N° D'ORDRE:

N° DE SERIE:

Algerian Democratic and Popular Republic
Ministry of Higher Education and Scientific Research

UNIVERSITY OF ECHAHID HAMMA LAKHDAR EL OUED
FACULTY OF EXACT SCIENCES
Dissertation

Academic Master

Domain: Mathematics and computer science
Field: Computer Science
Specialty: Distributed system and artificial intelligence

Presented by:

- GHERAISSA Tarek - LEMMOUCHI Nacer-Eddine

Title

Transformation of OCL Constraints to FoCaLiZe Specifications

Session of Jun 2016

Under the supervision of:

MESSAOUD Abbas	Univ. El-OUED	Reporter
OTHMANI Samir	Univ. El-OUED	President
GUIA Sana Sahar	Univ. El-OUED	Examiner

University year 2015 – 2016

Acknowledges

We would like to express our sincere and faithful thanks to professor **Messaoud Abbas** for his ideal supervision, sage advices and continuous encouragement. We would like to thank him deeply for his great and valuable help without which this work would not be possible. We would like to thank those who have devoted their time to help us. As we would like to present our respect to **MENACEUR Khadija** and **CHEKIMA Mohammed** for all the helps and supports their offered us. Also we would like to thank our family and our friends wishing a good luck to them in their life.

Abstract

FoCaLiZe is a development environment based on a formal approach, which integrates an automated theorem prover (Zenon) and a proof checker (Coq). On the other hand, the UML graphical language is widely used to model systems in a synthetic and intuitive way, but lacks formal basis. In this master thesis we studied the transformation rules of OCL constraints into FoCaLiZe specifications in the context of MDE (Model Driven Engineering). Then, we propose an implementation of the transformation rules using XSLT language. The proposed implementation directly supports the main OCL constraints (invariant and pre/post-condition). We illustrate our implementation with concrete examples.

Keywords: UML, OCL, FoCaLiZe, Transformation, XSLT and MDE.

.

Résumé

FoCaLiZe est un environnement de développement basé sur une approche formelle, qui intègre un prouver automatique (ZENON) et un outil d'aide à la preuve (Coq). D'autre part, le langage graphique UML est largement utilisé pour modéliser les systèmes de manière synthétique et intuitive, mais il manque les bases formelles. Dans ce mémoire de master, nous avons étudié les règles de transformation de contraints OCL en spécifications FoCaLiZe dans le contexte de MDE (Model Driven Engineering), puis nous proposons une implémentation des règles de transformation en utilisant le langage XSLT. L'implémentation proposée supporte directement les principaux contraints OCL (invariants et pre/post-conditions). Nous illustrons notre implémentation par des exemples concrets.

Mots-clés: UML, OCL, FoCaLiZe, Transformation, MDE, XSLT.

ملخص

فوكالايز(FoCaLiZe) هي بيئة تطوير ترتكز على المنهج النظامي، والذي يشتمل على دمج المبرهن الآلي (Zenon) والمدقق للبراهين (Coq). و من ناحية أخرى، لغة UML هي لغة بيانية يتم استخدامها على نطاق واسع لمنذجة النظم بطريقة تركيبية و بديهية، لكنها تفتقر إلى الإثباتات والبراهين اللازمة لتأكد من صحة النماذج. في هذه المذكرة درسنا قواعد التحويل قيود (ضوابط) OCL إلى فوكالايز في سياق MDE (Model Driven Engineering) ، حيث إقترحنا تطبيق قواعد التحويل بستخدام لغة XSLT .التنفيذ المقترح يدعم بشكل رئيسي قيود OCL (invariants وpre/post-conditions) . وقد قمنا بتوضيح عملية التنفيذ قواعد التحويل بأمثلة ملموسة.

الكلمات المفتاحية: UML, OCL, FoCaLiZe, Transformation, XSLT, MDE

Table of Content

List of Figures

List of Tables

General Introduction

The Unified Modelling 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 and simulation to code generation and transformation. In order to further specify UML models, the Object Constraint Language (OCL) [2] has been introduced. OCL is the current standard of the Object Management Group (OMG[1]) and is defined as part of UML standard. An OCL constraint is a precise statement which may be attached to any UML element. Because UML semantics is still informal and allows ambiguities, using UML and OCL, we can only describe systems and specify constraints upon them. No formal proof is available in a UML/OCL context to check 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 check the properties of an UML/OCL model. Among these approaches, those interested in the translation of UML/OCL models into formal methods. This transformation allows us first to obtain an abstract formal specification in the target language and later on, it is possible to verify and prove the original OCL constraints using proof techniques available in the formal language.

In this master thesis, we present a transformation of OCL constraints into the FoCaLiZe environment [3] following a compiling approach (by translation, as in [4]). The choice of FoCaLiZe does not solely rely on its formal aspects. FoCaLiZe supports most of the requirements mandated by standards upon the assessment of a software development life cycle [5]. In particular, most of the UML design features can seamlessly be represented in FoCaLiZe [6, 7]. It also supports all necessary logical constructors which enables a natural transformation of OCL constraints into FoCaLiZe. Our choice is motivated by the three following arguments:

- First, FoCaLiZe supports most of UML/OCL conceptual and architectural features such as encapsulation, inheritance (generalization/specialization) and multiple inheritance, function redefinition, late-binding, dependency, UML template (parametrized classes), template binding and OCL constraints(invariant and pre/post-conditions). Because the

[1] OMG: Object management group http://www.omg.org/

most of this features are presented in FoCaLiZe through its own constructs without need of additional structures or invariants, we may keep the same development logic. Most of these features are not seamlessly considered in the other works, usually with semantics loss of the original model and with deviation from the incremental intuition development provided by UML. While the inheritance feature of UML is indirectly supported by some formal methods, the transformation of late-binding, template and template binding are rarely considered.

- The second motivation of FoCaLiZe is hidden in the functional paradigm of its language.
- Finally, the use of the FoCaLiZe environment is also motivated by both the power of its automated theorem prover Zenon [8] and its proof checker Coq [9].

This document is organized as follows: Chapter 1 presents FoCaLiZe concepts, chapter 2 presents the UML/OCL concepts, chapter 3 presents a comparison with related works, and chapter 4 explains our transformation rules of OCL constraints into FoCaLiZe. Chapter 5 presents our implementation process.

Chapter 1: FoCaLiZe

FoCaLiZe

The FoCaLiZe project (initially Foc, then Focal Project) began in 1997 under the leadership of T.Hardin and R.Rioboo [10]. It is an environment in which it is possible to build step-by-step applications, starting from abstract specifications, and going to concrete implementations by successive stages of refinement. It also allows to state properties and to prove that the programs meet their specifications. The compiler produces OCaml code for execution, Coq code for formal certification, but also XML for documentation. Zenon [8] is the automatic proofer tool to first order, allows automates some of the proof. The species and its components are restricted from.

Our presentation of FoCaLiZe is inspired from [11, 13].

1.1. Species

In FoCaLiZe, the main entity of a development is the species, which corresponds to the highest level of abstraction in a specification. Species are the nodes of the hierarchy of structures that make up a development. A species can be seen as a set of "things", called methods, related to the same concept. Like in most modular design systems (i.e. objected oriented, algebraic abstract types), the idea is to group a data structure with the operations that operate on it. The next figure shows the general structure of FoCaLiZe developments.

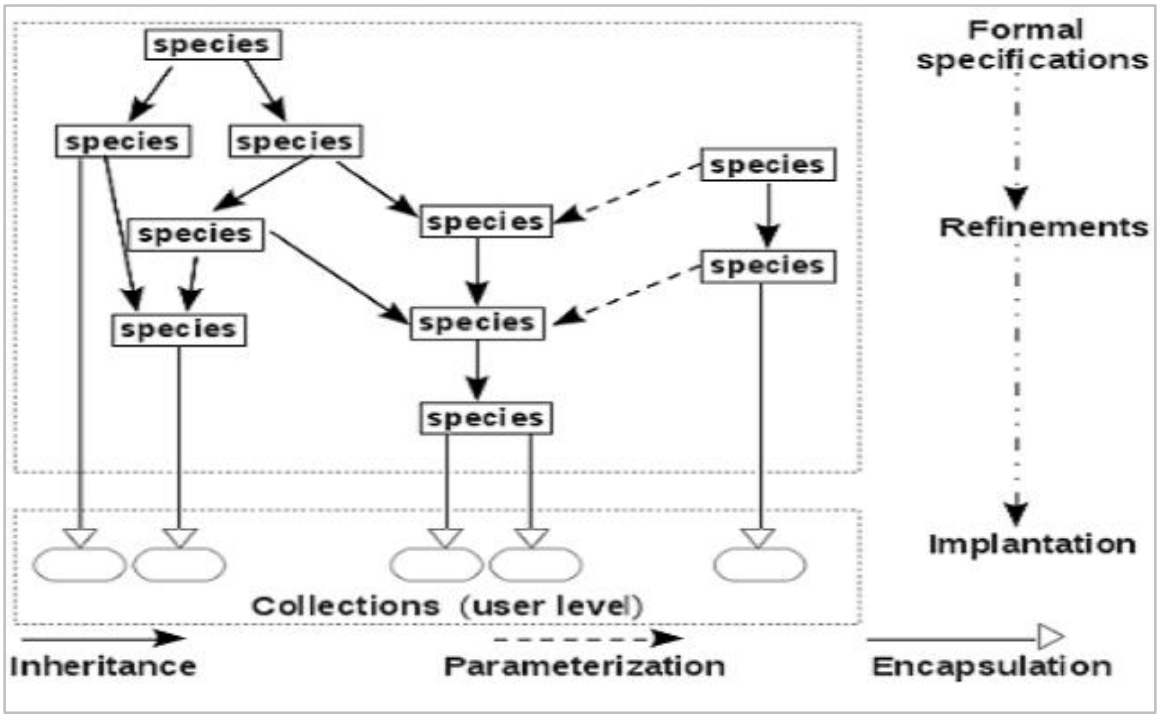


Figure 1.1: FoCaLiZe development

The general syntax of a species is the following:

species *species_name[(P1 is species_name1, p2 is species_name2,…)]=* ***[inherit*** *species_names****;***	
representation *= rep_type****;]***	***(*representation *)***
signature *function name : function type****;***	***(* declaration *)***
[local/logical] let [rec] function_name = function_body;	***(*definition *)***
property *property_name : property_specification****;***	***(* property *)***
theorem *theorem_name : theorem_specification*	***(* theorem *)***
proof *= theorem_proof* ***;***	
end ;;	

Table 1. 1: The general syntax of a species

Where **species_name** is simply a given name, **rep_type** and **function_type** are FoCaLiZe type expressions, **function_body** a function code, **property_specification** and **theorem_specification** are logical formulas. **Theorem_proof** is a proof written in FoCaLiZe proof language to be ultimately checked by the Coq proof assistant. The keyword **local** indicates that the function is only to use inside the species and the keyword **logical** introduces a parameterized logical expression, which can be applied to effective arguments to obtain a logical proposition. Recursive definitions are also available through the let **rec** keyword. Statements, definitions and proofs can freely use names of other methods of the species (denoted by **Self!m** or shorter **m** if there is no ambiguity in the scope).

Species can be created from scratch or by multiple **inheritance (**using the keyword **inherit** from other species**)**, and it can be parametrized by other species **(**using the keyword **is**).

Species names are always capitalized. As any toplevel-definition, a species ends with a double semi character ("**;;**"). In a species there are several kinds of methods.

1.1.1. Representation

The carrier type, called **representation**, which is the type of the entities that are manipulated by the functions of the species, the representation can be either abstract or concrete. The species representation types, denoted by keyword **Self** inside the species and by the name of their species outside of them. And we note that each species has a unique method representation.

Example: To start with our example, we define the species point that models the points of the plan. The representation of this kind is expressed by a couple of Real. The first represents the horizontal axis and the second refers to the ordinate of a point:

```
species Point =
representation = float * float ;
…
end ;;
```

1.1.2. Functions

The functions denote the operations allowed on the entities of the species. The functions can be either **definitions** (when a body is provided) specified by the keyword "**let**" or **declarations** (when only a type is given) specified by the keyword "**signature**".

- **Signatures**: They introduce names of constants and functions, uniquely providing their type as a type expression. A signature begins with the keyword signature followed by the introduced name and by a type expression.

  We complete our species Point by signing the deplacer_point constructor function (to move a point from one position to another) and the binary function equal (to decide the equality between two points) as follows:

```
species Point =
representation = float * float;
signature deplacer_point : Self -> float-> float -> Self;
signature egal : Self -> Self ->    Bool;
...
end;;
```

  The keyword Self (used in the types of functions and equal deplacer_point) denotes an entity of the species being specified, although the representation of a species is not defined (abstract).Type Self -> float-> float -> Self the function of moving the point specify a function set by an entity of the species Point (the first Self) and two Real, and returning a new entity of the species (the latter Self)

- **Let:** function definitions are made of the keyword let, followed by a name, an optional type, and an expression. They serve to introduce constants or functions. Mutually recursive definitions are introduced by **let rec**. Hence, definitions serve for implementation purposes. Continuing with our point here, the functions **get_x** (returns the abscissa of a point) and **get_y** (returns the ordinate of a point) are defined using **fst** functions (returning the first component of a pair) and **snd** (returning the second component of a pair) of the basics library FoCaLiZe. The function **different** (deciding the inequality of two points) is based on the **egal** signature (given above):

```
species Point =
...
 let get_x(p:Self) = fst(p);
 let get_y(p:Self) = snd(p);
 let different(a, b) = ~~(egal(a, b));
...
end;;
```

1.1.3. Properties

The properties that must be verified by any further implementation of the species, the properties can be either simply **declared properties** (when only the proposition is given) or **theorems** (when a proof is also provided).

1.1.3.1. Declared properties

They are first order properties statement which are composed of the keyword property followed by a name and by a first order formula. They serve to express requirements. Formula can use methods names known within the species context, especially to state requirements on theorems. These methods. Proof of properties are in fact delayed. Like for signatures, properties can be used to prove other properties.

To define a structure respecting an equivalence relation on equal operation of the species point, we introduce the following three properties:

```
species Point =
property egal_reflexive : all p:Self, egal(p, p);
property egal_symetrique : all p1 p2: Self, egal(p1, p2) -> egal(p2, p1) ;
property egal_transitive : all p1 p2 p3: Self, egal(p1, p2) -> egal(p2, p3)-> egal(p1, p3);
```

```
...
end;;
```

1.1.3.2. Theorems

Theorems (**theorem**) made of a name, a statement and a proof are in fact properties packed with the formal proof that their statement holds in the context of the species. As said above, the proof of a theorem may use properties available in the context of the species, despite these former are not yet proved.

FoCaLiZe has three modes of proof of theorems:

1. Either using the keyword **assumed** to admit a theorem without giving his evidence (axiom).
2. Either by manually inserting a script of proof in Coq.
3. Either using the language of proof FoCaLiZe (FPL, FoCaLiZe Proof Language) to write a proof that will be directly discharged by **Zenon** (the theorems of automatic prover FoCaLiZe).

A theorem may refer to any function or property rights in the context of the current case, as it can be used to specify other properties of the same species or its heirs.

Using the FPL language, the theorem proof egal_est_non_differen ((a = b) ⇒ ¬ (6 = b)) is accepted by ASSUMED as follows:

```
species Point =
...
theorem egal_est_non_different : all p1 p2:Self, egal(p1, p2) -> ~(different(p1, p2))
proof = assumed;
...
end;;
```

One important restriction on the type of the methods is that it cannot be polymorphic. However, FoCaLiZe provides another mechanism to circumvent this restriction, the parametrization as explained further.

1.2. Collections

A collection is a kind of "grey box", built from a *complete* species by abstraction of the representation. A collection has exactly the same sequence of methods than the complete species underlying it, apart the representation which is hidden.

Note that creating a collection from it is the only way to turn methods of a complete species into executable code. This point is emphasized by the syntax:

collection *name-collection* = ***implement*** *name-species ;* ***end****;;*

The interface of a collection is the one of the complete species it implements. The interface I1 of a collection C1 is *compatible* with an interface I2 if I1 contains all the components of I2. Thus, implementing a complete species creates a collection, which is a kind of abstract data-type. This especially means that entities of the collection cannot be directly created or manipulated as their type is not accessible. So they can only be manipulated by the methods of the *implemented* species.

Example:

```
species Full =
rep = int ; let create_random in Self = random_foc#random_int (42) ;
let double (x in Self) = x + x ;  let print (x in Self) ;  end ;;

collection MyFull_Instance =   implement Full;   end ;;

let v = Full.create_random ;;

Full.print (v) ;;
let dv = Full.double (v) ;;
Full.print (dv) ;;
```

In this example, we define a complete species Full. Then, we create the collection MyFull Instance. And we use methods of this collection to create entities of this collection. We print the result of the evaluation of the top-level definitions of v and dv.

Note that two collections created from a same species are not type-compatible since their representation is abstracted making impossible to ensure a type equivalence.

As a conclusion, collections are the only way to get something that can be executed since they are the terminal items of a FoCaLiZe development hierarchy. Since they are "terminal", this also means that no method can be added to a collection. Moreover, a collection may not be used to create a new species by inheritance (as explained in the next section).

1.3. Species Relationships

One species can be created from scratch as can be created through inheritance from other species or through parameterization by other species. In what follows we will detail the two possible relationships within a species (inheritance and parametrization).

1.3.1. Inheritance and late-binding

We can create a new species by inheritance of an already defined one. We can make this new species more "complex" by adding new operations and properties, or we can make it more concrete by providing definitions to signatures and proofs to properties, without adding new features. A species can inherit the declarations and definitions of one or several already defined species and is free to define or redefine an inherited method as long as such (re)definition does not change the type of the method.

When building by multiple inheritance some signatures can be replaced by functions and properties by theorems. It is also possible to associate a definition of function to a signature or a proof to a property. In the same order, it is possible to redefine a method even if it is already used by an existing method. All these features are relevant of a mechanism known as **late-binding**.

