Assembling the Proofs of Ordered Model Transformations

Maribel Fernández Jeffrey Terrell
Department of Informatics, King's College London, Strand, London WC2R 2LS, UK

jeffrey.terrell@kcl.ac.uk

In model-driven development, an ordered model transformation is a nested set of transformations between source and target classes, in which each transformation is governed by its own pre and post-conditions, but structurally dependent on its parent. Following the proofs-as-model-transformations approach, in this paper we consider a formalisation in Constructive Type Theory of the concepts of model and model transformation, and show how the correctness proofs of potentially large ordered model transformations can be systematically assembled from the proofs of the specifications of their parts, making them easier to derive.

1 Introduction

In this paper, we outline a mechanism to assemble correctness proofs of model transformations in the context of Model Driven Development (MDD). Although MDD is in widespread use, it is essentially an informal approach to software development which does not guarantee the correctness of model transformations. High-trust solutions are essential if MDD is to be used in safety critical systems and beyond.

The problem of establishing the correctness of a model transformation is well established, and work has been done towards formalising the process using for instance rewriting languages (e.g. Maude [4]) or typed multigraphs [24]. However, these approaches are first-order and do not reflect the higher-order nature of the UML-based techniques. The aim of our research is to lay the foundations on which a range of *certified* model transformations might be built, following a line of work that started in [17], where the use of constructive type theory to implement model transformations was first discussed. The notion of an ordered model transformation was introduced in [18], to describe how a complex transformation between models, built from a potentially large number of interrelated classes, might be derived from the specification of a series of mappings between classes, via a partially ordered traversal of the source and target models. This paper represents a significant advance on that work in that it a) formally defines the specification of an ordered model transformation in type theory, and b) provides a mechanism for assembling the proofs of ordered model transformations from their constituent parts.

In this paper, a *model* is a Unified Modelling Language (UML) [7] class model, and a *model transformation* is a function which maps the artefacts of a source model (classes, attributes and relationships) onto the artefacts of a target model [11, 12]. UML is a graphical language for specifying the structure and behaviour of object oriented systems. It is also a pillar of the Object Management Group's (OMG) Model Driven Architecture (MDA) [6] (a particular brand of MDD), along with the transformation language Query/View/Transform (QVT) [8] .

Consider a transformation between two models (see Fig. 1) in which each object of X is transformed into an object of Y, via a precondition at X and a postcondition at Y. In general, the postcondition at Y is composed of three components: *Data* asserts a relation between the attributes of X and Y; *Link* asserts a relation between Y and the class that contains it; and *Nest* defines the specification (in context) of the

B. Buhnova, L. Happe, J. Kofroň:
Formal Engineering Approaches to Software Components and Architectures 2013 (FESCA'13)
EPTCS 108, 2013, pp. 63–77, doi:10.4204/EPTCS.108.5

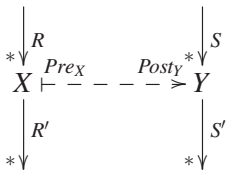

Figure 1: A transformation between classes X and Y, which is subject to a precondition on X and a postcondition on X and Y.

transformation between the classes that X and Y contain (clearly, if Y were a root class, the postcondition would not have a *Link* component, and if Y were a leaf class, the postcondition would not have a *Nest* component).

While *Data* components vary significantly between transformations (there is no reason why they should be the same), the *Link* and *Nest* components are generally quite similar. In fact the only assertion that a *Link* component can make is that Y participates in a relationship with the class that contains it; and all that a *Nest* component can do is pass control of the specification to the classes that X and Y contain. This opens up the prospect of removing from users the tedious task of proving *Link* components by hand.[1] Of course, this prospect only presents itself by virtue of the *ordered* nature of the models and transformations under consideration, where order is defined by *containment*. However, such transformations are sufficiently common in practice (see, for instance, [11] for examples) to make this a worthwhile pursuit.

In this paper, we focus on a particular but nonetheless ubiquitous kind of model transformation, in which the source and target models are either partially or totally ordered.[2] In particular, based on the definitions of model and model transformation given in [17], we show that the proofs of the specifications of large ordered model transformations can be systematically assembled from their parts, making them easier to derive. Our main contribution is a method to derive correctness proofs for ordered model transformations by assembling the proofs of their parts, within constructive type theory. We illustrate the method with examples.

The proofs in this paper have all been implemented in the Coq Proof Assistant [1], see the Coq scripts at `http://www.inf.kcl.ac.uk/pg/terrellj`.

The paper is organised as follows: In Section 2, we give a brief introduction to MDA and type theory, to try to make the paper self contained. In Section 3, we show how to formalise a model transformation (the specification and its correctness proof) in constructive type theory, including the key notion of a parametric proof (a proof with a *hole* over which it is possible to quantify and hence parametrise). We then use this idea to formally specify ordered model transformations in general in Section 4. In Section 5, there is a concrete example of an assembled proof. Finally, in Section 6, we sum up and discuss future developments.

[1] The proof of an assertion involving a many-valued relationship requires a proof by list induction, and the proof of a chain of many-valued relationships, which is not uncommon, requires a nested set of proofs by list induction.

[2] Hierarchical models are the rule rather than the exception in industry (the UML metamodel is fundamentally hierarchical), where transformations are notable for their size rather than their complexity.

2 Preliminaries

We recall the basic notions of Model Driven Architecture (MDA) and Constructive Type Theory (CTT) that are used in the paper. We refer the reader to [11] and [13] for more details on MDA and CTT, respectively.

2.1 Model Driven Architecture

The movement away from the machine to a higher level of abstraction began in earnest in the early 1990s, with the advent of a number of object-oriented analysis and design methodologies. The most influential, in the authors' view, was the one proposed by Shlaer and Mellor in [21, 22], for it played a huge part in shaping the MDA a decade later. The aims of the MDA are two-fold: first, that software systems should be developed independently of the platforms on which they will eventually run, and second, that they should be translated into specific implementations using standard parts, namely model to text and model to model transformations.

In its simplest form, a model transformation takes as input a model, written in a source modelling language (IL), and outputs a new model, written in a possibly different target modelling language (OL). The transformation should be applicable to any model written in IL, therefore it can be seen as a mapping from elements of IL to elements of OL.

