Proof-Carrying Model-Transformation Components

Jeffrey Terrell, Steffen Zschaler, and Iman Poernomo

King’s College London,
Department of Informatics, London, UK
{jeffrey.terrell|steffen.zschaler|iman.poernome}@kcl.ac.uk

King’s College, Department of Informatics, Technical Report No. TR-11-02

Abstract. As model transformations become a more common tool in a software engineer’s toolbox, there is an increasing need for systematic development techniques for them. Among other things, it becomes increasingly important to be able to safely modularise a model transformation specification as well as to compose new transformations from pre-existing transformation modules. While some initial work has been done to allow transformations to be expressed in a more modular fashion, reuse of such modules requires great care as there is no formal semantics nor a notion of contracts. We present an encoding of model transformations based on constructive type theory. In particular, this allows us to provide formal semantics to transformation modules including an explicit representation and verification of the dependencies of a model-transformation module.

1 Introduction

Model-Driven Engineering (MDE) focuses on using models as the central artefact of software development, using transformations to analyse these models and to eventually produce executable source code from them. Model transformations can encode design rules, platform choices, or even coding conventions. Ideally, MDE can result in better software quality by allowing developers to focus on high-level, domain-centered concepts and ensuring consistency of implementation and reliability of analysis.

As MDE is used increasingly within science and industry, transformations of interest become more complex and more difficult to develop and develop correctly. As a consequence, we need a better understanding of how to systematically develop, structure, and reuse model transformations. Centrally, we need to be able to safely modularise model transformation specifications and reuse partial transformations independently; that is, we need a notion of a model-transformation component. Some previous work on transformation modularisation exists [1–9], but safely reusing transformation modules is still difficult as there are currently no techniques for expressing or verifying a transformation module’s dependencies.

In [10], Terrell and Poernomo showed how model transformations can be specified within a formal method known as Constructive Type Theory (CTT). CTT possesses a property known as the Curry-Howard Isomorphism, where data, functions and their correctness proofs can be 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 (leading to correct-by-construction model transformations). Following [11], we call this implication the proofs-as-model-transformations paradigm. In this paper, we show how we can make use of the higher-order nature of CTT to formally represent model-transformation modules and extend their specifications with an explicit and verifiable description of their dependencies—thus turning them into components in the sense of [12]. As the specifications of such transformations explicitly reference proofs, we call them proof-carrying model-transformation components.

The remainder of the paper is structured as follows: Section 2 discusses recent work on the modularisation of model transformations, highlighting how the lack of a formal semantics leads to an inability to support formal interfaces or contracts for the transformation modules. Section 3 provides a summary of the approach presented in [10] and shows how this can be used to provide a monolithic specification of the example. Section 4 then shows how this specification can be modularised. In Sect. 5, we provide some evaluation of our approach: We first show that the modularised specification is indeed equivalent to the monolithic specification. We then show how modularisation allows us to reuse transformation components to construct a different transformation. Finally, we show how the explicit inclusion of a transformation component’s dependencies in its specification allows only validly composed transformations to be constructed.

Motivating Example. We reuse an example from [13]. The example was drawn from the domain of telecommunications; specifically from models of communications infrastructures expressed in the Common Information Model (CIM) industry standard [14]. 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 1 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. In particular, it shows 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.

2 Modularising Model Transformations

There has been considerable research interest in modularising model transformations for some time already. The approaches proposed and studied so far, may be characterised by the granularity of modules that they provide: At a first level, we can distinguish internal composition of transformation rules from external composition of entire model transformations [16]. We can further differentiate internal composition into
inter-rule composition, where entire rules are taken to be modules, and intra-rule composition, where rules themselves can be composed of finer-grained modules. In the following, we will briefly discuss each of these categories in turn.

### 2.1 External Composition

External composition takes entire model transformations to be modules that can be independently reused and composed. Early research on external composition focused mainly on languages and tools for describing and executing such compositions of reusable model transformations. This has led to early work on megamodelling [17] and transformation chaining [2, 3, 8, 18].

