binding, template and template binding which are not provided through other formal tools.

Keywords: UML, OCL, FoCaLiZe, Formal verification, Modeling, Semantics.

1. Introduction

The Unified Modeling Language (UML) [1] is the defacto standard to graphically and intuitively describe systems in an object oriented way. Currently, it is supported by a wide variety of tools, ranging from analysis, testing, simulation to code generation and transformation. The Object Constraint Language (OCL) [2] is the current formal standard of the Object Management Group. An OCL constraint is a precise statement which may be attached to any UML element. Using UML and OCL, we can only describe models and specify constraints upon them. No formal proof is available in a UML/OCL context to check the consistency of UML models or to check whether OCL properties hold in UML models.

In the last few years several approaches have been carried out in order to provide supporting tools to generate (using MDA techniques) a reliable software from an UML/OCL model and check its properties. In particular, several studies have focused on the transformation of UML/OCL models into formal methods. The most used formal tools are the B language [3], the Alloy formal tool [4], the Maude system [5] and the Isabelle/HOL [6] among several others.

In this paper, we present a translation from UML/OCL into the FoCaLiZe environment [7] following a compiling approach (by translation). The choice of FoCaLiZe does not solely rely on its formal aspects. Most of the UML design and architectural features are presented in FoCaLiZe [8] [9] [10]. It also supports all necessary logical constructors ($\forall$, $\exists$, $->$, $<->$, $\wedge$, $\vee$, ...) which enables a natural transformation of OCL constraints into FoCaLiZe. Our approach is motivated by the following arguments:

First, we support the transformation of most of UML conceptual and architectural features such as encapsulation, inheritance (generalization/specialization) and multiple inheritance, function redefinition, late-binding, dependency, UML template (parameterized classes) and template binding. Because these features are presented in FoCaLiZe through its own constructs without need of additional structures or invariants, we may keep the same development semantics of UML. Most of UML design and architectural features are not seamlessly considered through the aforementioned formal methods, usually with semantics loss of the original model and with deviation from the incremental intuition development provided by UML. In particular, multiple inheritance, function redefinition, templates and template bindings are not supported.

Second, the choice of FoCaLiZe environment is also motivated by its functional paradigm (with no side effects), by the power of its automated theorem prover Zenon [11] and its proof checker Coq [12]. Realizing proofs with Zenon makes the user intervention much easier since it manages to fulfill most of the proof obligations automatically.

This document is organized as follows: section 2 presents FoCaLiZe and the UML/OCL concepts, sections 3 and 4 explain our transformation approach. In section 5 we present the framework that integrates UML/OCL models and FoCaLiZe environment to check the consistency of UML/OCL models. Section 6 presents a comparison with related works.

2. FoCaLiZe and UML/OCL

2.1. FoCaLiZe

The FoCaLiZe [7] environment, initiated by T. Hardin and R. Rioboo, is an integrated development environment with formal features. A FoCaLiZe development is organized as a hierarchy of `species` that may have several roots. This hierarchy is built step by step (incremental approach), starting with abstract specifications and heading to concrete implementations using object oriented features such as inheritance and parameterization. A `species` groups together methods using ML-like types and expressions:

- The carrier type (`representation`), describes the data structure of the species. The representation of a species can depend on the representation of other species. It is mandatory and can either be explicitly given or obtained by inheritance.
- Function declarations (`signature`), specify functional types that will be defined later through inheritance (no computational body is provided at this stage).
- Function definitions (`let`), consist of optional functional types together with computational bodies.
- Properties (`property`), statements expressed by a first-order formula specifying requirements to be satisfied in the context of the species.
- Properties together with their proofs (`theorem`).

The general syntax of a species is given as follows:

species *species_name* [(*species_parameters*)]
= [**inherit** *species_names*] ;
[**representation** = *rep_type*;]
signature *function_name* : *function_type*;
[**local / logical**] **let** [**rec**] *function_name* = *function_body*;
property *property_name*:*prop_specification*;
theorem *theorem_name* : *theorem_specification*
proof = *theorem_proof* ;
end ;;

As we mentioned above, species have object-oriented flavors [9]. We can create a species from scratch or from other species using (multiple) inheritance. Through inheritance, it is possible to associate a definition of function to a signature or a proof to a property. Similarly, it is possible to redefine a method even if it is already used by an existing method. The late-binding mechanism ensures that the selected method is always the latest defined along the inheritance tree.