In MDA, both the input and output languages are defined as metamodels within the Meta-Object Facility (MOF) [15]. Metamodelling in the MOF is usually done according to a four level hierarchy [14]. The levels are related by an object-oriented style class/object instantiation relationship: classes at level M_{i+1} provide descriptions of objects at level M_i. Roughly speaking, we can think of entities at the M_0 level as objects representing instances of an M_1 UML class. The M_2 level is where metamodels are defined. Metamodels are collections of instances of the M_3 level classes (meta-meta-classes). The M_3 level of the MOF model is used to classify the elements that make up an M_2 level metamodel.

Following [17, 18], in this paper we will consider model transformations as higher-order functional programs satisfying certain pre and post conditions.

2.2 Constructive Type Theory

The type theory below is based on the one proposed by Martin-Löf [13]. A type is defined by prescribing how its inhabitants are formed. For example, if S is the successor function, then the inhabitants of the type Nat are given by

$$\frac{}{0 : Nat} \qquad \frac{n : Nat}{(S\,n) : Nat} \ .$$

If A and B are types, then $A \wedge B$, $A \vee B$ and $A \to B$ are defined to be types too, where $A \wedge B$ is inhabited by a pair of inhabitants of A and B, $A \vee B$ is inhabited by an inhabitant of A or B, together with an indication as to whether it is an inhabitant of A (on the left) or B (on the right), and $A \to B$ is inhabited by a function from A to B, i.e.

$$\frac{a : A \quad b : B}{\langle a, b \rangle : A \wedge B} \qquad \frac{a : A}{inl\ a : A \vee B} \qquad \frac{b : B}{inr\ b : A \vee B} \qquad \frac{\begin{array}{c}[a : A] \\ \vdots \\ b : B\end{array}}{\lambda a : A\,.\,b : A \to B} \ .$$

If a is an inhabitant of A, and $B(a)$ is a type whose inhabitants depend on a, then $\forall a\colon A\,.\,B(a)$ and $\exists a\colon A\,.\,B(a)$ are defined to be types, where $\forall a\colon A\,.\,B(a)$ is inhabited by a function that takes A to $B(a)$, and $\exists a\colon A\,.\,B(a)$ is inhabited by a pair of inhabitants of A and $B(a)$, i.e.

$$\frac{\begin{array}{c}[a\colon A]\\ \vdots\\ b\colon B(a)\end{array}}{\lambda a\colon A\,.\,b\colon \forall a\colon A\,.\,B(a)} \qquad \frac{a\colon A \quad b\colon B(a)}{\langle a,b\rangle\colon \exists a\colon A\,.\,B(a)}\ .$$

One particular type that we shall meet often in this paper is

$$\forall a\colon A\,.\,P(a) \to \exists b\colon B\,.\,Q(a,b)\ ,$$

which defines the specification of a transformation that takes a source class A to a target class B, subject to precondition $P(a)$ and postcondition $Q(a,b)$. Note that types $A \wedge B$ and $A \to B$ are special cases of types $\forall a\colon A\,.\,B(a)$ and $\exists a\colon A\,.\,B(a)$ where B is independent of a. A term $\lambda a\colon A.b$ of type $A \to B$ represents a function from A to B.

Application is written simply as juxtaposition:

$$\frac{t\colon \forall a\colon A.B(a) \quad s\colon A}{(t\ s)\colon B(s)}\ .$$

Further, the reduction relation is generated by the β-rule:

$$(\lambda a\colon A.t)s \to t\{a \mapsto s\}$$

The reflexive and transitive closure of the one-step reduction relation $\to$ is denoted by $\twoheadrightarrow$.

We shall also add to our language a predicate $=$, which we can use to build dependent types like $x = 0$, where $x\colon Nat$. When 0 is substituted for x, the type becomes $0 = 0$, which is inhabited by $r(0)$ (see [25] for more details); when 1 is substituted for x, the type becomes $1 = 0$, which is uninhabited. Lastly, we shall add the type $[E]$ of lists of elements of type E to our language, and two distinguished types *Set* and *Prop*, which will be used to classify types.

3 Type Theory for Model Transformations

In this section, we formalise UML classes and objects using constructive type theory.

Definition 1. *A UML class C is encoded as a type, that is, an inhabitant of type Set; and a UML object of C is encoded as an inhabitant of type C. Furthermore, a base attribute of C is encoded as an inhabitant of type $C \to \tau_1$, where τ_1 is a ground type, e.g. Nat; and a referential attribute of C is encoded as an inhabitant of type $C \to \tau_2$, where τ_2 is the type of some other UML class or class list.*

We shall assume that every UML class C has a single base attribute Id_C of type Nat, and as many referential attributes as it needs to encode the relationships in which it participates. For example, if C is linked by a one-valued relationship to UML class D, and a many-valued relationship to UML class E, then the rule for constructing the inhabitants of C is as follows, where $@_C$ denotes an anonymous constructor of C.

$$\frac{n\colon Nat \quad d\colon D \quad l\colon [E]}{@_C\,n\,d\,l\colon C}\ .$$

$$A \overset{Pre_A}{\vdash\!\!-\!\!-\!\!-\!\!-\!\!-\!\!-\!\!-\!\!\rightarrow} P$$

Figure 2: A transformation between A and P.

The judgement $a\colon A$ admits several different readings: a is an inhabitant of type A (as above), a is a program whose specification is A (which may be that of a model transformation), and a is a proof of proposition A (which may be that of a precondition). In the last reading, A is defined to be an inhabitant of type $Prop$, where A is considered to be true if and only if it is inhabited. The relationship between propositions and types, which was first discovered by Curry [5] and later extended by Howard [9], is known as the Curry-Howard isomorphism.

