Automated Inversion of Attribute Mappings in Bidirectional Model Transformations

Max E. Kramer
Karlsruhe Institute of Technology
max.e.kramer@kit.edu

Kirill Rakhman
Green Parrot GmbH
kirill.rakhman@gmail.com

Abstract

Bidirectional model transformations create or update a target model according to a base model and vice versa using a single transformation specification for both directions. Triple graph grammars, for example, define which model elements shall exist and how they should reference each other without repeating information for both directions. They can also copy values of simple-typed attributes, such as enumerations, strings, or numbers. But currently only the identity operator can be easily specified in bidirectional transformation languages: Other attribute mappings either have to be specified for both directions or with a special constraint language. In this paper, we present an approach that inverts attribute transformation expressions that can be written in a simple Java-like syntax. We also present an initial library of 30 operator-specific inverters that result in well-behaved view-update round-trips (GetPut) for all changes. For changes for which well-behaved update-view round-trips (PutGet) are impossible, we chose inversions that sustain as much information as possible. We realized our inversion approach for a prototypical transformation language that generates Java code. An evaluation using all 103 transformations of the ATL zoo shows that 26% of the LLOC of all non-trivial attribute transformation expressions could be inverted with our initial inverters. This may indicate that many transformation tasks could involve non-trivial attribute transformations that can easily be specified and inverted with our approach.

1 Introduction and Motivation

Bidirectional model transformations [Ste08] support two transformation directions: in forward mode, new or existing target models are created or updated according to source models, which are provided

In: A. Anjorin, J. Gibbons (eds.): Proceedings of the Fifth International Workshop on Bidirectional Transformations (Bx 2016), Eindhoven, The Netherlands, April 8, 2016, published at http://ceur-ws.org

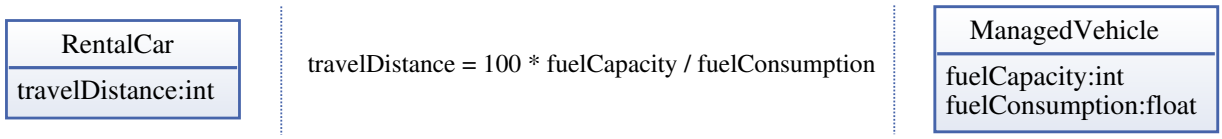

Figure 1: Two exemplary metamodels for cars and a simple attribute transformation expression

as input; in backward mode, source models are created or updated according to target models. The advantage of bidirectional model transformation languages and engines is that a single transformation specification can be executed for both directions. This can be less error-prone and less tedious than manually specifying both transformation directions with redundant parts.

Successful approaches for bidirectional transformations, such as triple graph grammars (TGGs) [GW09], can be used to define many but not all necessary computations in a bidirectional way: It is possible to define bidirectionally which model elements have to be created, how complex-typed references between them shall be established, and which values shall be assigned to simple-typed attributes of the model elements. Bidirectional specifications of attribute assignments in TGGs are, however, currently either restricted to identity mappings [Hil+13] or to special constraint languages [AVS12]: This means the engines can either only copy unchanged values or process expressions that use constraints for which forward and backward operations are provided. The relations between the attributes of model elements that are transformed with bidirectional languages cannot always be expressed with identity mappings, but using a separate constraint language and always defining specific constraint operations can introduce accidental complexity.

Consider, as a running example, two metamodels that could be used to model the promotion and management of rental cars as depicted in Figure 1: The metamodel used to promote cars for customers has a single attribute to represent the travel distance that can be covered with a full tank. The other metamodel, which is used to manage all vehicles internally, has two separate attributes for the fuel capacity of the tank and for the average fuel consumption per 100 km. In a bidirectional model transformation an attribute mapping `travelDistance = 100 * fuelCapacity / fuelConsumption` should not only be used to set or update the attribute of instances of the first metamodel based on the attributes of instances of the second metamodel but also the other way round.

With current approaches for bidirectional transformations, a transformation developer cannot easily specify such a mapping in a bidirectional way. Either separate unidirectional operations to calculate the attribute values in forward or backward mode have to be written. Or the mapping would have to be expressed using constraints, e.g. `divide(fuelCapacity,fuelConsumption,t1)`, `multiply(t1, 100, t2)`, and `floatToInt(travelDistance,t2)`. In the first case the developer has to ensure manually that both operations together fulfill round-trip properties in order not to ruin the bidirectional properties of his transformation and he has to use explicit type casts in both directions. In the second case the developer hast to ensure this for the atomic forward and backward operations that he has to be provide for `divide`, `multiply`, and `floatToInt`. This can be avoided with a transformation language that automatically derives transformations for both directions from attribute mappings that are expressed in a well-known syntax using a library of expression inverters. This can be particularly important if the correctness of bidirectional transformations has to be verified, e.g. in order to meet security requirements.

In this paper, we present two contributions: an approach for the automated inversion of attribute mappings and an extensible library of 30 operator-specific inverters for a Java-like expression language. These composable inverters can be used to obtain an inverse attribute assignment for a backward transformation when given an attribute assignment of a forward transformation. The obtained inverted expression and the original expression always fulfill the GetPut law, which demands that we obtain identical values after a round-trip that starts in forward mode [Fos+07]. The PutGet law, which demands identical values after a round-trip that starts in backward mode, is fulfilled whenever this is possible.

If a violation of PutGet cannot be avoided, the inverters yield backward expressions that minimize the cases of violations and sustain as much original information as possible. The inversion approach is not more powerful than the constraint-based programming approach of Anjorin et al. [AVS12] but the language for attribute mappings could be easier to use because only one transformation direction has to be specified. Our library of inverters could also be used to ship a set of constraints with already defined forward and backward operations with this constraint-based approach of Anjorin et al.

We implemented a prototype for our approach[1], which provides instant feedback to support developers in writing invertible mappings. Invertible forward attribute transformations can be written in a language that is based on the expression language *Xbase* [Eff+12] and almost identical to a subset of Java. For every forward transformation, our prototype generates a backward transformation that takes values for all attributes that appear in the forward transformation as input and outputs an updated value. The inversion procedure is limited to expressions that mention a metaclass attribute at most once and we do not present inverters that update more than one source attribute.

We evaluated how often attribute transformation expressions that are supported by our initial inverter library appear both, on the left-hand side and the right-hand side of expressions in all 103 transformations of the ATL zoo[2]. 55% of the LLOC of all attribute transformation expressions including the trivial identity operator and 26% of the LLOC of all non-trivial transformation expressions in these transformations use operations for which we present inverters. This may indicate that not only such unidirectional transformations but also bidirectional transformations involve many non-trivial attribute mapping expressions that can be easily expressed and inverted with our approach. Further research is, however, needed to investigate whether the inverters would indeed yield inverse transformations that meet all requirements.

