Correct-by-Construction Model Transformations from Partially Ordered Specifications in Coq [⋆]

Iman Poernomo and Jeffrey Terrell

Department of Computer Science, King's College London,
Strand, London WC2R 2LS, UK
iman.poernomo@kcl.ac.uk, jeffrey.terrell@kcl.ac.uk

Abstract. This paper sketches an approach to the synthesis of provably correct model transformations within the Coq theorem prover, an implementation of Coquand and Huet's Calculus of Inductive Constructions. It extends work done by Poernomo on *proofs-as-model-transformations* in the related formalism of Martin-Löf predicative Constructive Type Theory. We show how the impredicative theory of Coq, together with its treatment of coinductive types, lends itself to the synthesis of a wider range of model transformations than Poernomo had treated before. We illustrate the practical benefits and potential scalability of our approach by means of a case study taken from industry.

1 Introduction

This paper presents an approach to the synthesis of provably correct model transformations within the Coq theorem prover, an implementation of Coquand and Huet's Calculus of Inductive Constructions.

Model transformations within the Model Driven Architecture (MDA) paradigm are meant to enable designers to focus most of their time on providing a robust, architecturally sound platform independent model. Then, given a particular platform and associated metamodel choice, a designer applies a (possibly off-the-shelf) transformation to automatically obtain a specific, refined model. In this way, MDA eliminates some of the effort spent on making implementation decisions at the specification level, and, ideally, results in a better generated platform specific model than that obtained from a manual process.

MDA is increasingly being taken up by industry and, as a consequence, transformations of interest are becoming more complex and mission critical. It becomes essential, therefore, to have maximal levels of trust in the correctness of model transformations. The informality of MDA as it currently stands makes the approach untrustworthy and dangerous. If model transformations are incorrect, the MDA process can result in software of a lower quality than that produced by traditional software development. A small number of errors in the composition of a number of complex transformations can easily lead to an exponential number of errors in the code, that are difficult to trace and debug.

We solve this problem by leveraging a property of Constructive Type Theory known as the Curry-Howard Isomorphism, where data, functions and their correctness proofs

[⋆] This work was supported by Kennedy Carter Ltd and the Engineering and Physical Sciences Research Council (EPSRC grant EP/G03012X/1, "Higher-Order Refinement Techniques for Model Driven Architecture").

are treated as ontologically equivalent, and where a similar equivalence holds for the related trinity of typing information, program specifications and programs. A practical implication of the isomorphism is that, by proving the logical validity of a model transformation specification, we can automatically synthesize a model transformation that satisfies the specification. Following [7], we call this implication the *proofs-as-model-transformations* paradigm.

This paper presents a significant advance over our previous work in Martin-Löf type theory. Our previous work considered simple specifications of model transformations as relationships between source and target metamodel instances, but gave no detail on how a complex transformation might be derived if its specification involves *multiple* mappings between metamodels built from a large number of interrelated metaclasses. We now give a treatment of a subset of such specifications, which we call *partially ordered specifications*. In such specifications, source and target models can take on any graph structure, with metamodels allowing, as usual, bidirectionality and circular references. However, the *transformation specification* itself is given as a series of mappings, defined over each metaclass of the model via a partially ordered traversal of the source metamodel graph, from an given initial metaclass, with the target model requirements also provided according to a partially ordered traversal of the target metamodel graph. We contend that partially ordered specifications are sufficient to define the majority of industrially useful transformations.

We illustrate the practical benefits and potential scalability of our approach by presenting the end results of a case study we have conducted, involving part of a transformation used at Kennedy Carter for safety critical system development (the domain specific properties of the case study have been obfuscated for confidentiality reasons).

This paper assumes the reader is familiar with the UML representation of classes, relationships and objects and has a partial familiarity with the MOF specification document [5]. A detailed study of constructive type theory can be found in [2] or [8] (we follow the formulation of the latter here). Section 2 presents a brief overview of how proofs and programs are related within the type theory of Coq. Section 3 describes our approach to metamodelling and model transformation development and Section 4 provides an example. Finally, Section 5 contains our conclusions.

2 Proofs and Programs in Coq

This section presents a brief summary of the Calculus of Inductive Constructions (CIC), the formalism implemented by Coq.

2.1 Values and Types

The CIC is based on the principle that everything is a value of a particular type, where values include constants and functions, and types range over

- ordinary inductive data types, such as the booleans `bool` and the natural numbers `nat`;
- co-inductive types, such as the list of natural numbers starting from `10` (an example of an infinite object);
- parametrized types, such as the list of natural numbers `1::2::3::nil`;

– higher order types, such as the type of all basic types `Set`, and the type of all logical propositions `Prop`.

If `v` is a value and `t` is its type, we write `v: t`. For example:

– the natural number `3` is written `3: nat`[1];
– the addition function `plus`, which takes two natural numbers and returns a natural number, is written `plus: nat -> nat -> nat`;
– the logical statement "there exists a natural number greater than zero" is written `exists y: nat, ge y O: Prop`;
– the logical predicate `ge`, which takes two natural numbers and returns a `Prop`, is written `ge: nat -> nat -> Prop`.

2.2 Internal Programming Language (the Lambda Calculus)

Coq's *values* range over a theory of predefined constants and functions (such as the natural numbers and arithmetic functions) and a *lambda calculus* for building new functions. In this calculus, which is essentially an internal functional programming language with a syntax similar to Haskell or SML, functions can be constructed from built-in APIs, and various constructs exist for dealing with recursion, disjoint unions, matching by cases, record types and so on.