As all of this work considers transformations as black-box components to be composed into larger components, the ‘signature’ or ‘interface’ of a transformation becomes important. These terms refer to the information that can be obtained about a transformation without inspecting its implementation. Initial work on external composition [7, 19] defined transformation signatures by two sets of metamodels: one typing the models that the transformation consumed and another typing the models produced by the transformation. Later research [3] found that this is not always sufficient information for safely composing transformations. In particular, endogenous transformations transform between models of the same metamodel, but may well only address particular elements within this metamodel. Information about the metamodel thus becomes useless when composing a set of endogenous transformations. In addition, some endogenous transformations may be intended to be used with a fixpoint semantics (invoking them until no more changes occur), which makes composing them even more complex. It was concluded in [3] that in addition to the metamodel, there needs to be information about the particular subset of model elements that are used or affected by a transformation. Sen et al. [20] present a similar approach for identifying the effective metamodel used by a transformation. They go on to use it to make transformations reusable across structurally similar, but different, metamodels. In parallel to this work, [8] also identified a need to include information about the technical space of models (e.g., MOF or XML) into the transformation signature.
2.2 Internal Composition

Internal composition considers modules of a finer granularity than entire model transformations. *Inter-rule composition* considers individual rules to be modules, while *intra-rule composition* considers even finer-grained modules and looks at parts of rules.

**Inter-rule Composition** A number of transformation languages 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 [2, 6]. Approaches inspired from graph transformation—for example, VMT [21]—even allow for chaining of individual transformation rules. Module superimposition [9] applies the notion of superimposition from feature-oriented software development [22] to the development of transformation modules, allowing individual rules to be overridden by rules from superimposed modules.

All of these techniques create some flexibility in allowing developers to exchange or independently evolve rules. However, they do not distinguish a rule’s interface from its implementation, which means that rules and rule compositions cannot be verified or understood modularly without inspecting the complete implementation of each rule. Furthermore, some evaluations have shown that there are scenarios where the modularisation capabilities available at the level of complete rules are not sufficient [4–6].

**Intra-rule Composition** To improve modularisation capabilities, a number of mechanisms have been proposed that allow parts of rules to become units of modularity. Balogh and Varró [1] describe how matching and creation patterns can be defined as standalone units of modularity and can be composed to more complex patterns for use in transformation rules. Johannes et al. [13] allow rules to be composed and generated from a number of pattern instantiations annotated to the source metamodel.

While these approaches clearly improve the modularity capabilities of inter-rule composition approaches, they still do not enable modular verification or understanding.

2.3 Summary

Over the past years, transformation modularity has been an important research topic, leading to a range of different proposals for modularisation techniques for model transformations. While most of these techniques provide some assurances with respect to the syntactic correctness of models produced from a composed transformation (if only by virtue of the fact that they abide by a metamodel), there is very little support for modular reasoning about semantic properties. In this paper, we propose a formal encoding of transformation semantics, which allows us to provide modular reasoning and verification about semantic transformation properties.

3 A Formal Model of Model Transformations

In this section, we discuss a type-theoretic model of model transformations based on the one proposed by Poernomo [11].
3.1 Type Theory

The theory of types is predicated on the notion that every object has a type. In fact, a type is defined by prescribing how its objects are constructed. This applies to simple types like \( N \), the type of natural numbers, and complex types like \( (1) \), the type of programs that (as we shall see) implement model transformations. If \( a \) is an object (or inhabitant) of type \( A \), we write \( a : A \).

Our model of model transformations is based on a system of objects and types simultaneously defined by induction as follows.

