# Comparison of QVT-O and Henshin-TGG for Synchronization of Concrete Syntax Models

Stephan Seifermann
FZI Research Center for Information Technology
seifermann@fzi.de

Jörg Henß
FZI Research Center for Information Technology
henss@fzi.de

## Abstract

Concrete syntax models are synthetic views on information structured according to a meta model and allow tailoring information representation to various needs. Synchronization approaches must keep overlapping information consistent and must retain unmapped information such as positioning of graphical elements. Defining a bidirectional and incremental transformation between these models is one possible solution. Selecting a transformation language and tooling is, however, difficult, because there is no universal solution. Therefore, we compare two opposing approaches: complementing QVT-O with coupled unidirectional transformations including trace transformations and Henshin-TGG with additional support for reusable nodes. We achieved bidirectional, incremental, and fully automated transformations between textual and graphical concrete syntax models of UML diagrams with both approaches. We compare them for our use case, as well as discuss and rate restrictions. Our findings regarding the effort in writing and maintaining transformations indicate a need for further research and tool support.

## 1 Introduction

Concrete syntaxes serve as interface to the modeling language. Multiple concrete syntaxes for an abstract syntax enable using the meta model in multiple contexts easily. For instance, the Unified Modeling Language (UML) [Obj15] defines a graphical concrete syntax that leverages visual memory to speed up modeling. This syntax is, however, not accessible to visually impaired modelers. In contrast, a textual concrete syntax allows using the UML by those modelers.

In a research project, we face the issue of keeping multiple synthetic views on UML diagrams using different concrete syntaxes consistent. For instance, a graphical and a textual syntax model for UML share the information about classes and associations. Additionally, the graphical model contains the information about the positioning of shapes and edges representing those elements. If overlapping information of these syntax models changes, bidirectional and incremental transformations can reestablish the consistency. According to the classification of incrementality of Czarnecki and Helsen [CH06], we focus on *target-incrementality* and *preservation of user edits in the target*, i.e., existing models are updated and non-mapped modifications are preserved.



In: R. Eramo, M. Johnson (eds.): Proceedings of the Sixth International Workshop on Bidirectional Transformations (Bx 2017), Uppsala, Sweden, April 29, 2017, published at http://ceur-ws.org

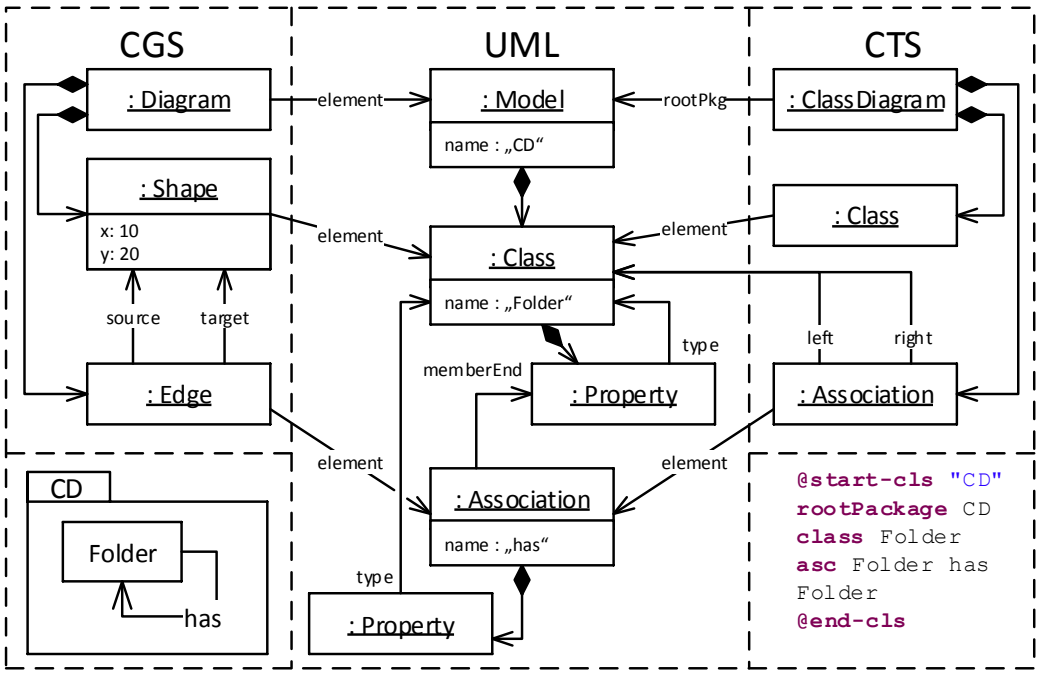



Figure 1: Simplified excerpt of graphical syntax (left), textual (right) syntax, and UML (middle) model.

