Bidirectionizing Model Transformation Languages through Partial Translation*

Soichiro Hidaka Massimo Tisi

While most model-transformation languages in Model-Driven Engineering are unidirectional, bidirectionality is a valuable property in a number of applicative contexts where artifacts need two-way synchronization. Several bidirectional transformation engines have been developed in various research communities, but the bidirectional semantics of a model transformation is generally considered more difficult to express and predict compared to the unidirectional case.

In this paper we propose a method to partially bidirectionalize existing unidirectional languages by compilation towards bidirectional languages and coupled execution of the two language engines. Users write the forward direction of their transformations in the same unidirectional language they are used to, and obtain a system that (besides performing the complete forward transformation) can automatically propagate in the backward direction a well-defined subset of the target updates. The method does not require any ad-hoc modification to the two transformation engines, and has clear well-behavedness properties that make the behavior of the combined system predictable. The approach is exemplified by bidirectionalizing the ATL model-transformation language on top of the GRoundTram bidirectional graph-transformation system.

1 Introduction

In the field of Model-Driven Engineering (MDE), model-transformation languages (MTLs) have been designed to facilitate the expression of relationships between models, in the form of transformation rules [5]. Transformation engines apply such rules to automatically generate new models or update existing ones, keeping the transformation's source and target models consistent. In several applications, synchronization between models needs to be bidirectional, as modifications can be performed in both sides of the transformation and global consistency has to be guaranteed. For this reason several bidirectional transformation languages have been developed, both within and outside the MDE community [17]. However, taking into account the bidirectional semantics during the development of the model transformation adds a significant layer of complexity to the transformation design: execution properties of a bidirectional system are generally more complex and it is more difficult for the developer to predict the exact system behavior [8]. Because of this, several MTLs are unidirectional, and focus on providing an efficient solution for the unidirectional scenario. It is still possible to achieve bidirectionality in these languages, by separately writing two opposite transformations. However this raises other reasons of complexity, since users need to verify case by case the consistency of the two unidirectional transformations, and keep their evolution coupled.

An alternative approach for the application of unidirectional transformation languages to bidirectional scenarios is the *bidirectionalization* of the unidirectional language [25][31]. In this approach, users write one direction of the transformation by using the unidirectional transformation language they are used to. The language is overloaded with a bidirectional semantics so that the resulting system is equivalent to the unidirectional system in the forward direction, but it exhibits the additional ca-

*This is an unrefereed paper. Copyrights belong to the Authors.

部分翻訳に基づくモデル変換言語の双方向化

日高 宗一郎, 法政大学情報科学部, Faculty of Computer and Information Sciences, Hosei University.

マシモ ティージー, AtlanMod チーム (フランス国立情報学自動制御研究所, 国立高等鉱業学校ナント校, LINA), AtlanMod team (Inria, Mines Nantes, LINA).

pability of propagating in the backward direction a set of changes to the target models. Full bidirectionalization of the unidirectional model transformation is in general not possible without imposing restrictions to the model transformation language, and existing work [27] is mainly focused on extending the part of the unidirectional language that is bidirectionalized.

Instead of focusing on maximizing the extent of the bidirectionalization, this paper proposes a coupled execution of unidirectional and bidirectional transformation engines. The user writes a unidirectional transformation in an existing unidirectional language, and the unidirectional transformation engine performs the complete forward transformation with its full expressiveness. A transformation compiler translates *part of* the unidirectional transformation into an equivalent transformation in an existing bidirectional language. The unidirectional and bidirectional transformation engines are executed in parallel and the system performs seamless synchronization between the respective models. The semantic gap caused by partial compilation makes the full system alignment challenging.

With respect to related work, our approach: 1) does not require restrictions to the forward transformation semantics; 2) clearly distinguishes the parts of the target models whose updates are subject to backward-propagation by the underlying bidirectional transformation engine; 3) does not require modifications to any of the two engines involved; 4) exhibits a easily user-predictable behavior (formally it satisfies a variation of well-behavedness property from [4]). We define the conditions for the applicability of our approach in terms of properties that need to be respected by the two transformation engines and by the partial translation. We show that a system respecting these properties can support a useful set of backward-propagable updates.

We exemplify the proposed approach by building a bidirectionalized version of the ATL model transformation language [19], on top of the GRoundTram [13][14] bidirectional graph-transformation system by partially translating ATL to UnQL [2], the graph transformation language bidirectionalized [12] in the GRoundTram system. The resulting tool is a contribution of its own and its source code is freely available at the paper page[†1]. The page contains also a technical report [16] that extends this paper with technical details.

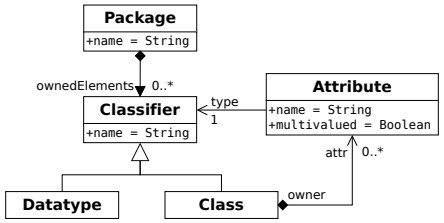


Fig. 1: Class diagram metamodel

The paper is structured as follows: Section 2 introduces a running case for the paper; Section 3 gives a general description of the approach and Section 4 formalizes its main properties; Section 5 illustrates the integration of ATL and GRoundTram; Section 6 compares the approach with related work and Section 7 concludes the paper.

2 Running Case

2.1 User scenario

Our research effort aims at simplifying the coupled evolution of model artifacts in MDE, hence it is applicable whenever some models are maintained and evolve while staying consistent with each other. To illustrate our approach, we refer in this paper to a simple example that considers two models: a UML class diagram and a relational schema. We can imagine, e.g., that the class diagram is modeling the code of an object-oriented program and the relational schema (that models tables in a relational database) represents its persistence layer. The user wants to develop a model-to-model transformation (MMT) to express the relation between the UML classes of the application's data model and the relational tables, according to an object-persistence policy. The transformation computes which tables and attributes to produce for each UML element. Figures 1 and 2 show the source and target metamodels of the transformation, while Fig. 3 shows an instance of each metamodel[†2].

We suppose that both the class model and the re-

†1 `https://github.com/atlanmod/ATLGT`.

†2 While in figure we show models in their abstract object-diagram notation, we can imagine them to be rendered in their particular graphical concrete syntax by the editing tools.

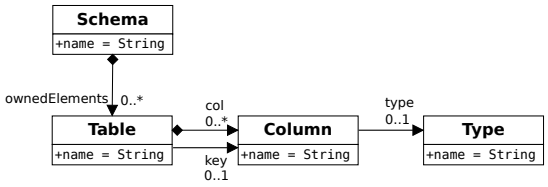


Fig. 2: Relational metamodel

lational schema may be manually edited during the application development or maintenance, depending on the user background (software architect or database expert) or the preferred tool (UML diagram editor or database editor). While application code will have to be manually evolved, an automated system should make sure that classes and tables of the data model are always synchronized. Hence our scenario requires a MMT system working in two directions.

While bidirectional MMT systems exist that would fit this scenario, they require the representation of consistency by a bidirectional transformation language. In this paper we propose an alternative approach that allows the user to specify the transformation using a unidirectional language. Depending on the particular unidirectional language, the user may prefer it because of higher expressive power, simpler semantics or just familiarity.

We propose to address the two synchronisation directions by using different tools:

Forward: The user writes the forward transformation using a unidirectional MMT language and the unidirectional engine performs forward transformation.

Backward: Part of the forward transformation is transparently compiled into a bidirectional transformation language. A bidirectional engine performs backward propagation of target updates.

In the following we describe the two transformation engines we selected for our experimentation: the unidirectional ATL model-transformation engine [19] and the bidirectional GRoundTram graph-transformation engine [12] [13] [14].

2. 2 ATL

In our example the user develops in the ATL language the simple MMT transformation in Listing 1 that transforms the left side of Fig. 3 into its right side[†3].

An ATL transformation (*module*) is constituted by a set of rules that describe how elements of the target model are generated when patterns in the source model are matched. The Object Constraint Language OCL [32] is used as expression language in the transformation rules. Rules are composed of an *input pattern* and an *output pattern*. Input patterns are associated with OCL *guards* that impose conditions on the input elements matched by the input pattern. Output patterns are associated with *bindings*, OCL expressions that define how to initialize properties of the elements created by the output pattern.

E.g., the rule *SingleValuedAttribute2Column* (lines 28-38) selects input model elements of type Attribute (line 30) and transforms them into output elements of type Column (line 33). A guard (line 31) imposes to match only attributes that are not multivalued (multivalued attributes would have a more complex representation in the relational schema). A first binding (line 34) computes the column name as the concatenation of class and attribute name; a second one (line 35) imposes that the owner of the Column corresponds to the owner of the Attribute; a third one (lines 36-37) states that if the attribute has a primitive type (*DataType*) then the type of the column should correspond to the type of the attribute (String otherwise).

Rule *Class2Table* (lines 20-27) creates a table for each class, initialises the list of columns with the respectively transformed attributes (line 25), and selects as table key the attribute whose name ends by the string *Id*, if any (line 26). Rule *DataType2Type* (lines 12-19) copies into relational types the class diagram types, but only if no class exists with the same name of the type, to avoid conflicts. Finally, rule *Package2Schema* (lines 3-11) transforms all non-empty packages into relational schemas and

†3 This transformation code is inspired from the Class2Relational case study (`http://sosym.dcs.kcl.ac.uk/events/mtip05`), the de-facto standard example for model transformations, but simplified and adapted for illustrative purpose. Notably this version of the transformation does not support multivalued attributes in the class diagram.

```
1  module Class2Relational;
2  create OUT : Relational from IN : ClassDiagram;
3  rule Package2Schema{
4    from
5      p:ClassDiagram!Package
6       (not p.ownedElements.isEmpty())
7    to
8      out:Relational!Schema
9       (name<-p.name,
10       ownedElements<-p.ownedElements)
11 }
12 rule DataType2Type {
13   from
14     d:ClassDiagram!DataType
15      (ClassDiagram!Class.allInstances()
16         ->select(c | c.name = d.name)->isEmpty())
17   to
18     t:Relational!Type (name<-d.name)
19 }
20 rule Class2Table {
21   from
22     c:ClassDiagram!Class
23   to
24     t:Relational!Table
25      (name<-c.name, col<-c.attr,
26       key<-c.attr->select(a | a.name.endsWith('Id')))
27 }
28 rule SingleValuedAttribute2Column {
29   from
30     a:ClassDiagram!Attribute
31      (not a.multivalued)
32   to
33     c:Relational!Column
34      (name<-a.owner.name+'_'+a.name,
35       owner<-a.owner,
36       type<-if a.type.oclIsTypeOf(ClassDiagram!DataType)
37         then a.type else thisModule.stringType endif)
38 }
```

Listing 1: Class2Relational in ATL

initializes their list of tables.

Note that ATL rules implicitly interact for generating cross-references among target model elements. E.g., at line 25 the binding `col<-c.attr` states that the columns (`col`) of the table to generate correspond to the attributes (`attr`) of the matched class (`c`): in practice, the result of the transformation of these attributes, by any rule, will be added as a column of the table.

Listing 1 exemplifies the core subset of the ATL language, for an illustration of the complete language we refer the reader to [19].

