Transformation Reuse: What is the Intent?

Rick Salay
Department of Computer Science, University of Toronto,
Toronto, Ontario, Canada M5S 3G4,
rsalay@cs.toronto.edu
Steffen Zschaler
Department of Informatics, King's College London,
London, UK, WC2R 2LS,
szschaler@acm.org
Marsha Chechik
Department of Computer Science, University of Toronto,
Toronto, Ontario, Canada M5S 3G4,
chechik@cs.toronto.edu

Abstract

The ability to reuse transformations across a range of related meta-models is highly desired for many model-driven approaches. For example, it would be useful to be able to reuse standard transformations like state-machine flattening across models instantiating different meta-models of hierarchical state-machines, instead of having to reimplement the same fundamental algorithm just because of small syntactic (or semantic) variations. *Typing* typically captures the question of identifying models for which a given transformation can be successfully applied. *Type compatibility between meta-models* (or the related notion of sub-typing), is intended to ensure that a transformation defined over one type can be successfully executed over any model instantiating a compatible meta-model. However, compatibility mechanisms have not explicitly addressed the question of what it makes for a transformation to be "successfully applied". In this paper, we argue that answering this question is central to a meaningful notion of transformation reuse and must take into account *the intent* of a transformation and seek to preserve it in reuse. We describe how current definitions of type compatibility fail to satisfy this criterion and propose a research agenda for addressing it.

1 Introduction

Model transformations are complex artifacts that require careful engineering. To reduce development effort and increase reliability, transformation reuse is an important tool for transformation developers. Consequently, there has been a good bit of research focus on supporting transformation reuse [1].

A substantial amount of work has focused on notions of model typing. Specific notions of *type compatibility* such as subtyping [2] or type matching [3] have been proposed to determine if a given model M_2 of type T_2 (an instance of a meta-model MM_2) can be provided as an input to a given transformation F accepting models of type T_1 (instances of a meta-model MM_1). In each case, type compatibility is defined by a number of syntactical constraints at the meta-model level. For example, typically, for every meta-class c_1 in MM_1 there must be a corresponding meta-class c_2 in MM_2 such that all attributes and associations of c_1 have a corresponding match in c_2.

Interestingly, there are small variations between the specific constraint sets proposed, but it is not clear which of the variations works better for a given situation. We believe this is in no small part due to a lack of clarity of what it means to "successfully reuse" a model transformation. Clearly, transformation reuse requires preservation of at least some properties of the original transformation F, yet the existing literature does not discuss the goal of transformation reuse at all.

In this paper, we explore the meaning of "successful transformation reuse" and how such a definition would influence the requirements on our notion of type compatibility. After a brief overview of some of the key existing work on transformation reuse and model typing in Sect. 2, in Sect. 3, we first show through an example what problems are caused if a clear notion of successful transformation reuse is missing. We then provide, in Sect. 4, a tentative definition based on a notion of *transformation intent* [4] and explore its implications. We describe one way of constructing valid type-compatibility relations in Sect. 5. It turns out that a completely formal and (potentially) automatable treatment is non-trivial and will require substantial further research. In Sect. 6, we outline the principal shape of any such solution and discuss key research challenges on the way.

2 Existing approaches

Typing has been an important aspect of model-driven engineering (MDE) from the very start. Models are typed by meta-models which define the set of modelling concepts available as well as their relationships and valid combinations. More specific notions of typing have initially been introduced in relation to transformation composition, in particular, to external composition or chaining where transformation signatures were used to decide whether two given transformations could be composed safely, as we discuss below.

Initial work on *external composition* [5, 6] 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 [7] found that this information is not always sufficient information for safely composing transformations. In particular, endogenous transformations transform between models of the same metamodel, but may well address only 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 [7] that the metamodel needs to be augmented with the particular subset of model elements that are used or affected by the transformation. In parallel to this work, [8] also identified a need to include information about the technological space [9] of models (e.g., MOF or XML) into the transformation signature. Alternatively, some of this information has been encapsulated by wrapping models as components themselves, providing interfaces for accessing and manipulating the model independently of its technical representation [5].

Typing of models has also been studied in relation to *transformation reuse*. Here, sub-typing or type matching are used to determine if a given model M_1 (an instance of a meta-model MM_1) can be provided as an input to a given transformation T (typed over a meta-model MM_2). Two notable approaches have been studied in this area: model (sub-)typing and model concepts.

