# Towards the verification of model transformations

Jean-Paul Bodeveix, David Chemouil, M Filali, Nathalie Lalevee, Martin Strecker

# Towards the verification of model transformations

J.-P. Bodeveix[1], D. Chemouil[2], M. Filali[1], N. Lalevée[1], M. Strecker[1],
1. FéRIA {bodeveix,filali,lalevee,strecker}@irit.fr
2. CNES

November 21, 2005

## Abstract

While verifying the correctness of model transformations is becoming increasingly important, current model transformation languages offer only weak support for checking statically that transformations cannot go wrong. This paper presents conditions under which transformations can be shown to produce models that conform to their meta-models, and it describes methods to derive prove obligations from a set of transformation rules.

## 1 Introduction

The recent shift from programming languages to modelling languages, such as UML (general purpose) or AADL [6] and Cotre [2] (domain-specific) makes it necessary to express transformations on models. These transformations can either be used to map models between different formalisms ("meta-models") or to simplify, refine or otherwise restructure a model within the same formalism.

Currently, a number of model transformation languages exist, following the QVT request for proposal. Most of them combine declarative rules, some iterative constructs and an execution engine which applies the rules in a specific manner.

For safety critical applications in particular, such as the avionics or automotive sector, correctness of transformations should be of paramount importance. Unfortunately, this problem has so far received little attention. Hardly any existing modelling language ensures that execution of a given rule set yields a model that conforms to the intended meta-model. Furthermore, there is no means to state or to verify domain-specific model properties. The work presented here is among the first to systematically address this issue.

## 2 Transformations with general purpose verification formalisms

A model transformation is no more than a function mapping an instance of a source meta-model to an instance of a target meta-model. Thus a functional language is a natural candidate for expressing model transformations. We have studied the Isabelle proof assistant for verification. Moreover, if source and target meta-models are identical, a model transformation can lead to an in place modification of a given model. In such a case, the use of an imperative framework can be considered: we have studied the B method.

In the following, we briefly sketch the two approaches without going into details. They have been discussed more extensively in [3], where we show how to specify a part of the AADL meta-model managing data flows and to define a transformation adding a filter to a flow. The encoding of the meta-model has been automated from its Ecore [4] specification.

### 2.1 Isabelle

Isabelle [10] is a proof assistant based on higher order logic. Its use in the context of model transformation consists in describing source and target meta-models as types. Then a transformation is a function from the source type to the target type. Type checking guarantees that the transformed model conforms to the target meta-model, which is an important issue of model transformation. Such a verification could be performed by any typed functionnal language such as Caml. Isabelle also allows the specification of properties over transformations, what is suggested by the QVT [8] request for proposal. Then computer aided proofs can be performed in order to verify that the

functional implementation of the transformation satisfies its requirements.

### 2.2 B

B [1] is a development method based on the refinement of abstract machines. A machine has an invariant which must be true at initialisation and preserved by each operation. The more concrete machine is written in a sublangage which can be directly translated into a programming language (C, Ada). Proof obligations for the chain of refinement guarantee, when verified, that the implementation conforms to the specification (i.e. the abstract machine).

Within this framework, the meta-model is defined as a collection of sets and constant functions modeling classes, attributes and roles. The model is declared using state variables constrained by an invariant to satisfy meta-model requirements. The transformation is encoded as an operation of the abstract machine. The fact that the transformation produces an instance of the target meta-model is guaranted by proof obligations expressing the preservation of the invariant.

## 3 ATL as a langage dedicated to model transformations

This section presents another approach to model translation exploiting a domain specific language (DSL) which is a candidate for the QVT request for proposal. ATL is a rule language designed to match elements of an instance of the source meta-model and to produce elements of an instance of the target meta-model. We will first sketch some typical problems of those transformation languages. We then describe how we can characterize the transformation process in abstract, i.e. largely implementation-independent terms. This characerization can serve as a basis for verification, as sketched in Section 2.

### 3.1 Problem description

The core of the problem of a language like ATL is that rules are not checked for "type correctness" in the same way programs in strongly typed programming languages (such as Java) are type checked in order to ensure consistency of the data manipulated by the program.

Take the example `Book2Publication`, which is taken from the ATL distribution. The following rule is supposed to convert elements of class `Book` to elements of class `Publication`, where a publication is characterized by title and author (both of them strings) and number of pages (an integer).

