Automated Inversion of Attribute Mapping Expressions for Multi-Model Consistency

Master's Thesis of

Kirill Rakhman

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Jun.-Prof. Dr.-Ing. Anne Koziolek
Advisor:	Dipl.-Inform. Max Kramer
Second advisor:	M.Sc. Michael Langhammer

08. July 2015 – 07. December 2015

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 07. December 2015

. . . . . .

(Kirill Rakhman)

Abstract

Because keeping two models synchronized in a multi-view modeling scenario is a tedious and error-prone task, bidirectional model transformation approaches have been developed to automatically ensure model consistency. Many existing approaches focus on the synchronization of structural aspects, i.e. creating and deleting corresponding meta class instances as well as creating, updating and deleting references between meta classes. However, they lack the possibility to express complex mappings between the attributes of these meta classes which go beyond assigning a constant value or the value of one attribute to another attribute.

In this thesis an approach is presented that allows synchronizing an attribute of one target meta class with a number of attributes of another corresponding source meta class by writing a single mapping expression using an expressive, Java-based mapping syntax. This mapping expression defines how the target attribute's value is computed using numerical, logical and string operations. A computation rule is then automatically generated that defines how one of the source attributes will be updated given a changed target value and the complete set of source values. This is achieved by defining so-called inversions for each of the supported operations.

We demonstrate that the individual operations can be combined and nested while, whenever possible, obeying round trip laws that have been defined for solving the related view-update problem. We then present an extensible architecture for implementing these inversions via code generation. Finally, we evaluate the selection of supported operations based on an analysis of the ATL transformation zoo to show that the operations we can invert make up a substantial portion of commonly used attribute mappings.

Abstract

Weil das Synchronisieren zweier Modelle beim Modellieren von Software mit mehreren Sichten eine aufwendige und fehleranfällige Aufgabe ist, wurden bidirektionale Modelltransformationsansätze entwickelt, um automatisch Konsistenz zwischen den Modellen zu gewährleisten. Viele existierende Ansätze fokussieren sich dabei auf das Synchronisieren struktureller Aspekte, d.h. das Erstellen und Löschen zusammengehöriger Metaklassen, sowie das Erstellen, Aktualisieren und Löschen von Referenzen zwischen Metaklassen. Allerdings bieten sie nicht die Möglichkeit, komplexe Mappings zwischen den Attributen dieser Metaklassen zu formulieren, die darüber hinausgehen, einem Attribut einen konstanten Wert oder den Wert eines anderen Attributes zuzuweisen.

In dieser Arbeit wird ein Ansatz vorgestellt, der es erlaubt, ein Attribut einer Zielmetaklasse mit einer Zahl von Attributen einer Quellmetaklasse zu synchronisieren, indem ein einziger Ausdruck in einer ausdrucksstarken, Java-basierten Mapping-Syntax geschrieben wird. Dieser Mapping-Ausdruck definiert, wie der Wert des Zielattributs berechnet wird und kann numerische, logische und Stringoperationen enthalten. Daraus wird automatisch eine inverse Berechnungsvorschrift erzeugt, die besagt, wie aus einem geänderten Zielwert und einem Satz Quellwerte ein neuer Quellwert berechnet wird. Dies geschieht, indem sogenannte Invertierungen für jede der unterstützten Operationen definiert werden.

Es wird demonstriert, dass die einzelnen Operationen kombiniert und geschachtelt werden können und dabei, wann immer möglich, Korrektheitseigenschaften erfüllen, die für Lösungen des verwandten View-Update-Problems definiert wurden. Anschließend wird eine erweiterbare Architektur für die Implementierung dieser Operationen mithilfe von Code-Generierung präsentiert. Schließlich wird die Auswahl der unterstützten Operationen anhand einer Analyse des „ATL Transformation Zoo“ evaluiert und gezeigt, dass die invertierbaren Operationen einen bedeutenden Anteil der genutzten Attribut-Mappings ausmachen.

Contents

Abstract i

Abstract iii

1. Introduction and Motivation 1
- 1.1. Introduction . . . 1
- 1.2. Motivation . . . 1
- 1.3. Goals of this Thesis . . . 3

2. Foundation 5
- 2.1. Model Consistency Preservation with Vitruvius . . . 5
  - 2.1.1. The Domain Specific Language MIR . . . 5
  - 2.1.2. Mapping Syntax . . . 6
  - 2.1.3. Used Technologies . . . 6
- 2.2. The View-Update Problem . . . 7
- 2.3. Lenses . . . 8

3. Contribution 11
- 3.1. Approach . . . 11
  - 3.1.1. Outline of the Approach . . . 11
  - 3.1.2. Correctness Properties . . . 14
  - 3.1.3. Supported Operations . . . 16
- 3.2. Inverted Operations . . . 18
  - 3.2.1. Unary Primitive Operators . . . 18
  - 3.2.2. Basic Arithmetic Operations . . . 19
  - 3.2.3. Primitive Casts . . . 21
  - 3.2.4. Arithmetic Operations Between Different Types . . . 22
  - 3.2.5. Advanced Arithmetic Operations . . . 22
  - 3.2.6. Primitive Parsing and Pretty Printing . . . 29
  - 3.2.7. String Operations . . . 30
- 3.3. Implementation . . . 34
  - 3.3.1. Static Code Analysis . . . 34
  - 3.3.2. Code Generation . . . 35
  - 3.3.3. IDE Features . . . 39
- 3.4. Evaluation . . . 39
  - 3.4.1. Case Study . . . 39
  - 3.4.2. Evaluation of Correctness . . . 40
  - 3.4.3. Limitations . . . 41

4. Related Work **43**
4.1. Automatic Bidirectionalization of Functional Programs 43
4.2. Triple Graph Grammars 43
4.3. Inversion of ATL transformations 44
4.4. Lenses 44

5. Future Work and Conclusion **47**
5.1. Future Work 47
5.2. Conclusion 48

Bibliography **51**

A. Appendix **55**
A.1. Analysis of ATL Transformations 55

List of Figures

1.1. Example of Two Views with Semantic Overlap 2

3.1. AST of a simple assignment expression 13
3.2. Class Diagram of the Code Generation Implementation 36
3.3. Class Diagram of the Code Generation Implementation 37
3.4. Xtend Template Expression and the Produced Output 38

List of Tables

3.1. Categories of the ATL transformation zoo analysis 16
3.2. Number of mappings in the ATL transformation zoo by category 40
3.3. Implemented operations by category 40

List of Code Listings

2.1. MIR program modeling the vehicle and car example 6

3.1. Explicitly defining the updated source attribute using the update keyword 14
3.2. Xbase assignment compiled to Java 35
3.3. Generated code for the source to target transformation 35

A.1. Source code for the analysis of the ATL transformation zoo 55

1. Introduction and Motivation

1.1. Introduction

In Software Development, modeling languages like UML are used to design and specify the System under Development. To capture the whole complexity of a system, multiple views of different *view types* are used to represent different aspects or dimensions of the system. This is called *multi-view modeling* and different views can be worked on at different times or by different developers. Optimally, these different views are *orthogonal*, meaning they have no semantic overlap. However, this is often not the case in practice which is why the developers need to invest additional work into keeping the different views consistent. This task is error prone and adds complexity to the process.

View synchronization approaches try to eliminate the manual work in keeping models consistent, meaning that changes in one view are automatically applied to other views as well. This is done by applying model transformations which take a *source* model of one type and convert it into a *target* model of a different or same type. Model transformations stem from the Model Driven Software Development (MDSD) community and can be classified as either horizontal or vertical [18]. Vertical transformations are the ones where the target and the source model are of different abstraction level, for example transforming UML models into source code. In contrast, horizontal transformations transform models of the same abstraction level and are therefore relevant for view synchronization.

Changes between two views have to be propagated in both directions, therefore transformations in both directions are needed. Additionally, some consistency properties between the two transformations are desirable. This means manually writing view synchronization transformations merely shifts the error-proneness and complexity of keeping models consistent to the transformation developer. Automatic approaches are therefore developed which enable bidirectionality without writing redundant transformations which in turn improves the maintainability and correctness of the transformation code.

1.2. Motivation

As a motivation, consider figure 1.1 that demonstrates a car modeling scenario with two different views consisting of one meta class each. The meta classes “Car” and “Vehicle” contain attributes, some of which are related and need to be kept in sync, while others are only present in one of the two meta classes. For attributes that are related, we need to formulate transformations to propagate changes in both directions in order to keep the models synchronized.

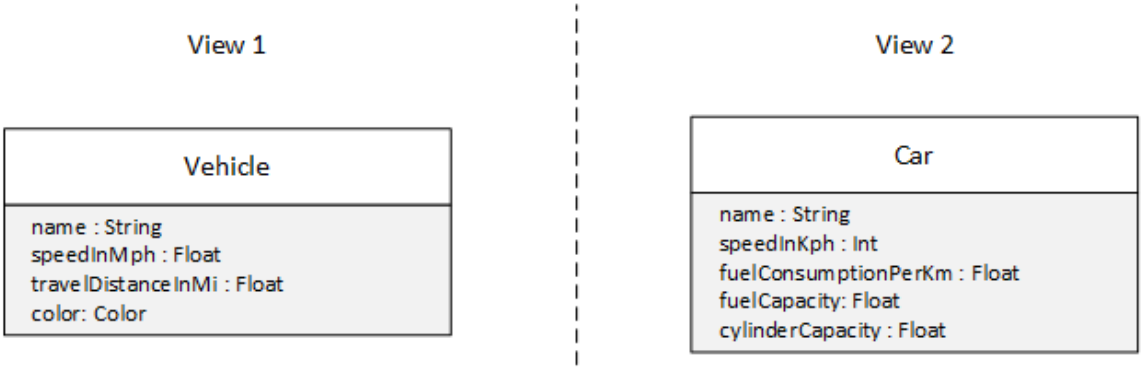

Figure 1.1.: Example of Two Views with Semantic Overlap

The meta classes “Car” and “Vehicle” both have an attribute called “name”. The attributes contain the same value and if one of them is changed, the other needs to be changed accordingly. We can trivially formulate the transformations in both directions as the identity function. Next, the attributes “cylinderCapacity” and “color” are only available in one of the two views, therefore no mapping is necessary.

More interestingly, both meta classes have an attribute for speed, however in the meta class “Vehicle”, the speed is in miles per hour while in “Car” it’s in kilometers per hour. Additionally, the “speedinMph” attribute is of floating point type while “speedinKph” is of integer type. We can formulate the transformation from “Car” to “Vehicle” as follows:

$$Vehicle.speedInMph = 0.62 * Car.speedInKph$$

Assuming the multiplication of a float value and an integer value is permitted and the result is of float type, the expression is correctly typed. To generate the backward transformation, one has to invert the multiplication with a constant factor resulting in the following expression:

$$Car.speedInKph = 1.61 * Vehicle.speedInMph$$

This expression however is not correctly typed, because the value on the right-hand side of the assignment is of float type and can’t be assigned to the integer attribute “speedInKph”. To fix this, a cast is required, however casting a float value to an integer value will discard the fractional part. This leads to a problem demonstrated by the following example: We first convert “speedInMph” (float) to “speedInKph” (integer) whereby the result has a fractional part that is discarded. We then immediately convert “speedInKph” back to “speedInMph”. The value of “speedInMph” has now changed although we merely applied the transformation in one direction and then in the opposite direction. Instead we would expect that, like in mathematics, the application of a function and then the application of the inverse function would yield the same result as the original value. This demonstrates the difficulty of writing *consistent* transformations meaning that the transformations obey some correctness properties.

Finally, the value of the travel distance in the meta class "Vehicle" is dependent on both the attributes for fuel consumption and fuel capacity in the meta class "Car". We can formulate the transformation in one direction as follows:

$$Vehicle.travelDistanceInMi = 0.62 * \frac{Car.fuelCapacity}{Car.fuelConsumptionPerKm}$$

In this case, it's not clear how to formulate the transformation in the opposite direction, because changes to the travel distance can either be reflected in the fuel capacity, the fuel consumption or both. Additionally, the expression on the right-hand side now consists of two nested operations (division, then multiplication). This increases the complexity of formulating the backward transformation.

1.3. Goals of this Thesis

Many existing model synchronization approaches like [22] allow the synchronization of models with different structures but similar semantics. In concrete terms, this means creating and deleting instances of corresponding meta classes between two models as well as creating, deleting and updating references between meta class instances. In contrast, we focus on the automatic synchronization of attributes between two meta class instances using sophisticated computation rules that go beyond assigning a constant value or the value of one attribute to another attribute. The goal is to enable the methodologist to write a single mapping that assigns the result of a possibly complex computation involving multiple attributes of one source meta class to an attribute of another target meta class. This mapping is treated as the forward transformation. A backward transformation is then automatically generated that takes a possibly changed target value as well as values for all the used source attributes and updates one explicitly defined source value.

For this purpose, a number of *operations* including numerical, logical and string operations that are relevant for a view synchronization scenario are selected and their effects as well as their *inversions* are defined. These basic operations are supposed to be combined and nested in the mapping expression. The defined operations and their inversions aim to fulfill a set of correctness properties that have been defined in similar, related approaches. They guarantee some desirable round trip invariants, enforcing that unchanged models on the source or target side are transformed into the same, unchanged model on the other side. When a changed target value is encountered that cannot be mapped to a source value without violating a round trip property, our approach allows to process it anyway with the goal of preserving as much of the information as possible. This is intended to offer a better user experience than limiting the set of accepted target values.

The implementation of our approach is integrated in the model consistency framework Vitruvius that automatically generates model transformations from declarative mapping expressions written in the domain-specific language MIR (for mappings, invariants and responses). Specifically, an approach is implemented that automatically generates bidirectional transformations between views from mapping expressions written in MIR.

2. Foundation

In this chapter, we discuss the foundations which our approach is based on. First, we discuss the view synchronization framework Vitruvius, which our approach is integrated in. Next, we discuss the View-Update problem which is the theoretical basis for our approach. Finally, we discuss Lenses, an approach which introduces a bidirectional programming language for synchronizing tree structures and which defines the correctness properties that our approach aims to fulfill.

