Automatic Transformation from Ecore Metamodels towards Gallina Inductive Types

Jérémy Buisson[1] and Seidali Rehab[2]
[1]*IRISA, Écoles de Saint-Cyr Coëtquidan, Guer, France*
[2]*MISC, University of Constantine 2, Abdelhamid Mehri, Nouvelle Ville Ali Mendjeli, Constantine, Algeria*

Keywords: Model-Driven Engineering, Model Transformation, QVT, Ecore, Xtext, Coq.

Abstract: When engineering a language (and its compiler), it is convenient to use widespread and easy-to-use MDE frameworks like Xtext that automatically generate a compiler infrastructure, and even a full-featured IDE. At the same time, a formal workbench such as a proof assistant is helpful to ensure the language specification is sound. Unfortunately, the two technical spaces hardly integrate. In this paper, we propose a transformation from Ecore's metametamodel to Coq's language named Gallina/Vernacular. The structural fragment of Ecore is fully handled. At the cost of not being bijective, our transformation has relaxed constraints over the input metamodel, in comparison to previous state of the art. To validate, we have used the proposed transformation with a complete and representative test suite, as well as a proof-carrying code type checker.

1 INTRODUCTION

In the context of designing a formal architecture description language for system of systems engineering, named SosADL (Oquendo et al., 2016), the work presented in this paper is specifically related to the implementation of supporting tools for this language.

On the one side, model-driven engineering provides effective tools like ASF+SDF (Klint, 1993), Xtext (Bettini, 2013) or MPS (Voelter, 2013) that ease the creation of a language and its supporting infrastructure. From a combined description of a concrete and/or abstract syntax, a complete editing environment is generated, which includes syntax-highlighting, auto-completion, error reporting. These tools often come with a compilation or interpretation framework, which is designed to smoothly interact with the generated editing environment.

On the other side, language theory promotes principled language design by means of well-established techniques to specify a language in terms of, *e.g.*, semantics and type system, then to study this specification, *e.g.*, proving a type soundness theorem. Existing literature hints at relevant properties, proof techniques, and mechanization approaches by means of proof assistants like Coq (Bertot and Castran, 2010) or Isabelle/HOL (Nipkow et al., 2002).

In our project, we expect to benefit from the two fields. But model-driven engineering tools like Xtext or MPS hardly integrate with proof assistants. For instance, often, the former use a graph-based formalism while the latter rely on inductive data types, despite some exceptions such as Rascal (Klint and van der Storm, 2016). Furthermore, the question arises whether the (informal) implementation, *e.g.*, in the Java technical space conforms to the (formal) specification, *e.g.*, in the Coq technical space. To address this issue, we consider the proof-carrying code approach (Necula, 1997). Figure 1 summarizes this context. From a concrete *grammar*, Xtext generates a *metamodel*, and an *editor*, a *parser* and a *compilation infrastructure*. The parser transforms the textual *source* into an object-oriented *model*, which is an instance of the metamodel. Thanks to the compilation infrastructure, we develop a *compiler* that transforms the model into *output artifacts*. Proof-carrying code appears in the lowest part of the figure: the compiler also uses a *proof generation infrastructure* to produce a *proof*, which is an instance of the language's specification, *i.e.*, of the *type system* or *semantics*. The proof contains *terms*, which are instances of *abstract syntax type*. Furthermore, terms and models map one each other, and therefore the metamodel and the abstract syntax type must be consistent.

Our long-term goal is to automatically derive (part of) the proof generation infrastructure. In this perspective, in this paper, we focus on how we can generate the abstract syntax type from the metamodel. In

Buisson, J. and Rehab, S.
Automatic Transformation from Ecore Metamodels towards Gallina Inductive Types.
DOI: 10.5220/0006608604880495
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 488-495
ISBN: 978-989-758-283-7