We could not find a universal solution for incremental and bidirectional transformations but had to select an approach based on our usage scenario described in Section 2. In this paper, we report on the applicability of QVT-O and Henshin-TGG for synchronizing the concrete syntax models in that usage scenario. Therefore, we present guidelines and restrictions for applying the approaches, as well as report on the effort for creating and maintaining transformations. Our contribution is the comparison of the approaches' applicability to our usage scenario rather than defining generalized extensions for the approaches.

The remainder of this paper is structured as follows: Our usage scenario and the rationale behind our tool selection is given in Section 2. Section 3 covers the realization of our scenario with these approaches. The experiences in writing and maintaining the transformations are described and compared in Section 4. We report on related work in Section 5 and draw conclusions in Section 6.

## 2 Usage Scenario and Tool Selection

Our scenario [SH] covers the bidirectional and incremental transformation between two concrete syntax models for UML class diagrams. Figure 1 illustrates a simplified excerpt from the involved models and their relationships: a concrete graphical syntax (CGS) model on the left and a concrete textual syntax (CTS) model on the right describe the same UML class diagram. They refer to UML model elements of a third model in the middle. Regarding the model structure, the UML model is no abstract syntax but an existing model that is referred to as shown in Figure 1. The graphical model conforms to GMF Notation [Ecl17]. The textual model conforms to a meta model developed in the Cooperate project [Coo17]. The third UML model conforms to Eclipse UML.

We implemented the synchronization as a feature of the Cooperate modeling environment, which implied the following constraints: a) There is no information about edit operations except for model states. This implies a state-based instead of a delta-based synchronization. b) Editors enforce consistency between the UML model and the changed syntax model. For instance, the Papyrus UML editor [Gér+07] adjustes the CGS and the UML model automatically: adding a class in the editor means adding a class in the UML model and a shape in the graphical model. The same holds true for the textual editor. Therefore, model transformations must not alter a referred UML model that has already been changed by the editor. Instead, they have to keep the other concrete syntax model consistent with the changed model. c) There is no single model that holds all information about the concrete syntax representations. Bijective mappings are not possible. Therefore, transformations must either work in-place or merge the result model into the previous model state. Otherwise, unmapped content such as coordinates of graphical elements cannot be retained. d) Cross references from other models to the syntax models exist. The transformation has to maintain element identities in order to maintain existing cross references.

Besides these conceptual restrictions, we formulate restrictions based on the execution environment of the Cooperate modeling environment and organizational aspects: a) regular updates, b) support of Windows, Linux, and OS X, c) open source, d) executable as Eclipse plug-in, e) support of EMF, f) support of exogeneous transformations, and g) n-to-n cardinality. The extensive survey of Kahani and Cordy about model transformation languages and according tooling [KC15] presents a wide range of approaches. We relied on the information given in the survey and did not try to complement it, which most probably lead to the elimination of several approaches. Considering them will be future work. Filtering the approaches based on our restrictions revealed QVT-O [Obj16], Henshin-TGG [Ehr+15, chap. 12.4], and Melange [Deg+15]. Melange, however, utilizes a general purpose programming language which implies less concise transformations. We chose Henshin-TGG over plain Henshin because a comparison of TGG tools [Leb+14] pointed to this incremental approach. Additionally, it supports bidirectional transformations out-of-the-box. We selected QVT-O and Henshin-TGG for our comparison.

Query/View/Transformation (QVT) [Obj16] is a set of model transformation languages and semantics by the OMG. QVT-O (operational mappings) describes model transformations imperatively by unidirectional mapping operations. Operation signatures consist of context, parameters, and results. The body basically consists of OCL expressions, assignments, and mapping calls. Subsection 3.1 covers experimental incremental updates [Boy16].

Henshin-TGG [Ehr+15, chap. 12.4] builds upon the Eclipse-based Henshin graph transformation tooling [Are+10] and provides tool support for Triple Graph Grammars (TGGs) [Sch94]. TGGs describe transformations as set of graph pattern rules. Each rule describes a correspondence by mapping a left-hand graph pattern to a right-hand graph pattern. A correspondence graph connects both patterns to keep track of transformed elements. Giese and Wagner [GW06] showed that TGGs allow incremental transformations by combining consistency, deletion, and unidirectional transformation steps. Therefore, the TGG approach supports bidirectional and incremental transformations. Henshin-TGG provides an editor for TGG correspondence rules and an execution engine for EMF-based models. The execution engine performs forward and backward transformations, consistency checks, and integration of models based on rules derived from correspondence rules. Henshin-TGG implements incremental transformations by combining consistency checks, marking of inconsistent source graph elements, executing the transformation for marked elements, and deleting remaining inconsistent elements in the target graph.

