Combining Verification and MDE Illustrated by a Formal Java Development

Selma Djeddai[1,⋆], Mohamed Mezghiche[2], and Martin Strecker[1,⋆]

[1] IRIT (Institut de Recherche en Informatique de Toulouse)
Université de Toulouse
118 Route de Narbonne, 31062 Toulouse Cedex 9, France
{firstname.lastname}@irit.fr
[2] LIMOSE, UMBB, Boumerdès, Algeria

Abstract. Formal methods are increasingly used in software engineering. They offer a formal frame that guarentees the correctness of developments. However, they use complex notations that might be difficult to understand for unaccustomed users. It thus becomes interesting to formally specify the core components of a language, implement a provably correct development, and manipulate its components in a graphical/textual editor.

This contribution constitutes a first step towards using Model Driven Engineering (MDE) technology in an interactive proof development. It presents a transformation process from functional data structures, commonly used in proof assistants, to Class diagrams in Ecore. To perform the transformation we use an MDE-based methodology. The resulting metamodels from the transformation process are used to generate textual or graphical editors for domain specific languages (DSLs) using tools provided by the Eclipse enviornment. To illustrate this approach we use as example a simple DSL description. It respresents a Java-like language enriched with timing annotations.

Keywords: Model Driven Engineering, Model Transformation, Formal Methods, Verification.

1 Introduction

Domain Specific Languages (DSL) have conquered many different aspects of computer science. They are used in different fields such as aerospace, web-services, multi-media, etc. [8]. Certain DSLs define their semantics in natural languages. However, even though these tend to be quite easy to understand, they usually suffer from incompleteness in some cases and ambiguity in others. Therefore, there emerges a need for defining the formal semantics of DSLs in a mathematically founded framework using proof assistants. Such a phase consists in defining the abstract syntax of a DSL and then grafting a semantics on top

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

V. Ermolayev et al. (Eds.): ICTERI 2012, CCIS 347, pp. 131–148, 2013.

of it, using well-understood mechanisms like structural recursion or inductive relations. Such a semantics is often not executable, but other elements of a formal development are, such as compilers or static analyses whose correctness is proved on the basis of the formal semantics.

Interactive proof assistants such as Coq [6] or Isabelle [18] often use paradigms stemming from functional programming (type systems, function definitions), but they are as such not a programming language. It is however possible to export the formal development to programming languages such as Caml [17] or Scala [19]. A formally verified compiler, for example, can therefore be effectively executed in a standard programming language.

In order to improve the user interface for interacting with a DSL, we aim at a textual or graphical concrete syntax as provided, for example, by the Eclipse Xtext or GMF environments. Frequent changes of the DSL during the design phase make it necessary to adapt this interface easily and to re-generate it automatically, as far as possible.

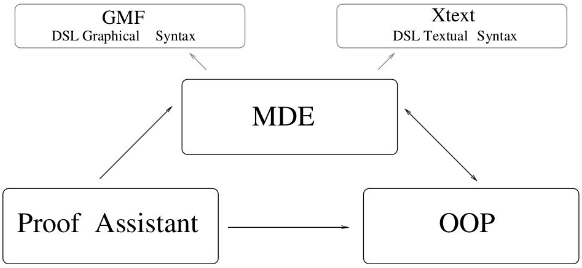


Fig. 1. Meta-modeling(MM), Verification environment and OO languages

Figure 1 depicts the essence of our approach based on studying the interplay of three formalisms that offer different and complementary aspects. On one hand, we have Model Driven Engineering (MDE) [4, 22] that supplies us with frameworks (for example Eclipse Modeling Framework) allowing to specify, visualize and understand DSLs. Also these frameworks are equipped with tools that permit to define graphical and textual syntax for these DSLs (Xtext or Graphical Modeling Framework GMF). They are rather close to Object Oriented programming which is the choice when it comes to developing graphical user interfaces. Besides these facilities, they often suffer from lack of precise semantics.

On the contrary, proof assistants (such as Isabelle) have solid formal bases and precise semantics. They are increasingly used to verify the correctness of software. Nevertheless, they use complex notations that might be difficult to understand for a non-initiated public.

Thus, this work constitutes a first step towards using MDE technology in an interactive proof development. The guiding example (see Section 3) is a Java-like language enriched with assertions developed by ourselves for which no off-the-shelf definition exists. This “meta-model” (in MDE parlance) is sufficiently complex to illustrate the method and to be a case study of realistic size for a DSL. However, its formal model can be entirely defined as an inductive datatype (and this is so for most formally defined languages). In this case study, we can therefore not demonstrate some aspects of our work, such as the translation of genuine graph structures that go beyond instances of inductive data types.