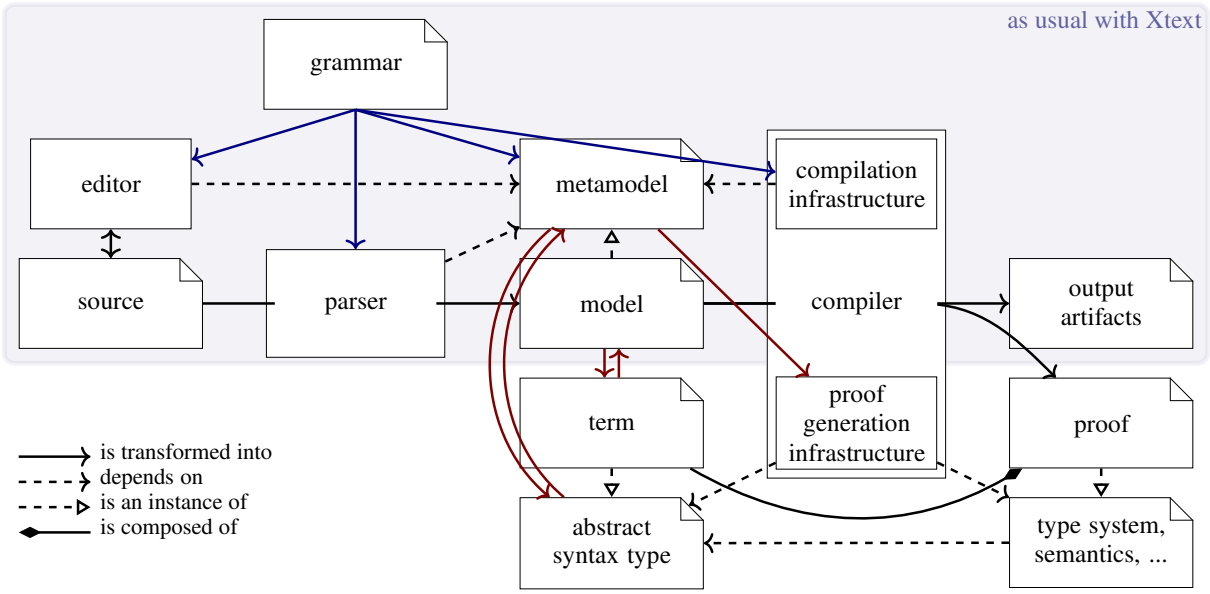


Figure 1: The big picture of our general approach.

comparison to the previous state of the art, our contribution is an improved transformation that has relaxed constraints over the input metamodel, especially with regard to inheritance, at the cost of not being bijective.

Section 2 presents related works. Section 3 describes a running example. Section 4 shows in details how the Ecore metamodel is translated into Gallina types. Section 5 discusses specific points, noticeably why we consider having a bijective transformation is not that important in our case. Section 6 gives indications about implementation issues. Section 7 summarizes how we validate the transformation. Finally, section 8 concludes the paper with perspectives.

2 RELATED WORKS

On the one side, language theory provides background to specify languages (their semantics and their type systems) as well as what properties of such specification should be investigated for in order to convince a language specification is sound. In addition to bare extraction mechanisms, several approaches in the field of language theory aims at generating artifacts from the description of a language. In the $\mathbb{K}$-framework (Roşu and Şerbănuţă, 2014), concrete syntax, operational semantics, type system, and other static analyses are specified by means of an executable semantic framework such that an interpreter and a set of tools are generated from their specification. Lem (Mulligan et al., 2014) is a domain specific functional programming language, which compiles inductive relations, e.g., an encoding of an operational semantics or of a type system, into executable functions. From the same specification, Lem is able to generate Coq and Isabelle/HOL artifacts along with LaTeX documentation. Ott (Sewell et al., 2010) aims at intuitive notations for inference rules and targets mainly proof assistant artifacts and documentation. It supports the generation of Lem code as well as boilerplate OCaml code. But none of these works addresses the user programming environment.

On the other side, model-driven engineering fosters the automatic generation of compiler infrastructure and user interface from a description of the language. Centaur (Borras et al., 1988) and ASF+SDF (Klint, 1993) aim at generating a complete programming environment, including user interfaces, given a combined abstract and concrete syntax description, along with an executable semantics. More recently, Xtext (Bettini, 2013) generates a full-featured text editor and a compilation framework from the combined definition of concrete and abstract syntax. MPS (Voelter, 2013) introduces a projectional editor, *i.e.*, edition is made directly at the level of the abstract syntax. MPS also provides a declarative language for executable type systems. But none of them allow formal specification and study of languages.

There is therefore a need for bridging the gap between model-driven engineering and formal methods.

Most of the works in this topic, *e.g.*, (Meyer and Souquières, 1999; Lano et al., 2004; Barbier and Cariou, 2012; Cabot et al., 2014) focus on a different issue: verification of properties of metamodels. Inhabi-

tation (or consistency) is for instance a widely-studied problem, which aims at verifying that a metamodel is contradiction-free, *i.e.*, that some instance exist. Because of this focus, the object manipulated in these works is the metamodel itself. Like depicted in Figure 1, we aim at being able to mechanize semantics and type systems of the language, hence properties of *instances* of the metamodel. So the metamodel has to be transformed into a type, such that terms (or instances) of that type can be manipulated.

