Effective Bridging Between Ecore and Coq: Case of a Type-Checker with Proof-Carrying Code

Jérémy Buisson, Seidali Rehab

Effective Bridging between Ecore and Coq: Case of a Type-Checker with Proof-Carrying Code

Jérémy Buisson[1] and Seidali Rehab[2]

[1] IRISA, Écoles de Saint-Cyr Coëtquidan, Guer, France
[2] MISC, University of Constantine 2 - Abdelhamid Mehri, Constantine, Algeria

Abstract. The work presented in this paper lies in the context of implementing supporting tools for a domain-specific language named SosADL, targeted at the description and analysis of architecture for systems of systems. While the language has formal definition rooted in the Cc-pi calculus, we have adopted the Eclipse ecosystem, including EMF, Ecore and Xtext for the convenience they provide in implementation tasks. Proof-carrying code is a well-known approach to ensure such an implementation involving non-formal technologies conforms to its formal definition, by making the implementation generate proof in addition to usual output artifacts. In this paper, we therefore investigate for an infrastructure that eases the development of proof-carrying code for an Eclipse/EMF/Ecore/Xtext-based tool in relation with the Coq proof assistant. At the core of our approach, we combine an automatic transformation of a metamodel into a set of inductive types, in conjunction with a second transformation of model elements into terms. The first one, reused from our previous work, provides necessary abstract syntax definitions such that the formal definition of the language can be mechanized using Coq. The second transformation is part of the proof generator.

Keywords: Ecore · Coq · Proof-carrying code · model transformation.

1 Introduction

In our previous work [4], we have presented a transformation that maps an Ecore metamodel [13] to a collection of inductive types, more specifically targeting Gallina and Vernacular, the language of the Coq proof assistant [2]. Thanks to this previous work, we are able not only to define instances of the metamodel within the proof assistant, but also to, e.g., quantify over objects of given classes in order to prove properties or provide specifications involving this metamodel. The latter is useful, for instance, to formally mechanize the semantics or the type system of the language whose abstract syntax is given by the metamodel.

Still, we think that this transformation, alone, is not sufficient to effectively bridge between the two technical spaces. Indeed, transforming model elements might be useful as well, especially when the application relying on the metamodel has to send parts of models to the proof assistant. This is for instance the case when this application generates proofs, e.g., in the context of proof-carrying

code [11], to increase confidence in the implementation of the tools supporting the language.

This paper presents our preliminary work in this specific area. We extend our previous transformation [4] in order to generate a secondary transformation, which translates models to terms, consistently with inductive types produced accordingly to our previous work [4]. *Consistent* means here: when an EMF object is translated into a Gallina term, the object is an instance of an Ecore class (let name it `C`), and the term is of a inductive type; this inductive type is the result of the transformation of that Ecore class `C`. Then we combine the two transformations, yielding to an overall infrastructure for proof-carrying code.

In Section 2, we first present the context that motivates our work, here the development of supporting tools for SosADL [12], a domain-specific language for describing and analyzing architecture of systems of systems, and following the proof-carrying code approach. Then Section 3 gives some background on how an Ecore metamodel can be turned into a collection of inductive types, mechanization of the type system, and typical approach to the implementation of the type checker. Section 4 depicts how the type checker can be extended in order to generate proofs. Section 5 addresses the transformation of a model element into a term. Section 6 presents the related works. Last, Section 7 concludes the paper and gives our agenda for future work.

2 Context

To motivate our work, we consider the case of developing tools supporting a domain-specific language. In our case, we consider the implementation of the tools supporting SosADL [12], an architecture description language for system of systems. The language is intended to let an architect describe systems, which can be flexibly assembled into a larger system of systems by means of a constraint-based description of the assembly, based on the Cc-π formal calculus [5]. The resulting system of systems can be analyzed and simulated, e.g., in order to discover emerging behavior or to ensure the expected behavior is achieved. Analysis and simulation are enabled by the formal definition of the language.

The supporting tools for SosADL are developed using model-driven engineering, and more specifically Ecore/EMF [13] and Xtext [3] technologies from the Eclipse ecosystem. The formal definition of the language is mechanized using the Coq proof assistant [2], which enables to verify that the language definition is sound. In order to ensure that the tools conform to the formal definition, we set up a proof-carrying code infrastructure [11]. That is, in addition to performing analysis or producing output artifacts, the SosADL supporting tools issue a proof that the produced outcomes are correct. By checking the proof, the user ensures that the tools performed in conformance to the formal definition.