The rule causes no problems, as it correctly assigns strings of the source model to strings of the target model:

```
rule Book2Publication {
  from
    b : Book!Book (
       b.getSumPages() > 2 )
  to
    out : Publication!Publication (
       title <- b.title,
       authors <- b.getAuthors(),
       nbPages <- b.getSumPages())
}
```

However, the following variant of the rule is problematic, because strings of the source model are assigned to numbers of the target model:

```
rule Book2Publication {
  from
    b : Book!Book (
       b.getSumPages() > 2 )
  to
    out : Publication!Publication (
       title <- b.title,
       authors <- b.getSumPages(),
       nbPages <- b.getAuthors())
}
```

Depending on the version, ATL either crashes on execution of this rule, or it silently produces a target model which does not conform to the target meta model. The first kind of behaviour is simply annoying, but produces no result (and a fortiori no wrong result). The second kind of behaviour can be fatal if the target model is subsequently used, but the user is not aware of the incoherence of the target model with respect to its meta model.

Altogether, the state of affairs is reminiscent of the early days of programming language development, when programs were not checked for type correctness. It might be argued that the resulting target model can still be checked after it has been created. This solution is unsatisfactory for several reasons:

- Checking models on a case-by-case basis may be very time-consuming, depending on the size of the model. Therefore, it is better to check a model transformation once and for all.

- The fact that a generated model is faulty does not indicate why it is, and which parts of a set of transformation rules have to be modified.
- A transformation might be split into several smaller transformation steps, each one relying on a well-formed model. If this is not granted, transformations would have to continuously verify the well-formedness of intermediate results, which would require carrying around dynamic type information, thus complicating model transformations considerably.

For these reasons, almost all modern programming languages have chosen a sufficiently strong type system that precludes certain kinds of errors in advance, i.e. statically, or that at least recognize type errors at runtime and signal an error in a controlled way (for example by raising an exception).

## 3.2 Overview of the formalization

In the following, we go a step further: we show under which conditions a transformation rule set generates a model that is well-formed. As the language for defining meta-models is quite expressive (allowing, among others, arithmetic properties to be stated), it is currently not clear which of these conditions can be verified statically and which cannot.

The generic reconstruction of ATL is illustrated by the architecture of the formalization, described graphically in Figure 1, where boxes are modules and dotted boxes are parameter modules.

The modules appearing in this figure are the followin

- `Model` defines the structure of a model as a Ocaml signature.
- `ImplModel` supports the definition of concrete models by implementing the signature `Model`. This module can be viewed as a simplified MOF implementation.
- `Expr` is an abstract structure of OCL expressions.
- `OCL : (M:Model) ⇒ Expr(M)` is a module parameterized by a model which defines OCL-like navigation expressions for that model.
- `ATL` is a module parameterized by an instance of `Expr` and an instance of model which defines the structure of model transformations and their operational semantics.
- `M : ATL(ImplModel)(OCL)` is the actual model transformer obtained by instantiating the generic module `ATL` with a specific model and a specific expression language.

Thus, ATL can be defined quite generically and adapted to a specific navigation formalism.

## 3.3 Rules

When talking about transformation rules, we refer to rules in the spirit of ATL, as the ones seen in Section 3.1. A rule transforms instances of an input model to instances of an output model. Applicability of a rule can be restricted by conditions expressed in OCL [11]. For example, the clause

```
from
  b : Book!Book (
     b.getSumPages() > 2 )
```

in rule `Book2Publication` expresses that the rule can be applied to elements of class `Book` (belonging to a meta-model which is equally called `Book`), under the condition that the book has more than 2 pages.

For each applicable instance, an instance of the target model is created. In ATL, the rule execution engine takes care of applying at most one rule to each instance – an attempt to apply several rules to an instance should lead to a runtime exception. Of course, here again it would be desirable to check that rules are mutually exclusive. In a last step, values are assigned to the attributes of the target instances. Thus, in

```
to
  out : Publication!Publication (
     title <- b.title,
     authors <- b.getAuthors(),
     nbPages <- b.getSumPages())
```

an instance of class `Publication` (of meta-model `Publication`) is created, whose attributes are set with values computed from the input model.

Altogether, a rule

- refers to a source and a target model. We will describe our formalization of models and meta-models in Section 3.4.
- uses OCL constraints to express applicability conditions. Our model of OCL is sketched in Section 3.5.