A species is said to be `complete` if all declarations have received definitions and all properties have received proofs. The representations of complete species are encapsulated through species `interfaces`. The interface of a complete species is the list of its function types and its logical statements. It corresponds to the end user point of view, who needs only to know which functions he can use, and which properties these functions have, but doesn't care about the details of the implementation. When complete, a species can be implemented through the creation of collections. A collection can hence be seen as an abstract data type, only usable through the methods of its interface. The following example presents the widely used species `Setoid`. It models any non-empty set with an equivalence relation on the equality (=) method:

```
species Setoid = inherit Basic_object;
 signature equal: Self->Self->bool;
 signature element: Self;
 property equal_reflexive:
    all x: Self, equal (x, x) ;
 property equal_symmetric: all x y: Self,
             equal(x, y) -> equal (y, x) ;
 property equal_transitive: all x y z: Self,
   equal(x, y)-> equal(y, z) -> equal(x, z);
 ...
 end;;
```

2.2. UML/OCL concepts

In this paper we focus on UML class diagrams. In order to provide a formal framework for the transformation of UML model into FoCaLiZe specifications, we propose an abstract syntax for the UML considered constructs. The vast majority of this syntax is extracted from the UML Metamodel [1].

A class is characterized by its name and contains attributes and operations. Each attribute (instance or state variable) of a class has a name and a specification which is either a primitive type or another class of the model. Specifications may describe multiple values of a type. Each operation (method) has a name, parameters and a specification. Parameters are pairs formed by names and their types. A class diagram also contains binary relations between classes which makes it possible to describe generalization (inheritance), dependencies, parameterization and binding relationship. Binary associations are also considered. An association between two classes has a name, roles, navigability features and multiplicities.

The Object Constraint Language (OCL) [2] allows us to enhance UML graphical notations with logical formulas. It is the current standard of the Object Management Group (OMG). An OCL constraint may be attached to any UML element (its context) to further specify it. We consider

class invariants and operation pre and post-conditions.

ocl_constraint ::= *context* {*ocl-stereotype* **:** *OCL-Exp*}$^+$
context ::= **context** {*classContext* | *opContext*}
ocl-stereotype ::= **inv** | **pre** | **post**

OCL expressions have types which are either primitive types or classes of the model [2]. OCL also defines set oriented types (collections, sets, ordered sets, bags and sequences) in order to describe logical quantification using an object oriented syntax.

3. From UML class diagram to FoCaLiZe

In what follows we will present the translation process from UML into FoCaLiZe. Our UML model describes the following element constructors: *class* and *association*.
For each constructor, a transformation rule is proposed. During the transformation process we will maintain two contexts, Γ_U for UML and Γ_F for FoCaLiZe. Γ_U is defined as follows: For a given class named c_n, $\Gamma_U(c\_n) = (c\_n = V_{c\_n})$, where V_{c_n} is the value of the class. A class value is a pair $(\Gamma_{c\_n}, body_{c\_n})$ composed of the local context of the class and the body of the class. The body of the class is composed of class attributes and class operations. In other words, the body of the class c_n is expressed as follows (the trailing star sign $*$ denotes several occurrences):
$body_{c\_n} = (Attr_{c\_n}, Opr_{c\_n})$, where
$Attr_{c\_n} = \{(att\_n : att\_type)\}^*$, att_n is an attribute name of the class c_n and att_type its type. Similarly $Opr_{c\_n} = \{(op\_n : op\_type)\}^*$ where op_n is an operation name of the class c_n and op_type its parameters and returned types. During the transformation process, the local context $\Gamma_U(c\_n)$ of the class c_n is to be enriched with all class identifiers appearing in the list $\mathcal{P}$ of parameter declarations, the list $\mathcal{H}$ of classes from which the current class inherits, and the list $\mathcal{D}$ of dependencies (see class definition in section 3.1).

In a symmetrical way, the FoCaLiZe typing context, is defined as follows: For a given species named s_n, $\Gamma_F(s\_n) = (s\_n = V_{s\_n})$, where V_{s_n} is the value of the species. A species value is a pair $(\Gamma_{s\_n}\ ,\ body_{s\_n})$ composed of the local context of the species together with its body.
A species body is composed of its representation (Rep), signatures (Sig), function definitions (Let), properties ($Prop$) and proofs ($Proof$). The body of the species s_n is denoted:
$body_{s\_n} = (Rep_{s\_n}, Sig_{s\_n}, Let_{s\_n}, Prop_{s\_n}, Proof_{s\_n})$

For an UML element U we will denote $[\![U]\!]_{\Gamma_U,\Gamma_F}$ its translation into FoCaLiZe. For an identifier $ident$, $upper(ident)$ returns $ident$ with its first character capitalized and $lower(ident)$ returns $ident$ with its first character in lowercase.