Section 2 constitutes the technical core of the article; it describes a translation from data models in the functional programming world, used in verification environments, to meta models in `Ecore`: the core language of the Eclipse Modeling Framework. We illustrate the methodology in Section 3 with a case study. In Section 4 we compare our work to other approaches, before concluding in Section 5 with perspectives of further work.

2 From Datatypes to Meta-models

In this part, we present in detail the translation process from functional data types to meta-models. We start in Section 2.1 by giving an overview of our methodology, then we introduce the source and the target of the transformation in Sections 2.2 and 2.3 respectively. The essence of the translation is further developed in Section 2.4.

2.1 Methodology

Model Driven Engineering (MDE) is a software development methodology where the (meta-)models are the central elements in the development process. A meta-model defines the elements of a language. The instances of theses elements 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 to an instance model of the target meta-model. The Object Management Group (OMG) [20] 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 process from datatypes (used in ML-style languages and interactive provers) into Ecore models. Figure 2 shows an overview of our approach. Using an EBNF representation of the datatype definition grammar [18], we derive a meta-model of datatypes. 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. The transformation rules are defined on the meta-level and map elements from the source meta-model to their counterparts in the target meta-model. They are detailed in Section 2.4. The *DataTypeToEcore* function implements these rules in Java. It takes as input models which conform to the source meta-model and returns their equivalent in a model which conforms to the target meta-model. The implementation process is further developped in Section 3.2.

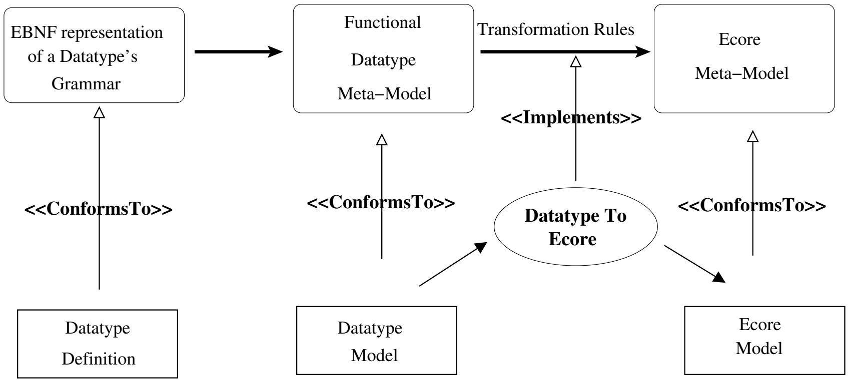


Fig. 2. Overview of the Transformation Method

2.2 Source Meta-model: The Datatype Meta-model

Functional programming is a programming paradigm that implements λ-calculus: a formal system in mathematical logic that formalizes systems through the notion of function. A function, in functional programming, consists in the mapping of elements from a set to another. These sets are called *types*. Usually, they restrict the set of legal programs. We can count among the languages implementing functional programming: *Lisp* [24], *Haskell* [21], and the *ML languages*.

We are interested in *ML languages*. ML stands for Meta Language. It is based on a user-friendly syntax of λ-calculus augmented with polymorphism. It is known for its ability to automatically infer the types of expressions without explicit type annotations. ML languages are considered as non purely functional languages. In fact, they admit the use of mutable data structures, features allowing to program in an imperative way. The most famous dialects of the ML family are SML (Standard ML) and OCaml (Objective Caml) [17].

To perform the transformation, taking all the features provided by ML languages, would be unnecessarily complex, because some features which are specific to functional programming are not used in MDE modeling and would have no equivalent supported by `Ecore`. This is why we defined a subset of data structure schemas provided by ML languages that allows to define data types and that is convenient to be translated into `Ecore` models.

In this subset, we treat primitive types (integers, Booleans, floats and strings) and user defined data types. We allow the use of some keywords introducing lists, references and type option. However, we do not handle mutable constructs and mutable data structures (including arrays). Also for now, we do not implement a specific treatment for mutually recursive types.

Figure 3 depicts the datatype meta-model that is constructed from the subset of datatype declaration grammars of typical functional languages [17, 18]. To construct this meta-model we were inspired by the work of [1] and [25]. They worked widely on defining generic processes to transform EBNF grammars into

Meta-models and vice-versa. We mainly focused on the definition of transformation rules and the correspondence between the elements of the two formalisms. However, we did not use any tools or algorithms developed.