Table 1.2 shows the species **ColoredPoint** that inherits the species **Point** in order to manipulate colored (graphical) points.

```
type color = | Red | Green | Blue ;;
species ColoredPoint = inherit Point;
representation = (int * int) * color;
let getColor(p:Self):color = snd(p);
let newColoredPoint(x:int, y:int, c:color):Self = ((x, y), c);
let getX(p) = fst(fst(p));
let getY(p) = snd(fst(p));
let move(p, dx, dy) = newColoredPoint(getX(p) + dx, getY(p) + dy, getColor(p));
let printPoint (p:Self):String =
```

```
let printColor (c:color) = match c with
| Red -> "Red"
| Green -> "Green"
| Blue -> "Blue"
in ( " X = " ^ String_of_int(getX(p)) ^ " Y = " ^ String_of_int(getY(p)) ^ " COLOR = " ^ printColor(getColor(p))
);
proof of distanceSpecification = by definition of distance;
end;;
```

Table 1. 2: Example of inheritance

The species **ColoredPoint** inherits all methods and logical properties of the species Point, plus the following enrichment:

- It defines its representation type by the expression **(int * int)** * **color** which models the type of the **x-coordinate** (**int**), the type of the **y-coordinate** (**int**) and the type of the **color** (**color**) of a point. The type **color** is defined as sum type in the example.
- It defines the methods: getX, getY and move that were only declared in the species Point.
- It introduces the method **getColor** (which does not exist in the species Point) to get the **color** of a point, the method **newColoredPoint** allows us to create a new instance (as the new method in the Object Oriented Programming) and the method **printPoint** to print the coordinates and the **color** of a given point.
- It provides the proof of the property **distanceSpecification**. Proofs of properties and theorems will be detailed at the end of the current section.

1.3.2. Parametrization

This section describes a first mechanism to incrementally build new species from existing ones the parametrization.

1.3.2.1. Collection parameters

A collection parameter is a species (**supplier**) used by another species (the client) as abstract type, through its interface. A collection of parameter is introduced by a name (**parameter_name**) and an interface designated by the name of a species (**supplier_species_name**), using the following syntax:

species *client_species_name* (*parameter_name* is *supplier_species_name*, . . .) . . . **end**;;

The parameter name (**parameter_name**) is used by the client case as a variable, and to call the methods of the species **supplier**. In geometry, a circle is defined by its center (one point) and diameter. A species Circle (modeling circles) can be created using the species **PointAbstrait** collection as a parameter:

```
species Cercle (P is PointAbstrait) =
representation = P * float ;
let get_centre(c:Self):P = fst(c);
let get_rayon(c:Self):float = snd(c);
let creeCercle(centre:P, rayon:float):Self = (centre, rayon);
let appartient(p:P, c:Self):   Bool = (P!distance(p, get_centre(c)) = get_rayon(c));
theorem appartient_spec : all c:Self, all p:P, appartient(p, c) <-> (P!distance(p, get_centre(c)) = get_rayon(c))
proof = assumed ;
end;;
```

To create a collection from a species set by collections of parameters, each parameter will be substituted with the name of an actual collection of the model. This is why they are called collections settings. For example, we can create the following two collections from the entire species Circle:

collection *CercleCollection1 =* ***implement*** *Cercle(PointConcretCollection);* ***end;;***

collection *CercleCollection2 =* ***implement*** *Cercle(PointColoreCollection);* ***end;;***

To generate the collection **CircleCollection1**, we substituted the case of setting the **Circle** by **PointConcretCollection** collection, created from the species **PointConcret**. For the second collection, the setting of the species Circle is substituted by **PointColoreCollection** collection.

1.3.2.2. Entity parameters

There is a second kind of parameter: the entity-parameter, is phrase the entity of an existing species (species supplier), used by the client species to serve as actual value in the development of its own methods.

A parameter entity is introduced by the name of an entity (entity_name) and the name of a collection (collection_name), using the syntax:

species *client_species_name* (*entity_name* in *collection_name*, . . .) . . . ***end***;;

The **collection_name** collection can serve as such in this case, as it can be used to invoke the functions of the full species underlying.

In this example geometry, the origin of an orthogonal coordinate is a particular point (value) which may be an entity of the data collection **PointConcretCollection** above. Thus, a species that models **RepereOrthogonal** orthogonal benchmarks uses a parameter entity of **PointConcretCollection** collection as follows:

```
species RepereOrthogonal (origine in PointConcretCollection) =
...
end;;
let o = PointConcretCollection!creerPointConcret(0.0, 0.0);;
collection RepereOrthogonalZero = implement RepereOrthogonal(o); end;;
```

To create a collection **RepereOrthogonalZero** from here Orthogonal mark, we substituted the original parameter entity by entity o = (0.0, 0.0) of the **PointConcretCollection** collection.

FoCaLiZe does not allow the specification of a parameter of primitive type of entity (int, String, Bool...). This means that primitive types must be integrated into the collections to be used as parameters of entities.

1.4. Proofs and compilation

Conduct a FoCaLiZe development requires to prove that the set properties are valid. It is necessary to demonstrate them. To do this, an automatic proof tool to first order (based on the methods of tables), called Zenon, helps the user to achieve the proof. The language for interacting with Zenon is a declarative proof language. To prove a property it is also possible to directly give a Coq proof [10]**.**

This option is useful when the proof uses some logical aspects that Zenon can't handle. Coq proofs are directly inserted into the Coq specification file generated by the compiler.

A proof is either a leaf proof or a compound proof. A leaf proof (introduced with the **by** or **conclude** keywords) invokes Zenon with the assumptions being the given facts and the goal being the goal of the proof itself (i.e. the statement that is proved by this leaf proof). The **conclude** keyword is used to invoke Zenon without assumptions.

A compound proof is a sequence of steps that ends with a **qed** step. The goal of each step is stated in the step itself, except for the **qed** step, which has the same goal as the enclosing proof.[13] Proof steps:

Proof_step ::= proof_ bullet statement proof

A proof step starts with a proof bullet, which gives its level of nesting. The top level of a proof is **0**. In a compound proof, the steps are at level one plus the level of the proof itself. As an example of a proof, Table 1.3 shows the proof of the theorem **belongs_specification** of the species **Circle**.

```
Species Circle (P is Point) =
representation = P * float ;
let newCircle(centre:P, rayon:float):Self = (centre, rayon);
let getCenter(c:Self):P = fst(c);
let getRadius(c:Self):float = snd(c);
let belongs(p:P, c:Self):   Bool =(P!distance(p, getCenter(c)) = getRadius(c));
theorem belongs_specification : all c:Self, all p:P,belongs(p, c) <->(P!distance(p, getCenter(c)) = getRadius(c))
proof =
<1>1 assume c : Self, p : P,prove belongs(p, c) <-> (P!distance(p, getCenter(c))= getRadius(c))
<2>1 hypothesis h1 : belongs(p, c), prove belongs(p, c) -> (P!distance(p, getCenter(c))= getRadius(c))
 by hypothesis h1 definition of belongs
<2>2 hypothesis h2 : (P!distance(p, getCenter(c)) = getRadius(c)),
prove (P!distance(p, getCenter(c)) = getRadius(c))-> belongs(p, c)
 by hypothesis h2 definition of belongs property basics#beq_symm
<2>3 qed by step <2>1, <2>2
<1>2 conclude ;
end;;
```

Table 1.3: Example of a Proof

In Table 1.3, the steps **<1>1** and **<1>2** are at **level 1** and form the compound proof of the top-level theorem. Step **<1>1** also has a compound proof (whose goal is belongs (p, c) <-> (P!distance(p, getCenter(c)) = getRadius(c))), made of steps **<2>1** and **<2>2**. These latter are at **level 2** (one more than the level of their enclosing step). After the proof bullet comes the statement of the step, introduced by the keyword **prove**. This is the proposition that is asserted and proved by this step.

At the end of this proof step, it becomes available as a fact for the next steps of this proof and deeper levels **sub-goals**. In our example, step **<2>1** is available in the proof of **<2>2**, and **<1>1** is available in the proof of **<1>2**.

Following the statement comes the proof of the step. This is where either we ask Zenon to do the proof from facts (**hints**) we give it, or we decide to split the proof in "**sub-steps**" on which Zenon will finally be called and that will serve to finally prove the current goal by combining these

"substeps" lemmas. For instance, the proof of the whole theorem (which is itself a statement) is split in the **sub-goals <2>1** and **<2>2**. In the proof of **sub-goal <2>1**, appears a fact: by hypothesis **h1** definition of belongs. Here Zenon is asked to find a proof of the current goal, using hints it is provided with. The fact definition of belongs refers to the definition of the method **belongs**. The **sub-goal** of the step **<2>2** is resolved by using of the hint property **basics#beq_symm** of the FoCaLiZe basics library in addition to the definition of the method **belongs** and the hypothesis **h2**.

The property **beq_symm** expresses the symmetric property of the basics equality operator (∀a , b : T, a = b ⇒ b = a).

A **FoCaLiZe** development contains both "computational code" (i.e. code performing operations

that lead to an effect, a result) and logical properties. When Compilation of FoCaLiZe sources, two outputs are generated:

- The "computational code" is compiled into **OCaml** source that can then be compiled with the **OCaml** compiler to lead to an executable binary. In this pass, logical properties are discarded since they do not lead to executable code.
- Both the "computational code" and the logical properties are compiled into a **Coq** model. This model can then be sent to the **Coq** proof assistant who will verify the consistency of both the "computational code" and the logical properties (whose proofs must be obviously provided) of the **FoCaLiZe** development.

This means that the **Coq** code generated is not intended to be used to generate an **OCaml** source code by automated extraction. As stated above, the executable generation is preferred using directly the generated **OCaml** code. In this idea, **Coq** acts as an assessor of the development instead of a code generator.

More accurately, **FoCaLiZe** first generates a **pre-Coq** code, i.e. a file containing **Coq** syntax plus "holes" in place of proofs written in the **FoCaLiZe** Proof Language. This kind of files is suffixed by "**.zv**" instead of directly "**.v**". When sending this file to **Zenon** these "holes" will be filled by effective **Coq** code automatically generated by **Zenon** (if it succeed in finding a proof), hence leading to a pure **Coq** code file that can be compiled by **Coq** [13].

1.5. Conclusion

FoCaLiZe is a development workshop and object-oriented programming that covers all phases of the life cycle of software, specification of needs to the executable code generation. The early stages of a development FoCaLiZe consist of the creation of species with signatures and properties.

The signatures are used for the specification of functional requirements and properties for specifying security requirements. Then, the design and architecture of the software are carried out gradually in several levels, using inheritance, parameterization, redefining methods, parameter substitution and late binding.

Each level of specification is a step towards the realization of the model, assigning computational body inheritable signatures and formal proofs to inherited properties. A level properties of Evidence provide a formal traceability with the previous level. Thus, the transition from one level to another leads to completely defines species, creating usable collections in safety by the end user.

Chapter 2
UML/OCL

UML/OCL

UML / OCL model is the conceptual description of a Real system using the graphic notation of UML and OCL. UML allows the description of a model through different views, each represented by one or more diagrams and OCL constraints. A UML diagram is represented by a connected graph where the vertices are the elements and arcs relationships. One chart generally represents one facet of the system built. The UML/OCL concept presented in this section are inspired from [16, 11, 14, 15, 17, 19]

2.1. UML Diagrams

UML uses many types of diagrams (UML 2.4 [16]) to describe the structural aspect and the behavioral aspect of a system. The nine types of commonly used UML diagrams [18] (those of UML1.4 version) are:

Structural diagrams show the **static structure** of the system and their elements represent the meaningful concepts of a system, and may include abstract, real world and implementation concepts.

The four structural diagrams:

- **Class Diagrams:** they are the most common diagrams used in UML. They consists of classes, interfaces, associations and collaborations. Class diagrams basically represent the object oriented view of a system, so it is generally used for development purpose. This is the most widely used diagram at the time of system construction.
- **Object Diagrams:** they can be described as an instance of class diagram. So, these diagrams are more close to real life scenarios where we implement a system. Object diagrams are a set of objects and their relationships just like class diagrams and also represent the static view of the system. The usage of object diagrams is similar to class diagrams but they are used to build prototype of a system from practical perspective.
- **Component Diagrams:** they represent a set of components and their relationships. These components consist of classes, interfaces or collaborations. So, component diagrams represent the implementation view of a system.

- **Deployment Diagrams:** they are a set of nodes and their relationships. These nodes are physical entities where the components are deployed. Deployment diagrams are used for visualizing deployment view of a system. This is generally used by the deployment team.

Behavior diagrams show the **dynamic behavior** of the objects in a system, which can be described as a series of changes to the system over **time**.

The five Behavior diagrams are:

- **Use Case Diagrams:** they are use case diagrams are a set of use cases, actors and their relationships. They represent the use case view of a system. A use case represents a particular functionality of a system. So, use case diagram is used to describe the relationships among the functionalities and their internal/external controllers. These controllers are known as actors.
- **Sequence Diagram:** It is an interaction diagram. As its name indicates, it deals with some sequences, which are messages passing from one object to another to perform a specific functionality.
- **State Machine Diagram (State chart Diagram):** A transition system which is expected to be reacted by some kind of internal/external events. These events are responsible for state change of the system. State chart diagram is used to represent the event driven state change of a system. It basically describes the state change of classes.
- **Activity Diagram:** It shows sequence and conditions for coordinating lower-level behaviors. These are commonly called control flow and **object flow** models.
- **Communication Diagram:** (previously known as **Collaboration Diagram**) is a kind of **interaction diagram**, which focuses on the interaction between **lifelines** where the architecture of the internal structure and how this corresponds with the **message** passing is centralized. The sequencing of messages is given through a **sequence numbering** scheme.

2.1.1. Class diagram

A class diagram provides an overview of a system specification in its classes and the relationships between them. A class diagram is represented by a graph, the nodes are classes and arcs model the relationships between classes (associations, inheritance and dependencies).

UML2 also allows the description of parameterized classes (templates) and substitution of generic formal parameters of a parameterized class by effective generic parameters (template binding), to generate bound models.

2.1.1.1. Attribute (Property)

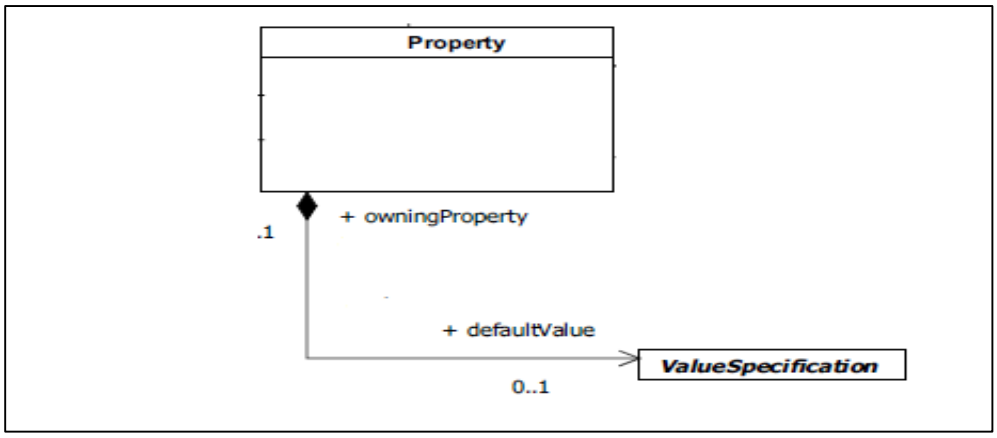


Figure 2.1: Property metamodel

A property is a structural feature that specifies an attribute for a given classifier. A property has a name, a visibility, a type and a multiplicity.

Attribute Type is the data type of the attribute. The type of each attribute should be a primitive type (String, Integer, Real, Boolean, UnlimitedNaturel), or a non-primitive (i.e. another class in model).

- **Visibility** markers indicate if an attribute or operation in a class can be accessed only from within that class (private), from within the class and any derived classes (protected), from within the package that the class is part of (package) or from anywhere (public). Visibility markers are placed at the start of the relevant attribute or operation, using the following signs:

Visibility	Symbol
private	-
protected	#
package	~
public	+

- **Multiplicity:**

  Multiplicity specifies how many instances of the attribute type are referenced by this attribute. Multiplicity is an inclusive interval of non-negative integers to specify the allowable number of instances of described element. Multiplicity interval has some lower bound and (possibly infinite) upper bound:

  Lower and upper bounds could be natural constants or constant expressions evaluated to natural (non negative) number. Upper bound could be also specified as asterisk "*" which denotes unlimited number of elements. Upper bound should be greater than or equal to the lower bound.

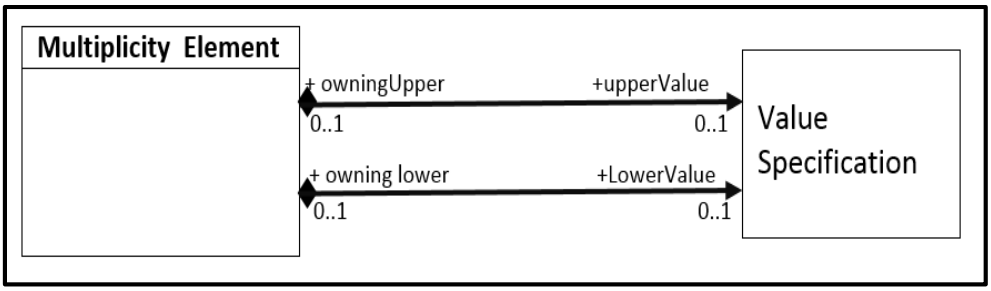


Figure 2.2: Multiplicity metamodel

Some typical examples of multiplicity:

Multiplicity	Cardinality
0..1	No instances or one instance
0..*|*	Zero or more instances
1..*	At least one instance
5..5|5	Exactly 5 instances
m..n	At least **m** but no more than **n** instances

Table 2.1: Multiplicity

2.1.1.2.Operations:

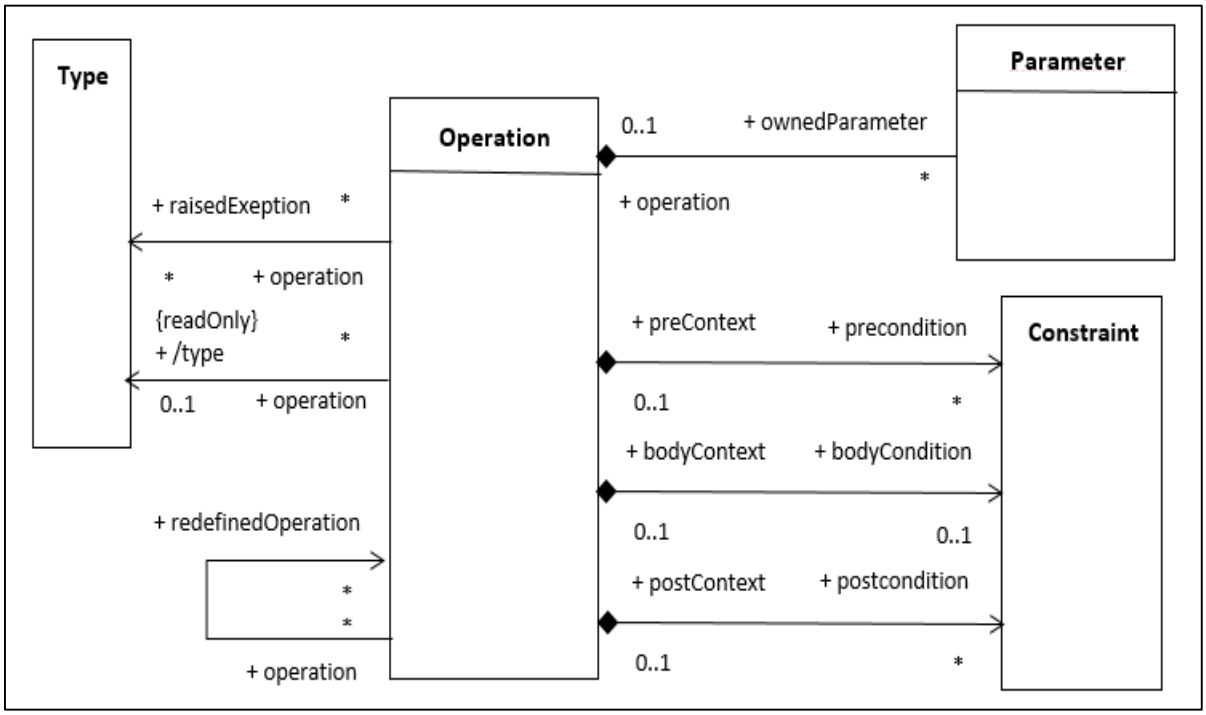


Figure 2.3: Operation metamodel

An operation represents a behavioral feature of a classifier that specifies the name, type, parameters, constraints and a visibility (see Attributes visibility). A parameter of an operation specifies the argument to be passed **in** or **out** during an invocation. For our concern, a parameter is characterized by a name, a direction and a type. Furthermore, an operation can have at most one return parameter.

