Noname manuscript No.
(will be inserted by the editor)

A Formal and Sound Transformation from Focal to UML

An Application to Airport Security Regulations

David Delahaye · Jean-Frédéric Étienne · Véronique Viguié Donzeau-Gouge

the date of receipt and acceptance should be inserted later

Abstract We propose an automatic transformation of Focal specifications to UML class diagrams. The main motivation for this work lies within the framework of the EDEMOI project, which aims to integrate and apply several requirements engineering and formal methods techniques to analyze airport security regulations. The idea is to provide a graphical documentation of formal models for developers, and in the long-term, for certification authorities. The transformation is formally described and an implementation has been designed. We also show how the soundness of our approach can be achieved.

Keywords Formal Methods · Graphical Documentation · Focal · UML · Airport Security Regulations

1 Introduction

Even though formal methods offer a systematic approach for verification, the validation process still relies on a high degree of interaction between the various stake-holders (developers, customers, end-users, certification authorities, etc) involved in a critical project. In addition, the use of formal methods requires a certain level of expertise in mathematics, which usually hinders communication. In fact, the mathematical notations used are often too obscure for inexperienced users to properly understand the exact meaning. As a result, the validation of requirements is difficultly achievable. This may even jeopardize the entire project as misinterpretations or specification errors may lead to the validation of a totally wrong implementation.

A widely adopted solution to these problems is the integration of formal and graphical specifications. In general, the use of graphical notations is quite useful when interacting with end-users. In fact, these tend to be more intuitive and are easier to grasp than their formal (or textual) counterparts. During the last few years, UML [16] has emerged as a standard in industry for modeling software systems. It provides a set of graphical constructs, which enables the modeling of systems in an object-oriented style. Currently, it is supported by a wide variety of tools, ranging from analysis, testing, simulation to code generation and transformation. Interoperability between these tools is generally achieved by exporting the UML models using the XMI interchange format.

There have been several researches devoted to establishing the link between UML and formal methods. One of the approaches that has been largely studied is the translation of UML diagrams into formal specifications [7, 11, 12], which attempts to benefit from the formal methods tools and techniques while still having control over the UML-based industrial practice. The converse approach is a rather new area of interest [9]. It is here considered to generate UML models as a means to provide a graphical documentation for Focal specifications.

The main motivation for this work lies within the framework of the EDEMOI[1] [14] project, which aims

D. Delahaye
CEDRIC/CNAM, Paris, France
E-mail: David.Delahaye@cnam.fr

J.-F. Étienne
CEDRIC/CNAM, Paris, France
E-mail: etiennje@cnam.fr

V. Viguié Donzeau-Gouge
CEDRIC/CNAM, Paris, France
E-mail: donzeau@cnam.fr

[1] The EDEMOI project is supported by the French National "Action Concertée Incitative Sécurité Informatique".

to integrate and apply several requirements engineering and formal methods techniques to analyze airport security regulations. For this project, we used Focal to realize the formal models of two regulations, namely the international standard Annex 17 and the European directive Doc 2320. The formalization is described in [3], while the certification part is presented in [4]. Within the project, the purpose of the UML diagrams is two-fold. First, to provide a graphical documentation of the formal models produced for developers. Second, to generate higher-level views of the formal models that would be more appealing to certification authorities.

For our concern, the choice of UML as a graphical notation mainly resides in the fact that most of the Focal design features can seamlessly be represented in UML. The creation of a domain specific language for Focal could be a better approach, as it avoids us from having to deal with the intricacies of the UML semantics. In fact, text-to-model tools [8], such as xText or TCS, generally facilitates such a process, whereby the target language is taken as input and the corresponding metamodel, parser and editor is generated as output. However, we still have to develop a graphical concrete syntax for each concept. The corresponding semantics might be intuitive to developers but not necessarily to end-users or certification authorities (which is our long-term objective). Finally, the choice for UML also allows us to have access to a wide variety of tools ranging from analysis to code generation and transformation. For instance, the UML models produced can be used to map Focal specifications to other object-oriented languages, e.g. Java or C#.

This paper is complementary to the work presented in [5] and completes the formal schema established for the translation of Focal specifications into UML diagrams. Here, the objective is to provide a graphical documentation for developers. Our major concern is not only to make our transformation automatic but also to prove the soundness of our approach. In this paper, we refine the abstract syntax proposed in [5] for a subset of the UML 2.1 static structure constructs [16]. The new syntax intends to facilitate reasoning. We here also consider all the different aspects of the Focal specification language and show how the UML metamodel can be tailored to consider some of its semantic specificities. We also describe how the soundness of our transformation can be achieved. This consists in showing that the UML model generated from a well-typed Focal specification preserves both the well-formedness rules of the UML metamodel and the constraints specified in the UML profile defined on purpose. Through this work, we also contribute to the formalization of the semantics relative to the template binding construct.

The paper is organized as follows: first, we present the Focal specification language; next, we propose a formal description for a subset of the UML 2.1 static structure constructs; we then show how the UML metamodel can be extended via the profile mechanism (lightweight extension) to cater for the semantic specificities of the Focal language; we afterwards formally describe our transformation rules and expose how the soundness of our approach can be achieved; finally, we introduce our implementation and illustrate our transformation with a concrete example.