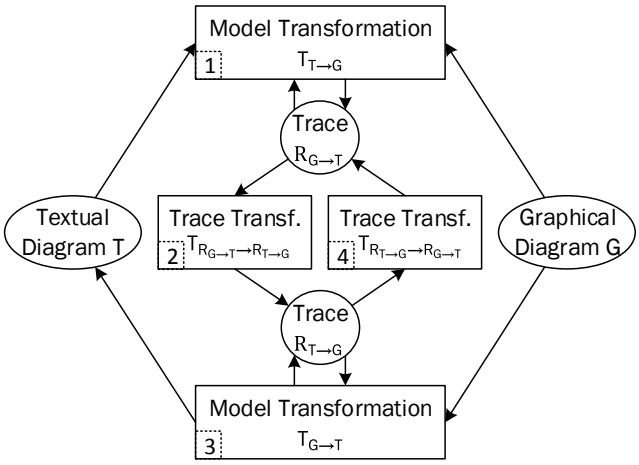
## 3 Case Study: Realizing Bidirectional and Incremental Transformations

In our case study, we synchronize two models representing a UML diagram with bidirectional and incremental transformations in a state-based way. We achieve this with complemented variants of QVT-O and Henshin-TGG, i.e. we leveraged existing mechanisms to work around restrictions rather than intrusively extending the approaches. We first report on the realization with QVT-O in Subsection 3.1 and afterwards show how to achieve consistent models with Henshin-TGG in Subsection 3.2.

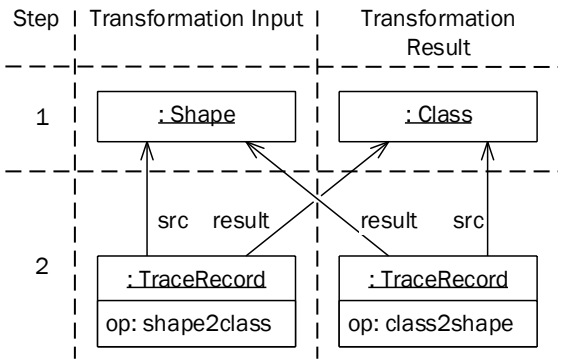
### 3.1 Realization using QVT-O

QVT-O is an imperative model transformation language that works unidirectionally and provides an experimental incremental update mode. In our usage scenario defined in Section 2, we, however, need incremental and bidirectional transformations.

The incremental update mode is only documented informally in the Eclipse forums [Boy16]. The mode leverages previous execution traces. Traces hold all executed mapping operations, used context, parameters, and results. The transformation re-executes every mapping operation on the result object from the trace if the context and parameters match. This maintains the identity of the target elements. Transformation writers have, however, to define their mappings carefully as the informal documentation as well as Willink and Matragkas [WM15] state. The incremental update does not work for every valid QVT-O transformation: The transformation executor of QVT-O cannot always safely detect if it shall execute a mapping operation incrementally based on the information contained in the trace. Willink and Matragkas present trace model extensions to address the issues but these extensions have not been adopted by the OMG yet. Instead, we derived transformation restrictions that make the incremental update mode work as expected: a) Created model elements must not change in a way that influences other mappings. For instance, the name of a class must not change after its creation if another mapping uses this name transitively in assignments or guards. b) There must not be mutable global context. c) Mutable lists must not be mapping inputs or its contents must not change. d) Multiple transformations or mappings must not be executed concurrently on the same model in memory. We discuss the influence of these restrictions in Section 4.




(a) Transformations and artifacts of roundtrip.




(b) Example of input and output for first two steps.

Figure 2: Transformation chain for roundtrip between graphical and textual syntax model using QVT-O.

To achieve bidirectional transformations with QVT-O, defining an unidirectional transformation for each direction is a common approach [Pos+14]. This leads to an increased maintenance effort but creates the expected results. This approach, however, does not work with the incremental update mode: during an incremental update, the trace of a previous transformation execution is fed back to the transformation engine. The traces of two separate model transformations are, however, not compatible because they describe the transformation for a different direction. If we run a transformation $T_{T\to G}$ that transforms a textual model $T$ to a graphical model $G$ with a resulting trace $R_{T\to G}$, we cannot run a transformation for the opposite direction $T_{G\to T}$ with trace $R_{T\to G}$.

