Producing UML Models from Focal Specifications
An Application to Airport Security Regulations

David Delahaye
CEDRIC/CNAM, Paris, France
David.Delahaye@cnam.fr

Jean-Frédéric Étienne
CEDRIC/CNAM, Paris, France
etiennje@cnam.fr

Véronique Viguié Donzeau-Gouge
CEDRIC/CNAM, Paris, France
donzeau@cnam.fr

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 regulations in the domain of airport security. 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 provide a concrete example coming from the* EDEMOI *project.*

1. Introduction

These days, formal methods are undoubtedly the most effective approach for the production of safety-critical systems. In essence, the use of formal modeling techniques provides support to build unambiguous specifications, to ensure the consistency of a specification and to guarantee that an implementation satisfies its specification. However, the validation process cannot be mechanized as it still relies on a high degree of interaction between the various stake-holders (developers, end-users, certification authorities, etc) involved in a project. In addition, the use of formal methods requires a certain level of expertise in mathematics, which usually hinders communication. As a result, the validation of requirements is quite difficult to achieve. To remedy these problems, the solution that is widely adopted is the integration of formal and graphical specifications. In fact, the use of graphical notations has been shown to be quite useful when interacting with end-users. For the last few years, UML [4] has emerged as a standard in industry for modeling software systems. It provides a set of graphical notations, 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.

The main motivation for this work lies within the framework of the EDEMOI[1] [5] project, which aims to integrate and apply several requirements engineering and formal methods techniques to analyze regulations in the domain of airport security. 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. This formalization is described in [2] and the certification part is presented in [3]. In this paper, we describe a schema for the translation of Focal specifications into UML diagrams, with the objective of providing a graphical documentation of our formal models for developers. In the long term, the idea is to provide higher-level views that would be pertinent to certification authorities. Our major concern is to make our transformation automatic and as formal as possible.

The paper is organized as follows: first, we give a brief presentation of the Focal language; next, we propose a formal description for a subset of the UML 2.1 static structure constructs; we then describe our transformation from Focal to UML; finally, we introduce our implementation and provide a concrete example extracted from our formalization of regulations in the context of the EDEMOI project.

2. The Focal environment

Focal [6], initiated by T. Hardin and R. Rioboo, is a language in which it is possible to build certified applica-