- If \( a : A \) and \( b : B \), then \( \langle a, b \rangle : A \land B \). Conversely, if \( p : A \land B \), then \( \text{fst} \ p : A \) and \( \text{snd} \ p : B \).
- If \( a : A \) and \( B \) is any type, then \( \text{inl} \ a : A \lor B \); and if \( A \) is any type and \( b : B \), then \( \text{inr} \ b : A \lor B \). Conversely, if \( p : A \lor B \), \( f : A \rightarrow C \) and \( g : B \rightarrow C \), then \( \text{cases} p f g : C \).
- If from an arbitrary \( a : A \) we derive \( b : B \), then \( \lambda a. b : A \rightarrow B \). Conversely, if \( p : A \rightarrow B \) and \( a : A \), then \( (p \ a) : B \).
- If \( \pi \) is a specific object of type \( A \), and \( b : B[\pi/a] \), i.e. \( b \) is an object of a type that not only depends on \( a \) but in which \( a \) is replaced by \( \pi \), then \( \langle \pi, b \rangle : \exists a : A. B(a) \). Conversely, if \( p : \exists a : A. B(a) \), then \( \text{fst} \ p : A \) and \( \text{snd} \ p : B[\text{fst} p/a] \).
- If from an arbitrary \( a : A \) we derive \( b : B(a) \), i.e. an object of a type that depends on \( a \), then \( \lambda a. b : \forall a : A. B(a) \). Conversely, if \( p : \forall a : A. B(a) \) and \( a : A \), then \( (p \ a) : B(a) \).
- If \( a_1 : A \) and \( a_2 : A \), and \( a_1 \) and \( a_2 \) are inter-convertible\(^1\), then \( r(a_1) : a_1 =_A a_2 \).

For example, if \( A =_{df} \{id : N\} \) and \( P =_{df} \{id : N\} \), and \( a : A \) and \( \overline{p} =_{df} \{id = a.id\} : P \), then

\[
\langle \overline{p}, r(a.id) \rangle : \exists p. (a.id =_N p.id)
\]

since \( \overline{p}.id \rightarrow a.id \).

The picture these rules paint is of a very static system. However, type theory is rich in dynamics too, for as Martin-Löf [23] points out, “although the theory of types was originally developed as a symbolism for the precise codification of constructive mathematics, it may equally well be viewed as a programming language”. Unlike other programming languages, though, programs written in type theory are provably correct at the time they are written, in virtue of the isomorphism that exists between objects and types, which was first discovered by Curry [24] and later extended by Howard [25].

3.2 Model Transformations with Type Theory

If we apply our theory of types to the specification and implementation of model transformations, what meaning can we attach to \( a : A \)? There are three possibilities. First, the data interpretation: that \( A \) is a class, and \( a \) is an object of the class. For example, if \( A =_{df} \{id : N\} \) and \( a =_{df} \{id = 1\} \), then clearly we can justifiably write \( a : A \).

\(^1\) In other words, \( a_1 \) and \( a_2 \) are essentially the same object. For example, \( ((\lambda n. n + 1) \ 2) : N \) and \( 3 : N \) are inter-convertible because \( (\lambda n. n + 1) \ 2 \rightarrow 3 \).
Second, the \textit{program} interpretation: that $A$ is the specification of a transformation, and $a$ is a program that implements the transformation. Third, the \textit{proof} interpretation: that $A$ is a proposition within the specification of a transformation, and $a$ is a proof that the proposition holds.

If we regard $S$ as the type of the root class of a source model, and $T$ as the type of the root class of a target model, the specification of the transformation $S \rightarrow T$ is

$$\forall s: S. \text{Pre}(s) \rightarrow \exists t: T. \text{Post}(s, t).$$

(1)

Thus, for every source class that satisfies the pre-condition, there is a target class that, taken with the source class, satisfies the post-condition. Earlier, we saw that functional objects inhabit both universally quantified and arrow types, and that pairs of objects inhabit existentially quantified types. It follows, therefore, that (1) is inhabited by an object of the form

$$\lambda s: S. \lambda h: \text{Pre}(s). (t, p),$$

(2)

that is a program that takes two arguments as input (a source class $s$, and a proof $h$ that $s$ satisfies $\text{Pre}$) and returns a pair of objects as output (the corresponding target class $t$, and a proof $p$ that $s$ and $t$ satisfy $\text{Post}$). This program is clearly certified because it carries alongside its implementation a proof that the post-condition is met.