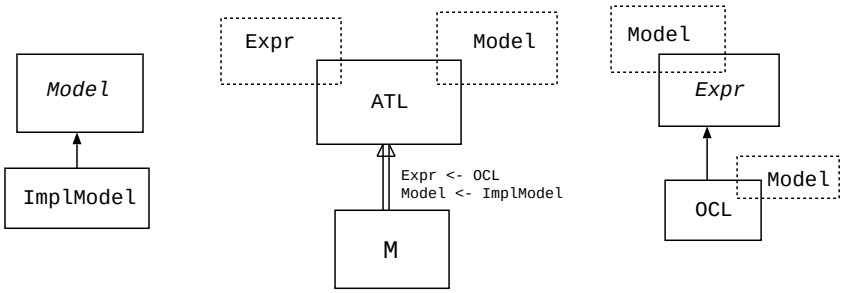


Figure 1: Architecture of ATL formalization

- is applied by an execution engine, as described in Section 3.6.

Due to space limitations, our treatment is necessarily incomplete. We refer the reader to [9] for a more exhaustive description.

The following discussion first presents a set-based description of the concepts, and then eventually an implementation in Ocaml.

## 3.4 Models

A *meta-model* consists of a set of classes (here taken to be identifiers which are left abstract) and a set of roles defined for the selected classes. A role is a partial map from a class and a role name to a class.

Similarly, a *model* consists of a set of instances, attributes and a typing function assigning a class to each instance.

A *type* associates a class to an instance.

$$
\begin{array}{rcl}
Classes & \subseteq & Id \\
Role(Cl) & := & Cl \to (Id \nrightarrow Cl) \\
MModel & := & \{Cl : \mathcal{P}(Classes); \\
 & & Rl : Role(Cl)\} \\
 & & \\
Objects & \subseteq & Id \\
Attribute(O) & := & O \to Id \nrightarrow O \\
Type(O) & := & O \to Classes \\
 & & \\
Model & := & \{Inst : \mathcal{P}(Objects); \\
 & & Att : Attribute(Inst); \\
 & & Tp : Type(Inst)\}
\end{array}
$$

To define well-formedness of a model with respect to its meta-model, we impose some conditions, such as the following, which states that the set of attributes defined for an object coincides with the set of roles defined for its type.

$$
\begin{array}{l}
\forall mm : MModel.\forall m : Model.\forall o \in m.Inst. \\
dom(m.Att(o)) = dom(mm.Rl(m.Tp(o)))
\end{array}
$$

Similar conditions state that types of related objects coincide.

## 3.5 OCL

We first give a view of OCL by means of functions on properties and paths, where we leave the corresponding types `Prop` and `Path` abstract (they will be defined further below). These functions are parameterized by an environment.

$$
\begin{array}{rcl}
checkProp & : & (Id \nrightarrow Classes) * Prop \to Bool \\
checkPath & : & (Id \nrightarrow Classes) * Path \to Classes \\
 & & \\
evalProp & : & (Id \nrightarrow Object) * Prop \to Bool \\
evalPath & : & (Id \nrightarrow Object) * Path \\
 & & \to \mathcal{P}(Object)
\end{array}
$$

We can now implement the types `Prop` and `Path`. We give here a definition in Ocaml as mutually dependent data types.

```
type prop_ty =
    And of prop_ty * prop_ty
  | Not of prop_ty
  | FunBool of fun_bool * path_ty
  | TrueProp
and path_ty =
    Var of string
  | Image of path_ty * string
  | FunPath of fun_path * path_ty
```

By means of example, we show how we can express some OCL expressions in our formalization:

```
(* self.points.couleur *)
let path1=Image (Image
          (Var "self","points"),
```

```
        "color")

(* self->
    select(s|not(s.color->isEmpty())) *)
let path2=FunPath (fun_select
    (Not (FunBool (fun_isempty, (Image
                (Var "self","color"))))),
    (Var "self"))
```

### 3.6 Execution engine

A transformation is given by a source and a target meta-model and a set of rules:

$$\begin{array}{rcl} Transfo & := & \{src : MModel; \\ & & dst : MModel; \\ & & rules : \mathcal{P}(Rule)\} \end{array}$$

As described in Section 3.3, a rule is composed of a filter (a condition governing whether the rule is applicable), a set of generated objects of the target meta-model, and a set of attribute assignments:

$$\begin{array}{rcl} Filter & := & \{prop : Prop; type : Classes\} \\ Assign & := & Id * Role * Path \\ ObjOut & := & \{id : Id; type : Classes\} \\ Rule & := & \{filter : Filter; \\ & & instances : \mathcal{P}(ObjOut); \\ & & ass : \mathcal{P}(Assign)\} \end{array}$$

Take as an example a rule for a (fictitious) reduced meta-model of the AADL [6] architecture description language:

```
rule Model2Model {
from
  model : MiniAADL!Model
to
  newmodel : MiniAADL!Model (
    name <- model.name,
    impls <- model.impls,
    specs <- model.specs
  )
}
```