An important aspect of this language, as with all functional programming languages, is its support for anonymous functions. In conventional mathematical notation, the lambda abstraction $\lambda x\colon nat. x + x$ defines a function that inputs $x\colon nat$ and returns $x + x$. In Coq, this function is written `fun x: nat => x + x` and its type is `nat -> nat` (the type of functions that take in naturals as input and return naturals as output). Furthermore, since functions can be applied to inputs, the application of `fun x: nat => x + x` to `3: nat`, say, which in its unevaluated form is written `(fun x: nat => x + x) 3`, is clearly a value of type `nat`.

Evaluation is formalized by rules that define how terms can be converted or *reduced*, into simpler terms. The transitive closure of these reduction rules define the system's *normalization* relation, providing the operational semantics for computing the final value of any function application. A key principle of Coq is *strong normalization*: that *all* possible sequences of applications of reduction rules are confluent and terminate. We do not admit terms that could lead to infinite reduction sequences (for example, infinite recursion). As a result, normalization provides a *canonical, normal value* for each term. The lambda calculus of Coq thus forms an internal functional programming language in which every program terminates. We may use it to define new programs from known programs.

2.3 Internal Logic

The CIC works with a collection of logical formulae, each of type `Prop`, which are built from a closure over basic propositions, predicates over terms, implication `A -> B`, conjunction `A /\ B`, disjunction `A \/ B`, typed universal quantification `forall x: T, A` and existential quantification `exists x: T, A`. The symbols `->`, `/\` and `\/` correspond

[1] The symbol `3` is an abbreviation for the expression `S(S(SO))`, where `S` is the successor function, and `O` (the letter, that is) is a representation of zero.

to the usual logical connectives, and the keywords `forall` and `exists` correspond to the quantifiers ∀ and ∃.

The connectives and quantifiers allow us to define well-typed logical propositions within the calculus, and because propositions may involve predicates over terms, we can define propositions that make statements about our lambda calculus (that is, we can specify properties of programs). But, in and of themselves, the logical formulae do not allow us to *reason*: we need rules to define a proof system in order to do that. We will return to this question shortly.

2.4 Co-inductive Types

Co-inductive types merit special attention because they play an important part in the representation of metamodels. A co-inductive type, just like any other type, is defined by "prescribing what we have to do to construct an object of that type" [4]. Let us start by defining a particular co-inductive type, namely

```
CoInductive X : Set :=
   Build'X : nat -> Y -> X
with Y : Set :=
   Build'Y : nat -> X -> Y.
```

This type definition (which actually defines the type of a pair of *mutually* co-inductive types `X` and `Y`, because `X` refers to `Y` and `Y` refers to `X`) tells us that there is only one way in which an object of type `X` can be constructed, and that is to call `Build'X` with an object of type `nat`, and an object of type `Y`. It also tells us something similar about `Y`.

Consider a specific object `x1` of type `X` (as depicted in Fig. 1), which is constructed

$$\begin{tikzcd}
\mathtt{x1} \arrow[d] & \mathtt{x3} \arrow[d] & \cdots \\
\mathtt{y1} \arrow[ur] & \mathtt{y3} \arrow[ur] &
\end{tikzcd}$$

Fig. 1. Object `x1` and its followers.

by evaluating `f 1`, where `f` is defined by

```
CoFixpoint f (n : nat) : X :=
   Build'X n (Build'Y (S n) (f (S (S n)))).
```

Clearly, `f` is recursive (it appears on both sides of the definition) and `x1` is infinite because the evaluation of `f 1`, of which the first few terms are

```
Build'X 1 (Build'Y 2 (Build'X 3 (Build'Y 4 ...,
```

never terminates. However, this does not prevent us from reasoning about `x1`, so long as we restrict ourselves to finite parts or arbitrary elements of `x1`. In that way, our computations are bound to terminate. For example, let us find the object of type `nat` that was used to construct the object of type `X` that follows a specified object of type `X`. Two navigation functions and an access function are required, namely

```
Definition nav'x (y : Y) : X :=
   let (n, x) := y in x.

Definition nav'y (x : X) : Y :=
   let (n, y) := x in y.

Definition val'x (x : X) : nat :=
   let (n, y) := x in n.
```

`nav'x` navigates from an object of type `Y` to the following object of type `X`, and `nav'y` does something similar. Furthermore, `val'x` picks off the object of type `nat` that was used to construct a specified object of type `X`. Applying a suitable composition of functions to `x1` yields `3`, the `nat` that was used to construct `x3`, the object of type `X` that follows `x1`.

```
Eval compute in (val'x (nav'x (nav'y (f 1)))).
= 3 : nat
```

Let us now consider a slightly different example, which is more germane to the representation of metamodels, by constructing a mutually referential pair of co-inductive objects `x1` and `y2` (see Fig. 2) as follows:

```
CoFixpoint x1 : X :=
   Build'X 1 y1
with y1 : Y :=
   Build'Y 2 x1.
```

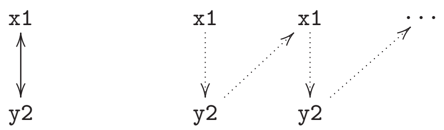


Fig. 2. Mutually referential objects `x1` and `y2`, and navigations back and forth.