Example: The Figure 2.4**,** shows an example of a UML class with attributes and operations. The class **Etudiant** has two attributes **Matricule (**Visibility is private **)** and **Moyenne** (Visibility is protected) of integer type, it also has three operations. **Set_Matricule() (**Visibility is public) has **in** parameter **m** of String type but has not a **return** parameter and **set_Moyenne() (**Visibility is public) has **in** parameter **n** of Real type but has not a **return** parameter, and **get_Moynne() (**Visibility is public**).** They do not have an **in** parameter but they have a **return** parameter of Real type.

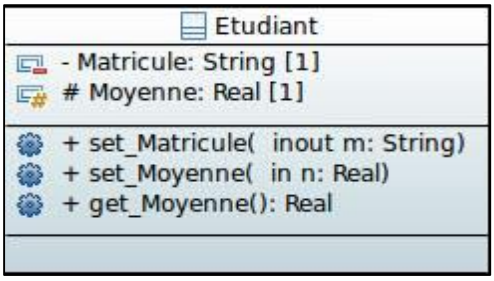


Figure 2.4: UML Class with Attributes and operations

2.1.2. Relationships between classes

2.1.2.1 Dependencies

A dependency is a weak relationship between two classes and is represented by a dotted line. In the example of figure 2.5, there is a dependency between Point and **LineSegment**, because **LineSegment**'s draw () operation uses the Point class. It indicates that **LineSegment** has to know about Point, even if it has no attributes of that type. This example also illustrates how class diagrams are used to focus in on what is important in the context, as you wouldn't normally want to show such detailed dependencies for all your class operations.

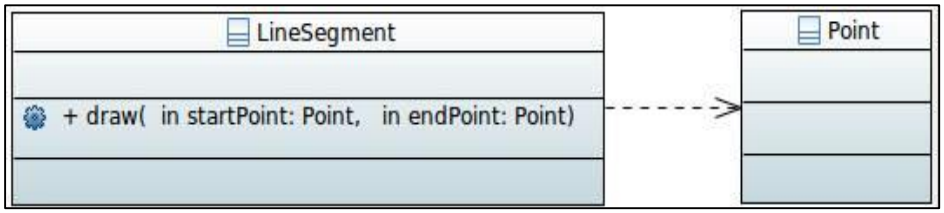


Figure 2.5: UML Class with dependencies

2.1.2.2 Inheritance

The inheritance relationship, also known as the *generalization* relationship, is used to indicate that one class is a specialization of another. In the following example of figure 2.6, the Ellipse and Rectangle classes are derived from the Shape class, because they both share characteristics that are common to all shapes in the system. The Shape class is the base class in both relationships. Inheritance relationships are shown in a hierarchy with the base classes at the top and the derived classes below them.

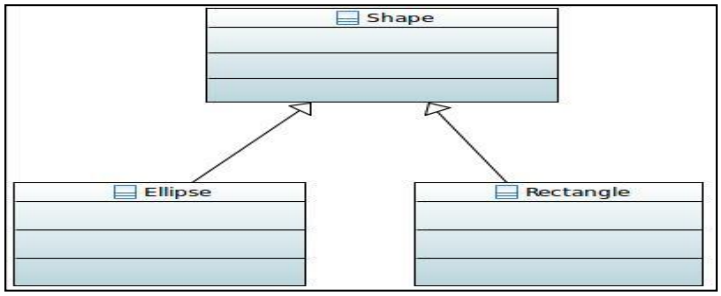


Figure 2.6: UML Class with generalization

It is also possible for a class to inherit from multiple base classes, although some programming languages do not support multiple inheritance. In the example, a managing director is both a manager and a director.

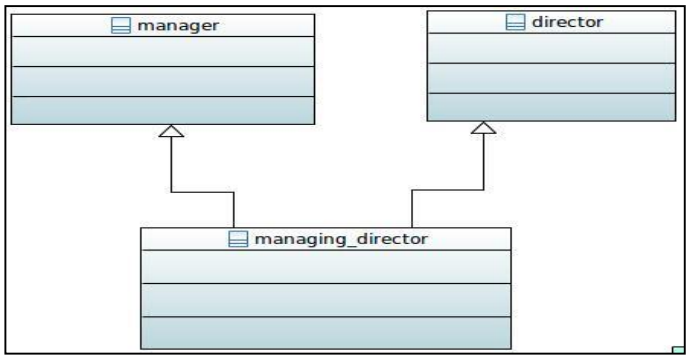


Figure 2.7: UML Class with multiple generalization

2.1.2.3. Parameterized class (Class Template)

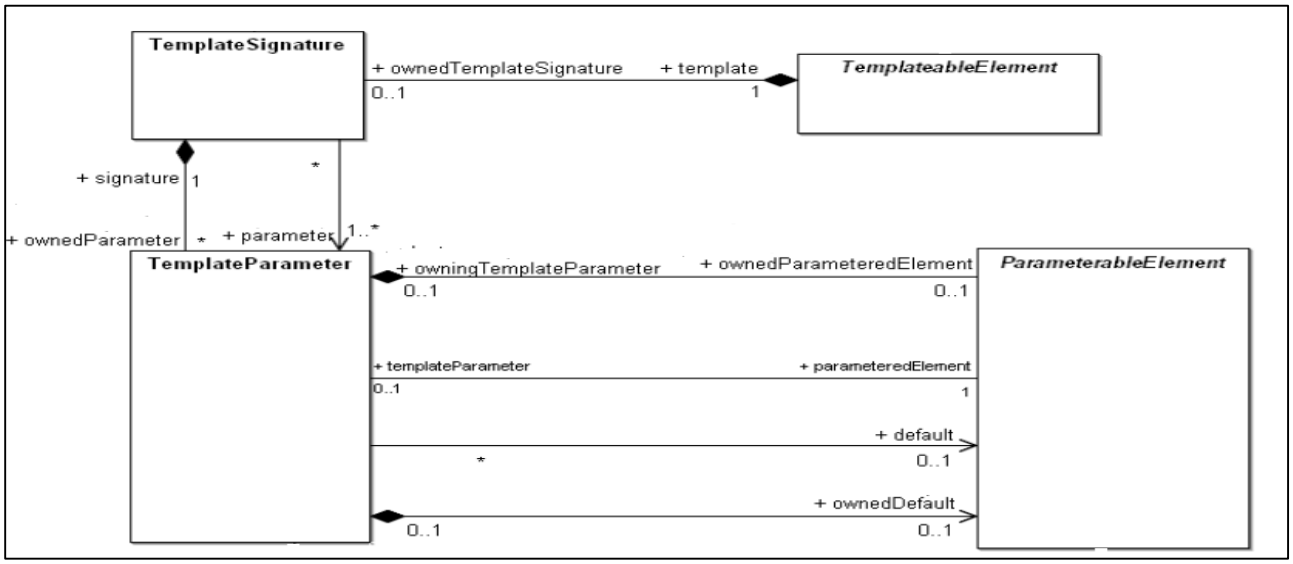


Figure 2.8: Template metamodel

In UML standard, a template is a model element parameterized by other model elements. To specify its parameterization, a template element owns a signature. A template signature corresponds to a list of formal parameters.

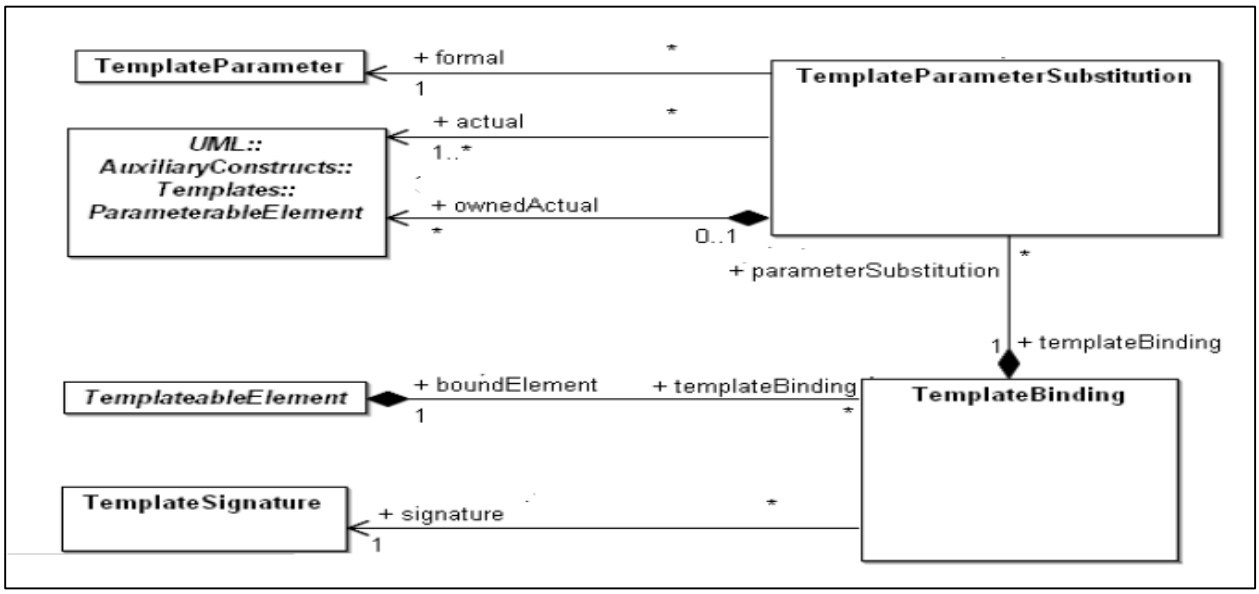


Figure 2.9: Template binding metamodel

A template can be used to generate other model elements using template binding relationship. A bind relationship links a "bound" element to the signature of a target template and specifies a set of template parameter substitutions that associate actual elements to formal parameters.

Example: Figure 2.10 shows how the class PersonStack (a list of persons) is constructed through substitutions of the formal parameters **T: Class** of the class List by the actual parameters Person (T ->Person), where Person is the class that models persons.

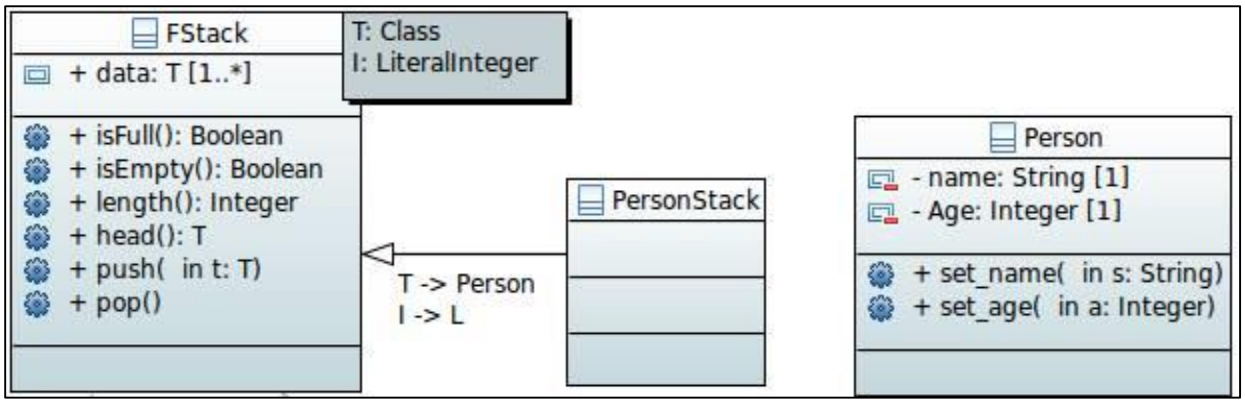


Figure 2.10: Template binding example

2.2. OCL Constraint

The Object Constraint Language (OCL) is a formal language used to describe expressions on UML models. These expressions typically specify invariant conditions and pre/post-conditions that must hold for the system being modeled or queries over objects described in a model.

OCL expressions can be used to specify operations / actions that, when executed, do alter the state of the system. UML modelers can use OCL to specify application-specific constraints in their models.

2.2.1. Why OCL?

A UML diagram, such as a class diagram, is typically not refined enough to provide all the relevant aspects of a specification. There is, among other things, a need to describe additional constraints about the objects in the model. Such constraints are often described in natural language. Practice has shown that this will always result in ambiguities. In order to write unambiguous constraints, so-called formal languages have been developed. The disadvantage of traditional formal languages is they are only usable to persons with a strong mathematical background, but difficult for the average business or system modeler to use.

OCL has been developed to fill this gap. It is a formal language that remains easy to read and write and is a pure expression language: therefore, an OCL expression is guaranteed to be without side effect.

OCL expression is evaluated, it simply returns a value. It cannot change anything in the model and state of the system, even though an OCL expression can be used to specify a state change (for example, in a post-condition).

OCL is not a programming language; Because OCL is a modeling language in the first place therefore, it is not possible to write program logic or flow control in OCL. You cannot invoke processes or activate non-query operations within OCL.OCL cannot to be directly executable.

OCL is a typed language, so that each OCL expression has a type. To be well formed. An OCL expression must conform to the type conformance rules of the language. For example, you cannot compare an Integer with a String. Each Classifier defined within a UML model represents a distinct OCL type.

2.2.2. Language Description

Figure 2.11 gives an overview on the OCL type system in form of a feature model. Feature models allow to describe mandatory and optional features of object, and they allow to specify alternative features as well conjunctive features. In particular, the figure pictures the different kinds of available types.

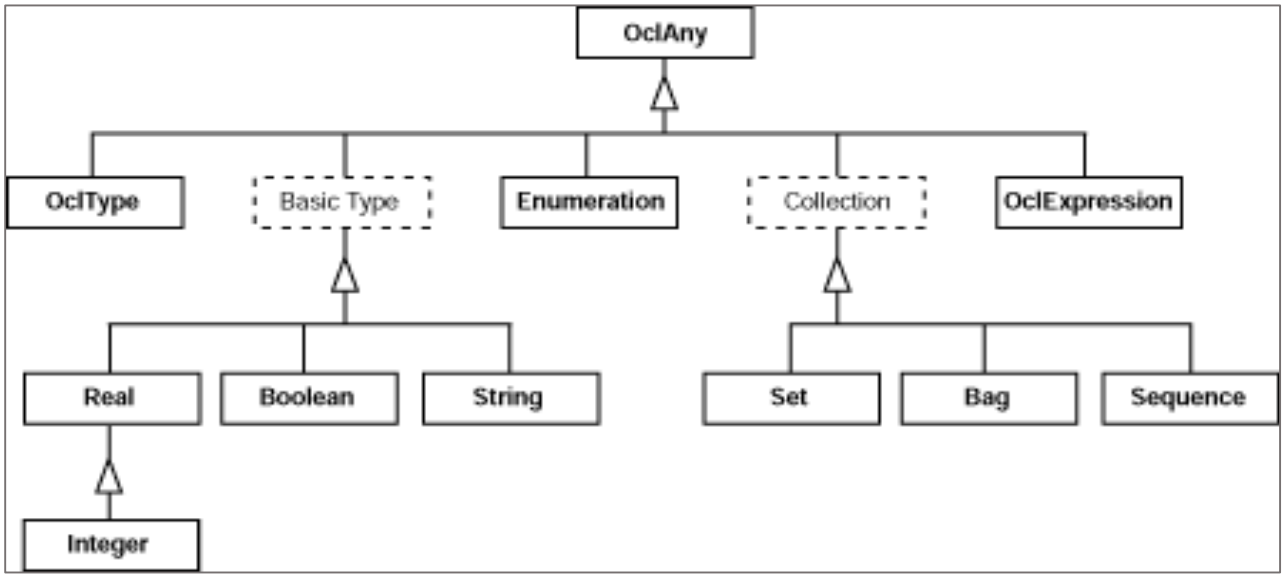


Figure 2.11. OCL type hierarchy

2.2.2.1. Invariants

The OCL expression can be part of an Invariant, which is a Constraint stereotyped as an «invariant». When the invariant is associated with a Classifier, the latter is referred to as a "type" in this clause. An OCL expression is an invariant of the type and must be true for all instances of that type at any time. (Note that all OCL expressions that express invariants are of the type Boolean.). For example, if in the context of the Company type in ***Figure 2.12***, the following expression would specify an invariant that the number of employees must always exceed 30:

self.*numberOfEmployees > 30*

Where *self* is an instance of type Company. This invariant holds for every instance of the Company type.