Some previous works have studied such transformations: (Djeddai et al., 2012) have defined a bidirectional transformation between Ecore and Isabelle's inductive type. To obtain a bijection, they restrict to single-level single inheritance: each abstract class `A` is mapped to an inductive type `t`; and each concrete class `C` is transformed to a constructor `c` of the inductive type `a` mapped from the super class `A` of `C`. Rascal (Klint and van der Storm, 2016) proposes to preprocess the metamodel before this scheme is used: step 1 structural features are pushed to concrete classes (same as our step ⑥ in Figure 3); step 2 references are generalized, *i.e.*, references to any class `C` are replaced with a reference to `C`'s most general super class. Rascal's step 2 assumes existence of a most general super class for any class, but this assumption does not hold when multiple inheritance is used[1].

The transformation of Section 4 follows the same principles as those of (Djeddai et al., 2012; Klint and van der Storm, 2016), but it does not require any restriction on the input Ecore metamodel. When multiple inheritance is not used, it generates narrower types than Rascal's transformation by duplicating constructors, so generating a term from a model is going to be harder, because the right constructor has to be selected with respect to the expected type for the term. Like Rascal's one, our transformation is not bijective.

3 A RUNNING EXAMPLE

To illustrate the discussion in subsequent sections, we use the metamodel for λ terms of Figure 2. A `File` is composed of `Definition`s, each containing a `Term`. A term is either an `Abstraction`, an `Application` or a `Variable`. In order to avoid issues related to naming and scopes, the abstract syntax assumes variables have already been resolved, hence `Variable` has a non-containment reference to `Binder`, which is either an abstraction or a definition. Classes are generic such that terms can be annotated, *e.g.*, with types.

[1]Unless a predefined super class implicitly generalizes any class, for instance, like Ecore's `EObject` class. In this case, all the reference types are generalized to `EObject`.

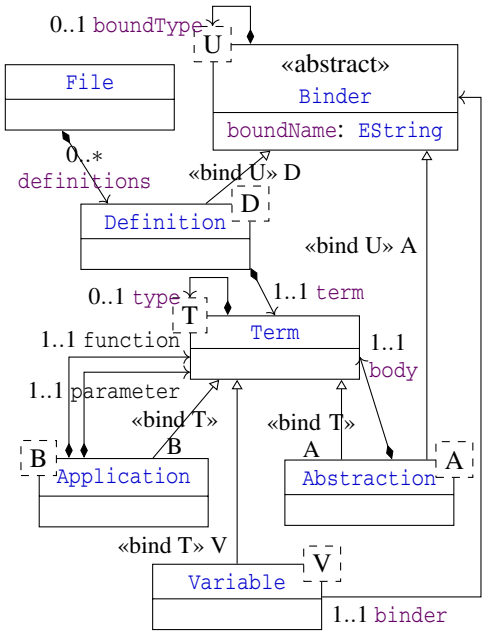


Figure 2: Ecore metamodel of the example.

We expect that our transformation generates the following Coq script (or equivalent), *i.e.*, inductive types such that any model that is an instance of the source metamodel can be written as a term whose type is one resulting from the transformation.

```
Inductive _Term: Type → Type :=
| Term_Abstraction: ∀ (A: Type) (type: option A)
    (boundName: string) (boundType: option A)
    (body: _Term A), _Term A
| Term_Application: ∀ (A: Type) (function: _Term A)
    (parameter: _Term A), _Term A
| Term_Variable:
  ∀ (A: Type) (binder: _URI (_Binder A)), _Term A
with _Binder: Type → Type :=
| Binder_Definition: ∀ (D: Type) (boundName: string)
    (boundType: option D) (term: _Term D),
    _Binder D
| Binder_Abstraction: ∀ (A: Type) (type: option A)
    (boundName: string) (boundType: option A)
    (body: _Term A), _Binder A.
Inductive _Definition: Type → Type :=
| Definition_Definition: ∀ (D: Type)
    (boundName: string) (boundType: option D)
    (term: _Term D), _Definition D.
Inductive _File: Type :=
| File_File:
  ∀ (definitions: list (_Definition _Type)), _File.
```

4 THE TRANSFORMATION

Ecore is an object-oriented language for metamodels. A metamodel consists in some *classes*, organized into