$$[\![< class\_def >]\!]_{\Gamma_U,\Gamma_F} = \texttt{species}\ \ s\_n\ \texttt{(}\ [\![\mathcal{P}]\!]_{\Gamma_U,\Gamma_F}\ \texttt{,}\ [\![\mathcal{T}]\!]_{\Gamma_U,\Gamma_F}\ \texttt{,}\ [\![\mathcal{D}]\!]_{\Gamma_U,\Gamma_F}\ \texttt{) =}\ \texttt{inherit Setoid,}\ [\![\mathcal{H}]\!]_{\Gamma_U,\Gamma_F}\ \texttt{;}\ [\![\mathcal{A}]\!]_{\Gamma_U,\Gamma_F}\ \texttt{;}\ [\![\mathcal{O}]\!]_{\Gamma_U,\Gamma_F}\ \texttt{end ;;}$$

Figure 1: Transformation of a class definition

3.1. Classes

The general definition of an UML class is given as follows:

$< class\_def >$ = [**public** | **private** | **protected**] [**final** | **abstract**] [«*class-stereotype*»] **class** c_n ($\mathcal{P}$) **binds** $\mathcal{T}$ **depends** $\mathcal{D}$ **inherits** $\mathcal{H}$ = $\mathcal{A}$; $\mathcal{O}$ **end**

where c_n is the name of the class, $\mathcal{P}$ is a list of parameter declarations, $\mathcal{T}$ is a list of substitutions of formal parameters with actual parameters, $\mathcal{H}$ designates the list of classes from which the current class inherits, $\mathcal{D}$ is a list of dependencies, $\mathcal{A}$ the attribute list of the class and $\mathcal{O}$ its operations (methods). For brevity we only consider the case where $\mathcal{P}$, $\mathcal{T}$, $\mathcal{H}$ and $\mathcal{D}$ are empty and focus on attributes and operations. Thus the class context Γ_{c_n} is empty and its body is simply the pair made of $\mathcal{A}$ and $\mathcal{O}$.

Class definitions are then translated by the rule of Figure 1. The derived species will be obtained after translating $[\![\mathcal{P}]\!]_{\Gamma_U,\Gamma_F}$, $[\![\mathcal{T}]\!]_{\Gamma_U,\Gamma_F}$, $[\![\mathcal{D}]\!]_{\Gamma_U,\Gamma_F}$, $[\![\mathcal{H}]\!]_{\Gamma_U,\Gamma_F}$, $[\![\mathcal{A}]\!]_{(\Gamma_U \oplus \Gamma_{c\_n}),\Gamma_F}$ and $[\![\mathcal{O}]\!]_{(\Gamma_U \oplus \Gamma_{c\_n}),\Gamma_F}$. These rules enable us to enrich progressively the local context of the current class Γ_{c_n}, the local context of the derived species Γ_{s_n} and their bodies.

3.1.1. Instance variables. The set of instance variables characterizes the state of an object. Let $\mathcal{A}$ be the list of attributes of a class named c_n:

$$\mathcal{A} = \left(\begin{array}{l} vis_1\ attrName_1 \textbf{ : } typeExp_1\ [mult_1] \\ \qquad\qquad \textbf{=} default_1\ modif_1 \\ \ldots \\ vis_n\ attrName_n \textbf{ : } typeExp_n\ \textbf{[}mult_n\textbf{]} \\ \qquad\qquad \textbf{=} default_n\ modif_n \end{array} \right)$$

where $vis_i \in \{$ **+** , **-** , **#** , **~** $\}$, (**+** : Public, **-** : Private, **#** : Protected, **~** : Package). $mult_i \in \{1\ ,\ 2\ ...\ \infty\}$(if $mult_i$ is different from 1, the attribute is multivalued).

$$[\![\mathcal{A}]\!]_{\Gamma_U,\Gamma_F} = ([\![\mathcal{A}_1]\!]^{attr}_{\Gamma_U,\Gamma_F,(Attr_{c\_n} \oplus A_1)}, \ldots, [\![\mathcal{A}_n]\!]^{attr}_{\Gamma_U,(Attr_{c\_n} \oplus A_n),\Gamma_F})$$

Each instance variable will be modeled by the signature of its getter function. The transformation rule of the attribute $\mathcal{A}_i$ is the following:

$$[\![\mathcal{A}_i]\!]_{\Gamma_U,\Gamma_F} = \texttt{signature}\ \ get\_AttrName_i\ \texttt{:}\ \texttt{Self ->}\ FocTypeExp_i\ \texttt{;}$$

For complete species (derived from leaf classes), we also generate the representation (carrier type) of the

species s_n (derived from the class c_n). It will be a cartesian product grouping types modeling instance variable types ($FocTypeExp_1, \ldots\ FocTypeExp_n$).