We have more specifically applied the proof-carrying code approach to the type checker in SosADL supporting tools. Figure 1 is an overview of it, described in subsequent sections. In the spirit of model-driven engineering, our challenge is to generate automatically (part of) the proof-carrying code infrastructure.

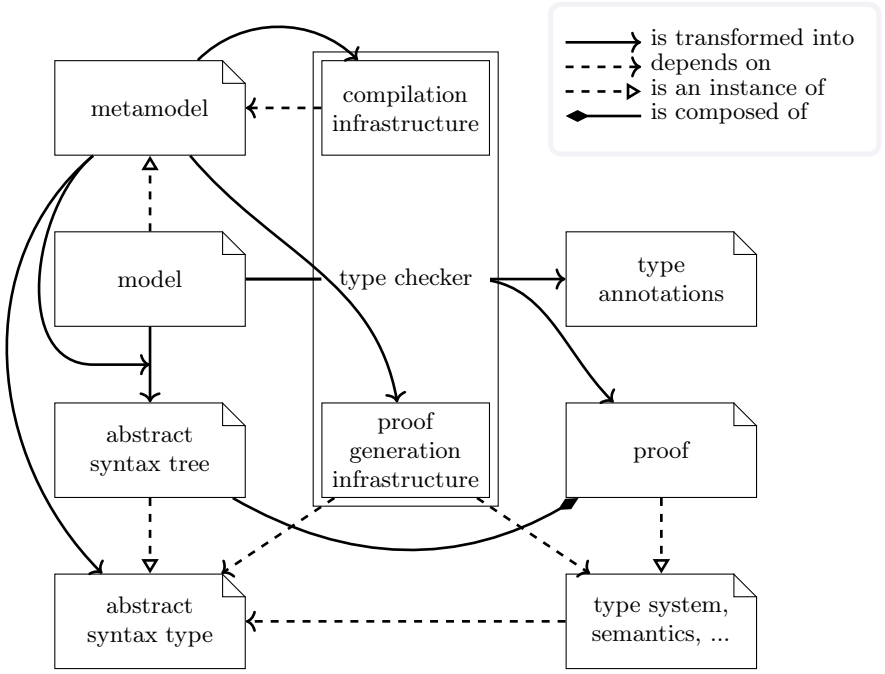

Fig. 1. Overview of applying the proof-carrying code approach to a type checker, by using model-driven engineering.

3 Background

In this section, we describe first in 3.1 how an Ecore metamodel can be turned into a collection of inductive types, following our previous work [4]. Then in 3.2, we depict a typical approach to mechanize a type system and to implement the corresponding type checker.

3.1 Transformation of a Metamodel to Inductive Types

On the one side, in the Ecore metamodel, each class defines one type of nodes of abstract syntax trees. The fields of the class describe the attributes of the nodes of that type, and the composition relationships between classes encode the parent-children relationships between the nodes in the abstract syntax tree. On the other side, an inductive type is a typical type definition in functional programming. An inductive type is made of a collection of constructors, such that any value of that type is the result of one of these constructors. Each inductive type defines one type of nodes of abstract syntax trees. The parameters of a constructor describe the data structure of the value, hence they encode both the attributes and the parent-children relationships between the nodes in the abstract syntax tree.

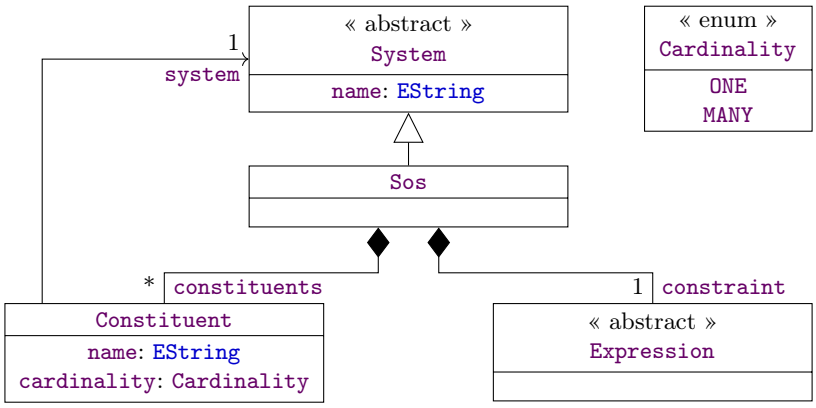

Fig. 2. An example metamodel, inspired by the SosADL case study.