In this paper, we describe a technique to derive proofs of potentially large ordered model transformations. To illustrate the ideas underlying this technique, we consider first a simple model transformation, where each object of class A (see Fig. 2) is transformed into an object of class P, subject to a precondition Pre_A of type $A \to Prop$ and a postcondition $Post_P$ of type $A \to P \to Prop$. [3] The specification of the transformation is formalised as a type, i.e.

$$\forall a\colon A\,.\,Pre_A\,a \to \exists p\colon P\,.\,Post_P\,a\,p\;, \tag{1}$$

and its proof is given by

$$\dfrac{\dfrac{\dfrac{\dfrac{[a\colon A]^1 \quad \begin{array}{c}[h\colon Pre\,a]^2\\ \vdots\\ Hole\end{array}}{u\colon Post\,a\,\overline{p}}}{\langle \overline{p}, u\rangle\colon \exists p\colon P\,.\,Post\,a\,p}\;(\exists I)}{\lambda h\,.\,\langle \overline{p}, u\rangle\colon Pre\,a \to \exists p\colon P\,.\,Post\,a\,p}\;(\to I)_2}{\lambda a\,.\,\lambda h\,.\,\langle \overline{p}, u\rangle\colon \forall a\colon A\,.\,Pre\,a \to \exists p\colon P\,.\,Post\,a\,p}\;(\forall I)_1\;, \tag{2}$$

i.e. a function that takes an object a of A and a proof h of $Pre_A\,a$, and returns as a pair the corresponding object $\overline{p}$ of P and a proof u of $Post_P\,a\,\overline{p}$. There is a hole in the proof above because the transformation is under specified. However, given suitable definitions of A, P, Pre_A and $Post_P$, the hole could be filled and the proof completed. Furthermore, given a second transformation with a different set of definitions of A, P, Pre_A and $Post_P$, we could apply the same procedure. However, the proofs would be so similar, at least in outline, that it should be possible to capture them all in a parametrised proof, by quantifying over all source and target classes, pre and postconditions, and proofs of holes, in the specification of the transformation. Based on this idea, we define the following correctness condition.

Definition 2 (Correct Model Transformation)**.** *A correct model transformation from X to Y should ensure that for each x in X that satisfies the precondition there is a y in Y that satisfies the postcondition. This is formalised using the following type:*

$$\begin{array}{c}\forall X\colon Set\,.\,\forall Y\colon Set\,.\,\forall Pre\colon X \to Prop\,.\,\forall Post\colon X \to Y \to Prop\,.\\ \forall f\colon X \to Y\,.\,\forall Hole\colon (\forall x\colon X\,.\,Pre\,x \to Post\,x\,(f\,x))\,.\\ \forall x\colon X\,.\,Pre\,x \to \exists y\colon Y\,.\,Post\,x\,y\;.\end{array} \tag{3}$$

[3] Preconditions serve several purposes. First, to allow a choice of rules in different cases, e.g. by checking that a class is a root class, if root classes are transformed by a different rule to non-root classes. Second, to ensure that a postcondition is well-defined, e.g. by insisting that $x \geq 0$ if the postcondition takes the square root of x. Third, to ensure that only certain source elements are transformed, e.g. by checking that a class is persistent, if only persistent classes are mapped to database tables. In the first and second cases, we might expect the precondition to contribute to the proof of the postcondition.

The proof of (3) is little more than two eliminations and a sequence of introductions, i.e.

$$
\begin{array}{c}
[X : Set]^1 \\
[Y : Set]^2 \\
[Pre : X \to Prop]^3 \\
[Post : X \to Y \to Prop]^4 \\
[f : X \to Y]^5 \\
\vdots
\end{array}
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[Hole : (\forall x : X . Pre\,x \to Post\,x\,(f\,x))]^6 \quad [x : X]^7}{Pre\,x \to Post\,x\,(f\,x)}\,(\forall E) \qquad [h : Pre\,x]^8}{Post\,x\,(f\,x)}\,(\to E)}{\exists y : Y . Post\,x\,y}\,(\exists I)}{Pre\,x \to \exists y : Y . Post\,x\,y}\,(\to I)_8}{\forall x : X . Pre\,x \to \exists y : Y . Post\,x\,y}\,(\forall I)_7}{\forall Hole : (\forall x : X . Pre\,x \to Post\,x\,(f\,x)) . \forall x : X \ldots}\,(\forall I)_6}{\forall f : X \to Y . \forall Hole : (\forall x : X . Pre\,x \to Post\,x\,(f\,x)) \ldots}\,(\forall I)_5}{\forall Post : X \to Y \to Prop . \forall f : X \to Y \ldots}\,(\forall I)_4}{\forall Pre : X \to Prop . \forall Post : X \to Y \to Prop \ldots}\,(\forall I)_3}{\forall Y : Set . \forall Pre : X \to Prop \ldots}\,(\forall I)_2}{\forall X : Set . \forall Y : Set \ldots}\,(\forall I)_1 \; . \tag{4}
$$

The fixed outline shape of the proof is captured by rules $(\exists I)$ to $(\forall I)_7$, and the variable proof of the hole is captured by assumption 6. Furthermore, the function K defined below can easily be shown to inhabit (3).

$$
K =_{df} \lambda X . \lambda Y . \lambda Pre . \lambda Post . \lambda f . \lambda Hole . \lambda x . \lambda h . \langle (f\,x), u \rangle \; . \tag{5}
$$

Note that the arguments X and Y are arbitrary source and target classes; Pre and $Post$ are arbitrary pre and postconditions; f is a function that maps source objects to target objects; $Hole$ is a proof of the hole (see (2)); x is a source object; and h is a proof that the precondition holds on the source object. K returns a target object $(f\,x)$, and a proof u that the postcondition holds on the source and target objects.

We shall now apply K to a particular transformation, i.e. the one between A and P. Let Pre_A be a predicate that holds on all objects of A, and let $Post_P$ be a predicate that holds on all objects of A and P which have the same base attribute values. Formally, let

$$
\begin{aligned}
Pre_A &=_{df} \lambda a . \top \\
Post_P &=_{df} \lambda a . \lambda p . (Id_A\,a = Id_P\,p) \; .
\end{aligned}
$$

Now, if

$$
f_A =_{df} \lambda a . @_P\,(Id_A\,a) \; ,
$$

then the proof of the hole is

$$
\begin{array}{c}
[a : A]^1 \quad [h : \top]^2 \\
\vdots
\end{array}
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\ulcorner Id_A : A \to Nat \urcorner \qquad [a : A]^1}{Id_A\,a : Nat}\,(\to E)}{r(Id_A\,a) : Id_A\,a = Id_A\,a}\,(II)}{\lambda h . r(Id_A\,a) : \top \to Id_A\,a = Id_A\,a}\,(\to I)_2}{\lambda a . \lambda h . r(Id_A\,a) : \forall a : A . \top \to Id_A\,a = Id_A\,a}\,(\forall I)_1}{\lambda a . \lambda h . r(Id_A\,a) : \forall a : A . Pre_A\,a \to Post_P\,a\,(f_A\,a)}\,(=_{df}) \; .
$$