3.1.2. Operations. UML class operations will be translated into signatures of a species. Here, we distinguish several cases according to the operation parameters and stereotype:

Let $\mathcal{O} = op_1, op_2, ...op_n$ be the list of operations, where each $op_i, i : 1..n$ has form:

$$op_i = vis_i\ op\_st_i\ opName_i \left(\begin{array}{l} dir_{i1}\ pName_{i1} : \\ \quad typeExp_{i1}\ [mult_{i1}] \\ \ldots \\ dir_{im}\ pName_{im} : \\ \quad typeExp_{im}\ [mult_{im}] \end{array} \right) : returnType_i\ [mult_i]$$

Visibilities vis_i are similar to those of attributes (see section 3.1.1). The stereotype op_st_i is either `create` or `destroy`. Multiplicities $mult_{ij} \in \{1\ ,\ 2\ ...\ \infty\ \}$ are the same as attribute multiplicities (see section 3.1.1). The parameter direction dir_{ij} is either **in** (by default) or **out**. The $pName_{ij}$ are parameter names of the operation and $typeExp_{ij}$ their types.

The return type is $returnType_i$ and has multiplicity $mult_i$. The corresponding FoCaLiZe functions will be obtained by applying the rule:

$$[\![\mathcal{O}]\!]_{\Gamma_U,\Gamma_F} = [\![op_1]\!]^{op}_{\Gamma_U,\Gamma_F,(Opr_{c\_n} \oplus O_1)} \cdots [\![op_n]\!]^{op}_{\Gamma_U,\Gamma_F,(Opr_{c\_n} \oplus O_n)}.$$

For each op_i, a signature (a function specified with its type) is generated in the derived species. The general transformation is presented as follows:

$$[\![op_i]\!]_{\Gamma_U,\Gamma_F} = \texttt{signature}\ lower(opName_i)\ :\ \texttt{Self -> }\ FocTypeExp_{i1}\ \texttt{->...}\ FocTypeExp_{im}\ \texttt{->}\ returnFocType_i\texttt{;}$$

where the keyword `Self` models the species entities on which the function will be applied, the expressions $FocTypeExp_{i1}\ \ldots FocTypeExp_{im}$ are the transformation of the expressions $typeExp_{i1}\ \ldots typeExp_{im}$ into FoCaLiZe and the $returnFocType_i$ is the corresponding FoCaLiZe type for the operation returned type ($returnType_i$).

In order to illustrate the three preceding rules (classes, instance variables and operations), we present the transformation of the class `Person` in Table 1.

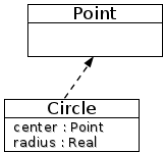
In this example, the representation (carrier type) of the species named `PersonSpecies` is `(string*int)`. It is the cartesian product of types of the attribute `age` and the attribute `name`. The function `newPerson` corresponds to the transformation of the class constructor. Even if it is implicit (not declared) in the original class, we add this function to the derived species.

3.1.3. Inheritances and dependencies. Since the FoCaLiZe specification language supports method redefinition, inheritance, multiple inheritance, encapsulation and late-binding features [7], we transform the inheritance relationship of UML into inheritance relationship of FoCaLiZe (see Figure 1). Let $\mathbb{H}$ be the list of inheritance associated to the class named c_n.
$\mathbb{H} = \mathbb{H}_1, \mathbb{H}_2\ \ldots \mathbb{H}_n$, where each $\mathbb{H}_i = c\_n_i$ and c_n_i, $i = 1..n$ are the class names from which c_n inherits. As we mentioned above, the inheritance list $\mathbb{H}$ of the class c_n is translated into an inheritance list of the species s_n derived from c_n.

An UML dependency is a relationship which indicates that the specification of one class (the client) requires other classes (suppliers).

Let $\mathbb{D}$ be the list of dependencies associated to the class named c_n (the client).
$\mathbb{D} = \mathbb{D}_1, \mathbb{D}_2\ \ldots \mathbb{D}_n$, where each $\mathbb{D}_i = c\_n_i$ and c_n_i, $i = 1..n$ are the supplier class names. In FoCaLiZe, the parameterization of one species with other species has exactly the same semantics of dependency in UML. Hence, we transform UML dependencies into FoCaLiZe parameterizations.