The rest of this paper, which is based on a thesis of one of the authors [Rak15] and complemented by a technical report [KR16], is structured as follows: In section 2, we present the foundations of our approach. In section 3, we discuss related work. In section 4, we introduce our inversion approach and in section 5, we explain the individual inverters. In section 6, we discuss formal properties and an evaluation of the applicability. In section 7, we draw some final conclusions and mention future work.

2 Background and Foundations

In this section we present the languages and laws on which our inversion approach is based.

2.1 Essential Meta Object Facility (EMOF)

The Object Management Group (OMG) developed a metamodeling language called Meta Object Facility (MOF) (ISO 19508:2014). A subset of it is called Essential Meta Object Facility (EMOF) and defines core concepts, such as metaclasses with properties and operations. The Ecore language is used in the Eclipse Modeling Framework (EMF) to define metamodels. It is closely aligned with EMOF but distinguishes complex-typed and simple-typed properties: Associations that link to metaclass instances are called *references* and properties that take simple values such as enumerations, strings, or numbers are called *attributes*. In order to simplify formulations, we use this terminology throughout this paper.

2.2 Round-Trip Laws for Bidirectional Transformations

There are several definitions for laws of bidirectional transformations that guarantee properties for round-trips that combine forward and backward direction. For our inversion approach we chose the well-known GetPut and PutGet laws that were formulated for lenses by Foster et al. [Fos+07]. For our

[1]The language and inverters are available as open-source: sdqweb.ipd.kit.edu/wiki/Attribute_Mapping_Inversion
[2]ATL Transformations Zoo: eclipse.org/atl/atlTransformations

special setting of attribute assignment expressions an operator op and its inverse operator op^{-1} fulfill the GetPut law if the subsequent application of op (get) and op^{-1} (put) always yields the same value:

$$op^{-1}(op(s), s) = s, \text{ for all source values } s \quad \text{(GetPut)}$$

Similar the operator and inverse operator fulfill the PutGet law if the subsequent application of op^{-1} (put) and op (get) always yields the same value:

$$op(op^{-1}(t, s)) = t, \text{ for all target values } t \text{ and all source values } s \quad \text{(PutGet)}$$

3 Related Work

The problem of inverting programs in general [Dij79] and complex attribute expressions of bidirectional model transformations in particular [AVS12] is clearly stated in the literature. Many approaches successfully invert such attribute transformations for special languages [Mat+07; YG07; Boh+08]. All these approaches have very strong properties but it is not possible to express bidirectional attribute transformations as easily as with popular unidirectional languages such as ATL or QVT-O: Numbers, for example, cannot be encoded [Mat+07], have to be positive [YG07], or are simply not in the focus [Boh+08]. Other approaches restrict the usage of transformed models by forbidding changes outside the transformation engine [Xio+07].

In the introduction, we have already mentioned the approach by Anjorin et al. [AVS12] for the inversion of attribute transformations using constraint-based programming. In contrast to our approach a special constraint language has to be used instead of a well-known expression syntax. Furthermore, there is no library of predefined constraint inverters: a forward and backward operation has to be provided for each atomic constraint.

4 Mapping Inversion Approach

In this section, we first introduce our general approach for inverting attribute mapping expressions. Then, we present our initial library of operator-specific inverters in the next section.

4.1 Overview

Our inversion approach transforms expressions according to common rules for rewriting mathematical equations. It takes an assignment expression that involves attributes of two metaclasses as input of the forward transformation direction and outputs an inverse assignment expression for the backward transformation direction. Our approach assumes that instances of both metaclasses have already been created by a transformation engine so that their attribute values can directly be manipulated by the forward and backward transformations.

The input assignment represents an initial equation and the output assignment represents the equation that results from solving the initial equation for the variable corresponding to the attribute that shall be updated in a backward transformation. The output assignment is obtained by transforming the abstract syntax tree (AST) of the input assignment: every operation node on the way down to the leaf node for the attribute to be updated is replaced with an inverse operation and all other nodes remain unchanged. Each operation is inverted independently using an inverter for the used operator. Only the result of the previously inverted parent operation is passed in form of a temporary variable and the final result is the result of the last inversion.

4.2 Mapping Expressions

The attribute mapping expressions that can be inverted with our current prototype are assignment expressions for a metaclass of the target metamodel and a metaclass of the source metamodel of the forward direction with the following restrictions: On the left side exactly one attribute of the target metaclass has to be given. The right side can contain nested operations that mention at least one

attribute of the source metaclass and every of these attributes at most once. This property is called linear [Wad88] or affine [Mat+07] and guarantees straightforward inversion. In the following we will call the left side the target side and the right side the source side of an assignment.

If more than one attribute of the source metaclass is mentioned, one of these attributes has to be marked as the one to be updated in the backward direction. The reason is that we currently do not support operators that can only be inverted by updating more than one operand. Operations that operate directly or indirectly on the attribute according to which the expression is inverted have to use operators for which an inverter is defined. In the AST these operations correspond to nodes that are direct or indirect parents of the attribute leaf. All other operations can use arbitrary operators as they do not have to be inverted.

The expression of our initial car rental example `travelDistance = 100 * fuelCapacity / fuelConsumption` is an assignment expression for the attribute `travelDistance` of the metaclass `RentalCar` of the metamodel that acts as target in in the forward transformation direction. The source side is a multiplication operation of a constant literal operand and a division operation that mentions the two attributes `fuelCapacity` and `fuelConsumption` of the source metaclass `ManagedVehicle`. To enable an inversion of this expression both of these source attributes could be marked as the one to be updated in the backward direction. In our scenario, an inversion according to the fuel consumption would probably be chosen to respond to a change of the monitored travel distance that indirectly reflects a change of the average consumption and not of the fixed tank size.

4.3 Inversion Procedure

The inversion procedure for an attribute assignment expression consists of three steps. First, the AST of the expression is statically checked to ensure that the assignment fulfills the above requirements in addition to properties checked by the compiler of the transformation language. Then, a copy of the AST is transformed: first the root and then every node on the way to the leaf for the attribute according to which the expression is inverted. Finally, the source code for the inverted assignment is generated from the transformed AST copy in form of a method, which returns the result of the last inversion and has a parameter for the target attribute and for every source attribute.

