

# Integrating a Formal Development for DSLs into Meta-Modeling <sup>★</sup>

Selma Djedjai<sup>1</sup>, Martin Strecker<sup>1</sup>, and Mohamed Mezghiche<sup>2</sup>

<sup>1</sup> IRIT (Institut de Recherche en Informatique de Toulouse)  
Université de Toulouse  
Toulouse, France

<sup>2</sup> LIMOSE, Université de Boumerdès  
Faculté des Sciences  
Boumerdès, Algeria

**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 allow to specify and understand software systems. Moreover, they permit to easily generate graphical interfaces or editors for Domain Specific Languages (DSLs) 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 presented with an illustrating example, along with an implementation in the Eclipse environment.

## 1 Introduction

Formal methods (such as interactive proof assistants [13, 19]) are increasingly used in software engineering 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) [3, 15] supplies us with visual specification languages as class diagrams [8] that use intuitive notations. They allow to specify, visualize, understand and document software systems. However, they suffer from 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.

---

<sup>★</sup> Research supported in part by the project *Verisync* (ANR-10-BLAN-0310)

One possible scenario is to define the abstract syntax of a Domain Specific Language (DSL) [20] to be used in the context of a formal verification, and then to generate a corresponding *Ecore* meta-model to be able to use an MDE-based tool chain for further processing. Inversely, the meta-model can then 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 client’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 [9]. 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 constitute a sufficiently complex case study of realistic size for a DSL.

This paper is structured as follows: we start in Section 2 by comparing our approach with related work. Then, we present some preliminaries, to introduce the main components of our work. Section 4 constitutes the technical core of the article; it describes a translation from data models used in verification environments, to meta models in *Ecore* and backwards. We then illustrate the methodology with an example in Section 5, before concluding with perspectives of further work.

## 2 Related Work

EMF (Eclipse Modeling Framework) [4] models are comparable to Unified Modeling Language (UML) class diagrams [8]. 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 [11] and vice-versa [10]. 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 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, 16].

These methods enable 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 [5, 7] 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 syntactic aspects and does not allow to model deeper semantic properties (such as an operational semantics of a programming language and proofs by bisimulation).

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

### 3 Preliminaries

#### 3.1 Methodology

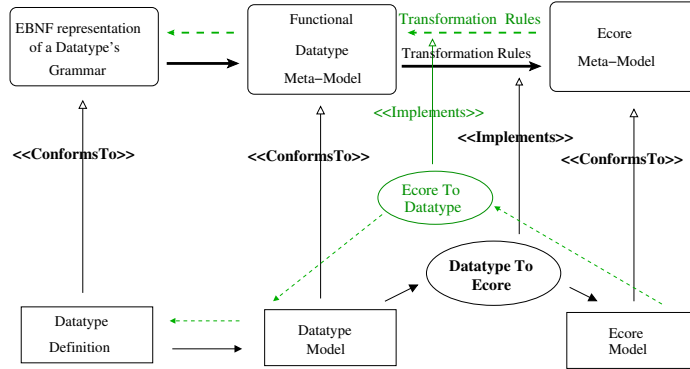
Model Driven Engineering is a software development methodology where the (meta-)models are the central elements in the development process. A meta-model defines the elements of a language. The instances of these elements are used to construct a model of the language. A model transformation is defined 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) [14] defined the Model Driven Architecture (MDA) standard [12], 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 [13]. This meta-model is the source meta-model of our transformation. We also define a subset of the Ecore meta-model [9] to be the target meta-model. In order to perform this transformation, we defined a set of transformation rules (detailed in Section 4.1) 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 rules. Bidirectionality [17] is one of 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. The transformation in the reverse direction is given in Section 4.2. The transformation rules of the two sides have been successfully implemented in an application presented with an illustrating example (see Section 5).

