Journal of Data Semantics (JoDS) manuscript No.
(will be inserted by the editor)

Integrating a Formal Development for DSLs into Meta-Modeling

Selma Djeddai · Martin Strecker · Mohamed Mezghiche

the date of receipt and acceptance should be inserted later

Abstract Formal methods (such as interactive provers) are increasingly used in software engineering. They offer a formal frame that guarantees the correctness of developments. Nevertheless, they use complex notations that might be difficult to understand for unaccustomed users. On the contrary, visual specification languages use intuitive notations and aiming at easing the specification and understanding of software systems.

Moreover, these languages and concomitant environments permit to automatically generate graphical interfaces or editors for Domain Specific Languages starting from a meta-model. However, they suffer from a lack of precise semantics. We are interested in combining these two complementary technologies by mapping the elements of the one into the other.

In this paper, we present a generic transformation process from functional data structures, commonly used in proof assistants, to Ecore models and vice-versa. This translation method is based on Model-Driven Engineering and defined by a set of bidirectional transformation rules. These rules are detailed and represented in a formal description. Our approach is implemented in the Eclipse environment and illustrated with a case study.

Keywords Model Driven Engineering · Model Transformation · Formal Methods · Verification

Part of this research has been supported by the project *Verisync* (ANR-10-BLAN-0310)

S. Djeddai, M.Strecker
IRIT (Institut de Recherche en Informatique de Toulouse)
Université de Toulouse
Toulouse, France
E-mail: {firstname.lastname}@irit.fr

M. Mezghiche
LIMOSE, Université de Boumerdès
Faculté des Sciences
Boumerdès, Algeria

1 Introduction

Formal methods using interactive proof assistants such as Coq [6] or Isabelle [18] are increasingly integrated into the software engineering process to verify the correctness of software. They have a solid formal basis and a precise semantics, but they use complex notations that might be difficult to understand for unaccustomed users. On the contrary, Model Driven Engineering (MDE) [4, 20] supplies us with visual specification languages such as class diagrams [11] that use intuitive notations. These languages permit to specify, visualize, understand and document software systems. However, they suffer from a lack of precise semantics. We are interested in combining these two complementary technologies by mapping the elements of the one into the other, using an MDE-based transformation method.

One possible scenario is to define the abstract syntax of a Domain Specific Language (DSL) [7] to be used in the context of a formal verification, and then to generate a corresponding Ecore meta-model in order to be able to use an MDE-based tool chain for further processing. Inversely, the meta-model can than be modified by an application engineer and serve as basis for re-generating the corresponding data types. This operation may be used to find a compromise between the representation of the software architect's wishes on the meta-model and functional data structures used in the proof. Furthermore, the meta-model can be used to easily generate a textual (or graphical) editor using Xtext (respectively GMF: Graphical Modeling Framework) facilities [12].

This work constitutes a first step towards using MDE technology in an interactive proof development. The illustrating example is a Java-like language enriched with assertions developed by ourselves for which no off-the-

shelf definition exists [2]. It constitutes a sufficiently complex case study of realistic size for a DSL. In this paper the transformation is applied only on a part of the Safety Critical Java DSL, corresponding to a method definition. A transformation for the whole language can be found in [8], together with other case studies.

This article is a considerably extended and revised version of a previous conference paper [9]. It provides a more detailed discussion of related work and of our case study, and gives a more formal treatment of the transformation rules for transforming meta-models into data types as used in functional programming (see Section 4). The definition of the inverse translation (see Section 5) is entirely new.

The structure of the paper is as follows: in Section 2, we compare our approach with related work. Then, we present some preliminaries, to introduce the main components of our work in Section 3. Sections 4 and 5 constitute the technical core of the article; they describe the translation from data models used in verification environments, to meta-models in Ecore and vice-versa. We then illustrate the methodology with an example in Section 6, before concluding with perspectives of further work.

2 Related Work

EMF (Eclipse Modeling Framework) [5] models are comparable to Unified Modeling Language (UML) class diagrams [11]. For this reason, we are interested in the mappings from other formal languages to UML class diagrams and back again. Some research is dedicated to establishing the link between these two formalisms. We cite the work of *Idani & al.* that consists in a generic transformation of UML models to B constructs [14] and vice-versa [13]. The authors propose a meta-model based transformation method defining a set of structural and semantic mappings from UML to B (a formal method that allows to construct a program by successive refinements, using abstract specifications).

Similarly, there is an MDE based transformation approach for generating Alloy (a textual modeling language based on first order logic) specifications from UML class diagrams and backwards [1, 21].

The purpose of these methods is to generate UML components from a formal description and backwards but their formal representation is significantly different from our needs: functional data structures used in proof assistants.

Additionally, graph transformation tools [10,16] permit to define source and target meta-models all along with a set of transformation rules and use graphical representations of instance models to ease the transformation process. However, the verification functionality they offer is often limited to syntax, typing and structural aspects (such as confluence of transformation rules).

Notable exceptions are codings of graph transformations as transition systems [3, 23] for model checking properties of the transformation system, such as invariants or reachability. These approaches are not applicable in our context as we aim at modeling deeper semantic properties (such as an operational semantics of a programming language and proofs by bisimulation that often require inductive arguments).

Our work aims at narrowing the gap between interactive proof and meta-modeling by offering a way to transform data structures used in interactive provers to meta-models and vice-versa.

3 Preliminaries

3.1 Methodology

Model Driven Engineering is a software development methodology where the (meta-)models are the central elements in the development process. A meta-model defines the constituents of a language. The instances of theses constituents are used to construct a model of the language. A model transformation is defined by a mapping from elements of the source meta-model to those of the target meta-model. Consequently, each model conforming to the source meta-model can be automatically translated into an instance model of the target meta-model. The Object Management Group (OMG) [19] defined the Model Driven Architecture (MDA) standard [15], as specific incarnation of the MDE.

We apply this method in order to define a generic transformation processes from datatypes (used in functional programming) to Ecore models and backwards. Figure 1 shows an overview of our approach. For the first direction of the translation, we derive a meta-model of datatypes starting from an EBNF representation of the datatype definition grammar [18]. This meta-model is the source meta-model of our transformation. We also define a subset of the Ecore meta-model [12] to be the target meta-model. In order to perform this transformation, we defined a set of transformation rules (detailed in Section 4) that maps components of the meta-model of datatypes to those of Ecore meta-models.