\subsection{Example}

Consider a transformation between a source model $A$ and a target model $PQR$ (see Fig. 2), in which every object $a: A$ is transformed into three things: an object $p: P$, an object $q: Q$, and a possibly empty list of objects $l: [R]$. Furthermore, suppose that the multiplicities at the directed ends of relationships $s_1$ and $s_2$ are derived from attribute $\text{num}$. E.g., if, for a particular $a: A$, the value of $a.\text{num}$ were 3, then $a$ would be transformed into five objects, three of class $R$ and one each of classes $P$ and $Q$. The reader is encouraged to confirm that this transformation is similar to the one given in Fig. 1, where $A$ is Hub, $P$ is LogicalModule, $Q$ is IPProtocolEndpoint, and $R$ is EthernetPort.

If we assume that for every $a: A$, there is without pre-condition some $p: P$, $q: Q$ and list of $r: R$ with the same $\text{id}$ as $a$, then the specification of the transformation $A \rightarrow PQR$ is given by \footnote{This specification differs slightly from the general form given in (1), because the target model has two root classes.}

$$\forall a: A. \exists p: P. \exists q: Q. \exists l: [R]. \text{Post}(a, p, q, l),$$

(3)
where

\[
Post(a, p, q, l) = (a.aid = p.pid) \land (a.aid = q.qid) \land \\
\forall r : R. r \in p.s_1 \rightarrow (a.aid = r.rid) \land \\
(l.length = a.num) \land (p.s_1 = l) \land (q.s_2 = l).
\]  

(4)

Using type theory, we can prove that (3) is inhabited by

\[
K = d f \lambda a. (f_P, \langle f_Q, \langle f_{[R]}, \ldots \rangle \rangle),
\]

(7)

where

\[
f_P = d f Build_P(a.aid, f_{[R]})
\]

(8)

\[
f_Q = d f Build_Q(a.aid, f_{[R]})
\]

(9)

\[
f_{[R]} = d f Build_{[R]}(a.num, a.aid).
\]

(10)

Build_{[R]} is a recursive function that constructs a list (of a specified length) of objects of \( R \). If we apply \( K \) to a particular \( a_1 : A \), where \( a_1 = d f \{ \text{aid} = 1, \text{num} = 3 \} \), we obtain

\[
K(a_1) \rightarrow (\{ \text{pid} = 1, s_1 = l \}, \{ \text{qid} = 1, s_2 = l \}, \langle l, \ldots \rangle),
\]

(11)

where \( l = d f \{ \text{rid} = 1 \} :: \{ \text{rid} = 1 \} :: \{ \text{rid} = 1 \} :: [] \).

4 A Model of Transformation Modularity

We now extend the proofs-as-model-transformations concept to deal with the specification and implementation of transformation modules. We do this by exploiting the higher-order nature of type theory, parametrising the structure of the \( \forall \exists \) form of model transformation specification (cf. (1)) over additional properties that might hold an augmented, refined version of the specification.

Higher-order quantification is the ability to make statements that are generic or parametrised over functions, statements or proofs of statements. By introducing a type of all propositions, \( Prop \), our type theory treats logical statements as forms of data to be quantified over, just like integers or strings. \( Prop \) permits statements like

\[
\exists X : \text{nat} \rightarrow Prop. \forall i : \text{nat}. X(i),
\]

(12)

which can be read as declaring “We can find a predicate \( X \) that is true over any integer \( i \) – that is, \( X(i) \) holds for any \( i \).” This statement is true because there does indeed exist such a predicate. In the place of \( X \), we could use, for example, \( \text{EvenOrOdd} : \text{nat} \rightarrow Prop \), defined to hold over a number whenever the number is an even or an odd. We could then derive a proof \( p \), i.e.

\[
p : \forall i : \text{nat}. \text{EvenOrOdd}(i).
\]

(13)