2 The Focal Environment

2.1 What is Focal?

Focal[2] [6, 15], initiated by T. Hardin and R. Rioboo with S. Boulmé, is a language in which it is possible to build certified applications step by step, going from abstract specifications, called species, to concrete implementations, called collections. These different structures are combined using inheritance and parameterization, inspired by object-oriented programming. Moreover, each of these structures is equipped with a carrier set, providing a typical algebraic specification flavor. Next, V. Prevosto developed a compiler for this language, able to produce OCaml code for execution, Coq code for certification, but also FocDoc code [13] for documentation. More recently, D. Doligez provided a first-order automated theorem prover, called Zenon [1], which helps the user to complete his/her proofs in Focal through a declarative-like proof language. This automated theorem prover can produce pure Coq proofs, which are reinserted in the Coq specifications generated by the Focal compiler and fully verified by Coq.

2.2 Specification: Species

The first main notion of the Focal language is the structure of *species*, which corresponds to the highest level of abstraction in a specification. A species can roughly be seen as a list of attributes of three kinds:

- 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 functions, which denote the operations allowed on the entities of the representation; the functions can be either *definitions* (when a body is provided) or *declarations* (when only a type is given);

[2] `http://focal.inria.fr/`.

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

The syntax of a species is the following:

```
species <name> =

  rep [= <type>]; (* abstract/concrete
                     representation *)

  sig <name> in <type>; (* declaration *)
  let <name> = <body>;  (* definition *)

  property <name> : <prop>; (* property *)
  theorem <name> : <prop>   (* theorem *)
  proof : <proof>;

end
```

where <name> is simply a given name, <type> a type expression (mainly typing of core-ML without polymorphism but with concrete data types), <body> a function body (mainly core-ML with conditional, pattern-matching and recursion), <prop> a (first-order) proposition and <proof> a proof (expressed in a declarative style and given to Zenon). In the type language, the specific expression "self" refers to the type of the representation and may be used everywhere except when defining a concrete representation.

As said previously, species can be combined using (multiple) inheritance, which works as expected. It is possible to define functions that were previously only declared or to prove properties which had no provided proof. It is also possible to redefine functions previously defined or to reprove properties already proved. However, the representation cannot be redefined and functions as well as properties must keep their respective types and propositions all along the inheritance path. Another way of combining species is to use parameterization. Species can be parameterized either by other species or by entities from species. If the parameter is a species, the parameterized species only has access to the interface of this species, i.e. only its abstract representation, its declarations and its properties. These two features complete the previous syntax definition as follows:

```
species <name> (<name> is <name>[(<pars>)],
                <name> in <name>, ...)
        inherits <name>, <name> (<pars>),
                 ... = ...
end
```

where <pars> is a list of <name>, which denotes the names used as effective parameters. When the parameter is a species parameter declaration, the "is" keyword is used. When it is an entity parameter declaration, the "in" keyword is used.

2.3 Implementation: Collection

The other main notion of the Focal language is the structure of *collection*, which corresponds to the implementation of a specification. A collection implements a species in such a way that every attribute becomes concrete: the representation must be concrete, functions must be defined and properties must be proved. If the implemented species is parameterized, the collection must also provide implementations for these parameters: either a collection if the parameter is a species or a given entity if the parameter denotes an entity of a species. Moreover, a collection is seen (by the other species and collections) through its corresponding interface; in particular, the representation is an abstract data type and only the definitions of the collection are able to manipulate the entities of this type. Finally, a collection is a terminal item and cannot be extended or refined by inheritance. The syntax of a collection is the following:

```
collection <name> implements <name>
           (<pars>) = ... end
```

3 UML Syntax

In order to establish a formal framework for our transformation, we propose in [5] an abstract syntax for a subset of the UML 2.1 static structure constructs [16]. The syntax was mainly derived from the UML 2.1/XMI schema to reflect as much as possible our implementation. In this section, we present a new syntax that hides some of the complexities inherent to the UML metamodel and thus less dependent on the XMI format. This not only allows us to increase the readability of our transformation rules but also to facilitate reasoning. Another less tedious approach can be to make use of a text-to-model tool [8], e.g. xText or TCS, to obtain a metamodel of the Focal specification language instead of defining an abstract syntax for UML. The automatic transformation from Focal to UML may then be realized at a metamodel level through the use of a model-to-model transformation language [10], such as ATL or QVT. Nevertheless, even though such an approach can be considered during the implementation phase, it does not allow us to prove the soundness of our transformation.

The new syntax is shown in Figure 1 and is given using a BNF-like notation, where: terminal symbols are written in bold, while non-terminal ones are in italic; square brackets [...] are used to denote optional components, while curly brackets {...} denote grouping; a trailing star sign * denotes zero, one or several occurrences, while a trailing plus sign $^+$ denotes one or several occurrences; and the non-terminal symbol *ident* is used to designate the identifier of each nameable UML construct. In our syntax, anonymous bound classes are denoted using the same notation as defined for template bindings. The class type *class-type* is defined accordingly to reflect that these classes may also be referenced as type.