We use the mapping between the constructs of the two meta-models to define the reverse direction transformation rules in order to ensure the bidirectionality of the transformations. Bidirectionality [22] is one of

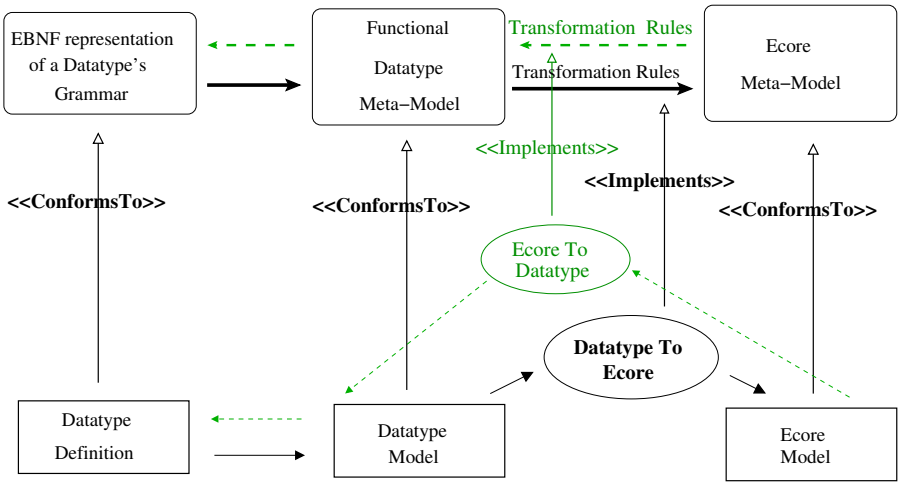


Fig. 1 Overview of the Transformation Method

the desired options of MDE-based transformations. Indeed, assuming we start from a source model M_S, then we perform a transformation using a function f to get a target model M_T. It is important to derive an equivalent model to M_S, as a result to the application of f^{-1} on M_T. Such a feature requires more restrictions on the Ecore models. This transformation function is not automatically derived from f, it is given in Section 5.

Introductory example: The use of the transformation rules will be illustrated in Section 6 with an example that is an excerpt of a real-life application. To give a flavour of the approach, we here present a tiny meta-model, namely a finite-state automaton.

Figure 2 represents a data type description of an automaton, in this case written in the Caml language. Each automaton is then composed of a list of states and a list of transitions. Every state is composed of an integer value (for identifying the state) and two Boolean values (defining whether a state is an initial state and/or a final state). A transition is then described by two states, a source and a target. Figure 3 consists in the representation of the same automaton as a meta-model in Ecore. This meta-model represents the result of applying our transformation on the presented data types.

```
type automaton =
   state list * transition list

type state = int * bool * bool

type transition = state * state
```

Fig. 2 Automaton Data Types (in Caml)

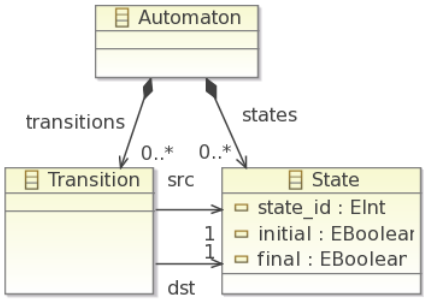


Fig. 3 Automaton Meta-model (in Ecore)

3.2 The Datatype Meta-Model

Functional programming supplies us with a rich way to describe data structures. However, since some features are not supported by Ecore (see Section 4.1 for a discussion), we have only defined a subset that contains the essential element composing datatypes. Figure 4 depicts the datatype meta-model that is constructed from the subset of datatype's declarations grammar presented in Figure 5 [17,18]. We point out that we are mainly interested in data structures. They correspond to the static part of the proofs. Except for the case of accessors, the functions are not treated.

A *Module* may contain several *Type Definitions*. Each *Type Definition* has a *Type Constructor*, which corresponds to the data type's name. It is also composed of at least one *Constructor Declaration*. These declarations are used to express variant types: a disjoint union of types. A *Type declaration* has a name, it is the name of a particular type case. It takes as argument some (optional) *type expressions* which can either represent a *Primitive Type* (`int`, `bool`, `float`, etc.) or also a data type defined previously in the *Module*. The *list* notation introduces the predefined data structure for lists. The

type option describes the presence or the absence of a value. The *ref* feature is used for references (pointers).

We enriched the type definition grammar with a specific function named *Accessor* (see Figure 6). It is introduced by the annotation `(*@accessor*)`. It allows assigning a name to a special field of the type declaration. This element is essential for the transformation process, its absence would lead to nameless structural features.

Representing Generic Types in Functional Programming Parameterized types are important features in functional programming. They are used to express polymorphic data structures. They are comparable to generics in Java and templates in C++. They permit to build different data structures that accept any kinds of values. Each definition of a parameterized type is formed of a *Type Constructor* and a set of *Type Parameter*s. The type expressions then can contain a previously defined parameterized type or one of the specified parameters.

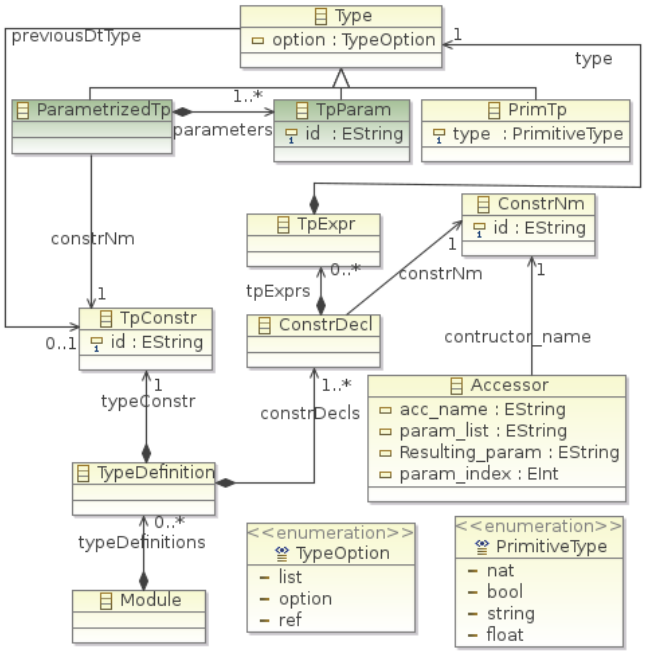


Fig. 4 Datatype Meta-model

3.3 The Ecore Meta-Model

Our destination meta-model is a subset of the Ecore meta-model. Ecore is the core language of EMF [5] which permits to build Java applications based on model definitions and to integrate them as Eclipse plug-ins.

Module	::=	`Module` *ModuleNm* {*typeDefinition*}*
typeDefinition	::=	`type` [*tpParams*] *tpConstr*
		`=`*constrDecl* { `|` *constrDecl* }
constrDecl	::=	*constrNm*
			constrNm `of` *comp-tpExpr* {`*` *comp-tpExpr*}
tpExpr	::=	*type*[*typeOption*]
typeOption	::=	`list`|`option`|`ref`
tpParams	::=	`(`*tpParam* { `,` *tpParam* }`)`
type	::=	[*tpParams*] *tpConstr*
			tpParam
			primitiveType
primitiveType	::=	`int` | `float` | `bool` | `string`

Fig. 5 Part of the Caml Data Types Grammar

```
(*@ accessor *)
    let  acc_name_i  (constr-name  (x_1, ..., x_n) ) =  x_i
                                                  /  1 ≤ i ≤ n
```

Fig. 6 Syntax of Accessor Functions in Caml

The Meta Object Facility (MOF) set by the OMG defines a subset of UML class diagrams [11]. It represents the meta-meta-model of UML. Ecore is comparable to MOF but simpler. They are similar in their ability to specify classes, structural and behavioral features, inheritance and packages.

To implement our approach, we use Eclipse and its core language Ecore. However, it would be possible to choose other solutions [16]. This choice is due to the place of Eclipse for meta-modeling and development, in particular its offer of a wide range of highly integrated tools.