The keyword self means any instance of the context in class. We can also express the invariant using the *numberOfEmployees* operation:

***Context** Company **inv**: **self** .numberOfEmployees > 30*

In most cases, the keyword *self* can be dropped because the context is clear, as in the above examples. As an alternative for self, a different name can be defined playing the part of self. For example:

***Context** c : Company **inv**: c.numberOfEmployees > 30*

This invariant is equivalent to the previous one.

Optionally, the name of the constraint may be written after the *inv* keyword, allowing the constraint to be referenced by name. In the following example the name of the constraint is *enoughEmployees.**context** c : Company **inv** enoughEmployees: c.numberOfEmployees > 30*

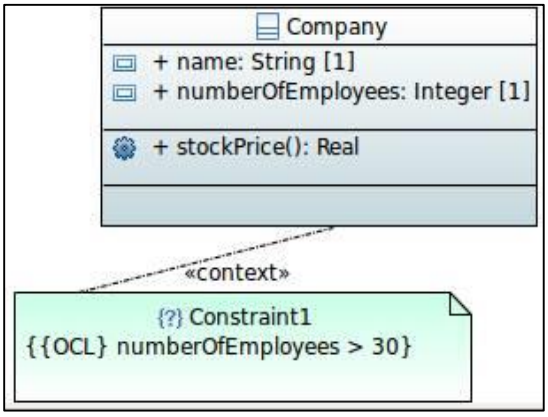


Figure 2.12: Class Company

2.2.2.2. Pre and Post-conditions

The precondition of a class of operation describes a constraint that must be true before the execution of the operation. In other words, it specifies a condition under which an operation result in the call correct behavior.

The post-condition of a class of operation describes a constraint that must be satisfied after the execution of the operation. In other words, it sets out how to be the system after performing an operation. The general syntax of an operation specification with pre- and post-conditions is defined as follows:

***context** C :: op(p1 : T1, . . . , pn : Tn)*
***pre**: P*
***post**: Q*

First, the context is determined by giving the signature of the operation for which pre and post-conditions are to be specified. The operation op which is defined as part of the classifier C has a set of typed parameters **PARAMSop = {p1 . . . pn}.** The UML model providing the definition of an operation signature also specifies the direction kind of each parameter. We use a function kind:

PARAMSop → {**in, out, inout, return**} to map each parameter to one of these kinds. Although UML makes no restriction on the number of return parameters, there is usually only at most one return parameter considered in OCL, which is referred to by the keyword result in a post-condition. In this case, the signature is also written as **C :: op(p1 : T1, . . . , pn-1 : Tn-1) : T** with T being the type of the result parameter.

The precondition of the operation is given by an expression P, and the post-condition is specified by an expression Q. P and Q must have a Boolean result type. If the precondition holds, the contract of the operation guarantees that the post-condition is satisfied after completion of op. Pre and post-conditions form a pair. A condition defaults to true if it is not explicitly specified.

Example: Before updating the value of a person's age attribute (**setAge** the operation (a), see fegure 2.13), it should be ensured that the new age (given as a parameter to **setAge** operation) or greater than the current age. After updating, the age attribute value must be equal to the given value parameter to **setAge** operation. These constraints express pre / post-conditions on setAge operation, defined as follows:

context Person::setAge(a:Integer) **pre** : (a > self.age) **post**: (self.age = a)

When a post-condition it is necessary to reference the value of an attribute before performing the operation in context, it must suffix the name of the attribute with the keyword @pre. For example, we can specify the following post-condition on **birthdayHappens ()** operation:

context Person::birthdayHappens() **post** : self.age = self.age@pre + 1

This constraint expresses the value of the age attribute increments day birthdays.

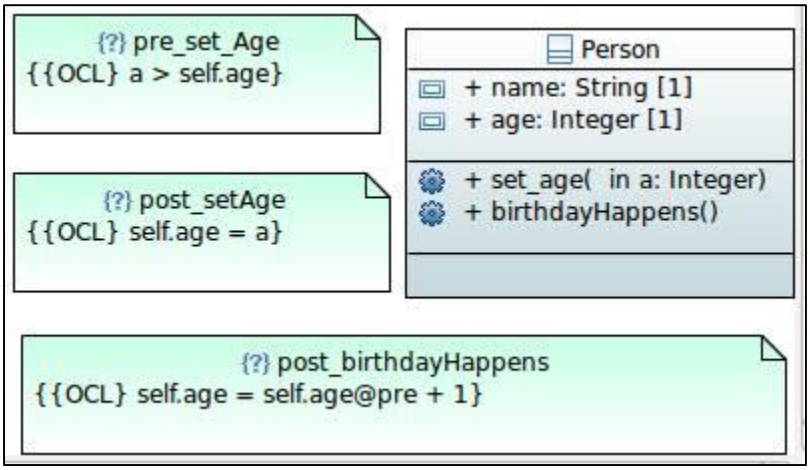


Figure 2.13: Pre and Post-conditions

2.2.2.3 Basic Values and Types

In OCL, a number of basic types are predefined and available to the modeler. These predefined value types are independent of any object model and are part of the definition of OCL. The most basic value in OCL is a value of one of the basic types. The basic types of OCL, with corresponding examples of their values, are shown in the following table:

Type	Values
Boolean	True , False
Integer	1, -5, 2, 32, 1234, …
Real	1.5, 13.2, …
String	'Type String ….. '

Table 2.2: OCL Basic type

OCL defines a number of operations on the predefined types [17]. Table 2.3 gives some examples of the operations on the predefined types:

Table 2.3: UML Class with OCL Operations

	Operations	Comment	Example	
			Expression	Result
Ocl operations for type *Classifier*	allInstances () : Set{T}	Returns a Set containing all of the existing instances of the current classifier (along with instances of all its inherited classifiers)	*let a : String = 'a', b : String = 'b', c : Integer = 2 in String.allInstances()*	*Set{'a','b'}*
Ocl operations for type *OclAny*	oclAsType (typespec : Classifier) : T	Returns self statically typed as typespec if it is an instance of this type. *Note* that this does not alter the runtime value of self, it only enables access to subtype operations. This operation allows users to cast self to another type.	*aPerson.oclAsType(Employee)*	*an object of Employee type*
	oclIsInvalid () : Boolean	Returns true if self is equal to *invalid*.	*let anObject : String = invalid in anObject.oclIsInvalid()*	*true*
	oclIsKindOf(Classifier typespec) : Boolean	Returns true if the type of self corresponds to the type or supertype of typespec, false otherwise. This operation allows users to check the class hierarchy of self much like would an instanceof Java.	*anEmployee.oclIsKindOf(Person)*	*true*
	oclIsTypeOf(typespec : Classifier) : Boolean	Returns true if the type of self is the same as typespec, or false otherwise. This operation allows users to check the exact class type of self	*anEmployee.oclIsTypeOf(Person)*	*false*

	oclIsUndefined () : Boolean	Returns true if self is equal to invalid or null	*let anObject : String = invalid in anObject.oclIsUndefined()*	*true*
	<> (object : OclAny) : Boolean	Returns true if self i s a different object from object.	*let a : Integer = 2, b : Real = 2.0 in a <> b*	*false*
	= (object : OclAny) : Boolean	Returns true if self i s equal to object	*let a : Integer = 2, b : Real = 2.0 in a = b*	*true*
	< (object : T) : Boolean	Returns true if self i s comparable to object and less than object.	*let a : Real = 1.5, b : Integer = 2 in a < b*	*true*
	> (object : T) : Boolean	Returns true if self i s comparable to object and greater than object	*let a : Real = 1.5, b : Integer = 2 in a > b*	*false*
	<= (object : T) : Boolean	Returns true if self i s comparable to object and less than or equal to object.	*let a : Real = 1.5, b : Integer = 2 in a <= b*	*True*
	>= (object : T) : Boolean	Returns true if self is comparable to object and greater than or equal to object.	*let a : Real = 1.5, b : Integer = 2 in a >= b*	*False*
Ocl operations for type *String*	concat (s : String) : String	Returns a String containing self followed by *s*.	*'concat'.concat('').concat('operation')*	*'concat operation'*
	size () : Integer	Returns the number of characters composing self.	*'size operation'.size()*	*14*
	subString (lower : Integer, upper : Integer) : String	Returns a String containing all characters from self starting from index *lower* up to index *upper* included. Both *lower* and *upper* parameters should be contained between *1* and *self.size()* included. *lower*	*'subString operation'.subString(1, 1)*	*'s'*

		cannot be greater than *upper*.		
	toInteger () : Integer	Returns an Integer of value equal to self, or |invalid| if self does not represent an integer.	*'14'.toInteger()*	*14*
	toLower () : String	Returns self with all characters converted to lowercase	*'LoWeR OpErAtIoN'.toLower()*	*'lower operation'*
	toReal () : Real	Returns a Real of value equal to self, or |invalid| if self does not represent a Real.	*'20'.toReal()*	*20.0*
	toUpper () : String	Returns self with all characters converted to uppercase.	*'UpPeR OpErAtIoN'.toUpper()*	*'UPPER OPERATION'*
Ocl operations for type *Number*	Number::abs () : Number	Returns the absolute value of self, self if it is already a positive number.	*-10.abs()*	*10*
	Number::floor () : Integer	Returns the integer part of self if it is a Real, self if it is an Integer.	*(3.7).floor()*	*3*
	Number::max (r : Number) : Number	Returns the greatest number between self and *r*.	*5.max(7.3)*	*7.3*
	Number::min (r : Number) : Number	Returns the lowest number between self and *r*.	*6.min(5.2)*	*5.2*
	Number::round () : Integer	Returns the nearest integer to self if it is a Real, self if it is an Integer	*(5.5).round()*	*6*
	Integer::div (i : Integer) : Integer	Returns the integer quotient of the division of self by *i*.	*11.div(3)*	*3*

	Integer::mod (i : Integer) : Integer	Returns the integer remainder of the division of self by *i*.	*11.mod(3)*	*2*
Ocl operations for type * Boolean*	And		*True And false* *False And false* *True And True*	*False* *False* *True*
	Implies		*True Implies false* *False Implies false* *True Implies True*	*False* *False* *True*
	Or		*True or false* *False or false* *True or True*	*True* *False* *True*
	Not		*True* *False* *Invalid*	*False* *True* *invalid*
	Xor		*True xor false* *False xor false* *True xor True*	*True* *False* *False*

2.3. Example of UML/OCL model

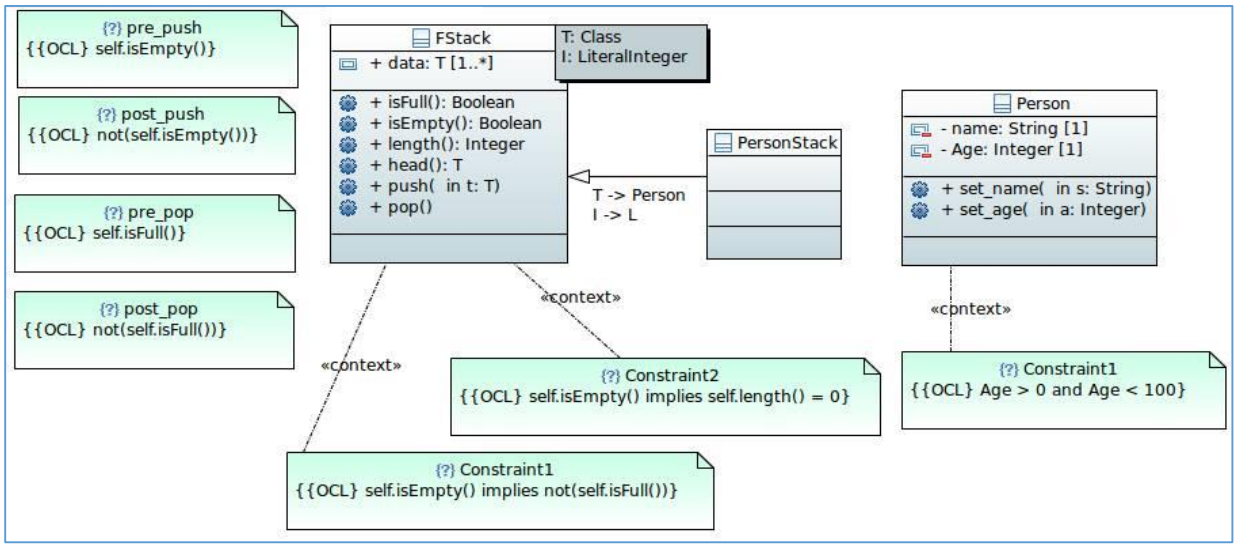


Figure 2.14: Class diagram with OCL constraints

This figure represents an example of class diagram model with OCL constraints, in this model we have the template **Fstack** that models finite stacks. The parameter signature states two elements: T of type **classe** and **I** of type **LiteralInteger**. The attribute **data** of the class **FStack** is multivalued (the multiplicity is greater than 1). The operations **isFull** (), **isEmpty**(), **length()**, **head()**, **push(in t: T)**, **pop ()** of the class **FStack** are usual methods to

manipulate stack instances. The class **PersonStack** (a stack of persons) is constructed through substitutions of the formal parameters of the class FStack by actual parameters (T →Person, l→L, where L = 100). The class Person models persons with tow attributes name and age and tow operation **set_name** and **set_age** (to change the attributes values).For the class Person we have an invariant to verify the persons age (age > 0).

The class Fstack is specified with tow invariants and tow pre/post-conditions specifying the stacks operations pop () and push ():

- **inv-Constraints1**: for all stack s, if s is empty then it is not full (not (s.isFull())).
- **inv-Constraints2**: for all stack s, if s is empty then it contains no elements (s.length()=0)
- **Pre-push/post-push**: stacking an element in an empty stack and then unstacking it out, the stack remains empty.
- **Pre-pop/post-pop**: stacking an element in an not full stack and then unstacking it out, the stack remains unchanged.

2.4. Conclusion

In this chapter, we presented the UML / OCL subsets supported by our study. This subset allows the modeling and specification of static and dynamic aspect of systems, using class diagrams, charts state chart and OCL constraints. It is noteworthy that are supported the UML features like multiple inheritance, late binding, the redefinition of methods and substitution of formal parameters of a parameterized class (template) with actual parameters (template binding), so what are rarely considered in similar studies. We also used OCL expressions to specify the conditions of guard's transitions, allowing a rigorous specification of state charts.

Chapter 3

UML/OCL: Related Work

UML/OCL: Related Work

Various approaches have been used to provide supporting tools and techniques to formalize UML models and check OCL constraints. The most widely used are set theory based methods such as the B method [20] and Alloy [21, 22]. Several approaches use first and higher-order logic based tools as Maude [23] and Isabelle/HOL [24]. Other approaches and techniques use Real time checker tools such as PROMELA, SMV and LOTOS.

Some other works are focusing on specific aspects of UML/OCL using formal methods such as rCOS [25] and CASL [26].

3.1.Transformation to B method

B is a method for specifying, designing and coding software systems based on set theory. Among works interested on the transformation of UML/OCL models into the B method based tools those focus on the study of the transformation rules and verification techniques [27,28], while others concentrate on the generation of concrete transformation tools (ArgoUML+ B [29], UML2B [30] and UML-B [31]).

Most of UML features and OCL constraints are supported in the transformation into B method. However, the multiple inheritance mechanism, UML templates (with template bindings) and the propagation of OCL constraints are not supported.

3.2. Transformation to Alloy

Alloy is a formal language with restricted set-based syntax, for describing structural properties. It has been recently used to formalize and check the consistency of UML/OCL models [32–33]. Using Alloy, a UML class is modeled by a signature (a set of atoms).

Simple inheritance is considered through the clause extends, and some kind of parameterization is taken care of at module level. However, as for B, the multiple inheritance cannot be directly represented in Alloy [34]. Furthermore, UML templates and template bindings are not supported, and it is not possible to import abstract modules (at specification level) and bind modules parameters to create actual modules.

3.3. Transformation to HOL-OCL and Maude

Isabelle [35] is a generic theorem prover of the Logic for Computable Functions (**LCF**) prover family implemented in Standard Meta Language (**SML**). We heavily use the possibility to build

SML programs performing symbolic computations over formulae in a logically safe way on top of the logical core engine: this is how the proof procedures of **HOL-OCL** are built technically.

Isabelle/HOL offers support to check the conservatively of definitions, data types, primitive and well-founded recursion, and powerful generic proof engines based on rewriting and tableaux provers. It may generate the proof objects and check the derivations by an independent program following the rules of the logic. Besides the **SML**-based programming interface, there is also an own input language to Isabelle theories, called Intelligible semi-automated reasoning, Isar. This letter allows for writing specifications consisting of definitions, proofs, and technical setups for the prover in a fairly readable way. In particular, Isabelle can generate **LATEX** documentation while checking the entire hierarchy of theories (this is, what this document is . . .). Isabelle theories written in **Isar** are supported by a fairly powerful E macs-based front-end called Proof General.

Higher-order Logic (HOL) [36, 37] is a classical logic with equality enriched by total polymorphic1 higher-order functions. It is more expressive than first-order logic, e. g., induction schemes can be expressed inside the logic. Pragmatically, **HOL** can be viewed as a combination of a typed functional programming language like **SML** or Haskell extended by logical quantifiers.

HOL-OCL is an integrated and customized for this front-end, such that **HOL-OCL** specific Isar commands and **HOL-OCL** specific fonts can be used. **HOL-OCL** [39, 38] maps OCL constraints into higher-order logic. It concentrates on producing OCL evaluator tools. HOL-OCL supports most of UML/OCL features, in particular simple inheritance (expressed as inclusion of sets) and late binding mechanisms.

The Maude system is based on rewriting logic and runtime checking tools. It is also used as target language in the formalization of UML/OCL models [40, 41]. The simple inheritance feature is supported through the definition of predicates which establish that one class is a subclass of another class. Moreover, Maude allows to specify a module parameterized with formal parameters. We think that this feature enables to formalize UML template and template binding.

But, although HOL-OCL and transformation to Maude are the most powerful tools, they also ignore multiple inheritance, UML templates, template bindings and the propagation of OCL constraints over these features.

3.4. Transformation to FoCaLiZe

In [11] they studied the transformation rules of UML models expressed through class diagrams into FoCaLiZe specifications, and then they propose an implementation of these transformation rules. The purpose of their transformation is to take advantage of the FoCaLiZe verification tools to check the properties of a UML models. The proposed implementation directly supports most of UML features such as inheritance, multiple inheritance, dependency, UML templates and template binding.