#### 3.2 The Datatype Meta-Model

Functional programming supplies us with a rich way to describe data structures. However, since some features cannot be supported by `Ecore`, we have only defined a subset that contains the essential element composing datatypes. Figure 2 depicts the datatype meta-model that is constructed from a subset of datatype's declarations grammar [13, 18]. We point out that we are mainly interested in data structures. It correspond to the static part of the proofs. Except for the case of accessors, the functions are not treated.



**Fig. 1.** Overview of the Transformation Method

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: a disjoint union of 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* 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*. 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 Parameters*. The type expressions then can contain a previously defined parameterized type or one of the specified parameters.

### 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 [4]. It allows to build Java applications based on model definitions and to integrate them as Eclipse plug-ins.

The Meta Object Facility (MOF) set by the OMG defines a subset of UML class diagram [8]. It represents the meta-meta-model of UML. *Ecore* is comparable to MOF but quite simpler. They are similar in their ability to specify classes, structural and behavioral features, inheritance and packages.

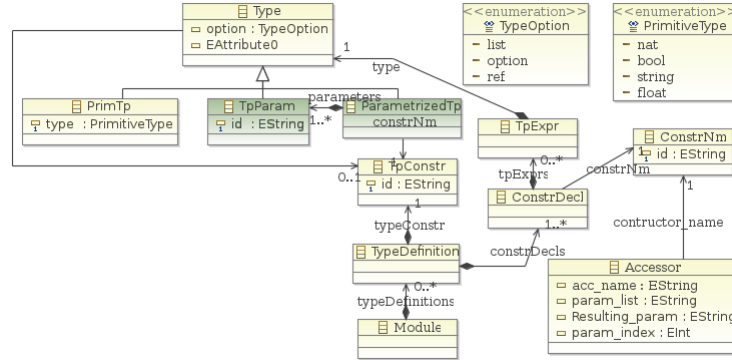


Fig. 2. Datatype Meta-model

We use in the implementation of our approach Eclipse and its core language Ecore. However, it would be possible to choose other solutions [5]. This choice is due to the place that take Eclipse for the development and metamodeling. Also, it offers a wide range of highly integrated tools.

Figure 3 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 **EClasses** and **EDataTypes**.
- The **EClass** component represents classes in Ecore. It describes structure of objects. It contains **EAttributes** and **EOperations**.
- The **EDataType** component represents the types of **EAttributes**, either predefined (types: Integer, Boolean, Float, etc.) or defined by the user. There is a special datatype to represent enumerated types **EEnum**
- **EReferences** 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 with Ecore* To support parametric polymorphism Ecore was extended. Actually, parameterized types and operations can be specified, and types 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 3 by the green color). A parameterized type is then represented by a simple **EClass** that contains one or more **ETypeParameters**. 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 **EGenericsTypes** representing the type parameters.