Instead of deriving the traces from the model states as suggested by Ehrig et al. [EEH08] for TGGs, we enable incremental transformations for our coupled unidirectional transformations by transforming the traces of the transformations into traces for the opposite direction as shown in Figure 2a. More precisely, we create additional transformations $T_{R_{T\to G}\to R_{G\to T}}$ and $T_{R_{G\to T}\to R_{T\to G}}$. After executing a transformation for the models, we execute the according trace transformation to gain consistency not only between the models but also between the traces. The consistent traces serve as input for the incremental update execution. Figure 2b demonstrates these steps: First, the transformation $T_{T\to G}$ produces the textual from the graphical model, e.g. by transforming a shape into a class element. Second, the trace transformation $T_{R_{T\to G}\to R_{G\to T}}$ transforms the trace record for mapping operation executions from the $T_{T\to G}$ to the $T_{G\to T}$ transformation by swapping input and outputs and changing the name of the executed mapping. For a sake of brevity, we do not show more complex trace transformations here but point to the available source code [Coo17].

Deleting elements requires additional treatment because the incremental update preserves unmapped elements. Elements deleted on one side lead to unmapped elements on the other side. To solve this issue, either a post processing step or a reset assignment has to be applied. The reset assignment operator `:=` can completely re-assign multi-valued references. Unmapped elements disappear. This is only possible if a reference cannot include elements that should be retained and are not created by the transformation.

In order to execute the transformation later, serialization of the trace is possible. In this case, transformation writers have to choose unique mapping operation names to avoid ambiguities in the serialized trace because the trace only serializes the mapping name, package, and module but not the full signature.

### 3.2 Realization using Henshin-TGG

The Henshin-TGG approach supports both bidirectional and incremental transformations out of the box. As stated before, a developer models the TGG patterns and then generates unidirectional rules from that patterns that are orchestrated to realize incrementality.

For our specific usage scenario described in Section 2, we encountered problems using Henshin-TGG when both left hand graph and right hand graph have to contain the *same* UML element for matching. It is not possible to place the *same* element on both sides or match nodes based on their identity rather than equality with respect to the modeled relations and attributes. In our scenario, elements on both sides should refer to the very same UML elements. Therefore, we defined the UML model as a second correspondence model besides the trace model.

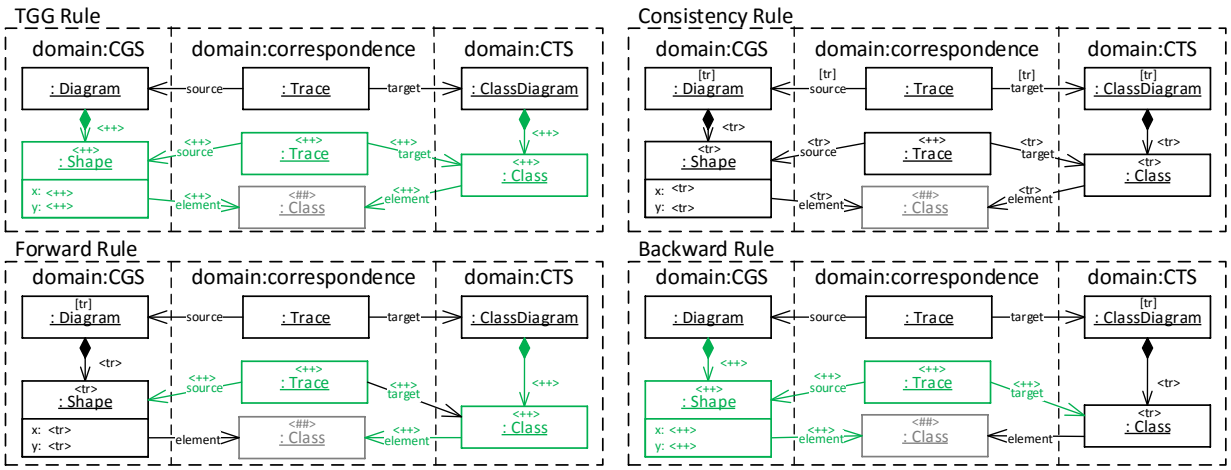



Figure 3: Example of a Henshin-TGG rule with a reusable node ⟨##⟩ and the adapted derived rules.

Because our transformations only change the concrete syntax model elements but leave UML model elements unchanged, these elements correspond to the TGG advanced concept of so-called *reusable nodes* [GK10]. This concept makes it possible to create an outgoing reference to an existing element in the correspondence domain without creating the element. Henshin-TGG does not include this advanced concept, therefore we had to adopt both the TGG editor as well as the unidirectional rule generation.