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

tions step by step, going from abstract specifications, called *species*, to concrete implementations, called *collections*. In this language, the first major notion is the structure of species, which corresponds to the highest level of abstraction in a specification and which has the following syntax:

```
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, <body> a function body, <prop> a (first-order) proposition and <proof> a proof.

Species can be combined using (multiple) inheritance (which works as expected) and can be parameterized either by other species or by entities from species. These two features complete the previous syntax definition as follows:

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

where <pars> is a list of <name> and denotes the names which are used as parameters. When the parameter is a species, the keyword is **is**, when it is an entity of a species, the keyword is **in**.

The other main notion of the Focal language is the structure of collection, which corresponds to the implementation of a specification (every attribute must be concrete). The syntax of a collection is the following:

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

For further information regarding Focal and, in particular, for examples of specifications, the reader can refer to [2, 6], as well as to the Focal Web site[2], which contains the Focal distribution (the Focal compiler, the Zenon automated theorem prover [1], and some other tools) and some documentation.

3. UML Syntax

In this section, we give a formal syntax for a subset of the UML 2.1 static structure constructs [4], used as a means to provide a graphical documentation for Focal specifications. Due to space restrictions, we here only consider the constructs necessary to represent the notion of species and collection in UML. Normally, the syntax and semantics of each UML construct are described in the form of a metamodel. The syntax is specified using class diagrams, while the semantics are well-formedness rules expressed in a combination of OCL and English. In order to provide a formal framework for the transformation of Focal specifications into UML diagrams, we choose to represent the syntax of each UML construct considered in a BNF-like format. The syntax proposed is derived from the UML 2.1/XMI schema, generally used for the exchange of UML models in text format. It is shown in Figure 1, where the non-terminal symbols are in italic, $\overline{X}$ is a list of X, and $\overline{X}^1$ is a list of X with at least one element. We also introduce the set $Name$ for names, and appropriate sets for identifiers (e.g. $ClassId$ for class identifiers).

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

4. From Focal to UML

4.1. Extending the UML Metamodel

In order to properly specify, visualize and document Focal models using UML notations, there is a need to extend the UML metamodel to cater for the semantic specificities of the Focal specification language (see Section 2). The necessary extensions are realized via the creation of a profile, where appropriate stereotypes are defined to reflect the semantics of each Focal construct. These stereotypes also have a set of OCL constraints, which are used to specify the well-formedness rules associated to each construct. Due to space restrictions, we do not detail the stereotypes constituting our profile, which are namely: «Parameterized-Inheritance», «Species», «Collection», «Inheritance», «Implements», «Is», «In» and «Method».

4.2. Transformation Rules

In spite of their similarities, Focal species and UML classes are based on two different concepts. In Focal, the functions defined within 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 a singleton factory class (stereotyped with «Species»), which defines an interface for manipulating immutable value objects of a given type. Let S denote a species of the form: S = **species** s $(\mathcal{P})$ **inherits** Γ_h = rep; $\mathcal{M}$; $\mathcal{R}$ **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 U_m is obtained by applying the transformation rule denoted by $[\![S]\!]_\Gamma$ in Figure 2 (due to space restrictions, we only focus on the translation of the representation). It should be noted that our transformation captures every aspect of the Focal language.

$$
\begin{array}{lll}
Um & ::= & \overline{CD} \\
CD & ::= & C \mid CO \mid OE \mid DE \\