In his thesis and in [10], Steel introduces the notion of *model typing* to specify generic types for model transformations. In this work, a model type is just another meta-model, without any distinguishing properties. Steel then provides a set of type-checking rules defining a 'matching' relationship [11] between meta-models. If a meta-model MM_1 *matches* a meta-model MM_2, Steel allows instances of MM_1 to be passed as parameters to transformations expecting instances of MM_2. The *matches* relationship is defined through a number of syntactic rules over meta-models.

At a later stage, [2] identified some problems with Steel's matching rules, proposing a slightly more restrictive set of rules instead as well as defining a number of variants of the strict matching relationship, called *isomorphic model subtyping*: *non-isomorphic sub-typing* allows the definition of an explicit model adaptation function to translate instances of MM_1 into instances of MM_2. Of particular interest are bi-directional model adaptations. The paper further distinguishes (on a separate dimension) total and partial sub-typing, where partial sub-typing establishes a sub-typing relationship between two meta-models only for the context of specific transformations which are to be applied to the models. This allows matching with a generalized model type that is constructed by keeping only those elements that are explicitly required for the execution of a model-management operation or for access to a particular feature. Partial sub-typing is realized by providing an *effective model type* which is a sub-set of the elements of a given model type and, thus, a super-type of that type. The definition of partial sub-typing is somewhat vague in [2]. In particular, no conditions are given constraining what can or cannot be generalized in an effective model type. Sen *et al.* [12, 13] present an algorithm for deriving effective model types through static analysis of a model transformation's code. However, it is not clear what level of transformation reuse is supported by partial sub-typing base on their algorithm.

De Lara *et al.* [3] and Rose *et al.* [14] have proposed an alternative to model typing, which they call *model concepts*, consisting of the following:

1. *Explicit binding model.* In model typing [10], mapping a meta-model to a model type is automatic and based on name identity. Model concepts require an explicit binding model to express which elements in a meta-model match which elements in a model concept. This is similar to the notions of non-isomorphic sub-typing and adaptation transformations from [2], but provides a more constructive description of the rules governing such an adaptation.

2. *Usage decorations / OCL constraints.* [14] allows elements in model concepts to be decorated as either 'create' or 'delete' to indicate that a model-management operation will create or delete instances of this model element. Usage decorations imply constraints on valid bindings to ensure that multiplicity constraints are not violated. [3] does not propose usage decorations, but instead allows model concepts to be annotated with arbitrary OCL constraints. It remains somewhat unclear whether these constraints could be used to address the same problem.

These ideas are usually discussed only for the input models of a model-management operation, but Cuadrado *et al.* [15] also briefly discuss an extension to input and output models. Cuadrado *et al.* [16] present a simple component model for chaining-based reuse of model transformations based on the notion of model concepts.

In previous work [17], one of the authors has argued that typing of model transformations should explicitly include the syntactic constraints that a transformation expects to hold rather than providing a fixed meta-model.

Kuehne [18, 19] gives a theoretical discussion of sub-typing–like relationships between meta-models. Most importantly, he notes that the validity of such relationships is often relative to context and introduces two notions to describe such contexts: (i) *observers* (called referees in [19]) which provide syntactic rules, and (ii) *contexts* which define more general semantic conditions. The exact construction, in particular, of contexts, is left somewhat open by this work.

To summarize, multiple model type compatibility relations have been proposed, differing in the degree of automation (model sub-typing uses automated type inference, while concepts require explicit bindings to be provided) and in the specific constraints they impose to distinguish compatible types from incompatible ones. However, no attempts have been made to understand whether any of the approaches are valid and which ones are "better than others".

3 The validity of type compatibility

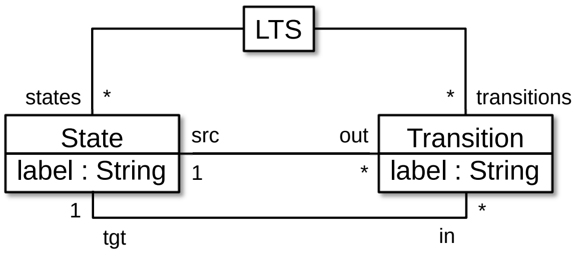
In this section, we attempt to develop criteria for checking the validity of a type compatibility relation by studying its impact on transformation reuse. We begin by assuming that the purpose of a type compatibility relation is *reuse*, i.e., types T_1 and T_2 are type compatible iff all transformations designed for T_1 are reusable for T_2. We formalize this below:

Definition 1 (Type compatibility relation) *Given a set of model types $\mathcal{T}$, a* type compatibility relation *$\mathcal{S}$ over $\mathcal{T}$ consists of a pair $\langle R_\mathcal{S}, \mathcal{F}_\mathcal{S}\rangle$, where*