Figure 3 shows the TGG rule for our running example, the derived forward $T_{G \to T}$ and backward transformation rules $T_{T \to G}$, as well as the consistency rule $C_{G \leftrightarrow T}$. The shown TGG represents the mapping of a class in the CTS domain to a shape in the CGS domain. The UML class element in the correspondence domain of the TGG is a reusable node and thus marked with a ⟨##⟩ annotation. A reference to this element has to be present on both mapped elements. Moreover, the figure shows new elements with a ⟨++⟩ annotation, while the ⟨*tr*⟩ and [*tr*] are graph traversal marks used by Henshin-TGG in the unidirectional rules.

## 4 Discussion

The approaches described in the previous section support our intended usage scenario. There are, however, limitations of their applicability and the efforts for writing and maintaining the transformations have to be considered. We first report restrictions of the approaches using QVT-O and Henshin-TGG in Subsection 4.1. Afterwards, we determine the effort of writing and maintaining the transformations and compare the approaches in Subsection 4.2. In Subsection 4.3, we discuss threats to validity.

### 4.1 Applicability of Transformation Approaches

The incremental update using QVT-O heavily relies on assumptions that hold for our use case but do not hold for arbitrary use cases. We argue, however, that those assumptions are often not hard to meet: Boyko provides guidelines for writing incremental transformations [Boy16] that help adhering to restrictions and are often already in use: usage of reset-assignments, and avoiding conditional assignments and assignments to result objects in init blocks. Often, transformation designers can follow the restrictions outlined by Willink and Matragkas [WM15] that match our guidelines described in Subsection 3.1: Mapping parameters can replace mutable global context. Reordering mapping rules and leveraging the trace model help avoiding mutable lists and objects. Concurrent transformations can be executed sequentially. As we pointed out, mapping operation names should be unique.

The alternative approach based on Henshin-TGG proved to be reliable as well. For realizing incremental transformations, however, it is necessary to have full model coverage when checking the consistency of graphs. This means developers have to write rules that cover every element, reference, and attribute of both models. Otherwise, the engine will delete those in the course of an incremental roundtrip transformation. Moreover, we encountered restrictions in matching and setting enum attributes and defining unset references in EMF models.

Both approaches are applicable in our use case: Simple guidelines for adhering to QVT-O restrictions lower the burden to implement incremental transformations in practice. Henshin-TGG provides basic editing functionality that misses support for achieving required full element coverage. While Henshin-TGG has a strong theoretical foundation, mature tool support that needs no complements makes QVT-O better applicable in practice.

### 4.2 Creation and Maintenance Effort

In order to rate the effort for writing transformations with both approaches, we collect the model transformation statements and according lines of code excluding empty and commented lines for our use case. The transformations cover 18 elements of the graphical and 20 elements of the textual diagram representation. Table 1 presents the results for the QVT-O approach. The source code of the transformations is available on Github [Coo17] in the `de.cooperateproject.modeling.transformation.transformations` bundle. The number of mappings of the transformation $T_{T \leftarrow G}$ matches the number of textual elements perfectly. For the opposite direction, this is not true because the same elements are mapped in slightly different ways within separate mappings. The LOC, however, do not grow with the same factor as the number of mappings. This indicates that the mappings become simpler. All in all, the bidirectional transformation requires 82 mappings and 24 queries with a total of about 750 LOC. The trace transformations that enable incremental execution more than double the LOC. This meets our expectations because a trace transformation has to incorporate the mapping logic of both directions, which is complex for transformation designers. Creating trace transformations can be considered a complex and costly task – even for experienced transformation designers. In contrast, writing model transformations is straight-forward with some basic knowledge.

Table 1: Statements, lines of code (LOC), and rule element count (REC) required for bidirectional, incremental synchronization of textual $T$ and graphical $G$ class diagram models.

| | QVT-O Model Transformation | | | QVT-O Trace Transformation | | | | Henshin-TGG | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Direction | Mappings | Queries | LOC | Mappings | Helpers | Queries | LOC | Rules | NAC | REC |
| $T \rightarrow G$ | 62 | 12 | 474 | 18 | 9 | 10 | 293 | — | — | — |
| $T \leftarrow G$ | 20 | 12 | 273 | 46 | 9 | 16 | 550 | — | — | — |
| $T \leftrightarrow G$ | 82 | 24 | 747 | 64 | 18 | 26 | 843 | 35 | 4 | 614 |

