Towards Modular Reasoning for Model Transformations

Steffen Zschaler[1], Jeffrey Terrell[1], and Iman Poernomo[2]

[1] King's College London, Department of Informatics, London, UK
{steffen.zschaler|jeffrey.terrell}@kcl.ac.uk
[2] J. P. Morgan, London, UK
iman.h.poernomo@jpmorgan.com

Abstract. Model transformations have been studied for some time, typically using a semantics based on graph transformations. This has been very successful in defining, optimising and executing model transformations, but has been less useful for providing a firm semantic basis for modular, reusable transformations. We propose a novel rendering of transformation semantics in terms of constructive type theory and explore how this may be employed for expressing dependencies and guarantees of transformation modules in a formal framework. This is a position paper laying out research challenges to be overcome before we will be able to successfully modularise verifiable transformations.

1 Introduction

In model-driven development (MDD) [1], model transformations are used to transform models into other models, for analysis, abstraction, refinement, and eventually code generation. The development of model transformations is supported by dedicated transformation languages, such as Kermeta [2], ATL [3], QVT [4], or ETL [5]. The latter, rule-based and (largely) declarative languages have helped raise the level of abstraction at which transformations can be developed: They enable developers to say *what* the result of a transformation should be without needing to say precisely *how* it will be obtained.

As transformations get bigger and more complex, they become more difficult to develop, reuse, evolve, and maintain. As a consequence, there is a need for systematic software engineering to be applied to the development of model transformations themselves. In this paper, we explore the modularisation of model transformations, and the modular verification of model transformations using constructive type theory.

This work is an extension of previous work presented in [6, 7]. It extends this work by exploring different types of contracts required between transformation modules depending on the type of verification intended.

2 Modularisation of Model Transformations

The problems inherent in "normal" software development have been addressed by decomposing software systems into modules that can be developed, reused, evolved, and

maintained independently as long as clearly defined interfaces between the modules are maintained [8]. Some benefits of modularisation include:

- The ability to distribute development work over several sub-teams each responsible for one module and each able to work relatively independently of the others;
- The ability to reuse modules in new contexts as long as their assumptions are satisfied by the new context; and
- The ability to exchange a module for a different one, as long as the new module maintains the same interface as the original module.

All of these benefits translate to the development of model transformations. To illustrate the last item, consider a transformation from class diagrams to relational database schemata. There are a number of different concerns:

1. *Base.* The transformation needs to ensure that there is a table in the database schema for each class, and a column in that table for each attribute in the class.
2. *Associations.* The database schema needs to provide a way of representing associations between classes.
3. *Inheritance.* The database schema needs to provide a way of representing inheritance relationships between classes.

Clearly, each of these concerns can be realised in a number of different ways by a transformation specification, and each realisation choice has a different effect on the performance and reliability of the resulting database, so in different circumstances we may wish to make different choices for each concern. We can only do so safely and efficiently if the concern implementations have been packaged into modules whose interfaces hide the details of the realisation choices.

There has been considerable research interest in modularising model transformations for some time. The approaches proposed and studied so far, may be characterised by the granularity of modules that they provide:

- *External Composition* takes entire model transformations to be modules that can be independently reused and composed. Early research focused mainly on languages and tools for describing and executing such compositions of reusable model transformations. This has led to early work on MDA components [9], megamodelling [10], transformation chaining [11–13], and transformation configuration [14]. A key discussion in this field is how the interface of a transformation should be characterised. For example, initial work on external composition [15, 16] defined transformation signatures by two sets of metamodels: one typing the source models and the other typing the target models.
- *Internal Composition* considers modules of a finer granularity:
  - *Inter-rule Composition* techniques consider transformation rules to be the unit of modularity. A number of mechanisms are provided for composing rules into transformations, including implicit and explicit rule invocation, and rule inheritance [11, 17]. Approaches inspired from graph transformation—for example, VMT [18]—even allow for chaining of individual transformation rules.

- *Intra-rule Composition.* Some evaluations have shown that there are scenarios where modularisation at the level of complete rules is inadequate [17, 19, 20], and a number of mechanisms have been proposed that allow parts of rules to become units of modularity. Balogh and Varró [21] describe how matching and creation patterns can be defined as standalone units of modularity, and composed into more complex patterns for use in transformation rules.