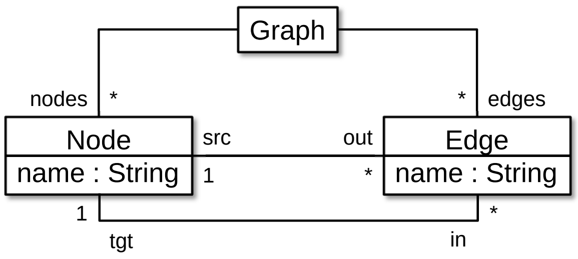
- *$R_\mathcal{S} \subseteq \mathcal{T} \times \mathcal{T}$ is the relation that identifies compatible types;*
- *$\mathcal{F}_\mathcal{S} = \{\Gamma_{(T_2,T_1)} | T_1, T_2 \in \mathcal{T} \text{and } R_\mathcal{S}(T_2,T_1)\}$ is a family of higher order transformations where for every transformation $F : T_1 \to T'$, we get transformation $\Gamma_{(T_2,T_1)}(F) : T_2 \to T'$.*

If $R_\mathcal{S}(T_2,T_1)$, we say that type T_2 is compatible with *type T_1. We write $\Gamma_{(T_2,T_1)}$ as Γ when the context is clear.*

In this definition, the higher order transformations Γ provide the reuse mechanism that comes with type compatibility. In this paper, we assume Γ only affects the source model type of a transformation and leave the more general case where it can affect both source and target types for future work. Note that in order to reuse any transformation $F : T_1 \to T'$ for models typed over T_2, we must first transform it because transformations



(a) LTS model type



(b) LGraph model type

Figure 1: Two different model types.

are strongly typed. The literature [20] typically considers two ways of implementing Γ: (i) (conversion-based) Γ may compose F with a transformation converting the input model from an instance of T_2 into an instance of T_1; or (ii) (rewrite-based) Γ may rewrite the specification of F so that it works directly on models typed by T_2. However, this definition of Γ is too simplistic as we illustrate below.

Example 1. Assume that the model type `LGraph` of labelled graphs is given as compatible with type `LTS` of labeled transition systems by some compatibility relation $\mathcal{S}$. The two types can be seen in Fig. 1. These types are deemed compatible by the current state-of-the-art approaches:

1. *Model Sub-typing [2].* There is no isomorphic sub-typing relationship between these two model types in either direction. However, a simple bi-directional renaming model adapter can be provided (mapping `LTS` to `Graph`, `State` to `Node`, `Transition` to `Edge`, and `label` to `name`, respectively). As a result, a non-isomorphic sub-typing relationship could be established in either direction.

2. *Model concepts [3].* Concepts are bound by providing a morphism between the meta-model MM_2 and the concept MM_1. Given that `LTS` and `LGraph` are structurally identical, such a morphism can easily be established and is identical to the model adapter provided for the model sub-typing case.

Since $\mathcal{S}$ gives us the reuse transformation Γ, we can convert any transformation F : `LTS` $\to T'$ into $\Gamma(F)$: `LGraph` $\to T'$.

Assume that we are given the LTS transformation `camelCase` that changes all transition labels to camel case (i.e., every word starts with capital letter). Aiming to apply this transformation to labeled graphs, we get $\Gamma($`camelCase`$)$: `LGraph` $\to$ `LTS`. Yet this is not what we wanted: `camelCase` was an endogeneous transformation that manipulates LTSs while $\Gamma($`camelCase`$)$ is an exogeneous transformation that converts labeled graphs into LTSs! Thus, clearly *applying* Γ *to a transformation should preserve its endogeneous/exogeneous character.* We formalize this observation as a rule:

Rule 1 ("Character" preservation) *Let types* $T_1, T_2 \in \mathcal{T}$ *where* $R_{\mathcal{S}}(T_2, T_1)$ *holds be given.* $R_{\mathcal{S}}$ *is* character preserving *iff for any transformation* $f : T_1 \to T'$

- *(exo) If* $T' \neq T_1$ *then* $\Gamma(f)$ *has type* $T_2 \to T'$*.*
- *(endo1)If* $T' = T_1$ *then* $\Gamma(f)$ *has type* $T_2 \to T_2$*.*
- *(endo2) If* $T' = T_2$ *then* $\Gamma(f)$ *is undefined.*

By adjusting the definition of Γ to conform to Rule 1, the problem with `camelCase` is fixed since we now get $\Gamma($`camelCase`$)$: `LGraph` $\to$ `LGraph` as the result of transformation reuse.