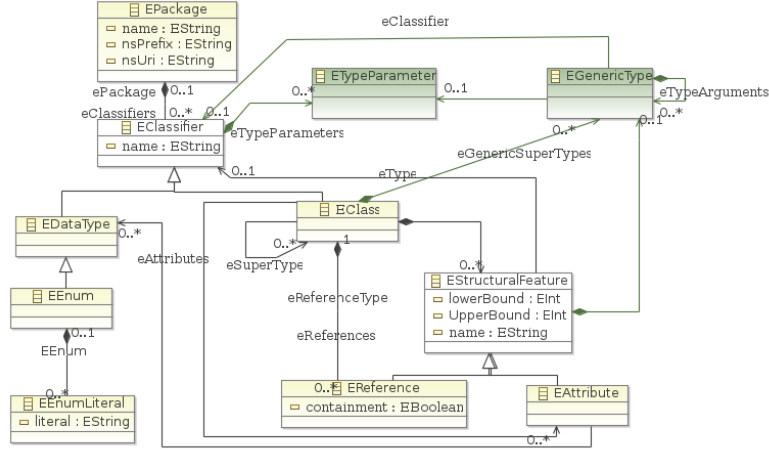


Fig. 3. Simplified subset of the Ecore Meta-model

## 4 From Datatypes to Meta-Models and Back again

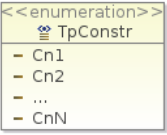
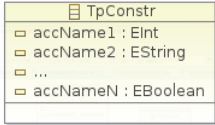
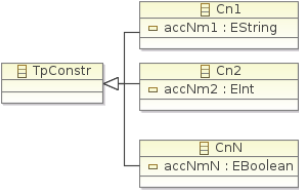
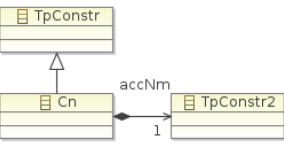
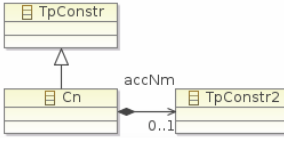
This part details the automatic translation from functional datatypes to meta-models, and backwards. The first direction of translation is further developed in Section 4.1 while the reverse direction is presented in Section 4.2. The transformation implementation is spelled out and illustrated by an example (Section 5).

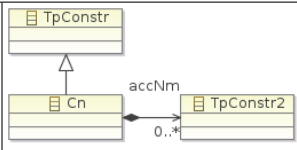
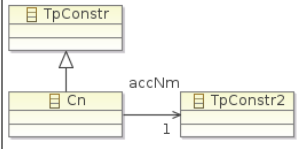
### 4.1 From Datatypes to Meta-Models

Table 1 presents the principal patterns of our recursive translation function. The translation process is detailed and described in a formal notation in [6]. In this table, we proceed with a description by example. The first column represents possible instances of data types, while the second contains the transformation rule applied for this kind of patterns. As for the last one, it shows the results of applying the rule on the instance of data type.

*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 `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 have the same `ETypeParameters` as the *Type Constructor* one.
- Setting its `eGenericSuperType` referring to the generic type representing the *Type Constructor* `EClass`.

Datatypes	translation description	Ecore Diagram Components
<pre>datatype tpConstr =   Cn1   Cn2   ...   CnN</pre>	<p>Datatypes composed only of <i>constr-names</i> (without <i>typeexprs</i>) are translated to <b>EEnums</b> which are usually used to model enumerated types in <b>Ecore</b>.</p>	 <pre>&lt;&lt;enumeration&gt;&gt; TpConstr - Cn1 - Cn2 - ... - CnN</pre>
<pre>datatype tpConstr =   Cn of nat* string * ...*   bool</pre>	<p>When the datatype is formed of only one constructor, it is translated to an <b>EClass</b>. The <b>EClass</b> name is the name of the type constructor. Primitive types give <b>EAttributes</b> in the <b>EClass</b>. The names of the attributes are given by the accessors names.</p>	 <pre>TpConstr + accName1 : EInt + accName2 : EString + ... + accNameN : EBoolean</pre>
<pre>datatype tpConstr =   Cn1 of string     Cn2 of nat     ...     CnN of bool</pre>	<p>When constructor declarations are composed of more than one constructor declaration containing type expressions, a first <b>EClass</b> is created to represent the type constructor (<i>tpConstr</i>). Then, for each constructor, an <b>EClass</b> is created too, and inherits from the <i>tpConstr</i> one.</p>	 <pre>TpConstr + Cn1 + Cn2 + CnN + accNm1 : EString + accNm2 : EInt + accNmN : EBoolean</pre>
<pre>datatype tpConstr =   Cn of tpConstr2</pre>	<p>When a type expression contains a type which is not a primitive type, the latter has to be previously defined in the Isabelle [13] theory. Then, a containment link is created between the current <b>EClass</b> and the <b>EClass</b> referring to the datatype <i>tpConstr2</i>, and the multiplicity is set to 1.</p>	 <pre>TpConstr + Cn + TpConstr2 + accNm : 1</pre>
<pre>datatype tpConstr =   Cn of tpConstr2 option</pre>	<p>The type expression <i>type option</i> is used to express whether a value is present or not. It returns <b>None</b>, if it is absent and <b>Some</b> value, if it is present. This is modeled by adopting the cardinality to 0..1.</p>	 <pre>TpConstr + Cn + TpConstr2 + accNm : 0..1</pre>