C & ::= & \textbf{Class}(c, [name,]\ [visibility,]\ isLeaf, isAbstract, \\
 & & \quad \overline{TB}, [RS,]\ \overline{GE}, \overline{CO}, \overline{AT}, \overline{C}, \overline{OP}) \\
GE & ::= & \textbf{Generalization}([isSubstitutable,]\ \textbf{general} = c) \\
OP & ::= & \textbf{Operation}(op, [name,]\ [visibility,]\ isLeaf, isStatic \\
 & & \quad isAbstract, \overline{PA}\ [, TB]\ [, TS]\ [, \textbf{redefinedOpr} = \overline{op_e}^1]\ ) \\
PA & ::= & \textbf{Parameter}(pa, [name,]\ direction\ [, \textbf{type} = typ]\ ) \\
AT & ::= & \textbf{Property}(at, [name,]\ [visibility,]\ isLeaf, isStatic, \\
 & & \quad isReadOnly\ [, \textbf{type} = typ]\ [, \textbf{redefinedProp} = \overline{at_e}^1]\ ) \\
typ & ::= & c \mid \textbf{Integer} \mid \textbf{Boolean} \mid \textbf{UnlimitedNautral} \mid \textbf{String} \\
RS & ::= & \textbf{RedTmpSig}(rs, isLeaf, \textbf{redefContext} = c, [\textbf{extendedSig} = \overline{rs_e}^1,] \\
 & & \quad \textbf{param} = \overline{fp}^1\ [, \textbf{ownedParam}(\overline{FP}^1)]\ )
\end{array}
$$

$$
\begin{array}{lll}
TS & ::= & \textbf{TmpSig}(ts, \textbf{param} = \overline{fp}^1\ [, \textbf{ownedParam}(\overline{FP}^1)]\ ) \\
FP & ::= & TP \mid CTP \\
TP & ::= & \textbf{TmpParam}(tp, \textbf{paramElem} = e\ [, \textbf{ownedParamElem}(PE)]\ ) \\
CTP & ::= & \textbf{ClsfrTmpParam}(cp, allowSubstitutable, \textbf{paramElem} = c_1 \\
 & & \quad [, \textbf{ownedParamElem}(C)]\ [, \textbf{constClsfr} = c_2]\ ) \\
PE & ::= & CO \mid OE \\
TB & ::= & \textbf{TmpBinding}(\textbf{signature} = s, \overline{PS}) \\
PS & ::= & \textbf{ParamSub}(\textbf{formal} = fp, \textbf{actual} = ap\ [, \textbf{ownedActual}(AP)]\ ) \\
AP & ::= & PE \mid C \\
CO & ::= & \textbf{Constraint}(co\ [name,]\ [visibility,]\ [\textbf{constElem} = \overline{n}^1,]\ \textbf{spec}(OE)\ ) \\
OE & ::= & \textbf{OpaqueExp}(oe\ [, name]\ [, visibility]\ [, \textbf{body}(body)] \\
 & & \quad [, \textbf{language} = lang]\ [, \textbf{type} = typ]\ ) \\
DE & ::= & \textbf{Dependency}(de, [name,]\ [visibility,]\ \textbf{client} = \overline{n_1}^1, \textbf{supplier} = \overline{n_2}^1)
\end{array}
$$

where,

$$
\begin{array}{lll}
fp & \in & TPrmId \cup ClsfrTPrmId \\
s & \in & RedTSigId \cup TSigId \\
n & \in & ConstId \cup OExpId \cup ClassId \cup AttrId \cup OprId \cup DepId \\
body & \in & String
\end{array}
\qquad
\begin{array}{lll}
e & \in & ConstId \cup OExpId \\
ap & \in & ConstId \cup OExpId \cup ClassId \\
name & \in & Name \\
lang & \in & String
\end{array}
$$

$$
\begin{array}{c}
isLeaf, isAbstract, isSubstitutable, \\
isReadOnly, isStatic, allowSubstitutable
\end{array}
::= \textbf{true} \mid \textbf{false}
\qquad
\begin{array}{lll}
direction & ::= & \textbf{in} \mid \textbf{inout} \mid \textbf{out} \mid \textbf{return} \\
visibility & ::= & \textbf{public} \mid \textbf{private} \mid \textbf{protected}
\end{array}
$$

Figure 1. Syntax for UML Static Constructs

The representation of a given species is handled in two ways (rule $[\![rep]\!]^{\text{PA}}_{\Gamma,s}$), depending on whether it is abstract or concrete. In the first case, two type parameters T and TSelf are defined, 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. 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 $[\![rep]\!]^{\text{OP}}_{\Gamma,\Gamma_h,\mathcal{P},s}$). In the second case, only the TSelf type parameter is generated.

Collections are modeled as concrete singleton factory classes stereotyped accordingly. 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 being substituted for a concrete type and TSelf is being substituted for the inner class Self created on purpose.

4.3. Implementation

Our implementation consists of two parts. First, we defined 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 allowed us to validate the constraints defined in our profile. Next, we developed an XSLT stylesheet that specifies the rules to transform a Focal specification generated (by the compiler) in FocDoc format (XML) into a UML model expressed in the XMI interchange format.

5. An Application Example

To illustrate our transformation process, we consider a relatively concise example extracted from the Focal specification realized within the framework of 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,