Starting from `x1`, it is clearly possible to navigate to `y2` and back again, an arbitrary number of times. As in the previous example, therefore, `x1` is an infinite object. Performing a dual navigation from `x1` followed by an access yields, as expected, `1`.

```
Eval compute in (val'x (nav'x (nav'y x1))).
= 1 : nat
```

What this example shows is that co-inductive types can be used to represent mutually referential objects, of the kind that inhabit most metamodels. We shall discuss this in more detail in Section 3.1.

2.5 Higher-Order Values

Coq implements a higher-order type theory in the sense that it contains *higher-order values* that can be used to *type* other values.

Propositions also possess this dual status. Not only are they values of type `Prop`, but they also stand as *types* of values. The idea is that the inhabiting values of a proposition are the possible *proofs* of that proposition. If a proposition has at least one inhabiting value, then it is said to be true (because it has a proof). In this way, the intended meaning of the various connectives is *constructive*. That is, we understand a formula F to be *true* only if we can construct a proof p that supports or realises its truth, written $p : F$. Rules of inference are included in a straightforward way to accommodate theories of various data types, as is reasoning about the simply typed lambda calculus, induction and recursion. We do not provide these rules here – the reader is referred to [8] for details.

2.6 Extracting Programs from Proofs

The Curry-Howard isomorphism supports the notion of proof-carrying code.

Theorem 1 (Program Extraction). Let `forall x: T, exists y: U, P(x, y)` be a well-formed formula built from well-defined predicates and functions in Coq's type theory. There is a mapping `extract` from proof-terms to *simply typed lambda terms* (terms that do not involve logical propositions) such that, if

$\vdash$ `p: forall x: T, exists y: U, P(x, y)`

is a well typed term, then

$\vdash$ `forall x: T, P(x, extract(p) x)`

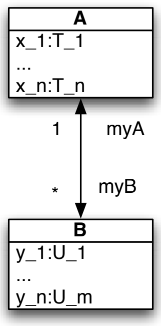
is provable.

Space does not permit us to describe the extraction mapping, but essentially it is developed using the generic machinery of [8]. The implication of this theorem is that, given a proof of a formula `forall x: T, exists y: U, P(x,y)`, we can automatically synthesize a lambda calculus *program* `f` that, given input `x: T`, will *correctly* produce an output `f x` that satisfies the constraint `P(x, f x)`.

3 Doing MDA in Coq

Our notion of proofs-as-model-transformations essentially follows from Theorem 1. A model transformation can be specified as a constraint in the OCL, over instances of an input PIM and an output PSM written in the Meta-Object Facility (MOF) [5], or any comparable constraint and class-based metamodelling languages. Assuming that we can develop types to represent the PIM and PSM metamodels, and that we can write the constraint as a logical formula over the metamodels, we can then specify the transformation as a `forall exists` formula. Then, in order to synthesize a provably correct model transformation, we can prove the formula's truth and apply the extraction mapping according to Theorem 1.

The main technical challenges posed by this approach are twofold. First, the formalization of MOF-based metamodels as types (as input/output types of transformations) is not clear. Second, a complex model transformation typically has a complex specification that will not have a straightforward representation or proof in Coq. Are there common patterns of specification definition and proof (and, consequently, synthesis) that can assist us in dealing with this complexity? These two challenges are now addressed.



Encoding:

```
CoInductive A : Set :=
  Build_A (x_1 : T_1) ...
    (x_n : T_n) (myB : list B)
with B : Set :=
  Build_B (y_1 : U_1) ...
    (y_m : U_m) (myA : A).
```

In addition: 1) a set of projection definitions for obtaining each attribute x_i of A:

```
Definition x_i : A -> T_i :=
  (fun a : A =>
  match a with
    Build_A x_1 ... x_n myB =>
      x_i
  end).
```

2) a definition for navigating to the associated class object list myB:

```
Definition myB : A -> list B :=
  (fun a : A =>
  match a with
    Build_A x_1 ... x_n myB => myB
  end).
```

3) similar definitions for B.

Fig. 3. Co-inductive encoding of metaclass structures involving a bidirectional reference.

3.1 Encoding Metamodels as Types

In [6], metamodel/model instantiation relationships of the MOF were treated using terms and types within the predicative type hierarchy of a Martin-Löf Type Theory. That work suggested a particular co-inductive encoding for bidirectional metaclasses, which required specialized rules to be added to the type theory. Here, we employ a variant implementation, which makes use of the co-inductive type definition schemas built into Coq. The advantage of this new approach is that we readily have an implementation of the metamodels-as-types idea within a robust tool kit.

Our approach is to define a Coq type for each metamodel ModelLang, so that $\vdash$ *model* : ModelLang is derivable if, and only if, the term *model* corresponds to a well formed model instance of the metamodel. Model transformations can then be defined as lambda functions within the internal programming language of Coq, typed by metamodel types.

We employ co-inductive types to represent metaclass references for both unidirectional and bidirectional associations.[2] This makes it possible to define the graph of any metaclass diagram using the `CoFixpoint` operator. For example, the two metaclasses that participate in the one-many bidirectional association in Fig. 3, are encoded as co-inductive types, which utilise the inductive parametrized `list` type to represent multiplicities. The fact that metaclass `A` has a reference to many objects of metaclass `B`, is represented by the constructor of metaclass `A` having a list attribute `myB` of type `list B`.

[2] In very special circumstances, inductive data types can also be used.