For estimating the effort required for creating the Henshin-TGG transformation, we collect the number of rules, negative application conditions (NAC), and the rule element count (REC). The TGG version of the transformation consists of 35 rules of which ten are unidirectional production rules used to create elements that have no correspondence. Four NACs had to be defined for handling ambiguous rules that led to the excessive creation of nodes in the target graph. A modeler defines a NAC by defining an additional triple graph pattern and then maps the nodes in the NAC to nodes in triple graph rule pairwise. This makes modeling NACs complicated and prone to errors. The Henshin interpreter log can help finding recursively called rules that lead to excessive node creation. It is, however, hard to configure for Henshin-TGG. Modelers can, however, create specific trace models to persist additional references in the correspondence graph. This can help to avoid NACs for single side production rules, as a reference to the produced elements can be added to the trace of a parent node. Henshin-TGG, unlike other TGG engines, does not support the definition of abstract rules that can be refined for specific cases. Modelers can, however, duplicate and change the rules manually. In some cases, a workaround can be using an attribute condition to avoid duplicates for similar subclasses of an abstract class. As only graphic editors exist for creating rules, the REC (nodes, references and attributes) can be considered similar to the LOC metric for text based transformation languages. In our scenario the overall REC is 614 elements.

Comparing LOC and REC is not trivial due to their different characteristics. Instead, we compare the number of mappings and rules: The QVT-O transformations require 72 mappings not considering abstract and trace mappings. Henshin-TGG requires 35 rules. Both transformations cover the whole scenario. This indicates that QVT-O requires a more fine-grained structuring than Henshin-TGG. The effort for creating rules can benefit from the more coarse-grained structure possible in Henshin-TGG that avoids duplicating elements. Beginners, however, struggle finding the correct granularity. Moreover, Henshin-TGG does not allow resolving ambiguous rules by specifying the order of rule application. QVT-O simplifies this choice by making it explicit. The additional trace transformation in QVT-O, however, doubles the effective effort. Both approaches do not allow fast rule creation in our scenario. The increased effort for Henshin-TGG is accidental because better tooling can heavily reduce the effort. In contrast, the additional effort for QVT-O trace transformations is essential in our scenario and hard to avoid, therefore. This means that a) the paradigm of specifying one bidirectional transformation as used by TGGs implies less effort in creating model transformations than using coupled unidirectional transformations and that b) tool support is a major factor in considering efforts for transformation creation.

To rate the maintenance effort for our transformations, we determine the changes required by a real evolutionary change in the textual meta model. A simplified version is illustrated in Figure 4: typed elements contain a type

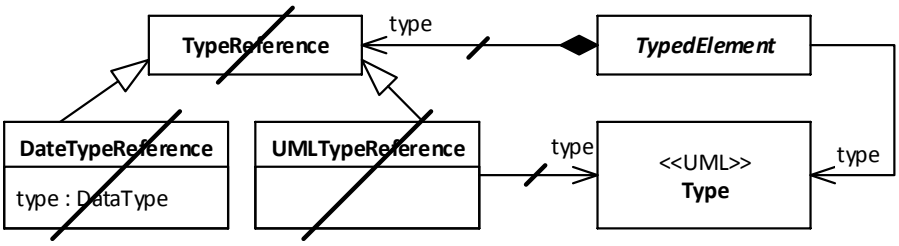



Figure 4: Meta model change for evolution scenario: crossed elements have been replaced by most right association.

reference that can be either a data type reference or a reference to a UML type. The change removes the type reference class and all of its subclasses. Instead, the typed elements now refer to a UML type directly. This typical maintenance task lead to three removed meta model classes and five changed references. For the QVT-O transformations, we collected the according changes from our code repository [Coo17, commit 93489a2]. Table 2 summarizes the required changes in the transformations. Transformation changes and trace transformation changes are related because trace transformations replay the effects of the according transformations. Precisely, maintaining the transformation $T_{T \to G}$ requires about the same effort as maintaining the trace transformation in the opposite direction. Even if the ratio between model and trace transformation changes depends on many factors including the chosen transformation structure, there is a trend for considerably increased maintenance efforts caused by the trace transformations.

Table 2: Changes required for adaptation to a meta model change. ± means changed.

| | Δ Mappings | | Δ Statements | |
|---|---|---|---|---|
| Artifact | Regular | Trace | Regular | Trace |
| $T \to G$ | -4 | 0 | ±12 | 0 |
| $T \leftarrow G$ | 0 | -4 | ±8 | -11 |

The above described meta model change also impacts the TGG implementation of the transformation. To estimate the effort for adapting to the new meta model, we counted the number of affected rules (five) and the change in rule element count (-19). For finding affected rules, however, modelers have to inspect all rules of the TGG manually because the tooling lacks syntax checking that indicates deleted elements and search capabilities that allow locating elements quickly. This introduces considerable additional effort in maintenance compared to QVT-O, especially for complex model transformations. Maintaining transformations with Henshin-TGG would benefit from the same tool support as QVT-O. A textual notation as available for eMoflon [Leb+14] could provide a quick overview and ease maintenance as well.