Figure 7 represents a subset of the Ecore language. It contains essentially the elements needed for our transformation process. Its main components are:

- The `EPackage` is the root element in serialized Ecore models. It encompasses `EClasse`s and `EDataType`s.
- The `EClass` component represents classes in Ecore. It describes the structure of objects. It contains `EAttribute`s and `EOperation`s.
- The `EDataType` component represents the types of `EAttribute`s, either predefined types (Integer, Boolean, Float, etc.) or defined by the user. There is a special datatype to represent enumerated types `EEnum`
- `EReference`s is comparable to the UML Association link. It defines the kinds of the objects that can be linked together. The `containment` feature is a Boolean value that makes a stronger type of relationships. When it is set to true, it represents a whole/part relationship.

Representing Generics Ecore has been extended to support parametric polymorphism. Actually, parameterized types and operations can be specified, and types

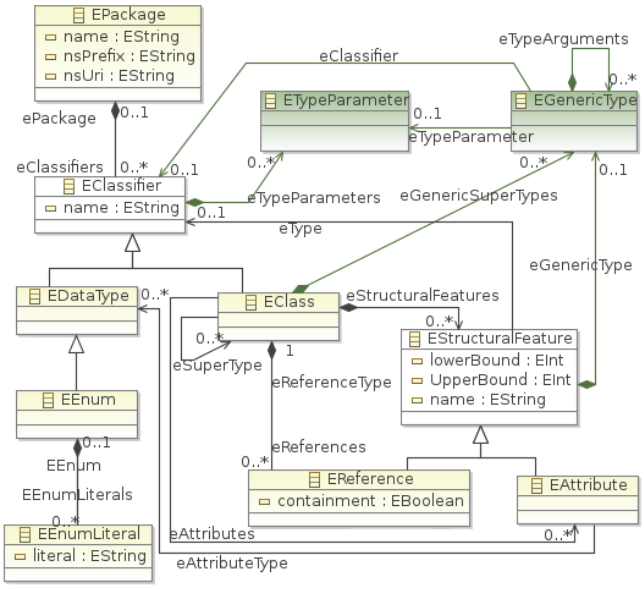


Fig. 7 Simplified subset of the Ecore Meta-model

with arguments can be used instead of regular types. The changes are represented in the Ecore meta-model mainly in two new classes `EGenericType` and `ETypeParameter` (they are distinguishable from the others on the Figure 7 by the green color). A parametrized type is then represented by a simple `EClass` that contains one or more `ETypeParameter`s. An `EGenericType` represents an explicit reference to either an `EClassifier` or an `ETypeParameter` (but not both at the same time). The `eTypeArguments` reference is used to contain the `EGenericType`s representing the type parameters.

4 From Datatypes to Meta-Models

This part details the automatic translation from functional data types to meta-models. It represents the first direction of translation. To precisely define transformation rules, the transformation method is presented in a formal notation in the form of a function noted *Tr()*. The transformation rules are presented as sub-functions relative to the component given as input. In each rule definition, we start by an informal description, then we present it formally and finally we show an effective example. The Table 1 summarizes the mappings performed between the elements of the two meta-models.

$$Tr : DataTypes \longrightarrow Ecore\ Meta\text{-}model$$

The following translation sub-functions are given for a concrete syntax in the style of Caml [17]. Since most functional languages (including the language of proof assistants) have great similarities, the concrete syntax can be mapped to different functional languages.

4.1 Well-formedness Constraints for Input Data Types

Our translation does not treat all of the features typically present in functional programming languages such Isabelle and Caml. The primary reason is that some features which are specific to functional programming have no counterpart in Ecore. This is particularly true of higher-order constructors, *i. e.* constructors taking functions as arguments.

Even though we allow reference types (stateful programming can be simulated in functional programming by a monadic style), we exclude some mutable data structures, in particular arrays. Also, for now, we have not implemented a treatment for mutually recursive types, except for the list, reference and option type constructors. Genuine mutual recursion considerably complexifies the transformation procedure, but apart from the exceptions mentioned, only occurs rarely in practice.

However, as presented previously in Section 3.2, we treat primitive types (integers, Booleans, floats, strings) and user defined data types, represented in variant types. We allow the use of parameterized types and our subset detects the use of Caml keywords introducing lists, references and type option.

4.2 Rule ModuleToEPackage

In ML programs (respectively in the Isabelle proof assistant), it is possible to group portions of programs into *modules*. We decided to represent these modules by `EPackage`s in Ecore. They are used to gather `EDataType`s and `EClass`es. Thus, the transformation process consists of creating an `EPackage` for each module. The name of the corresponding `EPackage` is the module name. We have also to specify the prefix and the URI of the XML namespace by instantiating the `NsPrefix` and `NsURI` values. To translate the data types contained in the module, we call the function $Tr_{dtp}()$ for each type definition.

$$\begin{aligned} Tr_{module}(Module\ md\_name\ Dtp_1 ... Dtp_n) = \\ & createEPackage(); \\ & setName(md\_name); \\ & setNsPrefix(md\_name); \\ & setNsURI(\text{"}http://md\_name/1.0\text{"}); \\ & Tr_{dtp}(Dtp_i) \qquad / \ 1 \leq i \leq n \end{aligned}$$

4.3 Rule DatatypeToEClass

This rule is applied when the datatype is formed of only one constructor. The latter is translated into an `EClass`. The `EClass` name is the name of the type constructor.

Functional Data Structures	Ecore Components
Type Constructor + Constructor	`EClass`
Record	`EClass`
Type Constructor + Constructors + Type Expressions	`Inheritance`
Type Expression(Primitive Type)	`EAttribute`
Type Expression	`EReference`
Type options	`Multiplicities`
Type Constructor + Constructors (without *Type Expressions*)	`EEnum`
Constructor (without *Type Expression*)	`EEnumLiteral`
Parameterized Datatype	`EGenericType`
Type Parameter	`ETypeParameter`

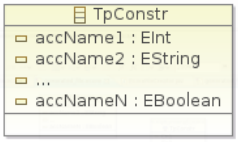
Table 1 Correspondence between elements of the two meta-models

The types composing the datatype are translated using other rules (*PrimitivTypeToEAttribute* or *TypeToEReference*).

$$Tr_{dtp}(tpConstr = cn\ t_1...t_n) = createEClass();\ setName(tpConstr);\ Tr_{type}(acc_i, t_i)\ /\ 1 \leq i \leq n$$

Example:

`type` *tpConstr* `=`
Cn `of int` $*$ `string`$*$
$...*$ `bool`



4.4 Rule DatatypeToEEnum

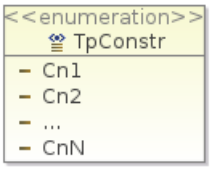
Datatypes composed only of constructors (without type expressions *typexpr*) are translated into `EEnum`s which are usually employed to model enumerated types in Ecore. Then, each constructor composing the datatype is translated into a literal named `EEnumLiteral`. The name of each constructor becomes the name of a literal.

$$Tr_{dtp}(tpConstr = cn_1|...|cn_p) = createEEnum();\ setName(tpConstr);\ Tr_{constrNm}(cn_i);\ /\ 1 \leq i \leq p$$

$$Tr_{constrNm}(cn_i) = EEnumLiteral(cn_i);\ /\ 1 \leq i \leq p$$

Example:

`type` *tpConstr*`=`
Cn1 |*Cn2*|... | *CnN*



4.5 Rule DatatypeToEClasses

When constructor declarations are composed of more than one constructor declaration containing type expressions: a first `EClass` is created to represent the type constructor (*tpConstr*). Then, for each constructor, an `EClass` is created too, and inherits from the *tpConstr* one. To transform the type expressions of each constructor, we call the functions for translating the type expressions.