A further higher-order aspect of type theory that we exploit is the following. Because proofs (such as (13) above) are also functional programs that have specification
formulae as types, it is possible for us to quantify over proofs as well. For example, the statement

$$\forall X : \text{nat} \to \text{Prop}. \forall y : (\forall i : \text{nat}. X(i)). \forall i : \text{nat}. X(i) \quad (14)$$

can be read as declaring the triviality “For any predicate $X$ over integers, given any proof $y$ that $X(i)$ holds for any integer $i$, then $X(i)$ holds for any integer $i$’. This statement has the following proof (one that essentially takes $y$ and uses it immediately):

$$\lambda X. \lambda y. \lambda i. \left( y_i \right) : \forall X : \text{nat} \to \text{Prop}. \forall y : (\forall i : \text{nat}. X(i)). \forall i : \text{nat}. X(i) \quad (15)$$

These features, together with the Curry-Howard isomorphism, allow us to treat a proved statement such as (15) as a componentized, functional module specification, parametrized over 1) an arbitrary predicate $X$ and 2) a proof (and program) $y$ that satisfies a constraining sub-specification over $X$. As a consequence, we can simultaneously treat the twin notions of component-style assembly and certification. For example, (15) can be treated as a functional module requiring as input another module $y$ that is certified to satisfy $\forall i : \text{nat}. X(i)$ for some given $X$. This module may be instantiated with the predicate $\text{EvenOrOdd}$ for $X$ and the proof $p$ (13) to yield a new proof term, i.e.

$$\left( \left( \lambda X. \lambda y. \lambda i. \left( y_i \right) \right) \text{EvenOrOdd} p \right) : \forall i : \text{nat}. \text{EvenOrOdd}(i) \quad (16)$$

This new proof term can be considered to be the modular proof/program of (15), now instantiated with an acceptable $X$ and certification proof $y$. But because $p$ is also a function that provides functionality to meet a particular specification about $\text{EvenOrOdd}$, we can treat the instantiation as the combination of a requiring module (15) together with a providing module (13), certified to hold over the required subspecification.

We exploit these properties to formally treat modularity in transformations and transformation specifications:

- We parametrize specifications over variables that stand as placeholders for more detailed subspecifications. This allows a transformation specification to be componentized: we can parametrize a transformation specification over properties that are required to hold over other, providing component transformations. These properties may be arbitrary, given as refining constraints over predicate variables. Their details “filled in” to provide a further refinement.
- We parametrize these specifications over required proofs that provider component transformation specifications are valid. By the proofs-as-model-transformations paradigm, this allows a parametrized specification proof to be treated as a transformation containing sub-transformation requirement placeholders. The transformation can then be connected with other component transformations, instantiating sub-transformations. Because the requirement sub-transformations are also proofs of their own specifications, instantiation guarantees the composition is certified to meet the requirement specification.

This componentized approach to transformation definition and certification is depicted in Fig. 3. In the simplest instance, we consider the componentization of two orthogonal concerns into two composed transformation modules, demarcating between structural transformations and data specific transformations. A structural transformation
defines how one graph of model elements should be mapped to another graph of model
elements. A data specific transformation identifies how the data attributes of one model
element should be mapped to those of another. Conceptually, these types of transforma-
tions address different concerns. One mapping of data might be used with a variety of
different structural mappings and vice versa.

We treat the dependencies between the modules as one of the data specific transfor-
mation requiring an instantiating structural transformation in order to form a working
composed transformation. The data specific transformation requires functionality while
a particular structural transformation provides functionality that may be used with the
data transformation. The motivation is that, on its own, the data specific transformation
will merely map single model elements to single model elements: a structural transfor-
mation is required to provide the hierarchy over this mapping.

Viewed as modular functions operating between the same source and target meta-
modes, we can view their composition as one in which the data transformation requires
the service of an arbitrary structural transformation, one that is guaranteed to be orthog-
onal to data concerns.

Consider transformations from a metamodel with type $M_1$ to metamodel with type
$M_2$. Assume $Data(a, b)$ is the form of the specification relating data of model elements
$a : M_1$ to those $b : M_2$.