In our subset represented by the meta-model depicted by Figure 3, a *Module* may contain several *Type Definitions*. Each *Type Definition* has a *Type Constructor*. It corresponds to the data types' name. It is also composed of at least one *Constructor Declaration*. These declarations are used to express variant types. *Type declarations* have names, 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* option is used to represent lists in functional programming. The *type option* feature describes the presence or the absence of a value. The *ref* option is used for references (pointers).

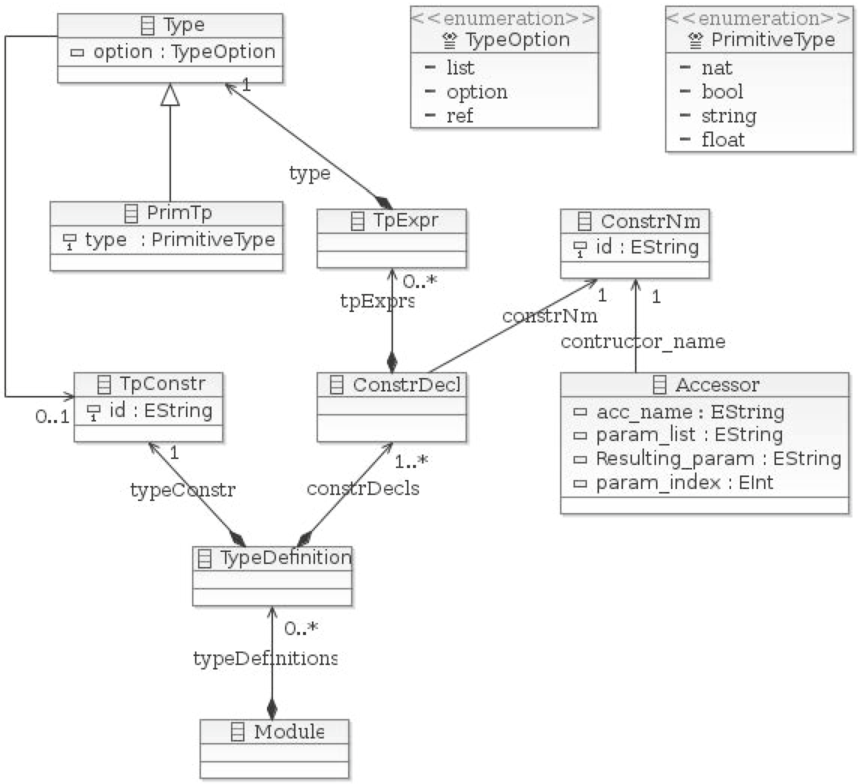


Fig. 3. Datatype Meta-model

We can notice that elements composing type definitions are often unnamed and just expressed with *type expressions*. However, for the rest of our work these typed elements have to be distinguishable by their names. Therefore, we enriched the type definition grammar with a new element named *Accessor*. It is a function introduced by a special annotation `(*@accessor*)`. It allows to assign a name to a special part of the type declaration. These accessor functions are essential for the transformation process, their absence would lead to nameless `EStructuralFeature`s. The syntax of these functions in the OCaml language is presented in Figure 4.

(*@ accessor *)
    let acc_name_i ([*constr-name*] $(x_1, ..., x_n)$) = x_i / $1 \leq i \leq n$

Fig. 4. Syntax of Accessor functions in OCaml

2.3 Target Meta-model: The Ecore Meta-model

Eclipse Modeling Framework (EMF) is an Eclipse framework for building applications based on model definitions. It unifies three technologies: Java, XML and UML. It allows to describe a model as a class diagram, class interfaces in the Java programming language or in the form of an XML schema. Moreover, it is possible to describe a model and generate it in the two others.

`Ecore` is the model that is used to describe and handle models in EMF. It has been developed as a small and simplified implementation of full UML. Its main components are:

– The `EPackage` is the root element in serialized `Ecore` models. It encompasses `EClass`es and `EDataTypes`.
– 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`, each enumeration is called `EEnumLiteral`.
– `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 relations. When it is set to true, it represents a whole/part relationship known as "by-value aggregation" in UML.

The Meta Object Facility (MOF) standardized by the OMG defines a subset of UML class diagram [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. However, their difference appears in the data type structures, package relationships and complex aspects of association links. EMOF (Essential Meta-Object Facility) is the new core meta-model that is very close to Ecore [5].