2.3 GRoundTram

GRoundTram is an integrated framework for developing well-behaved bidirectional transformations expressed in the language UnQL [2]. Figure 4 shows the syntax of UnQL. It has templates (T) at the top level to produce rooted graphs with labeled (L) branches ($\{L : T, \ldots, L : T\}$), union of templates ($T \cup T$), graph variable reference ($\$g$), conditionals, SQL-like select (**select**) queries with bind-

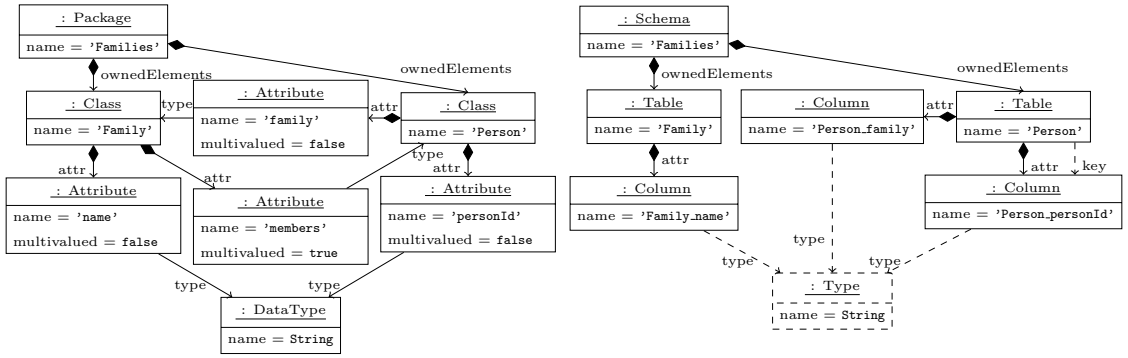


Fig. 3: Sample ClassDiagram model and corresponding Relational model (shown in object-diagram syntax). Corresponding elements have the same position in the two models. Updates to elements with dashed lines are not backward-propagable.

(template)	T	::=	$\{L:T,\ldots,L:T\}$ | $T \cup T$
		|	$\$g$ | **if** BC **then** T **else** T
		|	**select** T **where** $B,\ldots,B$
		|	**letrec sfun** $fname(L:\$G)$ $= \ldots$ **in** $fname(T)$
(binding)	B	::=	Gp **in** T | BC
(condition)	BC	::=	**not** BC | BC **and** BC
		|	BC **or** BC | $L = L$
		|	**isEmpty**(T)
(label)	L	::=	$\$l$ | `a`
(label pattern)	Lp	::=	$\$l$ | Rp
(graph pattern)	Gp	::=	$\$G$ | $\{Lp:Gp,\ldots,Lp:Gp\}$
(regular path pat.)	Rp	::=	`a` | `_` | $Rp.Rp$ | $(Rp\|Rp)$
		|	$Rp?$ | $Rp*$ | $Rp+$

Fig. 4: Syntax of UnQL

ings (B), and structural recursion (**sfun**). Bindings include graph pattern matching (Gp **in** $\$G$) and Boolean conditions (BC). Conditions include usual logical connectives, label equality and emptyness check **isEmpty**(T) which is true if there is at least one edge reachable from the root of the graph generated by T. Label expressions include label constants and variable ($\$l$) bound by the label patterns in the graph patterns and the argument of structural recursion. Graph patterns bind the entire subgraph $\$G$ or labels on the branches and following subgraphs $\{Lp : Gp, \ldots, Lp : Gp\}$. Label patterns may include regular expressions of the labels (Rp) along the path traversing multiple edges.

Listing 2 shows the UnQL transformation that corresponds to the bidirectional part of the ATL transformation in Listing 1. It transforms the graph encoding of the Class model in the left side of Fig. 3 to the graph encoding of the Relational model in the right side, without the dotted part.

The transformation consists of structural recursion functions (**sfun**s). In the listing, the last definition (function *Class2Relational* at lines 13-31) is the main function that corresponds to the ATL module *Class2Relational*. Its clauses generate elements according to different ATL rules — *Package2Schema* (lines 14-19), *Class2Table* (lines 20-22), and *SingleValuedAttribute2Column* (lines 23-30). As described in Section 3, the clause for *DataType2Type* is missing by intention, as to demonstrate the partiality of the translation. References between target elements are represented by definition and call of sibling mutually-recursive functions: *ownedElements_Package2Schema* called at line 17 for the reference ownedElements in the rule *package2Schema* (lines 2-5), *col_Class2Table* called at line 22 for col in the rule *Class2Table* (lines 6-8), and *owner_SingleAttribute2Column* called at line 30 for owner in the rule *SingleAttribute2Column* (lines 9-12). Several calls and definitions for other bindings are again missing by intention. The three SQL-like **select** expressions (lines 18, 21 and 27) perform shallow copy of primitive data type values by capturing the subgraph under label `name` and creating the new edge with the same label and the same subgraph. The conditional expressions **if** (lines 15-19 and 24-30) with the graph emptiness predicate **isEmpty** check the existence of particular subgraph patterns, and are used to express the ATL guards. The **where** clause allows regular expressions on labels like *multivalued.Boolean* (line 25) for complex pattern matching: we use it to encode traversals of attributes and references in OCL expressions.

```
select
 letrec sfun
    ownedElements_Package2Schema({ownedElements:$g})
    = {ownedElements:Class2Relational($g)}
  | ownedElements_Package2Schema({$l:$g}) = {}
and sfun
    col_Class2Table({attr:$g}) = {col:Class2Relational($g)}
  | col_Class2Table({$l:$g}) = {}
and sfun
    owner_SingleAttribute2Column({owner:$g})
    = {owner:Class2Relational($g)}
  | owner_SingleAttribute2Column({$l:$g}) = {}
and sfun (* main function *)
    Class2Relational({Package:$g})   =
    if (not isEmpty(
      select {dummy:{}} where {ownedElements:$g} in $g)
    then {Schema: (ownedElements_Package2Schema($g)
                ∪ (select {name:$g} where {name:$g} in $g))}
    else {}
  | Class2Relational({Class:$g})     =
    {Table:((select {name:$g} where {name:$g} in $g)
            ∪ col_Class2Table($g))}
  | Class2Relational({Attribute:$g}) =
    if isEmpty(select {dummy:{}}
                where {multivalued.Boolean:$g'} in $g,
                      {true:$d} in $g') then
    {Column:((select {name:{String:{$on ^ $n}}}
              where {owner._name.String:{$on:$d}} in $g,
                    {name.String:{$n:$d}} in $g)
             ∪ owner_SingleAttribute2Column($g))} else {}
in Class2Relational($db)
```

Listing 2: Class2Relational transformation in UnQL

GRoundTram extends UnQL as UnQL+ to support more user-friendly syntactic sugars [15] [14] but this paper does not depend on the extension.

3 Overview of the Approach

Our strategy for bidirectionalization is to combine a popular and powerful unidirectional model transformation language with an existing bidirectional transformation language with well-defined bidirectional properties. The user writes the transformation in the unidirectional language and does not need any knowledge of the bidirectional language, since the bidirectional system is executed completely behind the scenes. The bidirectional transformation language may sacrifice expressive power for the capability of back-propagating updates on the target. We tolerate the expressivity gap between the two languages by *partially* translating the unidirectional transformation language into the bidirectional transformation language.

Fig. 5 shows a global view on our approach. We run the unidirectional (UX in figure) and bidirectional (BX in figure) transformation systems in parallel, maintaining their artifacts in separate technical spaces. Because of partial translation the UX and BX transformations are not equivalent and our challenge is providing a generic method to keep the two transformation systems consistent with each other. Thanks to such synchronization the user of our example will be able to generate the relational model in Figure 3, to update it (e.g., by renaming a table or adding a column) and to see the change propagated to the class diagram (correspondingly, as a renamed class or an added attribute).

A full system roundtrip starts from a source

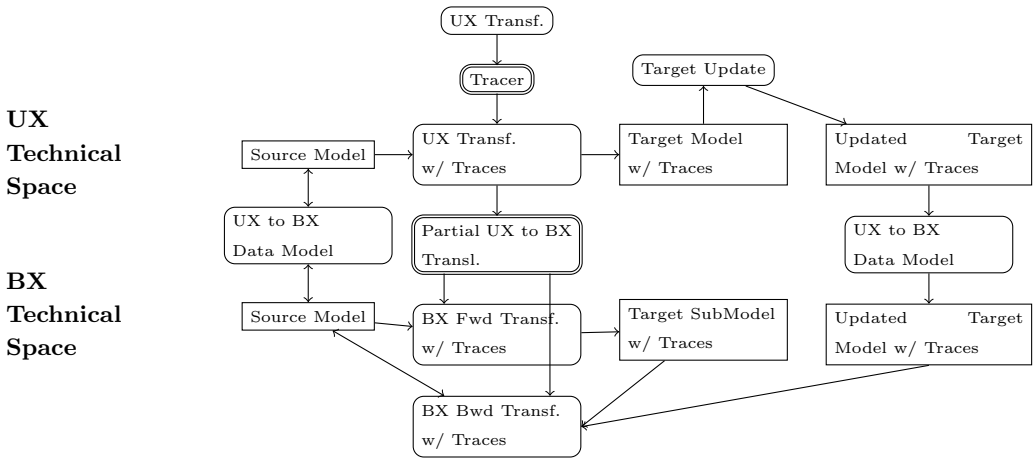


Fig. 5: Overview of the approach (models are rectangles, transformations are rounded rectangles and HOTs are double rounded rectangles)

model in one of the two technical spaces. When the transformation languages have different data models, we need a clearly defined bidirectional transformation (*UX to BX Data Model*) between these data models.

Once the correspondence among the data models is defined, we need a mechanism to align the execution of the two engines. Our solution is the injection of a *stable generation of model element identifiers* within the unidirectional transformation (*UX Transf.*). The generation is loosely-coupled [18] in the sense that the translation logic remains unchanged while the identifier-generation logic is injected transparently via the *Tracer* higher-order transformation (HOT) [30]. The resulting transformation (*UX Transf. w/ Traces*) is executed on the source model to produce a target model with stable traceability information. Stability here means that, given identical input model and transformation, identical identifiers are produced. In our example the *Tracer* HOT augments the rules in Listing 1 with bindings computing element identifiers. Listing 3 shows the result for rule *Class2Table* as it would appear in *UX Transf. w/ Traces*: the binding at line 7 assigns an identifier to the newly created table, computed by concatenating the identifier of the matched Class (`c.__xmiID__`), the name of the rule (`Class2Table`) and the name of the output pattern element (`t`).

The unidirectional transformation is then translated to the bidirectional language by a translation HOT (*Partial UX to BX Transl.*). It is important to note that the trace generation logic embedded in the unidirectional transformation is also translated to the bidirectional transformation language, generating identical model element identifiers (*BX Fwd Transf. w/ Traces*). In this way we are able to retrieve corresponding elements in the two technical spaces by coincident identifiers. In the running example, Listing 4 is translated from Listing 3, and lines 10-11 generate the new id of the model element, being the direct translation of line 7 in Listing 3.

In our example, the correspondence between the Listings 1 and 2 highlights the partiality of the *UX to BX HOT* at both the rule level and the binding level. We assume that ATL rules with complex guards are not translatable in UnQL: e.g., in Listing 2 we have no corresponding clause for the rule *DataType2Type* because the current ATL to UnQL compiler can not translate the OCL operation `allInstances()`. Analogously we assume that complex bindings are not translatable: e.g., the bindings of `key` and `type`, respectively in the rules *Class2Table* and *SingleValuedAttribute2Column*, are not translated to Listing 2, since string comprehension and type introspection is not supported by the current ATL to UnQL compiler.

Given the partiality of the translation, the target model of the forward bidirectional transformation is in general different from the target model of the original unidirectional one. However we impose a strict property to the partial translation: the tar-

get models generated from the bidirectional transformation need to be strictly included in the target models of the unidirectional system. The property will be formally discussed in the next section, but intuitively it aims to provide a clear separation of updatable and not updatable parts in the target model and make the system user-predictable. In the running case, the elements depicted with continuous lines in the right side of Fig. 3 are common to the target models of both the systems. The elements with dashed lines (*Type* elements, *key* references and *type* references) are exclusively produced by the unidirectional language, since the ATL code that produces them is not translated to UnQL (respectively, *DataType2Type* rule, *type* binding, *key* binding). Submodel[†4] matching is guided by the identifiers produced in the previous step. In our current approach the target model editor uses this information to forbid updates to the non-propagable part.