Example 2. Now consider a transformation `minimize` : `LTS` $\to$ `LTS` that maps an LTS to one that is behaviourally equivalent to it but has a minimum number of states. Reusing this transformation as $\Gamma($`minimize`$)$: `LGraph` $\to$ `LGraph` for labelled graphs yields a problem. While `minimize` has a well-defined *intent* for LTSs – to minimize size while maintaining behavioural equivalence – this intent is not preserved by $\Gamma($`minimize`$)$. In fact, there is no corresponding transformation for `LGraph` that has this intent because labeled graphs have no semantics! Although we can "technically" reuse `minimize`, the resulting transformation is meaningless. So, clearly, *applying* Γ *to a transformation should preserve its intent.*

We explore the implications of this observation in the next section.

4 Preservation of transformation intent

In this section, we explore what it means to preserve transformation intent. In previous work [4], we cataloged some general transformation intents such as *Refactoring*, *Translation*, *Analysis*, etc. that seemed to recur in MDE practice and characterized each intent using a set of *mandatory properties*. That is, every transformation with a given intent must satisfy the corresponding mandatory properties. Since these properties must be applied to many different transformations, they can be abstract and require *concretization* to be checkable for a specific transformation. In the current paper, our objective is not to define general intents but instead to capture the intent of a particular transformation that we wish to reuse. Thus, we approach intent from a different perspective, but as we shall see, we reach similar conclusions.

4.1 Intent as transformation properties

A given transformation F can be characterized by a set of properties – a specification it satisfies that captures its important characteristics. Given such properties, a natural way to determine what makes a transformation "F-like" is that it satisfies the properties that characterize F. Thus, in order to support transformation reuse, we require our higher order transformation Γ to preserve such characterizing properties. We formalize this rule as follows:

Rule 2 (Intent preservation (first attempt)) *A type compatibility relation $\mathcal{S}$ is* valid *iff for all $T_2, T_1 \in \mathcal{T}$, if $R_{\mathcal{S}}(T_2, T_1)$ then for all transformations $f : T_1 \to T'$ characterized by some property P_f, the condition $P_f(\Gamma(f))$ holds.*

Unfortunately, this rule has a problem. A characterizing property like P_f is typically too specific to a given type T_1 and thus cannot be checked for transformations on other types. We illustrate this using a more detailed analysis of transformation properties.

Although some holistic properties such as injectivity or surjectivity may in part characterize a transformation, most transformation-specific characteristics focus on the input / output behaviour. Assume that we are interested in transformations of the form $f : T_1 \to T'$. The general form of a characterizing I/O property is

$$P(f) := \forall x : T_1 \cdot C(x, f(x)),$$

where predicate C expresses a constraint that must hold between the input and output models.

For example, the transformation `minimize` discussed above, is characterized by the property

$$\mathtt{P}^{\mathtt{LTS}}_{min}(f) := \forall x : \mathtt{LTS} \cdot \mathtt{Bisim}_{\mathtt{LTS}}(x, f(x)) \wedge (\forall x' : \mathtt{LTS} \cdot \mathtt{Bisim}_{\mathtt{LTS}}(x', f(x)) \Rightarrow |x'| \leq |f(x)|))$$

which checks that $f(x)$ is behaviourally equivalent to x using an LTS bisimilarity relation and then ensures that there is no other LTS behaviourally equivalent to x that is smaller than $f(x)$.

It is clear that $\mathtt{P}^{\mathtt{LTS}}_{min}(\mathtt{minimize})$ holds, but when we attempt to check $\mathtt{P}^{\mathtt{LTS}}_{min}(\Gamma(\mathtt{minimize}))$, we run into a problem. Since this property is quantified over models of type `LTS`, it cannot be directly applied to a transformation over models of type `LGraph`. To make it work we need to *translate* $\mathtt{P}^{\mathtt{LTS}}_{min}$ to some property $\mathtt{P}^{\mathtt{LGraph}}_{min}$ that represents the same intent for labeled graphs. However, as we observed above, no such property exists.