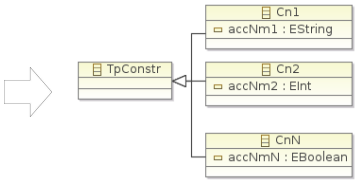
$$Tr_{dtp}(tpConstr = cd_1|...|cd_n) = createEClass();\ setName(tpConstr);\ Tr_{decl}(cd_i, tpConstr)\ /\ 1 \leq i \leq n$$

$$Tr_{decl} : ConstructorDeclaration \longrightarrow EClass$$

$$Tr_{decl}(cn_i\ t_1...t_m, tpConstr) = createEClass();\ setName(cn_i);\ setSuperType\ (EClass(tpConstr));\ Tr_{type}(acc_j, t_j) \quad / \ 1 \leq j \leq m$$

Example:

`type` *tpConstr* `=`
Cn1 `of string`
|*Cn2* `of int`
|...
|*CnN* `of bool`



4.6 Rule PrimitiveTypeToEAttribute

If a type expression is formed of a primitive type, the translation function generates a new `EAttribute`. The

name of this `EAttribute` is the name of its corresponding accessor, and its type is the EMF representation of the primitive type : `EInt` for *int*, `EBoolean` for *bool*, `EString` for *string*, etc.

$$\begin{array}{lcl} Tr_{type} : (accessor, type) & \longrightarrow & EStructualFeature \\ Tr_{type}(acc, primTp) & = & createEAtrribute(); \\ & & setName(acc); \\ & & setType(primTp_{EMF}); \end{array}$$

Example: Same example presented in Section 4.3.

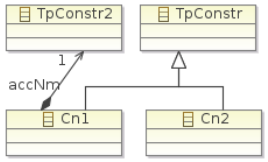
4.7 Rule TypeToEReference

When a type expression contains a type which is not a primitive type, the latter has to be previously defined in the Isabelle *theory*. Then, a containment link is created between the current `EClass` and the `EClass` referenced by the type constructor, and the multiplicity is set to `1`.

$$\begin{array}{lcl} Tr_{type} : (accessor, type) & \longrightarrow & EStructualFeature \\ Tr_{type}(acc, tpConstr) & = & createEReference(); \\ & & setName(acc); \\ & & setType\ (tp\_constr); \\ & & setContainment\ (true); \\ & & setLowerBound(1); \\ & & setUpperBound(1); \end{array}$$

Example:

`type` *tpConstr*=
Cn$_1$ `of` *tpConstr2*
| *Cn*$_2$



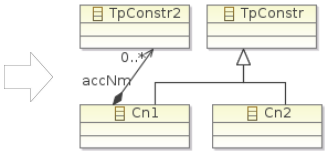
4.8 Rule TypeOptionToMultiplicity

The type expressions can also appear in the form of a *type* `list`. In this case the multiplicity is set to `0...*`. The type expression *type* `option` is used to express whether a value is present or not. It returns `None`, if it is absent and `Some` value, if it is present. This is modeled by changing the cardinality to `0...1`.

$$\begin{array}{lcl} Tr_{type} : (accessor, type) & \longrightarrow & EStructualFeature \\ Tr_{type}(acc, t\ list) & = & Tr_{type}(acc, t) \\ & & setLowerBound(0); \\ & & setUpperBound(*); \\ Tr_{type}(acc, t\ option) & = & Tr_{type}(acc, t) \\ & & setLowerBound(0); \\ & & setUpperBound(1); \end{array}$$

Example:

`type` *tpConstr*=
Cn$_1$ `of` *tpConstr2* `list`
| *Cn*$_2$

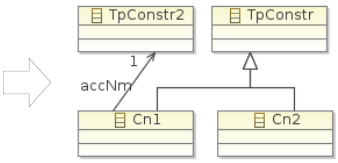


The last case that we deal with is references (*type* `ref`). References are used to represent pointers in ML programming and Isabelle. They are translated into simple references without containment option in Ecore.

$$\begin{array}{l} Tr_{type}(acc, t\ ref) = Tr_{type}(acc, t) \\ \qquad\qquad\qquad\quad setContainment(False); \end{array}$$

Example:

`type` *tpConstr*=
Cn$_1$ `of` *tpConstr2* `ref`
|*Cn*$_2$



4.9 Rule AccessorToStructuralFeaturesName

This rule is spelled out to define how the *accessor_name* is selected for naming a particular `EStructuralFeature`. Accessors are grouped in *accssors_list*. Each accessor structure is formed of an *accessor_name*, a *constructor_name* and an integer value named *index*. This index corresponds to the position of the type concerned by the accessor function.

The elements composing the accessor are used to select which (previously created) structural feature is concerned by the accessor. The accessor name is then used to name the selected `EStructuralFeature`.

The details of the process are given formally by the following representation.

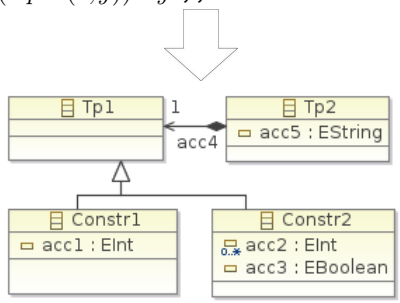
$$Tr_{acc} : Accessor \longrightarrow EStructualFeature$$

$$\begin{array}{l} Tr_{acc}(acc) = \\ \quad Tr_{acc}(acc\_name, constr\_nm, i) \\ \quad eCl\_list := package.getEClassifier(); \\ \quad select\_eCl := eCl\_list.search\_by\_name(constr\_nm); \\ \quad eSF\_list := select\_eCl.getEAllStructuralFeatures(); \\ \quad select\_eSF.set\_Name(acc\_name); \end{array}$$

Here is an example of transforming a data type description together with accessor functions into a class diagram represented in Ecore.

```
type tp1= Constr1 of int
        | Constr2 of (int list)* bool
type tp2 = Tp2 of tp1 * string

(*@accessor*)
let acc1 (Constr1(x)) =x ;;
(*@accessor*)
let acc2 (Constr2 (x,y)) =x ;;
(*@accessor*)
let acc3 (Constr2 (x,y)) =y ;;
(*@accessor*)
let acc4 (Tp2 (x,y)) =x ;;
(*@accessor*)
let acc5 (Tp2 (x,y)) =y ;;
```



4.10 Transforming Generics

In case the datatype definition is polymorphic, it is translated into the representation of generics in the meta-model. It consists in creating an `EClass` to represent the *Type Constructor* and for each type parameter creating an `ETypeParameter` related to the `EClass` via the `eTypeParameters` reference. Notice that we have to create an `EGenericType` for each class and type parameters (related to their `EGenericType` via the reference: `eTypeArguments`) each time we intend to use the `EClass` as a generic. Then, for each *constructor declaration*:

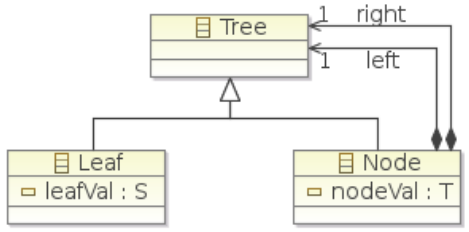
- Create an `EClass` to represent the *Constructor Declaration* which has the same `ETypeParameters` as the *Type Constructor* one.
- Setting its `eGenericSuperType` referring to the generic type representing the *Type Constructor* `EClass`.