Figure 5 represents a subset of the Ecore language. This subset contains essentially the elements that are needed for the transformation process. In this meta-model appear only basic classes features and operation. The items that do not appear are not used by our transformation process.

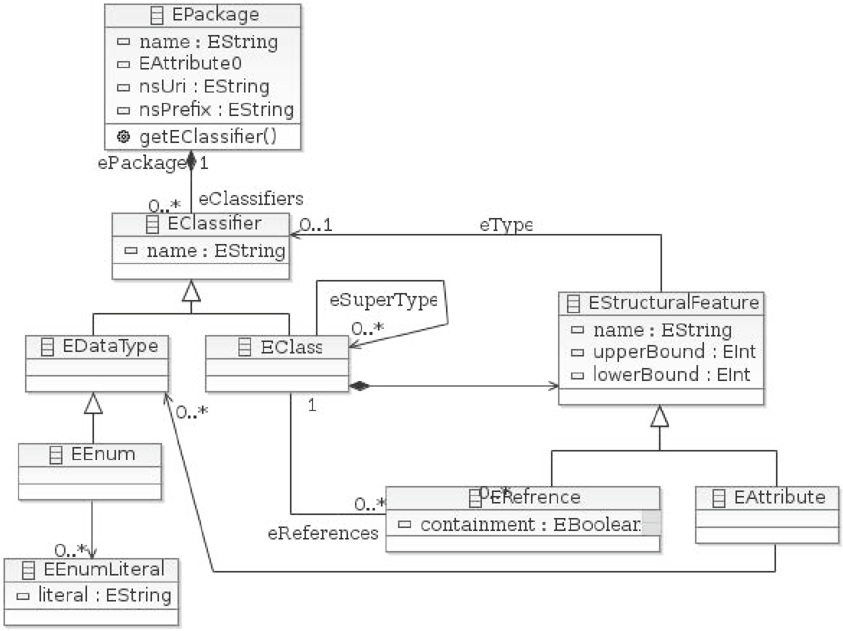


Fig. 5. Simplified subset of the Ecore Meta-model

2.4 From Datatypes to Meta-models

The transformation method is from functional datatypes to Ecore meta-models. 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 relatively 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.

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

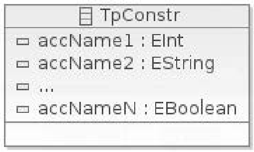
The following translation sub-functions are given for a concrete syntax in the style of Caml [17]. Since most functional languages (including the language of proof assistants) have great similarities, the concrete syntax can be mapped to different functional languages.

Rule DatatypeToEClass. This rule is applied when the datatype is formed of only one constructor. the latter is translated to an `EClass`. The EClass name is the name of the type constructor. The types composing the datatype are translated using other rules (PrimitivTypeToEAttribute or TypeToEReference).

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

Example:

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

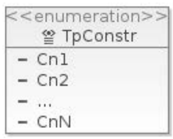


Rule DatatypeToEEnum. Datatypes composed only of constructors (without type expressions *typexpr*) are translated to `EEnums` which are usually employed to model enumerated types in `Ecore`. Then, each constructor composing the datatype is translated into a literal named `EEnumLiteral`. The name of each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

$$\begin{aligned} Tr(tpConstr = cn_1|...|cn_p) &= createEEnum(); \\ & \quad setName(tpConstr); \\ & \quad Tr_{constrNm}(cn_i) \quad /\ 1 \leq i \leq p \\ Tr_{constrNm}(cn_i) &= EEnumLiteral(cn_i) \quad /\ 1 \leq i \leq p \end{aligned}$$

Example:

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

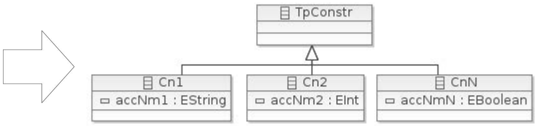


Rule DatatypeToEClasses. When constructor declarations are composed of more than one constructor declaration containing type expressions: a first `EClass` is created to represent the type constructor (*tpConstr*). Then, for each constructor, an `EClass` is created too, and inherits from the *tpConstr*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

$$
\begin{array}{lcl}
Tr(tpConstr = cd_1|...|cd_n) & = & createEClass(); \\
 & & setName(tpConstr); \\
 & & Tr_{decl}(cd_i) \\
 & & /\ 1 \leq i \leq n \\
Tr_{decl} : ConstructorDeclaration & \longrightarrow & EClass \\
Tr_{decl}(cn_i\ t_1...t_m) & = & createEClass(); \\
 & & setName(cn_i); \\
 & & setSuperType\ (EClass(tpConstr)); \\
 & & Tr_{type}(acc_j, t_j) \\
 & & /\ 1 \leq j \leq m
\end{array}
$$