With such an encoding, it is a simple matter to instantiate a model with (say) an instance `a_1` of metaclass `A` linked in both directions to instances `b_1` and `b_2` of metaclass `B`, as follows:

```
CoFixpoint a_1 : A :=
   Build_A v_1 ... v_n b_1::b_2::nil
with b_1 : B :=
   Build_B q_1 ... q_m a_1
with b_2 : B :=
   Build_B r_1 ... r_m a_1.
```

The encoding of conditional one-one relationships and generalizations, which we have omitted in this paper, can be accomplished using Coq's `option` and `sum` types. Please follow the link in Section 5 for further details.

3.2 Constraints

In general, a MOF metamodel consists of a metaclass structure over which a set of OCL constraints (defining well-formedness and semantics) range. Fortunately, there is a straightforward mapping from OCL constraints over metamodel elements into logical propositions of type `Prop` in Coq. Utilizing this mapping, it is possible to form a new *subset type* that represents the complete metamodel as a higher-order type, by pairing the structural information given by the encoding of a metamodel `MM` (say) with its constraints `B` (say), as a term `{x: MM | B}` of type `Set`, where `x: MM` is a typed variable, and `B: MM -> Prop` is a predicate function over `MM`.

The general reader may surmise the type's intuitive meaning from ordinary mathematical set theory, since any term `aModel` of type `{x: MM | B}` must not only instantiate the metaclass structure of type `MM`, but it must also satisfy `B[aModel/x]`. However, the difference in Coq is that inhabitation of this type requires *constructive evidence*: that is, an element `aModel` of type `{x: MM | B}` is necessarily a term of the form

```
exist (fun x: MM => B) w p,
```

where `exist` is a constructor that pairs together two important elements: a witness `w` and a proof `p`, which provides the evidence that `B[w/x]` is true. In other words, any instantiation of a metamodel type must include a proof that certifies its conformance to the metamodel's constraints. In this sense, our formalism promotes certified metamodelling.

3.3 Specifications

Whether we intend to handcraft the implementation of a transformation or formally synthesize it, first we need to specify what we expect of the transformation. As we have shown in [7], a simple model transformation between instances of metamodels `M` and `N`, can be specified as a `forall exists` formula of the form

```
forall x: M, Pre(x) -> exists y: N, Post(x, y),        (1)
```

where `Pre(x)` specifies an assumed precondition over instance `x` of metamodel `M`, and `Post(x, y)` prescribes the required input-output relationship between instances `x` and y of metamodels M and N respectively. By Theorem 1, if we can derive a proof of (1), then a provably correct transformation that satisfies the pre- and post-conditions can be obtained.

In [7] we demonstrated that this approach is applicable to the development of transformations from contractual specifications: given a contractual specification written in a first-order language such as the OCL over the input and output metamodel vocabularies, we can map it to a formula of the form (1). That treatment is what might be called an *unstructured*, *single* arbitrary form of contractual specification: one precondition and one postcondition, each potentially very large, predicating over the entirety of two metamodels.

The main problem with that approach is that, in practice, a model transformation is *not* specified as a single contract, but as a *sequence* of interrelated contracts between the elements of the source and target metamodels. When using a declarative model transformation language, if the transformation is sufficiently simple, these specifications can then be taken to be the actual transformation itself. Our approach is not used for synthesis of declarative transformations, but of algorithmic transformations. However, even in the case of algorithmic transformation languages such as Kermeta or the Executable UML toolkit of Kennedy Carter, a specification is always divided and conquered via a modular hierarchy of contractual specifications.

Further, from the perspective of the proofs-as-model-transformations paradigm, a disadvantageous side-effect of beginning with an unstructured, single arbitrary contractual specification, is that while we may have a straightforward mapping to the form of (1), we unfortunately have no heuristics to guide us in deriving its proof and, consequently, in extracting the correct transformation.

By focusing on how a transformation is typically given, on the common interdependencies of the hierarchy of modular requirements, we can also discover common patterns of proof implied by the corresponding structure of the specification in Coq.

3.4 Partially Ordered Specifications

We consider a particular form of complex transformation specification that we refer to as *partially ordered* (PO) in this paper. For the sake of illustration, we shall restrict ourselves to metamodels in which

- associations are one-many;
- metaclasses are associated with at least one other metaclass;
- there is at most one association between any two classes, but their ends may be of any multiplicity;
- there are no generalizations.

However, there is nothing significant about these restrictions.

Definition 1 (Partial Order). Consider a MOF-based metamodel consisting of a set of metaclasses $\mathsf{Class} = \{C_1, \ldots, C_n\}$ and a set of associations $\mathsf{Assoc} = \{R_1, \ldots, R_m\}$ holding between metaclasses. Let $(C_i\ \widehat{R_k}\ C_j)$ hold if there is an association $R_k \in \mathsf{Assoc}$ between C_i and C_j. A *partial order* over the metamodel $\{H, \leq\}$ consists of

- a reflexive, antisymmetric and transitive relation $\leq$, defined over all metaclasses in C so that $C_i \leq C_j$ if, and only if,

$$C_i\ \widehat{R^1}\ C^1\ \widehat{R^2}\ \ldots\ \widehat{R^l}\ C^l\ \widehat{R'}\ C_j$$

for some (possibly empty) chain of associated classes $C^1, \ldots, C^l \in \mathsf{Class}$ and associations $R^1, \ldots, R^l, R' \in \mathsf{Assoc}$;
- a root metaclass $H \in \mathsf{Class}$, such that $H \leq C$ for all $C \in \mathsf{Class}$.

If $C_i \leq C_j$ and there is an association $R \in \mathsf{Assoc}$ such that $(C_i\ \widehat{R}\ C_j)$ holds, then we say that R is a *PO association* (an association that defines the partial order). If an association is not a PO association, we call it an *ancillary association* (an association that cannot be included in a navigation of the metamodel graph conducted only with respect to PO associations).