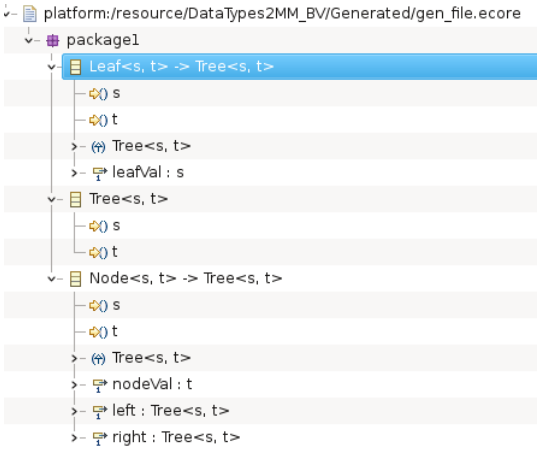
When it comes to use these generics to type `EStructuralFeatures`, we are faced with two scenarios. First, when the type expression is a *type parameter*. The `EStructuralFeature` is typed with an `EGenericType` referring to the `ETypeParameter` of the containing `EClass`. If instead the *type expression* corresponds to a *parameterized type* with *type parameters* it is typed with an `EGenericType` representing the `EClass` with `ETypeParameters`.

To clarify this process, we use the example below. It consists in transforming a parametrized *tree* data type. It has two parameters: the first corresponds to the type of leaves and the second to the type of values contained in a node. The result after performing the translation is displayed in the arborescent Ecore editor. The `EGenericTypes` are not explicitly represented in the `EcoreDiagram`.

Example:

```
type ('s,'t) tree =
    Leaf of 's
   |Node of 't *
('s,'t) tree * ('s,'t) tree
```





5 From Meta-Models to Datatypes

In this section, we present the second direction of the translation: from meta-models into data structures used in functional programming. We start by defining some well-formedness conditions on the entry meta-model. Next, we detail one by one the different transformation rules. As in the previous section, transformation rules are presented in the natural language with a formal notation. To avoid overloading the notation, we use again the notation $Tr()$ to represent the translation function.

5.1 Well-formedness Constraints for Input Meta model

To perform the reverse direction of the transformation, we draw heavily on the mapping performed on the forward translation (Section 4). In our view, it is important to successfully implement a function that is the inverse of the one from datatype to meta-models. Indeed, it seems important to insure that the composition of the two opposite transformation functions gives identity, even if it leads us to impose some additional restrictions on the meta-model. In the forthcoming PhD thesis of the first author [8], these well-formedness constraints are spelled out in more detail.

The first restriction concerns the depth of inheritance relations: the transformation of a meta-model containing inheritance of classes on more than one level (a class that inherits from a class that inherits from another one etc.) is not supported by our rules.

The second restriction aims at avoiding mutually dependent data types. We therefore define a partial order $\prec$ on classes for the transformation of `EClassifiers` contained in an `EPackage`. The `EEnums` have to be translated first, because they don't depend on any other elements. The `EClasses` left in the `EPackage` have then to be ordered using two criteria:

- **The Inheritance relation:** if an `EClass` C_1 is a `superType` (used in Ecore for determining a super class) of another `EClass` C_2, then C_1 has to be translated before C_2. We therefore add the constraint $C_1 \prec C_2$.
- **The reference relation:** if an `EClass` C_1 is a target (`eType` in Ecore) of an `EReference` belonging to another `EClass` C_2, then C_1 has to be translated before C_2, thus $C_1 \prec C_2$.

This order allows us to define the second well-formedness criterion: The order $\prec$ generated by the above two constraints has to be acyclic.

The last constraint we impose on the models is about inheritance and genericity. Indeed, if we have an inheritance relation between two generics (represented by `EClasses` with `ETypeParameters`), all the parameters used by the child class have to appear in the super class.

5.2 Rule EPackageToModule

The elements composing Ecore models can be gathered into `EPackages`. When we perform the translation from Ecore models to functional data type descriptions, we transform these packages into modules. The name of a particular `EPackage` gives the name of the *module*. The additional elements `nsPrefix` and `nsURI` are specific features of Ecore. They are not translated and not used in the functional description.

$$
\begin{aligned}
&Tr\ (\texttt{ePackage name} = p \\
&\qquad\qquad \texttt{nsPrefix} = pp \\
&\qquad\qquad \texttt{nsURI} = puri \\
&\qquad\qquad \{ECl_1\ ...\ ECl_n\}) = \\
&\quad createModule(); \\
&\quad setName(p); \\
&\quad Tr_{Cl}(ECl_i); \qquad / \ 1 \leq i \leq n
\end{aligned}
$$

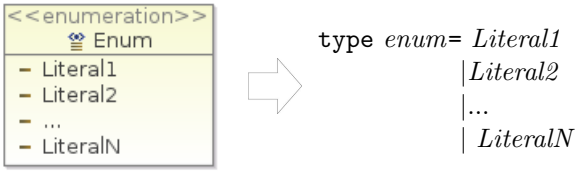
5.3 Rule EEnumToDatatype

Enumerated types are represented in Ecore by `EEnums`. To translate an `EEnum`, we first get all the `EClassifiers` contained in the `EPackage`, check for their instances and transform them to a data type definition composed of *constructors* without type expressions. Each `EEnumLiteral` is mapped to a *constructor* name.

$$
\begin{aligned}
&Tr_{Cl}(\texttt{eEnum name} = e \\
&\qquad\qquad \{ELit_1\ ...\ ELit_n\}) = \\
&\quad createDatatype(); \\
&\quad NewTp\_Constr(); \\
&\quad setName(e); \\
&\quad Tr_{Lit}(ELit_i); \qquad / \ 1 \leq i \leq n
\end{aligned}
$$

$$
\begin{aligned}
&Tr_{Lit}(\texttt{literal} = l) = \\
&\quad createConstructor(); \\
&\quad setName(l);
\end{aligned}
$$

Example:



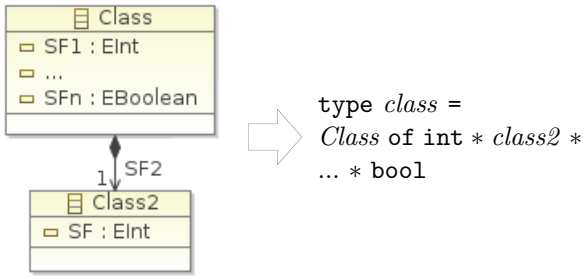
5.4 Rule EClassToDatatype

The simplest case that we deal with is the one consisting in transforming a simple `EClass` which is not related with other `EClasses` by any inheritance link. In such a case, the `EClass` is translated into a single *type definition* without *constructor* declarations. The `EClass` name gives the *type constructor* name. Then, for each `EStructuralFeature` contained in the `EClass`,

we call the appropriate sub-function: Tr_{SF} that stands for Translate Structural Feature.