For example, the class `Circle` requires to the class `Point` to define its center. The transformations of the example is the following:




```
species Point =  ... end;;
species Circle (P is Point)=
      representation = P * float;
 ...
end;;
```

3.1.4. Template classes and template binding. Template classes (parameterized classes), may be defined in the same manner as parameterized species [9], [10]. Therefore, we transform a parameterized class into a parameterized species.

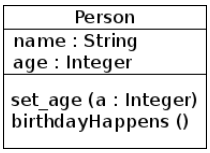
Let $\mathbb{P}$ be the list of parameters of the class named c_n.
$\mathbb{P} = \mathbb{P}_1, \mathbb{P}_2, \ldots \mathbb{P}_n$. Each $\mathbb{P}_i$ has form:
$\mathbb{P}_i = p\_n_i : typeExp_i$, where p_n_i is the parameter name and $typeExp_i$ its type.
These parameters are transformed into parameters of the species s_n derived from the class c_n.

Template bindings specify how the class is constructed by substituting the formal template parameters of the target template classes with actual parameters.

Table 2 presents the transformation of the template `Finite_List`. It has two parameters: `T` of type `Class` and `I` of type `Integer`. As it is described above, the first

Table 1: Example of the class `Person` transformation

UML	FoCaLiZe

	`species PersonSpecies = inherit Setoid;`
	`representation = string * (int) ;`
	`let get_name (p:Self):string = fst(p) ;`
	`let get_age (p:Self):int = snd(p);`
	`signature newPerson:string -> int -> Self;`
	`signature set_age : Self -> int -> Self ;`
	`signature birthdayHappens : Self -> Self;`
	`end;;`

parameter is converted into a parameter of type `Setoid` and the second to a parameter of type `IntCollection` (which models integers in FoCaLiZe). The attribute `data` of the class `Finite_List` is multivalued. It is transformed through the type `list` in FoCaLiZe. Table 2, also presents how the class `Year` (a list of 12 months) is constructed through substitutions of the formal parameters `T:Class` and `I:Integer` of the class `Finite_List` by the actual parameters `Month` and `12` (`T -> Month` and `I -> 12`), where `Month` is the class that models the months of the year.
To transform this binding, we create the species `Year` by inheritance from the species `Finite_List` (derived from the class `Finite_List`) and by substitution of its parameters (`T is Setoid`) and (`i in IntCollection`) by the parameters `T is Month` (the species derived from the class `Month`) and `e = IntCollection!createInt(12)`. More detail about the transformation of UML templates and template binding is presented in our paper [13].

3.2. Associations

A binary association $\mathcal{AS}$ between two classes has the following form:

$\mathcal{AS}$ = **association** ass_n ass_dir c_left $\mathcal{MU}_{left}$, c_right $\mathcal{MU}_{right}$ **end**

where ass_n indicates the association name, ass_dir the navigation direction which takes one of the values: `LeftToRight`, `RightToLeft` or `Both`, c_left and c_right represent the class names of the left and right sides of the association, $\mathcal{MU}_{left}$ is the multiplicity associated to the c_left class and $\mathcal{MU}_{right}$ associated to the c_right class.

Table 3 shows the transformation of the `Review` association between the classes `Person` and `Paper`. The `Person` end carries multiplicity 3 to specify that all instances of the `Paper` class must be reviewed exactly by 3 persons. The `Paper` end has multiplicity * to express that one person may review an arbitrary number of papers.

4. From OCL to FoCaLiZe

To transform OCL expressions we have built an OCL framework library support. In this library, we model OCL primitive types using FoCaLiZe primitive types. For collection types, we have a species named `OCL_Collection(Obj is Setoid)` implementing OCL operations on collections (`isEmpty`, `size ...`). Other kinds of collection (`Set(T)`, `OrderedSet(T)`, `Bag(T)` and `Sequence(T)`) are also described by species which are inherited from `OCL_Collection`.

All OCL constraints (invariants on classes and pre/post-conditions of class operations) are mapped into FoCaLiZe properties (`property` or `theorem`). We have proposed a formal transformation rule for each supported OCL construct. The full description of these rules is beyond the scope of this paper. Table 4 presents the transformation of three OCL constraints specified on the class `Person`.

5. Potential use of the framework

From an UML/OCL model, an abstract FoCaLiZe specification is generated (see Figure 2).