$$[\![S]\!]_\Gamma = \begin{cases} \textbf{Class}\,(s, \text{“S”}, \textbf{public}, \textbf{false}, \textbf{true}, [\![\mathcal{P}]\!]^{\text{RE}}_{\Gamma,rep,s}, [\![\Gamma_h]\!]^{\text{GE}}_{\Gamma,s}, [\![\mathcal{R}]\!]_{\Gamma,s}, [\![\mathcal{P}]\!]^{\text{OP}}_{\Gamma,s}\ [\![rep]\!]^{\text{OP}}_{\Gamma,\Gamma_h,s}\ [\![\mathcal{M}]\!]_{\Gamma,\Gamma_h,\mathcal{P},s})\ \textbf{«Species»} \\ [\![\Gamma_h]\!]^{\text{DE}}_{\Gamma,rep,s} \end{cases}$$

$$[\![\mathcal{P}]\!]^{\text{RE}}_{\Gamma,rep,s} = \begin{cases} \textbf{RedTmpSig}\,(s\text{-}rs, \textbf{true}, \textbf{redefContext} = s, \textbf{param} = s\text{-}tp\text{-}c_1 \ldots s\text{-}tp\text{-}c_n\ [\![rep]\!]^{\text{prm}}_s, \\ \qquad \textbf{ownedParam}\,([\![c_1 \odot I_1]\!]_{\Gamma,\mathcal{P},s} \ldots [\![c_n \odot I_n]\!]_{\Gamma,\mathcal{P},s}\ [\![rep]\!]^{\text{PA}}_{\Gamma,s})) \end{cases} \quad \text{where, } \mathcal{P} = c_1 \odot I_1 \ldots c_n \odot I_n \text{ and } \odot \in \{\textbf{is}, \textbf{in}\}$$

$$[\![rep]\!]^{\text{PA}}_{\Gamma,s} = \begin{cases} repPrm_s\ \textbf{ClsfrTmpParam}\,(s\text{-}tp\text{-}self, \textbf{false}, \textbf{paramElem} = s\text{-}pa\text{-}self, \\ \qquad \textbf{ownedParamElem}\,(\textbf{Class}\,(s\text{-}pa\text{-}self, \text{“TSelf”}, \textbf{public}, \textbf{false}, \textbf{false}))) & \text{if } rep = \bot \\ \textbf{ClsfrTmpParam}\,(s\text{-}tp\text{-}self, \textbf{false}, \textbf{paramElem} = s\text{-}pa\text{-}self, \\ \qquad \textbf{ownedParamElem}\,(\textbf{Class}\,(s\text{-}pa\text{-}self, \text{“TSelf”}, \textbf{public}, \textbf{false}, \textbf{false}))) & \text{otherwise} \end{cases}$$

$$repPrm_s = \textbf{ClsfrTmpParam}\,(s\text{-}tp\text{-}rep, \textbf{false}, \textbf{paramElem} = s\text{-}pa\text{-}rep, \textbf{ownedParamElem}\,(\textbf{Class}\,(s\text{-}pa\text{-}rep, \text{“T”}, \textbf{public}, \textbf{false}, \textbf{false})))$$

$$[\![rep]\!]^{\text{OP}}_{\Gamma,\Gamma_h,\mathcal{P},s} = \begin{cases} \textbf{Operation}\,(s\text{-}op\text{-}self, \text{“makeSelf”}, \textbf{protected}, \textbf{false}, \textbf{false}, \textbf{true}, \\ \qquad \textbf{Parameter}\,(s\text{-}op\text{-}self\text{-}in, \text{“x”}, \textbf{in}, \textbf{type} = \mathsf{repType}_{\Gamma,\mathcal{P},rep,s})\ \textbf{Parameter}\,(s\text{-}op\text{-}self\text{-}ret, \text{“ret”}, \textbf{return}, \textbf{type} = s\text{-}pa\text{-}self)) \\ \textbf{Operation}\,(s\text{-}op\text{-}rep, \text{“getRep”}, \textbf{protected}, \textbf{false}, \textbf{false}, \textbf{true}, \\ \qquad \textbf{Parameter}\,(s\text{-}op\text{-}rep\text{-}in, \text{“x”}, \textbf{in}, \textbf{type} = s\text{-}pa\text{-}self)\ \textbf{Parameter}\,(s\text{-}op\text{-}rep\text{-}ret, \text{“ret”}, \textbf{return}, \textbf{type} = \mathsf{repType}_{\Gamma,\mathcal{P},rep,s})) \end{cases} \quad \text{if } \Gamma_h = \emptyset$$

$$\mathsf{repType}_{\Gamma,\mathcal{P},rep,s} = \begin{cases} s\text{-}pa\text{-}rep & \text{if } rep = \bot \\ [\![\tau]\!]^{\text{type}}_{\Gamma,\mathcal{P},\bot} & \text{if } rep = \tau \end{cases} \qquad [\![rep]\!]^{\text{prm}}_s = \begin{cases} s\text{-}tp\text{-}rep\ \ s\text{-}tp\text{-}self & \text{if } rep = \bot \\ s\text{-}tp\text{-}self & \text{otherwise} \end{cases}$$

Figure 2. Transformation Rules: Focal **to** UML

species cabinPerson is instantiated with bag, which is a collection derived from cabinBaggage (to simplify, we here assume that its representation is set to the predefined type int).

Now, by applying the transformation rules described in Section 4, the UML classes shown in Figure 3 (using the corresponding graphical visualization) are obtained.

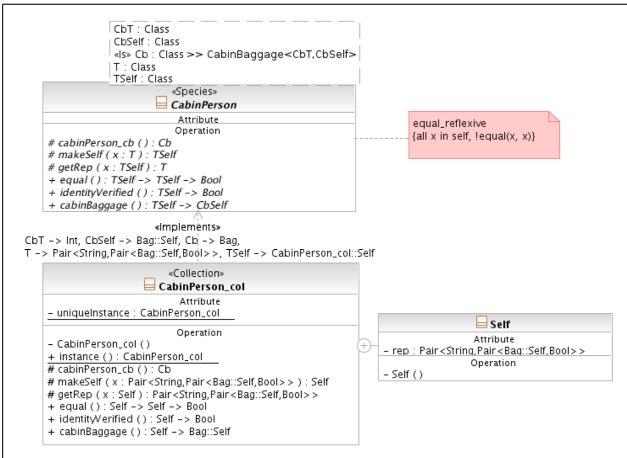

Figure 3. CabinPerson Classes

6. Conclusion

Regarding future work, we expect to use the present transformation rules as a basis to generate higher-level views that would be closer to conceptual models (similar to those produced at the preliminary stage of the EDEMOI project) and hence 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 formal models realized within the framework of 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 the definitions involved in 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] 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.

[3] 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 36–44, Nov. 2006.

[4] O. M. Group. UML*: Superstructure, version 2.1.1*. OMG, Feb. 2007. Available at: `http://www.omg.org/`.

[5] The EDEMOI Project, 2003. `http://www-lsr.imag.fr/EDEMOI/`.

[6] The Focal Development Team. Focal*, version 0.3.1*. CNAM/INRIA/LIP6, May 2005. Available at: `http://focal.inria.fr/`.