Then the general type theoretic form of such a data transformation $dataT$ and its
specification type is as follows:

\[
dataT : \begin{cases}
\forall strucT : \forall a : M_1 . \exists b : M_2 . \\
\rightarrow \forall a : M_1 . \exists b : M_2 . \\
\end{cases}
\]

The transformation type specifies a data transformation module, parametrized over a re-
quired arbitrary, guaranteed orthogonal, structural transformation variable $strucT$. The
structural transformation requirements are given by the type of $strucT$. By the Curry-
Howard isomorphism, its type specifies the constraints that must be satisfied by any
instantiation of $strucT$. In particular, the type prescribes a structural relation $StructR$
must hold between the graph structures of input and output models in such a way that
it does not constrain possible mappings between data values of source and target model
elements. Note that the $StructR$ relation is itself a variable: it is a predicate variable
over the data transformation. Taking $StructR$ as an arbitrary parameter, together the
constraint over the form of $strucT$, we add an additional orthogonality constraint, that
the data transformation will work with any kind of graph-to-graph mapping.

In the same way that we may produce instantiations such as (16) of the modular
proof term (15) above, we may instantiate the transformation (17) with a particular
By virtue of the Curry-Howard isomorphism, the composed transformation is immediately guaranteed to satisfy its type specification.

The data transformation of our ongoing transformation can be put into the form of (17) as follows.

\[
\begin{align*}
\forall \text{StrucR} : A \to P \to Q \to \text{Prop.} \\
\forall a : A. \\
\forall \text{pid} : \text{nat.} \\
\forall \text{strucT} : \forall \text{qid} : \text{nat.} \\
\forall \text{rid} : \text{nat.} \\
\exists p : P. \\
\exists q : Q. \\
\forall p, q, r : R. \\
\forall \text{r} : R. r \in p.s_1 \to r.\text{rid} = \text{rid}
\end{align*}
\]

(19)

The data manipulation aspect of the specification is the same as that of (3) above: it maps Hub ids to Module and IPProtocolEndpoint ids. However, the transformation is parametrized over the predicate variable \(\text{StrucR}\), typed to specify an arbitrary graph relationship between hubs \(A\) logical modules \(P\) and IPProtocolEndpoints \(Q\). It requires a structural transformation to instantiate the proof term \(\text{strucT}\), guaranteeing \(\text{StrucR}(a, p, q)\) holds between input Hubs \(a\) and output Modules and Endpoints \(p\) and \(q\). The orthogonality of this transformation with respect to data is stipulated by the additional conjuncts of the type for \(\text{strucT}\). In particular, \(\text{strucT}\) will produce the required structural transformation for any possible values of \(p.\text{pid}\) and \(q.\text{qid}\); this ensures that the particular values determined by the data transformation will not be inconsistent with its composition with an instantiating \(\text{strucT}\).

There are three types of constraints on \(\text{StrucR}\) that we can express in the proof term \(\text{strucT}\):
1. **Statements about what StrucR must not constrain.** These are expressed by conjoining predicates that equate (part of) a parameter to StrucR to something that is all-quantified in the proof term \( \text{strucT} \). For example, the conjoint \( p.pid = pid \) indicates that StrucR cannot constrain \( p.pid \).

2. **Statements about what StrucR must guarantee.** These are expressed by conjoining predicates that only constrain parameters to StrucR. For example, the conjoint \( p.s_1 = q.s_2 \) indicates StrucR must guarantee that these two lists are equivalent.

3. **Statements about what StrucR 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. We have not used this type of constraint above.

## 5 Evaluation

In the previous section, 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. In this section, we present three types of evaluations of this approach:

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.\(^3\)

### 5.1 Recomposing Transformation Modules

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

\[
\text{Link}1(a : A, p : P, q : Q) =_{df} p.s_1 = q.s_2 \land p.s_1.length = a.num
\]

It is possible to derive a proof \( \text{ProofLink1} \) that shows the structural transformation does indeed satisfy the dependency constraints of \( \text{Data} \):

\[
\text{ProofLink1} : \forall a : A. \forall pid : \text{nat}. \forall qid : \text{nat}. \forall rid : \text{nat} . \forall p : P. \exists q : Q.
\text{Link1}(a, p, q) \land p.pid = pid \land q.qid = qid \land p.s_1 = q.s_2 \land