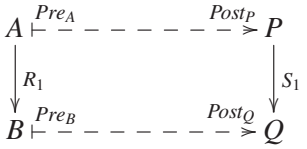

Figure 3: A two-runged transformation

Furthermore, if

$$Hole_A =_{df} \lambda a.\lambda h.r(Id_A\, a)\ ,$$

then

$$K\, A\, P\, Pre_A\, Post_P\, f_A\, Hole_A\, (@_A\, 1)\, Triv \twoheadrightarrow \langle (@_P\, 1), r(Id_A\, (@_A\, 1))\rangle\ .$$

Therefore, $@_P\, 1$ is the transform of $@_A\, 1$, and $r(Id_A\, (@_A\, 1))$ is the term that proves it.

In the next sections we generalise these ideas to families of ordered model transformations.

4 Ordered Model Transformations

In the previous section, we showed how the specification of a small transformation could be abstracted into the specification of a family of transformations, by quantifying over all possible source and target classes, pre and postconditions and holes. The proof that resulted contained a fixed part, common to all members of the family, and a variable part, specific to a particular member. In this section, we extend these ideas to ordered model transformations in general.

There are two kinds of ordered transformations: totally and partially ordered. Informally, a totally ordered set of transformations is one that has the shape of a ladder, in which the source and target models are the verticals, and the transformations are the rungs. A partially ordered set of transformations has the shape of a tree of ladders, in which the branches between nodes are totally ordered transformations. We give examples before presenting the formal definitions.

Consider the two-runged transformation in Fig. 3, in which A is transformed to P subject to conditions Pre_A and $Post_P$, and B is transformed to Q subject to conditions Pre_B and $Post_Q$. If an object a of A is transformed to an object p of P, then the object $R_1\, a$ of B is transformed to the object $S_1\, p$ of Q. In other words, the transformation of B to Q is nested within the transformation of A to P. As before, we shall define the specification of the transformation as a type, outline its proof, and abstract over classes, conditions and holes. The transformation along each rung is similar to the one between A and P earlier. However, the transformation between the verticals (relationships) is new.

The specification of the transformation that is depicted in Figure 3 is formalised as a type as follows (for ease of readability, we write one rung per line).

$$\begin{aligned} &\forall a\colon A\,.\,Pre_A\, a \to \exists p\colon P\,.\,(Post_P\, a\, p \wedge \\ &\forall b\colon B\,.\,Pre_B\, b \wedge (R_1\, a) = b \to \exists q\colon Q\,.\,Post_Q\, b\, q \wedge (S_1\, p) = q)\ . \end{aligned} \tag{6}$$

The transformation along the second rung is nested within the first rung, and has stronger pre and postconditions in virtue of the connectivity constraints placed on objects of B and Q. Note that a third or fourth rung would have the same shape as the second rung.

$$\begin{array}{ccc} x & \overset{f}{\dashrightarrow} & f\,x \\ \downarrow R & Com & \downarrow S \\ R\,x & \overset{f'}{\dashrightarrow} & S\,(f\,x) = f'\,(R\,x) \end{array}$$

Figure 4: The commutative square *Com*.

In outline, the proof of (6) is

$$\dfrac{\begin{array}{c}[a\colon A]\\ [Pre_A\,a]\\ \vdots\ Hole_A\\ Post_P\,a\,\overline{p}\end{array} \qquad \dfrac{\dfrac{\begin{array}{c}[a\colon A]\\ [b\colon B]\\ [Pre_B\,b \wedge R_1\,a = b]\end{array}}{\ulcorner Pre_B\,b \urcorner}\ (\wedge E_1) \quad \dfrac{\begin{array}{c}[a\colon A]\\ [b\colon B]\\ [Pre_B\,b \wedge R_1\,a = b]\end{array}}{\ulcorner R_1\,a = b \urcorner}\ (\wedge E_2)}{\begin{array}{cc} \vdots\ Hole_B & \vdots\ Com_A \\ Post_Q\,b\,\overline{q} & S_1\,\overline{p} = \overline{q} \end{array}}\ (\wedge I)}{\begin{array}{c}(Post_Q\,b\,q \wedge S_1\,\overline{p} = q)\,[\overline{q}/q]\\ \vdots\ Fixed\\ (\forall b\colon B\,.\,Pre_B\,b \wedge R_1\,a = b \rightarrow \ldots)\,[\overline{p}/p]\end{array}}\ (\wedge I)$$

$$\dfrac{}{\begin{array}{c}(Post_P\,a\,p \wedge \forall b\colon B\,.\,Pre_B\,b \wedge R_1\,a = b \rightarrow \ldots)\,[\overline{p}/p]\\ \vdots\ Fixed\\ \forall a\colon A\,.\,Pre_A\,a \rightarrow \exists p\colon P\,.\,Post_P\,a\,p \wedge \ldots\end{array}}$$

The fixed parts of the proof exist in virtue of the structure of the specification, whereas the variable parts exist in virtue of its under-specification. The variable parts are either of the *Hole* kind, i.e. proofs that postconditions are derived from preconditions, or the *Com* kind, i.e. proofs that adjacent rungs are linked.

Quantifying over all variables in (6), including proofs of variable parts, and changing the names of bound variables where appropriate, we obtain the specification of an arbitrary two-runged transformation.