The basic idea of a PO specification is illustrated in Fig. 4, which shows a specification as a series of sub-specifications that define required mappings between individual metaclasses in a rooted directed graph. The series of sub-specifications is defined according to the partial order of the input metamodel and preserves the partial order of the output metamodel, in the sense that a subspecification over a metaclass A can only involve references to metaclasses that preceed A in the PO, and cannot involve references to metaclasses that are greater than A in the PO.

Definition 2 (PO Specification). Let Source be a metamodel consisting of a set of metaclasses $\mathsf{Source} = \{A_1, \ldots, A_w\}$ and a partial order $\{A_1, \leq_{\mathsf{Source}}\}$ constructed from its set of associations. Let Target be a metamodel consisting of a set of metaclasses $\mathsf{Target} = \{B_1, \ldots, B_v\}$ and partial order $\{B_1, \leq_{\mathsf{Target}}\}$ constructed similarly. A *PO specification* describes a model transformation $\phi : \mathsf{Source} \rightarrow \mathsf{Target}$ via a series of OCL pre and postcondition pairs[3], that is

$$\mathsf{SPEC} = \{(Pre_i(x_i), Post_i(x_i, y_i)) \mid i = 1, \ldots, w\}\ .$$

Each pair specifies how ϕ should operate on an assumed instance x_i of A_i from Source, to produce a corresponding instance y_i of $\phi(A_i)$ from Target.

- The precondition $Pre_i(x_i)$ is a constraint that can predicate over any attribute of x_i and any aspect of $(R\ x_i)$, where R can be either a PO or ancillary association.
- The postcondition $Post_i(x_i, y_i)$, on the other hand, is restricted to predicate over any attribute of x_i or y_i and *only* navigations to, and variable instantiations of, “preceding” associated metaclasses in the POs – that is, it may predicate only over instance variables or navigated references $x' : X'$ of metaclasses $X' \leq_{\mathsf{Source}} A_i$ and instance variables or navigated references $y' : Y'$ of metaclasses $Y' \leq_{\mathsf{Target}} \phi(B)$.

The PO specification demands that ϕ is a surjective homomorphism from A to B, preserving the partial orders, so that if $A_i \leq_{\mathsf{Source}} A_j$, then $\phi(A_i) \leq_{\mathsf{Target}} \phi(A_j)$.

The transformation ϕ is then specified by SPEC to be a function with the following procedural semantics. ϕ is $\mathsf{Proc}(a_1, 1)$, where $\mathsf{Proc}(e, i)$ is defined recursively as follows.

1. Given an instance e of A_i, we add $\phi(e) : \phi(A_i)$ to the output metamodel instance, preserving $Pre_i(e)$ and $Post_i(e, \phi(e))$ in the sense defined above.
2. If $A_i \leq_{\mathsf{Source}} A_j$ and $A_i\ \widehat{R}\ A_j$ (so that R is a PO association) then, for each instance $f : A_j$ contained in the associated reference list $(R\ e)$, we repeatedly invoke $Proc(f, j)$.

[3] We allow free variables in our OCL constraints.

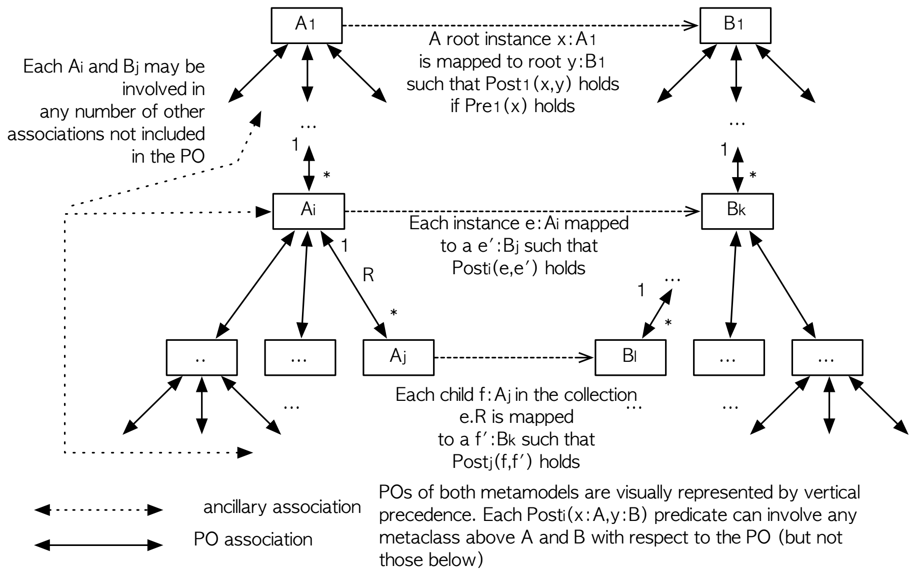

A1
B1
A root instance x:A1 is mapped to root y:B1 such that Post1(x,y) holds if Pre1(x) holds
Each Ai and Bj may be involved in any number of other associations not included in the PO
Ai
Bk
Each instance e:Ai mapped to a e′:Bj such that Posti(e,e′) holds
R
Aj
Bl
Each child f:Aj in the collection e.R is mapped to a f′:Bk such that Postj(f,f′) holds
ancillary association
PO association
POs of both metamodels are visually represented by vertical precedence. Each Posti(x:A,y:B) predicate can involve any metaclass above A and B with respect to the PO (but not those below)

Fig. 4. Relational form of a PO specification. In general, m need not equal n and each i need not equal k for each mapping from A_ni to B_mk.