In the context of our research project, we favored QVT-O over Henshin-TGG. The only reason for this decision is the mature tool support and active maintenance. We see benefits of using TGGs with respect to a strong theoretical foundation and automatic generation of unidirectional transformations from a declarative specification. The benefits, however, vanish in the light of weak tool support, which impedes applicability in practice.

### 4.3 Threats to Validity

The most important threat to validity is the small sample size of our study that prohibits statistically significant results. This holds true for recorded efforts, statements about applicability of the approaches, and the presentation of possible reasons for observed issues. Therefore, the presented results have to be seen as indicators that need further case studies for verification.

Our scenario is, however, realistic because it originates from an existing project. Additionally, an industrial project about synchronizing graphical and textual representations of a model [Mar+15] reported that incremental, bidirectional transformations can be a possible solution to similar problems. The change we used to determine the maintenance effort is not synthetic but is part of recent development activity. We selected this change because of its corrective nature that implies targeted modifications to reestablish correctness in a fast and efficient way. Therefore, we assume that the transformation adjustments did not include any unnecessary changes.

The transformation writers and the authors are the same persons. Therefore, some sort of bias cannot be neglected. Nevertheless, we tried to be as objective as possible: We estimated the efforts based on the numbers we derived from the created transformations and actual modifications.

## 5 Related Work

We see two areas of work related to our approaches: a) model synchronization using QVT-O and Henshin-TGG, and b) mapping of multiple models with different concrete syntaxes.

Cicchetti et al. [CCL11] tackle the view-update problem with delta-based model synchronization. Model comparison produces deltas that conform to generated delta meta models. Generated QVT-O transformations apply the deltas in an in-place transformation. The QVT-O transformations themselves are neither bidirectional nor incremental. Aranega et al. [AED11] do not perform model synchronization but discuss limitations in using QVT-O and its trace for this purpose. They use a customized trace model that records more information than the default trace model to address issues like reported by Willink and Matragkas [WM15], which we described above. In our approach, we adjusted our transformations to circumvent these issues.

Using QVT-O or TGGs is no novel approach in the field of concrete syntax mapping. Elaasar and Labiche [EL11] used QVT-O mappings to map a UML diagram interchange model to a UML diagram graphics model. The former describes the layout information implied by the abstract syntax. The latter describes the layout information detailed for a concrete syntax. Even if unidirectional mappings degrade comprehensibility, the authors argue that ambiguities of the conceptual mappings are the core issue. Andrés et al. [ALG07] use TGGs to transform parts of a graphical model into a textual view without focusing bidirectionality. They use TGGs to choose the trace model freely. Others [WM15; AED11] state this QVT weakness as well.

There is only little related work on our specific use case of synchronizing textual and graphical concrete syntax models for UML. Maro et al. [Mar+15] synchronize graphical and textual views on excerpts of UML using the Atlas Transformation Language (ATL) but do not maintain layout information. They, however, suggest using incremental transformations to solve this issue. In a survey about textual modeling languages and tooling for UML [SG16], we found three approaches that support the roundtrip between textual and graphical representations. We estimate their effort for creating the transformations considerable because, to the best of our knowledge, no dedicated model transformation language is used.

## 6 Conclusions

In this paper, we reported our experiences using QVT-O and Henshin-TGG for synchronizing models representing UML diagrams by a graphical and a textual concrete syntax. We described how to achieve incremental and bidirectional transformations for our use case. Precisely, we complemented QVT-O with trace transformations to achieve incrementality and coupled two unidirectional transformations to achieve bidirectionality. We complemented Henshin-TGG with support for reusable nodes. We discussed restrictions of the approaches and how to adhere to them. The effort for creating and maintaining transformations using QVT-O can be high because of trace transformation overhead. We found that Henshin-TGG divides the effort for creating transformations in half but requires considerable effort for maintaining transformations after changes because of weak tool support.

Practitioners and researchers benefit from our three main contributions, i.e., summarized restrictions, reported experiences, and the case studies: The case studies provide practitioners with hints on how to extend existing tooling and which restrictions apply. Researchers can empirically evaluate if the approaches including the complements used in the case studies become more and easier applicable. The summarized restrictions and our experiences help practitioners to select an approach and give implementation guidelines. Researchers can create new or extended approaches to tackle discovered challenges.

In future work, we want to evaluate if generating QVT-O traces based on model states lowers the transformation creation and maintenance effort compared to the trace transformation. Additionally, we plan to apply the approaches to more UML diagram types in order to validate that the restrictions hold in multiple, realistic use cases. With respect to Henshin-TGG, we aim to increase usability by improving the editing capabilities.

## Acknowledgements

This work is funded by the German Federal Ministry of Labour and Social Affairs under grant 01KM141108.