3 Formalisation and Verification of Model Transformations

Before we can discuss formalisation and verification of model transformations in detail, we need to introduce a few key terms. These are particularly relevant to declarative, rule-based transformations. Such transformations are described using a *transformation specification,* which consists of a number of rules defining how specific model elements of a source model should be mapped onto elements of a target model. Transformation specifications are expressed using a *transformation language*—for example, ATL [3], QVT [4], or ETL [5]. To execute these transformations, an operational *transformation implementation* must be generated. This can happen either implicitly through interpretation of the transformation specification by a tool (e.g., ATL or ETL) or explicitly by programming a function that maps source models onto target models (e.g., QVT).

From these concepts, we can identify two types of transformation verification:

1. *Verification of Implementations.* Given a specification and an implementation we need to be able to verify that the implementation does indeed satisfy the specification.
2. *Verification of Specifications.* Given a specification, we need to be able to verify that it does what we expect it to do. One way to do so is by defining higher-level properties over source and target models and their instances that a model transformation needs to maintain and to prove that a particular transformation specification does indeed maintain these properties. Note that these properties are, in effect, an underspecified version of the transformation specification. As has been remarked in [22], verification of specifications can be further divided. In extension to [22], we distinguish three categories:
   (a) *Verification of Syntactic Correctness.* Here, we aim to verify that a given transformation specification will only produce syntactically correct and well-formed target models, provided it is given syntactically correct and well-formed source models.
   (b) *Verification of General Semantic Properties.* Here, we aim to verify general properties of a transformation specification that go beyond syntactic correctness, but are independent of the semantics of the specific models being transformed. —for example, termination or confluence. (where the implementation semantics allows indeterminism).
   (c) *Verification of Semantic Correctness.* Here, we aim to verify that a given transformation specification will maintain some semantic properties of the source models in the target models produced. Figure 1 a) shows a commuting diagram illustrating this for the case of a transformation τ from a source model M_1 expressed in language $\mathcal{L}_1$ to a target model M_2 expressed in language $\mathcal{L}_2$.

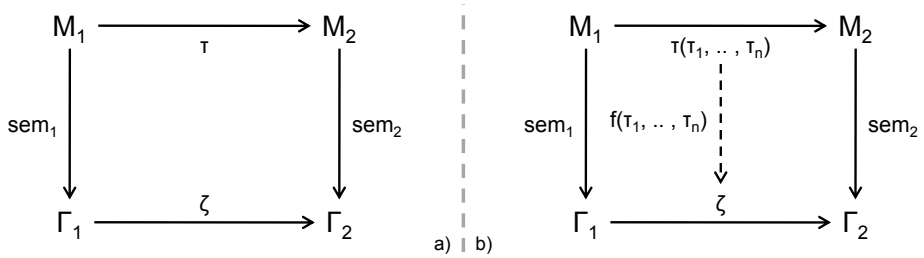

Fig. 1. Commuting diagrams for verification of semantic correctness: a) monolithic reasoning (based on [22]), b) modular reasoning

sem_1 indicates the relation between M_1 and its semantics Γ_1. sem_2 indicates the relation between M_2 and its semantics Γ_2. ζ represents the semantic property to be maintained. This basic schema underlies current approaches to the verification of transformations like [23].

Of course any such verification requires a formal representation of transformation specifications and their implementations. We have previously shown in [24] how transformation specifications can be expressed using constructive type theory (CTT) and how this can be used to verify model transformation implementations. Ongoing work first reported in [25] studies how proofs of complex transformation specifications can be provided efficiently using knowledge of the general structure of transformation specifications.

4 Modular Verification of Model Transformations

In our previous work [24, 25], we used so-called $\forall\exists$ formulas for expressing transformation specifications, i.e.

$$\forall s\colon Source\,.\,Pre\;s \rightarrow \exists t\colon Target\,.\,Post\;s\;t\;,$$

where the notation $a\colon A$ indicates that a is of type A, $Source$ and $Target$ are elements of the source and target metamodels respectively, Pre is the pre-condition that must be satisfied for the transformation to be applied, and $Post$ is the post-condition that the transformation will establish (should it be applied) between source and target model elements. An inhabitant of such a type is a function that takes a source model (of type $Source$) that satisfies Pre, and returns a target model (of type $Target$) that satisfies $Post$.