Now assume that our compatibility relation also says that state machine models (`SM`) are compatible with LTSs using higher-order transformation $\Gamma_{(\mathtt{SM},\mathtt{LTS})}$. In this case, the transformation `minimize` can be meaningfully reused for state machines but we must make sure that $\Gamma_{(\mathtt{SM},\mathtt{LTS})}$ produces a transformation with the correct intent. We can define the characterizing property that represents the intent of `minimize` for state machines as follows:

$$\mathtt{P}^{\mathtt{SM}}_{min}(f) := \forall x : \mathtt{SM} \cdot \mathtt{Bisim}_{\mathtt{SM}}(x, f(x)) \wedge (\forall x' : \mathtt{SM} \cdot \mathtt{Bisim}_{\mathtt{SM}}(x', f(x)) \Rightarrow |x'| \leq |f(x)|))$$

Thus, for $\Gamma_{(\mathtt{SM},\mathtt{LTS})}$ to preserve intent in this instance, we require that $\mathtt{P}^{\mathtt{SM}}_{min}(\Gamma_{(\mathtt{SM},\mathtt{LTS})}(\mathtt{minimize}))$ holds.

4.2 Intent as families of properties

The above discussion showed that specific characterizing properties are not sufficient to define preservation of intent. Instead, we need something more general: *a family of properties* that realize the given intent for each type of a model. In [4], we suggested that such a family can be characterized using a parameterized property that can be concretized by filling in the parameters. For example, we might characterize the "model minimization" intent as the parameterized property

$$\mathtt{P}_{min}^{\langle T\rangle}(f) := \forall x : T \cdot \mathtt{Bisim}_T(x, f(x)) \wedge (\forall x' : T \cdot \mathtt{Bisim}_T(x', f(x)) \Rightarrow |x'| \leq |f(x)|))$$

This still assumes that T is a state-based behavioural modeling language where bisimulation can be used to check equivalence. An even more general version of the intent could be

$$\mathtt{P}_{min}^{\langle T\rangle}(f) := \forall x : T \cdot \mathtt{SemEquiv}_T(x, f(x)) \wedge (\forall x' : T \cdot \mathtt{SemEquiv}_T(x', f(x)) \Rightarrow |x'| \leq |f(x)|)),$$

where $\mathtt{SemEquiv}_T$ is the semantic equivalence relation for models of type T.

Based on these considerations, we revise the intent preservation rule:

Rule 3 (Intent preservation) *Type compatibility relation $\mathcal{S}$ is valid iff for all $T_2, T_1 \in \mathcal{T}$, transformations $f : T_1 \to T'$ and intents I with parameterized characteristic property $P_I^{\langle T\rangle}$, the following condition holds:*

$$\textit{if } R_{\mathcal{S}}(T_2, T_1) \textit{ and } P_I^{T_1}(f) \textit{ and } P_I^{T_2} \textit{ exists then } P_I^{T_2}(\Gamma(f))$$

Note that this rule only requires $P_I^{T_2}(\Gamma(f))$ to hold when $P_I^{T_2}$ exists to take into account cases such as $\mathtt{P}_{min}^{\langle T\rangle}$ that does not exist for $T = \mathtt{LGraph}$. This can be interpreted either as saying that the value of $\Gamma(f)$ is *irrelevant* when $P_I^{T_2}$ does not exist, or that $\Gamma(f)$ is *undefined* when it does not exist (i.e., Γ is partial). In general, the latter interpretation seems more informative but may be harder to achieve in practice. That is, if Γ is a partial transformation then an undefined output can be used to indicate when a transformation cannot be reused.

5 Towards an approach for defining a valid compatibility relation

As discussed in Sect. 3, one way to define Γ [20] is to assume that the compatibility relation between T_2 and T_1 induces a transformation $get_{(T_2,T_1)} : T_2 \to T_1$ (or just *get* when the context is clear) that "converts" a model of type T_2 into a corresponding model of type T_1. The use of conversion (and coercion) functions is a standard technique of type theories [21] used in programming and we adopt them for model type compatibility. Given this conversion transformation, we can define $\Gamma(f)(x) := f(get(x))$ for the exogenous case.

To address the endogenous case (endo1), we define a bidirectional transformation [22] for conversion between T_1 and T_2. Following the *lenses* approach to bidirectional transformations [23], we define this as a pair of transformation $\langle get, put\rangle$, where *get* is as defined as above and *put* has the type $T_1 \times T_2 \to T_2$. Bidirectional transformations are a generalized approach to the "view-update" problem – a T_2 model is first "viewed" as a T_1 model using *get*, then this T_1 mode is updated manually or by a transformation and finally *put* reflects this update back in the original T_2 model which it takes as one of its inputs. Thus, for the endogenous case, we define $\Gamma(f)(x) := put(x, f(get(x)))$.

We now give the definition for a compatibility relation based on conversion transformations.