Following the principle drawn by Djeddai et al. [7], each abstract class `A` is turned into an inductive type A, and any concrete class `C` that specializes `A` is transformed to a constructor C that belongs to inductive type A. In the example of Figures 2 transformed into Figure 3, abstract class `System` is mapped to inductive type sosadl_System, which declares constructor sosadl_System_sosadl_Sos, which is mapped from concrete class `Sos` that specializes `System`.

Members of concrete classes are mapped to parameters of the corresponding constructors. For example, members `cardinality`, `name` and `system` of class `Constituent` are mapped to parameters *cardinality*, *name* and *system* of constructor sosadl_Constituent_sosadl_Constituent.

Klint *et al* [8] propose preprocessing steps to support arbitrary source metamodels. The transformation first pulls class members down the specialization relationship, generalizes referenced classes then flattens the specialization relationship. Then the same principle as in [7] is applied to generate inductive types.

We have further improved the transformation in [4] in order to better support multiple inheritance. Constructors are duplicated in as many inductive types as classes in the specialization relationship. In the example, class `Sos` illustrates this approach. First, each class is mapped to an inductive type: class `System` is mapped to sosadl_System, and class `Sos` is mapped to sosadl_Sos_0. Then, concrete classes are mapped to constructors that belong to the types mapped from the super classes: class `Sos` is therefore mapped to two constructors, sosadl_System_sosadl_Sos (in type sosadl_System mapped from `System`) and sosadl_Sos_sosadl_Sos (in type sosadl_Sos mapped from `Sos`).

3.2 Mechanization of the Type System and Implementation of the Type Checker

By using inductive types generated like described in Section 3.1, we mechanize the type system of SosADL. Following the usual approach, the mechanized type

```
Inductive sosadl_Expression_0: Type := .

Inductive sosadl_Cardinality_0: Type :=
| sosadl_Cardinality_MANY: sosadl_Cardinality_0
| sosadl_Cardinality_ONE: sosadl_Cardinality_0.

Definition ecore_EString_0: Type := string.

Definition _NoDupList_0: (Type → Type) := list.

Inductive sosadl_Constituent: Type :=
| sosadl_Constituent_sosadl_Constituent: (∀ (cardinality: sosadl_Cardinality_0),
   (∀ (name: ecore_EString_0), (∀ (system: (_URI_0 sosadl_System)),
     sosadl_Constituent)))
 with sosadl_System: Type :=
| sosadl_System_sosadl_Sos: (∀ (constituents: (_NoDupList_0 sosadl_Constituent)),
   (∀ (constraint: sosadl_Expression_0), (∀ (name_0: ecore_EString_0),
     sosadl_System))).

Inductive sosadl_Sos_0: Type :=
| sosadl_Sos_sosadl_Sos: (∀ (constituents_0: (_NoDupList_0 sosadl_Constituent)),
   (∀ (constraint_0: sosadl_Expression_0), (∀ (name_1: ecore_EString_0),
     sosadl_Sos_0))).
```

Fig. 3. Output of the transformation [4] for the metamodel of Figure 2.

```
Inductive system_is_well_typed: environment → sosadl_System → Prop :=
| sos_is_well_typed: ∀ (Γ: environment)
       (constituents: _NoDupList_0 sosadl_Constituent)
       (constraint: sosadl_Expression_0) (name: ecore_EString_0),
     ∀ constituents_exist: (∀ c, c ∈ constituents → constituent_exists Γ c),
     ∀ constraint_is_well_typed: expression_has_type Γ constraint type_boolean,
  system_is_well_typed Γ (sosadl_System_sosadl_Sos constituents constraint name).
```

Fig. 4. Example of a mechanized rule.

system is itself a collection of inductive types, where each inductive type defines a judgment and its constructors are the axioms that encode rules for that judgment. Figure 4 illustrates the approach. Judgment system_is_well_typed asserts that a system declaration conforms to the type system. The inductive type encoding this judgment accepts two parameters: an environment and the system under consideration. In the figure, we provide only one rule for this judgment. This rule is encoded by constructor sos_is_well_typed. The first four parameters of the constructor (Γ, *constituents*, *constraint* and *name*) are the variables that have to be bound in order to apply the rule. Additional parameters (*constituents_exist* and *constraint_is_well_typed*) are the two premises of the rule. The return type of the constructor, where the inductive type has effective parameters, is the conclusion of the rule.

```
public boolean proveSystemIsWellTyped(Environment g,
      System s) throws Unprovable {
  if(s instanceof Sos
    && ((Sos)s).constituents.stream()
         .allMatch(c -> proveConstituentExists(g, c))
    && proveExpressionHasType(g,
         ((Sos)s).constraint, BOOLEAN)){
    // proved by rule sos_is_well_typed
    return true;
  } else {
    throw new NoMatchingRule();
  }
}
```

Fig. 5. Typical code pattern for the type checker.