4 From Focal to UML

4.1 Extending the UML Metamodel

In order to properly visualize Focal models using UML notations, there is a need to extend the UML metamodel to cater for the semantic specificities of the Focal language. These extensions are realized through the creation of a profile, whereby appropriate stereotypes are defined to reflect the semantics of each Focal construct, namely «Species», «Collection», «FocalType», «Method», «In», «Is», «ParameterizedInheritance», «Inheritance» and «Implements». To validate our transformation, we also encode the semantics relative to the template binding construct via the introduction of intermediate stereotypes declared as *required* (i.e. mandatory when the corresponding profile is applied). Here, we base ourselves on the OCL formalization realized by Caron et al in [2], which we extend to handle nested *bound* classes and inherited members. To formally represent the application of our profile to a UML model, the abstract syntax given in Figure 1 is slightly extended. In fact, each keyword (e.g. **class inherits**, etc) representing a given UML construct is replaced by a non-terminal node to reflect the stereotypes that can be applied to the corresponding construct. For example, keyword **class** is replaced by the non-terminal node *class-head*, which is defined as follows:

class-head ::= **class** | **focalType** | **species** | **collection**

The syntax is also extended to consider the attributes characterizing each stereotype (e.g., see attributes *substitutes* and *bound* of stereotype «ParameterizedInheritance» in rule $[\![\Gamma_h]\!]^{\mathrm{DE}}_{\Gamma,\mathcal{P},rep,s}$ of Figure 3).

4.2 Transformation Rules

Despite their similarities, Focal species and UML classes are based on two different concepts. In Focal, the functions defined in a species are intended to manipulate entities of a given representation, which are static items having a unique value. Hence, we model a species as an abstract factory class (stereotyped with «Species»), which defines an interface for manipulating immutable value objects of a given type. Let S denote a species: $S = \textbf{species}\ s\ (\mathcal{P})\ \textbf{inherits}\ \Gamma_h = rep;\ \mathcal{M};\ \mathcal{R}\ \textbf{end}$, where s is the name of the species, $\mathcal{P}$ a list of parameters, Γ_h a list of species from which we inherit, rep the representation declaration, $\mathcal{M}$ the declared/defined functions, and $\mathcal{R}$ the properties/theorems defined in S. Given the context Γ, in which S is well typed, the corresponding UML model is obtained by applying the transformation rule denoted by $[\![S]\!]_\Gamma$ in Figures 2 and 3. Our transformation captures every aspect of the Focal specification language. Due to space limitations, we here only focus on the representation, parameter declarations and inheritance. In Figures 2 and 3, $\perp$ is used to denote undefinedness and "$\cdot$" the concatenation operator on identifiers. We also write $c :: \mathit{Self}$ to designate the inner class *Self* defined within c.

The representation of a given species (rule $[\![rep]\!]^{\mathrm{PA}}_{\Gamma,s}$), is characterized by two type parameters T and TSelf (stereotyped with «FocalType»), where T represents the type of the entities and TSelf the class in which T is encapsulated. The latter is used to represent the type of the immutable value objects. Parameter T is generated only if the representation is abstract. The correlation between T and TSelf is specified by the factory methods makeSelf and getRep, which are introduced only if the given species is a root node (rule $[\![rep]\!]^{\mathrm{OP}}_{\Gamma,\mathcal{P},\Gamma_h,\alpha}$).

Inheritance between species is modeled as a dependency relation stereotyped with «ParameterizedInheritance» (rule $[\![\Gamma_h]\!]^{\mathrm{DE}}_{\Gamma,\mathcal{P},rep,s}$), which specifies an intermediate bound class that instantiates the formal parameters of the target factory class. The specializing class inherits from this bound class via a generalization relation stereotyped with «Inheritance» (rule $[\![\Gamma_h]\!]^{\mathrm{GE}}_{\Gamma,s}$).

Function declarations are translated into class operations stereotyped with «Method», which are defined as function object types (using the parameterized class Fun). As for property/theorem declarations, they are represented by UML constraints specified as invariants.

Collections are modeled as concrete singleton factory classes stereotyped accordingly. This allows us to ensure that no method invocation is possible on species, as is the case in Focal. The abstraction of the concrete representation is achieved through the declaration of an inner class Self. This class is declared with a private constructor and a private read-only attribute to obtain the desired encapsulation. The type of the immutable value objects is fixed definitely through the use of the «Implements» stereotype. In essence, the type parameters T and TSelf are instantiated such that T is substituted for a concrete type and TSelf is substituted for the inner class Self created on purpose.

```
        Um  ::=  decl*
      decl  ::=  class | constraint | opaque | dep
     class  ::=  option class ident [ (cl-param {, cl-param}*) ]
                    [ binds bind {, bind}*] [ inherits ident {, ident}* ] =
                     constraint* attr* opr* class* end
    option  ::=  [ visibility ] [final | abstract]
visibility  ::=  public | private | protected
  cl-param  ::=  ident : class [> class-type] | ident : opaqueExpr [> type]
class-type  ::=  ident | bind
      type  ::=  class-type | Integer | Boolean | UnlimitedNatural | String
      bind  ::=  ident<subs [, subs*]>
      subs  ::=  ident → ident
       opr  ::=  option [static] operation ident ([ op-param {, op-param}* ])
                          [ redefines ident {, ident}*]
  op-param  ::=  dir ident [ : type ]
       dir  ::=  in | inout | out | return
      attr  ::=  at-option property ident [ : type ] [ redefines ident {, ident}*]
 at-option  ::=  [ visibility ] [static] [final] [readOnly]
    opaque  ::=  [ visibility ] opaqueExpr ident =
                          [ body ] [ : type ] [ in lang ] end
constraint  ::=  [ visibility ] constraint ident
                        [ restricts ident {, ident}*] =  opaque end
       dep  ::=  [ visibility ] dependency ident
                          (ident {, ident}* --→ ident {, ident}*)
```

Fig. 1 Syntax for UML Static Constructs

4.3 Implementation

Our implementation consists of two parts. In the first part, we define a UML profile for the Focal specification language through the use of the UML2 Eclipse plug-in. This plug-in provides an implementation of the UML 2.1 metamodel and its integrated OCL checker allows us to validate the constraints defined in our profile. The ability to specify statically defined profiles also facilitates the definition of the operations and derived attributes characterizing each stereotype constituting our profile. This step is essential as it provides the necessary tool to validate the UML models to which our profile is applied. In fact, each OCL constraint specified in our profile is parsed and evaluated at runtime. This mechanism offers a convenient way to validate the soundness of our transformation. The second part concerns the development of an XSLT stylesheet that specifies the rules to transform a Focal specification generated in FocDoc format [13] (an XML schema used by the compiler for documentation) into a UML model expressed in the XMI interchange format.