Table. 3.1: Formal methods

object-oriented functionality, technical evidence and programming paradigm

	Architectural and design features							Verification			Programming paradigm ñ	
	Encapsulate	**Simple inheritance**	**Multiple inheritance**	**Redefinition methods**	**Late binding**	**Specification of the parameters models**	**Substitution of formal parameters by actual parameters being specification**	**Propagation properties through inheritance or parameterization**	**Model checker or generation tests**	**Theorem prover**	**Imperative**	**Functional**
B	X	X				X				X	X	
Alloy	X	X				X			X			
Maude	X	X				X	X		X	X	X	
HOL-OCL	X	X		X	X	X				X		X
FoCaLiZe	X	X	X	X	X	X	X	X	X	X		X

3.5. Conclusion

To sum up, powerful methods like B, HOL, Maude and Alloy have proven their capabilities to support several UML and OCL features, but they ignore several essential features. In this thesis, we aim to propose a transformation of OCL constraints specified on UML classes into FoCaLiZe specification, based on the transformation approach of UML classes into FoCaLiZed presented in [11]. This transformation will take into consideration (among others) the multiple inheritance (with methods redefinition and late binding), UML templates, template bindings, dependencies and the propagation of OCL constraints via these features.

Chapter 4

Transformation of OCL constraints

Transformation of OCL constraints

An OCL constraint (see section 2.2, 35) is an expression involving OCL types and operations of OCL language. We distinguish the use of the following types:

- Primitive types: Integer, Boolean, Real and String.
- Enumeration types: Enumerations defined in a UML model.
- Object types: That issued from UML classes.
- Collection types: Collection (T) such that T is an OCL type.

Enumerated types and object types (classes) have already handled in [11] that was dedicated to the transformation of UML class diagrams, where each UML class is transformed into a FoCaLiZe species. Indeed, any UML enumeration is also an OCL enumeration and any UML class is an OCL object type. All OCL operations on the types Integer and Boolean are considered in FoCaLiZe through the types int and Bool. However, the OCL operations on the types Real, String and collection are not provided with FoCaLiZe. As well, to transform OCL expressions on these latter types, we had to build libraries formalizing the OCL expressions ignored in FoCaLiZe:

- ✓ The Real operation library (see Appendix A.1) defines the OCL operations on the Real type.
- ✓ The Real_operation library (see Appendix A.2) defines the OCL operations on the type String.

We should also mention that the constraints OCL supported by our study are the invariants of classes and the pre/post-conditions of class operations. As well, the OCL constraints specified in the context of a UML class are transformed into properties (property) of the corresponding species.

In this chapter, we describe the transformation of OCL operations on primitive types then we begin the transformation of the constraints OCL.

Notations: We will use this code [[]] to specify the conversion process, for example:

• For every element **e** in OCL model, we denote [[e]] transformation into Focalize.

4.1. primitive expressions

4.1.1. Integer expression

Using the correspondences between the Integer type of OCL language and the int type of FoCaLiZe, all OCL operations on the Integer type are expressed by FoCaLiZe expressions on the type int. The Table 4.1 defines the correspondence between the operations of these types:

OCL	FoCaLiZe	Comment
n	*N*	*n is an integer value.*
$\alpha + \beta$	[[α]] + [[β]].	*α and β are integer expressions.*
$\alpha - \beta$	[[α]] - [[β]]	
$-\alpha$	-[[α]]	
$\alpha * \beta$	[[α]] * [[β]]	
α div β	[[α]] / [[β]]	
α mod β	[[α]] % [[β]]	
α.min(β)	min0x([[α]], [[β]])	
α.max(β)	max0x([[α]], [[β]])	
α.abs	abs0x([[α]])	Absolute value.

Table 4.1: Transformation of OCL expressions on Integer

4.1.2. Real expression

The Real type is defined in the two languages: OCL (Real) and FoCaLiZe (float). As well, the OCL expressions on the Real type are modeled by equivalent expressions, using the float type to FoCaLiZe. Unlike the OCL type integer, all OCL operations on the Real type are not predefined in FoCaLiZe. Thus, for any OCL operation on the Real type, we define a corresponding operation in the Real_operations library. The OCL operator of Real addition (+) is modeled by the definition of the operator *+f* in the Real_operations Library, as follows:

```
let ( +f ) =
internal float -> float -> float
external
| caml -> {*(+.)*}
| coq -> {* Rplus *} ;;
```

The definition of the FoCaLiZe operation *+f* is given at the top level, based on the operations +. of the Ocaml language and Rplus of Coq. Because of the purely functional Paradigm of the FoCaLiZe language, we distinguish between the integers addition operation (denoted: int -> int -> int) and the Real addition operation (denoted. float -> float -> float). The other OCL operations on the Real (*,-, /, etc.) are modeled in a similar way (see Appendix A.1). The correspondence between the OCL operations on the Real type and FoCaLiZe operations on the float type (Table 4.2), is given as follows:

OCL	***FoCaLiZe***
x + y	*x +f y*
x – y	*x -f y*
*x * y*	*x *f y*
a / b	*a /f b*
a > b	*a >f b*
a >= b	*a >=f b*
a < b	*a <f b*
a <= b	*a <=f b*
a.max(b)	*max_f(a , b)*
a.min(b)	*min_f(a , b)*
a.abs	*abs_f(a)*
a.round	*round_f(a)*
a.floor	*floor_f(a)*

Table 4.2: Modeling OCL operations on the type Real

4.1.3. String expression

As the Real type, the type String (String in OCL / String in FoCaLiZe) is defined in the both languages. As well, the OCL expressions of type String are formalized by FoCaLiZe expressions of String type. Most of the OCL operations on the String type (with the exception of the concatenation operation of chains, ^) are not provided with FoCaLiZe. Thus, for any OCL operation on the String type, we define a corresponding operation in the library of String_operations.

The operation size () (returning the length of a String) in OCL is modeled by the definition of the operation length () in the library String_operations, as follows:

```
let length =
internal String -> int
external
| caml -> {* String.length *}
| coq -> {* length *} ;;
```

The definition of the FoCaLiZe length () operation is based on the operation length of the OCaml (String.lenght) and the operation length of Coq. The OCaml operation length will generate the executable of the new operation and the operation Coq will be used in the proof of properties on this operation.

The other OCL operations on the String type (subtraction of a sub-chain, conversion into tiny and conversion into uppercase) are modeled in a similar way (see Appendix A.2).

Using the FoCaLiZe operations defined in the library String_operations, the OCL expressions on the String type are formalized by the corresponding expressions, as follows (Table 4.3):

OCL	FoCaLiZe	Comment
a.concat (*b*)	([[*a*]] ^ [[*b*]])	Concatenating Strings.
a.size ()	Length ([[*a*]])	Length of a chain.
a.subString (*lower*, *upper*)	subStr([[*a*]] , *lower* , *upper*)	The extraction of a chain.
a.toUpper ()	toUpper ([[*a*]])	Conversion ‘a’ to ‘A’.
a.toLower ()	toLower ([[*a*]])	Conversion ‘A’ to ‘a’.

Table 4.3: Modeling OCL operations on a String

4.1.4. Boolean expression

The OCL formulas (the OCL expressions of Boolean type) are modeled by FoCaLiZe expressions of Bool type. As in the case of integer type, all OCL operations on the Boolean type are predefined in the standard library of FoCaLiZe (Table 4.4):

OCL	FoCaLiZe
True	True
False	False
not(φ)	~~([[φ]])
φ and ψ	[[φ]] /\ [[ψ]] / [[φ]] && [[ψ]]
φ or ψ	[[φ]] \/ [[ψ]] / [[φ]] || [[ψ]]
φ xor ψ	[[φ]] |<>| [[ψ]]
φ implies ψ	[[φ]] -> [[ψ]]
if φ then ψ else ϕ	if [[φ]] then [[ψ]] else [[ϕ]]
let x : *type* = *Exp* in φ	let x = [[*Exp*]] in [[φ]]
$\alpha = \beta$	[[α]] = [[β]]
$\alpha <> \beta$	~~([[α]] = [[β]])
$\alpha > \beta$	[[α]] >*0x* [[β]]
$\alpha < \beta$	[[α]] <*0x* [[β]]
$\alpha >= \beta$	[[α]] >=0x [[β]]
$\alpha <= \beta$	[[α]] <=0x [[β]]

Table 4.4 : Transforming OCL formulas

Where ϕ and ψ are two OCL formulas, α and β are two numerical expressions (two integers or two Reals), o is an object of the model.

4.2. OCL constraints

All the OCL constraints (invariants of classes and pre/post-conditions of classes operations) are translated into FoCaLiZe properties (property). The OCL constraints in the same class (the class into context) are translated into properties of the species derived derived from class. Then, as in UML, the properties of a species are automatically propagated through inheritance and the parameterization in FoCaLiZe.

4.2.1. Invariant

An OCL invariant *E_inv* of class *c_n* (see formula, page) is converted into FoCaLiZe property in the corresponding species to the class *c_n*, as follows:

OCL	FoCaLiZe
context *c_n inv : Einv*	***property*** *inv_i* : ***all*** *e* : ***Self***, *[[Einv]] ;*

The transformation of the OCL expression describing the invariant (E_inv) to FoCaLiZe is inspired from the the basic transformations rules presented above. In an OCL expression, to access

the attribute value of the class (in context) is transformed by the invocation of the getter function modeling the attribute in the corresponding species and the invocation of an operation of the class is modeled by the invocation of the corresponding function of the species derived from the class. At the OCL level, the invariants of classes are generally statements without identifiers. Contrary, at FoCaLiZe level, identifiers of the properties are required. Thus, we adopt an automatic generation mechanism of a unique identifier (inv_1, inv_2, etc) to each property derived from an invariant.

Example: We complete now, the transformation of the Company class (see section 2.2.2.1, page 36) by the transformation of the invariant of class (*numberOfEmployees* > 30, see figure 2.12, page 37), as follows:

public *class Company =* *...* *attribute - numberOfEmployees:Integer* ***end*** ***context*** *Company inv :* *numberOfEmployees > 30*	***species*** *Company =* *...* ***signature*** *get_ numberOfEmployees : Self -> int;* ***property*** *inv_1: all p:Self,* *(get_ numberOfEmployees (p) >0x 30);* ***end;;***

Note that if a UML class having several OCL invariants, the transformation will produce several properties in the corresponding species.

4.2.2. Pre/post-conditions

An OCL pre-condition (Epre) and post-Condition (Epost) of an operation op_n of Class c_n (see formula 2.3,page 40) specifies that the post-condition must be satisfied after the execution of the operation, when the pre-condition is satisfied just before the execution of the operation. As well, we transform a pro/post-condition (identified by pre_post_ident) into an implication (pre-condition -> Post-condition) in the corresponding species, as follows:

OCL	***FoCaLiZe***
context *c_n : : op_n(p1_n : typeExp1 . . . pk_n : typeExpk)* ***pre*** *:Epre* ***post*** *:Epost*	***property*** *pre_post_op_n_i :* ***all*** *e : Self,* ***all*** *p1_n : [[typeExp1]] , . . . , all p1_k : [[typeExpk]] , [[Epre]] -> [[Epost]] ;*

We integrate always the quantification all e: Self at the end of the use e in the transformation of Epre formulas and Epost. These latters are expressed by operators of different OCL types. Their transformations into FoCaLiZe ([Epre], [Epost]) use all the transformation rules of Classes, Enumerations and primitive types (integer, Real, Boolean, String), present above. As in the transformation of the invariants, a unique identifier (PRE_POST_OP) is assigned to each property FoCaLiZe derived from a pre/post-condition.

Example: The pre/post-condition of the operation push (t: T) of the class FStack (see figure 2.14, page 43) is transformed into a property in the corresponding species as follows:

public class *FStack =*	***species*** *FStack (Obj is Basic_object, i in IntCollection) =* ***inherit***
operation *+ head():T*	***Basic****_object;*
operation *+ push(t:T)*	***signature*** *head :* ***Self*** *->* ***Obj*** *;*
operation *+ pop()*	***signature*** *push : Obj ->* ***Self*** *->* ***Self****;*
end	***signature*** *pop :* ***Self*** *->* ***Self****;*
context *FStack :: push(t:T)*	***property*** *pre_post_push_1:*
pre *: self.isEmpty()*	***all*** *e : Self, all t : Obj,* ***isEmpty****(e) -> ~~(****isEmpty****(push (e , t)))*
Post*: not(self.isEmpty())*	***end;;***

Note that a class of operation may have multiple pre / post-conditions, which leads to several properties specifying the corresponding function in the species derived from the class.

4.2.3. Post-condition using @pre

In an OCL post-condition, to refer to the value of an attribute at the start of the operation (just before invocation of the operation), one has to post-fix the attribute-name with the commercial at sign @, followed by the keyword pre. Unlike in other formal languages, we find it very easy to map the expression @pre into FoCaLiZe, since FoCaLiZe distinguishes between the entity which invokes the function and the entity returned by the function due to its functional paradigm.

Example: When the anniversary of a person is reached, his age increments. This constraint (see section 2.13, page 39) expresses a post-condition of the operation birthdayHappens() of the Person class, transformed as follows:

context *Person : :birthdayHappens()*	***property*** *post_birthdayHappens:* ***all*** *x : Self,*
post *: age = age@pre + 1*	***get_age****(birthdayHappens(x)) =* ***get_age****(x) + 1;*

The transformation of this constraint is realized in the context of this case Person (derived from class Person).

4.3. Conclusion

In this chapter, we have proposed an approach of transformation of OCL constraints to FoCaLiZe. The OCL constraints supported are the invariants of classes and the pre/post-conditions of operations of classes. From an OCL specification conforms to the syntax of the subset of supported OCL (see Figure 2.2, page 30), we produce a transformation equivalent specification. The OCL expressions of primitive types are transformed into expressions of the corresponding types in FoCaLiZe. More specifically, the proposed transformation preserves the following properties on the constraints OCL through class diagrams features:

- All properties (of a species) derived from the constraints OCL are propagated through the mechanism of the (multiple) inheritance: The properties of a super-species are also properties of its inheritors (sub-species).

- All properties (of a species) derived from OCL constraints are propagated through the mechanism of parameterization in FoCaLiZe: The properties of a supplier species (used as formal parameter) can be used safely by client's species (parameterized species).

- The properties of a species derived from a parameterized class (UML template) also become properties of species derived from the bound models.

Chapter 5
Implementation

Implementation

In this chapter, we describe the different steps of our transformation process in order to transform UML/OCL Model into FoCaLiZe.

Our transformation process consists of three steps

- **Firstly**, we use a UML graphical environment that supports the OMG meta-model to create our UML/OCL model and generate its corresponding XMI format (XML Meta data exchange).
- **Secondly**, consists to restructure the UML/OCL model (to be transferred) according to the existing relations between classes (dependency, generalization, template binding).
- **Thirdly**, the last step consists to apply our transformation rules to generate the FoCaLiZe code.

To implement our transformation process we have used:

- ✓ Eclipse environment with Papyrus plugin.
- ✓ XSLT (Extensible stylesheet transformation language).

5.1. The development environment

5.1.1. Eclipse

Eclipse is an integrated development environment (IDE). It contains a base workspace and an extensible plug-in system for customizing the environment. Eclipse is written mostly in Java and its primary use is for developing Java applications, but it may also be used to develop applications in other programming languages through the use of plugins, including: Ada, ABAP, C, C++, COBOL, Fortran, Haskell, JavaScript, Julia, Lasso, NATURAL, PHP, Prolog, Python, Ruby, Rust, Scala, Groovy, Scheme. It can also be used to develop packages for the software Mathematica. Development environments include the Eclipse Java development tools (JDT) for Java and Scala, Eclipse CDT for C/C++ and Eclipse PDT for PHP, among others [42].

5.1.2. Papyrus plugin

Is plugin in eclipse environment for create UML/OCL model, this specific chose because the large flexibility of eclipse environment, and because the complete implementation of all a UML2

features by Papyrus and support OCL language [43].

5.1.3. XSLT

In this work, our choice is put on XSLT (Extensible Stylesheet Language Transformation), since it allows us to directly transform a UML/OCL model to FoCaLiZe source (text), so we do not need to define a meta-model for FoCaLiZe, as it is required by ATL (ATLAS Transformation Language) [44] and ETL (Epsilon Transformation Language) [45].

XSLT [46] is a W3C recommendation. It is a declarative language that enables the transformation of XML documents into various other XML (with a different structure), HTML document or text document. To perform a transformation we need to an XSLT Stylesheet that describes the transformation rules and XSLT processor. This latter is implemented in most of development environments (java, c#, php ...).

5.2. Implementation process

based on the eclipse environment and papyrus tools we have developed XSLT stylesheet that allows to order and transform the UML/OCL diagram into FoCaLiZe specification, and we have developed a plugin that enable to transform the UML/OCL (using the XSLT stylesheet) and compile the generated code (using the FoCaLiZe compiler).

The Figure 5.1 shows the different steps of our implementation process, starting by the creation of (class diagram & OCL constraint) and the generation of its XMI format, which is not ordered.

Then we use the XSLT stylesheet1 which is responsible for the ordering of the XMI document. After that, we use the XSLT stylesheet2 which implements our transformation rules to generate FoCaLiZe source.

Finally we compile the FoCaLiZe source to ensure the correctness of the output.