Finally, the cycle in Fig. 5 closes by receiving updates to the propagable part of the target model (*Target Update*), and transforming the updated target model to the bidirectional system (*UX to BX Data Model*). The bidirectional system will then be able to apply backward transformation (*BX Bwd Transf. w/ Traces*) to update the source, by taking into account the source model, target submodel and updated version of the target model, and maintaining the identifier correspondence.

4 Well-Behavedness under Partial Translation

In our approach only a subset of the unidirectional transformation language $\mathcal{L}_{\mathrm{U}}$ is translated, since not all the language constructs can be bidirectionalized due to the limited expressive power of the target language. In this section we formally prove that the global system we described in Fig. 5 is well-behaved [4] if its components respect certain conditions. Intuitively these conditions require that: 1) $\mathcal{L}_{\mathrm{U}}$ respects a modularity property that we call *rule additivity*, 2) we can build a bijective transformation between the data models of $\mathcal{L}_{\mathrm{U}}$ and the underlying bidirectional language $\mathcal{L}_{\mathrm{B}}$, 3) we can build a translator of the part of $\mathcal{L}_{\mathrm{U}}$ to $\mathcal{L}_{\mathrm{B}}$ such that the models produced by transformations in $\mathcal{L}_{\mathrm{B}}$ coincide with the ones produced by corresponding (partial) transformations in $\mathcal{L}_{\mathrm{U}}$, and 4) $\mathcal{L}_{\mathrm{B}}$ is well-behaved. The four conditions for applicability are sufficient for coupling any two transformation systems. Thanks to these properties we are able to reflect updates to the target model of the unidirectional system by passing through the bidirectional system.

In the next section we will present evidence of the practical applicability of the approach by showing that these four properties are satisfied by ATL, GRoundTram and the translation we have built. However the four conditions are satisfied by several transformation systems in model-driven engineering. In particular: 1) Property 1 is satisfied by other popular UX model-transformation languages besides ATL (e.g., Epsilon ETL [21]), 2) Property 2 is easily satisfiable by encoding models in low level representations, like well-known encoding of node-labeled graphs by edge-labeled graphs, 3) given Property 2, Property 3 is satisfiable by design of the translator (this requires a significant effort, in general depending on the chosen languages), 4) Property 4 (or analogous) is common among BX languages [4].

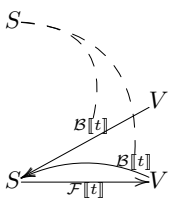
We first recap the notion of well-behavedness [4], that intuitively guarantees that a round-trip in a bidirectional transformation system does not have surprising effects for the user.

Definition 1 (Well-bahavedness)**.** *Given a transformation expression t, the pair of its forward semantics $\mathcal{F}[\![t]\!] : S \to V$ which is a total function from source artifact (e.g., models or graphs) to target artifact, and its backward semantics $\mathcal{B}[\![t]\!] : S \times V \to S$ which is a partial function from target artifact to source artifact, w.r.t. the original source artifact is* well-behaved *if the following two properties hold.*

$$\mathcal{F}[\![t]\!]a_s = a_t \Rightarrow \mathcal{B}[\![t]\!](a_s, a_t) = a_s \qquad \text{(GetPut)}$$

$$\mathcal{B}[\![t]\!](a_s, a'_t) = a'_s \Rightarrow \mathcal{B}[\![t]\!](a_s, \mathcal{F}[\![t]\!]a'_s) = a'_s \qquad \text{(WPutGet)}$$

Diagrammatically, WPutGet is represented as the figue on the right.



Our aim is to achieve this property at the level of model transformation (in $\mathcal{L}_{\mathrm{U}}$) on top of bidirectional graph transfor-

†4 Definition of submodel is given in Sec. 4.

mation (in $\mathcal{L}_{\mathrm{B}}$).
It is worth noting that the original source artifact $a_s \in S$ is needed to recover the information that may had been discarded by the forward transformation [4]. WPutGet [14] corresponds to Weak Invertibility by [6] that is a weaker variant of PutGet [9] that corresponds to Correctness [28].

Under partial translation we first project the transformation T to the one that is fully translated. To clearly distinguish the target of the projected transformation and that of the original transformation, we make the former embedded in the latter. This embedding is represented by the following submodel relation between models.

Definition 2 (Submodel Relation ($\prec_{\mathrm{M}}$)). *$m' \prec_{\mathrm{M}} m$ denotes m' is a submodel of m. This notion of submodel is similar to the inclusion morphism between graphs in general as in [7], extended to typed attributed graphs, in the sense that m_1 is a submodel of m_2 if there is an inclusion morphism from m_1 to m_2.*

Intuitively, bigger models have more model elements and more features for model elements with the same identifiers. The relation $\prec_{\mathrm{M}}$ between models is instantiated for each data model of $\mathcal{L}_{\mathrm{U}}$, for example by using set inclusion relationship between the set of model elements in the models.

We need the following relationship between the transformation $T \in \mathcal{L}_{\mathrm{U}}$ and its translatable part T_0.

Definition 3 (Inclusion Relation of Model Transformations ($\subseteq_{\mathcal{MT}}$)). *$T_0 \subseteq_{\mathcal{MT}} T$ denotes that model transformation $T_0 \in \mathcal{L}_{\mathrm{U}}$ is included in model transformation $T \in \mathcal{L}_{\mathrm{U}}$.*

The relation $\subseteq_{\mathcal{MT}}$ between transformations is instantiated for each model transformation language, for example by using set inclusion relationship between the set of rules in the transformations. The definition of $T_0 \subseteq_{\mathcal{MT}} T$ should reflect the restricted image produced by T_0 relative to T, as below.

Definition 4 (Rule Additivity). *A model transformation language $\mathcal{L}_{\mathrm{U}}$ is* rule additive *for the inclusion relation $\subseteq_{\mathcal{MT}}$ between model transformations of type $MM_{\mathrm{S}} \to MM_{\mathrm{T}}$ in $\mathcal{L}_{\mathrm{U}}$, if for all models $m_s \in MM_{\mathrm{S}}$, $T_0 \subseteq_{\mathcal{MT}} T$ implies $T_0\, m_s \prec_{\mathrm{M}} T\, m_s$.* □

We require that $\mathcal{L}_{\mathrm{U}}$ is rule additive.

Property 1 (Rule additivity of Model Transformation Language). *$\mathcal{L}_{\mathrm{U}}$ is rule additive.*

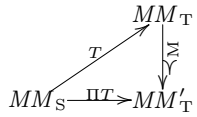
Rules may further contain components like bindings in ATL, in which case the inclusion relation between these bindings are also considered as we instantiate in Sec. 5. 2. Even in such a case, we call this relation rule additivity for simplicity.

Based on the rule additivity, we project the model transformation to produce transformed submodel using the following model transformation projection operation.

Definition 5 (Model Transformation Projection (Π)). *A function $\Pi : \mathcal{L}_{\mathrm{U}} \to \mathcal{L}_{\mathrm{U}}$ is called* model transformation projection *if $\Pi T \subseteq_{\mathcal{MT}} T$ for $T \in \mathcal{L}_{\mathrm{U}}$.*

For a transformation $T \in \mathcal{L}_{\mathrm{U}}$, since $\Pi T \subseteq_{\mathcal{MT}} T$, rule additivity of $\mathcal{L}_{\mathrm{U}}$ implies $\Pi T m \prec_{\mathrm{M}} T m$.

Rule additivity can be represented by the commuting diagram on the right.



Notice the difference between MM_{T} and MM'_{T} (correspondingly GS_{T} and GS'_{T}). Since the models in the latter may lack some model elements or features because of the partiality, the corresponding metamodel should be relaxed to allow the absence of these elements and properties. We do this by changing all the mandatory features to optional (and likewise the lower bound of multiplicity constraints to zero). To avoid clutter we do not make these difference explicit in the rest of this paper.

Next we need a bijective mapping beteen models and graphs, i.e., the pair of model to graph translation ${}^{\mathrm{M}}_{\mathrm{G}}{\downarrow}$ and graph to model translation ${}^{\mathrm{M}}_{\mathrm{G}}{\uparrow}$ to be inverse with each other.

Property 2 (Bijection between Models and Graphs).

$$({}^{\mathrm{M}}_{\mathrm{G}}{\downarrow} \circ {}^{\mathrm{M}}_{\mathrm{G}}{\uparrow})\ g \equiv_{\mathrm{G}} g \qquad ({}^{\mathrm{M}}_{\mathrm{G}}{\uparrow} \circ {}^{\mathrm{M}}_{\mathrm{G}}{\downarrow})\ m \equiv_{\mathrm{M}} m$$

where $\equiv_{\mathrm{G}}$ denotes graph equivalence (isomorphism) and $\equiv_{\mathrm{M}}$ denotes equivalence between models as isomorphism of node-attributed graphs. The graph isomorphism is not easy to decide in general, so we assume a distinguished attribute in the model element to uniquely identify the element, and that the identifier is translated bijectively through ${}^{\mathrm{M}}_{\mathrm{G}}{\downarrow}$ and ${}^{\mathrm{M}}_{\mathrm{G}}{\uparrow}$. In the rest of the paper, we simply use $=$ instead of $\equiv_{\mathrm{M}}$ assuming these identifiers.

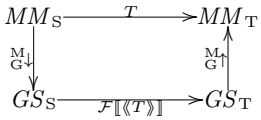
For the transformation produced by projection, we require the soundness of the translation of the transformation: given model transformation

$T : MM_{\mathrm{S}} \to MM_{\mathrm{T}}$, translation $\langle\!\langle \_ \rangle\!\rangle : \mathcal{L}_{\mathrm{U}} \to \mathcal{L}_{\mathrm{B}}$ from model transformation to graph transformation, model to graph translation ${}^{\mathrm{M}}_{\mathrm{G}}\!\downarrow$, graph to model translation ${}^{\mathrm{M}}_{\mathrm{G}}\!\uparrow$, and translated forward graph transformation $\mathcal{F}[\![\langle\!\langle T \rangle\!\rangle]\!] : GS_{\mathrm{S}} \to GS_{\mathrm{T}}$, where GS_{S} and GS_{T} represent source and target graph schema, respectively, we require that if we convert source model to graph, perform the graph transformation, and convert back to model, that is the same as directly transform the source model by model transformation.

Property 3 (Soundness of Total Translation)**.** *Given $T \in \mathcal{L}_{\mathrm{U}}$ and $\langle\!\langle T \rangle\!\rangle \in \mathcal{L}_{\mathrm{B}}$, the following equation holds.*

$$T = {}^{\mathrm{M}}_{\mathrm{G}}\!\uparrow \circ\, \mathcal{F}[\![\langle\!\langle T \rangle\!\rangle]\!] \circ {}^{\mathrm{M}}_{\mathrm{G}}\!\downarrow$$

It can be represented by the diagram on the right. It is worth noting that when T is an identity transformation, it still holds by the second equivalence (${}^{\mathrm{M}}_{\mathrm{G}}\!\downarrow$ being the right inverse of ${}^{\mathrm{M}}_{\mathrm{G}}\!\uparrow$) in Property 2.



Partial translation $\langle\!\langle T \rangle\!\rangle^{\mathrm{P}} (= \langle\!\langle T_0 \rangle\!\rangle)$ is implemented as $\langle\!\langle \Pi T \rangle\!\rangle$. Consequently, the range of Π should be within the domain of $\langle\!\langle \rangle\!\rangle$, i.e.,

$$\mathrm{ran}(\Pi) \subseteq \mathrm{dom}(\langle\!\langle \rangle\!\rangle).$$

Finally, we require that $\mathcal{L}_{\mathrm{B}}$ is well-behaved (Def. 1).