Remark 1. Our definition of a PO specification is essentially generic with respect to first-order constraint language, metamodelling notation and precise definition of procedural semantics. It should be clear how similar specifications could be provided using constraints written in, for example, OCL over MOF metamodels, contracts over Kermeta metamodels, and even assertions over Java representations of metamodels.

Remark 2. We have shown in [6] that OCL constraints over a MOF-based metamodel have an obvious mapping into a higher-order type theory. It is a trivial extension of that work to show that we have a mapping from such OCL constraints to formulae of type `Pop` in Coq, over elements of the metamodel's encoding. We denote this mapping in Definition 3, by changing the font of the encoded constraint to `typewriter`, and changing the format of logical connectives as appropriate.

For the purposes of developing a guaranteed correct transformation that implements the specification, we need to represent its intended meaning as a formula in Coq's type theory. This is now defined.

Definition 3 (Interpretation of PO Specification in Coq). Consider PIM and PSM metamodels Source and Target consisting of a set of metaclasses Source $= \{A_1, \ldots, A_w\}$ and Target $= \{B_1, \ldots, B_v\}$, with associated partial orders $\{A_1, \leq_{\mathsf{Source}}\}$ and $\{B_1, \leq_{\mathsf{Target}}\}$ derived over their associations. Let

$$\mathsf{SPEC} = \{(Pre_i(x_i), Post_i(x_i, y_i)) \mid i = 1, \ldots, w\}$$

be a *PO specification* describing a model transformation $\phi : \mathsf{Source} \rightarrow \mathsf{Target}$. Furthermore, let A_i be a metaclass from Source. We define

$$\mathtt{Traverse}(\mathtt{x} : \mathtt{A_i}) \equiv \bigwedge_{\mathtt{j=f_i(1)}}^{\mathtt{j=f_i(k)}} \mathtt{Traverse_j}(\mathtt{x} : \mathtt{A_i})$$

$$\mathtt{Traverse_j}(\mathtt{x} : \mathtt{A_i}) \equiv \mathtt{forall}\ \mathtt{x_j} : \mathtt{A_j},\ \mathtt{x_j} \in (\mathtt{R_j}\ \mathtt{x})\ \texttt{->}\ \mathtt{Pre_j}(\mathtt{x_j})\ \texttt{->}$$
$$\mathtt{exists}\ \mathtt{y_j} : \phi(\mathtt{A_j}),\ \mathtt{Post_j}(\mathtt{x_j},\ \mathtt{y_j})\ \texttt{/\textbackslash}\ \mathtt{Traverse}(\mathtt{x_j})$$

if there is a nonempty set $\{A_{f_i(1)}, \dots, A_{f_i(k)}\}$ of all metaclasses, constructed so that for each $l = 1, \dots, k$, there is a $R_{f_i(l)}$ where $(A_i\ \widehat{R_{f_i(1)}}\ A_{f_i(l)})$ and $A_i \leq_{\mathsf{Source}} A_{f_i(l)}$, with f_i forming an injection $\{1, \dots, k\} \mapsto \{1, \dots, w\}$. If there is no such set, then

$$\mathtt{Traverse}(\mathtt{x} : \mathtt{A_i}) \equiv \mathtt{True}\ .$$

With these definitions, the *type theoretic interpretation of the PO specification* in Coq becomes

$$\mathtt{forall}\ \mathtt{x_1} : \mathtt{A_1},\ \mathtt{Pre_1}(\mathtt{x_1})\ \texttt{->}\ \mathtt{exists}\ \mathtt{y_1} : \mathtt{B_1},\ \mathtt{Post_1}(\mathtt{x_1}, \mathtt{y_1})\ \texttt{->}\ \mathtt{Traverse}(\mathtt{x_1})\ . \qquad (2)$$

We then have the following theorem.

Theorem 2 (Transformations from PO Specifications). Consider PIM and PSM metamodels Source and Target with a PO specification SPEC as assumed in Definition 3. Let $P(\mathsf{SPEC})$ be the type theoretic interpretation of the PO specification (2). If $\vdash p : P(\mathsf{SPEC})$ is a well-typed term, then there is mapping $\mathsf{extract}$ such that $\mathsf{extract}(p) : \mathsf{Source} \rightarrow \mathsf{Target}$ is a model transformation function that satisfies the procedural semantics prescribed by SPEC according to Definition 2.

Proof. The proof follows by an extension of that given in [8] and reasoning over the definition of the procedural semantics of PO specifications (sketched in Definition 2).

This theorem allows us to synthesize provably correct model transformations from PO specifications. Assuming that PO specifications are scalable to a wider range of model transformation requirements, then this theorem suggests that the proofs-as-model-transformations paradigm is similarly scalable.

A further implication of this theorem is that the logical structure of (2) immediately simplifies the task of proof derivation (and, consequently, of synthesis), because each conjunct of $\mathsf{Traverse}(\mathtt{x} : \mathtt{A_i})$ begins with a universal quantification over a variable $\mathtt{x_{f_i(1)}} : \mathtt{A_{f_i(1)}}$ that both inhabits a *list* of instances $(\mathtt{R_{f_i(1)}}\ \mathtt{x}) : \mathtt{list}\ \mathtt{A_{f_i(1)}}$ and satisfies the precondition. This suggests that each conjunct might most easily be derived through application of list induction. The choice of induction is generally one that a human prover must make in Coq. However, once this is done, remaining proof steps can often be automated. We have found that a typical (2) specification can then be derived by selecting induction rules systematically as suggested by the quantification clauses, in conjunction with automated intermediate proof steps.