Tr_{Cl}(`eClass name` $= c$
$\{ESf_1 \ldots ESf_n\}) =$
$if(c.is\_superType() == false)$
$createDatatype()$;
$setName(c)$;
$Tr_{SF}(ESf_i)$; $/\ 1 \leq i \leq n$

Example:



5.5 Rule EClassInheritenceToDatatype

This rule transforms an `EClass` hierarchy into a *type definition*. When we are faced with an `EClass` transformation, we first check if it is a `SuperType` of other classes. In such a case, we create a new data type definition named with the `EClass` name. Then, we select all the classes that inherit from this super class. For each of them, we apply the rule *EClassToConstructor*.

If the super class is a generic type (an `EClass` augmented with `ETypeParameters`) we call the function $Tr_{prm}()$ for every `ETypeParameter`.

Tr_{Cl}(`eClass name` $= sClass$
$\{ESf_1 \ldots ESf_n\}$
$\{ETp_1 \ldots ETp_n\}) =$
$if(c.is\_superType() == true)$
$createDatatype()$;
$setName(sClass)$;
$class\_list = select\_child\_classes$;
$Tr_{ch\_cl}(class_i)$ $/\ class_i \in class\_list$
$Tr_{prm}(ETp_i)$; $/\ 1 \leq i \leq n$

5.5.1 Rule EClassToConstructor

Thanks to this rule, each (child) `EClass` is transformed into a constructor declaration in the corresponding type definition. First, a new *constructor* is created, the name of the *constructor* is the `EClass` name. Then for each `EStructuralFeature` contained in the `EClass` the function Tr_{SF} is called. The rules *EAttributeToType* (Section 5.6) and *EReferenceToType* (Section 5.7) are applied depending on the nature of the `EStructuralFeature`.

Tr_{ch_cl}((`eClass name` $= c$
`eSuperType` $= sClass$
$\{ESf_1 \ldots ESf_n\}) =$
$setName()$;
$createConstructor(c)$;
$Tr_{SF}(ESf_i)$; $/\ 1 \leq i \leq n$

The rule is applied in the same way when the super class is generic (in this case we have `eGenericSuperType` instead of `eGenericType`).

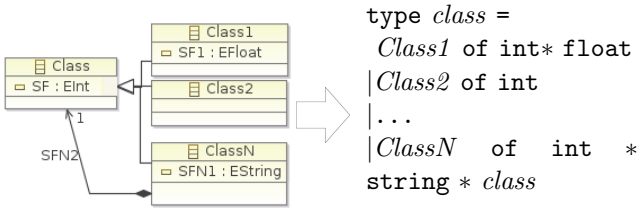
Tr_{ch_cl}(`eClass name` $=\ c$
`eGenericSuperType` $= sClass$
$\{ESf_1 \ldots ESf_n\})$

5.5.2 Rule ETypeParamaterToTypeParameter

Clearly, the `ETypeParameters` used in the representation of generics in Ecore, are translated into their equivalents in functional programming: *type parameters*.

Tr_{prm}(`eTypeParameter name` $= tp) =$
$createTypeParameter()$;
$setName(tp)$;

Example:



5.6 Rule EAttributeToType

Transforming each `EAttribute` consists in creating a new *type expression* in the corresponding *constructor declaration* (or *type definition*). This corresponding element can be selected by name in the list of created data types. `EAttribute`'s type becomes the equivalent type in the functional language (using the transformation function Tr_{Type}).

To translate the upper and lower bounds, the function Tr_{Bnd} is called.

$$\begin{aligned} &Tr_{SF}(\texttt{eAttribute name} = a \\ &\qquad\qquad LowerBound \\ &\qquad\qquad UpperBound \\ &\qquad\qquad EType) = \\ &\quad createTypeExpression(Tr_{Type}(EType)); \\ &\quad Tr_{Bnd}(LowerBound, UpperBound); \end{aligned}$$

$$\begin{aligned} Tr_{Type}(\texttt{eType} = \texttt{EInt}) &= \texttt{int} \\ Tr_{Type}(\texttt{eType} = \texttt{EBoolean}) &= \texttt{bool} \\ Tr_{Type}(\texttt{eType} = \texttt{EFloat}) &= \texttt{float} \\ Tr_{Type}(\texttt{eType} = \texttt{EString}) &= \texttt{string} \\ Tr_{Type}(\texttt{eType} = \texttt{eenum e}) &= e \end{aligned}$$

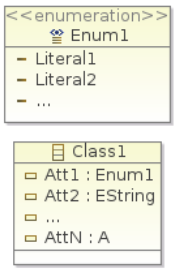
5.6.1 Rule EAttributeToTypeParameter

If the `EAttribute` is typed with an `ETypeParameter` (belonging to a generic type), it is translated (as the precedent case) into a *type expression* consisting of a *type parameter*. The name of the `ETypeParameter` becomes the name of the *type parameter* in the *type expression*.

$$\begin{aligned} &Tr_{SF}(\texttt{eAttribute name} = a \\ &\qquad\qquad LowerBound \\ &\qquad\qquad UpperBound \\ &\qquad\qquad \texttt{eGenericType}\{ETp_1 \ ... \ ETp_n\}) = \\ &\quad createTypeExpression(Tr_{TpPrm}(ETp_i)); \\ &\quad / \ 1 \leq i \leq n \\ &\quad Tr_{Bnd}(LowerBound, \ UpperBound); \end{aligned}$$

$$\begin{aligned} &Tr_{TpPrm}(\texttt{eTypeParameter name} = prm) = \\ &\quad createTypeExpression(prm) \end{aligned}$$

Example:



`type` *enum*`=`
Literal1|*Literal2* |...

`type` *'a class1* `=`
Class1 `of` *enum1* $*$
`string` $*$ ...$*$ *'a*

5.7 Rule EReferenceToType

To translate an `EReference` (pointing to an `EClass` c), we first create (as in the precedent rule) a new *type expression* in the corresponding *constructor declaration* (or *type definition*). This *type expression* is then represented by the name of the `EClass` to whom is targeting the `EReference`. The name of this `EClass` corresponds to a previously translated type definition.