\forall r : R, r \in p.s_1 \rightarrow r.rid = rid
\]

\(^3\) [http://www.steffen-zschaler.de/publications/models11/](http://www.steffen-zschaler.de/publications/models11/)
Proving this theorem is straightforward. Once we have this proof, we can produce the composed transformation

\[ \text{dataT} \ \text{Link1} \ \text{ProofLink1} \]

guaranteed to immediately meet the composite specification

\[
\forall a : A. \exists p : P. \exists q : Q. \ p.s_1 = q.s_2 \land p.s_1.length = a.num \land \\
\ p.pid = a.aid \land q.qid = a.aid \land \\
\ (\forall r : R. r \in p.s_1 \rightarrow r.rid = a.aid) \quad (21)
\]

It can be seen that this is equivalent to the monolithic specification from (3). The only substantive difference in the two formulae is that (3) 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:

\[ \text{Link2}(a : A,p : P,q : Q) = df \ p.s_1 = q.s_2 \land p.s_1.length = 1 \]

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

\[ \text{dataT} \ \text{Link2} \ \text{ProofLink2} \]

5.3 Enforcing Transformation-Module Contracts

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

\[ \text{Link3}(a : A,p : P,q : Q) = df \ p.s_1 = q.s_2 \land p.s_1.length = a.num \land p.pid = 7 \]

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

\[ \text{ProofLink3} : \forall a : A. \forall pid : \text{nat}. \forall qid : \text{nat}. \forall rid : \text{nat}. \exists p : P. \exists q : Q. \]

\[ \text{Link3}(a,p,q) \land p.pid = \text{pid} \land q.qid = \text{qid} \land p.s_1 = q.s_2 \land \\
\ (\forall r : R. r \in p.s_1 \rightarrow r.rid = \text{rid}) \quad (22) \]
It quickly becomes clear that this cannot be done: $Proof\ Link_3$ is required to hold for arbitrary values of $p.pid$, but at the same time $Link_3$ states that $p.pid$ should be 7. Because to construct a composed transformation we require a proof for $Proof\ Link_3$, 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 Outlook

As model transformations are becoming more complex, effective means of modularisation and reuse become more important. Approaches for transformation modularisation proposed so far, however, still make reusing transformation modules difficult as there is no possibility to include dependencies or contracts with each module. This is particularly true for internal composition of model transformations; that is, where the unit of composition is not identical with an entire model transformation, but may encompass only a few transformation rules or even only parts of a rule. In this paper, we have shown how constructive type theory and the idea of proofs-as-model-transformations [11] can be employed to provide formal, verifiable specifications of transformation modules including an explicit specification of their external dependencies. These transformation modules become transformation components in the sense of Szyperski’s definition:

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.” [12]

We have used the higher-order nature of constructive type theory to allow transformation components to parametrise over proofs of any other components they depend on. In this sense, transformation components become proof-carrying components.

We have shown a proof-of-concept of our approach and shown how proofs of complete transformation work in principle. As with many formal methods, there remains a question of scalability. We have orthogonally introduced the notion of partially-ordered specifications in [10]. This provides some support for scalability as it allows breaking transformation specifications into smaller transformation rules that build on each other using induction over model structures (similarly to the way rule-based transformation languages like ATL structure their transformation scripts [6]). In this paper, we have shown ways of further breaking down these rules enabling a more fine-grained separation of concerns—for example, by separating structural transformation from transformation of data. This raises a new question concerning scalability: How manageable are transformation specifications separated along multiple concern dimensions? In this paper, we have shown that such separation provides flexibility for reuse and exchange of parts of a transformation specification. As part of our future work, we plan to improve the usability of our approach by integrating it with ATL [26] or a similar high-level rule-based transformation language and evaluating it against larger case studies.

References

5. Goknil, A., Topaloglu, N.Y.: Composing transformation operations based on complex source pattern definitions. [16] 27–32