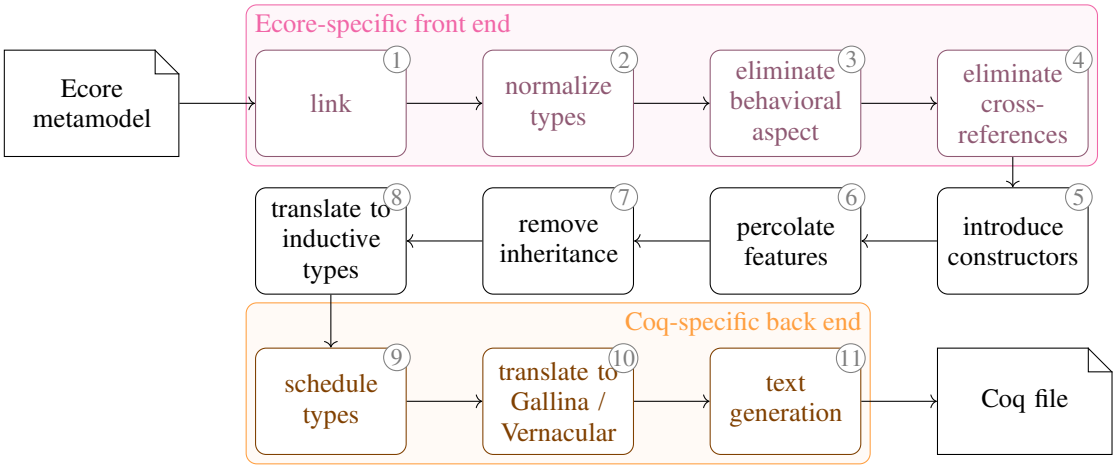


Figure 3: Decomposition of the Ecore-to-Coq transformation.

hierarchically-nested *packages*, and spread over one or several Ecore files. A specialization/generalization relationship provides inheritance and subtyping. Any class can specialize one or several general classes. Some classes can be *abstract*, meaning that they cannot be instantiated. Each class contains *structural features* (or *fields*) which are further refined as either *attributes* (store plain-Java objects only) or *references* (store Ecore objects only). References can be *containment* or non-containment references. A model, or instance of a metamodel, is therefore a tree of objects (following the containment references) with additional non-containment references. *Derived* features have their value computed on-the-fly; *transient* features are omitted from XMI serialization; *volatile* features are not stored in the in-memory instance. In this work, we do not consider *operations*.

Inductive types are mutually-recursive *types*. Each type is defined by a set of *constructors*, each of which has formal *parameters*. Each constructor defines a variant for the type, with its own data structure. A value of a given type is built by calling one of its constructors with effective parameters. A value is therefore a tree. Types are organized in hierarchically-nested *modules*, and each file is a module as well. Without lack of generality, in the following, we target Gallina and the Vernacular, Coq's languages for terms and for module-level commands.

Following the same principle as (Djeddai et al., 2012; Klint and van der Storm, 2016), our transformation maps classes to types, concrete classes to constructors, and structural features to constructor parameters. It ignores behavioral elements, *i.e.*, operations, and derived, transient or volatile features. The novelty of our transformation lies in pre-and-postprocessing, noticeably to address multiple inheritance. Like depicted in Figure 3, our transformation is

```
<LEPackage>
 <eClassifiers
    name="ecore_EString" ... />
 <eClassifiers
    name="lambda_Binder"
    eSupertypes="//ecore_EObject" ...>
  <eTypeParameters name="U" />
  <eStructuralFeatures
       name="boundName" ... />
  <eStructuralFeatures
       name="boundType" ...>
   <eGenericType eTypeParameter=
        "//lambda_Binder/U" />
  </eStructuralFeatures>
 </eClassifiers>
 ...
</LEPackage>
```

Figure 4: XMI excerpt after the *link* step.

decomposed into 11 steps. Each step targets a specific issue.

Steps ① and ② aim at first simplifying the representation of the metamodel. It would be tempting to map packages to modules. However, consider for instance an abstract class `A` in package `p1`, specialized by concrete class `C` of package `p2`. Class `C` maps to constructor C that belongs to type A, which in turn is mapped from class `A`. If packages were mapped to modules, type A should be in module `P1` and constructor C should be in module `P2`. This is impossible since constructors belong to types, not to modules. To avoid this issue, step ① removes packages and gathers all the dependencies within a single package.

Figure 4 shows an excerpt of the metamodel of Figure 2 after step ①. The metamodel is almost unchanged. Still, classes from the Ecore metamodel are pulled into the package, starting with `EObject` because it is used as the raw type of `type` in `Term`. Because of `boundName` in `Binder`, `EString` is pulled too. The pro-

cess is repeated until all the dependencies are gathered. To avoid clashes, we use a renaming scheme.