To understand what is needed to modularise such specifications, let us consider specifications in which $Post$ is the conjunction of a base predicate $Base$ and a predicate $Concern$ describing a partial transformation addressing a specific concern. To find an inhabitant of any particular transformation, we would repeatedly decompose its

specification into smaller and smaller sub-specifications, until a set of primitive sub-specifications were obtained. We would then unfurl the decomposition, and mechanically construct its inhabitant.[3] Hence, schematically, an inhabitant of the transformation specification would be constructed as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots \qquad \vdots}{f\colon Base\ s\ \bar{t} \quad g\colon Concern\ s\ \bar{t}}}{\langle f,\ g\rangle\colon Base\ s\ \bar{t} \wedge Concern\ s\ \bar{t}}}{\langle \bar{t},\ \langle f,\ g\rangle\rangle\colon \exists t\colon Target\,.\, Base\ s\ t \wedge Concern\ s\ t}}{\lambda h\,.\,\langle \bar{t},\ \langle f,\ g\rangle\rangle\colon Pre\ s \to \exists t\colon Target\,.\, Base\ s\ t \wedge Concern\ s\ t}}{\lambda s\,.\,\lambda h\,.\,\langle \bar{t},\ \langle f,\ g\rangle\rangle\colon \forall s\colon Source\,.\, Pre\ s \to \exists t\colon Target\,.\, Base\ s\ t \wedge Concern\ s\ t}\ .$$

We have assumed that f and g are known inhabitants of $Base\ s\ \bar{t}$ and $Concern\ s\ \bar{t}$, respectively[4], to enable the final inhabitant, namely

$$\lambda s\,.\,\lambda h\,.\,\langle \bar{t},\ \langle f,\ g\rangle\rangle\ ,$$

to be constructed. This is a function that takes an instance s of $Source$, and a proof h that s satisfies $Pre\ s$, and returns an instance $\bar{t}$ of $Target$ (dependent on s), and proofs that s and $\bar{t}$ both satisfy $Base\ s\ \bar{t}$ and $Concern\ s\ \bar{t}$.

To allow composing different realisations of the $Concern$ transformation, we can parametrise over it. As a first attempt, we might write

$$\begin{aligned}&\forall Concern\colon Source \to Target \to Prop\,.\\ &\quad \forall s\colon Source\,.\, Pre\ s \to \exists t\colon Target\,.\, Base\ s\ t \wedge Concern\ s\ t\ .\end{aligned}$$

However, this would not be enough to deduce an inhabitant of $Concern\ s\ \bar{t}$, which is what the proof tree above suggests we need. As a second attempt, we might additionally quantify over a higher order predicate, and write

$$\begin{aligned}&\forall Concern\colon Source \to Target \to Prop\,.\\ &\quad \forall Proof\colon (\forall s\colon Source\,.\, Pre\ s \to \exists t\colon Target\,.\, Concern\ s\ t)\,.\\ &\qquad \forall s\colon Source\,.\, Pre\ s \to \exists t\colon Target\,.\, Base\ s\ t \wedge Concern\ s\ t\ ,\end{aligned}$$

to allow an independent proof of $Concern$ to be passed in as a parameter. This effectively allows us to indicate what part $Concern$ plays in the proof of the composed specification. However, the specific proof parameter given above does not yet allow us to prove the transformation, because we cannot guarantee that the witness of $Concern\ s\ t$ in

$$\exists t\colon Target\,.\, Concern\ s\ t\ ,$$

would be the same as the one required by $Base\ s\ t$ in

$$\exists t\colon Target\,.\, Base\ s\ t \wedge Concern\ s\ t\ .$$

[3] This is made possible by the Curry-Howard Isomorphism – an expression of the close relationship that exists between objects and the types they inhabit.

[4] In reality, they would most probably require further decomposition.

In other words, the specification above allows $Concern$ transformations to break the choices made by the $Base$ transformation.