2.1. Model Consistency Preservation with Vitruvius

To ease the task of keeping models consistent, the Eclipse-based framework *Vitruvius* [15, 16] is being developed at the Institute for Program Structures and Data Organization (IPD). Vitruvius is based on the concept of Orthographic Software Modeling (OSM), meaning that a single underlying model (SUM) holds the complete information about the system under development. However, the architecture of Vitruvius differs from the original description of OSM [3] in which the user facing models are views that are generated from the SUM. Instead, the views are independent models whose meta models are imported in a so-called *meta repository*. Using bidirectional transformations, changes from one model are propagated to the others which results in a consistent Virtual Single Underlying Model (VSUM). These bidirectional transformations are automatically generated from declarative mappings written in the domain specific language *MIR*.

The Vitruvius framework incorporates a two-step development process. In the first step, a so-called *methodologist* works with the Vitruvius framework, that enabled him to import the necessary meta models in the meta repository and to write the mappings in the MIR language. In the next step, this information is used to compile the modeling environment containing the generated model transformations which is also based on Eclipse and can be used for the actual modeling tasks.

2.1.1. The Domain Specific Language MIR

To maintain consistency between the views of the VSUM, the domain-specific language MIR is used to write declarative mappings between the models. In this thesis, an approach is developed that takes a mapping in the form of an assignment expression as input and generates a model transformation in the opposite direction, i.e. the assignment is inverted. The output is generated Java code. In addition, because the assignment follows a Java-compatible syntax, it can be compiled to Java code, as well. This way, a bidirectional transformation is generated. The framework is responsible for incrementally applying these transformations when the user changes a model.

The meta models that are used with the Vitruvius framework are required to be EMOF-based and the meta classes are implemented as Java classes. A MIR program maps features between these meta classes of two different meta models using multiple, possibly nested *Map blocks*. For each pair of mapped meta classes, a Map block is written. Map blocks consist of *With blocks* that contain the declarative mappings which are the focus of this thesis. The MIR language also defines the *Invariant* construct which is not in the scope of this thesis and which can be specified to restrict the mappings. When an Invariant is violated, a *Response* construct can be used to correct the meta models using imperative model transformation constructs.

2.1.2. Mapping Syntax

In the following, the syntax of the With block, the focus of this thesis, is discussed. An example for a Map block containing a With block can be seen in listing 2.1.

```
1 map view1.Vehicle as vehicle
2 and view2.Car as car {
3    with-block {
4       car.name = vehicle.name
5       vehicle.speedInMph = 0.62 * car.speedInKph
6    }
7 }
```

Listing 2.1: MIR program modeling the vehicle and car example

The program models a part of the vehicle and car example from section 1.2 and contains two *With expressions*. A With expression has the form of an assignment. The meta class whose attribute appears on the left-hand side of the assignment is called the *target* meta class of the respective With expression while the class whose attributes appear on the right-hand side is called the *source* meta class. In the first With expression of our example (line 4), *Car* is the target meta class while *Vehicle* is the source meta class. The opposite is true in the second With expression (line 5). This demonstrates that With expressions within the same With block can have either of the two mapped meta classes as target or source meta class.

The left-hand side of the assignment consists of an attribute of the target meta class. The expression on the right-hand side of the assignment can contain access to attributes of the source meta class, different *supported* operations, as well as numeric, boolean and string literals. The different operations can be nested to form a complex expression. The restrictions are that it must access at least one attribute of the source meta class and it must not access any attributes of the target meta class.

2.1.3. Used Technologies

The Vitruvius framework is implemented as a set of Eclipse plugins, therefore the development of the framework itself is also done in the Eclipse IDE using the Plug-in Development Environment (PDE). The MIR language is implemented using the Xtext SDK, an Eclipse

plugin and SDK for building domain-specific languages (DSLs). Xtext allows the definition of the language's grammar using an EBNF-like form. Additionally, it offers APIs for features like code completion and static code analysis. A language implemented using Xtext can be compiled to Java which is used for generating the previously discussed model transformations.

The syntax of the With expression is based on the Xbase grammar which is part of the Xtext SDK. Xbase is an expression language that is based on the Java type system and allows embedding statically typed expressions into an Xtext DSL [7]. It offers some syntactic sugar for commonly used constructs like access to getters and setters or declaring anonymous classes as lambda expressions. Additionally, it includes language infrastructure components which allow for parsing, interpreting and compiling Xbase expressions to Java code.

One aspect of the syntactic sugar is the attribute-like access to getters and setters. This means that we can use the assignment syntax

$$a.attribute = b.attribute$$

for accessing attributes and getters/setters of meta classes. An expression of this form can then be automatically compiled to the following Java code:

$$a.setAttribute(b.getAttribute())$$

The Vitruvius framework itself is written in Java and Xtend. Xtend [10] is a statically typed language that is compiled to and fully interoperable with Java. It is based on the Xbase syntax which means it offers the same syntactic sugar features. Additionally, it offers some language features like extensions methods, multiple dispatch and template expressions which helps reduce boiler plate and are well suited for writing compiler related programs.

2.2. The View-Update Problem

The view-update problem was defined and primarily studied by the relational database community in 80s. The motivation was to allow views of a database to be updatable, i.e. to find a *translation* that maps an update to a view to an update of the underlying database.

In their work from 1981 [4], Bancilhon and Spyratos defined a database schema S as a set of all database states $s \in S$. A view is defined as a function f that maps the database state s to a view state $f(s)$. The set $f(S) = \{f(s) \mid s \in S\}$ is then called the view schema. Finally, an update is defined as a function that maps a database state to a new database state.

Next, the authors define that for a view update u, a translation T_u is an update to the underlying database that fulfills the following properties:

1. *Consistency*: A translation of a view update takes the database to a state that maps exactly to the updated view.

2. *Acceptability*: A translation of an update, that doesn't change the view, doesn't change the database either.

A *translator* T is then defined as a mapping from a view update u to a translation T_u. The authors go on to show that for every correct translator, its translations leave "the information not visible within the view" unchanged. This "information not visible within the view" is called the *complement* to the view f and is also defined as a function g of the database state. For each s_1, s_2 with $f(s_1) = f(s_2)$ it is required that $g(s_1) \neq g(s_2)$. Thus, the tupled function (f, g), defined as $(f, g)(s) = (f(s), g(s))$, is isomorphic. Because of this, every database state can be unambiguously represented by a tuple $(f(s), g(s))$, i.e. the translator is defined as the inverted function $(f, g)^{-1}$ which maps an update u to the translation $T_u = (f, g)^{-1}(uf, g)$.

To fulfill the correctness properties, a transformation must leave $g(s)$ unchanged which is why these translations are called by the authors "translations under constant complement". However, different complements can be chosen for (f, g) to be isomorphic, thus the choice of g influences which updates are possible. For example, the identity function $(f, id)(s) = (f(s), s)$ is a legal choice for the complement function. However, every view update that changes the underlying database would lead to a change of the complement which is why no updates would be permitted at all. This leads to the definition of a minimal compliment which allows for the maximum "updatability" and in general isn't unique.

In their work from 1988, [12] Gottlob, Paolini, and Zicari define different correctness properties for view updates. They show that their class of *dynamic views* is a superset for updates under constant complement but also allow updates that cause a loss of information in the complement. They argue that this class of updates is relevant for real-world applications because they allow operations like deletions.

The view-update problem for databases can be seen as a special case of a general view-update problem for abstract data structures where some form of view of a data structure exists and changes to the view need to be reflected in the original data structure. This way it laid the theoretical foundation for other bidirectional transformation approaches that we discuss in this thesis as well as our own approach.

2.3. Lenses

In 2007, Foster et al. proposed a compositional solution to the view-update problem called *lenses* [9]. A lens consists of two functions called *get* and *putback*. Get is defined to map an element c of a *source* or *concrete set* C to an element a of a *target* or an *abstract set* A while putback maps a tuple (c, a') with $c \in C, a' \in A$ to an element $c' \in C$. The abstract set corresponds to the view in the view-update problem and the intuition is that some information is lost when c is mapped to a.

When a change to a is propagated back to the concrete set (hence the name putback) the original c is considered to restore the lost information. This differs from the definition of update translations by Bancilhon and Spyratos [4] where only the complement of a view that holds the lost information is considered.

The authors go on to define two properties called *well-behavedness* and *totality* that lenses should fulfill. A lens is well-behaved if the following two properties apply:

1. GetPut: Mapping a source c to a target a and then immediately back to the source yields an unchanged c.

2. PutGet: Mapping a target a and some c to a source c and then immediately back to the target yields the same a.

In other words, no information is lost during putback and no false information is created during get.

The definition of totality states that "get and putback functions are defined on all the inputs to which they may be applied". This corresponds to the mathematical definition of total functions.

Foster et al. state that their well-behavedness properties are equivalent to the consistency and acceptability properties of Bancilhon and Spyratos [4]. However the translations are not leaving the complement, in this case the actual source, unchanged. Therefore the lenses-approach can be classified as a dynamic-view approach as defined by Gottlob, Paolini, and Zicari [12].

Finally, the authors discuss a number of basic lenses as well as combinators that can be used to create higher level abstractions. In their work the sets C and A are are chosen to be tree structures, motivated by the fact that most hierarchical information can be represented that way.

The lenses approach is what inspired our approach in this thesis. When mapping the attributes of two meta classes, one of them is considered the *source* meta class and the other the *target* meta class. The source-to-target transformation is the direct application of the declared mapping and only needs to source meta class as input. It is allowed to "lose" information which needs to be restored in the backward transformation and is equivalent to the Get lens. The backward, target-to-source transformation requires an instance of both meta classes as input and is equivalent to the PutBack lens. Information that is lost can be restored that way.

In addition, we refer to the round trip laws GetPut and PutGet to define the correctness of our approach. Similar correctness properties are used by other bidirectional transformation approach that we discuss in the 4, like the ATL-based approach by Xiong et al. [27] or the inversion of functional programs by Matsuda et al. [17] who all use an equivalent to GetPut in addition to other properties that are at least similar to PutGet. Fundamentally, all these approaches offer solutions to the problem of inverting a forward transformation that loses information. Like them, we formulate the requirement that this lost information is correctly restored in the backward transformation by using the round trip laws.

As a differentiation, while the lenses approach focuses on synchronizing tree structures with tree structures, our approach focuses on mapping a number of attributes of numeric, boolean or string type to one target attribute.

3. Contribution

In this chapter, the contribution of this thesis, the inversion of the supported operations, is discussed. After outlining our approach, we define the correctness properties that our implementation aims to fulfill and how invalid input is handled in regards to them. We then define our supported operations and their inversion using a mathematical notation. Following this, we describe the concrete implementation of our approach using Java code generation. Finally we discuss an evaluation of our approach as well as its limitations.

3.1. Approach

3.1.1. Outline of the Approach

In this thesis an approach is developed that takes an assignment expression with an attribute of the target meta class on the left-hand side and a possibly nested expression on the right-hand side that contains at least one attribute of the source meta class and inverts that assignment so that a transformation can be generated that propagates a change in the target value back to the source attribute. Our approach is similar to solving a mathematical equation whereby the variable that the equation is solved for is the to-be-updated attribute of the source meta class. For this purpose a set of *supported* operations is chosen and their inversions are defined which aim to fulfill a set of correctness properties guaranteeing that applying the operation and it's inversion (or the other way round) will not alter the original value. The operations that we support are either operators or methods of the Java language and its standard library.

To demonstrate our approach, consider the following simple assignment which can be interpreted as an equation:

$$y = -x$$

The right-hand side of the equation consists of the unary minus operation. In order to solve the equation for x, we can transform it by applying the inverse operation to both sides, which is also the unary minus.

$$\begin{aligned} -y &= -(-x) \\ &= x \end{aligned}$$

We can see that the unary minus operation and its inverse operations cancel each other out, so that after simplifying the term on the right-hand side of the equals sign, only x is left. This way we can construct the inverted assignment which has the following form:

$$x = -y$$

Next, consider an operation with two arguments like the addition of two numbers:

$$y = x + 1$$

Again, if y changes, we want to reflect the change back in x. We transform the formula by subtracting 1 on both sides:

$$\begin{aligned} y - 1 &= x + 1 - 1 \\ &= x \end{aligned}$$

The inversion of adding 1 to a number is subtracting 1 from it. We can generalize this by defining the inversion of adding an arbitrary number z to a number x as follows:

$$\begin{aligned} y = add(x, z) &= x + z \\ add^{-1}(y, z) &= y - z \end{aligned}$$

An important observation is that the argument z now appears in both, the definition of the operation as well as its inversion. That's because by adding two numbers, the information about the individual addends is lost and there are infinitely many pairs of numbers whose sum is the same. The inverted operation needs to access this lost information to produce the correct result. We therefore define that inverted operations can access all of the arguments of the original operation in addition to the to-be-inverted value. This is equivalent to the Lenses approach by Foster et al. [9] where the input of the putback lens is a tuple of an abstract element and a source element. Consequently, our approach can be classified as a dynamic-view approach as defined by Gottlob, Paolini, and Zicari [12].

The addition of two numbers is a commutative operation, however this isn't the case for different operations like the division of two numbers. Therefore the definition of the inverted operation depends on which argument is updated. This leads us to the following syntax: An operation op with its arguments o_1, o_2, ... has the following form:

$$target = op(o_1, o_2, \ldots) = \ldots$$

An inverted operation has the following syntax:

$$o'_n = op_n^{-1}(target', o_1, o_2, \ldots) = \ldots$$

The first argument $target'$ of the inverted operation is the possibly changed result of the original operation. The remaining arguments are the arguments of the original operation. The index n of the inverted operation op_n^{-1} denotes which argument is inverted. If the operation only has one argument, the index is omitted. If an argument is not used in the inverted operation, its argument declaration can be omitted.

Now consider the following equation which contains multiple nested operations on the right-hand side:

$$y = (x * z) + w$$

Figure 3.1 visualizes the assignment as an abstract syntax tree (AST).