Example:

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



Rule PrimitivTypeToEAttribute. 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 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:

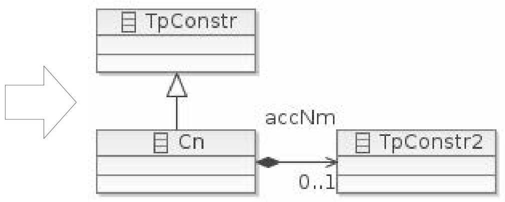
type $tpConstr$ =
Cn of int $*$ string $*$ ...$*$ bool

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 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 `of` *tpConstr2*

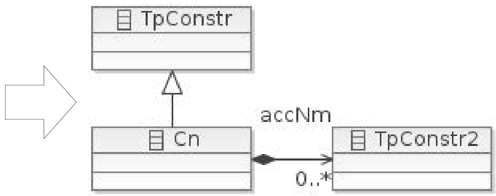


Rule TypeOptionToMultiplicity. The type expressions can also appear in the form of a *type* `list`. In this case the multiplicity is set to `0...*`. The type expression *type* `option` is used to express whether a value is present or not. It returns `None`, if it is absent and `Some` value, if it is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

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

Example:

`type` *tpConstr*=
Cn `of` *tpConstr2* `list`

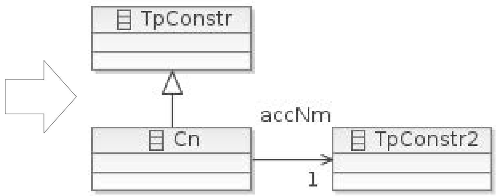


The last case that we deal with is references (*type* `ref`). References are used to represent pointers in ML programming and Isabelle. It is translated to simple references without containment option in Ecore.

$$Tr_{type}(acc, t\ \texttt{ref}) = Tr_{type}(acc, t)$$
$$setContainment(False);$$

Example:

```
type tpConstr=
        Cn of tpConstr2 ref
```

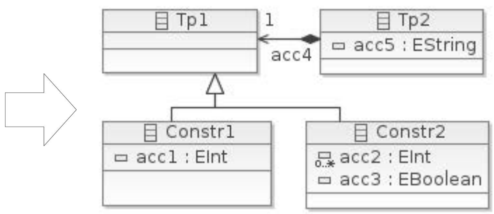


Rule AccessorToStructuralFeaturesName. This rule is spelled out to define how the *accessor_name* is selected for naming a particular `EStructuralFeature`. Accessors are regrouped in *accssors_list*. Each accessor structure is formed of an *accessor_name*, a *constructor_name* and an integer value named "*index*". This index corresponds to the place of the type the accessor is accessing in the type expressions.

The *constructor_name* is used to select the corresponding `EClass` where the `EStructuralFeature` is created. Then the index value is compared to the value `FeatureID` given by Ecore to represent the rank of the `EStructuralFeature` creation in a particular `EClass`. When these values are equal, the corresponding accessor's name is selected to name this `EStructuralFeature`.

Example:

```
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 ;;
```



3 Case Study

In this section, we apply the method presented in Section 2 on a detailed example that consist of a Domain Specific Language. We start by the DSL definition, then we show the architecture of the application before finishing with the effective results of the transformation.

3.1 Presentation of the Case Study

We are currently working on a real-time dialect of the Java language allowing us to carry out specific static analyses of Java programs. We only sketch this language here; details are described in [3]. This language is not a genuine subset of Java, since we have added annotations characterizing timing behavior of program parts that are inserted in particular comments into the program. Neither is the language a superset of Java, because we have to impose syntactic restrictions on the shape of the program, and also static restrictions on the number of objects that are allocated.

All this made us opt for writing our own syntax analysis, which is integrated into the Eclipse Xtext environment [9]. After syntax analysis and verification of the above-mentioned static restrictions, the program together with its timing annotations is translated to Timed Automata (TA) for model checking. The language is currently not entirely stable and will be modified while we refine and improve the translation from Java to TA, and while the formal model evolves.

The formal aspect comes into play at the following point: We are currently developing a real-time semantics of Java in the proof assistant Isabelle, based on an execution semantics using inductive relations. Performing the translation for the whole language description would generate a huge meta-model that couldn't be presented in the contribution. We thus choose to present only an excerpt of it, corresponding to a method definition.