When the type system is syntax directed, a typical approach to implement a type checker is to follow the principle of Milner's algorithm $\mathcal{W}$. Like illustrated in Figure 5, for each judgment, a function is implemented such that it attempts to prove a goal by selecting the adequate typing rule according to the content of the abstract syntax tree, then recursively calling itself (and other functions mapped from other judgments) in order to try to prove the premises of the chosen typing rule. In Figure 5, Java function `proveSystemIsWellTyped` is the function that aims at proving system_is_well_typed judgments. Depending on the syntactical type of the node `s` of the abstract syntax tree, it selects the rule it attempts. In the example, it attempts to prove the judgment by using rule sos_is_well_typed when `s` is an instance of `Sos`. If premises can in addition be proved, here by successfully calling `proveConstituentExists` and `proveExpressionHasType`, the function concludes that the judgment is successfully proved. Otherwise, if no rule applies, the function reports typing error, e.g., by throwing an exception. In addition to answering whether the source model is correctly typed or not, the type checker may annotate the source model with type information.

4 Generation of Proofs

By instrumenting the type checker, we extend it to generate a well-typed proof as well. Like illustrated by the example of Figure 6, each function that proves a judgment is changed such that it returns a proof object, that is, an instance of a class that corresponds to the constructor encoding the rule in the mechanized type system. In the given example, class `SosIsWellTyped` is the concrete class that corresponds to rule sos_is_well_typed; it specializes abstract class `SystemIsWellTyped` that corresponds to judgment system_is_well_typed.

We have not worked yet on how these classes could be generated, but we think that Coq's extraction mechanism may be used to address this issue.

```
public SystemIsWellTyped proveSystemIsWellTyped(
      Environment g, System s) throws Unprovable {
  if(s instanceof Sos) {
    return new SosIsWellTyped(g, ((Sos)s).constituents,
      ((Sos)s).constraint, ((Sos)s).name,
      proveForAll(((Sos)s).constituents,
          c -> proveConstituentExists(g, c),
      proveExpressionHasType(g,
          ((Sos)s).constraint, BOOLEAN));
    // proved by rule sos_is_well_typed, the proof object is returned
  } else {
    throw new NoMatchingRule();
  }
}
```

Fig. 6. Code pattern for the instrumented proof-generating type checker.

```
Definition proof: system_is_well_typed [("foo", foo); ("bar", bar)]
  (sosadl_System_sosadl_Sos
    [ sosadl_Constituent_sosadl_Constituent sosadl_Cardinality_ONE "a" ref_foo;
      sosadl_Constituent_sosadl_Constituent sosadl_Cardinality_MANY "b" ref_bar ]
    c "world") :=
  sos_is_well_typed [("foo", foo); ("bar", bar)]
    [ sosadl_Constituent_sosadl_Constituent sosadl_Cardinality_ONE "a" ref_foo;
      sosadl_Constituent_sosadl_Constituent sosadl_Cardinality_MANY "b" ref_bar ]
    c "world" P1 P2.
```

Fig. 7. Proof term, after generation of the Vernacular definition.

The proof object is then serialized into a Vernacular definition like the example of Figure 7, such that Coq's compiler can check whether the proof is correct. In this excerpt, proof is defined to be a proof of system_is_well_typed with the given parameters (environment and system definition). Its value is the proof term, here built by applying suitable parameters to constructor sos_is_well_typed. Section 5.2 explains the principles behind the generation of this term.