It is possible to invert every operation individually because they only depend on the value of the operands and not on the internal structure of the operands. This can be illustrated using the expression of our car rental example `travelDistance = 100 * fuelCapacity / fuelConsumption`. It is inverted in two steps to `fuelCapacity / fuelConsumption = travelDistance / 100 =: tmp` and then to `fuelCapacity = tmp * fuelConsumption` which yields `(travelDistance / 100) * fuelConsumption`. The temporary variables, which we use during the code generation in our prototype, are not necessary as they could be inlined, but they make the generated code more readable.

4.4 Inverter Properties

In order to build well-behaved bidirectional transformations from a given forward attribute assignment expression and the inverse expression obtained with our approach, the inverters have to fulfill the GetPut and the PutGet laws of Foster et al. [Fos+07]. It is, however, not possible to invert every expression that may be desired in a bidirectional transformation language in a way that always fulfills the PutGet law if all target updates are allowed: For every operation that is not right-total (surjective) only updates to target values that are in the image of the operation can be inverted in a way that fulfills the PutGet law. An operation that returns the absolute value of a source value, for example, cannot be inverted without breaking the PutGet law if the target may be updated to a negative value: no matter which value will be put as new source, the absolute target value that we will get from it will always be positive and therefore not identical to the negative target value after the update.

We stick to the terminology of Foster et al. and call transformations that always fulfill the GetPut and the PutGet law *well-behaved transformations*. We introduce the new term *best-possible behaved transformations* for transformations that fulfill the GetPut in all cases and the PutGet law for every target change that can be inverted without breaking the PutGet law. Inverters that yield well-behaved or best-possible behaved transformations are also called well-behaved respectively best-possible behaved inverters. All 30 inverters that we present in this paper and realized in our initial prototype are best-possible behaved inverters and 14 of them are even well-behaved inverters. Proofs for the well-behavedness of our inverters are presented in a technical report [KR16]. They always have the same structure: for a partition W, B of the set of possible target values we show that a) GetPut holds for all target values, b) PutGet holds for all values in W, and c) for every inverter that would fulfill PutGet for a target value in B we obtain a contradiction.

Best-possible behaved inverters have to deal with target updates for which a violation of the PutGet law cannot be avoided. These cases are always updates to target values that are in the codomain of the function represented by the forward operator but not in the image of this function. They can, however, be divided into two categories: For PutGet violations of the first category some of the information of the updated target value can be used to choose a new source value for which the new target after a round-trip will be closer to the initially updated target value than for all other choices of source values. For PutGet violations of the second category no choice for a new source value yields a target values after a round-trip that is closer to the initially updated target value than for all other choices of source values. Therefore, we call the first type of PutGet violations *restrictable PutGet violations* and the second type *desperate PutGet violations*.

A restrictable PutGet violation occurrs, for example, if the target of the arithmetic abs operator is changed to a negative value: the absolute value of the negative target is used to choose a new source value that yields a target after a round-trip that has the correct absolute value but inevitably an incorrect algebraic sign. A desparate PutGet violation occurs, for example, if the target of the trigonometric sin operator is changed to a value that is not in the interval $[-1, 1]$: all choices for a new source value that are of the form $2n \pm \frac{\pi}{2}$ for an $n \in \mathbb{N}_0$ yield the target value ± 1 after a round-trip and are as close as possible to the initially updated target value.

In our prototype, we respond to restrictable violations with a handler that updates the source according to a passed value that is derived from the updated target value. How the passed value is changed before updating the source or whether a target update shall be rejected by throwing an exception can be customized using a callback. The default implementation directly updates the source to the passed value without any further changes and rejects no target update. For desperate PutGet violations, no kind of exception handling would make any difference so our prototype simply updates the source to a default value that is independent of the updated target value.

5 Inverters

In this section, we present best-possible behaved inverters for 30 common operators, which we also realized for our prototypical language. Before we define and explain each inverse operator in detail, we provide an overview and classification for all inverters.

5.1 Classification, Notation, and Overview

In the previous section, we have introduced the notion of well-behaved and best-possible behaved inverters and distinguished restrictable and desperate PutGet violations. There are two further properties that can be used to classify inverters: Operators with more than one operand can be inverted in an *operand-agnostic* way if they represent a commutative function. For all other operators with more than one operand we define an individual inverse operator for inversion according to each operand.

We write $\text{op}(s_1 : T_1, s_2 : T_2) : T_3$ to denote an operator with the name "op", two operands named "s_1" and "s_2" of type T_1 and T_2, and a return type T_3. An operand-agnostic inverse operator of this operator is denoted by op^{-1} and $\text{op}_\text{i}{}^{-1}$ denotes an inverse operator for inversion according to the operand with 1-based index i. All inverse operators have at least one parameter to obtain the updated target value and may have additional parameters for the values of the operands of the operator to be inverted.

Some *target-agnostic* operators can be inverted in way that fulfills the GetPut law with a single definition that holds for all possible target values. The remaining operators are inverted with separate functions for target values with different properties.

We grouped the operator for which we define inverse operations in five categories: primitive casts, boolean logical operators, basic and advanced arithmetic operators, and string operators. Table 1 lists properties of the operators and their inverse operators. The 14 well-behaved inverters are those that neither have restrictable nor desperate PutGet violations. All operators for which we present inverters in this paper operate on single values not on collections of values and can be inverted by updating a single source attribute. Inverters for collection operators and for operators that require updates of more than one source attribute in backward direction are part of our future work.

Note that our inverters are just one possibility to invert a given operation. For many operations, however, there are not many different ways to define a best-possible behaved inverter that updates only a single source attribute. Inverters that update more than one source attribute have an important additional degree of freedom: the difference between the old and the updated target value Δ can now be split in different ways on several source attributes. Such inverters for binary arithmetic operators may, for example, apply the inverse arithmetic operation using $\frac{\Delta}{2}$ to both source attributes or using Δ to one of both source attributes. But this paper focuses on inverters that update only a single source attribute.

The presented extensible library is restricted to inverters that update only one attribute and to an incomplete set of common operations. But many inverters for operations that we did not address can reuse presented inverters or can be defined in a similar way. A new inverter for a string concatenation operator with more than two operands, for example, could easily be defined even if more than operand shall be updated in the inverse transformation.

In the following definitions we will use a helper `restrictPGV(p:T):T` to encapsulate the handling of restrictable violations of the PutGet law based on the value of the parameter p. In our prototype the default implementation always returns the passed value, but it can be customized to react differently depending on the value and / or operator that was inverted. For desperate violations of the PutGet law a helper `reportPGV(p:T):T` updates the source to the given fixed value and reports the violation.