Figure 7 shows the datatype definitions in the Isabelle proof assistant, where a method definition 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 timing annotations). In this example we use Booleans, integers, strings for types and values.

3.2 Implementation: DatatypesToEcore

Our approach is implemented using the Eclipse environment which includes among others

– Eclipse Modeling Framework (EMF) [5]: a framework for modeling and code generation that builds tools and applications based on data models.

– Eclipse Modeling Project (EMP) [12]: a framework allowing the manipulation of DSLs by defining their (textual/graphical) concrete syntax based on a corresponding meta-model using Xtext or GMF tools.

In this chapter we use the Xtext tool [9]. It is a tool that supports the development of textual concrete syntax for DSLs. In the first versions of Xtext, it was only possible to create a DSL textual editor starting from an Extended Backus-Naur Form-like grammar and generating a corresponding Ecore-based meta-model. But since Xtext 2.0, it is possible to start from a meta-model and get the corresponding EBNF-like grammar. Starting from this grammar, the generator creates a parser as well as a functional Eclipse textual editor, complete with syntax highlighting, code assist and outline view [12].

Figure 6 shows the architecture of our application. Non-dashed arrows represent automatic model transformations or code generation. On the contrary, the dashed one stands for a manual intervention added to Xtext code generation facilities. In our approach, the base element is an Isabelle *theory* where both of the datatypes and the properties to be checked are defined. The corresponding meta-model is generated using the translation function described in Section 2.4. Starting from a generated `Ecore` meta-model, we use the Xtext tool to define a textual concrete syntax. First, Xtext builds an EBNF grammar depending on the structure of the meta-model. The grammar is then adapted using the right key words of the language, yielding a textual editor as an Eclipse plug-in. We thus generated code for a DSL textual tool.

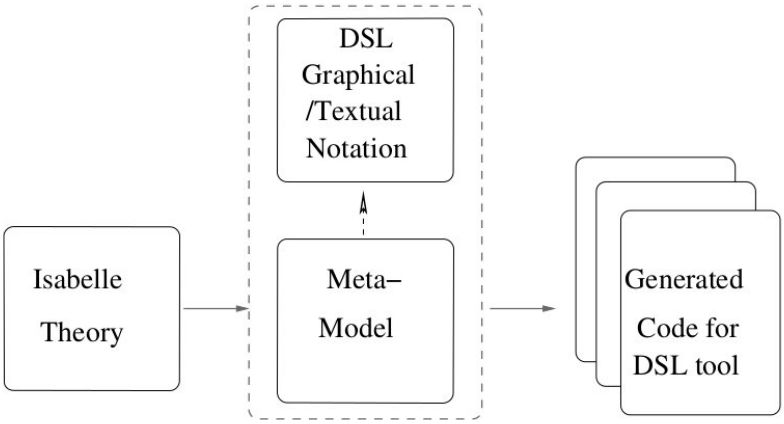


Fig. 6. Datatype To Ecore implementation architecture

3.3 Applying the Transformation

Figure 7 shows datatypes taken form the Isabelle *theory* where the verifications were performed. These datatypes are used to express the elements of a method declaration in our DSL. This part of the *theory* was given as input to the implementation of our translation rules presented in Section 2.4. The resulting `Ecore` diagram is presented in Figure 8.

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 metamodel, whereas *Datatype MethodDecl* composed of only one constructor derive a single class. As for type expressions that represent 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 translated to respectively `EInt`, `EBoolean` and `EString`.

$$
\begin{array}{ll}
\textit{datatype binop} & = BArith |\ BCompar |\ BLogic \\
\textit{datatype value} & = BoolV\ bool \\
 & |IntV\ int \\
 & |StringV\ string \\
 & |VoidV \\
\textit{datatype binding} & = Local |\ Global \\
\textit{datatype var} & = Var\ binding\ string \\
\textit{datatype expr} & = Const\ value \\
 & |VarE\ var \\
 & |BinOperation\ binop\ expr\ expr \\
\textit{datatype tp} & = BoolT |\ IntT |\ VoidT |\ StringT \\
\textit{datatype stmt} & = Assign\ var\ expr \\
 & |Seq\ stmt\ stmt \\
 & |Cond\ expr\ stmt\ stmt \\
 & |Return\ expr \\
 & |AnnotStmt\ int\ stmt
\end{array}
$$

$$
\begin{array}{l}
\textit{datatype accModifier} = \\
\quad Public\ |Private\ |Abstract|Static\ |Protected\ |Synchronized \\
\textit{datatype varDecl} = \\
\quad VarDecl\ (accModifier\ list)\ tp\ int \\
\textit{datatype methodDecl} = \\
\quad MethodDecl\ (accModifier\ list)\ tp\ string\ (varDecl\ list) \\
\textit{datatype methodDefn} = \\
\quad MethodDefn\ methodDecl\ (varDecl\ list)\ stmt
\end{array}
$$