Due to backward compatibility, Ecore has two representations of types: non-generic types are direct references to classifiers; other types are instances of `EGenericType`. Figure 4 contains examples of both. To state that `lambda_Binder` specializes the (non-generic) `ecore_EObject` class, `eSuperTypes` contains the reference string `"//ecore_EObject"` (legacy representation); while the type of `boundType` is given by an instance of `EGenericType`, here the type parameter `U` of `lambda_Binder`. Step ② translates types to a simpler uniform representation.

Since inductive types do not model behaviors, step ③ erases operations as well as derived, transient and volatile features from the metamodel. Step ④ replaces cross references with attributes of a `_URI` type, which is intended to store an identifier of the referred object. Feature multiplicities are expanded to appropriate collection types at the same time. For instance, features with 0..1 multiplicity are mapped to an `_Option` type. `_URI`, `_Option` and other collection types are suitably interpreted in subsequent steps, such that they are ultimately mapped to Coq types.

Steps ⑤ to ⑦ introduce constructors within classes, before the classes can be turned into inductive types. At step ⑤, a constructor is added to map each concrete class of the metamodel, like illustrated in Figure 5. At step ⑥, features are percolated through inheritance down to the constructors, like shown in Figure 6. When the concrete class is generic, so is the constructor, like the two constructors in Figure 6. For correct handling of generic classes, type variables are substituted in the type of the structural features, like in the type of `type` and `boundType`. At step ⑦, constructors are duplicated at each level of the generalization relation, like shown in Figure 7. At the end of step ⑦, each class, either concrete or abstract, has a set of constructors that corresponds to the set of all its specializing concrete subclasses. The generalization/specialization relation can therefore be discarded. Duplicating constructors addresses the fact that a constructor belongs to exactly one type, while a class belongs to all its super classes. An assignment records in addition, for each constructor, the precise type of the built value, to correctly handle generic classes.

Step ⑧ straightforwardly turns each class into an inductive types, without any further transformation.

Coq disallows referencing an inductive type that is not previously defined or that does not belong to the same group. To satisfy this constraint, step ⑨ groups the inductive types by strongly connected components, then sorts them according to a topological order. For instance, types `Binder` and `Term` shall be in the same group, since they refer each other. This group is put before `Definition`, which refers to `Term`.

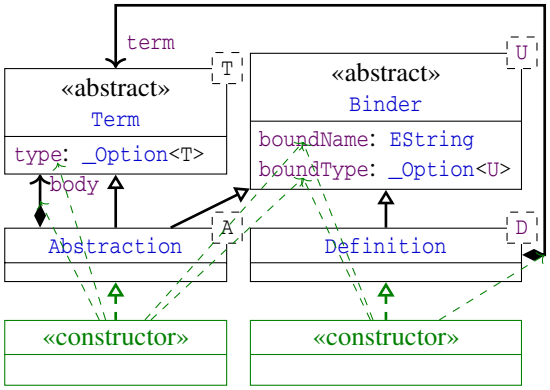


Figure 5: Metamodel excerpt after the *introduce* steps.

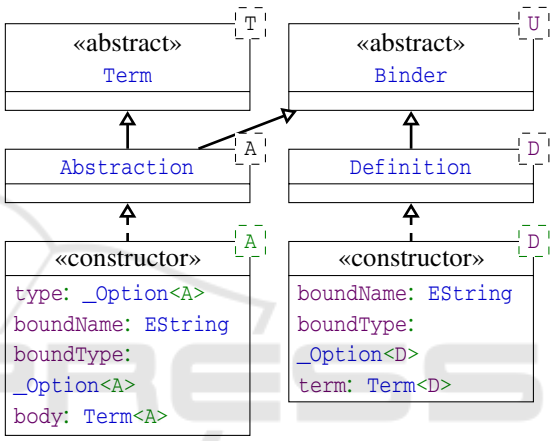


Figure 6: Metamodel excerpt after the *percolate* steps.