5.2 Primitive Casts

Type conversions and a notion of type-compatibility are necessary for some arithmetic operators. Therefore, we start by defining inverse operators for primitive type casts. These are the only possible casts that can appear in attribute mapping expressions. Casts of complex-typed references to metaclass instances have to be handled separately in Ecore-based bidirectional transformation languages.

If a numeric type T_2 can be converted without information loss to a numeric type T_1, we call T_1 wider than T_2 and write $T_1 > T_2$. For our prototype we use the relation that is defined by the widening primitive conversion int the Java language specification[3]: double > float > long > int > short > byte.

If a floating-point value x is equal to another floating-point value y with a relative tolerance of ε, i.e. $|\frac{x-y}{max(x,y)}| < \varepsilon$, we call x and y ε-equal and write $x \overset{\varepsilon}{=} y$. In our prototype values are ε-equal if a call to `org.apache.commons.math3.util.Precision.equalsWithRelativeTolerance` using the IEEE 754 machine epsilon 2^{-53} returns true, but the epsilon can be configured differently and the comparison could be replaced with a comparison based on the units in the last place (ulp).

[3]Java Widening Primitive Conversion: docs.oracle.com/javase/specs/jls/se8/html/jls-5.html#jls-5.1.2

Operator to Invert	Argument Types	Operand-Agnostic	Target-Agnostic	no restrictable PutGet violations	no desperate PutGet violations
Primitive Casts					
narrowing cast	numeric	–	✓	✓	✓
widening cast (ex- and implicit)	numeric	–	✗	✗	✓
Boolean Logical Operators					
not, xor	boolean	–	✓	✓	✓
Basic Arithmetic Operators					
unary minus	numeric	–	✓	✓	✓
addition, multiplication	numeric	✓	✓	✓	✓
float division	floats	✗	✓	✓	✓
int division	integers	✗	✗	✓	✓
Advanced Arithmetic Operators					
absolute value	numeric	–	✗	✗	✓
rounding	floats	–	✗	✓	✓
floor, ceil	double	–	✗	✗	✓
floor modulus	integers	✗	✗	✗	✗
exponentiation	b:numeric,e:integers	✗	✗	✗	✓
sin, cos	floats	–	✗	✓	✗
tan	floats	–	✗	✓	✓
asin, acos, atan	floats	–	✗	✗	✓
String Operators					
parse	boolean,numeric	–	✓	✓	✓
num printing	numeric	–	✗	✓	✗
bool printing	boolean	–	✗	✓	✓
length	strings	–	✗	✓	✓
concat	strings	✗	✗	✗	✓
suffix	strings	–	✓	✓	✓
substring with fixed indices	strings	–	✗	✗	✓
toUpperCase, toLowerCase	strings	–	✗	✗	✓

Table 1: Arguments of the operators and inverter properties (– : not applicable, ✗ : no, ✓ : yes)

For two numeric types $T_1 > T_2$ and the narrowing primitive cast operator $\text{ncast}_{\text{T}_1,\text{T}_2}(source : T_1) : T_2$ we define the inverse operator $\text{ncast}_{\text{T}_1,\text{T}_2}{}^{-1}(target : T_2) : T_1 := \text{wcast}_{\text{T}_2,\text{T}_1}(target)$.

For two numeric types $T_2 > T_1$ and the widening primitive cast operator $\text{wcast}_{\text{T}_1,\text{T}_2}(source : T_1) : T_2$ we define the inverse operator

$$\text{wcast}_{\text{T}_1,\text{T}_2}{}^{-1}(t : T_2) : T_1 := \begin{cases} \text{ncast}_{\text{T}_2,\text{T}_1}(t) & \text{if } \text{wcast}_{\text{T}_1,\text{T}_2}(\text{ncast}_{\text{T}_2,\text{T}_1}(t)) \overset{\varepsilon}{=} t \\ \text{restrictPGV}(\text{ncast}_{\text{T}_2,\text{T}_1}(t)) & \text{otherwise} \end{cases}$$

To invert all implicit casts in expressions, which are called "widening primitive conversions" for Java, we replace them with explicit widening casts before inverting an expression and use the inverse operator wcast^{-1} as defined above. As a result, all explicit and implicit widening casts are inverted using a narrowing cast without violating the PutGet-law whenever the target value can be cast with a relative error smaller than ε. In all other cases a PutGet violation cannot be avoided but its effect can be restricted by choosing the cast target value as new source value.

5.3 Boolean Logical Operators

The next group of operators with inverters consists only of the not and the xor operator, because conjunctions and disjunctions cannot always be inverted by updating only a single source attribute: If a target value is changed from 1 to 0 an inverter for the and operator has to update both source values and an inverter for the or operator has to do this if both source values were 1.

Not
For the operator $\text{not}(source : bool) : bool$ we define the trivial inverse operator $\text{not}^{-1}(target : bool) : bool := \text{not}(target)$

Xor
For the operator $\text{xor}(s_1 : bool, s_2 : bool) : bool$ we define the inverse operator $\text{xor}_1{}^{-1}(target : bool, s_2 : bool) : bool := \text{xor}(target, s_2)$ for inversion according to the first operand s_1 and the inverse operator $\text{xor}_2{}^{-1}(target : bool, s_1 : bool) : bool := \text{xor}(target, s_1)$ for inversion according to the second operand s_1.

5.4 Basic Arithmetic Operators

This group of operators realizes the four basic arithmetic operations on integer and floating-point types.

Unary Minus
For all numeric types T and the operator $\text{unaryminus}(source : T) : T$ we define the trivial inverse operator $\text{unaryminus}^{-1}(target : T) : T := \text{unaryminus}(target)$

Addition
For all numeric types T and the operator $\text{addition}(s_1 : T, s_2 : T) : T$ we define the inverse operator $\text{addition}^{-1}(target : T, s : T) : T := \text{addition}(target, \text{unaryminus}(s))$.

The language of our prototype provides no subtraction operator but replaces the syntactic sugar $s_1 - s_2$ with $\text{addition}(s_1, \text{unaryminus}(s_2))$ in order to reduce the inversion of subtraction operations to the inversion of unary minus operations.

Multiplication
For all numeric types T and the operator $\text{multiplication}(s_1 : T, s_2 : T) : T$ we define the inverse operator $\text{multiplication}^{-1}(target : T, s : T) : T := \text{xdivision}(target, s)$ where xdivision is floatdivision if T is a floating-point type and otherwise intdivision.