Fig. 7. Datatypes in Isabelle

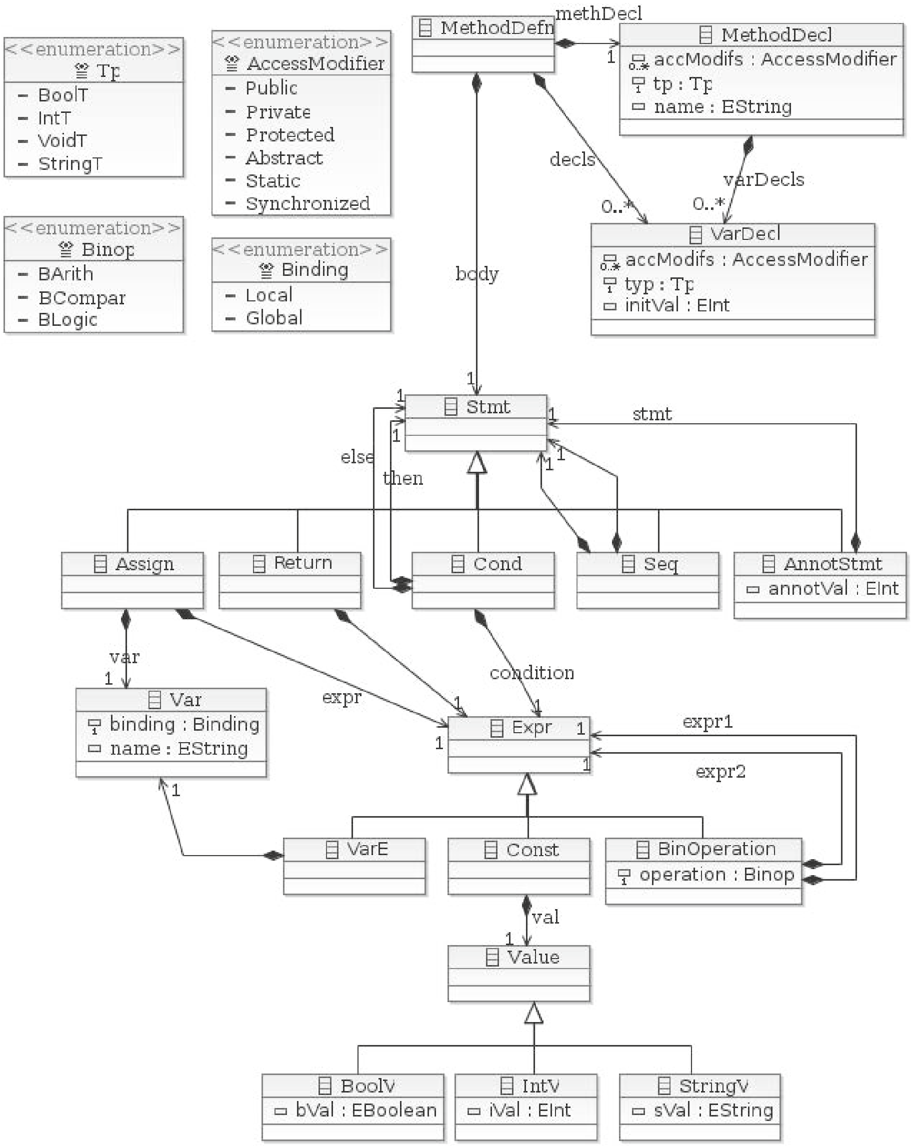


Fig. 8. Resulting Ecore Diagram after Transformation

4 Related Work

EMF models are comparable to Unified Modeling Language Class diagrams. For this reason, we are interested in the mappings from other formal languages to UML Class diagrams. Some research is dedicated to establishing the link between these two formalisms. We cite the work of *Idani & al.* that consists of a generic transformation of UML models to B constructs [14] and vice-versa [13]. The authors propose a metamodel-based transformation method based on defining a set of structural and semantic mappings from UML to B (a formal method that allows to construct a program by successive refinement, 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 [2, 23].