5 Soundness

In this section, we present how the soundness of our transformation can be established. Here, by soundness, we mean that the transformation of a well-typed Focal specification results in a well-formed UML model. We write Δ_p to denote the UML profile established for the Focal specification language. To simplify, we consider Δ_p to be a list of UML constraints $\Phi_1, \ldots, \Phi_m$. Symbol $\mathcal{U}$ is used to denote a UML model, which represents a list of construct declarations $\mathcal{D}_1, \ldots, \mathcal{D}_n$ as described by the abstract syntax shown in Figure 1. We write $\Delta_p(\mathcal{D}_i)$ to denote the list of constraints that relates to the current declaration $\mathcal{D}_i$ when profile Δ_p is applied. Finally, we write Δ_m to denote the UML metamodel, which is considered to be a list of UML constraints $\Omega_1, \ldots, \Omega_q$. Similarly, $\Delta_m(\mathcal{D}_i)$ denotes the list of constraints relative to a given construct declaration $\mathcal{D}_i$. The soundness theorem is the following (due to space restrictions, we omit the corresponding proof):

Theorem 1 (Soundness) *Let $\mathcal{F}$ a well-typed Focal specification within the context Γ s.t. $\mathcal{F} = \mathcal{E}_1, \ldots, \mathcal{E}_n$ and where each $\mathcal{E}_i$ is either a species S or a collection C. Let $\mathcal{U}$ be the UML model obtained when applying the transformation rule $[\![\mathcal{F}]\!]_\Gamma$. Our transformation is sound if the following conditions hold:*

1. $\Delta_p, \Delta_m \nvdash \bot$;
2. *For each $\mathcal{D}_i \in \mathcal{U}$,*
   - $\forall \Omega_j \in \Delta_m(\mathcal{D}_i),\ \Gamma \vdash \Omega_j$;
   - $\forall \Phi_k \in \Delta_p(\mathcal{D}_i),\ \Gamma \vdash \Phi_k$.

The first condition specifies that the constraints within profile Δ_p must not introduce any inconsistency w.r.t. the well-formedness rules of the UML metamodel Δ_m. The second condition states that $\mathcal{U}$ must satisfy both the well-formedness rules of the UML metamodel and the constraints within profile Δ_p.

The previous theorem essentially states that typing is preserved from Focal to UML (even if the well-

Focal Species:

$$[\![S]\!]_\Gamma = \begin{cases} \textbf{public abstract species } s\ [\![\mathcal{P}]\!]^{\text{RE}}_{\Gamma,rep,s}\ [\![\Gamma_h]\!]^{\text{GE}}_{\Gamma,s} = [\![\mathcal{R}]\!]_{\Gamma,s}\ [\![\mathcal{P}]\!]^{\text{AT}}_{\Gamma,s}\ [\![rep]\!]^{\text{OP}}_{\Gamma,\mathcal{P},\Gamma_h,\perp}\ [\![\mathcal{M}]\!]_{\Gamma,\mathcal{P},\Gamma_h,s,\perp}\ \textbf{end} \\ [\![\Gamma_h]\!]^{\text{DE}}_{\Gamma,\mathcal{P},rep,s} \end{cases}$$

Representation and Parameter Declarations:

Given $\mathcal{P} = p_1 \odot I_1, \ldots, p_n \odot I_n$, with $\odot \in \{\textbf{is}, \textbf{in}\}$: $[\![\mathcal{P}]\!]^{\text{RE}}_{\Gamma,rep,s} = ([\![p_1 \odot I_1]\!]_{\Gamma,P_1,s}\ ,\ \ldots,\ [\![p_n \odot I_n]\!]_{\Gamma,P_n,s}\ ,\ [\![rep]\!]^{\text{PA}}_{\Gamma,s})$
with $P_1 = \emptyset$ and $P_n = p_1 \odot I_1, \ldots, p_{n-1} \odot I_{n-1}$

$$[\![p_i \odot I_i]\!]_{\Gamma,P_i,s} = [\![e_i \textbf{ in } \tau]\!]_{\Gamma,P_i,s} \mid [\![c_i \textbf{ is } S_i]\!]_{\Gamma,P_i,s}$$

$$[\![e_i \textbf{ in } \tau]\!]_{\Gamma,P_i,s} = e_i : \textbf{opaqueExpr in } [\![\tau]\!]^{\text{type}}_{\Gamma,P_i,\perp}$$

$$[\![c_i \textbf{ is } S_i]\!]_{\Gamma,P_i,s} = \begin{cases} \left.\begin{array}{l} c_i \cdot T : \textbf{focalType}, \\ \textsf{selfType}_{\Gamma,P_i,c_i \textbf{ is } S_i} \end{array}\right\} \text{if } \Gamma(S_i).rep = \perp \\ \textsf{selfType}_{\Gamma,P_i,c_i \textbf{ is } S_i} \quad \text{otherwise} \end{cases}$$