Property 4 (Well-behavedness of underlying bidirectional graph transformation)**.** *Given a graph transformation expression $t \in \mathcal{L}_{\mathrm{B}}$, the pair of its forward semantics $\mathcal{F}[\![t]\!] : GS_{\mathrm{S}} \to GS_{\mathrm{T}}$ and its backward semantics $\mathcal{B}[\![t]\!] : GS_{\mathrm{S}} \times GS_{\mathrm{T}} \to GS_{\mathrm{S}}$ is* well-behaved.

We then implement the bidirectional model transformation. The forward transformation is simply $T \in \mathcal{L}_{\mathrm{U}}$. For backward transformation, we need the following notion of model restriction.

Definition 6 (Model Restriction)**.** Restriction *of model $m_1 \in MM$ with $m_2 \in MM$, denoted by $m_1|_{m_2}$, is a submodel of m_1 induced by the subset of nodes (model elements) in m_1 that have nodes in m_2 with matching identifiers.* □

Since we have $m_1|_{m_2} = m_1' \Rightarrow m_1|_{m_1'} = m_1'$ and $m_2 \prec_{\mathrm{M}} m_1 \Leftrightarrow m_1|_{m_2} = m_2$, we also denote $m_2 \xleftarrow{\prec_{\mathrm{M}}} m_1$ when we compute a restriction from m_1. The restriction $m_1|_{m_2}$ operator is instantiated for each data model of the model transformation language.

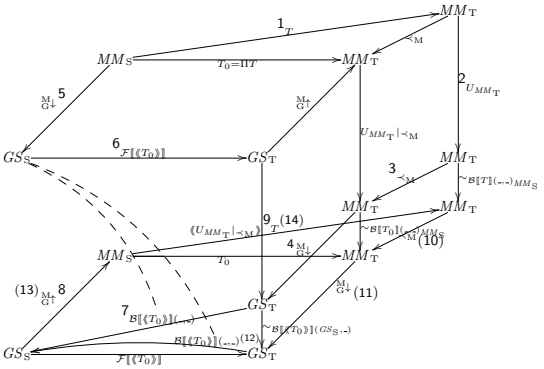


Fig. 6: Commuting diagram for bidirectional transformation based on partial translation of model transformation T

Now we define the backward model transformation.

Definition 7 (Backward Model Transformation)**.** *Given source model $m_s \in MM_{\mathrm{S}}$, updated target model $m_t' \in MM_{\mathrm{T}}$, $T \in \mathcal{L}_{\mathrm{U}}$, its projector Π, model to graph transformation ${}^{\mathrm{M}}_{\mathrm{G}}\!\downarrow$, graph to model transformation ${}^{\mathrm{M}}_{\mathrm{G}}\!\uparrow$, $\langle\!\langle \_ \rangle\!\rangle : \mathcal{L}_{\mathrm{U}} \to \mathcal{L}_{\mathrm{B}}$ and backward graph transformation $\mathcal{B}[\![\_]\!](\_,\_) : GS_{\mathrm{S}} \times GS_{\mathrm{T}} \to GS_{\mathrm{S}}$, the backward model transformation $\mathcal{B}[\![T]\!](\_,\_) : MM_{\mathrm{S}} \times MM_{\mathrm{T}} \to MM_{\mathrm{S}}$ is implemented by*

$$\mathcal{B}[\![T]\!](m_s, m_t') = {}^{\mathrm{M}}_{\mathrm{G}}\!\uparrow(\mathcal{B}[\![\langle\!\langle \Pi T \rangle\!\rangle]\!]({}^{\mathrm{M}}_{\mathrm{G}}\!\downarrow(m_s), {}^{\mathrm{M}}_{\mathrm{G}}\!\downarrow(m_t'|_{(\Pi T)m_s})))$$ □

Given the previous definitions, bidirectional model transformation in the global system can be performed by the concrete sequence of steps shown below.

1. Perform forward model transformation Tm_s to get m_t.
2. Update m_t to m_t' with $m_t' = u\, m_t$, where $u \in U_{MM}$ is an update function on models that conform to metamodel MM.
3. Compute the restriction $m_t'|_{T_0 m_s}$ where $T_0 = \Pi T$.
4. Obtain g_t' by converting the restricted model to graph with ${}^{\mathrm{M}}_{\mathrm{G}}\!\downarrow$.
5. Obtain $g_s = {}^{\mathrm{M}}_{\mathrm{G}}\!\downarrow m_s$.
6. Perform forward graph transformation $\mathcal{F}[\![\langle\!\langle T_0 \rangle\!\rangle]\!] g_s$ to obtain trace information needed for the next step.
7. Perform backward graph transformation $\mathcal{B}[\![\langle\!\langle T_0 \rangle\!\rangle]\!](g_s, g_t')$ to obtain g_s'.
8. Obtain the updated source model m_s' as ${}^{\mathrm{M}}_{\mathrm{G}}\!\uparrow g_s'$.
9. As post processing, perform forward model

transformation Tm'_s to get m''_t as the final updated target model, which may be different from m'_t[†5].

The following lemmas hold on the system built using the backward transformation in Def. 7.

Lemma 1. *Given $\mathcal{L}_{\mathrm{U}}$ and $\mathcal{L}_{\mathrm{B}}$ with Properties 1, 2, 3 and 4, the pair of $T \in \mathcal{L}_{\mathrm{U}}$ and backward model transformation $\mathcal{B}[\![T]\!](\_,\_)$ as defined by Def. 7 satisfies the* GetPut *property.*

Proof. We show $\mathcal{B}[\![T]\!](m_s, T\,m_s) = m_s$.

LHS
= { Def. 7, $T_0 = \Pi T$ }
$\overset{\mathrm{M}}{\mathrm{G}}\!\uparrow(\mathcal{B}[\![\langle\!\langle T_0 \rangle\!\rangle]\!](\overset{\mathrm{M}}{\mathrm{G}}\!\downarrow(m_s), \overset{\mathrm{M}}{\mathrm{G}}\!\downarrow(T\,m_s|_{T_0\,m_s})))$
= { $T_0\,m_s \prec_{\mathrm{M}} T\,m_s$ (Property 1) }
$\overset{\mathrm{M}}{\mathrm{G}}\!\uparrow(\mathcal{B}[\![\langle\!\langle T_0 \rangle\!\rangle]\!](\overset{\mathrm{M}}{\mathrm{G}}\!\downarrow(m_s), \overset{\mathrm{M}}{\mathrm{G}}\!\downarrow(T_0\,m_s)))$
= { $\overset{\mathrm{M}}{\mathrm{G}}\!\downarrow \circ T_0 = \mathcal{F}[\![\langle\!\langle T_0 \rangle\!\rangle]\!] \circ \overset{\mathrm{M}}{\mathrm{G}}\!\downarrow$ (Property 3, Property 2) }
$\overset{\mathrm{M}}{\mathrm{G}}\!\uparrow(\mathcal{B}[\![\langle\!\langle T_0 \rangle\!\rangle]\!](\overset{\mathrm{M}}{\mathrm{G}}\!\downarrow(m_s), \mathcal{F}[\![\langle\!\langle T_0 \rangle\!\rangle]\!]\overset{\mathrm{M}}{\mathrm{G}}\!\downarrow(m_s)))$
= { Property 4(GetPut) }
$\overset{\mathrm{M}}{\mathrm{G}}\!\uparrow(\overset{\mathrm{M}}{\mathrm{G}}\!\downarrow(m_s))$
= { Property 2 }
m_s
= RHS □

Lemma 2. *Given $\mathcal{L}_{\mathrm{U}}$ and $\mathcal{L}_{\mathrm{B}}$ with Properties 1, 2, 3 and 4, the pair of $T \in \mathcal{L}_{\mathrm{U}}$ and backward model transformation $\mathcal{B}[\![T]\!](\_,\_)$ as defined by Def. 7 satisfies the* WPutGet *property.*

The proof is in Figure 7. In this proof, we use shorthands $\downarrow$ and $\uparrow$ for $\overset{\mathrm{M}}{\mathrm{G}}\!\downarrow$ and $\overset{\mathrm{M}}{\mathrm{G}}\!\uparrow$, to avoid clutter. It is now obvious that $\mathcal{L}_{\mathrm{U}}$ on top of $\mathcal{L}_{\mathrm{B}}$ is well-behaved.

Theorem 1 (Well-behavedness of Model Transformation)**.** *Given $\mathcal{L}_{\mathrm{U}}$ and $\mathcal{L}_{\mathrm{B}}$ with Properties 1, 2, 3 and 4, the pair of $T \in \mathcal{L}_{\mathrm{U}}$ and backward transformation $\mathcal{B}[\![T]\!](\_,\_)$ defined by Def. 7 is well behaved.*

Proof. The pair satisfies the GetPut and WPutGet properties by Lemma 1 and Lemma 2. □

Theorem 1 and other related properties can be concisely illustrated by the commuting diagram in Fig. 6 by composing the diagrams we have presented so far. We try to render the same kind of arrows in parallel. In the diagram, the front plane represents the graph technical space, while the back plane represents the model technical space. Artifacts are converted between the two technical spaces by $\overset{\mathrm{M}}{\mathrm{G}}\!\downarrow$ and $\overset{\mathrm{M}}{\mathrm{G}}\!\uparrow$. WPutGet in the graph technical space (Prop.4) can be represented by the commuting triangular diagram at the bottom of the front plane. The dotted arrow from GS_{S} to $\mathcal{B}[\![\langle\!\langle T_0 \rangle\!\rangle]\!](\_,\_)$ indicates that the backward transformation is w.r.t. the original source graph in GS_{S}. The vertical arrow $\sim_{\mathcal{B}[\![\langle\!\langle T_0 \rangle\!\rangle]\!](\_,\_)}$ is an (induced) equivalence relation between the updated target graph and that with another forward transformation, which represents that these two target graphs are equivalent w.r.t. their effect on the source graphs (they result in the same updated source graph). It corresponds to [6]'s equivalence in the context of weak invertibility. The horizontal axis represents the (model and graph) transformation, while the vertical axis represents updates and restrictions. Of the model transformation rules T only $T_0 = \Pi T(\subseteq_{\mathcal{MT}} T)$ can be fully translated, and the commuting triangles in the back plane represent rule additivity (Prop. 1). The square on the top and the bottom represents the soundness of translations (Prop. 3). The vertical arrow on the back bottom labeled $\sim_{\mathcal{B}[\![T_0]\!](\_,\_)}$ is the equivalence relation on the model technical space that is induced by $\sim_{\mathcal{B}[\![\langle\!\langle T_0 \rangle\!\rangle]\!](\_,\_)}$. The update operation $(U_{MM_{\mathrm{T}}}|_{\prec_{\mathrm{M}}})$ represented by the rightmost vertical downward arrow on the back plane assumes that only the submodel part restricted by the image of T_0 is updated, so the square to which the arrow belongs commutes. The square right below also commutes where the equivalence $\sim_{\mathcal{B}[\![T]\!](\_,\_)}$ is induced by $\sim_{\mathcal{B}[\![\langle\!\langle T_0 \rangle\!\rangle]\!](\_,\_)}$.

Each one of the concrete steps listed before is depicted in the diagram as a thick arrow with a sans-serif number. The parenthesized numbers represent the sequence that corresponds to Lemma 2: another backward transformation with respect to the same original source model follows the path on the bottom plane, so, when finished, it returns to

†5 m'_t and m''_t may disagree when forward transformation duplicates part of the source model while user updates only one of the duplicates (handled as explained in the paragraph following Theorem 1 and in footnote 6). We take m''_t as the final updated target model in this case. WPutGet ensures that m'_t and m''_t has the same effect in updating source model.