Depending on the specific form of $Base$ and $Concern$ as well as the type of verification we are interested in, we will require different proof parameters. For specific instantiations of $Base$ and $Concern$ we can provide specific proof terms (e.g., we have defined proof terms for an example of separating structural and data transformations in Sect. 5). The research challenge we are facing currently is how to characterise the proof obligations in general depending on characteristics of the $Base$ and $Concern$ predicates and the type of verification we are interested in. Below, we elaborate on our current observations.

Generally, there are at least three kinds of contracts we can express using a proof parameter:

1. *Statements about what $Concern$ must not constrain.* These are expressed by conjoining predicates that equate (part of) a parameter to $Concern$ to something that is all-quantified in the proof term.
2. *Statements about what $Concern$ must guarantee.* These are expressed by conjoining predicates that only constrain parameters to $Concern$.
3. *Statements about what $Concern$ can rely on.* These are prepended to the proof term as the antecedent of an implication and may express a bi-directional dependency between the two transformations.

Further, depending on the type of verification, we make the following observations about what needs to be in the proof parameters. Further research is required to provide stronger support for these.

1. *Verification of Implementations.* Here, we are mainly concerned with ensuring that an inhabitant of the composed type can be found for any legal parameters. The proof term, therefore, must mainly ensure that the composition of $Base\, s\, t \wedge Concern\, s\, t$ remains consistent. This can most easily be ensured by requiring some kind of non-overlap between what the two predicates constrain, but there may be weaker options as well.
2. *Verification of Specifications.* Generally speaking, in addition to ensuring consistency, the proof parameter must ensure that the respective high-level properties can be proved. For syntactic correctness, it will therefore mainly need to constrain multiplicities of links between model elements and ensure other well-formedness rules are not violated. For the case of general semantic properties it is not immediately clear if and how these can be treated by modular reasoning. For the case of semantic correctness, the proof parameter needs to break down the overall semantic property into appropriate sub-constraints for the transformation parameters. Moreover, it is likely that ζ itself will depend on the specific choice of transformation parameters (cf. Fig. 1 b)). To explain, consider the example transformation from class diagrams to relational models from Sect. 2: We required that for every record `recordB` that represents an instance of class `B` there needs to be a linked record `recordA` representing an instance of class `A` if `B` is a sub-class of `A`. The notions of a record *representing* a class and of two records being *linked* are still quite abstract in this

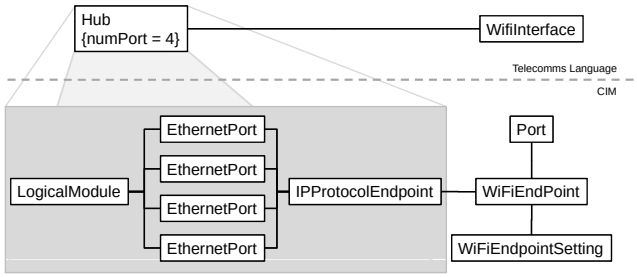

Fig. 2. An abstract model as required for telecommunications configuration (above) and an equivalent representation as a CIM model (below) [28]

formulation. What they mean precisely depends on the specific partial transformations that are chosen to realise the *Base* and *Inheritance* concerns. Consequently, we now also require a parametrised representation of the semantic property to be maintained and every partial transformation must provide formulas that can replace the formal parameters to construct an unparametrised ζ for a specific composition of transformations.

5 Example

We reuse an example from [26]. The example was drawn from the domain of telecommunications; specifically from models of communications infrastructures expressed in the Common Information Model (CIM) industry standard [27]. CIM is a very large language. For many purposes, a much smaller language could be used. Transformations are then needed to translate such more compact models into full CIM models. Figure 2 is an example of the type of transformations required. Specifically, it shows how the notion of a hub can be rendered into a corresponding CIM model, and how a model representing a hub with four ports, which are connected to a WiFi network, can be transformed into its more detailed counterpart in CIM. It should be noted how the abstract hub object is expanded into a model where each of the ports is explicitly reified as a separate object. The number of hub objects is determined by the value of the `numPort` attribute of the abstract hub object. In expressing this transformation, two concerns need to be considered: 1) the definition of the target structure (i.e., the number of and connections between port objects) and 2) the configuration of each object's data attributes. It would be useful to be able to separate these concerns in the definition of a model transformation, so that they can be independently understood, maintained, and evolved.