Step ⑩ introduces Vernacular commands (like `Inductive`) and builds Gallina terms for each type to build a correct script. Step ⑪ generates the text file. For our running example, the result is equivalent to the desired ones given at Section 3:

```
Definition ecore_EString: Type := string.
Definition ecore_EInt: Type := Z.
Definition ecore_EEList: (Type → Type) := list.
Inductive lambda_Term: (Type → Type) :=
| lambda_Term_lambda_Abstraction: (∀ (A: Type),
    (∀ (body: (lambda_Term A)),
    (∀ (boundName: ecore_EString),
    (∀ (boundType: (_Option A)),
    (∀ (type: (_Option A)), (lambda_Term A))))))
| lambda_Term_lambda_Application: (∀ (B: Type),
    (∀ (function: (lambda_Term B)),
    (∀ (parameter: (lambda_Term B)),
    (∀ (type: (_Option B)), (lambda_Term B)))))
| lambda_Term_lambda_Variable: (∀ (V: Type),
    (∀ (binder: (_URI (lambda_Binder V))),
```

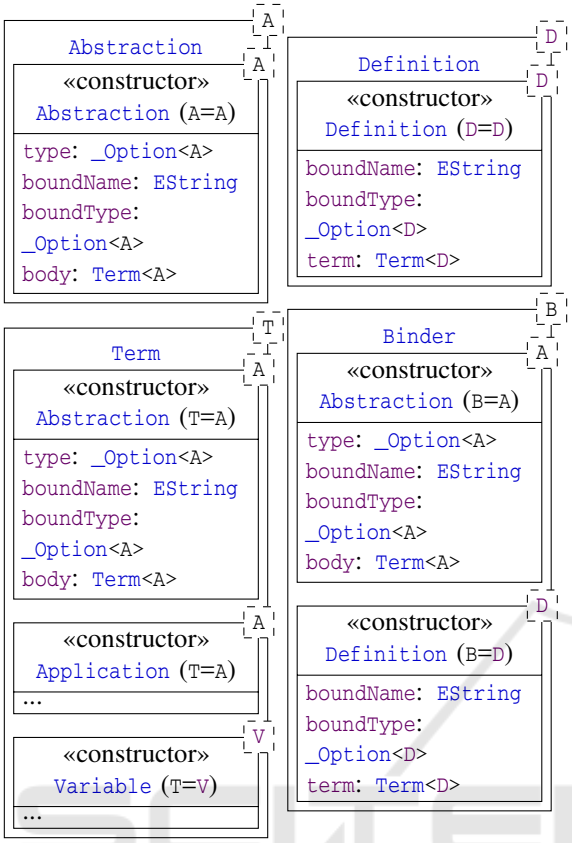


Figure 7: Metamodel excerpt after the *flatten* steps.

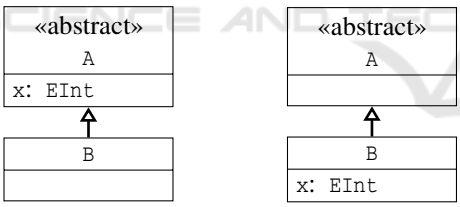


Figure 8: Different metamodels but identical Coq scripts.

```
    (∀ (type: (_Option V)), (lambda_Term V))))
with lambda_Binder: (Type → Type) :=
    (* and so on *)
```

5 DISCUSSION

Except for the limitations listed in Section 6, Ecore is fully supported. For this reason, the transformation cannot be bijective. Figure 8 shows a trivial example of two different metamodels that result in identical Coq scripts. Indeed, at step ⑥, all the structural features are pulled from classes to constructors, without tracking the class they originate from. We motivate this choice by two arguments. First, any instance of one of these metamodels is also an instance of the other one. Indeed, in both cases, models contain only instances of `B`, which contain one member `x` that is an integer. Second, our transformation is intended at allowing the language designer providing formal specifications, proving properties of these specifications, and setting up an infrastructure for proof-carrying code. As long as code and proofs written in the formal world need not be extracted back to Java and EMF, having a bijective transformation superfluously restricts the set of eligible metamodels.

Steps ①, ⑤, ⑧, ⑩ and ⑪ are bijective. No fundamental issue prevents step ④ from making this step bijective: more accurate types can easily be generated for collections, such that exact multiplicity and flags can be fully recovered. All the other steps are intrisically surjective: step ② because there is no uniqueness of Ecore's representation of types, especially for non-generic types; step ⑨ because Ecore does not take into account declaration order; step ③ because it erases behavioral elements of the metamodel; steps ⑥ and ⑦ because they discard information about inheritance.

Only the first steps ① to ④ of the transformation are specific to Ecore. We conjecture that subsequent steps can be reused to build a pipeline for another metametamodel. For instance, to switch to MOF, step ① has to deal with nested classes similarly to the way packages are merged; step ④ has to deal with MOF's richer reified associations; and data type mapping has to be updated, since MOF is not based on Java types. The back end intuitively starts at step ⑨. Switching to, say, Isabelle/HOL would require changing steps ⑩ and ⑪ in order to take into account the different abstract and concrete syntax.