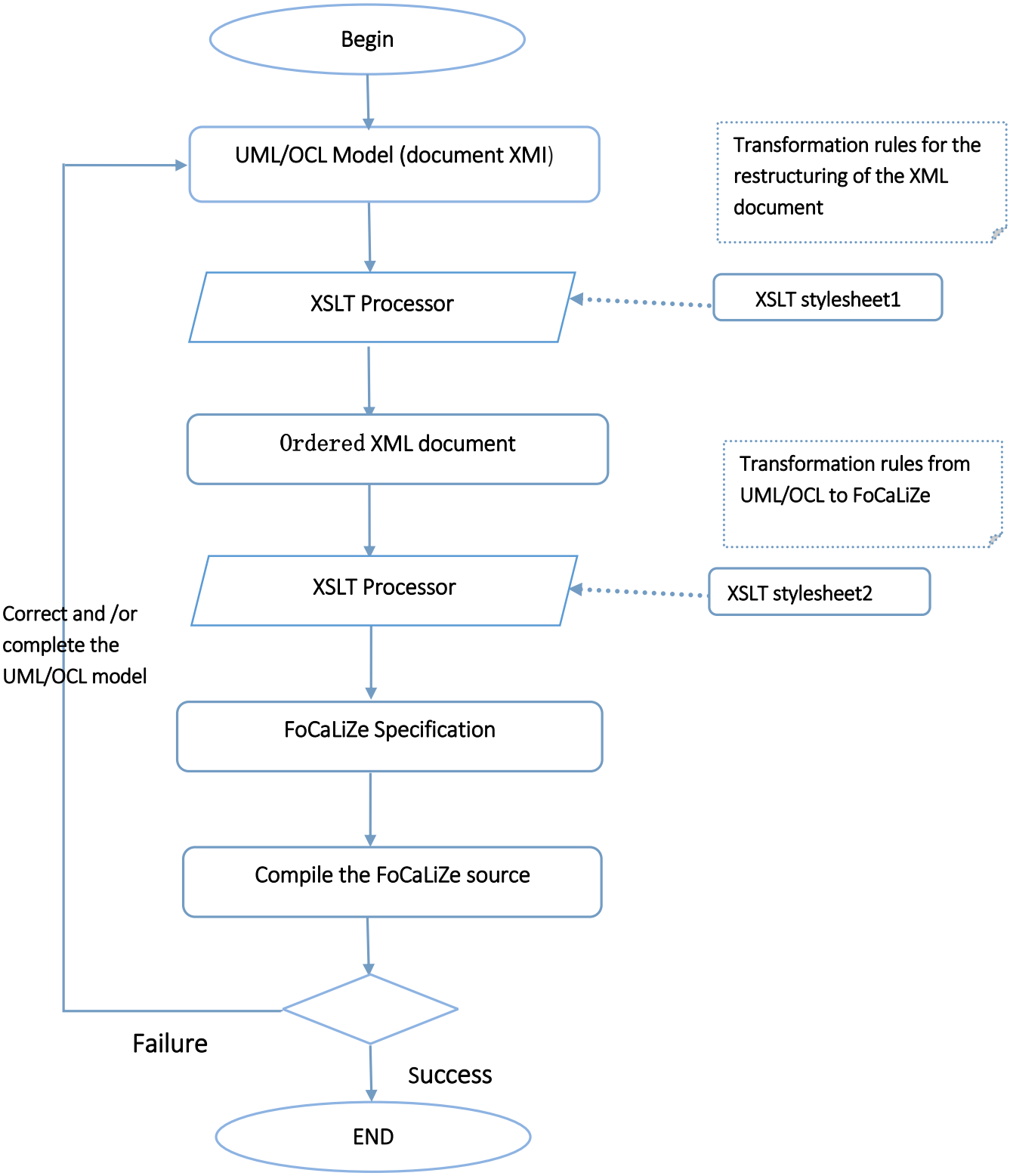


Figure 5.1: Systematic transformation from UML/OCL to FoCaLiZe

5.2.1. Creation of the UML/OCL model

First we start by creating a class diagram and OCL constraint using Papyrus[2] plugin, and generating its XMI format. For create a class diagram & OCL constraint, the entities (nodes and edges) can be dragged and dropped from the palette at the right side of the screen. The following figure shows an example of class diagram & OCL constraint, it consists of some classes with relationships between them.

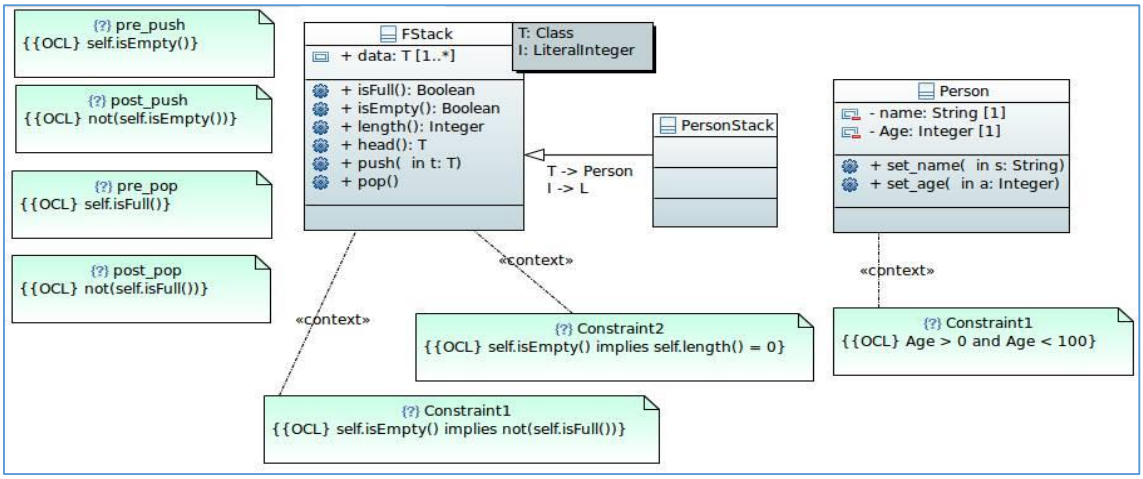


Figure 5.2: A class diagram & constraint OCL

To generate the XMI file we save the project, then the XMI file is automatically saved under the name model.uml file.

The next figure is an extract from the XMI format for the provirus class diagram.

Figure 5.3: Example of XMI format

[2] A fallible at http://www.eclipse.org/papyrus/downloads/index.php.

5.2.2. Generating the ordered XMI document

FoCaLiZe is a formal language that does not support the jump and return property. So, we need to make the classes in the correct order before their transformation. For example, if the class C1 depends on another class C2, in FoCaLiZe we must define the species C2 (derived from the class C2) before species C1 (derived from the class C1).

To respect dependency order, we generate a new XMI format (from the original one) in which we order nodes (each node in XMI format represents a class) according to their dependencies. For this, we have developed an XSLT style sheet that specifies the transformation rules to transform a XMI file original into new XMI format (orderly format).

To generate this new XMI format, we follow this Algorithm:

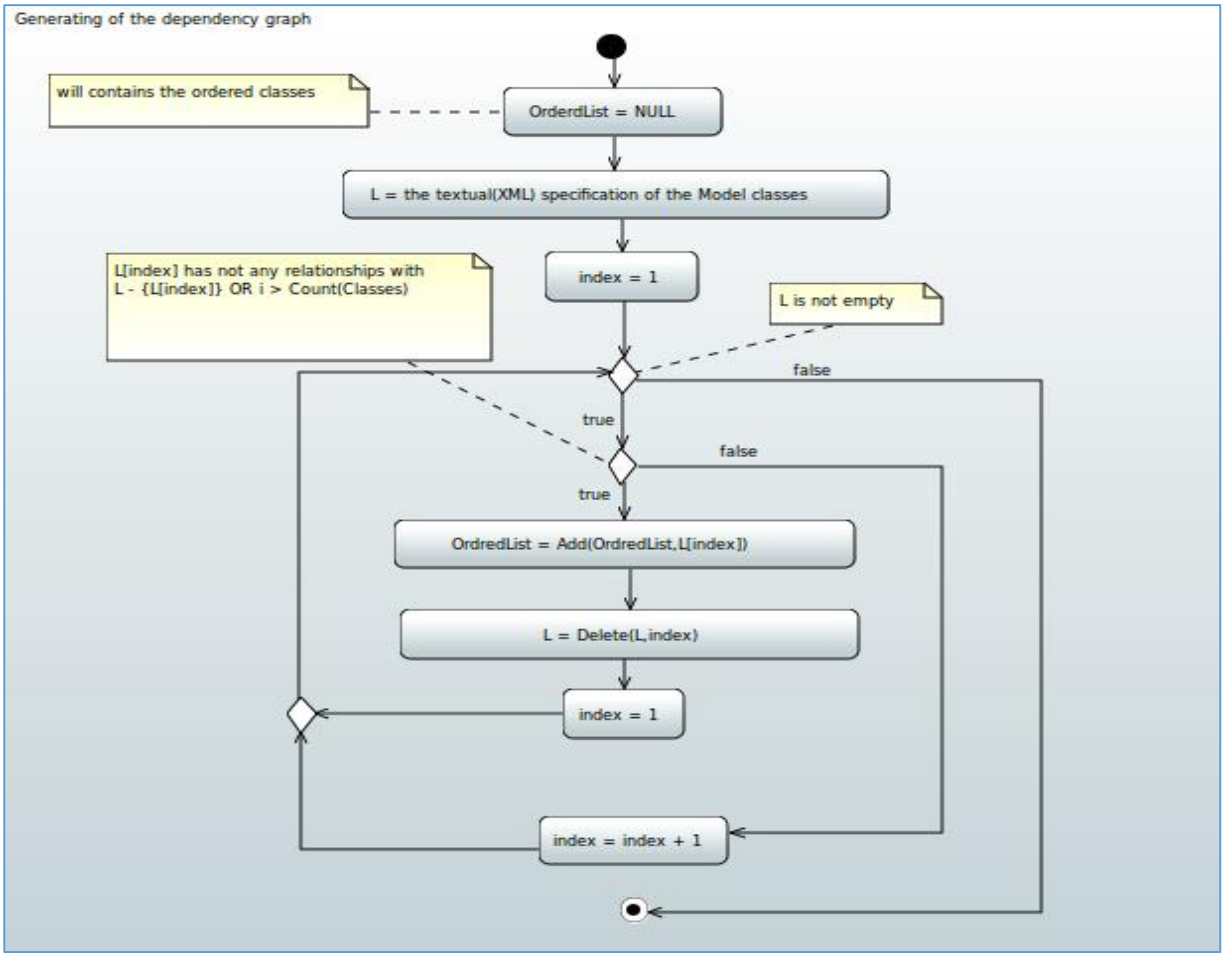
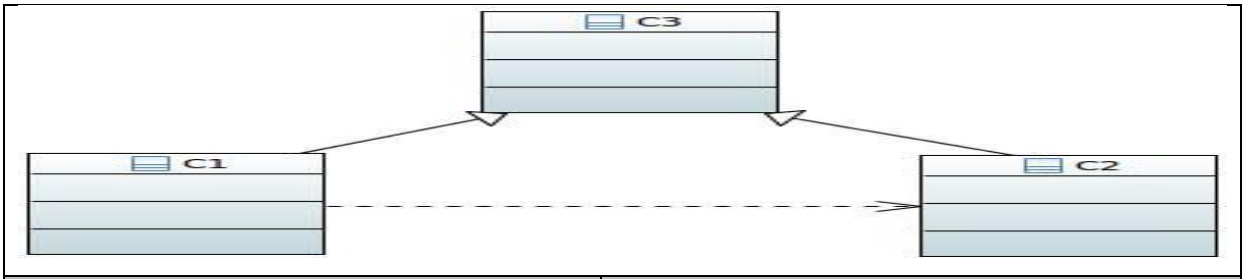


Figure 5.4: Generating of the dependency graph

The next table shows an example of the generation of an XMI ordered format using our XSLT stylesheet (“TrieUMLOrginal.xsl”).



Before the orders	After the orders
<!-- Class C1 --> <packagedElement xmi:type="uml:Class" xmi:id="_52BIkPTQEeWpG9AS1mkhkw" name="C1"> <generalization xmi:type="uml:Generalization" general="_5fDaE"/></packagedElement> <!-- Class C2 --> <packagedElement xmi:type="uml:Class" xmi:id="_6praIPTQE" name="C2"> <generalization xmi:type="uml:Generalization" general="_5fDaE"/></packagedElement> <!-- Class C3 --> <packagedElement xmi:type="uml:Class" xmi:id="_5fDaE" name="C3"/> <!--The relationship between C1 and C2 --> <packagedElement xmi:type="uml:Dependency" client="_52BIkPTQEeWpG9AS1mkhkw" supplier="_6praIPTQE"/>	<!-- Class C3 --> <packagedElement xmi:type="uml:Class" xmi:id="_5fDaE" name="C3"/> <!-- Class C2 --> <packagedElement xmi:type="uml:Class" xmi:id="_6praIPTQE" name="C2"> <generalization xmi:type="uml:Generalization" xmi:id="_vazRcPTREeWpG9AS1mkhkw" general="_5fDaE"/> </packagedElement> <!-- Class C1 --> <packagedElement xmi:type="uml:Class" xmi:id="_52BIkPTQEeWpG9AS1mkhkw" name="C1"><generalization xmi:type="uml:Generalization" xmi:id="_k5ZIMPTREeWpG9AS1mkhkw" general="_5fDaE"/></packagedElement> <packagedElement xmi:type="uml:Dependency" xmi:id="_LVmk4PTREeWpG9AS1mkhkw" client="_52BIkPTQEeWpG9AS1mkhkw" supplier="_6praIPTQE"/>

Table 5.1: The generation of Xml ordered format

5.2.3. Transformation

The last step consist to apply transformation rules (see Chapter 4) to generate a FoCaLiZe source. We have developed an XSLT stylesheet that specifies the transformation rules to transform OCL constraints into FoCaLiZe.

For the transformation of class diagrams, we use the transformation rule proposed by [11] last year. We note that we have improver the transformation of class diagrams by the consideration of UML templates and template binding.

5.2.3.1. Template binding transformation

The template binding transformation is already handled in [11] but we improved the types of substitutions parameters to support Basic_object type in addition to primitive types (litteralInteger, litteralString, litteralBoolean, litteralUnlimitedNatural and litteralReal).

Focalize supports the primitive types (int, string, bool, nat and float) but we can't use them directly as parameters to parametrize species, because they are in specification level. So, we have implement them in collections, in order to use them as entity species parameters. The following example show presents the implementation of the primitive type with a collection:

```
(*Create Type Collection Int*)
 species Int = inherit Basic_object;
 representation = int;
 let create (x:int):Self = x;
 let to_int(x:Self):int = x;
 end;;
collection IntCollection = implement Int; end;;
```

Note: The implementation of the other primitive types are presented in **appendix A**.

An example of UML template and template banding transformation is the following:

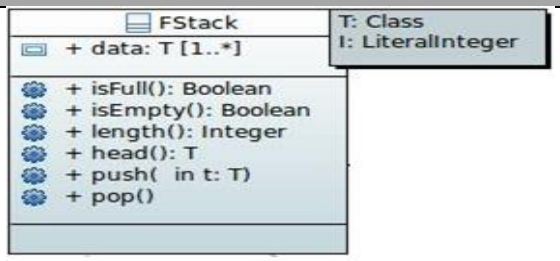
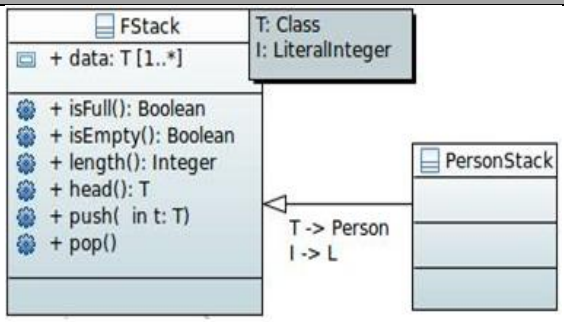
Parametrized class	**Parametrized species**
	species FStack(T is Basic_object, i in IntCollection) = inherit Basic_object . . end;;
Template Binding	**Parameter substitution**
	Let l = IntCollection!create(100); species PersonStack(T is Person, i in IntCollection) = inherit(T, l) . . end;;

Table 5.2: Templat and template binding transformation

5.2.3.2. OCL constraints

To implement the OCL constraint into FoCaLiZe, we follow the next Algorithms:

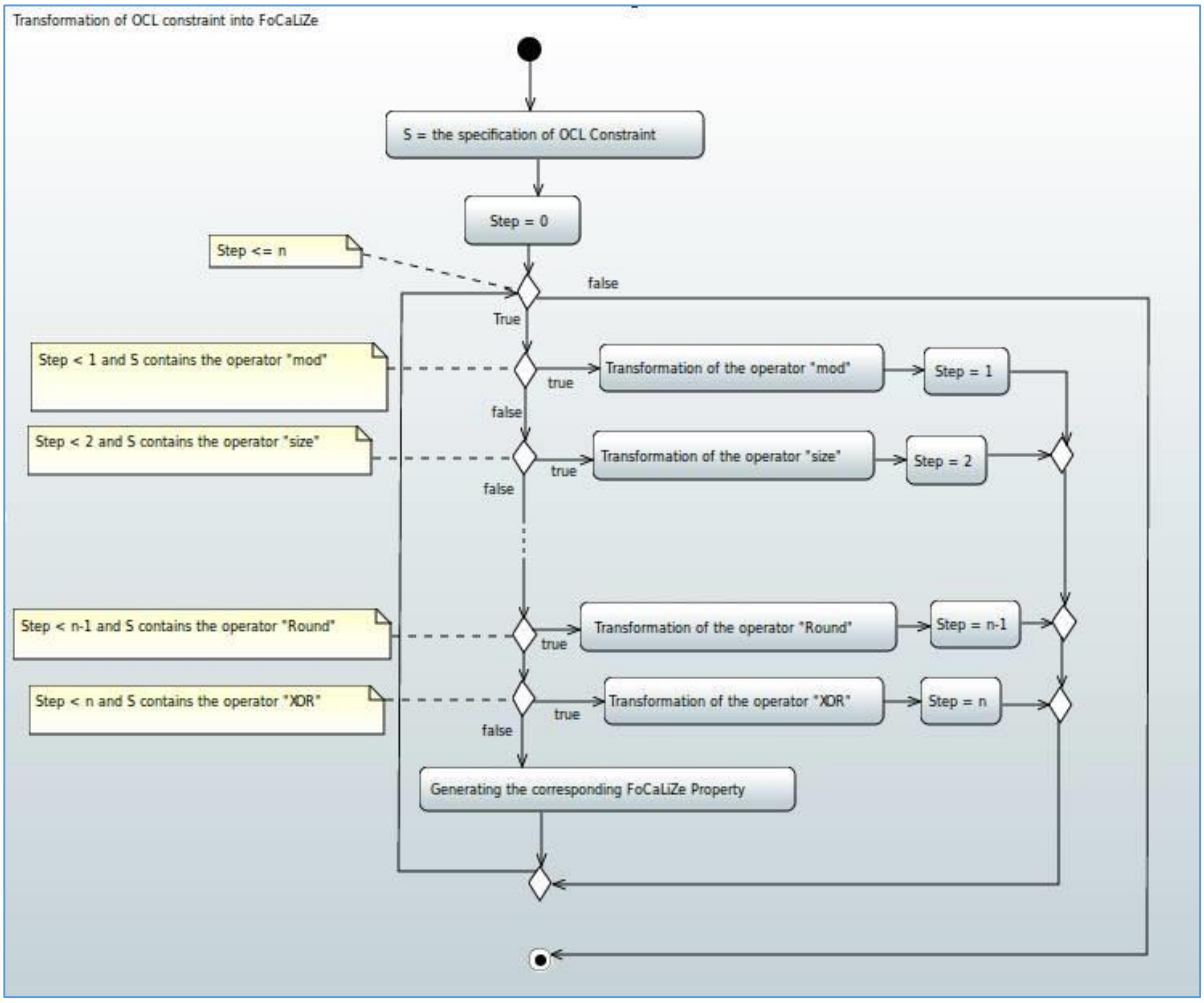


Figure 5.5: Transformation of OCL constraint into FoCaLiZe

5.3. Transformation Example

To illustrate our transformation, we have constructed a UML class diagram with OCL constraints containing all sorts of UML/OCL features: inheritance, template, template banding and OCL constraints, invariants and pre/post condition (See section 2.3, page 43).