In this section, we illustrate the ideas of the previous section by discussing a composed transformation based on the example of Fig. 2, in which A is short for Hub, P for LogicalModule, Q for IPProtocolEndpoint, and R for EthernetPort.

Consider a transformation between a source model A and a target model PQR (see Fig. 3), in which each object $a\colon A$ is transformed into three things: an object $p\colon P$, an

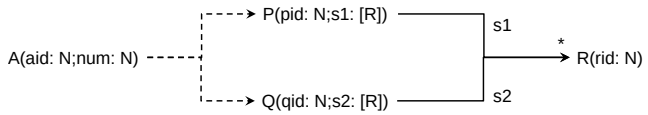

Fig. 3. The transformation $A \mapsto PQR$. The dotted lines depict inter-model mappings whereas the solid lines depict intra-model relationships

object $q\colon Q$, and a possibly empty list of objects $l\colon [R]$. Furthermore, suppose that the multiplicities at the directed ends of relationships s_1 and s_2 are derived from attribute num. For example, if, for a particular $a\colon A$, the value of $a.num$ were 3, then a would be transformed into five objects, three of class R and one each of classes P and Q.

If we assume that for every $a\colon A$, there is without pre-condition some $p\colon P$, $q\colon Q$ and list of $r\colon R$ with the same id as a, then the specification of the transformation $A \mapsto PQR$ is given by

$$\forall a\colon A \,.\, \exists p\colon P \,.\, \exists q\colon Q \,.\, \exists l\colon [R] \,.\, Post(a, p, q, l) \;, \tag{1}$$

where

$$\begin{aligned} Post(a, p, q, l) \;=_{df}\; & (a.aid = p.pid) \wedge (a.aid = q.qid) \wedge \\ & \forall r\colon R \,.\, r \in p.s_1 \rightarrow (a.aid = r.rid) \wedge \\ & (l.length = a.num) \wedge (p.s_1 = l) \wedge (q.s_2 = l) \;. \end{aligned}$$

Using type theory, we can derive a certified program K that implements the transformation, namely

$$K \;=_{df}\; \lambda a \,.\, \langle f_P, \langle f_Q, \langle f_{[R]}, \ldots \rangle \rangle \rangle \;,$$

where

$$\begin{aligned} f_P \;&=_{df}\; Build_P(a.aid, f_{[R]}) \\ f_Q \;&=_{df}\; Build_Q(a.aid, f_{[R]}) \\ f_{[R]} \;&=_{df}\; Build_{[R]}(a.num, a.aid) \;, \end{aligned}$$

and where $Build_{[R]}$ is a recursive function that constructs a list of objects of R. If we apply the program to a particular $a_1\colon A$, where $a_1 \;=_{df}\; \{aid = 1, num = 3\}$, i.e. a Hub with three ports, we obtain

$$K\; a_1 \twoheadrightarrow \langle \{pid = 1, s_1 = l\}, \langle \{qid = 1, s_2 = l\}, \langle l, \ldots \rangle \rangle \rangle \;,$$

where $l \;=_{df}\; \{rid = 1\} :: \{rid = 1\} :: \{rid = 1\} :: [\,]$, i.e. a LogicalModule and an IPProtocolEndpoint both linked to the same three instances of EthernetPort.

If we separate the data component of this transformation (what, in the previous section, we called $Base$), i.e.

$$(a.aid = p.pid) \wedge (a.aid = q.qid) \;\wedge \forall r\colon R \,.\, r \in p.s_1 \rightarrow (a.aid = r.rid) \;,$$

from the structural component (what, in the previous section, we called $Concern$), i.e.

$$(l.length = a.num) \wedge (p.s_1 = l) \wedge (q.s_2 = l) \ ,$$

and rewrite the transformation in a form that allows an arbitrary structural component $strucT$ to be passed into the specification as a parameter, we obtain

$$dataT \ : \ \left\{ \begin{array}{l} \forall StrucR : A \to P \to Q \to Prop. \\ \quad \forall strucT : \begin{array}{l} \forall a : A. \\ \forall pid : nat. \\ \forall qid : nat. \\ \forall rid : nat. \\ \exists p : P. \\ \exists q : Q. \end{array} \left(\begin{array}{l} StrucR(a,p,q) \wedge \\ p.pid = pid \wedge \\ q.qid = qid \wedge \\ p.s_1 = q.s_2 \wedge \\ \forall r : R.r \in p.s_1 \to r.rid = rid) \end{array} \right) \\ \quad \to \forall a : A. \exists p : P. \exists q : Q. \left(\begin{array}{l} StrucR(a,p,q) \wedge \\ p.pid = a.aid \wedge \\ q.qid = a.aid \wedge \\ \forall r : R.r \in p.s_1 \to r.rid = a.aid \end{array} \right) \end{array} \right\} .$$