$$[\![rep]\!]^{\text{PA}}_{\Gamma,s} = \begin{cases} T : \textbf{focalType},\ TSelf : \textbf{focalType} & \text{if } rep = \perp \\ TSelf : \textbf{focalType} & \text{otherwise} \end{cases}$$

$$\textsf{selfType}_{\Gamma,P_i,c_i \textbf{ is } S_i} = \begin{cases} c_i \cdot Self : \textbf{focalType}, \\ c_i : \textbf{collection is } [\![S_i]\!]^{\text{const}}_{\Gamma,P_i,c_i} \end{cases}$$

$$[\![S_i]\!]^{\text{const}}_{\Gamma,P_i,c_i} = [\![s_{q_i}]\!]^{\text{const}}_{\Gamma,P_i,c_i} \mid [\![s_{q_i}(a_1,\ldots,a_f)]\!]^{\text{const}}_{\Gamma,P_i,c_i}$$

$$[\![s_{q_i}]\!]^{\text{const}}_{\Gamma,P_i,c_i} = \begin{cases} s_{q_i}{<}T \rightarrow c_i \cdot T,\ TSelf \rightarrow c_i \cdot Self{>}_{\mathbf{s}} & \text{if } \Gamma(s_{q_i}).rep = \perp \\ s_{q_i}{<}TSelf \rightarrow c_i \cdot Self{>}_{\mathbf{s}} & \text{otherwise} \end{cases}$$

$$[\![s_{q_i}(a_1,\ldots,a_f)]\!]^{\text{const}}_{\Gamma,P_i,c_i} = \begin{cases} \left.\begin{array}{l} s_{q_i}\ [\![a_1]\!]_{\Gamma,P_i,\mathcal{S}_1,p_{q_1}\odot I_{q_1}}, \ldots, [\![a_f]\!]_{\Gamma,P_i,\mathcal{S}_f,p_{q_f}\odot I_{q_f}}, \\ \quad {<}T \rightarrow c_i \cdot T,\ TSelf \rightarrow c_i \cdot Self{>}_{\mathbf{s}} \end{array}\right\} \text{if } \Gamma(s_{q_i}).rep = \perp \\ \left.\begin{array}{l} s_{q_i}\ [\![a_1]\!]_{\Gamma,P_i,\mathcal{S}_1,p_{q_1}\odot I_{q_1}}, \ldots, [\![a_f]\!]_{\Gamma,P_i,\mathcal{S}_f,p_{q_f}\odot I_{q_f}}, \\ \quad {<}TSelf \rightarrow c_i \cdot Self{>}_{\mathbf{s}} \end{array}\right\} \text{otherwise} \end{cases}$$

with $p_{q_1} \odot I_{q_1}, \ldots, p_{q_f} \odot I_{q_f} = \Gamma(s_{q_i}).\mathcal{P}$, $\mathcal{S}_1 = \emptyset$ and $\mathcal{S}_f = \{(a_1, p_{q_1}), \ldots, (a_{f-1}, p_{q_{f-1}})\}$

$$[\![a_k]\!]_{\Gamma,P_i,\mathcal{S}_k,p_{q_k}\odot I_{q_k}} = \begin{cases} [\![e_k]\!]_{\Gamma,P_i,\mathcal{S}_k,e_{q_k} \textbf{ in } \tau_{q_k}} & \text{if } a_k = e_k \ \wedge\ p_{q_k} \odot I_{q_k} = e_{q_k} \textbf{ in } \tau_{q_k} \\ [\![c_k]\!]_{\Gamma,P_i,c_{q_k} \textbf{ is } S_{q_k}} & \text{if } a_k = c_k \ \wedge\ p_{q_k} \odot I_{q_k} = c_{q_k} \textbf{ is } S_{q_k} \end{cases}$$

$$[\![e_k]\!]_{\Gamma,P_i,\mathcal{S}_k,e_{q_k} \textbf{ in } \tau_{q_k}} = \begin{cases} {<}e_{q_k} \rightarrow e_j{>} & \text{if } \exists\ e_j \textbf{ in } \tau_j \in P_i \ s.t.\ e_j = e_k \\ {<}e_{q_k} \rightarrow id_k{>} & \text{otherwise} \end{cases}$$

where id_k is a new identifier s.t.,
$\textbf{opaqueExpr } id_k = e_k : [\![\tau_{q_k}]\!]^{\text{subs}}_{\Gamma,P_i,\mathcal{S}_k}$ **in** "Focal" **end**

$$[\![c_k]\!]_{\Gamma,P_i,c_{q_k} \textbf{ is } S_{q_k}} = \begin{cases} {<}c_{q_k} \cdot T \rightarrow \textsf{repBind}_{\Gamma,P_i,\beta,c_{q_k},c_k}{>}\ ,\ \textsf{selfBind}_{\beta,c_{q_k},c_k} & \text{if } \Gamma(S_{q_k}).rep = \perp \\ \textsf{selfBind}_{\beta,c_{q_k},c_k} & \text{otherwise} \end{cases}$$

$$\text{with } \beta = \begin{cases} c_j \textbf{ is } S_j & \text{if } \exists\ c_j \textbf{ is } S_j \in P_i \ s.t.\ c_j = c_k \\ \perp & \text{otherwise} \end{cases}$$