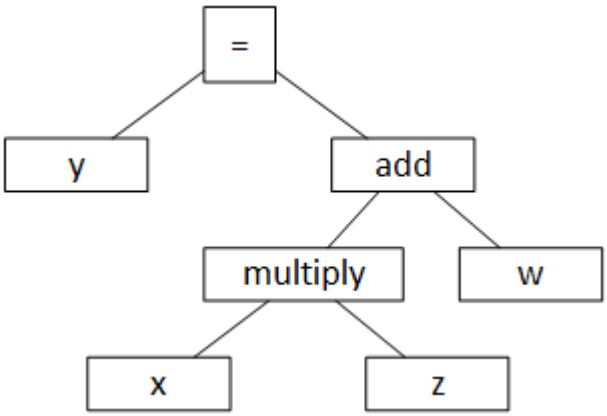

Figure 3.1.: AST of a simple assignment expression

Again, if the value of y changes, the change should be reflected back in x. We can rewrite the formula using the operations' names:

$$y = add(multiply(x, z), w)$$

Although the first argument of the *add* operation is another operation, we can transform it using the same inversion, we used previously:

$$tmp = add_1^{-1}(y, w) = multiply(x, z)$$

The result of the first inversion is assigned to a temporary variable, so that the formula can be simplified. In the next step, we apply the inversion of the first argument of the multiplication to receive a formula which only has x on one side:

$$multiply_1^{-1}(tmp, z) = x$$

This shows that inverting nested operations can be achieved by inverting the individual operations. This technique however only works if none of the variables are used more than once. This property is called "linear" [24] (or sometimes "affine" [17]) and is a limitation that we have to enforce in our approach.

To summarize, our approach is based on defining inversions for different supported operations which enables the methodologist to write complex mappings between one or multiple attributes of the source meta class and an attribute of the target meta class by nesting these supported operations. To generate the inverted transformation, we walk the right branch of the AST of the assignment down to the to-be-updated attribute and assign the result of the individual inverted operations to temporary variables. At the end, the last temporary variable contains the value that is assigned to the to-be-updated attribute. Our implementation generates Java code which implements the different inverted operations and assigns the final value to the to-be-updated source attribute.

Because we allow the right-hand side of a With expression to contain multiple attributes of the source meta class, we need a way to determine the to-be-updated attribute. For this reason we extend the syntax of the With expression by adding the *update* keyword. The syntax is demonstrated in listing 3.1.

```
1 map view1.Vehicle as vehicle
2 and view2.Car as car {
3    with-block {
4       vehicle.travelDistanceInMi = 0.62 *
5                           (car.fuelCapacity / car.fuelConsumptionPerKm)
6                           update fuelCapacity
7    }
8 }
```

Listing 3.1: Explicitly defining the updated source attribute using the update keyword

By marking one of the source attributes as *updated*, the methodologist declares that this attribute of the source meta class will be changed whenever the target attribute is changed, while all other source attributes in the respective With expression will stay constant. In terms of our car-vehicle example, changing the vehicle instance's travel distance will result in an updated fuel capacity of the car instance while the fuel consumption will stay unchanged. Of course, the fuel consumption value can still be changed manually and the change will be correctly reflected in in the travel distance.

We do not automatically decide which attribute should be updated, because the decision is often dependent on the specific domain the transformation is used in and our contribution is targeted for a general purpose framework. Therefore the automation of this decision is not in the scope of this thesis.

3.1.2. Correctness Properties

We previously discussed the Lenses approach by Foster et al. [9] and the correctness properties GetPut and PutGet that they introduced. In this thesis, we use these two properties for our inverted operations as well. The two laws state that performing a round trip in either direction, i.e. applying the operation and then the inversion or the other way round, should never change the original value.

An inverted operation fulfills **GetPut**, if the following is true for all o_i:

$$op_i^{-1}(op(o_1, \ldots, o_n), o_1, \ldots, o_n) = o_i,\ i \in \{1..n\}$$

If an operation only has one argument, the condition can be simplified to:

$$op^{-1}(op(o_1), o_1) = o_1$$

Similarly, an inverted operation fulfills **PutGet**, if for all values of $target$, the following applies:

$$op(op_1^{-1}(target, o_1, o_2, \ldots, o_n), o_2, \ldots, o_n) = target$$

$$\ldots$$

$$op(o_1, \ldots, o_{n-1}, op_n^{-1}(o_1, \ldots, o_{n-1}, target)) = target$$

If an operation only has one argument, the condition can be simplified to:

$$op(op^{-1}(target, o_1)) = target$$

The inverters are designed to always fulfill the GetPut law, meaning that calculating the target value and immediately updating the source value with the unchanged target value will always yield the same source value. This law can always be fulfilled by calculating the result of the operation with the unchanged source value and comparing it to the (possibly changed) value of the target attribute. If the result is the same, the source value is simply not changed. This technique will be used in those cases where some information is lost in the original operation, for example the fractional part during integer division.

Our source to target transformations are defined for all inputs that are correctly typed. Despite this, for some operations, there are inputs that cannot be processed, e.g. because they lead to exceptions. Because our operations can be nested, we would need to verify the input of every operation to prevent this. However, because our code generation approach, described in 3.3.2, is based on compiling the complete mapping expression in order to generate the source-to-target transformation, all kinds of exceptions are caught and wrapped in an internal exception instead.

The PutGet law states that calculating a source value from a target value and then calculating the target value anew must yield the same target value. This can only be fulfilled if calculating the source value from the target value doesn't involve a loss of information. This is not always the case, for example when casting a target value of type float to a source value of type integer. Differently put, if the to-be-inverted operation is not a surjective function, then PutGet can only be fulfilled if the updated target value is in the target set of the operation.

Target values that are not in the target set of the operation can be divided into two groups. First, there are values that can't be inverted at all. An example for this is parsing a string to an integer when the string does not contain a number. These cases are called **InversionErrors**. Second, there are values that can be inverted but some of the contained information is lost. The aforementioned float to integer casting example fits in this category because the fractional part will be ignored in the inversion. These cases are called **PutGetViolations** because performing a round trip will change the target value which violates the PutGet law.

A possible solution to handling both categories is simply prohibiting such target values. An alternative for prohibiting PutGetViolations is allowing the change to the target value and letting it be overridden by the next execution of the source-to-target transformation. The resulting target value will be different from the previous target value, however because some information of the target value is preserved, this can lead to a better user experience than prohibiting the change. The goal in our implementation will be to keep as much information of the target value as possible. An example is the absolute value operation which will discard the sign of negative target values, however the absolute value of the target value will stay unchanged after a round trip. InversionErrors can also be allowed by using a predefined default value (like 0). Again, after a round trip, the target value will

be overridden.

In our approach, the decision between these strategies is delegated to a separate implementation called a *violation handler*. A future implementation may display a dialog asking the user for a decision, choose automatically based on a predefined option, or do something completely different. The concrete implementation is not in the scope of this thesis. Instead, the generated code contains an additional parameter for the violation handler which is called if an invalid input is detected. Depending on the return value of the call to the violation handler, the respective strategy is chosen.

The syntax $PutGetViolation(d)$ and $InversionError(d)$, where d is the previously discussed default value, is supposed to be interpreted as "Depending on the violation handler, either reject the input or evaluate to d".

3.1.3. Supported Operations

In this thesis, our goal was to choose a set of operations that is relevant to a model synchronization scenario and to define and implement inversions for them. In order to validate the selection, we evaluated the commonly used attribute mappings in the model transformation community. Specifically, we evaluated the 103 model transformations from the transformation zoo found on the ATL website[1]. The source code of the analysis can be found in A.1. ATL is an unidirectional model transformation language that consists of imperative constructs as well as declarative mappings of attributes similar to MIR. We concentrated on the declarative mappings and, using regular expressions, categorized the mappings into 14 categories where a single expression is allowed to be categorized multiple times, i.e. multiple regular expressions could match a single mapping. We defined 14 categories using the following heuristics:

Group 1	
Identity	A direct mapping of an attribute
Arithmetic	Arithmetic operations or functions
ToString	A numeric or boolean value is pretty printed to a string
Parsing	A string is parsed to a numeric or boolean value
String Operations	Different operations on strings
Sequence	A sequence is constructed
List	List operations like filtering, mapping or extracting elements
Group 2	
Method or Attribute	A method call or attribute access and not in group 1
String Literal	The mapping only consists of a string literal
Conditional	If-Then-Else expressions
OCL	OCL specific operations

Table 3.1.: Categories of the ATL transformation zoo analysis

[1] http://www.eclipse.org/atl/atlTransformations/

We divided the 14 categories into two groups. The first group consists of operations that can be reasonably inverted. Our supported operations mostly fit in these categories. No operations of the Sequence and List categories are, however, supported. Using these basic operations, the methodologist can write complex mappings in a MIR program by nesting the operations. Using our approach, bidirectional model transformations are then automatically generated using Java code generation.

Sequence operations are meant to construct a sequence, i.e. a list from a number of values. List operations are used to map or filter existing lists or to retrieve values from a list, for example taking the first or last item. In this thesis we decided to create an extensible architecture and to focus on operations for primitive and string types. Therefore we don't support these operations, however in 5.1 we discuss possible approaches how to implement these operations in a future project.

Operations of the second group are not supported because of different reasons. In the "Method or Attribute" category, method calls were grouped that didn't belong to any of the group 1 categories. These methods were domain specific and therefore not relevant for this thesis. Additionally, this category contained mappings where the source attribute was of a complex type whose attribute was accessed. However, in this thesis we concentrated on primitive and string types which have no attributes.

In the mappings of the "String Literal" category, a constant string literal is mapped to a target attribute. This operation is not invertible, because we require that the mapping contains at least one source attribute.

Mappings from the "Conditional" category contain conditional expressions in the if-then-else form. Depending on the condition, either the result of then-branch or the else-branch is assigned to the target attribute. Because the result of the condition can change, in order to fulfill PutGet, we would need to be able to update both branches of the conditional expression. However, in our approach, only one source attribute can be updated. Another example for this kind of operation that didn't appear in the ATL transformation zoo is a logical operation like the logical and. Consider the following mapping:

$$target = o_1 \wedge o_2$$

Assuming the values for o_1 and o_2 are both $false$, the value for $target$ is also $false$. If $target$ is now updated with the value $true$, the inverted operation would need to set both source attributes to $true$ in order to fulfill PutGet.

The final category of group 2, the "OCL" category, contains OCL specific operations like "oclIsTypeOf" or "oclIsUndefined". These operations are roughly equivalent to Java's "instanceof" and null checks and map an arbitrary type to a boolean value. We could theoretically invert this operation and make it fulfill PutGet by using the technique described in 3.1.2. However when the target value changes, there is no canonical source value, the target value could be transformed into. Consider an example where the target value is null and the null check operation evaluates to $true$ which is assigned to the target value. Now the target value is changed to $false$. In order to fulfill PutGet, the target to source transformation needs to create a value of the correct type. But how should the value be created and initialized? The question can be answered in different

ways, however there is no canonical way, i.e. there is no universal way to instantiate an object of arbitrary type have different constructors or because the object could be an enum or a singleton. For these reasons, we decided to not support these kinds of operations.

For those operations, that we support, we aimed to define and implement inversions for all arguments. In some cases however, this was not feasible, because the inversion for some of the arguments would fall into one of the previously discussed categories from group 2. In those cases, where we do not invert all arguments, we discuss the reason for this.

3.2. Inverted Operations

In this section, the different supported operations and their inversions are discussed. The operations we support are Java's built-in operators like the numeric operators or methods defined in the Java standard library. When the operations can be defined more precisely in a mathematical notation, we provide the definition, otherwise we limit the definition to the inverted operations. To explicitly state the type of an expression e the syntax $e : T$, where T is the type, will be used. When we refer to numeric types, we mean Java's byte, short, int, long, float and double types. The char type is not considered, because it is not usually used for arithmetic operations.

3.2.1. Unary Primitive Operators

The first set of invertible expressions consists of unary operations for arithmetic and logical primitive types. To start off, the logical not is trivially defined as follows:

Definition 3.1 *Logical Not*

$$not(o_1) : boolean = \neg(o_1 : boolean)$$
$$not^{-1}(target) : boolean = \neg(target : boolean)$$

Similarly, the unary minus is defined as follows:

Definition 3.2 *Unary Minus*

$$minus(o_1) : T = -(o_1 : T)$$
$$minus^{-1}(target) : T = -(target : T)$$
$$T \in \{byte, short, int, long, float, double\}$$

Like many other arithmetic operations, the unary minus is defined for all of Java's numeric types.

We can see that the inverted operations don't access the previous value of the single argument because the operations are injective which means no information is lost in the Get direction.

3.2.2. Basic Arithmetic Operations

In this section, binary operations for numeric primitive types are discussed. For now, we require that the arguments and the target have the same type.

First, the addition and multiplication of two values is discussed. These operations are defined for all numeric types. Because these operations are commutative, only the inversion of o_1 is defined, the definition for the inversion of o_2 is analog. Using basic arithmetic rules, we define the operations as follows:

Definition 3.3 *Addition*

$$\begin{aligned} add(o_1, o_2) : T &= (o_1 : T) + (o_2 : T) \\ add_1^{-1}(target, o_2) : T &= target - o_2 \\ T &\in \{byte, short, int, long, float, double\} \end{aligned}$$

Definition 3.4 *Multiplication*

$$\begin{aligned} multiply(o_1, o_2) : T &= (o_1 : T) * (o_2 : T) \\ multiply_1^{-1}(target, o_2) : T &= target/o_2 \\ T &\in \{byte, short, int, long, float, double\} \end{aligned}$$

Subtraction is not commutative, which is why we define the inversion for the two arguments separately. Again, the operation is defined for all numeric types.

Definition 3.5 *Subtraction*

$$\begin{aligned} subtract(o_1, o_2) : T &= (o_1 : T) - (o_2 : T) \\ subtract_1^{-1}(target, o_2) : T &= target + o_2 \\ subtract_2^{-1}(target, o_1) : T &= o_1 - target \\ T &\in \{byte, short, int, long, float, double\} \end{aligned}$$

Division of float and double values is defined similarly.

Definition 3.6 *Division (Floating-Point)*

$$\begin{aligned} floatdivide(o_1, o_2) : T &= \frac{o_1 : T}{o_2 : T} \\ floatdivide_1^{-1}(target, o_2) : T &= target * o_2 \\ floatdivide_2^{-1}(target, o_1) : T &= o_1/target \\ T &\in \{float, double\} \end{aligned}$$

One can see that the GetPut law is fulfilled for the previously discussed operations because (in the absence of rounding errors) the value of the updated argument is unambiguously determined by the value of the target and the other argument. Otherwise put, the tupled function

$$f(o_1, o_2) = (op(o_1, o_2), o_2) = (target, o_2)$$

, where *op* is the operation, is injective.