The data manipulation aspect of the specification maps Hub ids to LogicalModule and IPProtocolEndpoint ids. However, the transformation is parametrized over the predicate variable $StrucR$, typed to specify an *arbitrary* graph relationship between Hubs A, LogicalModules P and IPProtocolEndpoints Q. It *requires* a structural transformation to instantiate the proof term $strucT$, guaranteeing $StructR(a,p,q)$ holds between input Hubs a, and output LogicalModules and IPProtocolEndpoints p and q. The orthogonality of this transformation with respect to data is stipulated by the addtional conjuncts of the type for $strucT$. In particular, $strucT$ will produce the required structural transformation for *any* possible values of $p.pid$ and $q.qid$: this ensures that the particular values determined by the data transformation will not be inconsistent with its composition with an instantiating $strucT$.

So far, we have shown a way of using higher-order parametrisation over propositions representing transformation specifications and proofs of these propositions to express dependencies between transformation modules. Next, we present three ways of using such modular transformations:

1. We show how we can recompose a transformation so modularised and that this results in a transformation specification that can be proven to be equivalent to the monolithic specification.
2. We show that modularising the transformation indeed allows us to reuse transformations in new contexts by composing them with different transformation modules.
3. We show that the explicitly expressed dependencies can be enforced, so that we cannot compose transformation modules that do not satisfy these dependencies.

Complete Coq scripts for all proofs are available on-line.[5]

[5] `http://www.steffen-zschaler.de/publications/kcl_workshop_11/`

5.1 Recomposing Transformation Modules

We have seen in the previous section how we can define the transformation orthogonally from a data transformation $dataT$, taking a predicate variable $StrucR$ parametrizing the structural transformation requirements. An example of such a structural requirement could be the following:

$$Link1(a : A, p : P, q : Q) =_{df} p.s_1 = q.s_2 \wedge p.s_1.length = a.num$$

It is possible to derive a proof $ProofLink1$ that shows the structural transformation does indeed satisfy the dependency constraints of $Data$:

$$\begin{aligned} ProofLink1 : \forall a : A.\forall pid : nat.\forall qid : nat.\forall rid : nat.\exists p : P.\exists q : Q. \\ Link1(a, p, q) \wedge p.pid = pid \wedge q.qid = qid \wedge p.s_1 = q.s_2 \wedge \\ \forall r : R, r \in p.s_1 \rightarrow r.rid = rid \end{aligned}$$

Proving this theorem is straightforward. Once we have this proof, we can produce the composed transformation

$$dataT\ Link1\ ProofLink1$$

guaranteed to immediately meet the composite specification

$$\begin{aligned} \forall a : A.\exists p : P.\exists q : Q.p.s_1 = q.s_2 \wedge p.s_1.length = a.num \wedge \\ p.pid = a.aid \wedge q.qid = a.aid \wedge \\ (\forall r : R, r \in p.s_1 \rightarrow r.rid = a.aid) \end{aligned}$$

It can be seen that this is equivalent to the monolithic specification from (1). The only substantive difference in the two formulae is that (1) uses a separate variable to refer to the list of Rs, but it is easy to prove that the overall specification is equivalent to $Composed$ above.

This section has shown that using modularised transformation specifications we can still express the same transformations; that is, we do not lose expressivity. The next section discusses what we gain from modularising transformations in this way.