6 IMPLEMENTATION

The transformation supports all the syntactical constructs of Ecore (Steinberg et al., 2009), but it assumes that the source metamodel validates correctly against all the Ecore constraints implemented in EMF, including the constraints at the *warning* level. Some of the patterns that are interpreted by Ecore's supporting tool such as the Java code generator are not recognized: hash maps are treated like lists of pairs; sets and bags are mapped to lists. Step ④ can easily be extended to specialize the generated type in these specific cases. Feature maps, that is, groups made of a $n..*$ attribute `f` and of derived volatile transient features `s` that subset `f` by key `s`, are ignored. Any feature map `f` is transformed into a list of items; and subset features `s` are eliminated at step ③. Our implemen-

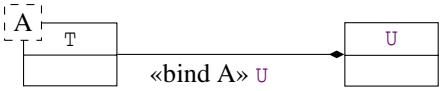
tation decodes the Java type that may appear in the `instanceTypeName` field of data type definitions, but EMF-generated classes are not converted to classifier-based types. It also handles generic types described by the *extended metadata* annotations. Restriction types are mapped to their base type. These extensions are handled in step ①.

Because of the decomposition of the transformation, we used 11 different (meta)metamodels. Even if Ecore and Gallina share almost nothing, the differences between consecutive (meta)metamodels in the pipeline are small. During development, the management of such 11 different but related (meta)metamodels shows to be challenging, despite they are rather small[2]. Following our implementation work, we observe that: interactive Ecore editors lack macro-like systems; transformation tools provide automation but lack interactivity; and XMI-level text editors (*e.g.*, Vim or Emacs) provides useful tools like macros and regexp-search-and-replace, but they are unaware of Ecore. We think that there is a need at the cross-line of these tools.

We favor industrial-strength freely-available technologies: Eclipse's mature EMF-and-Java ecosystem. Step ⑩ involves many changes in the structure of the transformed model, while its algorithm is trivial. QVT-Operational seems a good choice for this step: it avoids most of the notation burden; disjunct mappings provide an experience similar to pattern-matching; and collection operations like `iterate` reminds usual higher-order functions. Steps ① to ⑨ perform localized modifications in the transformed metamodel. At each step, most of the metamodel is unchanged except few subtrees. To implement these steps, we design an ad-hoc Java-based transformation framework that duplicates an EMF (meta)model up to class names, similarly to ATL's refining mode (without requiring the source and target (meta)metamodels be identical) and to Rascal's `visit` operation (without making data forcibly immutable). The deep copying mechanism can be customized in two ways: every time an object is copied, a hook allows customizing the class of the resulting object depending on the class of the source object; and for every feature in the target class, a hook allows customizing how the value is obtained. By default, the value of a feature having the same name in the source object is copied.

The Java code for steps ① to ⑨ contains 1300 SLOC. The QVT-Operational script for step ⑩ is made of 185 SLOC. The Acceleo template for step ⑪ contains 40 SLOC. The code is available at https://bitbucket.org/jbuisson/ecore2coq.

[2]Ecore contains 20 classes, 48 references, 33 attributes, and 33 data types. The other (meta)metamodels in the pipeline have similar size.




```
Inductive T: Type → Type := (* ... *).
Inductive T' (A: Type): Type := (* ... *).
Inductive U: Type := | U_U: ∀ (x: T U), U.
```

Figure 9: Anti-pattern leading to Coq error.

7 VALIDATION

First we ensure the transformation produces correct types, i.e., that the produced scripts compile. We fetch (meta)metamodels from four third-party open-source projects: EMF, Eclipse's OCL, Xtext extras, and Dresden OCL. Then we consider all the 151 (meta)metamodels that pass Ecore's validator. This test suite contains synthetic cases, as well as real-world (meta)metamodels such as Ecore, UML and OCL, containing up to 247 classes, each having up to 8 super types and up to 11 levels of inheritance. The suite covers all the synctactical constructs of Ecore. Of the resulting Coq scripts, 147 scripts compile correctly; Coq runs out of memory for 2 of them; it issues a "*non strictly positive occurrence*" error for 2 scripts. Examination of the erroneous scripts highlights the anti-pattern of Figure 9. Coq's error can be avoided by using the definition `T'` instead of `T`, but Coq has restrictions on this form for mutually-recursive definitions. This anti-pattern needs further investigation.