$$\textsf{selfBind}_{\beta,c_{q_k},c_k} = \begin{cases} {<}c_{q_k} \cdot Self \rightarrow c_k :: Self,\ c_{q_k} \rightarrow c_k{>} & \text{if } \beta = \perp \\ {<}c_{q_k} \cdot Self \rightarrow c_j \cdot Self,\ c_{q_k} \rightarrow c_j{>} & \text{if } \beta = c_j \textbf{ is } S_j \end{cases}$$

$$\textsf{repBind}_{\Gamma,\beta,P_i,c_{q_k},c_k} = \begin{cases} c_j \cdot T & \text{if } \beta = c_j \textbf{ is } S_j \ \wedge\ \Gamma(S_j).rep = \perp \\ [\![\tau_j]\!]^{\text{type}}_{\Gamma,P_i,\perp} & \text{if } \beta = c_j \textbf{ is } S_j \ \wedge\ \Gamma(S_j).rep = \tau_j \\ [\![\Gamma(c_k).rep]\!]^{\text{type}}_{\Gamma,\emptyset,c_k} & \text{otherwise} \end{cases}$$

Factory Methods for the Representation:

$$[\![rep]\!]^{\text{OP}}_{\Gamma,\mathcal{P},\Gamma_h,\alpha} = \begin{cases} \textbf{protected abstract method } makeSelf\ (\textbf{in } x : \textsf{repType}_{\Gamma,\mathcal{P},rep}\ ,\ \textbf{return } y : \textsf{selfBind}_\alpha) \\ \textbf{protected abstract method } getRep\ (\textbf{in } x : \textsf{selfBind}_\alpha\ ,\ \textbf{return } y : \textsf{repType}_{\Gamma,\mathcal{P},rep}) \end{cases}$$

when $\Gamma_h = \emptyset$

$$\textsf{repType}_{\Gamma,\mathcal{P},rep} = \begin{cases} T & \text{if } rep = \perp \\ [\![\tau]\!]^{\text{type}}_{\Gamma,\mathcal{P},\perp} & \text{if } rep = \tau \end{cases} \qquad \textsf{selfBind}_\alpha = \begin{cases} TSelf & \text{if } \alpha = \perp \\ c :: Self & \text{if } \alpha = c \end{cases}$$

Types:

Given a list of parameter declarations P and α either referencing a collection or set to $\perp$:

$$[\![\tau]\!]^{\text{type}}_{\Gamma,P,\alpha} = [\![c]\!]^{\text{type}}_{\Gamma,P,\alpha} \mid [\![t]\!]^{\text{type}}_{\Gamma,P,\alpha} \mid [\![\tau_1 \rightarrow \tau_2]\!]^{\text{type}}_{\Gamma,P,\alpha} \mid [\![\tau_1 * \tau_2]\!]^{\text{type}}_{\Gamma,P,\alpha} \mid [\![\textbf{self}]\!]^{\text{type}}_{\Gamma,P,\alpha}$$

$$[\![c]\!]^{\text{type}}_{\Gamma,P,\alpha} = \begin{cases} c_j \cdot Self & \text{if } \exists\ c_j \textbf{ is } S_j \in P \ s.t.\ c_j = c \\ c :: Self & \text{otherwise} \end{cases} \qquad [\![t]\!]^{\text{type}}_{\Gamma,P,\alpha} = t \qquad [\![\textbf{self}]\!]^{\text{type}}_{\Gamma,P,\alpha} = \begin{cases} TSelf & \text{if } \alpha = \perp \\ c :: Self & \text{if } \alpha = c \end{cases}$$

$$[\![\tau_1 \rightarrow \tau_2]\!]^{\text{type}}_{\Gamma,P,\alpha} = Fun{<}A \rightarrow [\![\tau_1]\!]^{\text{type}}_{\Gamma,P,\alpha},\ B \rightarrow [\![\tau_2]\!]^{\text{type}}_{\Gamma,P,\alpha}{>}_{\mathbf{f}} \qquad [\![\tau_1 * \tau_2]\!]^{\text{type}}_{\Gamma,P,\alpha} = Pair{<}A \rightarrow [\![\tau_1]\!]^{\text{type}}_{\Gamma,P,\alpha},\ B \rightarrow [\![\tau_2]\!]^{\text{type}}_{\Gamma,P,\alpha}{>}_{\mathbf{f}}$$

Fig. 2 Transformation Rules: Focal to UML (1)

Inheritance:

Given $\Gamma_h = S_{h_1}, \ldots, S_{h_m}$: $[\![\Gamma_h]\!]^{\mathrm{DE}}_{\Gamma,\mathcal{P},rep,s} = [\![S_{h_1}]\!]^{\mathrm{DE}}_{\Gamma,\mathcal{P},rep,s} \;\cdots\; [\![S_{h_m}]\!]^{\mathrm{DE}}_{\Gamma,\mathcal{P},rep,s}$

$$[\![S_{h_i}]\!]^{\mathrm{DE}}_{\Gamma,\mathcal{P},rep,s} = [\![s_{q_i}]\!]^{\mathrm{DE}}_{\Gamma,\mathcal{P},rep,s} = [\![s_{q_i}(a_1,\ldots,a_g)]\!]^{\mathrm{DE}}_{\Gamma,\mathcal{P},rep,s} = \begin{cases} \textbf{public paramInheritance } s \cdot s_{q_i} \cdot de \;\mathbf{(}s \dashrightarrow s_{q_i}\mathbf{)} \;= \\ \quad \textbf{substitutes(}[\![S_{h_i}]\!]^{\mathrm{prm}}_{\Gamma,\mathcal{P},rep}\textbf{)} \;\; \textbf{bound } s \cdot s_{q_i} \cdot bound \\ \textbf{end} \end{cases}$$