- ✓ **Step 1**

  The graphic modeling of an example of a class diagram & OCL constraint (see figure 2.14) as well as its textual specification are given in parallel in section 5.2.1.

✓ **Step 2**

This step generating the ordered XMI document, the next table shows the restructure classes before and after the process of arrangement:

Before transformation process	After transformation process
1. PersonStack 2. FStack 3. Person	1. FStack 2. Person 3. PersontStack

✓ **Step 3**

Now, we apply our transformation rules to generate the FoCaLiZe code:

```
open "basics" ;;
open "float_func" ;;
open "String_func";;
open "TypeCollection";;

(* species FStack *)
species  FStack(T is Basic_object, i in IntCollection) = inherit Basic_object ;
  signature data : Self ->  list(T) ;
  signature isFull : Self ->     Bool;
  signature isEmpty : Self ->    Bool;
  signature length : Self -> int;
  signature head : Self -> T;
  signature push : Self -> T -> Self;
  (* property Pre Post Condition de l'Operation push *)
  property pre_post_push : all t : T ,  all __e : Self ,
    ( isEmpty(__e)) -> (~~( isEmpty(push(__e , t))));
  signature pop : Self -> Self;
  (* property Pre Post Condition de l'Operation pop *)
  property pre_post_pop :  all __e : Self ,
    ( isFull(__e)) -> (~~( isFull(pop(__e))));
  (* property invariant *)
  property inv_1 :
    all __e : Self, isEmpty(__e) -> length(__e) = 0 ;
  property inv_2 :
```

```
    all __e : Self, isEmpty(__e) -> ~~( isFull(__e)) ;
signature new_FStack : list(T)  -> Self  ;
end;;
(* species Person *)
species  Person = inherit Basic_object ;
  signature name: Self -> String;
  signature age: Self -> int;
  signature set_name : Self -> String -> Self;
  signature set_age : Self -> int -> Self;
  (* property invariant *)
  property inv_3 :
    a= ll __e : Self, age(__e) >0x 0 /\ age(__e) <0x 100 ;
signature new_Person :String -> int -> Self  ;
end;;
(* species PersonStack *)
let l = IntCollection!create(100);;
species  PersonStack(T is Person, i in IntCollection) = inherit FStack(T, l);
end;;
```

Table 5.3: The FoCaLiZe source

Compilation of FoCaLiZe code:

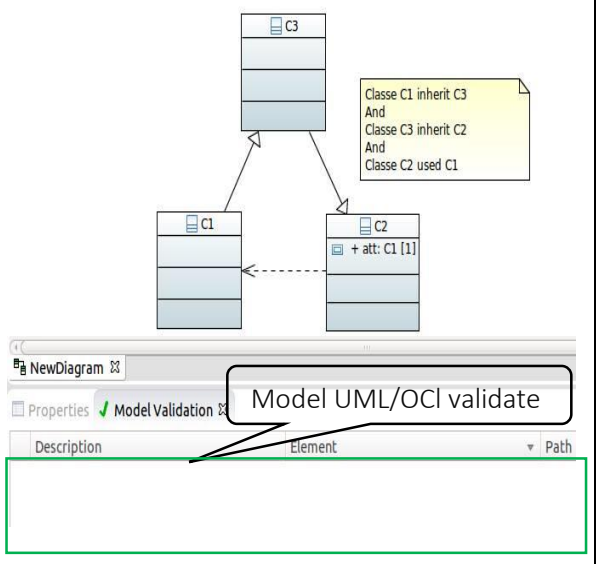
Finally, we compile the generated code using FoCaLiZe compiler (focalizec):

```
Result Run Foclize :------------------------
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/FoCaLiZe -c /home/vpoint/workspace/Test12/model2.ml
Invoking zvtov...
>> zvtov -zenon zenon -new  /home/vpoint/workspace/Test12/model2.zv
Invoking coqc...
>> coqc  -I /usr/local/lib/FoCaLiZe  -I /usr/local/lib/zenon
/home/vpoint/workspace/Test12/model2.v
```

Table 5.4: Compilation of FoCaLiZe code

No errors means that the compilation is succeed.

In the following example, we present the transformation of incorrect UML/OCL model, which leads an error during the compilation of the generated FoCaLiZe code:

<table>
<tr><th>UML/OCl Model</th><th>FoCaLiZe Code</th></tr>
<tr><td> 
</td><td>

```
1
2 open "basics" ;;
3 open "float_func" ;;
4 open "string_func";;
5 open "TypeCollection";;
6
7 (* species C3 *)
8
9 species  C3 = inherit C2;
10 end;;
11 (* species C1 *)
12 species  C1 = inherit C3;
13 end;;
14 (* species C2 *)
15 species  C2 = inherit Basic_object ;
16   signature att: Self -> C1 ;
17 signature new_C2 :C1 -> Self  ;
18 end;;
19
```

</td></tr>
<tr><th colspan="2">compilation</th></tr>
<tr><td colspan="2">

```
Console
Error Execute File Foclize :------------------
File "/home/vpoint/workspace/Test12/model4.fcl", line 9, characters 22-24:
Error: Unbound species 'C2'.
```

</td></tr>
</table>

Table 5.5: Error in compilation of FoCaLiZe code

5.4. Setup our transformation tool

The installation of our plugin will setup 2 components:

- ✓ Eclipse Environment with Papyrus plugin.
- ✓ FoCaLiZe compiler.

Installation Tools

Step 1:

To install Papyrus plugin see the following web site:

https://wiki.eclipse.org/Papyrus-RT/User_Guide/Installation

Step 2:

The installation of FoCaLiZe (http://FoCaLiZe.inria.fr/download/) requires the following external tools:

- ✓ OCaml (any recent version -- >= 3.11 --) should be fine.

  Install #sudo apt-get install ocaml

- ✓ Coq (any recent version -- >= 8.1pl5 --) should be fine.

  Install #sudo apt-get install coq

- ✓ Zenon (either download an archive or get it from the GIT repository by Invoking : get clone http://FoCaLiZe.inria.fr/zenon.git

  In the root directory of FoCaLiZe (usually "FoCaLiZe")).

Quick Installation:

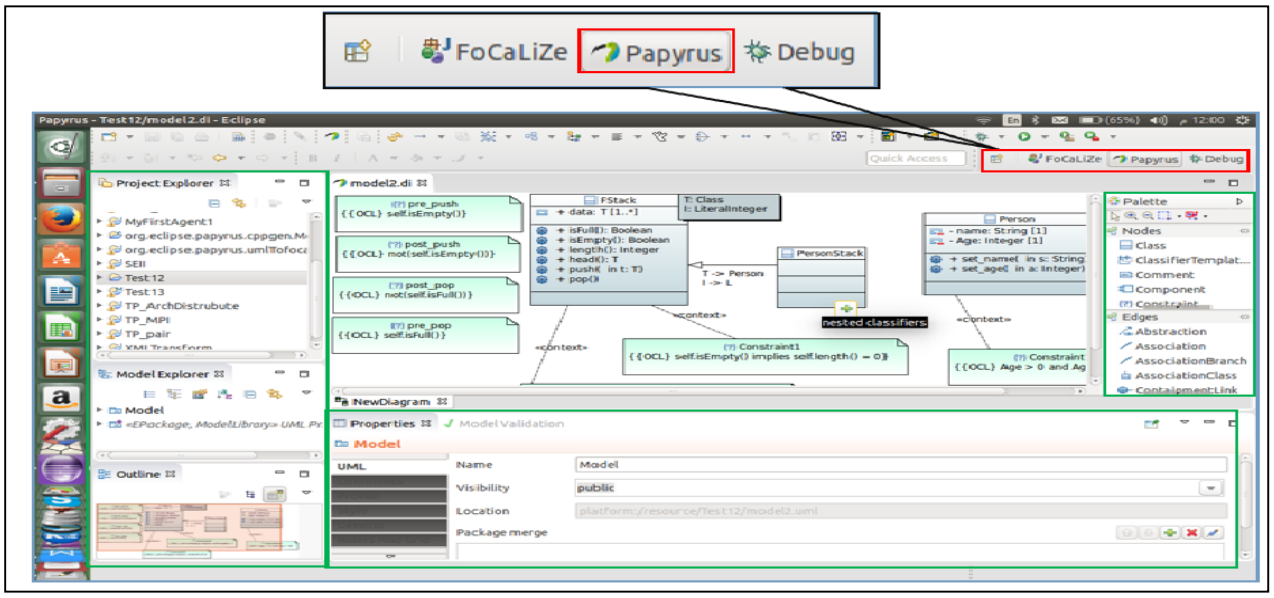
For hurry people, assuming default installation settings/directories from the top level 'FoCaLiZe' directory run: ***make all***

Step 3:

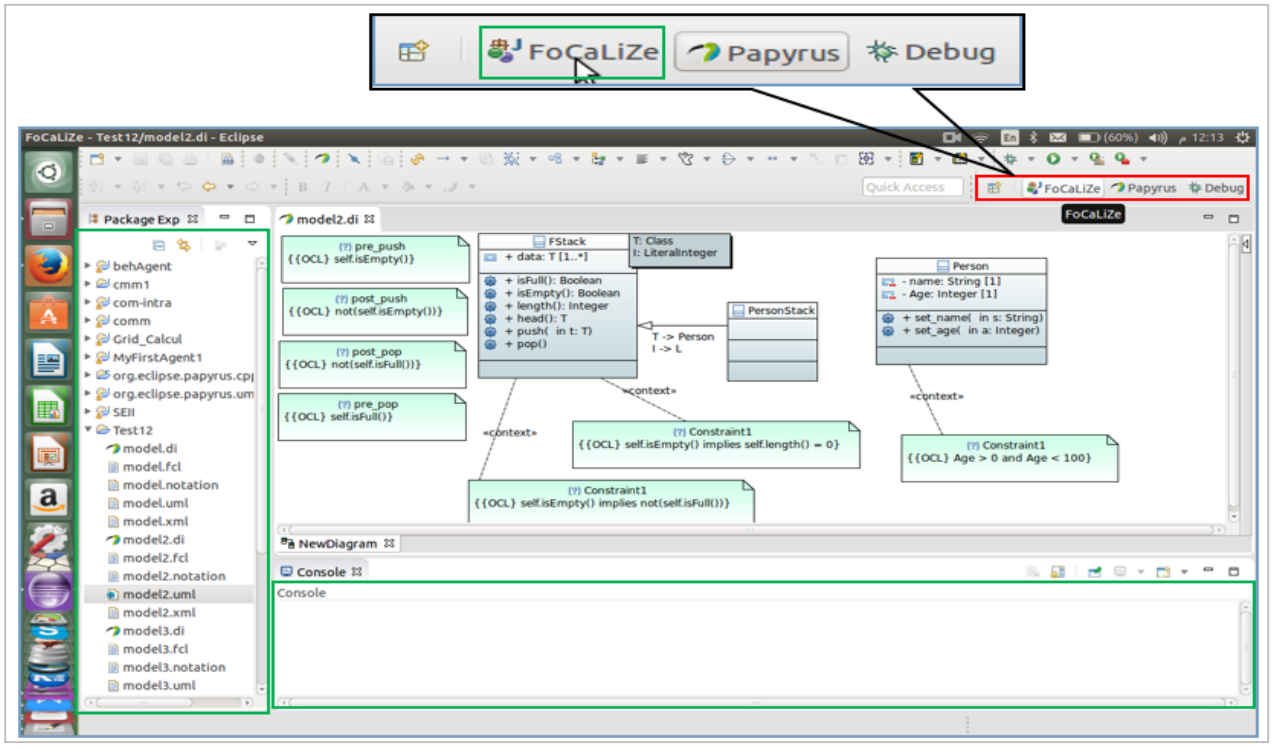
- ✓ Close eclipse environment
- ✓ Add plugin **'umlToFoCaLiZe' in path 'eclipse/plugins/'**
- ✓ Add files XSLT **('TrieUMLOrginal.xsl' and 'umlTofoclize.xsl') in path 'eclipse/'**
- ✓ open eclipse

5.5. How to use the transformation tool

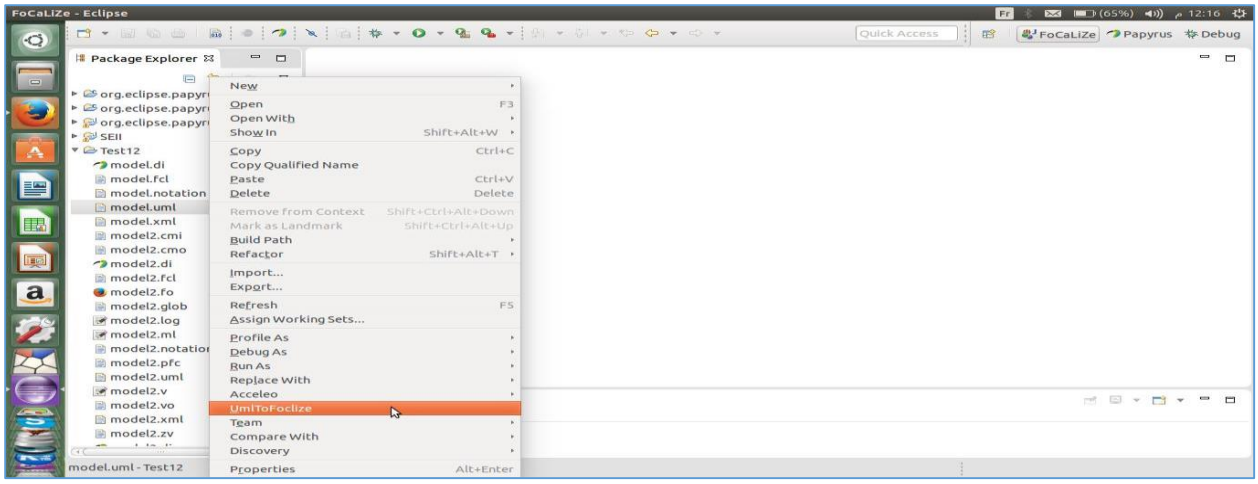
Press perspective Papyrus For create a UML/OCL Model



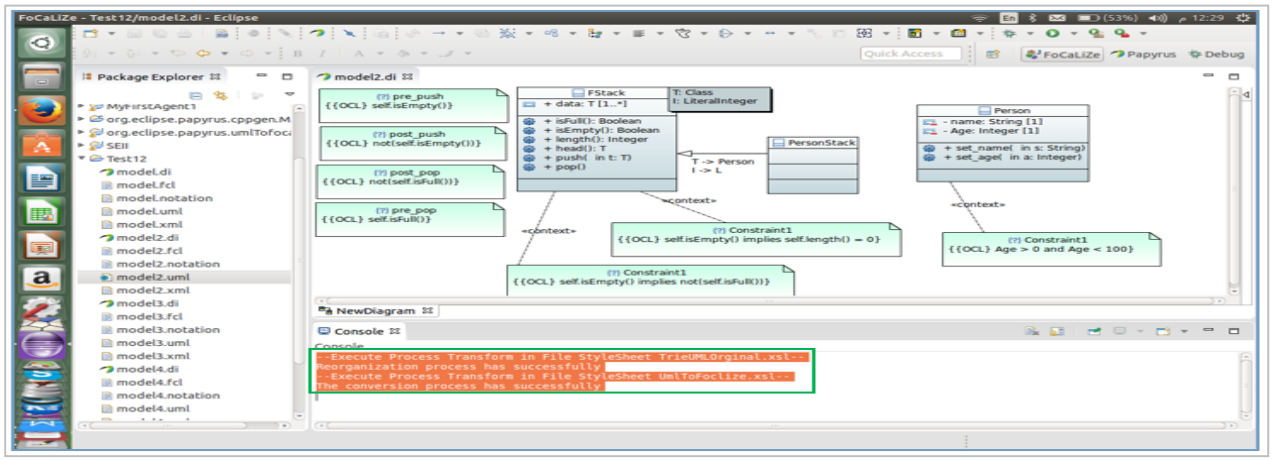
Press Perspective FoCaLiZe :



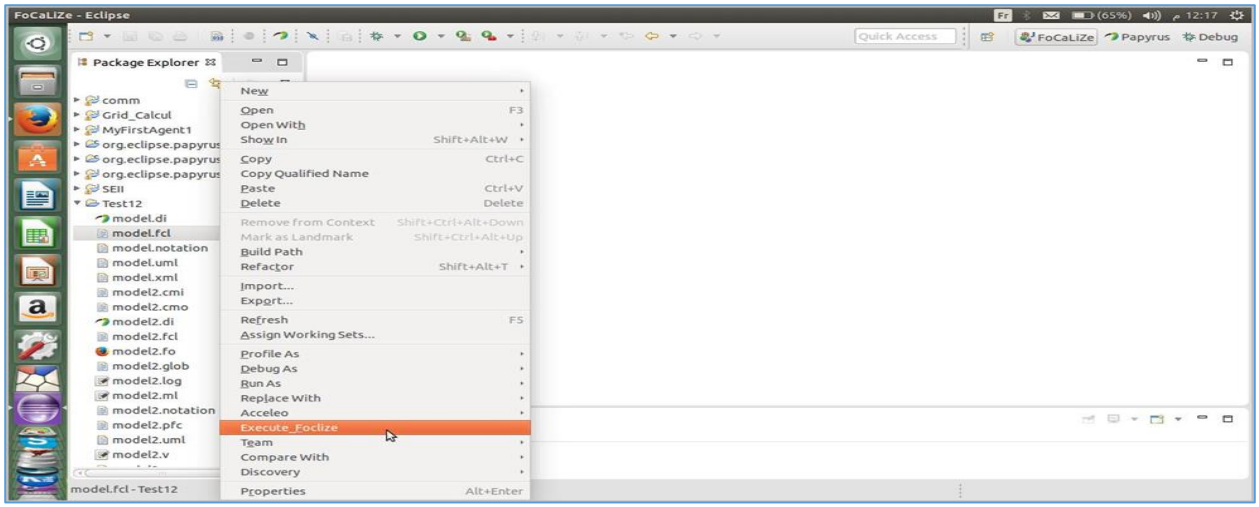
Right click on '**modelname.uml'**, then choose the command '**UmlToFoCaLiZe**'.