Division
For two floating-point types $T_1 > T_2$ or $T_1 = T_2$ and the operator $\text{floatdivision}(s_1 : T_1, s_2 : T_2) : T_1$ we define the inverse operator $\text{floatdivision}_1{}^{-1}(target : T_1, s_2 : T_2) : T_1 := \text{multiplication}(t, s_2)$ for inversion

according to the dividend s_1 and the inverse operator $\text{floatdivision}_2{}^{-1}(target : T_1, s_1 : T_1) : T_1 := \text{floatdivision}(s_1, t)$ for inversion according to the divisor s_2.

For two integer types $T_1 > T_2$ or $T_1 = T_2$ and the IEEE 754 round-toward-0 operator $\text{intdivision}(s_1 : T_1, s_2 : T_2) : T_1$ we define the inverse operator

$$\text{intdivision}_1{}^{-1}(t : T_1, s_1 : T_1, s_2 : T_2) : T_1 := \begin{cases} s_1 & \text{if } \text{intdivision}(s_1, s_2) = t \\ \text{multiplication}(t, s_2) & \text{otherwise} \end{cases}$$

for inversion according to the dividend s_1, and the inverse operator

$$\text{intdivision}_2{}^{-1}(t : T_1, s_1 : T_1, s_2 : T_2) : T_1 := \begin{cases} s_2 & \text{if } \text{intdivision}(s_1, s_2) = t \\ \text{intdivision}(s_1, t) & \text{otherwise} \end{cases}$$

for inversion according to the divisor s_2.

Integer division is an operator that is not left-unique (injective). Therefore, it cannot be inverted in a way that fulfills the GetPut law without inspecting the original target value. The presented inverse operators for intdivision avoid a violation of the GetPut law by checking whether the target was changed to another value than the one that we would get from the source values using the original operator. If this is the case, they return the original source value for the operand according to which the operation is inverted in order to fulfill the GetPut law. In all other cases it does not matter which of the values that would fulfill the GetPut law is chosen. Therefore, the common division inversion by multiplication with the divisor respectively division by the dividend is enough.

5.5 Advanced Arithmetic Operators

To simplify the definition of inverse operators for advanced arithmetic operators, we will use a helper, which returns the algebraic sign for uses in multiplications and is defined for numeric types T as

$$\text{sign4mult}(p : T) : T := \begin{cases} 1 & \text{if } p \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

Absolute Value

For a numeric type T and the absolute value operator $\text{abs}(source : T) : T$ we define the inverse operator

$$\text{abs}^{-1}(target : T, source : T) : T := \begin{cases} \text{sign4mult}(source) \cdot target & \text{if } target \geq 0 \\ \text{restrictPGV}(\text{sign4mult}(source) \cdot |target|) & \text{otherwise} \end{cases}$$

With this inverter we can sustain the information about the absolute value of an updated target and restrict the loss of information to the algebraic sign of it, which cannot be avoided for the abs operator.

For numeric x and y, we briefly write $|x|$ to denote $\text{abs}(x)$ and $x \cdot y$ to denote $\text{multiplication}(x, y)$.

Round to Nearest

For a floating-point type T and the IEEE 754 round-to-nearest operator $\text{round}(source : T) : int$ we define the inverse operator

$$\text{round}^{-1}(target : int, source : T) : T := \begin{cases} source & \text{if } \text{round}(source) \overset{\varepsilon}{=} target \\ \text{wcast}_{\text{int,T}}(target) & \text{otherwise} \end{cases}$$

Round toward Infinity

For the IEEE 754 round-toward-$-\infty$ operator $\text{floor}(source : double) : double$ we define the inverse operator

$$\text{floor}^{-1}(target : double, source : double) : double := \begin{cases} source & \text{if } \text{floor}(source) \overset{\varepsilon}{=} target \\ target & \text{if } \text{floor}(target) \overset{\varepsilon}{=} target \\ \text{restrictPGV}(target) & \text{otherwise} \end{cases}$$

floormod		dividend					
		−13	−6	*−5*	*5*	6	*13*
divisor	−9		−3	*−4*		3	*4*
	9	*−4*	−3		*4*	3	

Table 2: Old and *new* operand values of the floormod operator for target updates from ±3 to ±*4*

For the IEEE 754 round-toward-∞ operator ceil the inverse operator ceil^{-1} is defined completely analog to floor and floor^{-1}.

Modulus

Instead of defining an inverter for the modulus operator that uses round-to-zero division, which is denoted by `a % b` in Java, we present an inverter for the floor modulus operator[4]. It is defined as $\text{floormod}(\mathit{divisor}, \mathit{dividend}) := \mathit{divisor} - (\text{floordiv}(\mathit{divisor}, \mathit{dividend}) \cdot \mathit{dividend})$, where floordiv is the round-toward-$-\infty$ floor division operator and "returns the largest [...] integer value that is less than or equal to the algebraic quotient". This operator yields a modulus with the same sign as the divisor, which is helpful for example for array index arithmetic.

For an integer type T and the modulus or remainder operator $\text{floormod}(s_1 : T, s_2 : T) : T$ we define the inverse operator

$$\text{floormod}_1{}^{-1}(t : T, s_1 : T, s_2 : T) : T := \begin{cases} s_1 & \text{if } \text{floormod}(s_1, s_2) = t \\ \text{floordiv}(s_1, s_2) \cdot s_2 + t & \text{if } \text{floormod}(t, s_2) = t \\ \text{restrictPGV}(t) & \text{otherwise} \end{cases}$$

for inversion according to the dividend s_1, and the inverse operator

$$\text{floormod}_2{}^{-1}(t : T, s_1 : T, s_2 : T) : T := \begin{cases} s_2 & \text{if } \text{floormod}(s_1, s_2) = t \\ t + s_2 \cdot \text{sign4mult}(t) & \text{if } s_1 = t \\ |s_1 - t| \cdot \text{sign4mult}(t) =: s_2' & \text{if } \text{floormod}(s_1, s_2') = t \\ \text{reportPGV}(1) & \text{otherwise} \end{cases}$$

for inversion according to the divisor s_2.

In the case of $\text{floormod}(\mathit{target}, s_2) = \mathit{target}$ we could also make $\text{floormod}_1{}^{-1}(\mathit{target}, s_1, s_2)$ return simply *target*. For every $n \in \mathbb{N}_0$ returning $n \cdot s_2 + \mathit{target}$ would fulfill PutGet. Our choice of $n = \text{floordiv}(s_1, s_2)$ preserves information about the old range of the divisor s_1 before the update of the target: For example, if the target for a divisor of 5 and a dividend of 3 is changed from 2 to 1, our inversion of the remainder operator would update the divisor to 4 instead of 1.