This is not the case for byte, short, integer and long division (denoted by the symbol ÷). Because the fractional part of the result is lost, multiple values for the numerator (the first argument) can produce the same result even when the denominator (the second argument) is fixed. Consider the following proof which shows a violation of GetPut law when using the naive division inversion:

Proof 3.1 *Naive inversion of Integer Division violates GetPut*

$$o_1 : int := 9, o_2 : int := 2$$

$$\begin{aligned} target : int &= floatdivide(o_1, o_2) &&= o_1 &&\div o_2 = 9 \div 2 = 4 \\ o_1' : int &= floatdivide_1^{-1}(target, o_2) &&= target * o_2 &&= 4 * 2 = 8 \neq 9 \end{aligned}$$ □

This leads to the more complex definition of the division between values of integer types:

Definition 3.7 *Integer Division*

$$intdivide(o_1, o_2) : T = (o_1 : T) \div (o_2 : T)$$

$$intdivide_1^{-1}(target, o_1, o_2) : T = \begin{cases} o_1, & if\, o_1 \div o_2 = target \\ target * o_2, & otherwise \end{cases}$$

$$intdivide_2^{-1}(target, o_1, o_2) : T = \begin{cases} o_2, & if\, o_1 \div o_2 = target \\ o1 \div target, & otherwise \end{cases}$$

$$T \in \{byte, short, int, long\}$$

The inverted operation accesses both original arguments to check if the result equals the target value. This technique ensures that if the target is not changed, it is always mapped back to the original arguments which fulfills the GetPut law.

Note that when inverting the denominator in the Java-based implementation, we additionally need to ensure that the old denominator is not zero to prevent an exception caused by division by zero. Although zero is not a valid source value and would be rejected in the Get direction, this case needs to be handled anyway because when a new instance of the source meta class is created, the field can be initialized with a default value of zero.

3.2.3. Primitive Casts

To start off, we define the > relation on types as

$$T_1 > T_2 \Leftrightarrow T_1 \text{ is wider than } T_2$$

where *wider* is defined according to the Java Language Specification meaning a value of narrower type can be converted to a wider type and "does not lose information about the overall magnitude of a numeric value" [19]. Java's numeric types are therefore ordered as follows:

$$byte < short < int < long < float < double$$

We also define the notation

$$x \overset{\epsilon}{=} y \Leftrightarrow |x - y| < \epsilon \ (x \text{ is } \epsilon\text{-equal to } y)$$

to compare for equality tolerating small differences caused by rounding errors where ϵ is a sufficiently small value.

A primitive cast converts a value of one numeric type to another numeric type. It's important to distinguish between the cast operation for which we define an inversion and the actual cast implemented by the JVM. The latter is written as

$$cast_T(e)$$

where T is the type that the expression e is casted to.

A cast from T_1 to T_2 is called a widening cast if $T_2 > T_1$ and a narrowing cast if $T_1 > T_2$. The former is defined as follows:

Definition 3.8 *Widening Primitive Cast*

$$wideningcast(o_1) : T_2 = cast_{T_2}(o_1 : T_1)$$

$$wideningcast^{-1}(target) : T1$$

$$= \begin{cases} cast_{T_1}(target), & if\ cast_{T_2}(cast_{T_1}(target)) \overset{\epsilon}{=} target \\ PutGetViolation(cast_{T_1}(target)), & otherwise \end{cases}$$

$$T_1, T_2 \in \{byte, short, int, long, float, double\}$$

$$T_2 > T_1$$

The widening cast operation is not a surjective operation. Therefore, a check is necessary to determine if the target value is in the target set of the cast in the Get direction. This is done by casting the value to the source type and back to the target type. If the difference is not ϵ-equal, a PutGet violation is handled using the lossy casting result as default value.

Often, a widening cast operation is implicitly performed when an argument has a narrower type than expected. We detect these cases and insert the necessary inversion for the target-to-source transformation.

A narrowing cast is a surjective function and is therefore trivially defined as follows:

Definition 3.9 *Narrowing Primitive Cast*

$$
\begin{aligned}
narrowingcast(o_1) : T_2 &= cast_{T_2}(o_1 : T_1) \\
narrowingcast^{-1}(target) : T1 &= cast_{T_1}(target : T_2) \\
T_1, T_2 &\in \{byte, short, int, long, float, double\} \\
T_1 &> T_2
\end{aligned}
$$

3.2.4. Arithmetic Operations Between Different Types

Until now, for the arithmetic operations defined in 3.2.2 we required that arguments (and thus the target value) have the same type. We now extend the definition for arguments with different types.

According to the Java Language Specification [19], whenever a value of type T_1 is required, a value of type T_2 with $T_1 > T_2$ can be inserted. This is achieved through an implicit "Widening Primitive Conversion" which is equivalent to the value being cast to the wider type. In order to support this language feature in our framework, we insert a cast operation (and it's inversion) wherever an implicit cast happens. Consequently, the return type of an operation (and thus the type of the target) is the wider of the types of the arguments.

For the following definition, we assume that o_1 has the wider type. The definition for the opposite case is analog and is therefore omitted. Using the cast operations defined in 3.2.3, we define the operations from 3.2.2 for arguments with different types as follows:

Definition 3.10 *Basic Arithmetic Operations With Different Types*

$$
\begin{aligned}
op'(o_1, o_2) : T_1 &= op(o_1 : T_1, wideningcast(o_2 : T_2) : T_1) \\
op_1'^{-1}(target, o_1, o_2) : T_1 &= op_1^{-1}(target, o_1, wideningcast(o_2)) \\
op_2'^{-1}(target, o_1, o_2) : T_2 &= wideningcast^{-1}(op_2^{-1}(target, o_1, wideningcast(o_2))) \\
op &\in \{add, multiply, subtract, floatdivide, intdivide\} \\
T_1, T_2 &\in \{byte, short, int, long, float, double\} \\
T_1 &> T_2
\end{aligned}
$$

As an important note regarding the two division operations, If one of the arguments' types is float or double, $floatdivide$ is used, otherwise $intdivide$ is used.

3.2.5. Advanced Arithmetic Operations

After having discussed basic arithmetic operations as well as type casts, we introduce some advanced arithmetic operations that build upon the basic techniques.

First, the exponentiation operation is discussed. The result type of the exponentiation operation is always double which is mandated by the API of the respective methods in Java's standard library. We allow the base to be of any type, however the exponent is

required to be of integer (i.e. byte, short, integer or long) type. This restriction was chosen because it simplifies the definition and implementation and is not expected to have a significant impact on real-world applications.

First we define the helper functions *signum* and *abslog* as follows:

$$signum(a) = \begin{cases} 1 & \text{if } a >= 0 \\ -1 & \text{otherwise} \end{cases}$$

$$abslog_b(a) = \begin{cases} 0 & \text{if } a = 1 \\ log_{|b|}(|a|) = \dfrac{log(|a|)}{log(|b|)} & \text{otherwise} \end{cases}$$

Note, that unlike Java's signum method, our *signum* function returns 1 for an input of 0. Using these helper functions, the exponentiation operation is defined as follows:

Definition 3.11 *Exponentiation*

$$pow(o_1, o_2) : double = (o_1 : T_1)^{o_2 : T_2}$$

$$pow_1^{-1}(target, o_1, o_2) : T_1$$

$$= \begin{cases} \begin{cases} signum(o_1) * \sqrt[o_2]{target} & if\ target \geq 0 \\ PutGetViolation(signum(o_1) * \sqrt[o_2]{|target|}) & otherwise \end{cases} & if\ o_2\ is\ even \\ signum(target) * \sqrt[o_2]{|target|} & otherwise \end{cases}$$

$$pow_2^{-1}(target, o_1, o_2) : T_2$$

$$= widening cast^{-1}\Big(\begin{cases} o_2, & if\ o_1^{o_2} = target \\ abslog_{o_1}(target), & if\ o_1^{abslog_{o_1}(target)} \overset{\epsilon}{=} target \\ PutGetViolation(abslog_{o_1}(target)), & otherwise \end{cases} \Big)$$

$$T_1 \in \{byte, short, int, long, float, double\}$$
$$T_2 \in \{byte, short, int, long\}$$

The inversion of the base distinguishes between three cases:

1. If the exponent is even and the target is greater or equals than zero, then the result is the o_2-nd root of the target and has the sign of the original base. This way, GetPut is fulfilled even for negative bases.

2. If the exponent is even and the target is less than zero, the result would be an imaginary number which we forbid, so an InversionError is handled. The default value is the same as in case 1 except we take the absolute value of the target. After applying Get to the result, the target value will have the same absolute value, but a positive sign.

3. If the exponent is odd, the result is the o_2-nd root of the absolute value of the target and has the same sign as the target.

The inversion of the exponent is defined using the helper function *abslog* which has the following property: If a real exponent exp exists, so that

$$o_1^{exp} = target$$

holds true, then exp can be calculated as $exp = abslog_{o_1}(target)$. Otherwise

$$o_1^{abslog_{o_1}(target)} = -target$$

will hold true. Otherwise put, *abslog* produces the correct real exponent, if one exists. We check for the existence of a real exponent by comparing

$$o_1^{abslog_{o_1}(target)} \overset{\epsilon}{=} target$$

(again, tolerating small rounding errors) and if none exists, an InversionError is handled. We choose $abslog_{o_1}(target)$ as the default value because after a round trip only the sign of the target value will change while the absolute value is kept constant.

Additionally, the inversion deals with the following problems:

1. If $|o_1| = 1$ and $|target| = 1$, there are possibly an infinite number of solutions.

2. If $|o_1| = 1$, then $log(|o_1|) = 0$ which will lead to division by zero.

The first measure is to check if the original exponent is still correct. This deals with issue 1 in which case there are an infinite number of solutions, however in order to fulfill GetPut the result needs to be equal to the original exponent. Furthermore, this prevents some cases of issue 2. Finally, this deals with the edge case, in which $target = 0$ and $o_1 = 0$, where, again, there are an infinite number of solutions but the logarithm is not defined.

To show, that the equality check is not enough to deal with all instances of problem 2, consider the case where $target = 1, o_1 = -1$ and $o_2 = 3$. We see, that the original exponent is not correct anymore, however calculating

$$\frac{log(|1|)}{log(|-1|)}$$

would lead to division by zero. We prevent this through the conditional definition of *abslog*, where an input of 1 is mapped to a result of 0 which handles the remaining instances of issue 2.

The final result is however not necessarily a whole number, which we require. We reuse the inverted narrowing casting operation from definition 3.10 to safely cast the number to integer type T_2 handling an additional PutGetViolation if necessary.

Next, we define the absolute value operation:

Definition 3.12 *Absolute Value*

$$abs(o_1) : T = |o_1 : T|$$

$$abs^{-1}(target, o_1) : T = \begin{cases} signum(o_1) * target & if\ target >= 0 \\ GetPutViolation(signum(o_1) * |target|) & otherwise \end{cases}$$

$$T \in \{int, long, float, double\}$$

In order to fulfill GetPut, the inversion preserves the sign of the source attribute. For values < 0, a PutGetViolation is handled, where after a round trip, the absolute value of the target will stay constant while the sign will change.

Java's absolute value method is not implemented for the byte and short types. If you feed a value of type byte or short into the method, a Widening Primitive Conversion to int happens and the int variant of the method will be invoked. Using the *wideningcast* operation defined in 3.2.3 we define the absolute value operation for values of byte and short types:

Definition 3.13 *Absolute Value (Implicit Cast)*

$$abs(o_1) : int = |wideningcast(o_1 : T) : int|$$

$$abs^{-1}(target, o_1) : T = wideningcast^{-1}(abs^{-1}(target, wideningcast(o_1) : int))$$

$$T \in \{byte, short\}$$

This isn't technically a separate operation because the implicit *wideningcast* operation is merely nested in the absolute value operation and as we described in 3.1.1, nested operations can be inverted separately. For this reason, we don't provide any more similar definitions for operations whose arguments are implicitly casted.

Next, we define the *round*, *ceil* and $floor$ operations which convert a floating point number to the nearest, next or previous whole number. We use the notation $[a]$ for rounding to the nearest whole number, $\lceil a \rceil$ for rounding up and $\lfloor a \rfloor$ for rounding down. The operations are defined as follows:

Definition 3.14 *Rounding*

$$round(o_1) : int = [o_1 : T]$$

$$round^{-1}(target, o_1) : T = \begin{cases} o_1 & if\ [o_1] \overset{\epsilon}{=} target \\ cast_T(target) & otherwise \end{cases}$$

$$T \in \{float, double\}$$

Definition 3.15 *Floor*

$$floor(o_1) : double = \lfloor o_1 : double \rfloor$$

$$floor^{-1}(target, o_1) : double = \begin{cases} o_1 & if \lfloor o_1 \rfloor \overset{\epsilon}{=} target \\ target & if \lfloor target \rfloor \overset{\epsilon}{=} target \\ PutGetViolation(target) & otherwise \end{cases}$$

Definition 3.16 *Ceiling*

$$ceil(o_1) : double = \lceil o_1 : double \rceil$$

$$ceil^{-1}(target, o_1) : double = \begin{cases} o_1 & if \lceil o_1 \rceil \overset{\epsilon}{=} target \\ target & if \lceil target \rceil \overset{\epsilon}{=} target \\ PutGetViolation(target) & otherwise \end{cases}$$

The first noticeable difference between the operations is that *round* expects a value of type float or double and returns a value of type int while $floor$ and *ceil* expect and return a value of type double. This is, again, in line with Java's implementation of the methods. We can however make use of Implicit Widening Conversions, e.g. to feed a value of type float into $floor$ or *ceil*.

A consequence of the return type of *round* being int is that the function is surjective. This is the reason why $round^{-1}$ doesn't need to check that the to-be-inverted target value is a whole number while $floor^{-1}$ and $ceil^{-1}$ do. If the input of $floor^{-1}$ or $ceil^{-1}$ is not a whole number, a PutGetViolation is handled using the (not-whole) target value as a default value. After a round trip, the value will be rounded up or down depending on the operation.