with $s \cdot s_{q_i} \cdot bound$ referencing the bound class,

species $s \cdot s_{q_i} \cdot bound$ **instantiates** s_{q_i} $[\![S_{h_i}]\!]^{\mathrm{prm}}_{\Gamma,\mathcal{P},rep} = [\![S_{h_i}]\!]^{\mathrm{AT}}_{\Gamma,\emptyset,\bot} \; [\![rep]\!]^{\mathrm{OP}}_{\Gamma,\mathcal{P},\emptyset,\bot} \; [\![\mathsf{allMethods}(\Gamma, S_{h_i})]\!]_{\Gamma,\mathcal{P},\emptyset,s,\bot}$ **end**

$$[\![S_{h_i}]\!]^{\mathrm{prm}}_{\Gamma,\mathcal{P},rep} = [\![s_{q_i}]\!]^{\mathrm{prm}}_{\Gamma,\mathcal{P},rep} \mid [\![s_{q_i}(a_1,\ldots,a_g)]\!]^{\mathrm{prm}}_{\Gamma,\mathcal{P},rep}$$

$$[\![s_{q_i}]\!]^{\mathrm{prm}}_{\Gamma,\mathcal{P},rep} = \begin{cases} <T \rightarrow T,\; TSelf \rightarrow TSelf> & \text{if } \Gamma(s_{q_i}).rep = \bot \;\wedge\; rep = \bot \\ <T \rightarrow [\![\tau]\!]^{\mathrm{type}}_{\Gamma,\mathcal{P},\bot},\; TSelf \rightarrow TSelf> & \text{if } \Gamma(s_{q_i}).rep = \bot \;\wedge\; rep = \tau \\ <TSelf \rightarrow TSelf> & \text{otherwise} \end{cases}$$

$$[\![s_{q_i}(a_1,\ldots,a_g)]\!]^{\mathrm{prm}}_{\Gamma,\mathcal{P},rep} = [\![a_1]\!]_{\Gamma,\mathcal{P},\mathcal{S}_1,p_{q_1}\odot I_{q_1}}, \ldots, [\![a_g]\!]_{\Gamma,\mathcal{P},\mathcal{S}_g,p_{q_g}\odot I_{q_g}}, [\![s_{q_i}]\!]^{\mathrm{prm}}_{\Gamma,\mathcal{P},rep}$$

with $p_{q_1} \odot I_{q_1}, \ldots, p_{q_g} \odot I_{q_g} = \Gamma(s_{q_i}).\mathcal{P}$, $\mathcal{S}_1 = \emptyset$ and $\mathcal{S}_g = (a_1, p_{q_1}), \ldots, (a_{g-1}, p_{q_{g-1}})$

$[\![\Gamma_h]\!]^{\mathrm{GE}}_{\Gamma,s} =$ **inheritance** $[\![S_{h_1}]\!]^{\mathrm{GE}}_{\Gamma,s}, \ldots, [\![S_{h_m}]\!]^{\mathrm{GE}}_{\Gamma,s}$ $\qquad [\![S_{h_i}]\!]^{\mathrm{GE}}_{\Gamma,s} = [\![s_{q_i}]\!]^{\mathrm{GE}}_{\Gamma,s} = [\![s_{q_i}(a_1,\ldots,a_g)]\!]^{\mathrm{GE}}_{\Gamma,s} = s \cdot s_{q_i} \cdot bound$

Fig. 3 Transformation Rules: Focal to UML (2)

formedness rules are said to characterize the semantics of UML). Another form of soundness, not considered in this paper, would be to establish that the semantics of Focal is also preserved by the transformation, which is equivalent to show that there exists a model of the UML metamodel (together with the profile), for which the well-formedness rules are correct and which is compatible with a model of Focal.

6 An Application Example

To illustrate our transformation process, we consider a relatively concise example extracted from the formalization realized within the EDEMOI project. This concerns the specification established for cabin persons. The corresponding Focal species is defined as follows:

```
species cabinPerson (cb is cabinBaggage) =
  rep;
  sig equal in self → self → bool;
  sig identityVerified in self → bool;
  sig cabinBaggage in self → cb;
  property equal_reflexive:
    all x in self, !equal (x, x); ...
end
```

It can be observed that cabinPerson is a parameterized species and its representation is left undefined. We also assume that the representation of species cabinBaggage is still abstract. To give an example of inheritance and show how the abstraction of a concrete representation is handled during the transformation process, we also introduce collection cabinPerson_col, which provides an implementation for species cabinPerson:

```
collection cabinPerson_col
           implements cabinPerson (bag) =

  rep = string * bag * bool;
  let name (s in self) in string = #first (s);
  let cabinBaggage (s in self) in bag =
    #first (#scnd (s));
  let identityVerified (s in self) in bool =
    #scnd (#scnd (s)); ...
end
```

In this collection, the representation is specified as a triple, with the functions name, cabinBaggage and identityVerified defined accordingly. In the "implements" clause, species cabinPerson is instantiated with bag, which is a collection derived from cabinBaggage.