Datatypes	translation description	Ecore Diagram Components
<code>datatype tpConstr = Cn of tpConstr2 list</code>	The type expressions can also appear in the form of a <i>type list</i> . In this case the multiplicity is set to $0..*$ .	
<code>datatype tpConstr = Cn of tpConstr2 ref</code>	The last case that we deal with, is <i>type ref</i> which is used to represent pointers. It is translated to references without containments.	

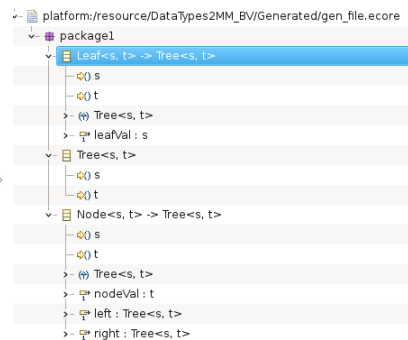
**Table 1.** Table illustrating the transformation rules from datatypes to meta-models

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 `EGenericsTypes` are not explicitly represented in the `EcoreDiagram`.

**Example:**

```
datatype ('s,'t) tree =
  Leaf 't
  | Node 's (('s,'t) tree) (('s,'t) tree)
```





## 4.2 From Meta-Models to Datatypes

To perform the reverse direction of the transformation, we draw heavily on the mapping performed on the forward translation (Section 4.1). In our view, it is important to successfully implement a function that is the inverse of the one from datatype to meta-models. Indeed, the possibility of composing the two functions, apply them on a model and find an equivalent model is paramount. Even if it leads us to set some additional restrictions on the meta-model. For example, the meta-models that contain inheritance of classes on more than one level (degree) (a class that inherits of a class that inherits from another one etc.) are not supported by our transformation rules. Table 2 summarizes the most important features taken into account in our transformation process and their translation in the functional world.

Ecore Components	Functional Data Structures
<b>EClass</b>	<i>Type Constructor + Constructor</i>
<b>EAttribute</b>	<i>Type Expression (Primitive Type)</i>
<b>EReference</b>	<i>Type Expression</i>
<b>EEnum</b>	<i>Type Constructor + Constructors (without Type Expressions)</i>
<b>EEnumLiteral</b>	<i>Constructor (without Type Expression)</i>
<b>Inheritance</b>	<i>Type Constructor + Constructors + Type Expressions</i>
<b>EGenericType</b>	<i>Parameterized Datatype</i>
<b>ETypeParameter</b>	<i>Type Parameter</i>

**Table 2.** Table summarizing the transformation rules from Ecore meta-models to datatypes

## 5 Implementation and Example

To illustrate our approach, we decided to take as example a description of a DSL. It 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]). Our approach is implemented using the Eclipse environment.

Performing the translation for the whole language description would generate a huge metamodel that couldn't be presented in the paper. We thus choose to present a only an excerpt of it, corresponding to a method definition. Figure 4 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. 4.** Datatypes in Isabelle

This part of the *Isabelle theory* was given as input to the implementation of our translation rules presented in Section 4.1. The resulting *Ecore* diagram is presented in Figure 5. 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 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 translated to respectively *EInt*, *EBoolean* and *EString*.