Definition 2 (Conversion-based type compatibility relation) *A compatibility relation $\langle R_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}}\rangle$ is* conversion based *if it provides a family of bidirectional conversion transformations $\{\langle get_{(T_2,T_1)} : T_2 \to T_1, put_{(T_2,T_1)} : T_1 \times T_2 \to T_2\rangle | T_1, T_2 \in \mathcal{T}$ and $R_{\mathcal{S}}(T_2, T_1)\}$, and for $\Gamma_{(T_2,T_1)} \in \mathcal{F}_{\mathcal{S}}$ and transformation $f : T_1 \to T'$, the following hold:*

- $\Gamma_{(T_2,T_1)}(f)(x) = put(x, f(get(x)))$, *if* $T' = T_1$;
- $\Gamma_{(T_2,T_1)}(f)(x)$ *is undefined, if* $T' = T_2$; *and*
- $\Gamma_{(T_2,T_1)}(f)(x) = f(get(x))$, *otherwise.*

For example, assume that we have a conversion-based compatibility relation $\mathcal{S}_{bi}$ that is limited to bijective conversion transformations and we define the bidirectional transformation between `LGraph` and `LTS` as follows:

- *get* : `LGraph` $\to$ `LTS` – converts a labeled graph into an LTS by changing nodes into states, edges into transitions and names into labels.
- *put* : `LGraph` $\times$ `LTS` $\to$ `LGraph` – converts an LTS into a labeled graph by changing states into nodes, transitions into edges and labels into names.

Note that *put* ignores its first input argument (i.e., the original labeled graph) since both it and *get* are simple bijective transformations and thus the original model is not needed to construct the update.

At this point, we can ask whether Γ defined as in Defn. 2 preserves the intent of particular LTS transformations. We have seen that the transformation `minimize` has no corresponding transformation for labeled graphs, so the action of Γ on this transformation is irrelevant. Unfortunately, Γ cannot "tell" us that this is not a case of good reuse by being undefined on this input since the conversion transformations *get* and *put* are total, and so Γ is total as well. This points to one weakness of $\mathcal{S}_{bi}$ – the conversion transformations will be total for any pair of types defined as compatible by $\mathcal{S}_{bi}$ because they are restricted to being bijective and so we cannot rely on the partiality of Γ to indicate when a transformation cannot be reused.

Now consider the transformation `camelCase`. It can be characterized by the following parameterized property:

$$\mathtt{P}_{cc}^{\langle T\rangle}(f) := \forall x : T \cdot \mathtt{Camelize}_{\langle T\rangle}(x, f(x))$$

where $\mathtt{Camelize}_{\langle T\rangle}(x, y)$ holds when model x is isomorphic to model y and for each pair $\langle e, e'\rangle$ of elements mapped by the isomorphism, the attribute of e' representing its name (if one exists) is the camel case version of the corresponding attribute of e. It is easy to see that $\mathtt{P}_{cc}^{\mathtt{LTS}}(\mathtt{camelCase})$ holds, as we have described it and that $\mathtt{P}_{cc}^{\mathtt{LGraph}}(\Gamma(\mathtt{camelCase}))$ also holds. Thus, the intent of `camelCase` is preserved for this instance, but is it preserved for any pair of compatible types in $\mathcal{S}_{bi}$?

The characteristic of the transformations *get* and *put* that allow this preservation is the fact that they are bijective but also that named elements are always mapped to named elements, i.e., the meta-attribute of an element being "named" is preserved by the conversion transformations. Thus, $\mathcal{S}_{bi}$ as currently defined is not guaranteed to preserve the intent of `camelCase`. However, it can be *fixed* to do so by adding the constraint that conversion transformations must preserve named elements.

In the above example, the *intent of the conversion transformations* in the fixed $\mathcal{S}_{bi}$ is characterized by the property of being bijective and preserving named elements. In the general case, since Γ is formed as the composition of conversion transformations with the reused transformation, the problem of coming up with a good conversion-based compatibility relation can be expressed in terms of the composition of transformation intents. That is, we want to determine the intents of *get* and *put* so that when we compose these with a transformation f as in Defn. 2, the intent of the result is the same as the intent of f. It should be clear that we can achieve this in specific cases, e.g., we can determine the required intent of *get* and *put* to guarantee the intent preservation for `camelCase`. It is less clear whether we can ensure the preservation of intent for a broad class of transformations. We leave this investigation to future work.

6 Research Agenda

From our analysis in the previous sections, we have seen that *intent preservation* should be the driving factor behind deciding whether a type compatibility relationship is valid. It would thus be highly beneficial to provide a simple and easily checkable (ideally, automatically) criterion of whether two model types are type-compatible to each other. Below, we explore what is required to accomplish this.