Proof. We show $\mathcal{B}[\![T]\!](m_s, m_t') = \mathcal{B}[\![T]\!](m_s, T(\mathcal{B}[\![T]\!](m_s, m_t')))$.

$$
\begin{array}{ll}
 & \text{RHS} \\
= & \quad \{ \text{ Def. 7, } T_0 = \Pi T \ \} \\
 & \mathcal{B}[\![T]\!](m_s, T(\uparrow(\mathcal{B}[\![\langle\!\langle T_0 \rangle\!\rangle]\!](\downarrow(m_s), \downarrow(m_t'|_{T_0\, m_s}))))) \\
= & \quad \{ \text{ Def. 7, } T_0 = \Pi T \ \} \\
 & \uparrow(\mathcal{B}[\![\langle\!\langle T_0 \rangle\!\rangle]\!](\downarrow m_s, \\
 & \qquad \downarrow(T(\uparrow(\mathcal{B}[\![\langle\!\langle T_0 \rangle\!\rangle]\!](\downarrow m_s, \downarrow(m_t'|_{T_0\, m_s}))))|_{T_0\, m_s}))) \\
= & \quad \{ \text{ Property 1 } \} \\
 & \uparrow(\mathcal{B}[\![\langle\!\langle T_0 \rangle\!\rangle]\!](\downarrow(m_s), \downarrow(T_0(\uparrow(\mathcal{B}[\![\langle\!\langle T_0 \rangle\!\rangle]\!](\downarrow(m_s), \downarrow(m_t'|_{T_0\, m_s}))))))) \\
= & \quad \left\{ \begin{array}{l} \text{Property 2 and Property 3, i.e.,} \\ \mathcal{F}[\![\langle\!\langle T_0 \rangle\!\rangle]\!] = \downarrow \circ T_0 \circ \uparrow \end{array} \right\} \\
 & \uparrow(\mathcal{B}[\![\langle\!\langle T_0 \rangle\!\rangle]\!](\downarrow(m_s), \mathcal{F}[\![\langle\!\langle T_0 \rangle\!\rangle]\!](\mathcal{B}[\![\langle\!\langle T_0 \rangle\!\rangle]\!](\downarrow(m_s), \downarrow(m_t'|_{T_0\, m_s}))))) \\
= & \quad \{ \text{ Property 4(WPutGet) } \} \\
 & \uparrow(\mathcal{B}[\![\langle\!\langle T_0 \rangle\!\rangle]\!](\downarrow(m_s), \downarrow(m_t'|_{T_0\, m_s}))) \\
= & \quad \{ \text{ Def. 7, } T_0 = \Pi T \ \} \\
 & \text{LHS}
\end{array}
$$

□

Fig. 7: Proof of Lemma 2

the original position in the diagram.

So far we have focused on achieving well-behavedness of bidirectionalized $\mathcal{L}_{\mathrm{U}}$. However, the extreme case where Π produces empty transformation for every input still satisfies the well-behavedness, but the whole system accepts no modification because the target of $\langle\!\langle \Pi T \rangle\!\rangle$ is empty. Similarly, $\langle\!\langle\rangle\!\rangle$ needs to have as large domain as possible. If it is undefined in any input, the whole system would not be bidirectional. So we have the following quantitative requirements, though they are not the focus of our paper.

- dom($\langle\!\langle\rangle\!\rangle$) is maximized
- ran(Π) is maximized (within dom($\langle\!\langle\rangle\!\rangle$))

5 ATL on GRoundTram

In this section we exemplify our proposal by integrating ATL and GRoundTram: we instantiate $\mathcal{L}_{\mathrm{U}}$ as ATL, $\mathcal{L}_{\mathrm{B}}$ as UnQL, models as EMF models conforming to Ecore metamodels serialized in XMI format, graphs as edge-labeled graphs [2] conforming to the KM3 [20] metamodel implemented as graph schema [15], serialized to GraphViz DOT format [14]. We first describe the integration of the two data models (models and edge-labeled graphs) through instantiation of ${}^{\mathrm{M}}_{\mathrm{G}}\!\downarrow$ and ${}^{\mathrm{M}}_{\mathrm{G}}\!\uparrow$ and then provide a partial translation $\langle\!\langle \_ \rangle\!\rangle^{\mathrm{P}}$ from ATL to UnQL.

Then we show that the integrated bidirectional model transformation satisfies well-behavedness (Theorem 1). We have shown in Section 4 that the global well-behavedness as Theorem 1 is implied by Rule Additivity (Property 1) of $\mathcal{L}_{\mathrm{U}}$, bijection between models and graphs (Property 2), soundness of $\langle\!\langle\rangle\!\rangle$ (Property 3), and well-behavedness (Property 4) of $\mathcal{L}_{\mathrm{B}}$. Here we show that these properties hold for this combination of ATL and GRoundTram.

5.1 Integrating Data Models

The ATL (EMF) and GRoundTram data models exhibit some significant semantic gaps.

First, since GRoundTram relies on node identities to store trace information and manage updates to graphs, we can not always freely generate identifiers of graphs in ${}^{\mathrm{M}}_{\mathrm{G}}\!\downarrow$. Therefore, we refine the bijection in Property 2 to a bidirectional transformation, where ${}^{\mathrm{M}}_{\mathrm{G}}\!\downarrow$ takes as input the original graph (i.e., the graph before the ${}^{\mathrm{M}}_{\mathrm{G}}\!\uparrow$ conversion), in addition to the (possibly updated) model. In this bidirectional transformation, ${}^{\mathrm{M}}_{\mathrm{G}}\!\uparrow$ corresponds to forward transformation and ${}^{\mathrm{M}}_{\mathrm{G}}\!\downarrow$ to backward transformation satisfying the following:

$$
\begin{array}{rl}
{}^{\mathrm{M}}_{\mathrm{G}}\!\downarrow\ g\ ({}^{\mathrm{M}}_{\mathrm{G}}\!\uparrow g) = g & \text{(MGGetPut)} \\
({}^{\mathrm{M}}_{\mathrm{G}}\!\uparrow({}^{\mathrm{M}}_{\mathrm{G}}\!\downarrow m)) \equiv_{\mathrm{M}} m & \text{(MGCreateGet)} \\
({}^{\mathrm{M}}_{\mathrm{G}}\!\uparrow({}^{\mathrm{M}}_{\mathrm{G}}\!\downarrow\ g\ m)) \equiv_{\mathrm{M}} m & \text{(MGPutGet)}
\end{array}
$$

MGCreateGet corresponds to the case where the original graph is not available, like in step 5 in the concrete sequence of steps following Definition 7. In this case, node identities in the graph are generated by ${}^{\mathrm{M}}_{\mathrm{G}}\!\downarrow$ arbitrarily but MGCreateGet ensures

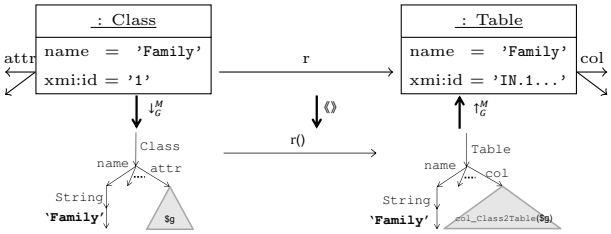


Fig. 8: Overview of ATL to UnQL translation

that the model m is recovered by ${}^{\mathrm{M}}_{\mathrm{G}}\!\uparrow$ using model element identifiers also discussed below.

Since Lemmas 1 and 2 assume bijection before the above refinement, their proofs should be reworked. Fortunately they still hold as follows. In Lemma 1, the first use of Property 2 is replaced by MGGetPut where Property 3 feeds the original source graph for ${}^{\mathrm{M}}_{\mathrm{G}}\!\downarrow$, while the second use is replaced by MGCreateGet. Lemma 2 holds similarly.

The other semantic gap between the two technical spaces is caused by the respective use of isomorphism or bisimulation. Since the GRoundTram technical space uses bisimulation equivalence [2] instead of isomorphism equivalence, the translated UnQL can introduce arbitrary duplicates and unfold of cycles in the graphs. Naively converting these duplicates back to the ATL technical space would hamper the synchronization. Since GRoundTram is equipped with a normalization phase [14] where bisimilar nodes are contracted to one node, this duplication is eliminated. We assumed in Section 3 that all model elements have unique identifiers. Such an identifier is encoded by a primitive-type attribute label within the model-element subgraph. This prevents accidental contraction of subgraphs encoding isomorphic submodels. A graph in the equivalence class $[g]_{\sim \mathrm{G}}$ where $\sim_{\mathrm{G}}$ denotes bisimilarity, maps to the same model as g does, i.e., we have

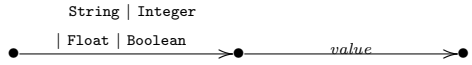
$$g_1 \sim_{\mathrm{G}} g_2 \Rightarrow {}^{\mathrm{M}}_{\mathrm{G}}\!\uparrow(\mathsf{norm}\ g_1) \equiv_{\mathrm{M}} {}^{\mathrm{M}}_{\mathrm{G}}\!\uparrow(\mathsf{norm}\ g_2)$$

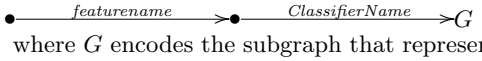
where norm is the normalizer.

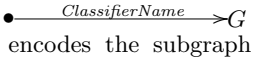
The third gap is based on set semantics of the graph model, so that order between edges is not preserved. Consequently, ordered sequences need additional encoding of the order or introduction of ordered graph models [11] [33].

5.1.1 Encoding Models by UnCAL Graphs

The graph data model we use is rooted and edge-labeled. This means there is no label on nodes. Our graph encoding of models [15] [10] [14] is based on the graph schema KM3 (Kernel MetaMetaModel [20]). An XMI model is converted to graph in the following way.

- **Basic data type:** Datatypes in KM3 are encoded as
  

- **Class data type:** Attributes in KM3 are encoded as
  

  where G encodes the subgraph that represents the feature.
  Classes in KM3 are encoded as graph union of the subgraphs encoding the attributes.
- **Entire Model:** The model encoded in the XMI file is a sequence of top level XML elements encoding top level model elements. These are encoded by a graph union of
  

  where G encodes the subgraph representing the object of the class. Note that only class instances appear at the top level.

Metamodels are used for both ${}^{\mathrm{M}}_{\mathrm{G}}\!\downarrow$ and ${}^{\mathrm{M}}_{\mathrm{G}}\!\uparrow$. ${}^{\mathrm{M}}_{\mathrm{G}}\!\uparrow$ reuses and extends the KM3 validator to extract the structure of the graph, while ${}^{\mathrm{M}}_{\mathrm{G}}\!\downarrow$ traverses the tree structure of the XMI file recursively, keeping track of references identified by `xmi:id` attributes. Please refer to the long version [16] for details on the conversion algorithm.

5.2 Integrating Transformation Languages

The integration of ATL and UnQL is realized by partial translation from ATL to UnQL and alignment of the results of the two transformations. We give the translation algorithm in Section 5.2.1, and

discuss its partiality in Section 5. 2. 2. Comparison with the translation in our previous work [27] is conducted in Section 5. 2. 3.

We first instantiate the relation $\prec_{\mathrm{M}}$ in the ATL (EMF) data model by the following partial order, where a model m is composed of a set of model elements $m.\mathrm{E}$, and each model element $e \in m.\mathrm{E}$ having unique identifier $e.\mathrm{id}$ contains the set $e.\mathrm{F}$ of features. Then,
$m_1 \prec_{\mathrm{M}} m_2 \Leftrightarrow$
$\forall e_1 \in m_1.\mathrm{E}, \exists e_2 \in e_2.\mathrm{E}.\ e_1.\mathrm{id} = e_2.\mathrm{id} \wedge e_1.\mathrm{F} \subseteq e_2.\mathrm{F}.$
$\succ_{\mathrm{M}}$ is instantiated similarly. We consider two features equal if they have the same name and have equal values (in case of primitive attributes) or if they refer to elements having the same identifiers.