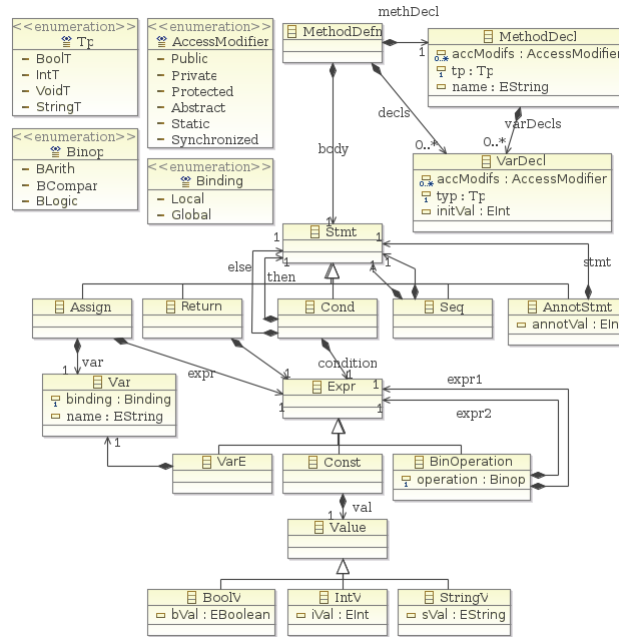


Fig. 5. Resulting Ecore Diagram after Transformation

## 6 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 transformation rules.

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. Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. Uml2alloy: A challenging model transformation. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 436–450. Springer, 2007.

2. Nadezhda Baklanova, Martin Strecker, and Louis Féraud. Resource sharing conflicts checking in multithreaded Java programs. Available on: <http://www.irit.fr/~Nadezhda.Baklanova/papers/FAC2012.pdf>, April 2012.
3. Jean Bézivin. Model driven engineering: An emerging technical space. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 36–64. Springer Berlin / Heidelberg, 2006.
4. Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.
5. Juan de Lara and Hans Vangheluwe. Using AToM<sup>3</sup> as a meta-case tool. In *ICEIS*, pages 642–649, 2002.
6. Selma Djedjai, Mohamed Mezghiche, and Martin Strecker. A case study in combining formal verification and model-driven engineering. In Vadim Ermolayev, Heinrich C. Mayr, Mykola Nikitchenko, Aleksander Spivakovsky, Grygoriy Zholtkevych, Mikhail Zavileysky, and Vitaliy Kobets, editors, *ICTERI*, volume 848 of *CEUR Workshop Proceedings*, pages 275–289. CEUR-WS.org, 2012.
7. Karsten Ehrig, Claudia Ermel, Stefan Hänsgen, and Gabriele Taentzer. Generation of visual editors as Eclipse plug-ins. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 134–143, New York, NY, USA, 2005. ACM.
8. Robert B. France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, 1998.
9. Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, Upper Saddle River, NJ, 2009.
10. Akram Idani. UML models engineering from static and dynamic aspects of formal specifications. In Terry A. Halpin, John Krogstie, Selmin Nurcan, Erik Proper, Rainer Schmidt, Prina Soffer, and Roland Ukor, editors, *BMMDS/EMMSAD*, volume 29 of *Lecture Notes in Business Information Processing*, pages 237–250. Springer, 2009.
11. Akram Idani, Jean-Louis Boulanger, and Laurent Philippe. A generic process and its tool support towards combining UML and B for safety critical systems. In Gongzhu Hu, editor, *CAINE*, pages 185–192. ISCA, 2007.
12. Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained : The Model Driven Architecture : Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
13. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer Verlag, 2002.
14. OMG. *Meta Object Facility (MOF) Core v. 2.0 Document*, 2006.
15. Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.
16. Seyyed M. A. Shah, Kyriakos Anastasakis, and Behzad Bordbar. From UML to Alloy and back again. In Sudipto Ghosh, editor, *MoDELS Workshops*, volume 6002 of *Lecture Notes in Computer Science*, pages 158–171. Springer, 2009.
17. Perdita Stevens. A landscape of bidirectional model transformations. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *GTTSE*, volume 5235 of *Lecture Notes in Computer Science*, pages 408–424. Springer, 2007.
18. <http://caml.inria.fr>. Caml programming language website.
19. <http://coq.inria.fr/>. Coq proof assistant website.
20. Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.