Definition 3 (Two-Runged Model Transformation)**.** *An arbitrary two-runged model transformation is formalised in constructive type theory by the following type:*

$$\begin{array}{c} \forall X\colon Set\,.\,\forall Y\colon Set\,.\,\forall Pre\colon X \rightarrow Prop\,.\,\forall Post\colon X \rightarrow Y \rightarrow Prop\,. \\ \forall f\colon X \rightarrow Y\,.\,\forall Hole\colon (\forall x\colon X\,.\,Pre\,x \rightarrow Post\,x\,(f\,x))\,. \\ \forall X'\colon Set\,.\,\forall Y'\colon Set\,.\,\forall Pre'\colon X' \rightarrow Prop\,.\,\forall Post'\colon X' \rightarrow Y' \rightarrow Prop\,. \\ \forall f'\colon X' \rightarrow Y'\,.\,\forall Hole'\colon (\forall x'\colon X'\,.\,Pre\,x' \rightarrow Post\,x'\,(f'\,x'))\,. \\ \forall R\colon X \rightarrow X'\,.\,\forall S\colon Y \rightarrow Y'\,. \\ \forall Com\colon (\forall x\colon X\,.\,S\,(f\,x) = f'\,(R\,x))\,. \\ \forall x\colon X\,.\,Pre\,x \rightarrow \exists y\colon Y\,.\,Post\,x\,y \wedge \\ \forall x'\colon X'\,.\,Pre'\,x' \wedge R\,x = x' \rightarrow \exists y'\colon Y'\,.\,Post'\,x'\,y' \wedge S\,y = y'\,. \end{array} \tag{7}$$

This formalisation is similar to the one in (3) except that it also quantifies over *Com*, i.e. a proof that starting from every object x of X (X being the source end of the first rung) and navigating to some object y' of Y' (Y' being the target end of the second rung), first via f and S, and then via R and f', the same y' is obtained. The commutative square, after which *Com* is named, is shown in

Fig. 4. Furthermore, given a proof of *Com*, it is a trivial matter to prove that the rungs are linked, i.e.

$$\dfrac{\dfrac{[x\colon X] \quad [Com\colon \forall x\colon X\,.\,S\,(f\,x) = f'\,(R\,x)]}{S\,(f\,x) = f'\,(R\,x)}\ (\forall E) \quad \ulcorner R\,x = x' \urcorner}{S\,(f\,x) = f'\,x'}\ (IE)\ .$$

Property 1. *The function that inhabits* (7) *is*

$$\lambda X\,Y\,Pre\,Post\,f\,Hole\,X'\,Y'\,Pre'\,Post'\,f'\,Hole'\,R\,S\,Com\,x\,h.\langle (f\,x), u\rangle$$

Proof. Direct, using the typing rules given in Section 2. Note that the first element of the output pair is the result of applying the root function f to an arbitrary root object x, reflecting the fact that an ordered transformation is essentially a transformation between root classes. □

We are now ready to formalise the notion of ordered model transformation. First we consider totally ordered transformations, and later we extend the results to partially ordered transformations.

4.1 Totally Ordered Transformations

In order to formalise totally ordered model transformations, we will define a dependent type $T\,X\,Y\,f$, where T is the type name, and X, Y and f are the parameters on which it depends, i.e. root source class, root target class and root function respectively. A totally ordered transformation is an inhabitant of this dependent type. The inhabitants of T are defined inductively, in much the same way as *Nat*, by means of a base rule and a step rule.

Definition 4. *(Type $T\,X\,Y\,f$) The type T is defined inductively, with a rule defining the base case and a rule defining the inductive step, as follows:*

$$\dfrac{\begin{array}{l} X\colon Set \quad Y\colon Set \quad f\colon X \to Y \\ X'\colon Set \quad Y'\colon Set \quad f'\colon X' \to Y' \\ Pre'\colon X' \to Prop \quad Post'\colon X' \to Y' \to Prop \\ Hole'\colon \forall x'\colon X'\,.\,Pre'\,x' \to Post'\,x'\,(f'\,x') \\ R\colon X \to X' \quad S\colon Y \to Y' \quad Com\colon \forall x\colon X\,.\,f'\,(R\,x) = S\,(f\,x) \end{array}}{T_{Base}\,X\,Y\,f\,X'\,Y'\,f'\,Pre'\,Post'\,Hole'\,R\,S\,Com\colon T\,X\,Y\,f}\ (TI_1)$$

$$\dfrac{\begin{array}{l} X\colon Set \quad Y\colon Set \quad f\colon X \to Y \\ X'\colon Set \quad Y'\colon Set \quad f'\colon X' \to Y' \\ Pre'\colon X' \to Prop \quad Post'\colon X' \to Y' \to Prop \\ Hole'\colon \forall x'\colon X'\,.\,Pre'\,x' \to Post'\,x'\,(f'\,x') \\ R\colon X \to X' \quad S\colon Y \to Y' \quad Com\colon \forall x\colon X\,.\,f'\,(R\,x) = S\,(f\,x) \\ t'\colon T\,X'\,Y'\,f' \end{array}}{T_{Step}\,X\,Y\,f\,X'\,Y'\,f'\,Pre'\,Post'\,Hole'\,R\,S\,Com\,t'\colon T\,X\,Y\,f}\ (TI_2)$$

The base rule constructs a transformation of the kind shown in Fig. 5 (left). Note that the root hole and root pre and postconditions (to clarify, the root is at the top) are not part of the construction. The step rule constructs the successor of a transformation t', of the kind shown in Fig. 5 (middle). Again, the root hole and root pre and postconditions are not part of the construction.

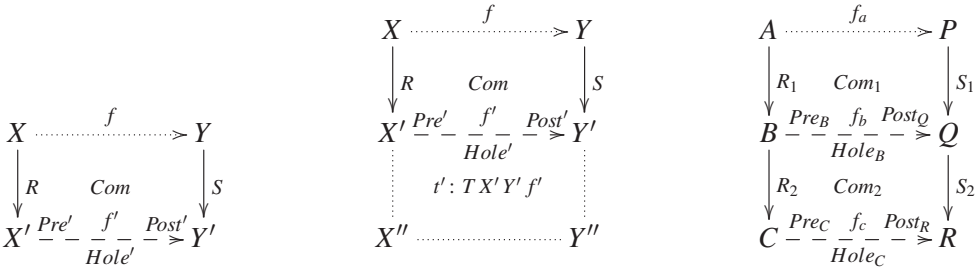

Figure 5: From left to right: representations of the base and step rules of type T, followed by an inhabitant of type $T\,A\,P\,f_a$, namely t_{AP}.