Next, we define the modulo operation. This operation returns the remainder of an integer or long division $o_1 \div o_2$. Java implements two versions of the modulo operation. On the one hand, there is the "%" operator which is defined for all numeric types, on the other hand, there is the static function $floorMod$, defined in the *Math* class which was introduced in the Java 8 release. The latter is only defined for the int and long types. Both versions return a value in the open interval $(-|o_2| \,..\, |o_2|)$, however unlike the operator, the $floorMod$ function's result always has the sign of the divisor o_2.

In our implementation we chose to support the $floorMod$ function because often times the modulo operation is used for calculating the index of circular data structures and even for negative dividends one expects a positive result. Because we described the behavior of the operation textually, we omit the mathematical definition. The inversion of the $floorMod$ operation is therefore defined as follows:

Definition 3.17 *Modulo*

$$floorMod_1^{-1}(target, o_1, o_2) : T$$
$$= \begin{cases} o_1 & if\ floorMod(o_1, o_2) = target \\ target & if\ floorMod(target, o_2) = target \\ PutGetViolation(target) & otherwise \end{cases}$$
$$floorMod_2^{-1}(target, o_1, o_2) : T$$
$$= \begin{cases} o_2 & if\ floorMod(o_1, o_2) = target \\ target + 1 * signum(target) & if\ o_1 = target \\ o_2' & if \exists\, o_2' \in \mathbb{Z} \,.\, floorMod(o_1, o_2') = target \\ InversionError(1) & otherwise \end{cases}$$
$$T \in \{int, long\}$$

The inversion of the first argument preserves the original value if the target value is unchanged, otherwise it returns the changed target value. The check is necessary to fulfill GetPut, because the $floorMod$ operation is not injective, i.e. multiple o_1 will be mapped to the same target value. If the target value is not in the target set of the original operation, i.e. for positive o_2 it's not in $[0 \,..\, o_2 - 1]$ and for negative o_2 it's not in $[o_2 + 1 \,..\, 0]$, a PutGetViolation is handled. The default value is the target value and after a round trip, it will become the remainder of the old value divided by o_2.

The inversion of the second argument also preserves the original value if the target value is unchanged. This is necessary, because multiple o_2 can also be mapped to the same target value and our implementation would otherwise always return the result that is closest to zero.

If the target value has changed and it is equal to the dividend o_1, every value that is further away from 0 than o_1 would fulfill PutGet, because for every o_1, o_2 with the same sign and $|o_1| < |o_2|$, $floorMod(o_1, o_2) = o_1$. We chose to return the next whole number that is further away from 0.

If this is not the case, we search for a value o_2', so that $floorMod(o_1, o_2') = target$. We know that if an o_2' exists, then $|o_2'| < |o_1|$ must be true because otherwise $target = o_1$ would be true and we handled that case earlier. Because the results of $floorMod(o_1, o_2')$ are in the open interval $(-|o_2'| \,..\, |o_2'|)$ and $|o_2'| < |o_1|$, we know that an o_2' can only exist if $target < o_1$. If it's not, we can immediately handle an InversionError. Otherwise, we start at 1 or -1 and iterate over all whole numbers in the half of the interval with the same sign as $target$. If a result is found, we return it, otherwise an InversionError is handled.

The choice for a default value for an InversionError is arbitrary, however it mustn't be 0 which is forbidden as the second argument to $floorMod$ and will cause a division-by-zero exception. Instead we chose to return 1.

Next, we discuss the trigonometrical operations sine, cosine, tangent as well as the operations for their inverse functions. The operations in Java's standard library are named like the well-known corresponding mathematical functions, so in the following, we limit

the definition to the inverted operations.

We first define the inversions of the three trigonometrical operations sine, cosine and tangent as follows:

Definition 3.18 *Sine*

$$sin^{-1}(target, o_1) : double = \begin{cases} o_1 & if\, sin(o_1) \overset{\epsilon}{=} target \\ asin(target) & if\, -1 \leq target \leq 1 \\ InversionError(0) & otherwise \end{cases}$$

Definition 3.19 *Cosine*

$$cos^{-1}(target, o_1) : double = \begin{cases} o_1 & if\, cos(o_1) \overset{\epsilon}{=} target \\ acos(target) & if\, -1 \leq target \leq 1 \\ InversionError(0) & otherwise \end{cases}$$

Definition 3.20 *Tangent*

$$cos^{-1}(target, o_1) : double = \begin{cases} o_1 & if\, cos(o_1) \overset{\epsilon}{=} target \\ acos(target) & otherwise \end{cases}$$

All three operations expect a value of type double and return a value of the same type. The functions are periodic, e.g. the inputs 0 and 2π lead to the same result. This is why, in order to fulfill GetPut, we need to compare the target value to the to-be-updated argument. Because sine and cosine return values between -1 and 1, i.e. they're not surjective, we also need to check if the target value is in this interval. If you input a value outside this interval into the function, the value *NaN* (for not a number) is returned in the Java implementation. If target has such a value, an InversionError is handled. This does not apply to tangent, which is surjective, i.e. its inverse function is defined for all double values.

In addition to the trigonometric operations, we also define the inverted operations for the inverse trigonometric functions arcsine, arccosine and arctangent as follows:

Definition 3.21 *Arcsine*

$$asin^{-1}(target, o_1) : double = \begin{cases} sin(target) & if\, \frac{-\pi}{2} \leq target \leq \frac{\pi}{2} \\ PutGetViolation(sin(target)) & otherwise \end{cases}$$

Definition 3.22 *Arccosine*

$$acos^{-1}(target, o_1) : double = \begin{cases} cos(target) & if\, 0 \leq target \leq \pi \\ PutGetViolation(cos(target)) & otherwise \end{cases}$$

Definition 3.23 *Arctangent*

$$atan^{-1}(target, o_1) : double = \begin{cases} tan(target) & if\, \frac{-\pi}{2} \leq target \leq \frac{\pi}{2} \\ PutGetViolation(tan(target)) & otherwise \end{cases}$$

The inverse trigonometrical functions are not surjective, e.g. arccosine returns values between 0 and π. This is why we need to check if the to-be-inverted target value is in the target set of the original operation. If it's not, we handle a PutGetViolation. The default value is still the function applied to the target value, because sine, cosine and tangent are defined for all double values. After a roundtrip, the target value will lie in the respective interval.

3.2.6. Primitive Parsing and Pretty Printing

Pretty printing means converting a value of primitive type to a textual representation. Xbase allows using Java's "toString" method on numeric and boolean values. When the Xbase code is compiled, in the resulting Java code, the primitive value will first be converted to a boxed type by using the static "valueOf" method and then the "toString" method on the boxed value will be called. For our definition we will use the functional notation $toString(x)$. The inverse operation is called parsing. For each primitive type (including boolean), a static parse method exists in the respective boxed class. For example, the method for parsing an integer is called "parseInt" and is located in the "Integer" class. In our definition, we will use the notation $parse_T(e)$.

Because parsing is a surjective function, the operation and its inversion is trivially defined as follows:

Definition 3.24 *Parsing*

$$\begin{aligned} parse(o_1) : T &= parse_T(o_1 : String) \\ parse^{-1}(target) : String &= toString(target : T) \\ T \in \{&boolean, byte, short, int, long, float, double\} \end{aligned}$$

On the other hand, pretty printing is not surjective. Java's parse methods for numeric types will throw an exception if the value cannot be parsed. In our implementation we catch this exception and forward it to the violation handler. If the exception is handled, a default value of 0 is chosen instead. For types $T \in \{byte, short, int, long, float, double\}$ the operation of pretty printing is defined as follows:

Definition 3.25 *Pretty Printing Numeric Values*

$$toString(o_1) : String \quad = \text{decimal representation of } o_1$$

$$toString^{-1}(target) : T = \begin{cases} parse_T(target : String), & \text{if target is valid number} \\ InversionError(0), & \text{otherwise} \end{cases}$$

$$T \in \{byte, short, int, long, float, double\}$$

Parsing boolean values works the same way, except the parse method never throws an exception. Instead, everything except "true" (case insensitive) is parsed to *false*. Therefore, the definition is simply:

Definition 3.26 *Pretty Printing Boolean Values*

$$toString(o_1) : String \quad = \begin{cases} "true", & if\ o_1 = true \\ "false", & otherwise \end{cases}$$

$$toString^{-1}(target) : boolean = parse_{boolean}(target : String)$$

$$= \begin{cases} true, & \text{if target equals "true" (case insensitive)} \\ false, & otherwise \end{cases}$$

3.2.7. String Operations

In the following, operations on strings are discussed. We start by discussing the inversion of string concatenation. For the concatenation of two strings a and b we use the following syntax:

$$a \cdot b$$

In Xbase, like in Java, string concatenation is achieved using the "+" operator on string values. The operation and its inversion is defined as follows:

Definition 3.27 *String Concatenation*

$$concat(o_1, o_2) : String \quad = (o_1 : String) \cdot (o_2 : String)$$

$$concat_1^{-1}(target, o_2) : T_1 = \begin{cases} o_1', & if\ target = o_1' \cdot o_2 \\ PutGetViolation(target), & otherwise \end{cases}$$

$$concat_2^{-1}(target, o_1) : T_2 = \begin{cases} o_2', & if\ target = o_1 \cdot o_2' \\ PutGetViolation(target), & otherwise \end{cases}$$

The inverted operation checks if the target value starts, respectively ends, with the not inverted argument. If it does, the result is the substring of the target value that you get if you remove the affix. If the affix is however not present in the target string, a PutGetViolation is handled where the target value acts as the default value. This means that

after a round trip, the target value will become the old target value concatenated to the affix.

Next, we define the inversion of the substring operation. Again, we use the functional notation $substring(s, beginIndex, endIndex)$ to refer to the equivalent Java method invocation $s.substring(beginIndex, endIndex)$. In our definition, we use the function $length(s)$ which returns the number of characters in the string s which represents Java's $s.length()$. Based on the definition of the string concatenation, we also define s^n as the string s concatenated with itself n times. Next, we define $s[n]$ as the nth character in the string s. Furthermore, we define the following helper functions:

$$prefix(s : String, e : int) : String = \begin{cases} substring(s, 0, e), & \text{if } length(s) \geq e \\ \text{"\_"}^e, & \text{otherwise} \end{cases}$$

$$suffix(s : String, b : int) : String = substring(s, min(length(s), b), length(s))$$

$$reconcat(t : String, s : String, n : int, m : int) : String = prefix(s, n) \cdot t \cdot suffix(s, m)$$

$$pad(t : String, l : int) : String = t \cdot \text{"\_"}^{(l - length(t))}$$

To save space, in the following, we abbreviate $PutGetViolation$ with PGV. Using the previously defined helper functions, we define the substring operation as follows:

Definition 3.28 *Substring*

$$substring(o_1, o_2, o_3) : String = o_1[o_2] \cdot \ldots \cdot o_1[o_3 - 1]$$

$$substring_1^{-1}(target, o_1, o_2, o_3) : String$$

$$= \begin{cases} reconcat(target, o_1, o_2, o_3), & if\, length(target) = o_3 - o_2 \\ PGV(reconcat(target, o_1, o_2, o_3)), & if\, length(target) > o_3 - o_2 \\ PGV(reconcat(pad(target, o_3 - o_2), o_1, o_2, o_3)), & otherwise \end{cases}$$

When inverting the substring operation, we concatenate the cut off prefix and suffix of the original argument to the target value. The position and length of the cut is fixed by the second and third arguments, so we need to restore the prefix and make sure that the target value has the correct length.

First, when extracting the prefix of the original argument using the $prefix$ helper function, we make sure that the original argument's length is equal to or greater than the length of the assumed prefix. If this condition is not met, e.g. when instantiating an empty source meta class, we use a placeholder instead. For the placeholder, we concatenate the underscore character "_" o_2 times with itself. This is necessary to fulfill PutGet, because the index of the cut is fixed and if we omitted the prefix, a part of the target value would be cut off after a round trip.

When extracting the suffix and the original argument's length is smaller than the assumed position of the cut, $suffix$ will return the empty string. We don't need to use a placeholder in this case, because the length of the suffix doesn't affect the result of the substring. However we need to prevent accessing out-of-bound indices to not cause exceptions.

Next, we make sure that the target value's length is equal to the length of the cut. If it is, we use the *reconcat* helper function, which extracts the pre- and suffix of the original argument and concatenates them to the target value. If this is not the case, we handle a PutGetViolation. This is necessary, because, again, the indices of the cut are fixed and the target value will have the length $o_3 - o_2$ after a round trip. If the target value is longer than that, we can use the result of *reconcat* as the default value which will be cut to the correct size after a round trip. However, if the length of the target value is smaller, we first need to append characters to it to make up for the difference. Otherwise an index-out-of-bounds exception could occur when performing a round trip. This is handled by the *pad* function which appends enough underscore characters as placeholders to reach the desired length.

Note that we don't support inverting the second and third argument of *substring* because it carries too little information as described in 3.1.3.

In addition to the substring function with a start and an end index, Java offers an implementation that that only takes a start index as input and therefore cuts of a prefix and returns the whole suffix. We support this version as well, the operation is defined as follows:

Definition 3.29 *Substring (Suffix)*

$$substring(o_1, o_2) : String \quad = o_1[o_2] \cdot \ldots \cdot o_1[length(o_1) - 1]$$
$$substring_1^{-1}(target, o_1, o_2) : String = prefix(o_1, o_2) \cdot target$$

The definition of the two-argument version of *substring* is simpler than the version with three arguments. Because the length of the cut is not fixed anymore, the target value can now have any length. Also, because no suffix is cut off, none needs to be reattached.

Next, we define the length operation for a string which maps a string value to it's length as an int. As previously discussed, the operation represents Java's $s.length()$, therefore we omit the definition of the original operation. The inverted operation is defined as follows:

Definition 3.30 *String Length*

$$length^{-1}(target, o_1) : String$$
$$= \begin{cases} prefix(o_1, target), & if\, length(o_1) \geq target \\ pad(o_1, target), & otherwise \end{cases}$$

The inverted operations preserves the prefix of length $target$ of the original argument. If $target$ is equal to the length of o_1, this is the whole string. If the target value is greater than the length of the original argument, underscore characters are appended as placeholders

using the *pad* helper function so that the result has the correct length. Otherwise, PutGet would be violated.