Model restriction $m_1|m_2$ is defined if for all model elements $e_2 \in m_2.\mathrm{E}$ there exists $e_1 \in m_1.\mathrm{E}$ with matching identifier and for each feature f in such model element e_2 there exists a feature in e_1 with the same name.

$$m_1|m_2 = \{\mathrm{id} = e.\mathrm{id},\ \mathrm{F} = \{f \mid f \in e.\mathrm{F}, f_2 \in e_2.\mathrm{F}, f.\mathrm{name} = f_2.\mathrm{name}\} \mid e \in m_1.\mathrm{E}, e_2 \in m_2.\mathrm{E}, e.\mathrm{id} = e_2.\mathrm{id}\}$$

We use this restriction under the scenario where model $m_1 \succ_{\mathrm{M}} m_2$ is modified and the modified part of m_1 corresponding to m_2 is extracted. $m_1|m_2 = m_2$ is a special case in which m_1 is not modified.

Next we instantiate the relation $\subseteq_{\mathcal{MT}}$ between ATL transformations defined by the following partial order, where a transformation T is composed of set of rules $T.\mathrm{R}$, each rule $r \in T.\mathrm{R}$ having unique identifier $r.\mathrm{id}$ contains the set $r.\mathrm{O}$ of output patterns, each pattern $p \in r.\mathrm{O}$ having unique identifier $p.\mathrm{id}$ contains the set $p.\mathrm{B}$ of bindings. Then,

$$T_1 \subseteq_{\mathcal{MT}} T_2 \Leftrightarrow \forall r_1 \in T_1.\mathrm{R}, \exists r_2 \in T_2.\mathrm{R}, r_1.\mathrm{id} = r_2.\mathrm{id} \wedge \forall p_1 \in r_1.\mathrm{O}, \exists p_2 \in r_2.\mathrm{O}, p_1.\mathrm{id} = p_2.\mathrm{id} \wedge p_1.\mathrm{B} \subseteq p_2.\mathrm{B}.$$

We consider two bindings equal if they have the same feature name and have syntactically the same binding expressions. $T_1 \supseteq_{\mathcal{MT}} T_2$ is defined similarly. The definition of $T_1 \subseteq_{\mathcal{MT}} T_2$ reflects the restricted image produced by T_1 relative to T_2 with respect to rule, output pattern and binding levels.

5. 2. 1 ATL to UnQL

We then instantiate Π in Def. 5 to generate, from the full ATL transformation, a transformation in the translatable subset of ATL. The sequence of rules in an ATL module is filtered as follows. An ATL rule which has a guard that does not match with the production rule *guard* in Listing 5 (such as a **let** expression in the production rule *xguard*) is discarded. For each ATL rule, its sequence of bindings is filtered as follows. A binding whose OCL expression on the right hand side of `<-` does not match with the production rule *oclExp* in Listing 5 (such as a **let** expression in the production rule *xOclExp*) is discarded. Since these filterings discard each rule and binding in its entirety, this operation is a model transformation projection Π defined in Def. 5, i.e., it is easy to show that $\Pi T \subseteq_{\mathcal{MT}} T$. A brief discussion on this projection is given in Section 5. 2. 2. This projection produces T_0 from T in Fig. 6.

We then fully translate the transformation after projection to UnQL. The range of Π is still a superset of our previous work [27] which did not include guards. The set of ATL rules are translated into structural recursions in UnQL as shown in Fig. 10. The translation rule $\langle\!\langle \_ \rangle\!\rangle_\rho$ for an ATL module named id_{m} with the list *rules* of rules under type environment ρ that includes source and target metamodel is defined in Fig. 9a. A module is translated to a combination of main clauses *MainClauses* of main structural recursion and a list of mutually recursive function definitions *pathfuns* to traverse the path expressions in the bindings. They are generated by an auxiliary translation $\langle\!\langle \rangle\!\rangle^{\mathrm{r}}$ applied for each rule. The last "catch-all" clause following *MainClauses* corresponds to the semantics of ATL in which nothing is produced for omitted rules. $\langle\!\langle \rangle\!\rangle^{\mathrm{r}}$ (Fig. 9b) generates a pair of 1) a corresponding clause in the main function and 2) the list of structural recursion function definitions that are called in the clause as a result of translating OCL expressions in bindings. All the clauses returned by $\langle\!\langle \rangle\!\rangle^{\mathrm{r}}$ constitute the clauses *MainClauses* of the main function, and all the function definitions constitute *pathfuns*. These aggregations are conducted by standard functions on lists `map`, `unzip` and `concat`, and this form of collecting the return value is similar in other auxiliary translations below, while the translation traverses from top level rules to bindings.

Given a guard OCL expression g and the environment ρ, $\langle\!\langle g \rangle\!\rangle^{\mathrm{G}}_\rho$ (Fig. 9c) generates a boolean expression in UnQL. Logical connectives are translated directly, and the predicate e.`isEmpty` that tests whether e produces an empty collection is translated using UnQL's **isEmpty**. Equality test with

$\langle\!\langle \textbf{module}\ id_{\mathrm{m}}\ rules \rangle\!\rangle_{\rho} =$
let $\rho = \rho\{\text{modname} \mapsto id_{\mathrm{m}}\}$ in
let $(MainClauses, pathfuns) =$
  $(\mathsf{id} \times \mathsf{concat}) \circ \mathsf{unzip} \circ \mathsf{map}(\langle\!\langle \_ \rangle\!\rangle^{\mathrm{r}}_{\rho})\ rules$ in
letrec
  pathfuns
and sfun
  MainClauses
| $id_{\mathrm{m}}(\{\$l : \$g\})$ = {}
in $id_{\mathrm{m}}(\$db)$

Main translation rule ($\langle\!\langle\rangle\!\rangle$)

$\langle\!\langle e_1\ \mathsf{and}\ e_2 \rangle\!\rangle^{\mathrm{G}}_{\rho} = \langle\!\langle e_1 \rangle\!\rangle^{\mathrm{G}}_{\rho}\ \textbf{and}\ \langle\!\langle e_2 \rangle\!\rangle^{\mathrm{G}}_{\rho}$
$\langle\!\langle e_1\ \mathsf{or}\ e_2 \rangle\!\rangle^{\mathrm{G}}_{\rho} = \langle\!\langle e_1 \rangle\!\rangle^{\mathrm{G}}_{\rho}\ \textbf{or}\ \langle\!\langle e_2 \rangle\!\rangle^{\mathrm{G}}_{\rho}$
$\langle\!\langle \mathsf{not}\ e \rangle\!\rangle^{\mathrm{G}}_{\rho} = \textbf{not} \langle\!\langle e \rangle\!\rangle^{\mathrm{G}}_{\rho}$
$\langle\!\langle e.\texttt{isEmpty}() \rangle\!\rangle^{\mathrm{G}}_{\rho} = \textbf{isEmpty}(\mathsf{oclExp2select}\ \rho\ e)$
$\langle\!\langle e.f \rangle\!\rangle^{\mathrm{G}}_{\rho} = \textbf{isEmpty}($
  select {`dummy`:{}}
  where {`Boolean`:$g} **in** ($\mathsf{oclExp2select}\ \rho\ e.f$),
  {`true`:$*dummy*} **in** $g)

Translation rule for guards ($\langle\!\langle\rangle\!\rangle^{\mathrm{G}}$)

$\langle\!\langle \textbf{rule}\ id_{\mathrm{r}}\ \textbf{from}\ id_{\mathrm{s}} : oclType_{\mathrm{s}}\ (guard)\ \textbf{to}\ outPats \rangle\!\rangle^{\mathrm{r}}_{\rho}$
$=$ let $\rho = \rho\{id_{\mathrm{s}} \mapsto oclType_{\mathrm{s}}\}\{\text{rulename} \mapsto id_{\mathrm{r}}\}$ in
  let $\rho = \rho\{id_{\mathrm{t}} \mapsto T_{\mathrm{t}} \mid (id_{\mathrm{t}} : T_{\mathrm{t}}\ \_) \in outPats\}$ in
  let $id_{\mathrm{m}} = \rho(\text{modname})$ in
  let $(mbodies, pathfuns) =$
  $(\mathsf{id} \times \mathsf{concat}) \circ \mathsf{unzip} \circ \mathsf{map}(\langle\!\langle \_ \rangle\!\rangle^{\mathrm{op}}_{\rho})\ outPats$
  in $(id_{\mathrm{m}}(\{oclType_{\mathrm{s}} : \$g\})$ =
    let $\$id_{\mathrm{s}}$ = $\$g$ **in**
    if $\langle\!\langle guard \rangle\!\rangle^{\mathrm{G}}_{\rho}$ **then** *mbodies* **else** {}, *pathfuns*)

Translation rule for ATL rules ($\langle\!\langle\rangle\!\rangle^{\mathrm{r}}$)

$\langle\!\langle id_{\mathrm{t}} : oclType_{\mathrm{t}}\ (bindings) \rangle\!\rangle^{\mathrm{op}}_{\rho} =$
let $(mbodies, pathfuns) =$
$(\mathsf{id} \times \mathsf{concat}) \circ \mathsf{unzip} \circ \mathsf{map}(\langle\!\langle \_ \rangle\!\rangle^{\mathrm{b}}_{\rho})\ bindings$
in $(\{oclType_{\mathrm{t}} : mbodies\}, pathfuns)$

$\langle\!\langle id \leftarrow oclExp \rangle\!\rangle^{\mathrm{b}}_{\rho} =$
if isReference *oclExp* then
  $\mathsf{oclExp2sfuns}\ \rho\ (id \leftarrow oclExp)$
else
  $(\{id : \mathsf{oclExp2select}\ \rho\ oclExp\}, [])$

Translation rule for output patterns ($\langle\!\langle\rangle\!\rangle^{\mathrm{op}}$) and bindings ($\langle\!\langle\rangle\!\rangle^{\mathrm{b}}$)

Fig. 10: ATL to UnQL Translation Rules

`OCLUndefined` could be translated similarly. Access to a boolean feature f of a model element is translated to existence of subgraphs encoding the feature. Due to this usage of existence, f should be mandatory with cardinality one. The other predicates like comparison operations would be translated to existence of a combination that satisfies such relation. This is left to our future work. Since the source model element identifier and its type is specified for each rule, this association is registered to the type environment ρ and passed to the auxiliary translation function $\langle\!\langle\rangle\!\rangle^{\mathrm{op}}$. $\langle\!\langle\rangle\!\rangle^{\mathrm{r}}$ generates one clause of the main function which traverses one edge corresponding to $oclType_{\mathrm{s}}$ and binds the subgraph pointed by the edge to variable $\$id_{\mathrm{s}}$. Translation of each output pattern is delegated to $\langle\!\langle\rangle\!\rangle^{\mathrm{op}}$.

The translation $\langle\!\langle\rangle\!\rangle^{\mathrm{op}}$ (Fig. 9d) in turn generates one edge that encodes the output type $oclType_{\mathrm{t}}$, while generation of the body expression of the main function clause and the structural recursion function definition for each binding is delegated to $\langle\!\langle\rangle\!\rangle^{\mathrm{b}}$.

A binding that accesses a non primitive type requires recursion, so the traversal path is translated to a corresponding set of structural recursion definitions that traverse the path. It is similar to the process of translating regular path patterns to mutually recursive structural recursion via the automaton that encodes the pattern in [2]. If instead the binding retrieves primitive data type values, the translation is almost identical to that in [27] that uses **select**.