## References

[AED11] Vincent Aranega, Anne Etien, and Jean-Luc Dekeyser. “Using an Alternative Trace for QVT”. In: *ECEASST* 42 (2011).

[ALG07] Francisco Pérez Andrés, Juan de Lara, and Esther Guerra. "Domain Specific Languages with Graphical and Textual Views". In: *Proceedings of AGTIVE*. 2007, pp. 82–97.

[Are+10] Thorsten Arendt et al. "Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations". In: *Proceedings of MODELS*. 2010, pp. 121–135.

[Boy16] Sergey Boyko. *Eclipse Community Forums – QVT-OML Incremental Transformation*. `https://www.eclipse.org/forums/index.php?t=msg&th=1078533#msg_1735591`. 2016.

[CCL11] Antonio Cicchetti, Federico Ciccozzi, and Thomas Leveque. "Supporting Incremental Synchronization in Hybrid Multi-view Modelling". In: *Proceedings of MODELS*. 2011, pp. 89–103.

[CH06] Krzysztof Czarnecki and Simon Helsen. "Feature-based survey of model transformation approaches". In: *IBM Systems Journal* 45.3 (2006), pp. 621–646.

[Coo17] Cooperate Project Team. *Cooperate Modeling Environment - Github*. `https://github.com/Cooperate-Project/CooperateModelingEnvironment`. accessed 16.01.17. 2017.

[Deg+15] Thomas Degueule et al. "Melange: A Meta-language for Modular and Reusable Development of DSLs". In: *Proceedings of SLE*. 2015, pp. 25–36.

[Ecl17] Eclipse Foundation. *Eclipse Graphical Modeling Framework (GMF) Notation*. `https://www.eclipse.org/gmf-notation/`. accessed 16.01.17. 2017.

[EEH08] Hartmut Ehrig, Karsten Ehrig, and Frank Hermann. "From Model Transformation to Model Integration based on the Algebraic Approach to Triple Graph Grammars". In: *ECEASST* 10 (2008).

[Ehr+15] Hartmut Ehrig et al. "Tool Support". In: *Graph and Model Transformation: General Framework and Applications*. 2015, pp. 351–399.

[EL11] Maged Elaasar and Yvan Labiche. "Diagram Definition: A Case Study with the UML Class Diagram". In: *Proceedings of MODELS*. 2011, pp. 364–378.

[Gér+07] Sébastien Gérard et al. "Papyrus: A UML2 Tool for Domain-Specific Language Modeling". In: *Model-Based Engineering of Embedded Real-Time Systems - Dagstuhl Workshop*. 2007, pp. 361–368.

[GK10] Joel Greenyer and Ekkart Kindler. "Comparing relational model transformation technologies: implementing query/view/transformation with triple graph grammars". In: *SoSyM* 9.1 (2010), p. 21.

[GW06] Holger Giese and Robert Wagner. "Incremental Model Synchronization with Triple Graph Grammars". In: *Proceedings of MODELS*. 2006, pp. 543–557.

[KC15] Nafiseh Kahani and James R. Cordy. *Comparison and Evaluation of Model Transformation Tools*. Tech. rep. 2015-627. School of Computing, Queens University Kingston, Ontario, 2015.

[Leb+14] Erhan Leblebici et al. "A Comparison of Incremental Triple Graph Grammar Tools". In: *ECEASST* 67 (2014).

[Mar+15] Salome Maro et al. "On integrating graphical and textual editors for a UML profile based domain specific language: an industrial experience". In: *Proceedings of SLE*. 2015, pp. 1–12.

[Obj15] Object Management Group (OMG). *Unified Modeling Language (UML) – Version 2.5*. 2015.

[Obj16] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification – Version 1.3*. 2016.

[Pos+14] Christopher M. Poskitt et al. "Towards Rigorously Faking Bidirectional Model Transformations". In: *Proceedings of AMT*. 2014, pp. 70–75.

[Sch94] Andy Schürr. "Specification of Graph Translators with Triple Graph Grammars". In: *Proceedings of WG*. 1994, pp. 151–163.

[SG16] Stephan Seifermann and Henning Groenda. "Survey on Textual Notations for the Unified Modeling Language". In: *Proceedings of MODELSWARD*. 2016, pp. 28–39.

[SH] Stephan Seifermann and Jörg Henß. COOPERATE UML CLASS DIAGRAM SYNTAX MODELS *v0.1 in Bx Examples Repository*. `http://bx-community.wikidot.com/examples:home`. accessed: 01.03.2017.

[WM15] E. Willink and N. Matragkas. "QVT Traceability: What does it really mean?" In: *Proceedings of AMT*. invited talk. 2015.