Next, we define the *toUpperCase* and *toLowerCase* operations. The operations take an arbitrary string and convert all of its characters to the respective case. To save space, we abbreviate *toUpperCase* with *tUC* and *toLowerCase* with *tLC*. The inverted operations are abbreviated accordingly. The operations represent Java's *s.toUpperCase*() and *s.toLowerCase*, therefore we omit the definition of the original operation. The inverted operations are defined as follows:

Definition 3.31 *String to Upper Case*

$$toUpperCase^{-1}(target, o_1) : String$$
$$= \begin{cases} PGV(tUC^{-1}(tUC(target))), & if\ target \neq tUC(target) \\ x, & if\ \exists p, x, s\ .\ o_1 = p \cdot x \cdot s \wedge tUC(x) = target \\ tLC(p) \cdot o_1 \cdot tLC(s), & if\ \exists p, x, s\ .\ target = p \cdot x \cdot s \wedge tUC(o_1) = x \\ tLC(target), & otherwise \end{cases}$$

Definition 3.32 *String to Lower Case*

$$toLowerCase^{-1}(target, o_1) : String$$
$$= equivalent\ to\ toUpperCase^{-1}\ if\ toUpperCase\ and\ toLowerCase\ are\ replaced$$

In the following, we discuss the *toUpperCase* operation, the *toLowerCase* operation is defined analogously.

The inversion checks if the target value has the correct casing, i.e. it's completely in upper case. If it isn't, a PutGetViolation is handled. The default value is defined as a recursive call to the inverted operation itself, only with the value converted to upper case. This means, that after a round trip, the target value will be the same string, only in upper case.

In order to fulfill GetPut, it would only be necessary to return the original argument, if the target value is equal to it, ignoring the case. In all other cases, any string could be returned that would be equal to the target value when converted to upper case. A possible implementation could return the unaltered target value or the target value converted to lower case. However, in our implementation, the goal was to preserve as much of the original string as possible. For this purpose, we first check if characters were removed at the start or at the end of the target value. If this is true, there is a substring x of the original argument o_1 which, when converted to upper case, is equal to the new target value. If this is the case (pun not intended), this substring is then returned, preserving the case of its characters. This also applies if the target value was not changed, because the affixes p and s can be defined as the empty string. Alternatively, we check if characters were added at the start or at the end of the target value. In this case, we take the added characters, convert them to lower case and append them to the original argument. Finally, if none

of the previous applies, the target value is simply converted to lower case. Although, we don't handle replacements or insertions of characters, this implementation is arguably more user-friendly than always converting the target value to lower case.

3.3. Implementation

After discussing the theory of inverting operations in the previous sections, in this section the actual implementation is discussed. Our approach is integrated into the Vitruvius framework which is based on Xtext, a framework and SDK for building domain specific languages. Xtext parses the MIR program and generates an AST. Our implementation takes the AST of a With expression as input and processes it. The processing consists of static code analysis to verify that the expression is semantically correct as well as Java code generation.

3.3.1. Static Code Analysis

The Xtext based implementation of our approach uses static code analysis to verify that a given With expression is semantically correct, i.e. it fulfills all necessary preconditions to be inverted automatically. The following checks verify, that the expression fulfills the necessary preconditions:

- The left-hand side of the assignment consists exactly of one attribute of the target meta class. The attribute is called the target attribute.
- The right-hand side of the expression exclusively contains expressions, constants and attributes of the source meta class.
- The right-hand side of the expression contains at least one attribute of the source meta class.
- No attribute of the source meta class is used more than once on the right-hand side of the expression.
- An attribute to be updated is explicitly specified iff more than one attribute of the source meta class is used on the right-hand side of the expression
- The path to the to-be-updated attribute only contains expressions that can be inverted.

In other words, a With expression must contain an attribute of the target meta class on the left-hand side and only expressions, constants and differing attributes of the source meta class on the right-hand side while one source attribute with an invertible path is specified as to-be-updated. It should be noted that arbitrary expressions can however appear in other paths in the right-hand side expression because they don't need to be inverted.

Note that we don't need to check if the expression is syntactically correct or whether it only uses methods and identifiers that actually exist because this is automatically handled by the parser which is part of Xbase.

3.3.2. Code Generation

The code generation is the part that is responsible for translating a With expression in a MIR program to bidirectional model transformations in Java. For every With block, as described in 2.1.2, a Java class with two methods is generated. Each method is responsible for updating one of the two mapped meta classes. Because a With block can contain multiple With expressions where either of the meta classes acts as the source/target meta class, both methods receive instances of both meta classes as parameters. In addition, an instance of the violation handler, as described in 3.1.2, is passed as a parameter. The code generation is divided in two parts, first generating the transformation from the source meta class to the target meta class and then generating the transformation in the opposite direction.

Source to Target Transformation

Xtext offers a built-in Xbase to Java compiler which makes it trivial to generate a transformation from the source meta class to the target meta class. The Xbase compiler takes an Expression in its AST form as input and produces Java code in string form. In our implementation, the assignment expression that is part of the With expression is fed into the Xbase compiler which produces the appropriate Java code. For example, the simple mapping $aa.id = bb.id$ is translated to the following Java code:

```
1 String _id = bb.getId();
2 aa.setId(_id);
```

Listing 3.2: Xbase assignment compiled to Java

As described in 3.1.2, we don't generate explicit checks for the input of the source to target transformation. Instead we surround the code by a try-catch block and rethrow any exception as an internal exception called *InversionException*. The resulting code looks like this:

```
1 try {
2    String _id = bb.getId();
3    aa.setId(_id);
4 } catch (Exception e) {
5    throw new InversionException(e);
6 }
```

Listing 3.3: Generated code for the source to target transformation

Target to Source Transformation

The second part of the code generation which is generating the transformation from the target meta class to the source meta class involves the actual inversion of operations. In order to generate Java code, we could construct a new AST that contains the inverted operations and let the Xbase compiler generate Java code. However, the drawback of this approach is that this makes input validation as described in 3.1.2 harder to implement.

The reason is that we check the input for each inverted operation separately as opposed to once for the whole expression. For this reason, we generate Java code directly for each inverted operation.

In 3.1.1 we described that nested operations can be solved individually and the formula can be simplified by assigning the results of the individual inverted operations to temporary variables. This is how our code generation approach works. To demonstrate it, consider the following abstract assignment expression with two nested operations on the right-hand which should be inverted for x:

$$y = f(g(x, w), z)$$

The Java code that we generate then has the following structure:

$$\begin{aligned} tmp_0 &= y \\ tmp_1 &= f^{-1}(tmp_0, g(x, w), z) \\ tmp_2 &= g^{-1}(tmp_1, x, w) \end{aligned}$$

Every inverted operation accesses the previous temporary variable and the arguments of the original operation. The last temporary variable tmp_2 contains the value that will be assigned to the source attribute that is represented by x.

In the following we describe our architecture for generating Java code for the inverted operations.

To represent a path through the AST of an expression, we implemented the class *ExpressionPath*. The class consists of a list of *Operands*, an interface that allows to navigate from a node of the AST to one of its children. There are different kinds of expressions with different numbers of arguments, like operator invocations or method calls, for which there are different implementations of *Operand*. The *Operand* holds the index of the child it will navigate to, but also allows retrieving the other children.

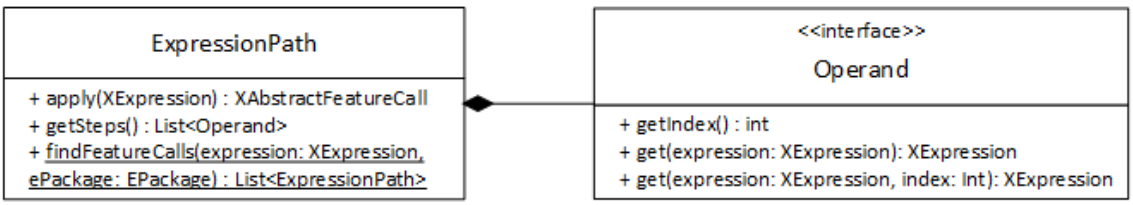

Figure 3.2.: Class Diagram of the Code Generation Implementation

For inverting the individual operations, as described in 3.2, we wrote a number of classes that implement the interface *OperationInverter*. The interface has exactly one method which is responsible for generating the Java code for the inversion. In addition to the to-be-inverted expression, the method takes an instance of *Operand* as argument. This way it can retrieve the inverted and all other arguments of the operation. Additionally, the

method takes as a parameter an instance of *ITreeAppendable* which is an internal data structure for generating code. The inverters are mostly parameterizable so that a single class is responsible for inverting multiple related methods with possibly different types of arguments.

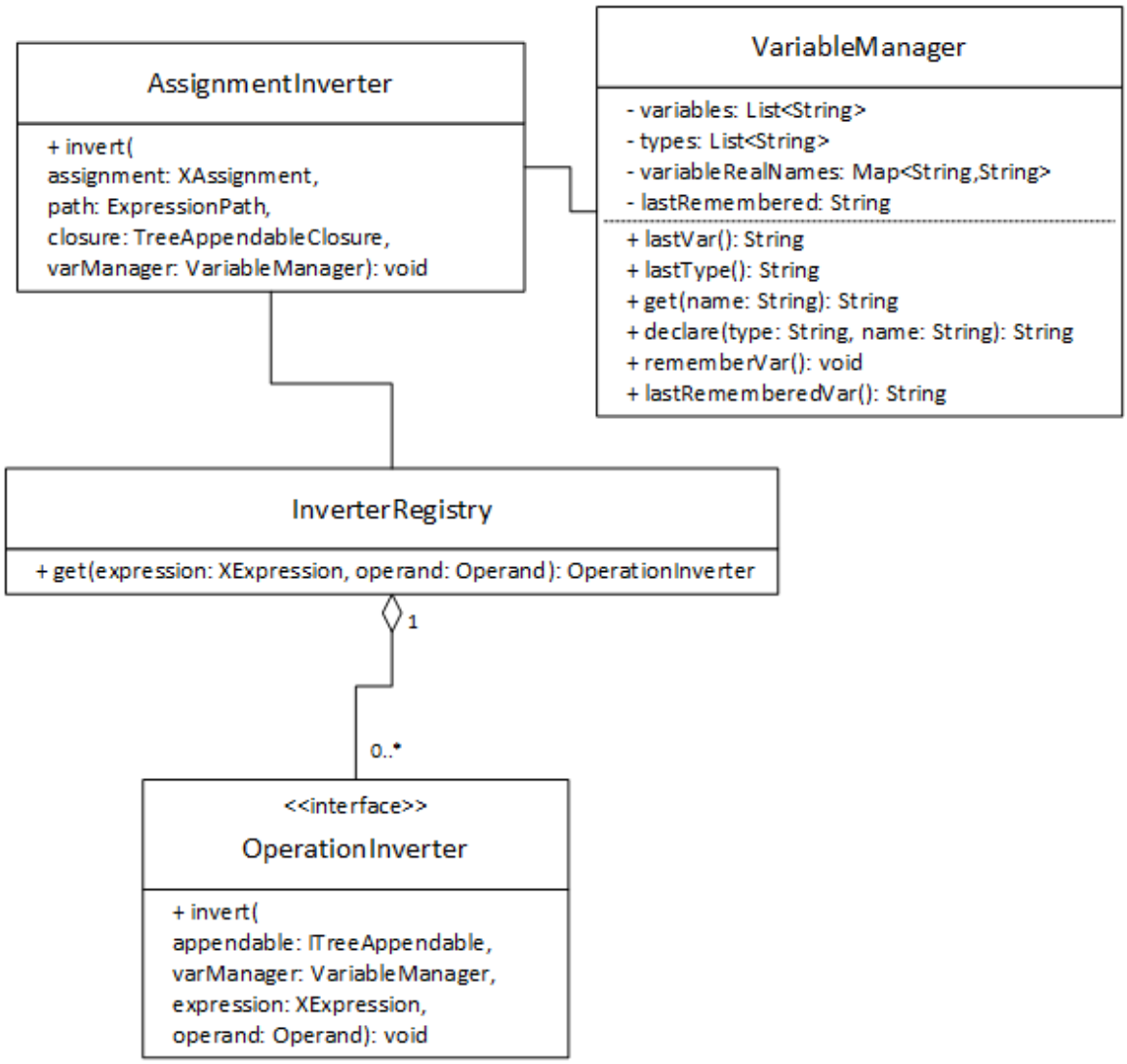

Figure 3.3.: Class Diagram of the Code Generation Implementation

The input and output of every inverted operation is passed through an instance of *VariableManager* which is another argument of the *OperationInverter*. The class is used to declare local variables in the generated code while resolving naming conflicts. The workflow is as follows: The client calls the *VariableManager*'s *declare* method in order to declare a variable with a name and a type. The manager keeps a list of all declared variables and resolves naming conflicts by adding a number to the end of the variable's name. The method then returns the type and the actual name of the variable using Java's variable declaration syntax. The manager also keeps a map of real names using the "declared"

names as the key. This way, as long as a client never declares two variables with the same name, it is able to retrieve the actual name of each temporary variable. In addition, the *VariableManager* keeps track of the types of the variables.

The different inverters have no knowledge of each other. They function by assuming that the last declared variable in the given *VariableManager* holds the result of the last inverted operation. This principle corresponds to the temporary variables we discussed earlier. Consequently, for the first executed inverter, the (only) previously declared variable contains the value of the target attribute. The *VariableManager* offers a method to *remember* the most recently declared variable's name so that it can be retrieved without a key. The inverter uses this feature to access the result of the previous inversion.

If necessary, the inverter generates validation for the input value. If the validation fails, the generated code calls the violation handler as described in 3.1.2 and depending on the return value it either throws an exception or uses the appropriate default value. The result of the inversion is stored in a new variable so that the next inverter can access it. After the final inversion code is generated, the most recently declared variable's value is assigned to the source attribute.