The above translation generates almost the same UnQL transformation in Listing 2 generated from ATL in Listing 1, except that it is augmented with explicit model element identifier generation as in Listing 4 correspondingly to the id-generation logic like in Listing 3. See the long version [16] for full listings.

5.2.2 Discussions on the partiality of translation and limitations

The expressive power of the ATL language has universality [1], while UnQL is expressed in first-order logic extended with transitive closure [2] albeit its terminating property (any transformation in UnQL terminates by construction and so does in GRoundTram in both forward and backward di-

rection). This gap prohibits total translation from ATL to UnQL. Essentially no fragment that exceeds the expressive power can be translated. For example, in UnQL the recursion scheme of functions cannot be arbitrary, but is subject to a syntactic restriction called structural recursion.

We describe the gap in semantics and expressive power of ATL and UnQL, and the partiality of the translation caused by this gap.

Non-translated ATL features

Language features like rule inheritance and OCL helpers (useful to factorize commonly used OCL expressions) may be in principle bidirectionalized but they are not considered in our current translation. Exploring the exact ATL subset having the same expressive power as UnQL is outside the scope of this paper, we focus instead on the integration problem.

Recursions in OCL

Even though UnQL has recursive functions traversing graphs, the recursion scheme is limited to the form of *structural recursion* function [2] f that satisfies the following equations.

$$\begin{array}{lcl} f\,(\{\}) & = & \{\} \\ f\,(\{l:g\}) & = & (l,g) \odot f(g) \\ f\,(g_1 \cup g_2) & = & f(g_1) \cup f(g_2) \end{array}$$

These equations show that f is like a homomorphism. f returns empty graph for empty inputs (first equation). If the input is an edge labeled l connecting to subgraph g, f returns the final result by combining l and g and the result of the recursive call on g (second equation). If the input is the union of two graphs, then the result is the union of the recursive calls on these graphs (third equation).

The second equation suggests that recursive call is strictly limited to the subgraph (g) that is accessible from the target node of the edge (labeled l). Recursive calls like $f(\{\texttt{a} : \{\texttt{b} : g\}\})$ are disallowed. This restriction guarantees the termination of transformation in UnQL, producing finite graphs.

Moreover, the results of the recursive calls can only be used in constructors, and not passed as arguments to other functions or predicates. So $h(f(g))$ is disallowed. Such composition is only allowed outside of recursive calls. It is also worth noting that it is not permitted to call the outer functions from their nested inner functions. Therefore, an OCL expression containing recursions that do not fit into the above recursion scheme is not translatable.

Multiple traversals from a node

In ATL, model elements are traversed multiple times, e.g., in output-pattern bindings as in the following one from Listing 1 line 34.

```
name <- s.owner.name+'_'+s.name
```

As long as these references access the values of primitive data types, they can be encoded as a single **select** construct, like

```
1 select {name:{String: {($s1^ '_' ^ $s2):{}}}}
2 where  {owner.name.String:{$s1:$dummy}} in $g
       ,
3          {name.String:       {$s2:$dummy}} in $g
```

This translation performs multiple traversals from the same graph variables using multiple **where** clauses. If the traversed values are not primitive, then this translation cannot be used, because beyond the complex values, structural recursion should be applied. So the traversal should be implemented using sibling **sfun**s, the returned value from which cannot be traversed because of the limitation on structural recursions.

5.2.3 Improvements from previous translation

Compared with the previous work [27], we have the following differences: (1) We deal with partiality, so we cover non-restricted ATL at the user level. Previously total translation from ATL to UnQL was assumed, so the source ATL language was restricted to a bidirectionalizable subset. (2) We have a bigger translated subset, including guards. (3) The translated UnQL is more robust in the sense that cycles are better maintained, translation is metamodel-directed, and graph-model transformation is clearly defined based on XMI. See the long version [16] for details.

5.3 Well-behavedness of the entire bidirectional model transformation

As we wrote in the beginning of this section, we show that the integrated bidirectional model transformation satisfies well-behavedness (Theorem 1) by showing Properties 1,2, 3 and 4. We show that these properties hold for this combination of ATL and GRoundTram.

Rule Additivity of ATL (Property 1)

Rule additivity can be derived by the instantiation of $\prec_{\mathrm{M}}$ and $\subseteq_{\mathcal{MT}}$ in Section 5. 2 and informal specifications of ATL, and we just assume that it holds in this paper. Intuitively, ATL[†6] satisfies rule additivity because the more rules the transformation have, and the more output-pattern elements the rule has, the more target elements are included in the target model, and the more bindings each output pattern has, the more features are included in the target model elements. Note that for each source element, no more than one rule is applied. If more than one rule is applicable to a source element, then a run-time error occurs. If no rule is applicable to a source element, then no target element is produced from the source element.

Bijection between models and graphs (Property 2)

We have already shown this property by refinement to bidirectional transformation between models and graphs in Section 5. 1.

Soundness of ATL to UnQL Translation (Property 3)

We show that the target model created by GRoundTram through forward transformation of translated UnQL is isomorphic to the target model created by the ATL transformation. Fig. 8 sketches the proof using our running case introduced in Section 2. For each rule r matching a source model element e_s, ATL creates a target model element e_t. In the figure e_s corresponds to `Family` model element of type Class and e_t corresponds to `Family` of type Table. Suppose the entire rule is translated to UnQL. Then the corresponding UnQL structural recursion function $r()$ creates the corresponding target model, assuming each binding is successfully translated. In the figure, the `name` feature is encoded in the source graph as three consecutive edges as described in Section 5. 1, and the rule *Class2Table* is translated to the clause of the UnQL function as shown in lines 20–22 in Listing 2. The transformation of the feature `attr` is delegated to the `col_Class2Table` sibling mutually recursive function. Note that rule additivity of ATL is translated to that of the translated UnQL code in the sense that when ATL rule is added, the corresponding clause in the main **sfun** of UnQL is added by the definition of translation rule $\langle\!\langle \_ \rangle\!\rangle$. Similarly, the more *outPat* we have in an ATL rule, the more components in the **then** clause is generated by the translation rule $\langle\!\langle \rangle\!\rangle^{\mathrm{r}}$ and, the more *bindings* we have in an output pattern, the more branches under the edge encoding $oclType_{\mathrm{t}}$ are generated by the translation rule $\langle\!\langle \rangle\!\rangle^{\mathrm{op}}$.

As for the translation of guards ($\langle\!\langle \rangle\!\rangle^{\mathrm{G}}$), we show that an enclosing guard holds if and only if the translated UnQL expression using **isEmpty** predicate holds. Since logical connectives can be treated by simple induction, we focus on the rest of the two cases, both of which uses UnQL's **isEmpty** predicate with oclExp2select auxiliary translation rules. OCL expression e.`isEmpty`() evaluates true if the expression exhibits non-empty collection. UnQL translation of e generates corresponding graph with **select** expression which returns dummy graph fragment if the translation of e captures the graph pattern that corresponds to patterns in the OCL expression. Example can be seen in lines 15-19 and 24-30 of Listing 2. OCL Boolean predicate expressions can be treated similarly.

By the definition of translation rules, when we have atarget model element created by an ATL rule, we have the corresponding clause of main UnQL **sfun** in GRoundTram that creates the corresponding target element. The correspondence is identified by the model element identifier `xmi:id` discussed in Section 3.

Formally, the above correspondence can be shown by induction on the number of rules, number of output patterns under the hypothesis on the succesfull translation of the bindings. Base case with no rule ($rules = []$) produces just the catch-all clause with body producing empty graph, since $(\mathsf{id} \times \mathsf{concat}) \circ \mathsf{unzip} \circ \mathsf{map}(\langle\!\langle \rangle\!\rangle^{\mathrm{r}})$ produces the pair of empty lists for the empty input list. So both ATL and GRoundTram $({}^{\mathrm{M}}_{\mathrm{G}}\uparrow \circ\ \mathcal{F}[\![\langle\!\langle T \rangle\!\rangle]\!] \circ {}^{\mathrm{M}}_{\mathrm{G}}\downarrow)$ produce an empty model. As for the base case of $\langle\!\langle \rangle\!\rangle^{\mathrm{r}}$ with empty output patterns, it still produces no element since *mbodies* and *pathfuns* are both empty in $\langle\!\langle \rangle\!\rangle^{\mathrm{r}}$ for empty *outPats*. As for the base case of $\langle\!\langle \rangle\!\rangle^{\mathrm{op}}$ with empty bindings, it produces the empty skeleton of a model element of type $oclType_{\mathrm{t}}$, since *mbodies* and *pathfuns* are both empty in $\langle\!\langle \rangle\!\rangle^{\mathrm{op}}$ for

†6 In this paper the term *ATL* refers to the rule-based declarative transformation language. The ATL engine supports also the execution of imperative instructions that we do not consider.

empty *bindings*. Inductive case assumes translations of rules, output patterns and bindings as induction hypothesis and shows that a rule, an output pattern or a binding additionally produces the corresponding element or binding. For each rule, the matching of source element type captured by the **from** clause is translated as **sfun**'s pattern match $(\{oclType_{\mathrm{s}} : \$g\})$, and for each output pattern, an edge representing the output OCL type of the produced element is produced by $\langle\!\langle\rangle\!\rangle^{\mathrm{op}}$ at $\{oclType_{\mathrm{t}} : mbodies\}$. For each binding, the corresponding feature name given by *id* in the rule of $\langle\!\langle\rangle\!\rangle^{\mathrm{b}}$ is used to create an edge labeled with *id* encoding that feature name as $\{id : \mathsf{oclExp2select}\ \rho\ oclExp\}$ in case of primitive values. These edges are translated to the corresponding elements and attributes in XMI format by $\overset{\mathrm{M}}{\mathrm{G}}\!\uparrow$ instantiated in Section 5. 1.

Next we show that if we have a reference from element e_1 in the target model created by ATL to another element e_2 in the same model, then in the target graph created via GRoundTram we have a corresponding sequence of edges from a node encoding e_1 to another node encoding e_2. It can be shown by the definition of translation rule $\langle\!\langle\rangle\!\rangle^{\mathrm{b}}$ in which we have the corresponding binding $id \leftarrow oclExp$ that creates reference named *id* in the target model of ATL. Mutually recursive functions that finally calls the main function of name id_{m} will produce such edges.

Well-behavedness of GRoundTram (Property 4)

This property is already shown in [14] [12].

Since we have shown all the properties necessary to deduce well-behavedness, we can conclude that the entire bidirectional model transformation system exhibits well-behavedness. □

As for the partiality, ATL language constructs whose translation is not defined produce empty subgraphs. For example, for an ATL rule whose guard has no matching translation, the entire rule is discarded by Π before $\langle\!\langle\rangle\!\rangle$ is applied, which is captured by the catch-all clause like in Listing 4. At the binding level, the model element of type `Class` with `name` attribute `Person` will create `key` attribute of `Table` element in the ATL transformation, but the corresponding `key` binding is discarded by Π before $\langle\!\langle\rangle\!\rangle$ is applied as we exemplified in Section 2. 3. This will cause the absence of `key` edge in the target graph and that of the target model generated from the target graph (not depicted in Fig. 8).

6 Related Work

Bidirectionalization [25] [31] [22] [23] allows users to write one direction of the transformation by using the unidirectional transformation language they are used to. The language is overloaded with a backward semantics that enables to propagate in the backward direction a set of changes to the target. Full bidirectionalization of the unidirectional model transformation is in general not possible without imposing restrictions to the forward transformation language, and existing work [27] is mainly focused on maximizing the part of the unidirectional language that is bidirectionalized.