To translate the multiplicities and the containment values we call respectively the functions Tr_{Bnd} and Tr_{cont}.

$$\begin{aligned} &Tr_{SF}(\texttt{eReference} \ Containment \\ &\qquad\qquad \texttt{name} = r \\ &\qquad\qquad LowerBound \\ &\qquad\qquad UpperBound \\ &\qquad\qquad \texttt{eType} = c) = \\ &\quad createTypeExpression(c \ Tr_{cont}(Containment)); \\ &\quad Tr_{Bnd}(LowerBound, UpperBound); \end{aligned}$$

If the Boolean value "containment" is set to False, the translated type expression is augmented with keyword `ref`. `ref` is used to represent pointers in functional languages.

$$Tr_{cont}(\texttt{containment} = \texttt{false}) = ref;;$$

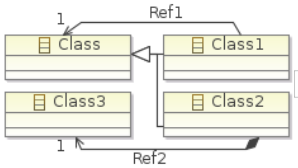
5.7.1 Rule EReferenceToParametrizedType

When the `EReference` target is a generic type (in the shape of an `EClass` augmented with type parameters), a new *type expression* in the corresponding *constructor declaration* (or *type definition*) is created to represent the `EClass`. Next, to each `ETypeParmeter` related to the `EClass` a *type parmater* is created in the *type expression*.

$$\begin{aligned} &Tr_{SF}(\texttt{eReference name} = a \\ &\qquad\qquad LowerBound \\ &\qquad\qquad UpperBound \\ &\qquad\qquad \texttt{eGenericType name} = genTp \\ &\qquad\qquad\qquad\qquad \{ETp_1 \ ... \ ETp_n\}) = \\ &\quad createTypeExpression(genTp) \\ &\quad createTypeParameters(prm_i) \ / \ 1 \leq i \leq n \end{aligned}$$

Where ETp_i has the form :
$ETp_i = \ (\texttt{eTypeParameter name} = \ prm_i) \ / \ 1 \leq i \leq n$

Example:



`type` *class* `=`
Class1 `of` *class* `ref`
|*Class2* `of` *class3*

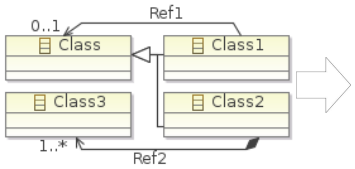
5.8 Rule MultiplicitiesToTypeOptions

This rule permits to translate multiplicity values contained in structural feature definitions (represented by an upper and a lower bound). They are used to determine the number of features that composes an instance. When the `lowerBound` is `0` and the `upperBound` is set to `1`, this signifies that in the instance this `EStructuralFeature` might be present or absent. These values are translated into type `option` in the *type expressions*.

If the `upperBound` is represented by a `*` this implies the ability of creating more than one instance of the concerned `EStructuralFeature`. It is mapped to the type *list* in the data type description.

Tr_{Bnd}(`lowerBound="0"`, `upperBound="1"`) = `option`;
Tr_{Bnd}(`lowerBound="0"`, `upperBound="*"`) = `list`;

Example:



`type` *class* `=`
Class1 `of` *class* `ref option`
|*Class2* `of` *class3* `list`

6 Case Studies

To evaluate our approach we have applied our method on case studies that combine our transformation with the generation of tools for graphical/textual syntaxes. More details can be found in [8]. For lack of space we here illustarte our approach with a part of a description of a DSL.

This DSL is a Java-like language enriched with assertions developed by ourselves for which no off-the-shelf definition exists. It represents a real-time dialect of the Java language allowing us to carry out specific static analyses of Java program (details are described in [2]), where a formal semantics of Java is defined in the Isabelle proof assistant. Because of this application context, we do not use a Caml grammar in this case. Our approach is implemented using the Eclipse environment.

Performing the translation for the whole language description would generate a huge meta-model that could not be presented in the paper. We thus choose to present only an excerpt of it, corresponding to a method definition. Figure 8 shows a datatype taken from the Isabelle *theory* where the verifications were performed.

A method definition (in our DSL) is composed of a method declaration, a list of variables, and statements. Each method declaration has an access modifier that specifies its kind. It also has a type, a name, and some variable declarations. The *stmt* datatype describes the statements allowed in the method body: Assignments, Conditions, Sequence of statements, Return and the annotation statement (for time annotations). In this example we use Booleans, integers, strings for types and values.

```
datatype binop    = BArith| BCompar| BLogic
datatype value    = BoolV bool| IntV int
                    |StringV string| VoidV
datatype binding  = Local| Global
datatype var      = Var binding string
datatype expr     = Const value
                    |VarE var
                    |BinOperation binop expr expr
datatype tp       = BoolT| IntT| VoidT| StringT
datatype stmt     = Assign var expr
                    |Seq stmt stmt
                    |Cond expr stmt stmt
                    |Return expr
                    |AnnotStmt int stmt
datatype accModifier =
 Public |Private |Abstract|Static |Protected |Synchronized
datatype varDecl =
 VarDecl (accModifier list) tp int
datatype methodDecl =
 MethodDecl (accModifier list) tp string (varDecl list)
datatype methodDefn =
 MethodDefn methodDecl (varDecl list) stmt
```

Fig. 8 Datatypes in Isabelle

This part of the *Isabelle theory* was given as input to the implementation of our translation rules presented in Section 4. The resulting Ecore diagram is presented in Figure 9. As it is shown on the figure, data type definitions built only of type constructors (*Tp*, *AccModifier*, *Binop*, *Binding*) are treated as enumerations in the meta-model. Whereas *Datatype MethodDecl* composed of only one constructor derive a single class. As for type expressions that represent a list of types (like *accModifier list* in *varDecl*), they generate a structural feature in the corresponding class and their multiplicities are set to *(0...*)*. The result of type definitions containing more than one constructor and at least a type expression (*stmt* and *expr*) is modeled as a number of classes inheriting from a main one. Finally, the translation of the *int*, *bool* and *string* types is straightforward. They are mapped to respectively `EInt`, `EBoolean` and `EString`.

7 Conclusion

Our work constitutes a first step towards a combination of interactive proof and Model Driven Engineering. We have presented an MDE-based method for transforming data type definitions used in proof assistants to Class diagrams and back again, using bidirectional transformations.

The approach is illustrated with the help of a Domain Specific Language developed by ourselves. It is a Java-like language enriched with annotations. Starting from data type definitions, set up for the semantic modeling of the DSL, we have been able to generate an EMF meta-model. The generated meta-model is used for documenting and visualizing the DSL, it can also be manipulated in the Eclipse workbench to generate a textual editor as an Eclipse plug-in.

We are working on coupling our work with the generation of provably correct object oriented code from proof assistants.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A challenging model transformation. In: G. Engels, B. Opdyke, D.C. Schmidt, F. Weil (eds.) MoDELS, *Lecture Notes in Computer Science*, vol. 4735, pp. 436–450. Springer (2007)
2. Baklanova, N., Strecker, M.: Abstraction and verification of properties of a real-time java. In: V. Ermolayev, H.C. Mayr, M. Nikitchenko, A. Spivakovsky, G. Zholtkevych (eds.) ICT in Education, Research, and Industrial Applications, *Communications in Computer and Information Science*, vol. 347, pp. 1–18. Springer Berlin Heidelberg (2013). DOI 10.1007/978-3-642-35737-4_1. URL http://www.irit.fr/~Martin.Strecker/Publications/icteri2012.html