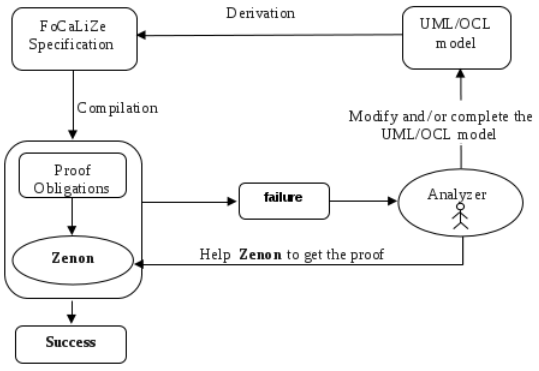


Figure 2: Proof process

When we compile the FoCaLiZe source, several proof obligations are generated. In particular the proof of the derived properties from associations and the derived properties from OCL constraints. When a proof fails, the FoCaLiZe compiler indicates the line of code responsible for the error. The developer analyses the source to determine the reason for the failure. There are two main kinds of errors: either because of inappropriate proof hints given

Table 2: Transformation of a template and template binding

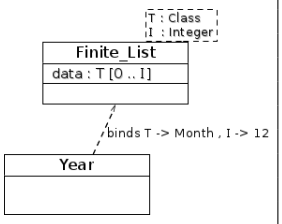
UML	FoCaLiZe
	`species Finite_List (T is Setoid, i in IntCollection)= inherit Setoid ; representation = list(T); ... end;; species Month = inherit Setoid; ... end;; let e = IntCollection!createInt(12);; species Year (T is Month, i in IntCollection)= inherit Finite_List(T, e) ; ... end;;`

Table 3: Transformation of the association `Review`

UML	FoCaLiZe

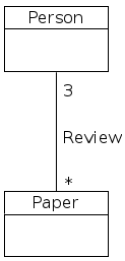
	`species Review (Left is Person , Left_Set is Set(Left), Right is Paper, Right_Set is Set(Right) ) = inherit Association (Range_collection, Left, Left_Set, mult_left, Right, Right_Set, mult_right, Direction_collection, direction ); end;;`
	Where `mult_left`, `mult_right` and `d` are implemented at top level as follows:
	`let direction = Direction_collection!make_direction(Both);;`
	`let mult_left = Range_collection!make_range(Unlimited_Nat!make(I(3)), Unlimited_Nat!make(I(3)));;`
	`let mult_right = Range_collection!make_range(Unlimited_Nat!make(I(0)), Unlimited_Nat!make(Infini));;`
	`Association` is a general species models associations in our library support.

Table 4: Example of OCL constraints transformation

OCL	FoCaLiZe

`context Person inv : (age <= 150) and (age > 0)`
`context Person::set_age(a:Integer) Pre : (a <= 150)and (a > 0) and (a >= age) Post: age = a`
`context Person inv : Person.allInstances -> forAll(p1, p2 | p1 <> p2 implies p1.name <> p2.name)`
	`species Person_constraints = inherit PersonSpecies; property invariant_Person_1 : all x : Self, (get_age(x) <= 150) /\ (get_age(x) > 0);`
	`property pre_post_set_age : all x : Self, all a : int , ( (get_age(x) <= 150) /\ (get_age(x) > 0) /\ (a >= get_age(x) ) ) -> (get_age(set_age(x , a)) = a ) ;`
	`(* "~~" designates the logical negation *) property invariant_Person_2: all p1 p2: Self, ~~(equal(p1, p2)) -> ~~(get_name(p1) = get_name(p2)) end ;;`

by the developer, or because of inconsistencies in the original UML/OCL model. The first reason leads to modify the FoCaLiZe source in order to provide a tractable proof script for Zenon, while the second leads to correct and/or complete the original UML/OCL model.

OCL errors are detected when proving properties derived from OCL constraints, since the proof of such properties may use a conflicting properties (derived from other OCL constraints) as hypothesis or axioms. Also, thanks to the direct transformation in our approach (FoCaLiZe and UML share the same features semantics), UML errors such as the violation of UML template semantics, inheritance semantics, the dependency semantics or the specification of incorrect values or types are automatically

detected when FoCaLiZe compiler checks the derived source. In the near future, our approach will be enhanced by the transformation of behavioral diagrams such as UML state diagram. The transition from one state to another in UML state diagram defines a property that can be contradictory with the property derived from the pre and post-condition of the triggering event. This leads to detect an inconsistency, either in the UML statechart or in the OCL pre and post-condition.

In order to clarify the proof framework of figure 2, we present the proof of the class `Person` invariant presented in table 4.