To construct an inhabitant of T, we first apply the base rule and then repeatedly apply the step rule. For example, the transformation t_{AP} in Fig. 5 (right) is constructed as follows:

$$\begin{aligned} t_{BQ} &=_{df} T_{Base}\,B\,Q\,f_b\,C\,R\,f_c\,Pre_C\,Post_R\,Hole_C\,R_2\,S_2\,Com_2\ , \\ t_{AP} &=_{df} T_{Step}\,A\,P\,f_a\,B\,Q\,f_b\,Pre_B\,Post_Q\,Hole_B\,R_1\,S_1\,Com_1\,t_{BQ}\ . \end{aligned}$$

By extension of (6), it would be easy to write down the specification of t_{AP}, including the root transformation we have so assiduously excluded. However, much more useful would be to write down a function that could compute it, not only for t_{AP} but also for every other inhabitant of $T\,X\,Y\,f$ as well. Such a function is given below.

Definition 5. *(Spec)*

$$Spec\colon \forall X\colon Set\,.\,\forall Y\colon Set\,.\,\forall f\colon X\to Y\,.\,T\,X\,Y\,f\to (X\to Y\to Prop)$$

$$\begin{gathered} Spec\,X\,Y\,f\,(T_{Base}\,X\,Y\,f\,X'\,Y'\,f'\,Pre'\,Post'\,Hole'\,R\,S\,Com) =_{df} \\ \lambda x\colon X\,.\,\lambda y\colon Y\,.\,\forall x'\colon X'\,.\,Pre'\,x'\wedge x'=R\,x\to \exists y'\colon Y'\,.\,Post'\,x'\,y'\wedge y'=S\,y \end{gathered}$$

$$\begin{gathered} Spec\,X\,Y\,f\,(T_{Step}\,X\,Y\,f\,X'\,Y'\,f'\,Pre'\,Post'\,Hole'\,R\,S\,Com\,t') =_{df} \\ \lambda x\colon X\,.\,\lambda y\colon Y\,.\,\forall x'\colon X'\,.\,Pre'\,x'\wedge x'=R\,x\to \exists y'\colon Y'\,.\,Post'\,x'\,y'\wedge y'=S\,y\wedge \\ Spec\,X'\,Y'\,f'\,t'\,x'\,y'\ . \end{gathered} \tag{8}$$

Now, the specification of a transformation is an inhabitant of *Prop*. However, *Spec* returns an inhabitant of type $X\to Y\to Prop$. Why? To allow it to be integrated with the root objects of type X and Y, passed down to it by the root transformation.

In its general form, an ordered model transformation is formalised as follows.

Definition 6. *(Ordered Model Transformation) An ordered model transformation is an inhabitant of type*

$$\begin{gathered} \forall X\colon Set\,.\,\forall Y\colon Set\,.\,\forall f\colon X\to Y\,.\,\forall t\colon T\,X\,Y\,f\,. \\ \forall Pre\colon X\to Prop\,.\,\forall Post\colon X\to Y\to Prop\,. \\ \forall Hole\colon \forall x\colon X\,.\,Pre\,x\to Post\,x\,(f\,x)\,. \\ \forall x\colon X\,.\,Pre\,x\to \exists y\colon Y\,.\,Post\,x\,y\wedge Spec\,X\,Y\,f\,t\,x\,y\ . \end{gathered} \tag{9}$$

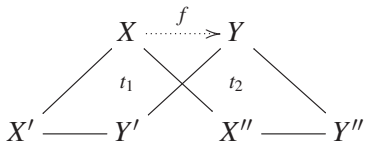

Figure 6: A representation of the join rule of type T.

A proof of (9) (an inhabitant of the type) has the form:

$$\lambda X.\lambda Y.\lambda f.\lambda t.\lambda Pre.\lambda Post.\lambda Hole.\lambda x.\lambda h.\langle (f\,x), u\rangle\ , \tag{10}$$

where u is a proof of

$$Post\,x\,(f\,x) \wedge Spec\,X\,Y\,f\,t\,x\,(f\,x)\ .$$

According to (10), if we could construct an inhabitant of type T from suitable values of $Hole$ (a proof of the root hole), X and Y (the root classes), f (the root function), and Pre and $Post$ (the root conditions), then we could justifiable claim that $\langle (f\,x), u\rangle$ is a certified implementation of the transformation, for an arbitrary source object x. In other words, constructing a suitable value of t is tantamount to proving the specification.

4.2 Partially Ordered Transformations

We generalise the construction to take into account the case where transformations are partially ordered. Without loss of generality, we assume that a partially ordered transformation can be constructed from two ordered transformations, as shown in Fig. 6. Thus, we extend the definition of $T\,X\,Y\,f$ in Definition 4 with a *join* rule, i.e.

$$\frac{t_1 : T\,X\,Y\,f \quad t_2 : T\,X\,Y\,f}{T_{Join}\,t_1\,t_2 : T\,X\,Y\,f}\ (TI_3)\ ,$$

and extend the definition of $Spec$ (Definition 5) with a case for T_{Join}, which returns the conjunction of the specifications of t_1 and t_2, i.e.

$$Spec\ X\,Y\,f\,(T_{Join}\,t_1\,t_2) =_{df} \lambda x : X.\lambda y : Y.Spec\,X\,Y\,f\,t_1\,x\,y \wedge Spec\,X\,Y\,f\,t_2\,x\,y\ .$$

See Fig. 7 for an example.

5 Concrete Example

Consider a transformation between the UML and SQL models in Fig. 8, in which

- each model m is mapped to a schema s of the same name;
- each class c in m is mapped to a table t in s of the same name, and a primary key column in t of the same name;
- each attribute in c is mapped to a non-primary key column in t of the same name;
- the mappings are unconditional, i.e. the preconditions always hold.