[27] fully bidirectionalized a small (e.g., without guards) subset of ATL by translating it to bidirectionalized UnQL graph transformation [13] [14]. Since the translation itself was total, there was no need to run the ATL engine. On the other hand, our partiality supports the untranslatable part of the language in the forward direction, requiring the image of the complement to be created by the unidirectional engine. Hence we run both the unidirectional and bidirectional engines in parallel and combine the results. With respect to the translation, our improvements over Sasano et al. were already summarized in Section 5. 2. 3. See also our long version [16] for technical details. As for correctness, Sasano et al did not formalize the overall bidirectional (round-tripping) property as we did in this paper.

The idea of partially translating programs has been studied, for example by [26] and [24]. Magnusson's objective is to accelerate the execution of a sequential program by partially translating it to another intermediate representation, and the rest runs ordinarily. The control is passed between the two representations, so there is no overlap between the executions of the two representations. Performance improvement also motivated [26], where the dynamic nature of the source language semantics prevented its total compilation before execution. In our setting, the original and the translated program run concurrently and the results are merged. Our objective is to bidirectionalize the source program, and we have provided an integration framework, in

addition to a translation, between these two programs.

Our partial translation may be considered as an abstraction so that work using Galois Connection such as by [3] may be utilized to simplify parts of our discussions in Sections 4 and 5. 3. Note that Galois Connection is not directly applicable as our backward transformation takes the original source as well as the updated target.

Recent study by Stevens [29] mentions partial transformation with a different meaning, since the partiality refers to the idea of tolerating imperfect consistency between source and target. Her notion of subspace is interesting; subspace is a subset of the domain or codomain of the transformation in which the user agreed to stay during update propagation. The subspace also characterizes the degree of consistency in that an element within a subspace is more/less compatible than the others outside the subspace. In our setting, the target subspace consists of the subset of the codomain covered by target models within which only the submodel created by the partially translated transformation is modified, because only the submodel is allowed to be updated, though we do not use this subspace to characterize the degree of consistency. We disallow the modification outside of the subspace – Steven's partiality is not tolerated[†7].

7 Conclusion and Future Work

We proposed to bidirectionalize unidirectional model transformation languages by partial compilation to bidirectional languages and coupled execution of the two language engines. Users can write the transformation in the unidirectional languages they are used to, while updates on well-separated subregions of the target models are back-propagated by the bidirectional engine. We proved that the well-behavedness of the global system can be inferred from the well-behavedness of the bidirectional engine.

We applied our approach to combine ATL and GRoundTram, but any other combination satisfying the properties described in this paper is possible. Application of the approach to new languages requires to develop a partial mapping from the unidirectional language to the bidirectional one. This requires in general a significant effort, depending on the chosen languages. The developer needs to clearly identify the translatable fragment and totally translate it, while translating the rest to *null* transformation (which generates nothing). A small datamodel gap simplifies the translation. As a special case, a bidirectional engine with the PutGet, which is stronger than WPutGet, would make the whole system satisfy PutGet.

In future work, we plan to extend the translatable part of ATL, experiment with different pairs of languages and use the properties of the approach to provide user-friendly feedback in model editors for bidirectional systems.

Acknowledgments

We thank Frédéric Jouault, Zheng Cheng and the members of IPL for their valuable comments and suggestions. This research is funded by the National Institute of Informatics and JSPS KAKENHI Grant 26330096 and by the MONDO (EU ICT-611125) project.

†7 The WPutGet property itself does tolerate imperfect consistency though, if the transformation duplicates (part of) model elements. To be perfectly consistent, these duplicates are simultaneously updated to the identical values, but the backward transformation accepts (tolerates) the update to identical values on only a subset of these duplicates.

References

[1] Al-Sibahi, A. S.: On the Computational Expressiveness of Model Transformation Languages, *ITU Technical Report Series*, (2015).

[2] Buneman, P., Fernandez, M. F., and Suciu, D.: UnQL: a query language and algebra for semistructured data based on structural recursion, *VLDB Journal: Very Large Data Bases*, Vol. 9, No. 1(2000), pp. 76–110.

[3] Cousot, P. and Cousot, R.: Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation, *PLILP '92*, 1992, pp. 269–295.

[4] Czarnecki, K., Foster, J. N., Hu, Z., Lämmel, R., Schürr, A., and Terwilliger, J. F.: Bidirectional Transformations: A Cross-Discipline Perspective, *ICMT'09*, 2009, pp. 260–283.

[5] Czarnecki, K. and Helsen, S.: Feature-based survey of model transformation Approaches, *IBM Syst. J.*, Vol. 45, No. 3(2006), pp. 621–645.

[6] Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., and Orejas, F.: From State- to Delta-Based Bidirectional Model Transformations: The

Symmetric Case, *MODELS'11*, 2011, pp. 304–318.

[7] Ehrig, H., Ehrig, K., Prange, U., and Taentzer, G.: *Fundamentals of Algebraic Graph Transformation*, Springer-Verlag, 2006.

[8] Eramo, R., Pierantonio, A., and Rosa, G.: Uncertainty in Bidirectional Transformations, *Proc. 6th International Workshop on Modeling in Software Engineering (MiSE)*, ACM, 2014, pp. 37–42.

[9] Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem, *ACM Trans. Program. Lang. Syst.*, Vol. 29, No. 3(2007).

[10] Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., and Nakano, K.: GRoundTram Version 0.9.3 User Manual, `http://www.biglab.org/download.html`, 2012.

[11] Hidaka, S., Asada, K., Hu, Z., Kato, H., and Nakano, K.: Structural Recursion for Querying Ordered Graphs, *ICFP'13*, September 2013, pp. 305–318.

[12] Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., and Nakano, K.: Bidirectionalizing graph transformations, *ICFP'10*, ACM, 2010, pp. 205–216.

[13] Hidaka, S., Hu, Z., Inaba, K., Kato, H., and Nakano, K.: GRoundTram: An Integrated Framework for Developing Well-Behaved Bidirectional Model Transformations (short paper), *26th IEEE/ACM International Conference On Automated Software Engineering*, IEEE, 2011, pp. 480–483.

[14] Hidaka, S., Hu, Z., Inaba, K., Kato, H., and Nakano, K.: GRoundTram: an integrated framework for developing well-behaved bidirectional model transformations, *Progress in Informatics*, No. 10(2013), pp. 131–148.

[15] Hidaka, S., Hu, Z., Kato, H., and Nakano, K.: Towards a compositional approach to model transformation for software development, *SAC'09: Proceedings of the 2009 ACM symposium on Applied Computing*, New York, NY, USA, ACM, 2009, pp. 468–475.

[16] Hidaka, S. and Tisi, M.: ATLGT: bidirectional ATL on top of GRoundTram, `https://github.com/atlanmod/ATLGT`, 2015. Accessed: 2015-05-15.

[17] Hidaka, S., Tisi, M., Cabot, J., and Hu, Z.: Feature-based classification of bidirectional transformation approaches, *Software & Systems Modeling*, (2015), pp. 1–22.

[18] Jouault, F.: Loosely coupled traceability for ATL, *ECMDA Traceability Workshop (ECMDA-TW)*, 2005, pp. 29–37.

[19] Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I.: ATL: A model transformation tool, *Sci. Comput. Program.*, Vol. 72, No. 1–2(2008), pp. 31–39.

[20] Jouault, F. and Bézivin, J.: KM3: a DSL for metamodel specification, *8th IFIP international conference on formal methods for open object-based distributed systems*, LNCS, Vol. 4037, 2006, pp. 171–185.

[21] Kolovos, D. S., Paige, R. F., and Polack, F. A.: The epsilon transformation language, *Theory and practice of model transformations*, Springer, 2008, pp. 46–60.

[22] Liu, D., Hu, Z., and Takeichi, M.: Bidirectional Interpretation of XQuery, *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '07, New York, NY, USA, ACM, 2007, pp. 21–30.

[23] Liu, D., Hu, Z., and Takeichi, M.: An expressive bidirectional transformation language for XQuery view update, *Progress in Informatics*, No. 10(2013), pp. 89–129.

[24] Magnusson, P.: Partial Translation, Technical Report T93:05, The Swedish Institute of Computer Science, 1993.

[25] Matsuda, K., Hu, Z., Nakano, K., Hamana, M., and Takeichi, M.: Bidirectionalization Transformation based on Automatic Derivation of View Complement Functions, *ICFP 2007*, 2007, pp. 47–58.

[26] Pinter, R. Y., Vortman, P., and Weiss, Z.: Partial Compilation of REXX, *IBM Systems Journal*, Vol. 30, No. 3(1991), pp. 312–321.

[27] Sasano, I., Hu, Z., Hidaka, S., Inaba, K., Kato, H., and Nakano, K.: Toward Bidirectionalization of ATL with GRoundTram, *ICMT*, LNCS, Vol. 6707, Springer, 2011, pp. 138–151.

[28] Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions, *Software and System Modeling*, Vol. 9, No. 1(2010), pp. 7–20.

[29] Stevens, P.: Bidirectionally Tolerating Inconsistency: Partial Transformations, *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering - Volume 8411*, New York, NY, USA, Springer-Verlag New York, Inc., 2014, pp. 32–46.

[30] Tisi, M., Jouault, F., Fraternali, P., Ceri, S., and Bézivin, J.: On the Use of Higher-Order Model Transformations, *ECMDA-FA'09*, LNCS, Vol. 5562, 2009, pp. 18–33.

[31] Voigtländer, J.: Bidirectionalization for free! (Pearl), *POPL '09: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA, ACM, 2009, pp. 165–176.

[32] Warmer, J. B. and Kleppe, A. G.: *The Object Constraint Language: Precise Modeling With UML*, Addison-Wesley Professional, 1998.

[33] Yang, F. and Hidaka, S.: Bidirectional Transformation on Ordered Graphs, Technical Report 2015-08, GRACE Center, 2015.

```
rule Class2Table {
  from
    c:ClassDiagram!Class
  to
    t:Relational!Table
     (__xmiID__ <-
       'IN'.concat(c.__xmiID__.concat('.Class2Table.t')),
      name<-c.name, col<-c.attr, owner<-c.owner,
      key<-c.attr->select(a | a.name.endsWith('Id')))
}
```

Listing 3: Class2Table with identifier injection in ATL

```
select sfun
    col_Class2Table({attr:$g}) = {col: Class2Relational($g)}
  | col_Class2Table({$l:$g}) = {}
and sfun owner_Class2Table({owner:$g})
    = {owner:Class2Relational($g)}
  | owner_Class2Table({$l:$g}) = {}
and sfun (* main function *)
  |  Class2Relational({Class:$g})          =
    {Table:(
      (select {xmi_id:{String:{("IN"^$id^".Class2Table.t"):{}}}}
       where {xmi_id._:{$id:$dummy}} in $g)
    ∪ (select {name:String:$g} where {name._:$g} in $g)
    ∪  col_Class2Table($g) ∪owner_Class2Table($g))}
  |  Class2Relational({$l:$g})          = {}
  in Class2Relational($db)
```

Listing 4: Class2Table with identifier injection in UnQL

```
ATL = module id; create id:id from id:id; r+
        r = rule id from inPat to outPat+
    inPat = id:oclType xguard
   xguard = guard | let id:ocltype = xOclExp in xOclExp
          | id:id(...) | xOclExp.xOclExp->xOclExp | ...
    guard = oclExp.isEmpty() | oclExp.id
          | not guard | guard and guard
          | guard or guard
   outPat = id:oclType binding*
  binding = id ← xOclExp
  xOclExp = oclExp | let id:ocltype = xOclExp in xOclExp
          | id:id(...) | xOclExp.xOclExp->xOclExp | ...
   oclExp = id
          | oclExp.id
          | string
          | oclExp + oclExp
```

Listing 5: ATL syntax given to projection Π