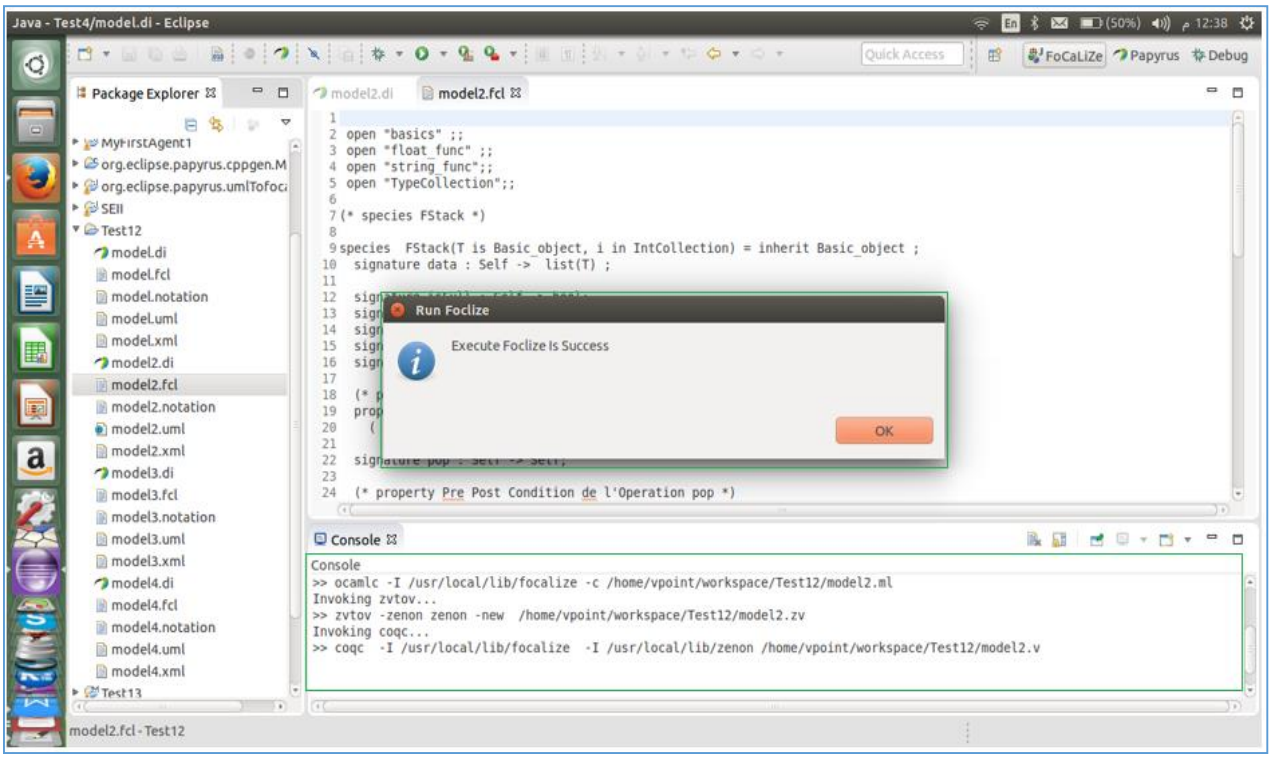
The result of the transformation will appear in the consol.



After the derivation of FoCaLiZe code, press the F5 or refresh, then right click the file '**modelname.fcl**', then choose the command '**ExecuteFoCaLiZe**' to invoke the FoCaLiZe compiler.



the result of compilation is shown at the console.



5.6. Conclusion

In this chapter we have presented a process for the transformation of UML/OCL models into FoCaLiZe. This process is realized within a framework grouping a UML/OCL graphical tool, the FoCaLiZe environment and our transformation rules. The proposed approach enables a systematic transformation of UML/OCL model into FoCaLiZe. Then, a FoCaLiZe developer can

complete the obtained abstract specification and provide proofs for its properties until reach the executable code. An implementation of the transformation process is performed by the development of an XSLT style sheet describing the transformation rules.

General Conclusion

In this master thesis, we have established another alternative for the transformation of OCL constraints, using the FoCaLiZe environment. In this transformation, we have taken into account OCL invariants and pre/post-condition specified on UML classes and classes operations, where we consider UML features ignored in similar works. In particular, from a UML model annotated with OCL constraints, we have been able to generate a UML/OCL textual specification, then the elements of the textual specification obtained are arranged following their syntactic and semantic dependencies. After that, we apply the proposed transformation rules to produce a FoCaLiZe specification.

In the proposed transformation, the classes correspond to species, the attributes of classes are transformed into signatures modelling their getters, the operations of classes are converted into signatures specifying their functional types and OCL constraints of a class are mapped into properties in the corresponding species.

Concretely, we supported the following UML/OCL features in our transformation approach:

- The multiple inheritance: we derive a hierarchy of species which reflects the inheritance relationships between classes. The mechanisms of the redefinition of methods and of late binding are also preserved between the derived species.

- The parametrized classes: a parameterized class is formalized by parametrize species.

- Template Binding: This relationship is formalized through the substitution of the formal parameters of a species by actual parameters.

- The dependency between classes: The dependency between two classes (client and supplier) is formalized by the parametrization of the species client (derived from the client class) by the supplier species (derived from the supplier class).

- Propagation of the OCL constraints through the mechanism of inheritance: the properties of a super-species are also properties of its sub-species.

- Propagation of the OCL constraints through parametrization and the dependency: The properties (of a species) derived from OCL constraints are propagated through the

mechanism of parameter substitution in FoCaLiZe. The properties of a supplier species (used as formal parameter) can be used safely by client species (parameterized species).

• Propagation of the OCL constraints through the parameters substitution: The properties of a species derived from a parameterized class (UML template) are also properties of its bound models.

In this work, we have only dealt with the main OCL constraints: invariant, pre-condition and post-condition. We would like to expand our mapping to deal with a larger subsets of OCL. In particular, we will deal with the constraint derive which specifies the value of a derived attribute or association role, and with the type Collection and its sub-types which are Set, OrderedSet, Bag and Sequence.

Appendix A

A.1. Definition of OCL operations on the type Real

```
open "basics" ;;

 (* real addition *)

let ( +f ) =
  internal float -> float -> float
  external
  | caml ->  {*(+.)*}
  | coq -> {* Rplus *}
;;

(* real subtraction *)

let ( -f ) =
  internal float -> float -> float
  external
  | caml ->  {*(-.)*}
  | coq -> {* Rminus *}
;;

  (* real multiplication *)

let ( *f ) =
  internal float -> float -> float
  external
  | caml ->  {* ( *. )*}
  | coq -> {* Rmult *}
;;

  (* real division *)

let ( /f ) =
  internal float -> float -> float
  external
  | caml ->  {*( /. )*}
  | coq -> {* Rdiv *}
;;
```

```
  (* absolute value of a real number *)
let abs_f =
  internal float -> float
  external
  | caml ->  {* abs_float *}
  | coq -> {* Rbasic_fun.Rabs *}
;;
   (*comparison between real: the upper function *)
let ( >f ) =
  internal float -> float -> bool
  external
  | caml ->  {* function (x : float) -> function (y:float) -> x > y  *}
  | coq -> {* operation_float.float_gt *}
;;
 (*comparison between real: the lower function *)
let ( <f ) =
  internal float -> float -> bool
  external
  | caml ->  {* function (x : float) -> function (y:float) -> x < y *}
  | coq -> {* operation_float.float_lt *}
;;
 (*comparison between real: the Superior or equal function *)
let ( >=f ) =
  internal float -> float -> bool
  external
  | caml ->  {* function (x : float) -> function (y:float) -> x >= y *}
  | coq -> {* operation_float.float_ge *}
;;
(*comparison between real: the inferior or equal function *)
let ( <=f ) =
  internal float -> float -> bool
  external
  | caml ->  {* function (x : float) -> function (y:float) -> x <= y *}

  | coq -> {* operation_float.float_le *}
;;

(* the maximum of two real *)
let  max_f  =
  internal float -> float -> float
  external
  | caml ->  {* function (x : float) -> function (y:float) -> max x y *}
  | coq -> {* Rbasic_fun.Rmax *}
;;
```

```
(* the minimum of two real *)
let  min_f  =
  internal float -> float -> float
  external
  | caml ->  {* function (x : float) -> function (y:float) -> min x y *}
  | coq -> {* Rbasic_fun.Rmin *}
;;

 (*returns integer less than or equal to the value given in parameter *)

let floor_f =
  internal float -> int
  external
  | caml ->  {* function (x : float) -> int_of_float(floor(x)) *}
  | coq -> {* R_Ifp.Int_part *}
;;

(*returns the smallest integer greater than or equal to the value given in parameter *)

let round_f =
  internal float -> int
  external
  | caml ->  {* function (x : float) -> int_of_float(ceil(x)) *}
  | coq -> {* operation_float.Rround *}
;;

(*convert integer type to float  *)

let intTofloat =
  internal int -> float
  external

  | caml ->  {* float_of_int *}
  | coq -> {* IZR *}
;;

(*convert integer type to int  *)

let floatToint =
  internal float -> int
  external
  | caml ->  {* int_of_float *}
  | coq -> {* R_Ifp.Int_part *}
;;
```

```
  (*sqrt value of a real number *)

let sqrt =
  internal float -> float
  external
  | caml ->  {* sqrt *}
  | coq -> {* R_sqrt.sqrt *}
;;
```

A.2. Definition of the OCL operations on the type String

```
open "basics";;
(*Length String*)
let length=
  internal string -> int
  external
 | caml ->  {* String.length *}
 | coq -> {* operation_string.length *}
;;
(*subString  *)
let substring=
  internal string -> int -> int -> string
  external
 | caml ->  {* function (s : string) -> function (lower : int) -> function (upper : int) -> String.sub
s  (lower-1)  (upper-lower+1) *}
 | coq -> {* operation_string.substring *}
;;
(*Upper Case String *)
let upper_case=
  internal string -> string
  external
 | caml ->  {* String.uppercase *}
 | coq -> {* operation_string.upper_case *}
;;
```

```
(*Lower Case String *)
let lower_case=
  internal string -> string
  external
 | caml ->  {* String.lowercase *}
 | coq -> {* operation_string.lower_case *}
;;
(*Convert string to float*)
let float_of_string=
  internal string -> float
  external
 | caml -> {* float_of_string *}
 | coq -> {* fun (x : string) => 42%R *}
;;
```

A.3. Definition of collection on primitive types

```
open "basics";;

(*Create Type Collection Boolean*)

species Bool = inherit Basic_object;

representation = bool;

let create (x:bool):Self = x;

let to_int(x:Self):bool = x;

end;;

collection BoolCollection = implement Bool; end ;;

(* Create Type Collection Int*)

species Int = inherit Basic_object;

representation = int;

let create (x:int):Self = x;

let to_int(x:Self):int = x;

end;;
```

```
collection IntCollection = implement Int; end ;;

(*Create Type Collection Real*)

species Real = inherit Basic_object;

representation = float ;

let create (x:float):Self = x ;

let to_int(x:Self):float = x;

end;;

collection RealCollection = implement Real; end;;

(*Create Type Collection String*)

species String = inherit Basic_object;

representation = string;;

let create (x:string):Self = x;

let to_int(x:Self):string = x;

end;;

collection StringCollection = implement String; end;;
```

References

[1] OMG.: UML : Superstructure, version 2.4 (2011). Available at: http://www.omg.org/spec/UML/ 2.4/Infrastructure.

[2] OMG: OCL : Object constraint language 2.3.1 (2012). Available at: http://www.omg.org/spec/ OCL.

[3] Team, F.D.: FoCaLiZe : Tutorial and reference manual, version 0.8.0. CNAM/INRIA/LIP6 (2012). Available at: http://focalize.inria.fr

[4] Facon, P., Laleau, R.: Des spécifications informelles aux spécifications formelles : compilation ou interprétation ? Actes du 13éme congrès INFORSID (1995)

[5] Ayrault, P., Hardin, T., Pessaux, F.: Development life-cycle of critical software under FoCaL. Electronic Notes in Theoretical Computer Science 243, 15–31 (2009)

[6] Fechter, S.: Sémantique des traits orientes objet de FoCaL. Ph.D. thèses, Université PARIS 6 (2005)

[7] Delahaye, D., Etienne, J.F., Donzeau-Gouge, V.V.: 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, pp. 121–124. IEEE (2008)

[8] Doligez, D.: "the Zenon tool" Software and documentations freely available at http://focal.inria.fr/zenon/

[9] Coq.: The Coq proof assistant, Tutorial and reference manual. INRIA – LIP – LRI – LIX – PPS (2010). Distribution available at: http://coq.inria.fr/ .

[10] Damien Doligez, Mathieu Jaume, Renaud Rioboo. Development of secured systems by mixing programs, specifications and proofs in an object-oriented programming environment. A case study within the FoCaLiZe environment. PLAS - Seventh Workshop on Programming Languages and Analysis for Security, Jun 2012, Beijin, China. <hal 0077365>

[11] MENACEUR Khadija, GHENDIR MABROUK Nacira. Hama Lakhdar University-El-oued, Automatic transformation tool of UML class diagrams into FoCaLiZe, Jun 2015.

[13] Thérèse Hardin, François Pessaux, Pierre Weis, Damien Doligez. FoCaLiZe Reference Manual.version 0.9.0. (October 2014)

[14] http://www.sparxsystems.com/resources/uml2_tutorial/uml2_classdiagram.html , Current Release: Version 12.1, Build 1229 /16-March-2016

[15] http://www.classdraw.com/help.htm. (2005-2011)

[16] OMG. UML: Superstructure, version 2.4. January 2011. Available at: http://www.omg.org/spec/UML/2.4/Infrastructure .

[17] https://wiki.eclipse.org/Acceleo/OCL_Operations_Reference , This page was last modified 07:34, 23 October 2015 by Stephane Begaudeau. Based on work by Frederic Madiot

[18] Pierre-Alain Muller and Nathalie Gaertner. Modélisation objet avec UML, volume 514. Eyrolles Paris, 2000.

[19] OMG: OCL: Object constraint language Version 2.0 (formal/06-05-01). Available at: http://www.omg.org/spec/ OCL.

[20] Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (2005)

[21] Jackson, D.: Alloy 3.0 reference manual. Software Design Group (2004)

[22] Jackson, D.: Software Abstractions: Logic, Language and Anlysis. MIT press (2012)

[23] Clavel, M., Duran, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J., Talcott, C.: All about Maude a High Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic. Springer-Verlag (2007)

[24] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-order Logic. Volume 2283. Springer (2002)

[25] Ke, W., Li, X., Liu, Z., Stolz, V.: rCOS: A formal model-driven engineering method for component-based software. Frontiers of Computer Science 6(1) (2012) 17–39

[26] Mosses, P.: CASL reference manual: The complete documentation of the common algebraic specification language. Volume 1. Springer (2004)

[27] Truong, N., Souquieres, J., et al.: Validation des propriétés d'un scénario UML/OCL à partir de sa dérivation en B. Proc. Approches Formelles dans l'Assistance au développement de Logiciels, France (2004)

[28] Truong, N., J., S.: Verification of UML model elements using B. Information Science and Engineering (22) (2006) 357–373

[29] Ledang, H., Souquieres, J., Charles, S., et al. Argouml+ B: Un Outil de Transformation Systématique de Spécifications UML en B. In: Approches Formelles dans l'Assistance au Développement de Logiciels. (2003)

[30] Hazem, L., Levy, N., Marcano-Kamenoff, R.: UML2B : Un outil pour la génération de modelés formels. AFDL (2004)

[31] Snook, C., Butler, M.: UML-B : Formal modeling and design aided by UML. ACM Transactions on Software Engineering and Methodology (15) (2006) 92–122

[32] Cunha, A., Garis, A., Riesco, D.: Translating between Alloy Specifications and UML Class Diagrams Annotated with OCL Constraints. Software & Systems Modeling (2013) 1–21

[33] Anastasakis, K., Bordbar, B., Georg, G., Ray, I. : UML2Alloy: A Challenging Model Transformation. In: Model Driven Engineering Languages and Systems. Springer (2007) 436–450

[34] Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software & Systems Modeling 9(1) (2010) 69–86

[35] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. URL http://www4.in.tum.de/~nipkow/LNCS2283/. (Cited on page 19.)

[36] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940. (Cited on page 17.)

[37] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Computer Science and Applied Mathematics. Academic

Press, Orlando, May 1986. ISBN 0120585367. URL http://gtps.math.cmu.edu/andrews.html . (Cited on pages 13 and 17.)

[38] Brucker, A.: An interactive proof environment for object-oriented specifications, phd thesis. ETH, Zurich (2007)

[39] Brucker, A., Wolff, B.: The HOL-OCL tool. (2007) http://www.brucker.ch/ .

[40] Duran, F., Gogolla, M., Roldan, M.: Tracing properties of UML and OCL models with Maude. arXiv preprint arXiv:1107.0068 (2011)

[41] Mokhati, F., Sahraoui, B., Bouzaher, S., Kimour, M.T.: A Tool for Specifying and Validating Agents' Interaction Protocols: From Agent UML to Maude. Object Technology 9(3) (2010)

[42] https://en.wikipedia.org/wiki/Eclipse_(software) , This page was last modified in 8 May 2016, at 15:52.

[43] On the Papyrus' USE: Usage, Specialization and Extension **,** PhD. Sébastien Gérard CEA LIST Senior Expert Laboratory of model driven engineering for embedded systems (LISE) 2010-09-06, https://eclipse.org/papyrus/documentation.html.

[44] **F**. Jouault, F. Allilaire, J. B_ezivin, and I. Kurtev, \ATL: a model transformation tool," Science of Computer Programming, vol. 72, pp. 31-39, June 2008.

[45] D. Kolovos, R. Paige, and F. Polack, \The epsilon transformation language," in Theory and Practice of Model Transformations, ser. Lecture Notes in Computer Science, A. Vallecillo, J. Gray , and A. Pierantonio, Eds.Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 5063، ch. 4, pp.46-60. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69927-9_4 .