In the case of $s_1 = \mathit{target}$ the inversion according to the divisor $\text{floormod}_2{}^{-1}(\mathit{target}, s_1, s_2)$ fulfills PutGet if it returns $\mathit{target} + n \cdot \text{sign4mult}(t)$ for an $n \in \mathbb{N}_{\setminus\{0\}}$. Our choice of $n = s_2$ preserves information about the old value of the dividend s_2 before the update of the target: For example, if the target for a divisor of 5 and a dividend of 3 is changed from 2 to 5, our inversion of the remainder operator would update the divisor to 8 to indicate that the divisor was $8 - 5 = 3$ before the update.

An example for which all four possible target changes from ±t to $\pm t'$ can be inverted is given in Table 2 based on the divisor values ±9 and the dividend values ±6.

[4]Java floor modulus operator: docs.oracle.com/javase/8/docs/api/java/lang/Math.html#floorMod-int-int-

Exponentiation
For a numeric type T_1, a floating-point type T_2, and the exponentiation operator $\text{pow}(b : T_1, e : T_2) : \mathit{double}$ we define the inverse operator

$$\text{pow}_1{}^{-1}(t : \mathit{double}, b : T_1, e : T_2) : T_1 := \begin{cases} \begin{cases} \text{sign4mult}(b) \cdot \sqrt[e]{t} & \text{if } t \geq 0 \\ \text{restrictPGV}(\text{sign4mult}(b) \cdot \sqrt[e]{|t|}) & \text{otherwise} \end{cases} & \text{if } e \text{ is even} \\ \text{sign4mult}(t) \cdot \sqrt[e]{|t|}) & \text{otherwise} \end{cases}$$

for inversion according to the base b, and the inverse operator

$$\text{pow}_2{}^{-1}(t : \mathit{double}, b : T_1, e : T_2) : T_2 := \text{wcast}_{\mathrm{T}_1,\mathrm{T}_2}{}^{-1} \left(\begin{cases} e & \text{if } b^e = t \\ log_{|b|}(|t|) & \text{if } b^{log_{|b|}(|t|)} \stackrel{\varepsilon}{=} t \\ \text{restrictPGV}(log_{|b|}(|t|)) & \text{otherwise} \end{cases} \right)$$

for inversion according to the exponent e.

Trigonometric Operators
For the trigonometric operator $\sin(\mathit{source} : \mathit{double}) : \mathit{double}$ we define the inverse operator

$$\sin^{-1}(t : \mathit{double}, \mathit{source} : \mathit{double}) : \mathit{double} := \begin{cases} \mathit{source} & \text{if } \sin(\mathit{source}) \stackrel{\varepsilon}{=} t \\ \text{asin}(t) & \text{if } -1 \leq t \leq 1 \\ \text{reportPGV}(\text{sign4mult}(t) \cdot \frac{\pi}{2}) & \text{otherwise} \end{cases}$$

For the trigonometric operator cos the inverse operator $\cos^{-1}$ is defined completely analog to sin and $\sin^{-1}$: only $\text{sign4mult}(t) \cdot \frac{\pi}{2}$ has to be replaced with $\frac{\pi}{2} - \text{sign4mult}(t) \cdot \frac{\pi}{2}$ for $\cos^{-1}$. For the trigonometric operator $\tan(\mathit{source} : \mathit{double}) : \mathit{double}$ we define the inverse operator

$$\tan^{-1}(\mathit{target} : \mathit{double}, \mathit{source} : \mathit{double}) : \mathit{double} := \begin{cases} \mathit{source} & \text{if } \tan(\mathit{source}) \stackrel{\varepsilon}{=} \mathit{target} \\ \text{atan}(\mathit{target}) & \text{otherwise} \end{cases}$$

Inverse Trigonometric Operators
For the inverse trigonometric operator $\text{asin}(\mathit{double}) : \mathit{double}$ we define the inverse operator

$$\text{asin}^{-1}(\mathit{target} : \mathit{double}, \mathit{source} : \mathit{double}) : \mathit{double} := \begin{cases} \sin(\mathit{target}) & \text{if } |\mathit{target}| \leq \dfrac{\pi}{2} \\ \text{restrictPGV}(\sin(\mathit{target})) & \text{otherwise} \end{cases}$$

For the inverse trigonometric operators acos and atan the inverse operators acos^{-1} and atan^{-1} are defined analog to sin and $\sin^{-1}$: only $|\mathit{target}| \leq \frac{\pi}{2}$ has to be replaced with $0 \leq \mathit{target} \leq \pi$ for acos^{-1}.

5.6 String Operators

The last group of operators for which we define inverse operators involves character strings.

Parsing, Printing and Length
For all types T and the operator $\text{parse}(\mathit{source} : \mathit{string}) : T$ we define the trivial inverse operator $\text{parse}^{-1}(\mathit{target} : T) : \mathit{string} := \text{print}(\mathit{target})$.

For all numeric types T and the operator $\text{numprint}(s : T) : \mathit{string}$ we define the inverse operator

$$\text{numprint}^{-1}(t : \mathit{string}, s : T) : T := \begin{cases} \text{parse}(t) & \text{if } t \text{ represents a number of type } T \\ \text{reportPGV}(0) & \text{otherwise} \end{cases}$$

For the operator $\text{boolprint}(\mathit{source} : \mathit{bool}) : \mathit{string}$ we define the inverse operator

$$\text{boolprint}^{-1}(\mathit{target} : \mathit{string}, \mathit{source} : \mathit{bool}) : \mathit{bool} := \begin{cases} \mathit{true} & \text{if } \mathit{target} = \text{“}\mathit{true}\text{”} \text{ (case insensitive)} \\ \mathit{false} & \text{otherwise} \end{cases}$$

We define a helper $\text{pad}(\mathit{source} : \mathit{string}, \mathit{length} : \mathit{integer})$, which appends as many underscore characters to a given string *source* as are needed to obtain a string with *length* characters. We also

define a helper to obtain prefixes that are automatically padded to a desired length using the pad helper:

$$\text{prefix}(source : string, end : int) : T := \begin{cases} \text{substring}(source, 0, end) & \text{if } end \leq \text{length}(source) \\ \text{pad}(source, end) & \text{otherwise} \end{cases}$$