Second we ensure that the generated types are actually usable in the context of proof-carrying code (Necula, 1997), following the approach depicted in Figure 1 in the context of SosADL (Oquendo et al., 2016). Given a concrete grammar, Xtext (Bettini, 2013) generates an Ecore metamodel that contains 85 classes, and which is in turn transformed to Coq types. We use these Coq types to mechanize the type system. Then we design an ad-hoc transformation that transforms instances of the metamodel (SosADL architecture descriptions) to Gallina terms, whose types are the generated Coq types. We instrument SosADL's type checker to produce a Gallina term that witnesses the architecture description is well typed. The fact that Coq successfully compiles the generated proof term shows that the transformation produces correct and usable types.

8 CONCLUSION

In this paper, we propose a transformation from Ecore metamodels to inductive types. This transformation allows to set up a model-driven language engineering chain, *e.g.*, involving Xtext and, at the same time, to specify the language using a proof assistant, such as Coq, and then prove properties of this specification. In comparison to previous work (Djeddai et al., 2012; Klint and van der Storm, 2016), our transformation has fewer constraints on the source Ecore metamodel and ensures stronger typing in the generated inductive types, but it is not bijective.

To validate our proposal, we implement the transformation using QVT-Operational, Acceleo, and EMF-and-Java. Then we fetch 151 (meta)metamodels, which contains both synthetic and real-world (meta)metamodels coming from public repositories. 147 of the generated Coq scripts compile successfully; 2 make Coq run out of memory; the last 2 ones need further study in order to handle the corresponding specific pattern. To our knowledge, no previous work discusses nor deals with this pattern.

In future work, we will study how additional parts of such infrastructure can be automatically derived from the Ecore metamodel, such that model-driven engineering better integrates with proof assistants.

REFERENCES

Barbier, F. and Cariou, E. (2012). Inductive UML. In *Proceedings of the 2nd International Conference on Model and Data Engineering*, MEDI'12, pages 153–161, Poitiers, France. Springer.

Bertot, Y. and Castran, P. (2010). *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition.

Bettini, L. (2013). *Implementing Domain-Specific Languages with Xtext and Xtend.* Packt Publishing.

Borras, P., Clement, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., and Pascual, V. (1988). Centaur: The system. In *Proceedings of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 3, pages 14–24, Boston, Massachusetts, USA. ACM.

Cabot, J., Clarisó, R., and Riera, D. (2014). On the verification of uml/ocl class diagrams using constraint programming. *Journal of Systems and Software*, 93:1–23.

Djeddai, S., Strecker, M., and Mezghiche, M. (2012). Integrating a formal development for DSLs into metamodeling. In *Proceedings of the 2nd International Conference on Model and Data Engineering*, MEDI'12, pages 55–66, Poitiers, France. Springer.

Klint, P. (1993). A meta-environment for generating programming environments. *ACM Trans. Softw. Eng. Methodol.*, 2(2):176–201.

Klint, P. and van der Storm, T. (2016). *Model Transformation with Immutable Data*, pages 19–35. Springer International Publishing, Cham.

Lano, K., Clark, D., and Androutsopoulos, K. (2004). *UML to B: Formal Verification of Object-Oriented Models*, pages 187–206. Springer, Berlin, Heidelberg.

Meyer, E. and Souquières, J. (1999). A systematic approach to transform omt diagrams to a b specification. In *Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I - Volume I*, FM '99, pages 875–895, London, UK. Springer-Verlag.

Mulligan, D. P., Owens, S., Gray, K. E., Ridge, T., and Sewell, P. (2014). Lem: Reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 175–188, Gothenburg, Sweden. ACM.

Necula, G. C. (1997). Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, Paris, France. ACM.

Nipkow, T., Wenzel, M., and Paulson, L. C. (2002). *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg.

Oquendo, F., Buisson, J., Leroux, E., Moguérou, G., and Quilbeuf, J. (2016). The SoS Architect Studio: Toolchain for the Formal Architecture Description and Analysis of Software-intensive Systems-of-Systems with SosADL. In *Proceedings of the ECSA International Colloquium on Software-intensive Systems-of-Systems (SiSoS)*, Copenhagen, Denmark.

Roşu, G. and Şerbănuţă, T. F. (2014). K overview and simple case study. *Electronic Notes in Theoretical Computer Science*, 304:3–56. Proceedings of the Second International Workshop on the K Framework and its Applications (K 2011).

Sewell, P., Nardelli, F. z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., and StrniŠa, R. (2010). Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd edition.

Voelter, M. (2013). *Language and IDE Modularization and Composition with MPS*, pages 383–430. Springer, Berlin, Heidelberg.