To generate code, a language feature of Xtend called *template expressions* is utilized. Template expressions are an alternative to string concatenation and allow for building strings from a template whereby arbitrary expressions like variables or method calls can be embedded using a special placeholder syntax (similar to JSP). Furthermore, conditional and loop statements make it possible to dynamically alter the template. Finally, template expressions have an intelligent white space and line break handling, meaning they can be easily embedded in the source code of an Xtend program and produce nicely indented output. Figure 3.4 demonstrates the usage of template expressions for code generation.

```
appendable.append('''
    «varManager.declare(targetType, "numerator")»;
    if («varManager.get("oldNumerator")» / «varManager.get("denominator")» == «varManager.lastRememberedVar») {
        «varManager.get("numerator")» = «varManager.get("oldNumerator")»;
    } else {
        «varManager.get("numerator")» = «varManager.lastRememberedVar» * «varManager.get("denominator")»;
    }
''')
```

```
int numerator;
if (oldNumerator / denominator == view) {
    numerator = oldNumerator;
} else {
    numerator = view * denominator;
}
```

Figure 3.4.: Xtend Template Expression and the Produced Output

The process of generating code is realized in the so called *AssignmentInverter*. The class' only method *invert* is responsible for traversing the AST of an expression according to an *ExpressionPath*'s list of arguments and delegating the code generation to the appropriate inverters. The different inverters are stored in a so called *InverterRegistry* where they can be retrieved using the expression of the to-be-inverted operation as well as the operator as key. The expressions are mapped to underlying Java methods using an identifier which consists of the fully qualified method name including the parameter types. Operators

like the binary plus are represented by special extension methods in the Xbase SDK so that they can be identified the same way. The reason why the argument is part of the key is that the inversion of the same operation can be implemented by different inverters, depending on which argument is inverted, as seen in 3.2.

Before each invocation of an inverter, using the *VariableManager*, it is checked whether the type of the result of the previous inversion has the same type as the input to the next inversion. In cases of widening primitive conversion, no explicit cast is necessary for the source to target transformation but is necessary in the opposite direction. In these cases, a call to an appropriate *CastInverter* is inserted as described in 3.2.3.

3.3.3. IDE Features

Our approach is base on the Xtext framework which helps build Eclipse plugins for DSLs and offers APIs for different IDE features that help make editing programs written in the IDE easier. Semantic errors that were caught by the static code analysis, described in 3.3.1, appear in the "problems" window of the IDE and cause the respective parts of the program in the editor to have red underlines. By defining the grammar of the DSL, the editor also automatically offers content assist, i.e. code completion for syntactically correct keywords. By using the Xbase grammar, content assist for correctly typed expressions is available, too. In addition, we implemented content assist for the source attribute that follows the update keyword. Finally, if the update is required but is missing, we offers "quick fixes", a mechanism to automatically insert the update keyword followed by one of the source attributes in the current With expression.

3.4. Evaluation

In this thesis, different operations were defined and inverted that can be used to write bidirectional transformations. For most of the operations that have more than one argument, inversions for all arguments were defined. In addition, some of the transformations are defined for different types of arguments. The different inversions were implemented by 22, mostly parameterizable inverter classes. Counting all possible type combination for all inverted arguments of all supported operations, 370 inversions were implemented in total. These operations can be freely nested to produce complex transformations.

In the following, we evaluate the choice of operations based on the analysis of commonly used transformations that we discussed earlier. Furthermore, we discuss the correctness of our implemented operations. Finally, we give an overview over the limitations of our approach.

3.4.1. Case Study

In 3.1.3 we provided an overview over our analysis of commonly used mapping operations in the ATL transformation zoo. As we discussed earlier, we grouped the operations into 2 groups. Group 1 consisted of operations that are reasonable to invert while group 2 consisted of operation that we ruled out. Table 3.2 shows the number of mappings in the

different categories. One can see that our supported operations make up a substantial portion of commonly used attribute mappings.

Category	Number of mappings
Identity	2238
Arithmetic	119
String Operations	356
ToString	367
Parsing	74
Sequence	1478
List	1062

Table 3.2.: Number of mappings in the ATL transformation zoo by category

As discussed in 3.1.3, our implementation supports operations from the categories "Arithmetic", "String Operations", "ToString" and "Parsing". The "Identity" category is implicitly supported because it doesn't involve any operations. Operations from the categories "Sequence" and "List" were not implemented. For the supported categories, all encountered operations were implemented. Additionally, some operations were implemented that didn't occur in the ATL transformation zoo analysis. Table 3.3 shows which implemented operations were found in the analysis and which were our own contribution.

Category	Found in analysis	Not found
Arithmetic	+, -, *, /	Exponentiation
	sin, cos	tan, asin, acos, atan
	floor, ceil	round, abs
		floorMod
String	String concatenation	
	Substring With with end index	Substring without end index
	String Length	
	toLowerCase	toUpperCase
ToString	Numbers and boolean	
Parsing	Numbers and boolean	

Table 3.3.: Implemented operations by category

In addition to the discussed categories, we also implemented the casting operation which didn't fit in any category of the ATL transformation zoo analysis but turned out to be important to support nesting operations of primitive type where implicit casts were involved.

3.4.2. Evaluation of Correctness

The correctness of our operations is based on the two laws GetPut and PutGet that were defined by Foster et al. [9] and whose application to our operations we discussed in 3.1.2.

We aimed to fulfill these laws for all permitted inputs and implemented a mechanism for handling target values that cannot be inverted while fulfilling PutGet. When defining the supported operations, we discussed how the inversions are designed to fulfill the correctness properties. However, because this is not the scope of this thesis, we give no formal proof of the correctness. Instead we tested the behavior of our inverters using unit tests.

In our unit tests, we verify that the generated code from our inverters fulfills the GetPut and PutGet correctness properties, but also performs as intended for inputs that are not produced by a round trip and generates InversionErrors and PutGetViolations when expected. For this purpose we wrote an MIR program that contains 42 representative With blocks which cover all of the supported operations. We then wrote test cases for all of our 22 inverter classes, which are mostly parametrized and which amount to 242 test methods. These methods test the generated code with different inputs that also cover the different edge cases and all of them are passing.

3.4.3. Limitations

In this section we discuss the limitations that our approach brings with it.

First, as we discussed in 3.1.1, our approach is similar wo solving an equation by applying the inverse operation so that at the end the to-be-updated source attribute is left on one side of the equation. This approach only works if the to-be-updated attribute only appears once on the right-hand side of the equation. This property is called affine [17] (or linear [24]). We use static code analysis as described in 3.3.1 to enforce this property.

Next, our approach is limited by the fact that the supported operations need to be implemented manually. Although this allows combining supported operations to build complex mappings, it limits the possible applications to those cases that can be expressed using the supported operations. Furthermore, the fact that it is a manual process makes extending the system by adding new operations slower.

Finally, our approach doesn't support updating more than one source attribute at a time. This prevents us from supporting operations like the logical "and" and "or" or accepting more target values that currently cause a PutGetViolation.

4. Related Work

In this chapter, existing approaches to bidirectional transformations and model consistency and their relation to our approach are discussed.

4.1. Automatic Bidirectionalization of Functional Programs

In 2007, Matsuda et al. proposed an algorithm that automatically derives backward transformations from view functions written in a first-order functional language called VDL [17]. The approach is based on automatically derriving a minimal complement function g from a view function f as discussed in 2.2 and inverting the tupled function (f, g).

The authors classify their approach as a constant complement approach and make use of the same correctness properties as in the work of Bancilhon and Spyratos [4].

The authors state that view functions written in VDL have two limitations. Firstly, the functions need to be in *treeless* form as defined by Wadler [24], meaning that no intermediate data structures are created during evaluation. Secondly, the view functions need to be *affine*, meaning every variable occurs at most once on the right-hand side of a function definition. If a view function fulfills these two properties, its backward transformation can be derived automatically.

The lambda-calculus like language VDL that view functions must be written in presents a drawback of the approach. The functional language offers pattern matching and recursive function definitions with a Haskell-like syntax, however the type system only knows manually defined type constructors. Linked lists can be defined using a cons construct and numbers can be defined recursively. Because the algorithm works on the source level of a view function definition, library functions cannot be used unless they are also written in VDL.

Other approaches have been made to derive backward transformation from more powerful functional languages. In the approach by Voigtländer from 2008 [23], a function is inspected semantically, meaning only it output and not its definition is relevant. Backward transformation that fulfill desirable round trip laws are then derived using free theorems [25].

4.2. Triple Graph Grammars

One approach for keeping models consistent was proposed by Grunske, Geiger, and Lawley in 2005 [13] and is based on *Triple Graph Grammars* (TGG), a formalism by Schürr [22] to describe bidirectional graph transformations.

A graph grammar (or graph transformation) is a kind of context sensitive grammar for graphs [21]. It is made up of replacement rules which consist of a left-hand side

that is matched to a sub graph of the source graph and a right-hand side that is used to replace the found occurrences. TGGs in turn consist of replacement rules for the meta models of the source and the target model as well as a third correspondence graph that links the two. This way a triple graph grammar rule can be applied in both directions. Additionally, preconditions can be formulated in order to further restrict the application of the transformation rules.

TGGs are similar to the given thesis' approach in that they offer declarative mappings between models from which model transformations are generated. They also allow for incremental application of the transformations as demonstrated by Giese and Wagner in 2009 [11]. However they mostly focus on synchronizing models with different structures while our focus lies the mapping of attributes. The only mapping of attributes possible with TGGs is the identity mapping. Attempts to enable TGG to handle attribute mappings using Constraint Satisfaction Problems (CSP) have been made by Anjorin [2, 1] since 2012.

4.3. Inversion of ATL transformations

The approach of Xiong et al. [27] deals with the extension of the Atlas Transformation Language (ATL) with the goal of synchronizing models. In this case, synchronization means the propagation of changes from both the source and the target models to both of them. The approach is based on ATL which defines a virtual machine (VM) that executes a special bytecode for model transformations. This VM is modified so that models carry so-called *extensions* consisting of functions for deletion, replacement and validity-checking. Both source models and models produced by transformations carry these extensions. Additionally, library methods, e.g. methods for string operations, are modified to return tuples consisting of the return value and an extension. The result is that models produced by the modified transformations carry enough of the lost information to make it possible to reverse them.

According to the authors, the derived transformations exhibit consistency properties similar to the ones proposed by Bancilhon and Spyratos [4] and Foster et al. [9] (as discussed in sections 2.2 and 2.3).

The similarity of the discussed work to the given thesis lies in the automatic inversion of transformation expressions. Furthermore, the usage of ATL allows for more involved modifications of properties including string operations. Similar to our approach, library methods can be used to write complex mappings. As a differentiation, while in our approach, the backward transformation receive an instance of the source meta model to restore lost information, the approach of Xiong et al. is based on putting the lost information directly into the target model.

4.4. Lenses

Besides the initial work of Foster et al. in 2007 [9], the lenses approach has been developed further for other applications. The Boomerang project by Bohannon et al. [6] utilizes lenses to synchronize ordered data like lists using dictionary lenses that marks "reorderable

chunks". This way, when the data is reordered, no inconsistencies are introduces. The project specializes on string data which is a very specific use case but can potentially be incorporated in other approaches like ours.

Additionally, extensions of the lenses approach have been proposed which include quotient lenses by Foster, Pilkiewicz, and Pierce [8] which allow ignoring certain changes to the abstract data like white spaces or formatting or symmetrical lenses by Hofmann, Pierce, and Wagner [14] which redefine the lens operations so that *get* produces a tuple of an abstract structure and a "complement" structure which is similar to the complement of the original view-update definition of the view-update problem by Bancilhon and Spyratos [4]. Other work on lenses includes [20, 5].

5. Future Work and Conclusion

In this chapter we take a look at future work that can be done by building on our results and draw the conclusions of our work.

5.1. Future Work

In the following, future work is discussed that can improve the applicability and usability of our work and help evolve the Vitruvius framework that our approach is integrated in.

First, additional research could be done to analyze which operations are typically used in real world model synchronization scenarios. These operations could be defined and implemented in order to increase the number of supported operations.

A particular category of operations that could be implemented are list operations. These operations like “map” or “filter” differ from the operations that we implemented in this thesis because they are higher-order functions. The imposes additional difficulties when defining the inverted operations because on the target side, there are now a number of values as opposed to one single value and it needs to be sorted out, how deletions, reordering and insertions could be handled. On the technical side, Xbase is well suited for writing these kinds of expressions because it supports lambda expressions and offers extension functions for list types. However the code generation approach would need to be adopted to this task. For example, in order to statically verify that a With expression can be inverted, the functional arguments of higher order functional operations would need to be verified themselves. This verification would need to be recursive because in case of lists of lists, the functional arguments themselves could contain list operations. In contrast, currently, the “invertibility” of a With expressions only depends on the operations in the path of the AST that leads to the to-be-updated source attribute, not the arguments of the operations.

The currently supported operations are chosen because they are general purpose and can be used in a broad range of scenarios. In addition, other domain specific operations could be supported for usage of the Vitruvius framework in specific scenarios. The domain knowledge could help design operations that are hard to design for general purpose like the null check operation. If the source attribute was previously null, i.e. the null check resulted in the boolean value “true”, and then the target value was set to “false”, a new instance of the particular type would need to be instantiated. Specific domain knowledge could help answer the question, how to instantiate the type, e.g. by using dependency injection or a factory.

Furthermore, in order to support an even larger number of use cases, a mechanism could be developed that allows the methodologist to define the inversion of unsupported

operations at design time. Whether the inverted operations fulfill our correctness properties could either be trusted the knowledge of the methodologist or could be checked using automated tests or code analysis.

Next, there is a class of operations with more than one argument that can't be inverted by updating only one argument. Because of this restriction which we discussed in 3.1.3, these operations are not compatible with our approach of only ever updating one source argument of an operation. Among these operations are commonly used ones like the "minimum" and "maximum" operations for numbers, the logical "and" and "or" and the conditional operation "if-then-else". By adapting our approach so that multiple arguments of an operation can be updated, these operations could be supported. Additionally, some supported operations could be improved to accept more target values that currently cause a PutGetViolation. These operations include the "substring" operation with three arguments, where a target value with a different length could be accepted by updating the end index argument as well as the string itself.

Finally, future work could improve our currently supported operations even further for cases that are not covered by the correctness laws GetPut and PutGet. For example, our implementations of the inverted "toUpperCase" and "toLowerCase" operations on strings preserve unmodified parts of the original argument if only characters were added or removed at the start or at the end of the string. We do not preserve any parts of the original argument if simultaneous additions and removals were performed or if any modifications were made in the middle of the string. These cases could be covered using an adapted string distance algorithm like the one proposed by Wagner and Fischer in 1974 [26]. Apart from that, periodic operations like the trigonometric functions or "floorMod" could be improved as well, so that a changed target value is always mapped to the interval, the original source value was in. As of now, a changed target value is always mapped to a default interval, namely the one closest to zero.

