Common Pitfalls of Using QVT Relations – Graphical Debugging as Remedy*

Angelika Kusel, Wieland Schwinger
Johannes Kepler University Linz
Austria
kusel@bioinf.jku.at, wieland@schwinger.at

Manuel Wimmer
Vienna University of Technology
Austria
wimmer@big.tuwien.ac.at

Werner Retschitzegger
University of Vienna
Austria
werner.retschitzegger@univie.ac.at

Abstract

OMG's Model-Driven Architecture (MDA) has emerged as a new approach for the development of software. For this, the Query/View/Transformation (QVT) standard plays a central role, since it allows for the specification of model transformations. Nevertheless, until now, QVT-tool support in general and debugging support in particular in the context of MDA are rather limited, supposable being a reason, that the adoption of QVT in practice has not yet been achieved. We therefore propose graphical debugging for the QVT Relations language based on TROPIC - a model transformation approach on the basis of Coloured Petri Nets. By enabling debugging on the TROPIC level, one gains several advantages when developing transformations. Firstly, debugging can take place at a high level of abstraction. Secondly, it serves for explicating the operational semantics of a transformation. Thirdly, it provides a homogenous representation of all transformation artifacts. As a first step towards QVT debugging, this paper aims at a deeper understanding of the operational semantics of QVT, classifying common pitfalls by using QVT and discussing how they may be identified at the TROPIC level.

1. Introduction

OMG's Model-Driven Architecture (MDA) [1] has emerged as a new approach for the development of software, placing models as first-class artifacts throughout the software lifecycle. Thereby a collection of standards arose, whereby *Query/View/Transformation (QVT)* [2] plays a central role, allowing for the specification of model transformations. The QVT standard specifies three sub-languages for transforming models, being the declarative high-level *Relations* and the low-level *Core* languages and the imperative *Operational Mappings* language extending the two previous ones in order to express more complex transformations, challenging to be described purely declaratively. Although the QVT standard claims that the operational semantics of the QVT Relations language is specified by a mapping to the low-level QVT Core language, the actual meaning thereof is very hard to grasp, possibly being a reason that the pragmatics of the language in order to transform models, i.e., how to use the QVT Relations language, is poorly understood. Moreover, until now, QVT-tool support in general and debugging support in particular are still in its infancy [3] supposable being one of the reasons, that the adoption of QVT in practice has not yet been achieved. The main problems are firstly, that the debugging process today takes place on a considerably lower level of abstraction and therefore transformation designers have to cope with an impedance mismatch between the high-level design time, i.e., QVT Relations code, and the low-level runtime in terms of the underlying execution engine. Secondly, since the QVT Relations language is declarative in nature, the operational semantics remains hidden to the transformation designer unless the tool supports an explicit runtime model. And thirdly, information offered by current execution engines is limited with respect to the current transformation state, i.e., only variable values as well as logging messages from the execution engine are offered.

We therefore propose graphical debugging for the QVT Relations language based on our TRansformations On Petri nets In Color (TROPIC) framework [4], [5], [6], which has been developed in the course of the ModelCVS project [7] in order to enhance the understanding and debugging of model transformations. By accomplishing the debugging on the TROPIC level, one gains several advantages. Firstly, it allows for the debugging on a high level of abstraction, thereby overcoming the impedance mismatch between the high-level specification and low-level execution of QVT Relations code. Secondly, it allows for the debugging on the basis of an explicit runtime model, making explicit the afore hidden operational semantics. Thirdly, it allows for a homogeneous representation of all artifacts involved in a transformation, including metamodels, models and the actual transformation logic itself, thus offering comprehensive information about the current transformation state.

The remainder of this paper is structured as follows. Section 2 motivates the need for debugging support by means of a simple example. Generalizing from this simple example, Section 3 proposes a classification of potentially emerging pitfalls when using the QVT Relations language. Section 4 investigates related work and finally, Section 5

* *This work has been partly funded by the Austrian Science Fund (FWF) under grant P21374-N13.*

concludes this paper with an outlook on future work.

2. Graphical Debugging for QVT Relations

In this section we show on the basis of a very simple example, namely a snippet of the well-known Class2Relational (http://sosym.dcs.kcl.ac.uk/events/mtip05) transformation (cf. Figure 1), that the understanding of the operational semantics is far from being a trivial task even for very simple examples like this. Nevertheless, it allows us to demonstrate the benefits of employing TROPIC for identifying the incorrect usage of QVT Relations.

Thereby, the Class metamodel has been reduced to the concepts `Package` consisting of `Classes`, whereby a `Class` has a `name` and can be marked as being persistent (cf. upper left box of Figure 1). The Relational metamodel comprises the equivalent concepts, being `Schema` and `Table` (cf. upper right box of Figure 1). Furthermore, the bottom part of Figure 1 illustrates an example input model (left side) and a desired output model (right side) of the transformation process, which reveals that only persistent classes should be transformed into tables.