5 Transformation of Model Elements into Terms

Like seen in the previous section, the generated proof contains terms that encode some elements from the model being type checked. In this section, we first extend in 5.1 the transformation of Section 3.1 with correspondence information. Correspondence information is used in 5.2 in order to first present a generic algorithm that transforms any model element into a term, whose type is the inductive type mapped from the class of the model element (mapped from by the transformation of Section 3.1). Then, to avoid runtime introspection of correspondence

```
// x: model element that is going to be transformed to a term
// c: class that denotes the type of the generated term
// trace: correspondence information between the metamodel and inductive types
element_to_term(x, c, trace) {
  inductive ← trace.inductives[c]
  ctor ← filter(trace.constructors[x.eClass],
       f ↦ f.inductive = inductive)
 return apply(ctor, map(ctor.parameters,
       p ↦ feature_to_term(t.features[p], x, trace)))
}

// x: value that is going to be transformed to a term
// t: data type of the value
attribute_to_term(x, t) { // ad-hoc code that deals with primitive types
   if (EINTEGER == t) { // EInteger may be mapped to nat
    return x.toString
  } else if (ESTRING == t) { // EString may be mapped to string;
    return '"' + x + '"'
  } else ... // and so on
}

// f: structural feature (either attribute or reference) to be transformed to a term
// x: model element to which the structural feature belongs
// trace: correspondence information between the metamodel and inductive types
feature_to_term(f, x, trace) {
   if (f.isMany) { // generate a list if the structural feature is a collection one
    return list(map(x.eGet(f), o ↦ value_to_term(o, f, x, trace)))
  } else {
    return value_to_term(x.eGet(f), f, x, trace)
  }
}

// o: value to be transformed to a term
// f: structural feature from which o comes from
// x: model element to which the structural feature belons
// trace: correspondence information between the metamodel and inductive types
value_to_term(o, f, x, trace) {
   if (EREFERENCE.isSuperTypeOf(f) ∧ f.isContainment) {
    return element_to_term(x.eGet(f), f.eType, trace)
  } else if (EREFERENCE.isSuperTypeOf(f) ∧ ¬f.isContainment) {
    return uri(x.eGet(f)) // non-containment references are mapped to URIs
  } else if (EDATA_TYPE.isSuperTypeOf(f)) {
    return attribute_to_term(x.eGet(f), f.eType)
  }
}
```

Fig. 8. Generic algorithm that transfoms any model element into a Gallina term.

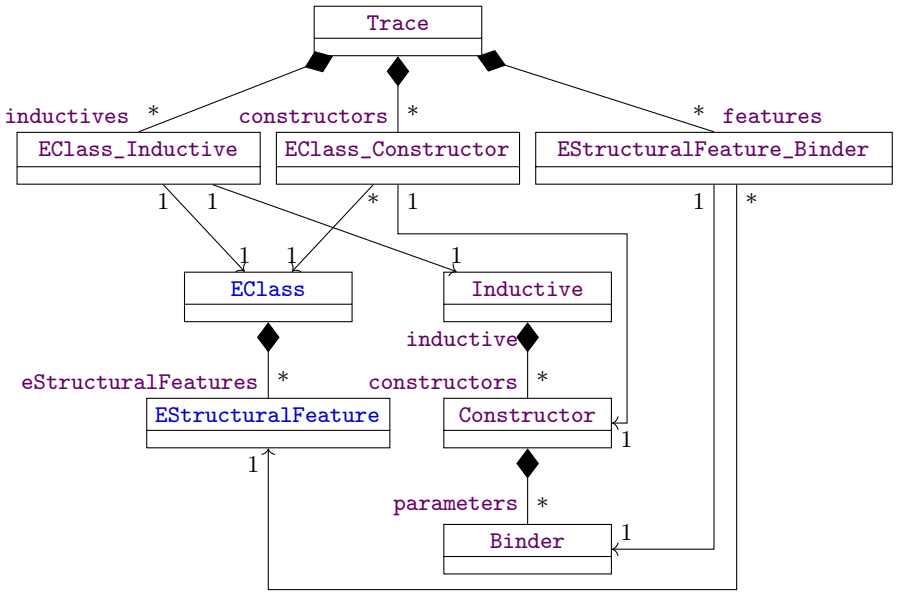

Fig. 9. Structure of correspondence information between a source Ecore metamodel and generated inductive types.

information, we present in 5.3 how we can generate an ad hoc transformation, which transforms any element whose class belongs to the given metamodel into a term of the inductive type mapped the element's class.

5.1 Correspondence Information

In addition to the inductive types generated like described in Section 3.1, correspondence information that maps between Ecore classes of the source metamodel and generated inductive types have to be produced. This information is going to be used like described in Section 5.2 in order to generate Gallina terms for model elements.

Figure 9 presents the structure of correspondence information. A `Trace` contains three reified associations. The first one named `inductives` contains one-to-one mapping between classes and inductive types. The second one named `constructors` contains many-to-one mapping between concrete classes and constructors. The last one named `features` contains many-to-one correspondence information between structural features and constructor parameters.

In this class diagram, `EClass` and `EStructuralFeature` are imported from the Ecore metametamodel, and `Inductive`, `Constructor` and `Binder` are imported from our own metamodel for Gallina and Vernacular.