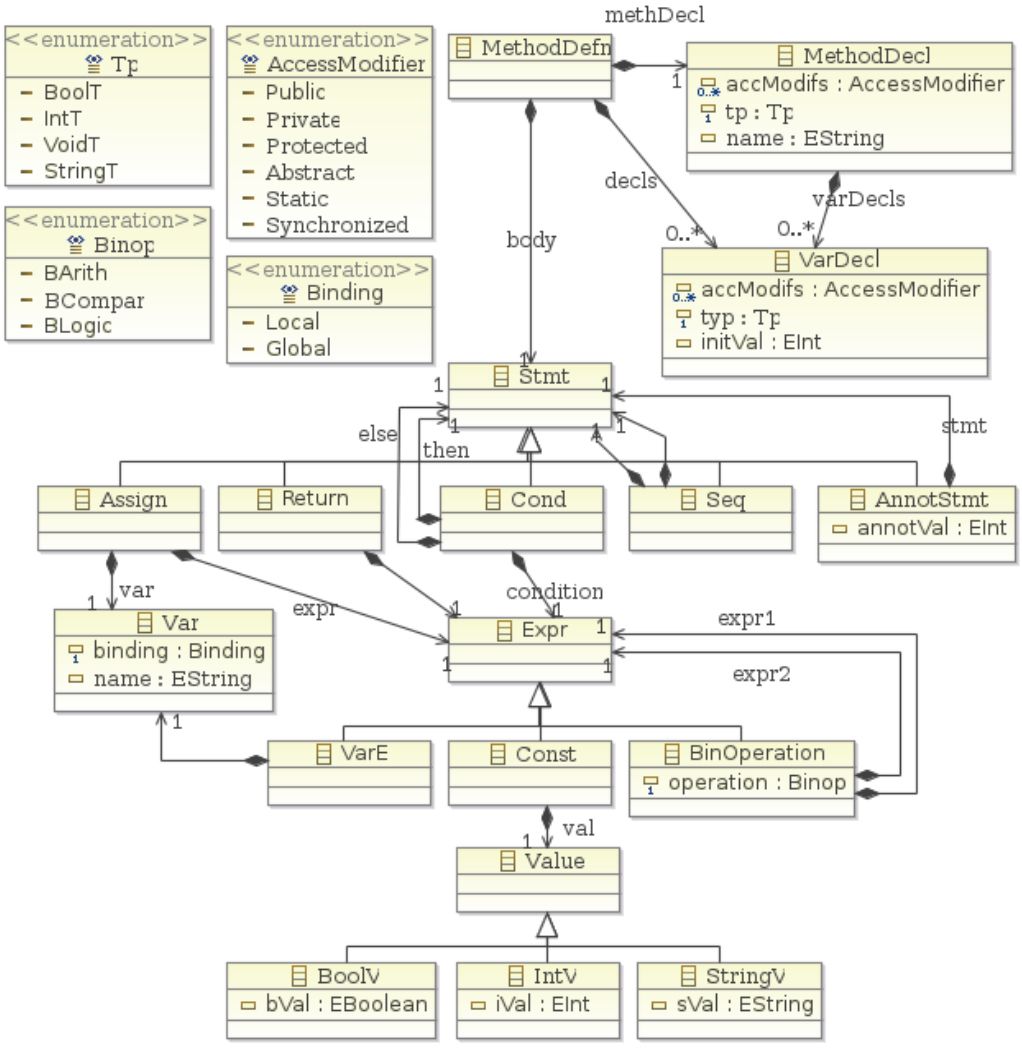


Fig. 9 Resulting Ecore Diagram after Transformation

3. Baldan, P., Corradini, A., König, B.: A framework for the verification of infinite-state graph transformation systems. Information and Computation **206**, 869–907 (2008). URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.155.4078&rep=rep1&type=pdf
4. Bézivin, J.: Model Driven Engineering: An emerging technical space. In: R. Lämmel, J. Saraiva, J. Visser (eds.) Generative and Transformational Techniques in Software Engineering, *Lecture Notes in Computer Science*, vol. 4143, pp. 36–64. Springer Berlin / Heidelberg (2006). DOI http://dx.doi.org/10.1007/11877028_2. URL https://www.uni-koblenz.de/~laemmel/gttse/2005/pdfs/41430036.pdf
5. Budinsky, F., Brodsky, S.A., Merks, E.: Eclipse Modeling Framework. Pearson Education (2003)
6. Coq Development Team: The Coq proof assistant reference manual. version 8.31 (2010). URL http://coq.inria.fr/refman/. http://coq.inria.fr/refman/
7. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. SIGPLAN Notices **35**(6), 26–36 (2000)
8. Djeddai, S.: Combining formal verification environments and model-driven engineering. Ph.D. thesis, Université de Toulouse (2013). http://www.irit.fr/~Selma.Djeddai/PhD_selma_djeddai.html
9. Djeddai, S., Strecker, M., Mezghiche, M.: Integrating a formal development for DSLs into meta-modeling. In: A. Abelló, L. Bellatreche, B. Benatallah (eds.) MEDI, *Lecture Notes in Computer Science*, vol. 7602, pp. 55–66. Springer (2012)
10. Ehrig, K., Ermel, C., Hänsgen, S., Taentzer, G.: Generation of visual editors as Eclipse plug-ins. In: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ASE '05, pp. 134–143. ACM, New York, NY, USA (2005). DOI http://doi.acm.org/10.1145/1101908.1101930. URL http://doi.acm.org/10.1145/1101908.1101930
11. France, R.B., Evans, A., Lano, K., Rumpe, B.: The UML as a formal modeling notation. Computer Standards & Interfaces **19**(7), 325–334 (1998)
12. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley, Upper Saddle River, NJ (2009)
13. Idani, A.: UML models engineering from static and dynamic aspects of formal specifications. In: T.A. Halpin, J. Krogstie, S. Nurcan, E. Proper, R. Schmidt, P. Soffer, R. Ukor (eds.) BMMDS/EMMSAD, *Lecture Notes in Business Information Processing*, vol. 29, pp. 237–250. Springer (2009)
14. Idani, A., Boulanger, J.L., Philippe, L.: A generic process and its tool support towards combining UML and B for safety critical systems. In: G. Hu (ed.) CAINE, pp. 185–192. ISCA (2007)
15. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained : The Model Driven Architecture : Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)
16. de Lara, J., Vangheluwe, H.: Using AToM^3 as a meta-case tool. In: ICEIS, pp. 642–649 (2002). URL http://www.cs.mcgill.ca/~hv/publications/02.ICEIS.MCASE.pdf
17. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml system release 3.12. documentation and user's manual. Online (2011). http://caml.inria.fr/pub/docs/manual-ocaml/index.html
18. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL. A Proof Assistant for Higher-Order Logic. LNCS 2283. Springer Verlag (2002). URL http://isabelle.in.tum.de
19. OMG: Meta Object Facility (MOF) Core v. 2.0 Document (2006). URL http://www.omg.org/spec/MOF
20. Selic, B.: The pragmatics of model-driven development. IEEE Software **20**(5), 19–25 (2003). DOI http://doi.ieeecomputersociety.org/10.1109/MS.2003.1231146
21. Shah, S.M.A., Anastasakis, K., Bordbar, B.: From UML to Alloy and back again. In: S. Ghosh (ed.) MoDELS Workshops, *Lecture Notes in Computer Science*, vol. 6002, pp. 158–171. Springer (2009)
22. Stevens, P.: A landscape of bidirectional model transformations. In: R. Lämmel, J. Visser, J. Saraiva (eds.) GTTSE, *Lecture Notes in Computer Science*, vol. 5235, pp. 408–424. Springer (2007)
23. Varró, D.: Automated formal verification of visual modeling languages by model checking. Software and System Modeling **3**(2), 85–113 (2004). URL http://www.springerlink.com/index/10.1007/s10270-003-0050-x