5.2 Reusing Transformation Modules

Having modularised our transformation specification, we can now reuse individual modules to produce different composed transformations. For example, $Link1$ is a structural transformation predicate that used the num attribute of an element $a : A$ to determine the number of elements to produce in $p.s_1$. Instead, we could also use a structural transformation that simply produces a hard-coded number of elements:

$$Link2(a : A, p : P, q : Q) =_{df} p.s_1 = q.s_2 \wedge p.s_1.length = 1$$

We can provide a proof term $ProofLink2$ similar to $ProofLink1$ above that states that $Link2$ satisfies the dependency constraints of $dataT$. Because we can prove this theorem, we can then use this proof to construct a new composed transformation producing this changed structure:

$$dataT\ Link2\ ProofLink2$$

5.3 Enforcing Transformation-Module Contracts

Finally, let us consider the following alternative structural transformation predicate:

$$Link3(a : A, p : P, q : Q) =_{df} p.s_1 = q.s_2 \wedge p.s_1.length = a.num \wedge p.pid = 7$$

This is the same as $Link1$ except that it additionally sets $p.pid$ to 7. Can we compose this with the parametrized data transformation $dataT$ as well? To be able to do so, we need to provide a proof of the following theorem:

$$\begin{aligned} ProofLink3 : \forall a : A.\forall pid : nat.\forall qid : nat.\forall rid : nat.\exists p : P.\exists q : Q. \\ Link3(a, p, q) \wedge p.pid = pid \wedge q.qid = qid \wedge p.s_1 = q.s_2 \wedge \\ \forall r : R.r \in p.s_1 \rightarrow r.rid = rid \end{aligned}$$

It quickly becomes clear that this cannot be done: $ProofLink3$ is required to hold for arbitrary values of $p.pid$, but at the same time $Link3$ states that $p.pid$ should be 7. Because to construct a composed transformation we require a proof for $ProofLink3$, we cannot compose this structural transformation with the data transformation. In this way, the type specification accompanying the proof parameter constrains what compositions are permissible, effectively providing a semantic interface to the requirement.

6 Conclusions and Future Work

As model transformations become more important to software development, systematic development of these transformations becomes itself more important. We have shown our current ideas on how constructive type theory may be used to formally express the interfaces of transformation components. Centrally, higher-order type theory allows us to quantify (i.e., parametrise) over predicates and proofs of these predicates, enabling a transformation module to express precisely what it expects of other transformation modules with which it can be composed.

In the present paper, we have only sketched out the essential idea behind our approach to the modularisation of model transformations. We are currently working to apply this idea to examples of more complex modular transformations and hope to report on this in a further publication.

References

1. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. IEEE Software **20**(5) (September 2003) 42–45
2. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In Briand, L., Williams, C., eds.: Model Driven Engineering Languages and Systems. Volume 3713 of LNCS., Springer Berlin / Heidelberg (2005) 264–278
3. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P.: ATL: a QVT-like transformation language. In: OOPSLA Companion. (2006) 719–720
4. Object Management Group: Query / view / transformations (QVT). OMG Document pct/05-11-01 (2005)