5.2 Generic Transformation

The Vernacular definition of Figure 7 is composed of two Gallina terms that have to be generated: the value (after the := symbol – here sos_is_well_typed ... *P2*) and the type (between : and := – here system_is_well_typed ... "world").

To begin with, we consider the value bound by the definition. It is the translation of the objects lying in the Ecore technological space. Figure 8 provides the pseudo-code of a generic algorithm that transforms a model element into such a term. It proceeds by introspecting the type of object by means of EMF reflection facilities. For each object `x` (function `element_to_term`) the constructor corresponding to its class is applied in order to build the subterm. The constructor is found thanks to the `trace` parameter, which is the object that records correspondence information described in Section 5.1 and obtained after the transformation of the metamodel into inductive types. Because the transformation may generate several constructors for each concrete class of the metamodel, function `element_to_term` has to select the right one. The suitable constructor depends on the inductive type the generated term is expected to have. This is the reason motivating the `c` parameter of `element_to_term`: this parameter `c` is the static type of the reference from which `x` has been got. Like shown in function `feature_to_term`, the effective parameter for `c` is indeed the static type of the reference in the parent object.

Once the constructor has been found, a function application term is generated by calling function apply. To generate effective parameters, each formal parameter of the constructor is first mapped back to the Ecore structural feature it comes from. Then function `feature_to_term` is called to generate the term for the effective parameter. This function deals with collections by issuing a list if necessary, and it calls `value_to_term` to convert individual objects. This latter function discriminates between containment references, non-containment references and attributes. The first ones, i.e., containment references, are converted by a recursive call to `element_to_term`. Non-containment references are translated into an URI, i.e., an identifier of the referenced object. Attributes are plain-old Java objects, which are translated to corresponding Coq terms by hard-coded rules.

The second term, the type of the generated definition, is the translation of the class of the object translated by the algorithm of Figure 8. Because of how the type system is mechanized, the type has parameters that are values, like in the example of Figure 7. However, none of Java nor EMF supports using an object as a type parameter in static types, and both Java and EMF erase type parameters from dynamic types. Therefore, we rely instead on type inference in Coq's compiler in order to suitably generate the type of the definition. In SosADL supporting tools, we have not faced any case when type inference fails.

5.3 Generation of the Transformation

The algorithm presented in the previous subsection is a generic one that uses Java and EMF reflection at runtime, in conjunction with correspondence infor-

```
sosadl_Constituent(Constituent c) {
 return apply(sosadl_Constituent_sosadl_Constituent,
    sosadl_Cardinality(c.cardinality), string(c.name),
    uri(c.system));
}

sosadl_System(System s) {
   if (SOS.isSuperTypeOf(s.eClass)) {
    return apply(sosadl_System_sosadl_Sos,
       map(s.constituents, c ↦ sosadl_Constituent(c)),
       sosadl_Expression(s.constraint), string(s.name))
  } else {
     raise error
  }
}

sosadl_Sos(Sos s) {
 return apply(sosadl_Sos_sosadl_Sos,
    sosadl_Constituent(s.constituents),
    sosadl_Expression(s.constraint), string(s.name))
}

// and so on
```

Fig. 10. Generated algorithm that transfoms a SosADL model element into a term.

mation issued at the same time as inductive types generated by the transformation of Section 3.1. Instead of interpreting these data structures at runtime, a model element transformation can be statically generated specifically for the metamodel.

Figure 10 illustrates the generated code. For each inductive type, that is, for each class that may be used as a static type in the EMF technological space, a function is generated. When the class is concrete and when it has no known specializing class, the corresponding inductive type owns a single constructor. The generated function, e.g., `sosadl_Constituent` and `sosadl_Sos` in Figure 10, applies that constructor to effective parameters got by (possibly recursively calling) other generated functions. When the class is abstract or when it has specializing classes, the generated function, e.g., `sosadl_System` in Figure 10, uses Java or EMF reflection to find out the concrete class of the object and select the constructor accordingly.