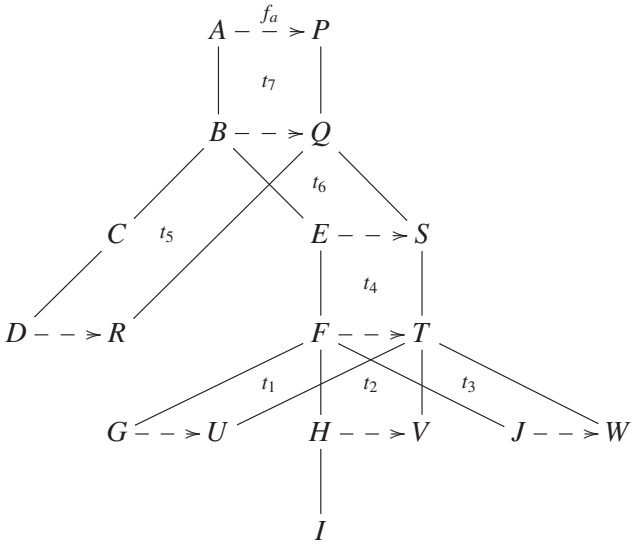

Figure 7: An example of a partially ordered transformation, in which $A, B, \ldots, J$ are the source classes, $P, Q, \ldots, W$ are the target classes, and A is transformed to P, B to Q and so on. The transformation (minus the root artefacts) is given by t_7, where $t_1 =_{df} T_{Base} \ldots$, $t_2 =_{df} T_{Base} \ldots$, $t_3 =_{df} T_{Base} \ldots$, $t_{123} =_{df} T_{Join}\,(T_{Join}\,t_1\,t_2)\,t_3$, $t_4 =_{df} T_{Step} \ldots t_{123}$, $t_5 =_{df} T_{Base} \ldots$, $t_6 =_{df} T_{Step} \ldots t_4$, $t_{56} =_{df} T_{Join}\,t_5\,t_6$, and $t_7 =_{df} T_{Step} \ldots t_{56}$. The specification of the transformation is given by $Spec\,A\,P\,f_a\,t_7$.

The specification of the transformation is given by

$$\forall m\colon Model\,.\,Pre_{Model}\,m \rightarrow \exists s\colon Schema\,.\,Post_{Schema}\,m\,s \wedge Spec\,Model\,Schema\,f_{Model}\,t_{Model-Schema}\,m\,s\ ,$$

where

$$t_{Model-Schema}\colon T\,Model\,Schema\,f_{Model-Schema}\ .$$

If

$$\begin{aligned} m_1 &=_{df} @_{Model}\,1\,[c_1, c_2, c_3] \\ c_1 &=_{df} @_{Class}\,2\,[@_{Attribute}\,5, @_{Attribute}\,6, @_{Attribute}\,7] \\ c_1 &=_{df} @_{Class}\,3\,[@_{Attribute}\,8] \\ c_3 &=_{df} @_{Class}\,4\,[]\ , \end{aligned}$$

i.e. a model with 3 classes and 4 attributes, then

$$K\,Model\,Schema\,f_{Model}\,Pre_{Model}\,Post_{Schema}\,Hole_{Model}\,m_1\,Triv$$

reduces to $\langle s_1, p\rangle$, where s_1 is given by

$$\begin{aligned} s_1 &=_{df} @_{Schema}\,1\,[t_1, t_2, t_3] \\ t_1 &=_{df} @_{Table}\,2\,[@_{Column}\,2\,true, @_{Column}\,5\,false, @_{Column}\,6\,false, @_{Column}\,7\,false,] \\ t_1 &=_{df} @_{Table}\,3\,[@_{Column}\,3\,true, @_{Column}\,8\,false] \\ t_3 &=_{df} @_{Table}\,4\,[@_{Column}\,4\,true]\ , \end{aligned}$$

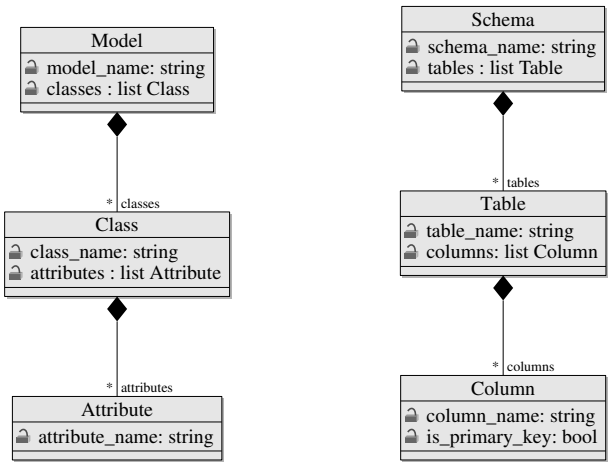

Figure 8: The UML and SQL models.

i.e. a schema with 3 tables and 7 columns (3 of which are primary keys), and p is an unspecified proof (through lack of space) that s_1 is indeed the transform of m_1.

6 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 Abstract State Machines [20]; and Rivera and Vallecillo have exploited the class-based nature of the Maude specification language to formalize metamodels written in KM3 [19]. Further, a related algebraic approach is given by Boronat and Meseguer in [2]. More recently, Calegari et al. [3] proposed a framework for encoding models and metamodels in the Calculus of Inductive Constructions (CIC) [23, 1], and in doing so showed how parts of the ATL model transformation language [10] could be expressed in the CIC, including matched rules, helpers and expressions based on the Object Constraint Language (OCL) [16].

In this paper, we have shown how to assemble the proof of a potentially large ordered model transformation, by decomposing it into a number of smaller proofs which are easier to derive. [4] We focused on a particular kind of transformation with uniform characteristics, which we hope to extend to other kinds of transformations in the future, although we have already incorporated a number of additional variants into the scheme outlined above, including support for many-valued relationships, unmapped source classes (of which C is an example in Fig. 7) and multiple target classes.

In future work, we will extend the techniques to a larger class of model transformations, by abstracting over the non-hierarchical parts of models too. One way of achieving this would be to quantify over arbitrary propositions in each postcondition so that users could include a non-hierarchical proof fragment where necessary. To a certain extent, this is already supported because the *Data* component of a postcondition is user-defined and therefore arbitrary. However, another option would be to add a separate conjunct to the postcondition.

[4] The reader should note that this approach could never be fully automatised in virtue of the unlimited scope of pre and postconditions. However, once the smaller proofs are available, the process of assembling them into a proof of the whole could indeed be automatised.

Clearly, we do not exclude the possibility of an ordered model being embedded within a larger model, like an ordered core with an unordered covering (for example, see Fig. 9). In fact, our experience suggests that the majority of industrial models (which are characterised by their size rather than their complexity) are like this, and that the algorithms which transform them invariably perform preorder traversals of the ordered cores of the source models. That is not say that every model transformation fits this mould. However, it is reasonable to suppose that the lessons learnt from this study may also be applicable to other kinds of model transformations.