Remark 3. Co-induction presents the complications of a bisimular notion of equality and of infinite co-inductive schema for reasoning over the structure of metaclass types. In the case of PO specifications and the approach we take to deriving proofs from them, these complications do not present a problem for our purpose of extracting model transformations from proofs of specifications involving formal metamodel types, because the proofs are generally not made over the structure of the metamodel types. Co-induction is only important for potentially complex bidirectional navigations between elements of metaclass types, within the specification and constructive content of the proof.

4 Case Study

The majority of large-scale model transformations, covering a wide range of industrial requirements, can be defined as PO specifications. This assertion is based on the experience of the authors in model transformation development but, of course, requires a larger empirical study to verify.

In lieu of such a study, we consider an important fragment of a model-to-text transformation developed at Kennedy Carter, which distributes platform-independent UML models across multiple processes.

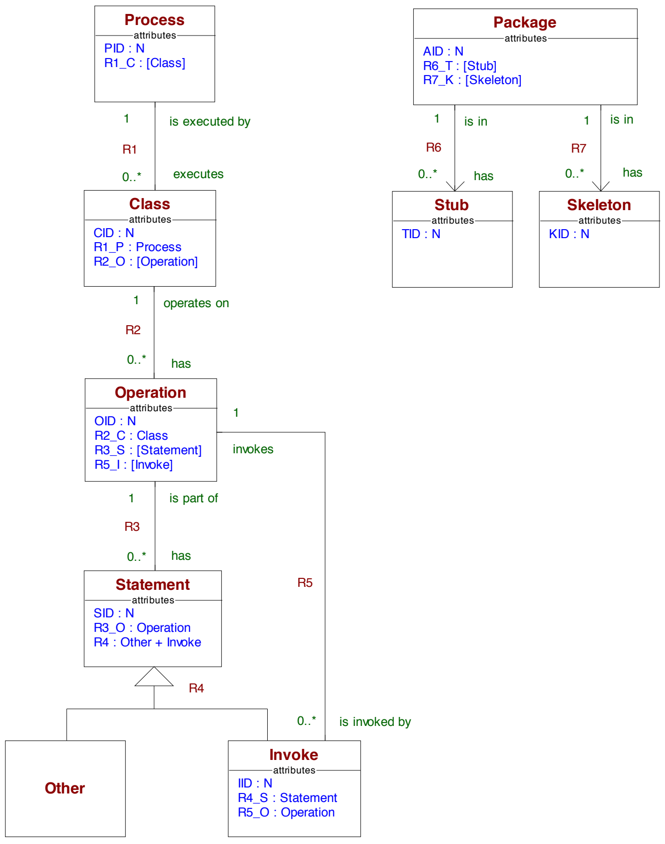


Fig. 5. The Source (LHS) and Target (RHS) metamodels.

The input metamodel Source (the left hand side metamodel of Fig.5) is a fragment of executable UML, in which classes have operations, operations have statements and statements are either invocation statements or (simplifying for the purposes of this illustration) statements of other unspecified kinds. Each class is associated with an operating system process during system configuration, so that its operational code will run in the context of a specific process.

The purpose of the transformation is to instantiate an output metamodel Target (the right hand side metamodel of Fig.5), in which there is a package for each process, and within each package, appropriate stubs and skeleton code for each remote operation invocation.

Remote operation invocation is understood in the following sense. Let i be an invocation statement in Source and let p_i be the process that executes the class that has the operation that contains i. Furthermore, let p_j be the process that executes the class that has the operation that i invokes. Clearly, $p_i = i.R4.R3.R2.R1$ and $p_j = i.R5.R2.R1$. If $p_i \neq p_j$, the invocation is said to be remote. A remote invocation is implemented by a *stub* in the source process and a *skeleton* in the destination process, to manage the flow of control and data between the invoking and invoked operations.

Consider a PO $\{Process, \leq_{\mathsf{Source}}\}$ over the source metamodel that preserves the vertical order of Fig. 5, so that $A \leq_{\mathsf{Source}} B$ if class A is higher than B in the figure, and in which all associations are PO apart $R5$, which is ancillary. Similarly, consider a PO $\{Package, \leq_{\mathsf{Target}}\}$ over the target metamodel, in which all associations are PO.

The transformation ϕ : Source $\rightarrow$ Target is then defined by a PO specification, whose content can be written (in informal English, but a first-order logical representation is straightforward) as the following sequence of pre and postconditions over metaclasses *Process* and *Invoke* (the other pre and postconditions are nugatory and can be assumed to be "True"):

i	Source_i	$\phi(\mathsf{Source}_i)$	$Pre_i(x)$	$Post_i(x, y)$
1	*Process*	*Package*	True	$x.PID = y.AID$, where $x : Process$ and $y : Package$
6	*Invoke*	*Stub*	x is remote	$x.R5_O.OID = y.TID$, where $x : Invoke$ and $y : Stub$

Each metaclass is encoded as a co-inductive type. For example, consider the following fragment of the encoding of the source metamodel:[4]

```
CoInductive Process : Set :=
   Build_Process (PID: nat) (R1_C: list Class)
with Class : Set :=
   Build_Class (CID: nat) (R1_P : Process) (R2_O: list Operation)
with Operation : Set :=
   Build_Operation (OID: nat) (R2_C: Class) (R3_S: list Statement)
      (R5_I: list Invoke)
...
```

[4] We omit the treatment of generalization here, treating $\mathtt{R}_4$ as a bidirectional relationship between `Statement` and `Invoke`. In our full version, this is treated as a bidirectional relationship to the disjoint union type of `Other` and `Invoke` metaclasses, allowing us to represent the fact that two children can access the attributes of their supertype, but have no access to each others' attributes.