Figure 11 outlines an algorithm to automatically generate such functions from correspondence information depicted at Section 5.1. To generate the ad-hoc transformation, function `generate_transformation` generates a function for each class (or inductive type) of the metamodel for which the transformation is generated. This is done by invoking `generate_function` on each class. Like its name tells, this function generates one generator function, for one inductive type.

```
// trace: correspondence information from which the transformation is derived
generate_transformation(trace) {
 return map(trace.inductives,
     (c,i) ↦ generate_fun(trace, c, i))
}

// class: Ecore class (abstract or concrete) that corresponds to the inductive type
// inductive: inductive type generated that corresponds to the class
generate_fun(trace, class, inductive) {
   if (class.isAbstract ∨ ∃ c, class.isSuperTypeOf(c)) {
     cases ← map( filter (trace.constructors,
         (cl, ctor) ↦ ctor.inductive == inductive),
       (cl, ctor) ↦ generate_case(trace, cl, ctor, «x»))
    return function(inductive.name, [(class, «x»)], cases)
  } else {
     ctor ← filter (trace.constructors[class],
       c ↦ c.inductive == inductive)
    return function(inductive.name, [(class, «x»)],
       generate_generate(trace, ctor, «x»)
  }
}

// class: Ecore concrete class of the object that corresponds to the constructor
// constructor: constructor mapped from the class
// object: name of the parameter in the generated function
generate_case(trace, class, constructor, object) {
 return « if (» object «instanceof» class.name «) {»
     generate_generate(trace, constructor, object) «}»
}

// constructor: constructor that is going to be issued to transform the object
generate_generate(trace, constructor, object) {
 return «return apply(» constructor.name «(»
     map(constructors.parameters,
       p ↦ generate_call(trace, p, object)) «)»
}

// parameter: binder (in Gallina) that declares the parameter of the constructor
generate_call(trace, parameter, object) {
  feature ← trace.features[parameter]
   if (¬feature.isMany) {
     inductive ← trace.inductives[feature.eType]
    return inductive.name «(» object «.» feature.name «)»
  } else // deal with lists, and so on
}
```

Fig. 11. Automatic generation of the ad-hoc transformation of Figure 10.

Regarding the body of the generated function, `generate_function` first checks whether the class under consideration is abstract or has any specializing class. If so, it generates tests for each specializing class (by calling `generate_case`). If not, it directly invokes `generate_generate`, which generates instructions to issue a call to the constructor. Function `generate_generate` uses `generate_call` for each formal parameter of the constructor in order to generate function calls that issue terms for effective parameters of the constructor.

In this paper, we omit details to deal with collections and attributes.

6 Related Works

To the best of our knowledge, no previous work has studied automatic generation of an infrastructure for proof-carrying code for a language whose abstract syntax is described by a metamodel. Though, the approach is appealing since several popular language workbenches such as the Eclipse-EMF-Ecore-Xtext combination or MPS hardly integrate formal tools that may help in the verification of language definitions and implementations. In this regard, even if our work is still preliminary, it provides a novel step towards bridging semi-formal metamodels and formal approaches.

Like stated in Section 3.1, our work is based on and improves previous work on transforming a metamodel into a group of inductive types. In comparison to [7], our improved transformation does not suffer from any restriction on the source metamodel. In comparison to [8], we further improve support for multiple inheritance as we need not assume existence of a unique most-general super class for any class. As a consequence, generated inductive types are stricter. In comparison to our own previous work [4], improvements cover the handling of correspondence information and of model elements. In addition of transforming the metamodel, in this paper, we propose two approaches to consistently transform instances of that metamodel, i.e., model elements into terms: a generic algorithm that introspects the metamodel at runtime, and a algorithm that automatically generates ad-hoc code, hence avoiding the need for runtime introspection. In this paper, we also propose the combination of the transformations in order to build an infrastructure for proof-carrying code in the context of Eclipse and related DSL technologies (EMF-Ecore-Xtext).

Several previous work such as [10, 9, 1, 6] have proposed approaches to transform OMT or UML class diagrams into terms or values in various formal calculus, hence enabling formal verification of these class diagrams. If we consider the abstract syntax for OMT or UML class diagrams (or even the Ecore metametamodel) as the metamodel, we think that our work may be able to generate automatically one such transformation, instead of hard-coding the transformation. Additional work is required in order to better evaluate how our own work might be usable in such a context.

7 Conclusion

In this paper, we have proposed to automatically generate an infrastructure for proof-carrying code given a metamodel produced in the context of the Eclipse-EMF-Ecore-Xtext ecosystem [13, 3] language workbench. The work presented in this paper is motivated by our effort on providing supporting tools for the SosADL domain-specific language [12]. Our proposal is the combination of transforming the Xtext-generated metamodel into a collection of inductive types suitable for the Coq proof assistant. Then, from the same metamodel, we automatically derive a transformation that, consistently with the generated inductive types, transforms any model element into a term that can be successfully compiled by Coq.

Even if the infrastructure has been fully implemented in the type checker of SosADL supporting tools, we think that the area needs further investigation. This work allows us to define our agenda for future work in the area.