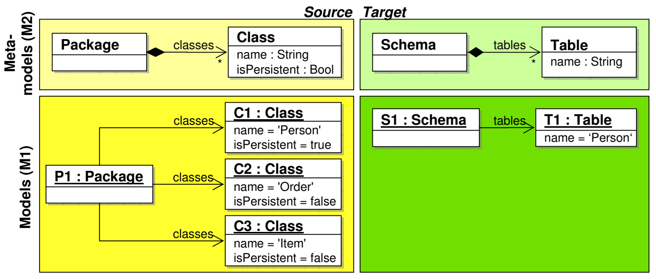


Figure 1. Class and Relational Metamodels and Models

2.1. Potential Realizations of the Running Example

The left part of the Figures 2, 3 and 4 present three QVT Relations realizations of the example described above, which transformation designers might come up with in a first shot, whereas only one of them produces the desired result. In order to spot the right transformation and to identify the incorrectness in the wrong ones, one has to resort to the limited debugging facilities of existing QVT Relations implementations (cf. Section 4 for a detailed discussion).

Therefore, we employ in this paper graphical debugging for the QVT Relations language based on TROPIC. Thereby all transformation artifacts get represented into one single view, whereby metamodels and models get represented as *places* and *tokens*, respectively, and the actual transformation logic is embodied by means of *transitions* in the petri net. For implementing the transition firing behavior, we employ color patterns which can be seen as variables which are bound to the colors of incoming tokens. For details of how to use color patterns for implementing transformation logic, we kindly refer the interested reader to [6].

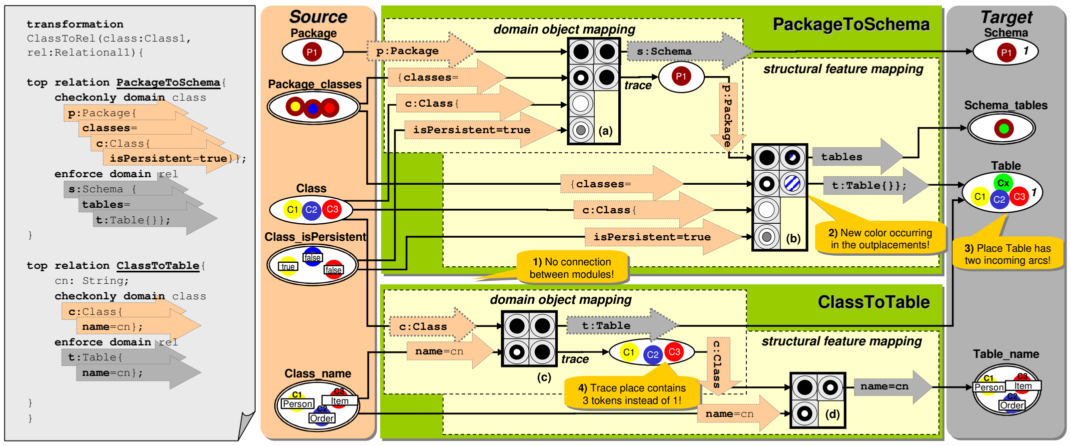
2.1.1. Realization I: Multiple unrelated top relations. One first attempt of solving the example described above could be the realization I shown in Figure 2. Thereby, two relations `PackageToSchema` and `ClassToTable` - marked as being top - are defined. The TROPIC view depicted right aside of the QVT Relations code makes explicit the operational semantics. Thereby, each relation is translated into a corresponding TROPIC unit, containing two transitions, whereby the first of these—(a) and (c)—are responsible for producing the domain objects, i.e., `Schema` and `Table` instances, respectively, and the second of these—(b) and (d)—are responsible for mapping the structural features of the respective domain objects, i.e., attributes and references as described by the domain patterns.

Taking a brief look at the resulting output marking of the TROPIC view, representing the instances of the target metamodel, one can see that this is obviously not the intended result, since, e.g., the place `Table` contains four tokens instead of just one, i.e., four tables have been generated. But since it is not always that easy to recognize a wrong output model, especially if the model size is growing larger, an interesting point is, if we could spot any incorrectness of the transformation specification already in the TROPIC transformation logic.

One potential bug in this realization, that can be spotted quite easily, is that the two TROPIC units `PackageToSchema` and `ClassToTable` work entirely independent of each other since there is no arc between them, giving a hint that there might be a missing `when` or `where` clause, respectively, connecting the two relations. Therefore, the resulting output model will contain model elements which are unconnected to each other. This can be verified on the target side of the TROPIC view, since there are three tokens in the place `Table` (`C1`, `C2`, `C3`), which have no corresponding tokens in the `Schema_tables` place, i.e., the connection to the schema is missing.

Another part in the TROPIC view that could cause scepticism is the transition (b), since a new color (the shaded one) is occuring in the outplacements, i.e., the element in the target domain pattern was not bound by a variable to the element of the source domain pattern. This could have been of course intended by the transformation designer if she wanted to create an object that has no direct correspondence to a source model element, but in our case this is not intended since there is a direct correspondence between the element `Class` and the element `Table`.