5. Kolovos, D.S., Paige, R.F., Polack, F.A.: The Epsilon transformation language. [29]
6. Zschaler, S., Poernomo, I., Terrell, J.: Towards using constructive type theory for verifiable modular transformations. In: Proc. 1st Workshop on Free Composition (FREECO'11, short paper). (2011)
7. Terrell, J., Zschaler, S., Poernomo, I.: Proof-carrying model-transformation components. Technical Report TR-11-02, King's College London, Department of Informatics (2011)
8. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM **15**(12) (December 1972) 1053–1058
9. Bézivin, J., Gérard, S., Muller, P.A., Rioux, L.: MDA components: Challenges and opportunities. In Evans, A., Sammut, P., Willans, J.S., eds.: Proc. 1st Int'l Workshop Metamodelling for MDA, York, UK (2003) 23–41
10. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the large and modeling in the small. In Aßmann, U., Aksit, M., Rensink, A., eds.: Proc. MDAFA 2003/04. Volume 3599 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2005) 33–46
11. Belaunde, M.: Transformation compositon in QVT. [30] 39–45
12. Chenouard, R., Jouault, F.: Automatically discovering hidden transformation chaining constraints. [31] 92–106
13. Vanhooff, B., Ayed, D., Baelen, S.V., Joosen, W., Berbers, Y.: UniTI: A unified transformation infrastructure. In Engels, G., Opdyke, B., Schmidt, D.C., Weil, F., eds.: Proc. 10th Int'l Conf. Model Driven Engineering Languages and Systems (MoDELS'07). Volume 4735 of Lecture Notes in Computer Science., Springer (2007) 31–45
14. Wagelaar, D., Straeten, R.V.D.: A comparison of configuration techniques for model transformations. In Rensink, A., Warmer, J., eds.: Proc. ECMDA-FA 2006. Volume 4066 of LNCS., Springer (2006) 331–345
15. Marvie, R.: A transformation composition framework for model driven engineering. Technical report, University of Lille 1 (2004) LIFL technical report 2004-n10.
16. Vignaga, A., Jouault, F., Bastarrica, M.C., Brunelière, H.: Typing in model management. In Paige, R., ed.: Proc. 2nd Int'l Conf. on Theory and Practice of Model Transformations (ICMT'09). Volume 5563 of Lecture Notes in Computer Science., Springer-Verlag (2009) 197–212
17. Kurtev, I., van den Berg, K., Jouault, F.: Evaluation of rule-based modularization in model transformation languages illustrated with ATL. In: Proc. 21st Annual ACM Symposium on Applied Computing (SAC'06'). (April 2006) 1202–1209
18. Sendall, S., Perrouin, G., Guelfi, N., Biberstein, O.: Supporting model-to-model transformations: The VMT approach. In Rensink, A., ed.: Proc. MDAFA 2003. (2003) Published as CTIT Technical Report TR–CTIT–03–27, University of Twente.
19. Cleenewerck, T., Kurtev, I.: Separation of concerns in translational semantics for DSLs in model engineering. In: ACM Symposium on Applied Computing. (2007) 985–992
20. Goknil, A., Topaloglu, N.Y.: Composing transformation operations based on complex source pattern definitions. [30] 27–32
21. Balogh, A., Varró, D.: Pattern composition in graph transformation rules. [30] 33–37
22. Lano, K., Clark, D.: Model transformation specification and verification. In: Proc. 8th Int'l. Conf. Quality Software (QSIC '08). (August 2008) 45–54
23. Baresi, L., Ehrig, K., Heckel, R.: Verification of model transformations: A case study with BPEL. In Montanari, U., Sannella, D., Bruni, R., eds.: Proc 2nd Symposium on Trustworthy Global Computing (TGC'06). Volume 4661 of Lecture Notes in Computer Science., Springer (2007) 183–199
24. Poernomo, I.: Proofs-as-model-transformations. [29] 214–228
25. Poernomo, I., Terrell, J.: Correct-by-construction model transformations from partially ordered specifications in Coq. In Dong, J.S., Zhu, H., eds.: Proc. 12th Int'l Conf. Formal Meth-

ods and Software Engineering (ICFEM 2010). Volume 6447 of Lecture Notes in Computer Science., Springer (2010) 56–73
26. Johannes, J., Zschaler, S., Fernández, M.A., Castillo, A., Kolovos, D.S., Paige, R.F.: Abstracting complex languages through transformation and composition. [31] 546–550
27. Distributed Management Task Force Inc. (DMTF): Common information model standards. http://www.dmtf.org/standards/cim/ (2008) Last visited 28/10/2008.
28. Johannes, J., Zschaler, S., Fernández, M.A., Castillo, A., Kolovos, D.S., Paige, R.F.: Abstracting complex languages through transformation and composition. Technical Report TUD-FI09-08 July 2009, Technische Universität Dresden (2009)
29. Vallecillo, A., Gray, J., Pierantonio, A., eds.: Proc. 1st Int'l. Conf. on Theory and Practice of Model Transformations (ICMT'08). In Vallecillo, A., Gray, J., Pierantonio, A., eds.: Proc. 1st Int'l. Conf. on Theory and Practice of Model Transformations (ICMT'08). Volume 5063 of Lecture Notes in Computer Science., Springer-Verlag (July 2008)
30. Kleppe, A.G.: 1st European workshop on composition of model transformations (CMT'06). Technical Report TR-CTIT-06-34, Centre for Telematics and Information Technology, University of Twente (June 2006)
31. Schürr, A., Selic, B., eds.: Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MoDELS'09). In Schürr, A., Selic, B., eds.: Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MoDELS'09). Volume 5795 of LNCS., Springer (2009)