After derivation of the species `PersonSpecies` (from the class `Person`), we generate the species `Person_constraints` which inherits from `PersonSpecies` and contains theorems and properties derived from the OCL constraints (for brevity, we only focus on the property `invariant_Person_2`):

```
species Person_constraints =
        inherit PersonSpecies;
let get_name(p:Self):string = fst(p);

let equal (x: Self, y : Self): bool =
   (get_name(x) = get_name(y)) ;
...
 theorem  invariant_Person_2 :
                   all p1 p2 : Self,
 ~~(equal(p1, p2)) ->
 ~~(get_name(p1) = get_name(p2))
 proof = by definition of equal ;
 end ;;
```

The symbols `~~` present the logical negation in FoCaLiZe. In this example, we ask Zenon to find a proof using the hint "`by definition of equal`". Finally, the compilation of the above FoCaLiZe source ensures the correctness of the specification. If no error has occurred, this means that the compilation, code generation and Coq verification were successful.

6. Related works

First, in the transformation from UML/OCL into B language based tools such as UML2B [14], UML-B [15], [16] and [17], the inheritance mechanism is indirectly considered through the clause `Uses`. The clause `Uses` of B language (as its name indicates) corresponds to the dependency feature of UML, which leads to add supplementary invariants in the abstract machines derived from the sub-classes [15] to preserve the inheritance semantics of UML. In B language, a formal parameter of an abstract machine may only be of two kinds: scalar parameters or set parameters [3]. The formal parameters of an abstract machine can not be an implementation or an other abstract machine of the model. This limitation makes it difficult to model the UML template with B constructs.

Second, the works interested in using high order logic (HOL) such as the HOL-OCL tool [18] or those based on rewriting logic and runtime checking tools such as the use of the Maude system in [19]. Also for this category of tools, the UML multiple inheritance, late binding, template and template binding are manufactured and not perfectly supported. For example, in [19] the inheritance feature is indirectly transformed through the definition of predicates `isSubclass(A,B)`, which establish that a class A is a subclass of another B. We can do the same remark for the HOL-OCL tool [18],

The use of older works such as SMV model checker in [20] and PROMELA in [21] is limited to the verification of isolated UML statecharts. We also find a transformation from UML into LOTOS in [22], however the concept of object references and dynamic links between objects are not supported.

In works such as [23] and [24], an UML class is modeled by a signature (a set of atoms) of Alloy system. The only supported architecture feature is the `extends` (simple inheritance in UML) relationship between signatures.

Finally, works such as [25] and rCOS [26] are interested in providing formal semantics for UML. However, they do not integrate proof tools.

While inheritance is indirectly considered, the multiple inheritance, UML templates and template bindings are ignored in all the aforementioned works. Using FoCaLiZe, we directly support the inheritance and multiple inheritance features of UML through the clause `inherit` which enables multiple inheritance and late-binding mechanism.

Through our approach, UML templates correspond perfectly to parameterized species and UML template bindings are similar to parameter substitutions in FoCaLiZe. In fact, FoCaLiZe enables the use of a species as a parameter of another species, even at the specification level. Later on, the collection and late-binding mechanisms ensure that all methods appearing in a species (used as formal parameter) are indeed implemented and all properties are proved.

7. Conclusion and perspectives

In this paper, we have presented a process to transform an UML/OCL model composed of class diagrams and OCL constraints into a FoCaLiZe specification. A FoCaLiZe species is derived from each UML class, where the representation of the derived species is a cartesian product that represents the instance variables of the class. Class operations are converted into methods of the derived species. The OCL constraints correspond to FoCaLiZe properties. A binary association between two classes is

translated into a species parameterized by the two derived species, the corresponding multiplicities and by the direction of the association navigation.

To implement the presented approach, we propose to use the XMI technology (XML Metadata Interchange) through the UML2 Eclipse plug-in. We parse the XMI document to translate it into our UML syntax (using an XSLT stylesheet), so that it is possible to apply the proposed transformation rules. These later ensure the correctness of the transformation.

The presented work provides a direct transformation of most of UML specification features. In addition to classes associations, it supports encapsulation, multiple inheritance, late-binding, templates, template bindings and dependency which permit to derive a formal specification expressed through species hierarchy that keeps the same incremental modeling provided by UML. To our knowledge, there is no formal tool that supports the above features as they are taken care of in the FoCaLiZe environment.

In the present work we have not taken into account the dynamic aspect of UML models. In the next step, we plan to add to our UML model, diagrams describing the behavioral aspect of UML classes such as statechart diagrams.