The filter predicate is true, and we have three assignments, the first of which assigns a name:

```
let the_filter = {
  prop = true_prop ;
  ty = class_Model
}
let assign1 = {
  var_name = "newmodel" ;
  role = "name" ;
  path = ...
}
```

Together with the other assignments, we can now define the rule as:

```
let rule_Model2Model = {
  filter = the_filter ;
  mapsto = "newmodel"
  env_out = [ "newmodel" , class_Model ] ;
  assigns = SetAssign.add assign1
      (SetAssign.add assign2
      (SetAssign.singleton assign3))
}
```

With these ingredients, the semantics of an execution of a transformation can be defined. It proceeds in three steps:

1. filtering, which determines a set of applicable rules and the objects they can be applied to.
2. instantiation, which generates the target objects from the source objects determined in the previous step.
3. assignment, which carries out the assignments on the generated target objects.

Again, transformations give rise to well-formedness conditions, such as:

$$\begin{array}{l} \forall t \in Transfo.\ \forall r \in t.rules. \\ \forall var \in r.instances,\ \forall role \in dom(t.dst.Rl(var.type)). \\ \exists assign \in r.ass. \\ assign.var = var\ \wedge\ assign.att = role \end{array}$$

which says that for all roles occurring in instances of applicable rules, there has to be an assignment. Whereas this condition can be statically checked, another condition expresses that the applicability conditions of rules have to be disjoint:

$$\begin{array}{l} \forall t \in Transfo. \forall obj \in Objects. \\ \exists! rule \in t.rules. \\ isTypeOf(obj, rule.filter.type) \wedge \\ rule.filter.prop(obj) \end{array}$$

Currently, these are proof obligations that would have to be handled by an interactive proof assistant. We are in the process of investigating to which extent subproblems of this condition can be handled by extended type checking.

# 4 Conclusions

In order to study the verification of ATL transformations, we have given a formal semantics of an abstraction of ATL.

ATL [5] as well as most QVT-based transformation languages extend the object constraint language OCL in order to allow object creations. Thus, ATL formalization [9] is parameterized by a constraint and navigation language which can be defined to be OCL later. The navigation language acts on a model which is seen as a graph.

The verification of ATL transformations is a key issue for using such a language for producing safety critical systems. For the moment, very few checks are performed by existing tools. Static verification such as type checking is difficult because ATL rules are implicitly called and their domain is not statically known. Either the expressive power of the language has to be reduced or proof obligations must be generated. We have studied the second way: our formal model performs some static checks and generates proof obligations that could be discharged using a proof assistant.

# References


[1] J. Abrial. *The B-Book Assigning programs to meanings.* Cambridge University Press, 1996.

[2] B. Berthomieu, P.-O. Ribet, F. Vernadat, J.-L. Bernartt, J.-M. Farines, J.-P. Bodeveix, M. Filali, G. Padiou, P. Michel, P. Farail, P. Gaufillet, P. Dissaux, and J.-L. Lambert. Towards the verification of real-time systems in avionics: The Cotre approach. In *Eigth International workshop for industrial critical systems, ROROS*, pages 201–216. Thomas Arts, Wan Fokkink, 5-7 juin 2003.

[3] J.-P. Bodeveix, D. Chemouil, M. Filali, and M. Strecker. Towards formalizing AADL in proof assistants. In J. Kuster-Filipe, I. Poernomo, R. Reussner, and S. Shukla, editors, *Formal Foundations of Embedded software and component-based softare architectures (ETAPS), Edinburgh*, pages 137–153. LFCS (University of Edinburgh), 2005.

[4] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. Grose. *Eclipse Modeling Framework.* Addison-Wesley, 2003.

[5] J. Bézin, E. Breton, G. Dupé, and P. Valduriez. The ATL Transformation-based Model Management Framwork. Technical report, IRIN, 2003.

[6] P. H. Feiler, B. Lewis, and S. Vestal. The SAE Avionics Architecture Description Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering. Technical report, SAE.

[7] P. H. Feiler, B. Lewis, and S. Vestal. The SAE architecture analysis & design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. In *RTAS Workshop 2003*, pages 1–10, May 2003.

[8] O. M. Group. MOF 2.0 Query / Views / Transformations RFP. Technical report, OMG, 2002.

[9] N. Lalevée. Formalisation du langage ATL. Master's thesis, IRIT, Université Paul Sabatier, 2005.

[10] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic.* LNCS 2283. 2002.

[11] Object Management Group. *OCL 2.0 Specification*, June 2005.