First, we plan to further study the transformation of model elements into terms. In addition to using this transformation in the context of our proof-carrying code infrastructure, we plan to assess how this automatic transformation could be used to verify properties of some models like done with various hard-coded UML-to-B transformations proposed in previous work.

Second, we have left open the question of defining classes that implement in the Java or Ecore the inductive types encoding the mechanized type system. These classes are indeed required in order to instrument the type checker such that it produces proofs. Existing Coq's extraction mechanism translates Gallina and Vernacular definitions into other languages. While this mechanism is a basis, it is designed to skip any proof-related item from the translation, which are precisely the items we want to translate to Java or Ecore. Changing the mechanism in this regard would need to study how it must be adapted in order to conform to restrictions and constraints imposed by Java and Ecore type systems.

Last, one may ask how much confidence can be put in our proposed approach. To address this issue, we consider applying our proof-carrying code infrastructure to itself. Namely, we consider instrumenting the transformations involved in our approach in order to generate conformance proofs that could be checked by the Coq proof assistant.

References

1. Barbier, F., Cariou, E.: Inductive UML. In: Abelló, A., Bellatreche, L., Benatallah, B. (eds.) Model and Data Engineering - 2nd International Conference, MEDI 2012, Poitiers, France, October 3-5, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7602, pp. 153–161. Springer (2012). https://doi.org/10.1007/978-3-642-33609-6, https://doi.org/10.1007/978-3-642-33609-6_15
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004). https://doi.org/10.1007/978-3-662-07964-5, https://doi.org/10.1007/978-3-662-07964-5

3. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing (2013)
4. Buisson, J., Rehab, S.: Automatic transformation from ecore metamodels towards gallina inductive types. In: Hammoudi, S., Pires, L.F., Selic, B. (eds.) Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22-24, 2018. pp. 488–495. SciTePress (2018), https://doi.org/10.5220/0006608604880495
5. Buscemi, M.G., Montanari, U.: Cc-pi: A constraint language for service negotiation and composition. In: Wirsing, M., Hölzl, M.M. (eds.) Rigorous Software Engineering for Service-Oriented Systems - Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing, Lecture Notes in Computer Science, vol. 6582, pp. 262–281. Springer (2011), https://doi.org/10.1007/978-3-642-20401-2_12
6. Cabot, J., Clarisó, R., Riera, D.: On the verification of UML/OCL class diagrams using constraint programming. Journal of Systems and Software **93**, 1–23 (2014), https://doi.org/10.1016/j.jss.2014.03.023
7. Djeddai, S., Strecker, M., Mezghiche, M.: Integrating a formal development for dsls into meta-modeling. J. Data Semantics **3**(3), 143–155 (2014), https://doi.org/10.1007/s13740-013-0030-4
8. Klint, P., van der Storm, T.: Model transformation with immutable data. In: Gorp, P.V., Engels, G. (eds.) Theory and Practice of Model Transformations - 9th International Conference, ICMT 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-5, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9765, pp. 19–35. Springer (2016), https://doi.org/10.1007/978-3-319-42064-6_2
9. Lano, K., Clark, D., Androutsopoulos, K.: UML to B: formal verification of object-oriented models. In: Boiten, E.A., Derrick, J., Smith, G. (eds.) Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, April 4-7, 2004, Proceedings. Lecture Notes in Computer Science, vol. 2999, pp. 187–206. Springer (2004). https://doi.org/10.1007/b96106, https://doi.org/10.1007/978-3-540-24756-2_11
10. Meyer, E., Souquières, J.: A systematic approach to transform OMT diagrams to a B specification. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, 1999, Proceedings, Volume I. Lecture Notes in Computer Science, vol. 1708, pp. 875–895. Springer (1999). https://doi.org/10.1007/3-540-48119-2, https://doi.org/10.1007/3-540-48119-2_48
11. Necula, G.C.: Proof-carrying code. In: Lee, P., Henglein, F., Jones, N.D. (eds.) Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997. pp. 106–119. ACM Press (1997), http://doi.acm.org/10.1145/263699.263712
12. Oquendo, F., Buisson, J., Leroux, E., Moguérou, G.: A formal approach for architecting software-intensive systems-of-systems with guarantees. In: 13th Annual Conference on System of Systems Engineering, SoSE 2018, Paris, France, June 19-22, 2018. pp. 14–21. IEEE (2018), https://doi.org/10.1109/SYSOSE.2018.8428726
13. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley Professional, 2nd edn. (2009)