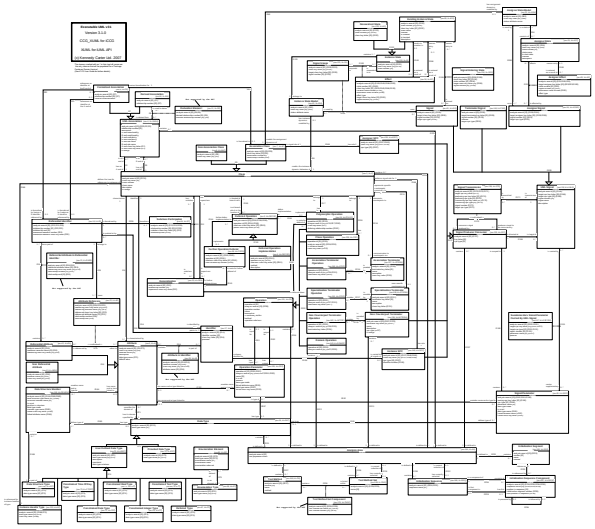

Figure 9: An industrial strength model (of executable UML, courtesy of Abstract Solutions Ltd) comprising a strongly ordered core (admittedly, one defined by generalisation—an area for further study—as well as containment, and visually apparent only to a subject matter expert) surrounded by an unordered covering.

7 Acknowledgements

The authors would like to thank Iman Poernomo for his support in writing this paper.

References

[1] Y. Bertot & P. Castéran (2004): *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science, Springer-Verlag, doi:10.1007/978-3-662-07964-5.

[2] A. Boronat & J. Mesegeur (2008): *An Algebraic Semantics for the MOF*. In J. Fiadeiro & P. Inverardi, editors: *FASE 2008 Proceedings*, *LNCS* 4961, Springer, pp. 377–391, doi:10.1007/978-3-540-78743-3_28.

[3] Daniel Calegari, Carlos Luna, Nora Szasz & Alvaro Tasistro (2011): *A Type-Theoretic Framework for Certified Model Transformations*. In Jim Davies, Leila Silva & Adenilso da Silva Simão, editors: *Formal Methods: Foundations and Applications - 13th Brazilian Symposium on Formal Methods, SBMF 2010, Natal, Brazil, November 8-11, 2010, Revised Selected Papers*, *Lecture Notes in Computer Science* 6527, Springer, pp. 112–127, doi:10.1007/978-3-642-19829-8_8.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer & C. Talcott (2003): *The Maude 2.0 System*. In: *Rewriting Techniques and Applications (RTA 2003)*, *Lecture Notes in Computer Science* 2706, Springer-Verlag, pp. 76–87, doi:10.1007/3-540-44881-0_7.

[5] H.B. Curry & R. Feys (1958): *Combinatory Logic*. 1, North Holland Publishing Company.

[6] Object Management Group (2003): *MDA Guide Version 1.0.1*.

[7] Object Management Group (2009): *Unified Modelling Language, Version 2.2*. Formal/09-02-02, formal/09-02-04.

[8] Object Management Group (2011): *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Formal/2011-01-01.

[9] W.A. Howard (1980): *The Formulae-as-Types Notion of Construction*. In J.P. Seldin & J.R. Hindley, editors: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press Inc.

[10] F. Jouault & I. Kurtev (2006): *Transforming Models with ATL*. *LNCS* 3844, MODELS 2005 Workshops, Springer-Verlag, pp. 128–138, doi:10.1007/11663430_14.

[11] K. Lano (2009): *Model-Driven Software Development with UML and JAVA*. Cengage Learning EMEA.

[12] K. Lano, editor (2009): *UML 2 Semantics and Applications*. Wiley, doi:10.1002/9780470522622.

[13] P. Martin-Löf (1984): *Intuitionistic Type Theory*. Bibliopolis.

[14] S.J. Mellor, K. Scott, A. Uhl & D. Weise (2004): *MDA Distilled*. Addison-Wesley.

[15] Object Management Group (2006): *Meta Object Facility (MOF) Core Specification*. Formal/06-01-01.

[16] Object Management Group (2012): *Object Constraint Language (OCL), Version 2.3.1*. Formal/2012-01-01.

[17] I.H. Poernomo (2008): *Proofs-as-Model-Transformations*. In Antonio Vallecillo, Jeff Gray & Alfonso Pierantonio, editors: *Theory and Practice of Model Transformations, First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008, Proceedings*, *Lecture Notes in Computer Science* 5063, Springer, pp. 214–228, doi:10.1007/978-3-540-69927-9_15.

[18] I.H. Poernomo & J.W. Terrell (2010): *Correct-by-Construction Model Transformations from Partially Ordered Specifications*. In J.S. Dong & H. Zhu, editors: *Formal Methods and Software Engineering*, *Lecture Notes in Computer Science* 6447, 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, Springer, doi:10.1007/978-3-642-16901-4_6.

[19] J. Rivera & A. Vallecillo (2007): *Adding Behavioural Semantics to Models*. In: *The 11th IEEE International EDOC Conference (EDOC 2007)*, IEEE Computer Society, Annapolis, Maryland, USA, pp. 169–180.

[20] D. Ruscio, F. Jouault, I. Kurtev, J. Beźivin & A. Pierantonio (2006): *Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs*. Technical Report 06.02, Laboratoire d'Informatique de Nantes-Atlantique (LINA), Nantes, France.

[21] S. Shlaer & S.J. Mellor (1988): *Object-Oriented Systems Analysis – Modelling the World in Data*. Yourdon Press, Prentice Hall.

[22] S. Shlaer & S.J. Mellor (1991): *Object Lifecycles – Modelling the World in States*. Yourdon Press, Prentice Hall.

[23] The Coq Development Team (2010): *The Coq Proof Assistant, Reference Manual*. Available at `http://coq.inria.fr/refman`.

[24] Xavier Thirioux, Benoît Combemale, Xavier Crégut & Pierre-Loïc Garoche (2007): *A Framework to Formalise the MDE Foundations*. In: *TOWERS*, pp. 14–30.

[25] S. Thompson (1999): *Type Theory & Functional Programming*. Computing Laboratory, University of Kent.