Delahaye & al. describe in [7] a formal and sound framework for transforming Focal specification into UML models.

These methods enable to generate UML components from a formal description but their formal representation is significantly different from our needs: functional data structures.

Also, graph transformation tools [10, 16] permit to define source and target metamodels 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 syntactic aspects (such as confluence of transformation rules) and does not allow to model deeper semantic properties (such as an operational semantics of a programming language and proofs by bisimulation).

Our approach combines the two views by offering the possibility to define the abstract syntax of a DSL, to run some verifications on the top of it and to generate the corresponding metamodel to graphically document the formal developments. Furthermore, this metamodel can be used to easily generate a textual editor using Xtext facilities.

5 Conclusion

Our work constitutes a first step towards a combination of interactive proof and Model Driven Engineering. We have presented a generic method based on MDE for transforming data type definitions used in proof assistants to class diagrams.

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. In addition to its benefits for documenting and visualizing the DSL, it is manipulated in the Eclipse workbench to generate a textual editor as an Eclipse plug-in.

Currently, we are working on extending the subset of data type definitions by adding a way to transform parameterized types to generic types in Ecore, and coupling our work with the generation of provably correct object oriented code from proof assistants. Moreover, we intend to work on the opposite side of the transformation, namely the possibility to generate data structure definitions from class diagrams.

References

1. Alanen, M., Porres, I.: A relation between context-free grammars and meta object facility metamodels. Tech. rep., Turku Centre for Computer Science (TUCS) (March 2003), http://www.cis.uab.edu/courses/cs593/spring2010/TR606.pdf
2. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A Challenging Model Transformation. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 436–450. Springer, Heidelberg (2007)
3. Baklanova, N., Strecker, M., Féraud, L.: Resource Sharing Conflicts Checking in Multithreaded Java Programs. In: Informal Proceedings FAC 2012 (April 2012)
4. Bézivin, J.: Model Driven Engineering: An Emerging Technical Space. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 36–64. Springer, Heidelberg (2006), 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), http://coq.inria.fr/refman/, http://coq.inria.fr/refman/
7. Delahaye, D., Étienne, J.F., Viguié Donzeau-Gouge, V.: A Formal and Sound Transformation from Focal to UML: An Application to Airport Security Regulations. In: UML and Formal Methods (UML&FM), vol. 4, pp. 267–274 (2008), http://cedric.cnam.fr/~delahaye/?page=publis
8. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. SIGPLAN Notices 35(6), 26–36 (2000)
9. Eclipse Community: Tutorials and documentation for Xtext 2.0 (2011), http://www.eclipse.org/Xtext/documentation/
10. Ehrig, K., Ermel, C., Hänsgen, S., Taentzer, G.: Generation of visual editors as Eclipse plugins. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005, pp. 134–143. ACM, New York (2005), 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 (2009)
13. Idani, A.: UML Models Engineering from Static and Dynamic Aspects of Formal Specifications. In: Halpin, T., Krogstie, J., Nurcan, S., Proper, E., Schmidt, R., Soffer, P., Ukor, R. (eds.) BPMDS 2009 and EMMSAD 2009. LNBIP, vol. 29, pp. 237–250. Springer, Heidelberg (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: Hu, G. (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 (2003)

16. de Lara, J., Vangheluwe, H.: Using AToM^3 as a meta-case tool. In: Proceedings of the 4th International Conference on Enterprise Information Systems (ICEIS), Ciudad Real, Spain, pp. 642–649 (April 2002), `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 (July 2011), `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`
18. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002), `http://isabelle.in.tum.de`
19. Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An Overview of the Scala Programming Language. Tech. Rep. IC/2004/64, EPFL Lausanne, Switzerland (2007)
20. OMG: Meta Object Facility (MOF) Core v. 2.0 Document (2006), `http://www.omg.org`
21. Peyton-Jones, S.: Haskell 98 language and libraries: the revised report. Cambridge University Press, Cambridge (2003), `http://www.worldcat.org/isbn/9780521826143`
22. Selic, B.: The pragmatics of model-driven development. IEEE Software 20(5), 19–25 (2003)
23. Shah, S.M.A., Anastasakis, K., Bordbar, B.: From UML to Alloy and Back Again. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 158–171. Springer, Heidelberg (2010)
24. Steele, G.L.: Common LISP, 2nd edn. Digital Press (1990)
25. Wimmer, M., Kramler, G.: Bridging Grammarware and Modelware. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 159–168. Springer, Heidelberg (2006)