References

[1] OMG., "UML : Superstructure, version 2.4," Jan. 2011, available at: http://www.omg.org/spec/UML/2.4/Infrastructure.

[2] OMG, "OCL : Object constraint language 2.3.1," Jan. 2012, available at: http://www.omg.org/spec/OCL.

[3] J. Abrial., "The B book." 1996, cambridge University Press.

[4] D. Jackson, *Software Abstractions: Logic, Language, and Anlysis*. The MIT Press, 2011.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All about Maude a high performance logical framework: how to specify, program and verify systems in rewriting logic*. Springer-Verlag, 2007.

[6] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002., vol. 2283.

[7] F. D. Team, "FoCaLiZe : Tutorial and reference manual, version 0.8.0," *CNAM/INRIA/LIP6*, 2012, available at: http://focalize.inria.fr.

[8] P. Ayrault, T. Hardin, and F. Pessaux, "Development life-cycle of critical software under Focal," *Electronic Notes in Theoretical Computer Science*, vol. 243, pp. 15–31., 2009.

[9] S. Fechter, "Sémantique des traits orientés objet de focal," Ph.D. dissertation, Université PARIS 6, 2005.

[10] D. Delahaye, J.-F. Étienne, and V. V. Donzeau-Gouge, "Producing UML models from focal specifications: an application to airport security regulations," in *Theoretical Aspects of Software Engineering, 2008. TASE'08. 2nd IFIP/IEEE International Symposium on*. IEEE, 2008, pp. 121–124.

[11] D. Doligez, "The Zenon tool," software and documentations freely available at http://focal.inria.fr/zenon/.

[12] coq., "The coq proof assistant, Tutorial and reference manual," *INRIA – LIP – LRI – LIX – PPS*, 2010, distribution available at: http://coq.inria.fr/.

[13] M. Abbas, C.-B. Ben-Yelles, and R. Rioboo, "Modeling UML Template Classes with FoCaLiZe," in *Integrated Formal Methods*. Springer, 2014, pp. 87–102.

[14] L. Hazem, N. Levy, and R. Marcano-Kamenoff, "UML2B : Un outil pour la génération de modèles formels," *AFDL*, 2004.

[15] C. Snook and M. Butler, "UML-B : Formal modeling and design aided by UML," *ACM Transactions on Software Engineering and Methodology*, no. 15, pp. 92–122, 2006.

[16] N. Truong and S. J., "Verification of UML model elements using B," *Information Science and Engineering*, no. 22, pp. 357–373, 2006.

[17] J. GU, M. CHEN, and X. ZHOU, "Formal language B and UML/OCL comparison," *Computer Knowledge and Technology*, vol. 34, p. 044., 2009.

[18] A. Brucker and B. Wolff, *The HOL-OCL tool*, 2007, http://www.brucker.ch/.

[19] F. Durán, M. Gogolla, and M. Roldán, "Tracing properties of UML and OCL models with maude," *arXiv preprint arXiv:1107.0068*, 2011.

[20] G. Kwon, "Rewrite rules and operational semantics for model checking UML statecharts," in *UML2000—The Unified Modeling Language*. Springer, 2000, pp. 528–540.

[21] J. Lilius and I. P. Paltor, "vUML: A tool for verifying UML models," in *Automated Software Engineering, 1999. 14th IEEE International Conference on.* IEEE, 1999, pp. 255–258.

[22] P. J. Carreira and M. E. Costa, "Automatically verifying an object-oriented specification of the steam-boiler system," *Science of Computer Programming*, vol. 46, no. 3, pp. 197–217., 2003.

[23] A. Cunha, A. Garis, and D. Riesco, "Translating between Alloy specifications and UML class diagrams annotated with OCL," *Software & Systems Modeling*, pp. 1–21., 2013.

[24] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "UML2Alloy: A challenging model transformation," in *Model Driven Engineering Languages and Systems*. Springer, 2007, pp. 436–450.

[25] M. Broy, M. Crane, J. Dingel, A. Hartman, B. Rumpe, and B. Selic, "Formal semantics for uml," *Models in Software Engineering*, pp. 318–323, 2007.

[26] J. Yang, "A framework for formalizing uml models with formal language rCOS," in *Frontier of Computer Science and Technology, 2009. FCST'09. Fourth International Conference on*. IEEE, 2009, pp. 408–416.

[27] C. Dubois, T. Hardin, and V. Donzeau-Gouge, "Building certified components within FOCAL," *Trends in Functional Programming*, vol. 5, pp. 33–48, 2006.