Now, by applying the transformation rules described in Section 4, the UML classes shown in Figure 4 (using the corresponding graphical visualization) are obtained, where we write TSelf → Bool for the bound class Fun<TSelf, Bool>.

7 Conclusion

In this paper, we present a formal and sound framework for the transformation of Focal specifications into UML models, with the objective to provide a graphical documentation for developers. The transformation rules proposed attempt to provide an appropriate design pattern for the representation of algebraic structures and algorithms within an object-oriented paradigm. Hence, from the UML models produced, it may be possible to map a Focal specification to any appropriate object-oriented programming language, e.g. Java or C#.

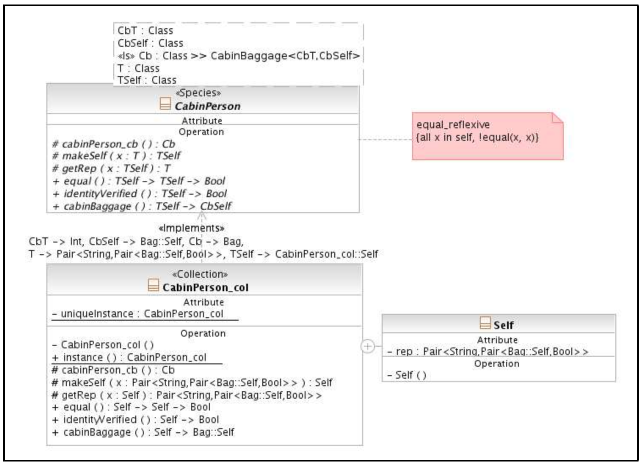

Fig. 4 CabinPerson Classes

Regarding future work, we expect to use the present transformation rules as a basis to generate higher-level views that would be more pertinent for certification authorities or end-users (not only for developers). Another perspective is to apply our transformation process to more concrete specifications (the models realized for the EDEMOI project are quite abstract), such as the standard library of Focal, which consists of a large formalization of Computer Algebra. In this way, it would be possible to see whether the generated UML models are fairly comprehensible and can be used for managing libraries. Finally, we aim to generate more dynamic views of the formal models (sequence and state-transition diagrams) through static analysis performed on Focal specifications.

References

1. R. Bonichon, D. Delahaye, and D. Doligez. Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 4790 of *LNCS/LNAI*, pages 151–165. Springer, Oct. 2007.
2. O. Caron, B. Carré, A. Muller, and G. Vanwormhoudt. An OCL Formulation of UML2 Template Binding. In *UML Modeling Languages and Applications (UML)*, volume 3273 of *LNCS*, pages 27–40. Springer, Oct. 2004.
3. D. Delahaye, J.-F. Étienne, and V. Viguié Donzeau-Gouge. Certifying Airport Security Regulations using the Focal Environment. In *Formal Methods (FM)*, volume 4085 of *LNCS*, pages 48–63. Springer, Aug. 2006.
4. D. Delahaye, J.-F. Étienne, and V. Viguié Donzeau-Gouge. Reasoning about Airport Security Regulations using the Focal Environment. In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 45–52. IEEE CS Press, Nov. 2006.
5. D. Delahaye, J.-F. Étienne, and V. Viguié Donzeau-Gouge. Producing UML Models from Focal Specifications: An Application to Airport Security Regulations. In *Theoretical Aspects of Software Engineering (TASE)*. IEEE CS Press, June 2008.
6. C. Dubois, T. Hardin, and V. Viguié Donzeau-Gouge. Building Certified Components within Focal. In *Trends in Functional Programming (TFP)*, volume 5, pages 33–48. Intellect, Nov. 2004.
7. S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. An Overview of RoZ: A Tool for Integrating UML and Z specifications. In *Conference on Advanced Information Systems Engineering (CAiSE)*, volume 1789 of *LNCS*, pages 417–430. Springer, June 2000.
8. T. Goldschmidt, S. Becker, and A. Uhl. Classification of Concrete Textual Syntax Mapping Approaches. In *European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, volume 5095 of *LNCS*, pages 169–184. Springer, June 2008.
9. A. Idani and Y. Ledru. Dynamic Graphical UML Views from Formal B Specifications. *International Journal of Information and Software Technology*, 48(3):154–169, Mar. 2006.
10. F. Jouault and I. Kurtev. On the Interoperability of Model-to-Model Transformation Languages. *Science of Computer Programming*, 68(3):114–137, Oct. 2007.
11. S.-K. Kim and D. A. Carrington. A Formal Mapping between UML Models and Object-Z Specifications. In *International Z and B Conference (ZB)*, volume 1878 of *LNCS*, pages 2–21. Springer, Sept. 2000.
12. R. Laleau and F. Polack. Coming and Going from UML to B: A Proposal to Support Traceability in Rigorous IS Development. In *International Z and B Conference (ZB)*, volume 2272 of *LNCS*, pages 517–534. Springer, Jan. 2002.
13. M. Maarek and V. Prevosto. FocDoc: The Documentation System of Foc. In *Calculemus*. LIP6, Sept. 2003.
14. The EDEMOI Project, 2003.
    `http://www-lsr.imag.fr/EDEMOI/`.
15. The Focal Development Team. *Focal, version 0.3.1.* CNAM/INRIA/LIP6, May 2005.
    Available at: `http://focal.inria.fr/`.
16. The Object Management Group. *Unified Modeling Language: Superstructure, version 2.1.1*, Feb. 2007.
    Available at: `http://www.omg.org/`.