In Sect. 5, we have discussed how Γ could be realized as a conversion transformation composed with the transformation to be reused. We have briefly discussed that for this to work, we would need to reason about the composition of transformation intents. How to perform such a reasoning remains a research challenge. Two things will need to be understood to this end:

1. We require a precise technique for describing transformation intents. In Sect. 4.2, we made a first attempt at describing transformation intent as a family of properties specified by a parameterized formula. It remains to be understood how to correctly instantiate such a a parameterized formula, e.g., what are the constraints for instantiating $\mathtt{Bisim}_T$? How can these constraints be effectively expressed and verified?

2. We require a calculus for deriving transformation intents from the composition of other transformation intents as induced by the composition of the transformations themselves. Such a calculus can then be employed for implementing the reasoning employed in Sect. 5.

There is another issue here as well: It seems likely that no type-compatibility relation $\mathcal{S}$ could preserve every intent of every transformation $T_1 \to T'$. Moreover, there will be weaker type-compatibility relations that only preserve a certain subset of transformation intents, but allow more types to be considered compatible. Consequently, we argue with Kuehne [19] that type-compatibility needs to be considered as a contextual notion

– type compatibility with respect to a class of intents (perhaps written as $\mathcal{S}_{\mathcal{I}}$ for a given class $\mathcal{I}$ of transformation intents). For example, we may consider `LGraph` and `LTS` compatible for intents that only consider the syntactical structure of a model, but not for intents that refer to a model's semantics. If this is the case, then a substantial amount of research is required to

1. Identify suitable classes of intents;
2. Identify suitable type-compatibility relations to go with each class of intents;
3. Understand whether there are classes of transformation intents for which no suitable type-compatibility relation exists; and
4. Understand the relationships between these intent classes. For example, is there a lattice of increasingly more constrained notions of type compatibility that provide increasing assurances while decreasing flexibility of reuse? If so, where is the sweet spot in this tradeoff?

Note that, in particular, the goal cannot be to require transformation reusers to provide a full proof *at the point of reuse* that the type-compatibility relation they chose maintains the specific intent of the transformation to be reused. Instead, they should be able to identify the class of transformation intent and reuse a type-compatibility relation for which it has been previously proved that it maintains intents of that class. In this way, the hard work of proving validity of type-compatibility relations can be reused.

7 Conclusion

In this paper, we aimed to assess the validity of the current proposals for model type compatibility relations [3, 2, 17] and the essence of differences between them. In so doing, we have understood that current approaches miss a key ingredient, namely, *the intent of the transformations to be reused*. While transformation intent has been studied in other contexts [4], we have explored the potential impact of intent on transformation reuse and our notion of type compatibility. Although we have added some more clarity to the discussion, the main contribution of the paper is the identification of a new research agenda: we believe there is a need to explore formal representations of transformation intent, as well as the precise relationship between transformation intent and reuse-oriented type-compatibility relations. We invite workshop participants to join us in working on this research agenda.

References

[1] A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger, "Reuse in model-to-model transformation languages: are we there yet?" *SoSyM*, vol. 14, no. 2, pp. 537–572, May 2015. [Online]. Available: http://dx.doi.org/10.1007/s10270-013-0343-7

[2] C. Guy, B. Combemale, S. Derrien, J. R. Steel, and J.-M. Jézéquel, "On model subtyping," in *Proc. 8th European Conf. on Modelling Foundations and Applications (ECMFA'12)*, ser. LNCS, A. Vallecillo, J.-P. Tolvanen, E. Kindler, H. Störrle, and D. Kolovos, Eds., vol. 7349. Springer, 2012, pp. 400–415.

[3] J. de Lara and E. Guerra, "From types to type requirements: Genericity for model-driven engineering," *SoSyM*, vol. 12, no. 3, pp. 453–474, 2013.

[4] L. Lúcio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G. M. Selim, E. Syriani, and M. Wimmer, "Model transformation intents and their properties," *Software & Systems Modeling*, pp. 1–38, 2014.

[5] R. Marvie, "A transformation composition framework for model driven engineering," University of Lille 1, Tech. Rep., 2004, lIFL technical report 2004-n10.

[6] A. Vignaga, F. Jouault, M. C. Bastarrica, and H. Brunelière, "Typing in model management," in *Proc. 2nd Int'l Conf. on Theory and Practice of Model Transformations (ICMT'09)*, ser. Lecture Notes in Computer Science, R. Paige, Ed., vol. 5563. Springer-Verlag, 2009, pp. 197–212.