5.2. Conclusion

In this thesis we have presented an approach for writing bidirectional model transformations using declarative mappings to synchronize an attribute of one target meta class with a number of attributes of another corresponding source meta class. These mappings are written in Xbase and have the syntax of an assignment. They can contain different supported operators or methods from Java's standard library which can have multiple attributes of the source meta class as arguments and can be nested. Our system then generates Java code that can synchronize the meta classes in both directions. For the source-to-target direction, the operations are simply applied to the source attributes and the result is assigned to the target attribute. For the opposite direction, instances of both, the source meta class and the target meta class are taken as input and one, explicitly selected attribute of the source meta class is updated.

Our approach works by inverting the individual operations that make up the mapping expression. An inverted operation takes all of the arguments of the original operation

as well as its (possibly changed) result as input and updates one of the arguments. This means, for operations with multiple arguments, there are multiple inverted operations. These inverted operations aim to fulfill correctness properties that mandate that a round trip in either direction will not change the original input. This is always guaranteed when applying the source-to-target transformation and then the target-to-source transformation. When applying the transformations the other way round, this is not always guaranteed. For this reason we identify target values that violate the round trip rule and either reject or permit them depending on a provided callback. In the latter case, a round trip will cause the target value to be overridden, however as much information as possible is preserved which is why in our opinion permitting the violation of the round trip rule can lead to a better user experience.

In this thesis we have defined and implemented the inversions for a number of supported operations that were assumed to be useful in a general purpose model consistency scenario and some of which turned out to require rather complex inversions. The selection was validated by an analysis of commonly used mapping expressions in the model transformation community which showed that our selected operations make up a substantial portion of the analyzed mappings. For formulating the correctness of our inversions, we have adopted round trip rules, that were previously used in the bidirectional transformation community, and have evaluated our implementation's compliance with these rules using unit tests. In addition, we have proposed a mechanism to allow values that break these rules because we believe that this can lead to a better user experience. Furthermore, we showed that by building on a DSL framework like Xtext, a highly expressive mapping syntax can be achieved using the Xbase grammar which can be accompanied by a high usability of the design environment through the IDE features like static code analysis, content assist and quick fixes. This implementation is based on an extensible architecture which we hope will help the Vitruvius framework make an impact in the model consistency community.

Bibliography

[1] Anthony Anjorin. "Synchronization of Models on Different Abstraction Levels using Triple Graph Grammars". PhD thesis. Darmstadt: Technische Universität, 2014. URL: http://tuprints.ulb.tu-darmstadt.de/4399/.

[2] Anthony Anjorin, Gergely Varró, and Andy Schürr. "Complex Attribute Manipulation in TGGs with Constraint-Based Programming Techniques". In: *Bx 2012* 49 (2012), pp. 1–16.

[3] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. "Orthographic software modeling: a practical approach to view-based development". In: *Evaluation of Novel Approaches to Software Engineering*. Springer, 2010, pp. 206–219.

[4] F. Bancilhon and N. Spyratos. "Update semantics of relational views". In: *ACM Transactions on Database Systems* 6.4 (Dec. 1981), pp. 557–575. ISSN: 03625915. DOI: 10.1145/319628.319634. URL: http://dl.acm.org/citation.cfm?id=319628.319634.

[5] Davi M J Barbosa et al. "Matching lenses: alignment and view update". In: *Icfp* (2010), pp. 193–204. ISSN: 15232867. DOI: 10.1145/1863543.1863572. URL: http://repository.upenn.edu/cis{_}reports/915.

[6] Aaron Bohannon et al. "Boomerang: resourceful lenses for string data". In: *ACM SIGPLAN Notices*. Vol. 43. 1. ACM. 2008, pp. 407–419.

[7] Sven 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. Dresden, Germany: ACM, 2012, pp. 112–121. ISBN: 978-1-4503-1129-8. DOI: 10.1145/2371401.2371419. URL: http://doi.acm.org/10.1145/2371401.2371419.

[8] J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. "Quotient lenses". In: *ACM SIGPLAN Notices* 43.9 (2008), p. 383. ISSN: 03621340. DOI: 10.1145/1411203.1411257. URL: http://dl.acm.org/citation.cfm?id=1411203.1411257.

[9] J. Nathan Foster et al. "Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem". In: *ACM Transactions on Programming Languages and Systems* 29.3 (May 2007), 17–es. ISSN: 01640925. DOI: 10.1145/1232420.1232424. URL: http://dl.acm.org/citation.cfm?id=1232420.1232424.

[10] The Eclipse Foundation. *Xtend - Documentation*. (accessed on 09/01/2015). 2015. URL: http://www.eclipse.org/xtend/documentation/index.html.

[11] Holger Giese and Robert Wagner. "From model transformation to incremental bidirectional model synchronization". In: *Software and Systems Modeling* 8.1 (2009), pp. 21–43. ISSN: 16191366. DOI: 10.1007/s10270-008-0089-9.

[12] Georg Gottlob, Paolo Paolini, and Roberto Zicari. "Properties and update semantics of consistent views". In: *ACM Transactions on Database Systems* 13.4 (Oct. 1988), pp. 486–524. ISSN: 03625915. DOI: 10.1145/49346.50068. URL: http://dl.acm.org/citation.cfm?id=49346.50068.

[13] Lars Grunske, Leif Geiger, and Michael Lawley. "A Graphical Specification of Model Transformations with Triple Graph Grammars". In: *Ecmda-Fa 2005* (2005), pp. 284–298. DOI: 10.1007/11581741_21.

[14] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. "Symmetric lenses". In: *ACM SIGPLAN Notices* 46.1 (Jan. 2011), p. 371. ISSN: 03621340. DOI: 10.1145/1925844.1926428. URL: http://dl.acm.org/citation.cfm?id=1925844.1926428.

[15] Max E. Kramer. "A Generative Approach to Change-Driven Consistency in Multi-View Modeling". In: *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures - QoSA '15* (May 2015), pp. 129–134. DOI: 10.1145/2737182.2737194. URL: http://dl.acm.org/citation.cfm?id=2737182.2737194.

[16] Max E. Kramer et al. "Change-Driven Consistency for Component Code, Architectural Models, and Contracts". In: *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering - CBSE '15*. New York, New York, USA: ACM Press, May 2015, pp. 21–26. ISBN: 9781450334716. DOI: 10.1145/2737166.2737177. URL: http://dl.acm.org/citation.cfm?id=2737166.2737177.

[17] Kazutaka Matsuda et al. "Bidirectionalization transformation based on automatic derivation of view complement functions". In: *ACM SIGPLAN Notices* 42.9 (Oct. 2007), p. 47. ISSN: 03621340. DOI: 10.1145/1291220.1291162. URL: http://dl.acm.org/citation.cfm?id=1291220.1291162.

[18] Tom Mens et al. "A Taxonomy of Model Transformation". In: *Language Engineering for ModelDriven Software Development* 152.04101 (Mar. 2005), pp. 1–10. ISSN: 18624405. DOI: 10.1016/j.entcs.2005.10.021. URL: http://www.sciencedirect.com/science/article/pii/S1571066106001435http://drops.dagstuhl.de/opus/volltexte/2005/11.

[19] Inc. Oracle America. *Java Language Specification - Chapter 5. Conversions and Promotions*. (accessed on 07/27/2015). 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html.

[20] Hugo Pacheco and Alcino Cunha. "Generic point-free lenses". In: (June 2010), pp. 331–352. URL: http://dl.acm.org/citation.cfm?id=1886619.1886640.

[21] Grzegorz Rozenberg and Hartmut Ehrig. *Handbook of graph grammars and computing by graph transformation*. Vol. 1. World scientific Singapore, 1999.

[22] Andy Schürr. "Specification of graph translators with triple graph grammars". English. In: *Graph-Theoretic Concepts in Computer Science*. Ed. by ErnstW. Mayr, Gunther Schmidt, and Gottfried Tinhofer. Vol. 903. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1995, pp. 151–163. ISBN: 978-3-540-59071-2. DOI: 10.1007/3-540-59071-4_45. URL: http://dx.doi.org/10.1007/3-540-59071-4_45.

[23] Janis Voigtländer. "Bidirectionalization for free! (Pearl)". In: *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '09*. Vol. 44. 1. New York, New York, USA: ACM Press, Jan. 2008, p. 165. ISBN: 9781605583792. DOI: 10.1145/1480881.1480904. URL: http://dl.acm.org/citation.cfm?id=1480881.1480904.

[24] Philip Wadler. "Deforestation: Transforming programs to eliminate trees". In: *ESOP'88*. Springer. 1988, pp. 344–358.

[25] Philip Wadler. "Theorems for free!" In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture*. ACM. 1989, pp. 347–359.

[26] Robert A. Wagner and Michael J. Fischer. "The String-to-String Correction Problem". In: *J. ACM* 21.1 (Jan. 1974), pp. 168–173. ISSN: 0004-5411. DOI: 10.1145/321796.321811. URL: http://doi.acm.org/10.1145/321796.321811.

[27] Yingfei 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*. New York, New York, USA: ACM Press, Nov. 2007, p. 164. ISBN: 9781595938824. DOI: 10.1145/1321631.1321657. URL: http://dl.acm.org/citation.cfm?id=1321631.1321657.

A. Appendix

A.1. Analysis of ATL Transformations

The analysis of the ATL transformation zoo was implemented using the Kotlin programming language[1]. Listing A.1 contains the complete source code of the analysis program.

```
1  package edu.kit.ipd.sdq.vitruvius.casestudy
2
3  import java.io.File
4
5  val regexIdentity = "\\w+ *<- *#?\\w+".toRegex()
6  val regexMethodCall = "\\w+ *<- *#?\\w+(\\.\\w+\\(?.*\\)?)*".toRegex()
7  val regexStringLiteral = "\\w+ *<- *'[\\w\\d:# -_,?!$]*'".toRegex()
8  val regexStringConcat = "(' ?\\+)|\\+ ?'".toRegex()
9  val regexPlusMinus = "(\\d ?\\+)|\\+ ?\\d".toRegex()
10
11 val arithmetic = listOf("*", "/", "sin(", "cos(", "floor(", "ceil(",
12       "mod(", "abs(", "roundValue(")
13 val conditional = listOf("if", "then")
14 val listOperations = listOf("select", "any", "first")
15 val parseOperations = listOf("toInteger", "toReal", "toBoolean")
16 val caseOperations = listOf("toLower", "toUpper")
17
18 val checks = linkedMapOf<String, (String) -> Boolean>(
19       "identity" to { line -> regexIdentity.matches(line) },
20       "arithmetic" to { line ->
21          (arithmetic.any { line.contains(it) }
22                || regexPlusMinus.containsMatchIn(line))
23                && !regexStringLiteral.matches(line)
24                && !regexStringConcat.containsMatchIn(line)
25       },
26       "stringLiteral" to { line -> regexStringLiteral.matches(line) },
27       "parseString" to
28             { line -> parseOperations.any { line.contains(it) } },
29       "toString" to { line -> line.contains("toString()") },
30       "string" to { line ->
31          regexStringConcat.containsMatchIn(line)
```

[1] https://kotlinlang.org/

```
32                || line.contains("concat")
33                || line.contains("substring")
34                || line.contains(".size()")
35                || caseOperations.any { line.contains(it) }
36       },
37       "sequence" to { line -> line.contains("Sequence") },
38       "list" to { line -> listOperations.any { line.contains(it) } },
39       "ocl" to { line -> line.contains("ocl") },
40       "conditional" to { line -> conditional.all { line.contains(it) } },
41       "methodCall" to { line ->
42          regexMethodCall.matches(line)
43                && !(arithmetic + parseOperations
44                + caseOperations + listOf("toString"))
45                .any { line.contains(it) }
46       }
47 ).run { this + ("other" to { line -> this.values.none { it(line) } }) }
48
49 val group1Categories = listOf("arithmetic", "string", "toString",
50       "parseString", "identity", "sequence", "list")
51
52 fun main(vararg args: String) {
53    val resultsDir = File("results/atl")
54    resultsDir.mkdirs()
55
56    val linesToFiles = getLines(listFiles("atl", "atl"))
57          .fold(arrayListOf<Pair<String, File>>()) { list, s ->
58             val last = list.lastOrNull()
59
60             val lastLine = last?.first
61             if (lastLine != null
62                   && !lastLine.endsWith(",")
63                   && (!lastLine.endsWith(")")
64                   || (lastLine.count { it == '(' }
65                   == lastLine.count { it == ')' }))
66             ) {
67                list += (list.removeAt(list.lastIndex).first
68                      + " " + s.first to s.second)
69             } else if (s.first.contains("<-")) {
70                list += s
71             }
72
73             list
74          }
75          .asSequence()
76          .map {
```

```
77            it.first.trim(',').let {
78               if (it.endsWith(")")) {
79                  if (it.endsWith(" )")) {
80                     it.dropLast(2)
81                  } else if (it.count { it == '(' }
82                        != it.count { it == ')' }) {
83                     it.dropLast(1)
84                  } else {
85                     it
86                  }
87               } else {
88                  it
89               }
90            } to it.second
91         }
92
93   println()
94
95   val (group1, group2) = checks
96         .map {
97            it.key to linesToFiles.filter { lf ->
98               it.value(lf.first)
99            }
100        }
101        .partition { group1Categories.contains(it.first) }
102
103  val printFunc: (Pair<String, Sequence<Pair<String, File>>>) -> Unit
104        = {
105     println("${it.first}: ${it.second.count()}")
106
107     File(resultsDir, "${it.first}.txt").apply {
108        val counts = it.second.asSequence()
109              .map { it.second }
110              .groupBy { it }
111              .map {
112                 (it.key.relativeTo(File("download/atl"))
113                       to it.value.size)
114              }
115              .sortedByDescending { it.second }
116              .toMap()
117
118        writeText(counts.toString().replace(',', '\n'))
119        appendText("\n\n")
120
121        val lines = it.second.map { it.first }
```

```
122             appendText(lines.joinToString(separator = "\n"))
123         }
124     }
125
126     println("Group 1:")
127     group1.forEach(printFunc)
128
129     println("\nGroup 2:")
130     group2.forEach(printFunc)
131 }
```

Listing A.1: Source code for the analysis of the ATL transformation zoo