It uses the substring operator with fixed indices $\text{substring}(s : string, b : int, e : int)$, which returns $e - b$ subsequent characters of s including the character at index b and excluding the character at index e. For the operator $\text{length}(source : string) : int$, for which we briefly write $|source|$, we can now define the inverse operator

$$\text{length}^{-1}(target : int, source : string) : string := \text{prefix}(source, target)$$

Concatenation and Substrings

For the string concatenation operator $\text{concat}(s_1 : string, s_2 : string) : string$, for which we briefly write $s_1 \frown s_2$, we define the inverse operator

$$\text{concat}_1{}^{-1}(target : string, s_2 : string) : string := \begin{cases} s_1' & \text{if } target = s_1' \frown s_2 \\ \text{restrictPGV}(target) & \text{otherwise} \end{cases}$$

for inversion according to the first operand s_1, and the inverse operator

$$\text{concat}_2{}^{-1}(target : string, s_1 : string) : string := \begin{cases} s_2' & \text{if } target = s_1 \frown s_2' \\ \text{restrictPGV}(target) & \text{otherwise} \end{cases}$$

for inversion according to the second operand s_2.

We define a specialized substring operator:

$$\text{suffix}(s : string, b : int) : T := \begin{cases} \text{substring}(s, b, |s|) & \text{if } b < |s| \\ \text{“”} & \text{otherwise} \end{cases}$$

where “” denotes the empty string. Its inverse operator is

$$\text{suffix}^{-1}(t : string, s : string, b : int) : string := \text{prefix}(s, b) \frown t$$

We define a helper that concatenates a circumfix c and an infix i by prepending the first e characters of the circumfix to the infix while appending the last $|c| - b$ characters of the circumfix:

$$\text{circumcat}(c : string, e : int, i : string, b : int) := \text{prefix}(c, e) \frown i \frown \text{suffix}(c, b)$$

Now we can define an inverse operator for the substring operator with fixed indices $\text{substring}(s : string, b : int, e : int) : string$ using the pad and circumcat helpers:

$$\text{substring}^{-1}(t : string, s : string, b : int, e : int) : string :=$$
$$\begin{cases} \text{circumcat}(s, b, t, e) & \text{if } |t| = b - e \\ \text{restrictPGV}(\text{circumcat}(s, b, t, e)) & \text{if } |t| > b - e \\ \text{restrictPGV}(\text{circumcat}(s, b, \text{pad}(t, b - e), e)) & \text{otherwise} \end{cases}$$

We illustrate the inversion of the substring operator with fixed indices using the example input $s =$“inverse”, $b = 2$, and $e = 6$: If the target “vers” is changed to “plac” the first case applies because $|\text{“plac”}| = 4 = 6 - 2$ and the source is changed to “in”⌢“plac”⌢“e”. If the target is changed to “carnat” the second case applies because $|\text{“carnat”}| = 6 > 6 - 2$ and the source is changed to “in”⌢“carnat”⌢“e”. If the target is changed to “di” the third case applies because $|\text{“di”}| = 2 < 6 - 2$ and the source is changed to “in”⌢“di___”⌢“e”. Without the third case a target change to “di” would yield “indie” for which an application of *substring* with $b = 2$ and $e = 6$ would not be possible because $e = 6 > 5 = |\text{“indie”}|$. Therefore, we have to ensure that the source string has at least the length of the target string.

Letter Case

To invert letter case conversions we define a helper that returns the index of the first occurrence of a pattern p in a string s or the length of s if the pattern does not occur:

$$\text{firstIndex}(s : \mathit{string}, p : \mathit{string}) : \mathit{int} := \min(\{i \in \mathbb{N}_0 \mid \text{substring}(s, i, i + |p|) = p\} \cup \{|s|\})$$

For the to-upper-case-conversion operator $\text{tUC}(s : \mathit{string}) : \mathit{string}$ we define the inverse operator

$$\text{tUC}^{-1}(t : \mathit{string}, s : \mathit{string}) : \mathit{string} :=$$

$$\begin{cases} \text{restrictPGV}(\text{tUC}^{-1}(\text{tUC}(t), s)) & \text{if } t \neq tUC(t) \\ \text{substring}(s, i, i + |t|) & \text{if } |t| < |s| \wedge i := \text{firstIndex}(tUC(s), t) < |s| \\ \text{tLC}(\text{prefix}(t, i)) \frown s \frown \text{tLC}(\text{suffix}(t, i + |s|)) & \text{if } |t| > |s| \wedge i := \text{firstIndex}(t, tUC(s)) < |t| \\ \text{tLC}(t) & \text{otherwise} \end{cases}$$

We illustrate the inversion of the upper-case conversion operator based on the example input $s =$"CamelCase": If the target "CAMELCASE" is changed to "Cas", the first case of the definition applies because "Cas" $\neq$ "CAS" = tUC("Cas"). The inverse operator is recursively called with the new target "CAS" and the obtained string will be used as default value during the handling of the PutGet violation. The recursive call is identical to what happens if the target is directly changed to "CAS": the second case applies because |"CAS"| $= 3 < 9 =$ |"CamelCase"| and firstIndex(tUC("CamelCase"), "CAS") $= 5 < 9$. Therefore, "Cas" is returned. If the target is changed to "NOCAMELCASED", the third case applies because |"NOCAMELCASED"| $= 12 > 9$ and firstIndex("NOCAMELCASED", tUC("CamelCase")) $= 2 < 12$. Therefore, "no"$\frown$"CamelCase"$\frown$"d" is returned. If the target is changed to "DROMEDAR", the last case applies and "dromedar" is returned.

For the to-lower-case-conversion operator tLC the inverse tLC^{-1} is defined completely analog.

6 Discussion and Evaluation

In this section we briefly discuss formal properties of the presented inverters, present the results of our evaluation of applicability, and summarize the limitations of our approach and the presented inverters.

6.1 Formal Discussion

To prove that an inverter is best-possible behaved, we have to show that GetPut holds for all new target values and that PutGet holds for all target values for which it can hold. We present an exemplary proof for the inverter of the abs operator. Further proofs are available in a technical report [KR16].