The PO specification of the transformation is then defined in Coq by the following proposition:

```
forall $l_p$: list Process, exists $l_a$: list Package,
    forall p: Process,
        p ∈ $l_p$ -> exists a: Package,
            a ∈ $l_a$ /\
            a.AID = p.PID /\
            forall c: Class,
                c ∈ p.$R1_C$ -> forall o: Operation,
                    o ∈ c.$R2_O$ -> forall s: Statement,
                        s ∈ o.$R3_S$ -> $equal_P$ p i.$R5_O$.$R2_C$.$R1_P$ = False ->
                            exists t: Stub,
                                t ∈ a.$R6_T$ /\
                                t.TID = i.$R5_O$.OID .
```

Note that we have simplified the form of the specification, removing references to pre or postconditions if they are $True$.

The proof of the specification proceeds by induction over the lists mentioned in the respective Traverse clauses: l_p, $p'.R1_C$, $c'.R2_O$ and $o'.R3_S$. While the full proof is considerable, besides the careful choice of list induction (suggested by the form of the specification), much of the derivation is achieved automatically in Coq. The extracted model transformation consists of a series of list recursions (corresponding to the uses of induction in the proof) and comes quite close to the kind of transformation a human developer might produce – but with the advantage of being guaranteed correct.

5 Related Work and Conclusions

A number of authors have attempted to provide a formal understanding of metamodelling and model transformations. Ruscio et al. have made some progress towards formalizing the KM3 metamodelling language using the Abstract State Machines [10]. Rivera and Vallecillo have exploited the class-based nature of the Maude specification language to formalize metamodels written in the KM3 metamodelling language [9]. The intention was to use Maude as a means of defining dynamic behaviour of models, something that our approach also lends itself to. Their work has the advantage of permitting simulation via rewriting rules. A related algebraic approach is given by Boronat and Meseguer in [1]. These formalisms are useful for metamodel verification purposes, but are currently not amenable to the test case generation problem.

Rule-based model transformations (in contrast to procedural/functional ones found in language such as Kermeta and Converge), have a natural formalization in graph rewriting systems [3]. However, their approach is by definition applicable within the rule-based paradigm: in contrast, because our tests are contractual and based in the very generic space of constructive logic, we need not restrict ourselves to rule-based transformations.

We believe that model transformations will never reach their full industrial potential without some guarantee of correctness. This is in contrast to ordinary business programming, where correctness has a benefit-cost ratio that is logarithmic with respect to completeness of guaranteeing proofs. However, a single error in a transformation can have potentially drastic and untraceable effects on code. In terms of benefit-cost ratios, model transformations, while potentially useful throughout all sectors of development, including enterprise computing, should be developed once and verified by experts. We believe that CIC and Coq are a natural choice, as they permit encoding of models, metamodels, transformation specifications and model transformations within a single language and with uniform semantics. For this reason, we see this work as opening up a very promising line of research for the formal metamodelling community. Further details of the case study can be found at

`http://refine-mda.group.shef.ac.uk/sites/default/files/report.pdf` .

References

1. Artur Boronat and José Meseguer. An algebraic semantics for the MOF. In José Luiz Fiadeiro and Paola Inverardi, editors, *FASE 2008. Proceedings*, volume 4961 of *LNCS*, pages 377–391. Springer, 2008.
2. Robert Constable, N. Mendler, and D. Howe. *Implementing Mathematics with the Nuprl Proof Development System.* Englewood Cliffs, NJ: Prentice-Hall, 1986. Updated edition available at `http://www.cs.cornell.edu/Info/Projects/NuPrl/book/doc.html` (Accessed May 2003).
3. Alexander Königs and Andy Schürr. Multi-domain integration with MOF and extended triple graph grammars. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005.
4. P. Martin-Löf. An Intuitionstic Theory of Types: Predicate Part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium.* North-Holland, Oxford, 1973.
5. OMG. *Meta Object Facility (MOF) Core Specification, Version 2.0.* Object Management Group, January 2006.
6. Iman Poernomo. A type theoretic framework for formal metamodelling. In *Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*, pages 262–298. Springer, 2006.
7. Iman Poernomo. Proofs-as-model-transformations. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations, First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008, Proceedings*, volume 5063 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2008.
8. Iman Poernomo, John Crossley, and Martin Wirsing. *Adapting Proofs-as-Programs: The Curry-Howard Protocol.* Monographs in computer science. Springer, 2005.
9. José Rivera and Antonio Vallecillo. Adding behavioural semantics to models. In *The 11th IEEE International EDOC Conference (EDOC 2007), 15-19 October 2007, Annapolis, Maryland, USA*, pages 169–180. IEEE Computer Society, 2007.
10. Davide Di Ruscio, Frédéric Jouault, Ivan Kurtev, Jean Bézivin, and Alfonso Pierantonio. Extending AMMA for supporting dynamic semantics specifications of DSLs. Technical Report 06.02, Laboratoire d'Informatique de Nantes-Atlantique (LINA), Nantes, France, April 2006.