[7] R. Chenouard and F. Jouault, "Automatically discovering hidden transformation chaining constraints," in *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MoDELS'09)*, ser. LNCS, A. Schürr and B. Selic, Eds., vol. 5795. Springer, 2009, pp. 92–106.

[8] B. Vanhooff, D. Ayed, S. V. Baelen, W. Joosen, and Y. Berbers, “UniTI: A unified transformation infrastructure,” in *Proc. 10th Int'l Conf. Model Driven Engineering Languages and Systems (MoDELS'07)*, ser. Lecture Notes in Computer Science, G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, Eds., vol. 4735. Springer, 2007, pp. 31–45.

[9] I. Kurtev, J. Bézivin, and M. Aksit, “Technological spaces: An initial appraisal,” in *CoopIS, DOA'2002 Federated Conferences, Industrial track*, 2002. [Online]. Available: http://www.sciences.univ-nantes.fr/lina/atl/www/papers/PositionPaperKurtev.pdf

[10] J. Steel and J.-M. Jézéquel, “On model typing,” *SoSyM*, vol. 6, no. 4, pp. 401–413, 2007.

[11] K. B. Bruce, L. Petersen, and A. Fiech, “Subtyping is not a good “match” for object-oriented languages,” in *Proc. 11th European Conf. on Object-Oriented Programming (ECOOP'97)*, ser. LNCS, M. Akşit and S. Matsuoka, Eds., vol. 1241. Springer, 1997, pp. 104–127. [Online]. Available: http://dx.doi.org/10.1007/BFb0053376

[12] S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel, “Meta-model pruning,” in *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MoDELS'09)*, ser. LNCS, A. Schürr and B. Selic, Eds., vol. 5795. Springer, 2009. [Online]. Available: http://www.irisa.fr/triskell/publis/2009/Sen09b.pdf

[13] S. Sen, N. Moha, V. Mahé, O. Barais, B. Baudry, and J.-M. Jézéquel, “Reusable model transformations,” *SoSyM*, vol. 11, no. 1, pp. 1–15, 2010. [Online]. Available: http://dx.doi.org/10.1007/s10270-010-0181-9

[14] L. Rose, E. Guerra, J. de Lara, A. Etien, D. Kolovos, and R. Paige, “Genericity for model management operations,” *SoSyM*, 2011, published on-line first.

[15] J. S. Cuadrado, E. Guerra, and J. de Lara, “Flexible model-to-model transformation templates: An application to ATL,” *Journal of Object Technology*, vol. 11, no. 2, pp. 4:1–28, 2012. [Online]. Available: http://dx.doi.org/10.5381/jot.2012.11.2.a4

[16] ——, “A component model for model transformations,” *IEEE Trans. Software Eng.*, vol. 40, no. 11, pp. 1042–1060, 2014.

[17] S. Zschaler, “Towards constraint-based model types: A generalised formal foundation for model genericity,” in *Proc. 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO'14)*, C. Atkinson, E. Burger, T. Goldschmidt, and R. Reussner, Eds., 2014.

[18] T. Kühne, “An observer-based notion of model inheritance,” in *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MoDELS'10)*, ser. LNCS, D. Petriu, N. Rouquette, and Ø. Haugen, Eds., vol. 6394. Springer, 2010, pp. 31–45.

[19] ——, “On model compatibility with referees and contexts,” *Software & Systems Modeling*, vol. 12, no. 3, pp. 475–488, 2013.

[20] J. de Lara and E. Guerra, “Towards the flexible reuse of model transformations: A formal approach based on graph transformation,” *Journal of Logical and Algebraic Methods in Programming*, vol. 83, no. 5–6, pp. 427–458, 2014, 24th Nordic Workshop on Programming Theory (NWPT 2012). [Online]. Available: http://dx.doi.org/10.1016/j.jlamp.2014.08.005

[21] Z. Luo, S. Soloviev, and T. Xuea, “Coercive subtyping: Theory and implementation,” *Information and Computation*, vol. 223, pp. 18–42, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.ic.2012.10.020

[22] Z. Hu, A. Schürr, P. Stevens, and J. F. Terwilliger, “Bidirectional transformation" bx"(dagstuhl seminar 11031).” *Dagstuhl Reports*, vol. 1, no. 1, pp. 42–67, 2011.

[23] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt, “Combinators for bi-directional tree transformations: a linguistic approach to the view update problem,” *ACM SIGPLAN Notices*, vol. 40, no. 1, pp. 233–246, 2005.

[24] A. Schürr and B. Selic, Eds., *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MoDELS'09)*, ser. LNCS, vol. 5795. Springer, 2009.