Let s be a source value. Then $\text{abs}^{-1}(\text{abs}(s), s) = \text{abs}^{-1}(|s|), s) = \text{sign4mult}(s) \cdot |s|$. If $s \geq 0$, this yields $1 \cdot s = s$. Otherwise $s < 0$, which yields $-1 \cdot -1 \cdot s = s$. Thus the GetPut law holds for all s. Let t be a target value such that $t \geq 0$ and let s be an arbitrary source value. Then $\text{abs}(\text{abs}^{-1}(t, s)) = \text{abs}(\text{sign4mult}(s) \cdot t)$. If $s \geq 0$, this yields $\text{abs}(1 \cdot t) = \text{abs}(t) = t$. Otherwise $s < 0$, which yields $\text{abs}(-1 \cdot t) = \text{abs}(-t) = t$. Thus the PutGet law holds for all $t \geq 0$. Assume $\text{abs}^{-1'}$ is an inverse operator for abs that fulfills PutGet for a target value t such that $t < 0$ and a source value s. Then $\text{abs}(\text{abs}^{-1'}(t, s)) = t$. This yields $|\text{abs}^{-1'}(t, s)| = t < 0$ which is a contradiction to the definition of abs because $|x| \geq 0$ for all x. Because the GetPut law holds for all s, the PutGet law holds for all $t \geq 0$ and cannot hold for any inverse operator $\text{abs}^{-1'}$ and $t < 0$, we conclude that abs^{-1} is a best-possible behaved inverter.

6.2 Evaluation

In addition to unit tests for all inverters and target use cases we evaluated how often attribute mapping expressions with operators that can be inverted with our inverters appear in available model transformations. We categorized the operators used in all 103 transformations of the so-called ATL Transformations Zoo[2]. The sum of logical lines of code (LLOC) for each operator category is shown

Category	LLOC
Identity Operator	2238
Arithmetic Operators	119
Parsing or Printing	441
Other String Operators	356
Sequence Operators	1478
List Operators	1062

Table 3: Categories for operators of attribute mapping expressions of the ATL Transformation Zoo

in Table 3. The identity operator is used in 39% of all lines. The three operator categories arithmetic, parsing or printing, and other string operators, for which we defined non-trivial inverters, make up additional 16%. Sequence and list operators that cannot yet be inverted with our approach are used in 44% of all lines. If we exclude the trivial identity operator, we can conclude that we defined inverters for 26% of the LLOC of all non-trivial attribute expressions.

6.3 Limitations

Currently our approach is bound to one limitation and the presented inverters to two restrictions. As we already stated in subsection 4.2, our approach can only be used for expressions in which every source attribute appears at most once. Furthermore, we currently only defined inverters for operators that can be inverted by updating a single source attribute. Finally, all supported operators only operate on single-valued attributes not on collections or sequences.

In our opinion, the restriction to linear or affine expressions is not a big limitation for realistic applications. The restriction to operators that can be inverted with a single update limits the applicability of our approach but it is only temporary: the conceptual framework and implementation prototype can easily be adapted in the future to support inverters that update several source attributes. Even defining inverters for operators on collections or sequences should not be conceptually more difficult: If the source value collections before an update of the target collection are given, then the inversion of a collection operator is often similar to the inversion of single-element operators. The technical realization and static analysis e.g. of higher-order functions is, however, challenging.

7 Conclusions and Future Work

In this paper, we have presented an approach for the automated inversion of attribute mappings in forward specifications of bidirectional transformations. It is based on an expression language with a Java-aligned syntax and supports the inversion of mappings that assign the result of a linear compound expression to an attribute of a metaclass. We have explained the overall inversion process based on operator-specific inverters that can be independently composed and we have introduced a notation for inverters that always fulfill GetPut and that fulfill PutGet for all target values where this can be achieved: best-possible behaved inverters. Next, we have presented an initial library of 30 best-possible behaved inverters for common logical, arithmetic, and string operators. Finally, we have discussed an evaluation of the applicability of our approach which may indicate that many of the non-trivial attribute mapping expressions that appear in available transformations could be inverted with our approach.

In future work, we will provide inverters for further operators. Operators that involve collections of source attribute values and operators that can only be inverted if different source attributes are updated, e.g. the boolean logical and operator, are our main interest.

Acknowledgments

This work was partly supported by the German Federal Ministry of Education and Research within the framework of the project "Security for the Internet of Everything" in the Competence Center for Applied Security Technology (KASTEL).

References

[AVS12] A. Anjorin et al. "Complex Attribute Manipulation in TGGs with Constraint-Based Programming Techniques." In: *Proceedings of the First International Workshop on Bidirectional Transformations (BX 2012)*. Vol. 49. Electronic Communications of the EASST. 2012.

[Boh+08] A. Bohannon et al. "Boomerang: Resourceful Lenses for String Data." In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. ACM, 2008, pp. 407–419.

[Dij79] E. W. Dijkstra. "Program inversion." In: *Program Construction*. Vol. 69. LNCS. Springer Berlin Heidelberg, 1979, pp. 54–57.

[Eff+12] S. Efftinge et al. "Xbase: Implementing Domain-specific Languages for Java." In: *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*. GPCE '12. ACM, 2012, pp. 112–121.

[Fos+07] J. N. Foster et al. "Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29.3 (2007).

[GW09] H. Giese and R. Wagner. "From model transformation to incremental bidirectional model synchronization." In: *Software and Systems Modeling* 8 (1 2009), pp. 21–43.

[Hil+13] S. Hildebrandt et al. "A survey of triple graph grammar tools." In: *Electronic Communications of the EASST* 57 (2013).

[KR16] M. E. Kramer and K. Rakhman. *Proofs for the Automated Inversion of Attribute Mappings in Bidirectional Model Transformations*. Tech. rep. Karlsruhe Institute of Technology, Department of Informatics, 2016.

[Mat+07] K. Matsuda et al. "Bidirectionalization Transformation Based on Automatic Derivation of View Complement Functions." In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. ICFP '07. ACM, 2007, pp. 47–58.

[Rak15] K. Rakhman. "Automated Inversion of Attribute Mapping Expressions for Multi-Model Consistency." MA thesis. Karlsruhe Institute of Technology (KIT), Germany, 2015.

[Ste08] P. Stevens. "A Landscape of Bidirectional Model Transformations." In: *Generative and Transformational Techniques in Software Engineering II*. Vol. 5235. LNCS. Springer Berlin Heidelberg, 2008, pp. 408–424.

[Wad88] P. Wadler. "Deforestation: Transforming Programs to Eliminate Trees." In: *Theoretical Computer Science* 73.2 (1988), pp. 231–248.

[Xio+07] Y. Xiong et al. "Towards automatic model synchronization from model transformations." In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ASE '07. ACM, 2007, pp. 164–173.

[YG07] T. Yokoyama and R. Glück. "A Reversible Programming Language and Its Invertible Self-interpreter." In: *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*. PEPM '07. ACM, 2007, pp. 144–153.