A further source of distrust is that the place `Table` has two incoming arcs, i.e., both relations (`PackageToSchema` and `ClassToTable`) generate elements of this type. Therefore, if the two relation conditions do not select disjoint subsets, the resulting output model could potentially contain elements twice. This is in fact the case in our example, since the tokens `Cx` and `C1` in the place `Table` both originate from the class `C1`.



```
transformation
ClassToRel(class:Class1,
rel:Relational1){

top relation PackageToSchema{
    checkonly domain class
      p:Package{
        classes=
          c:Class{
            isPersistent=true}};
    enforce domain rel
      s:Schema {
        tables=
          t:Table{}};
}

top relation ClassToTable{
    cn: String;
    checkonly domain class
      c:Class{
        name=cn};
    enforce domain rel
      t:Table{
        name=cn};

}
}
```

Figure 2. Potential Realization I and the Translation in the Corresponding TROPIC View

Therefore, such a case gives a strong indication to inspect the conditions of the two relations critically, if they really select disjoint subsets of the incoming places.

Another interesting source of debugging information can be found in the trace places of the TROPIC view. Having a look at the trace place of the `ClassToTable` TROPIC unit, one can see, that it contains three tokens, representing the three classes transformed into tables. Having in mind, that only one class of the input model was marked as being persistent, the condition of the transition is obviously too weak, giving a hint to review it.

2.1.2. Realization II: Calling relation is too weak with respect to the called relation. After having realized that the two relations must be interrelated by a `when` or `where` clause, a second attempt to obtain the desired result could be the realization II as depicted in Figure 3. Thereby, again two relations `PackageToSchema` and `ClassToTable` have been defined, but this time only the relation `PackageToSchema` is marked as being top, whereby the relation `ClassToTable` is invoked in the `where` clause of the former. Right of this code example, the resulting TROPIC view is depicted. Again each relation is translated into a corresponding TROPIC unit, whereby only the `PackageToSchema` TROPIC unit contains two transitions (again one for the domain object mapping and one for the structural feature mapping), since the second relation `ClassToTable` does not have to generate a domain object as this is already done by the calling relation `PackageToSchema`. Therefore, only one transition for the mapping of the structural features is required.

Having a look at the output, one can detect that this is again not the desired result, since, e.g., there are three tokens in the place `Table` instead of just one.

At first sight, no error can be detected within the TROPIC view, but the inspection of the trace places highlights the problem. The trace place, holding the tokens representing the generated tables, contains three instead of just one. Therefore it can be inferred that the condition of this relation is too weak. Obviously, the condition of the relation `ClassToTable` is more restrictive, since it lets pass one token only, giving a hint, that this relation specifies the correct strictness.

2.1.3. Realization III: Correct solution. After having corrected this last bug by shifting the condition that only persistent classes should be transformed into tables from the relation `ClassToTable` into the calling relation, one could come up with a correct solution as depicted in Figure 4. Finally, only classes, marked as being persistent, got translated into a table with the corresponding structural features.

3. A Taxonomy of common QVT Pitfalls

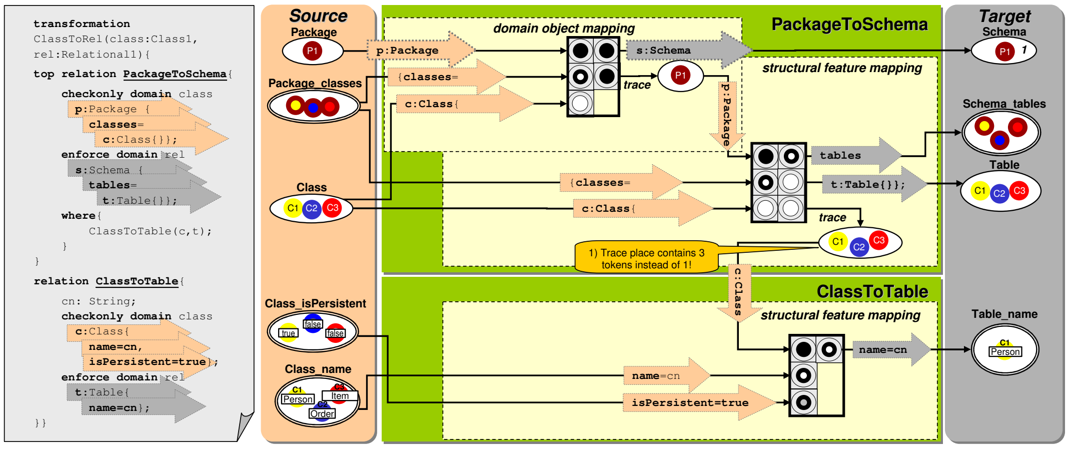
In the course of the development of the graphical debugging, knowledge about how to use or better not use the language, has been gained by systematically varying QVT Relation descriptions which led to a taxonomy of common pitfalls (cf. Figure 5). We distinguish between intra-relational pitfalls, concerning one relation only, and inter-relational pitfalls, concerning more than one relation. In the following each category is described in a pattern like style, whereby foremost the pitfall is described succeeded by a hint, how to spot this kind of pitfall in the TROPIC view.

3.1. Intra-Relational

3.1.1. Source domain pattern.

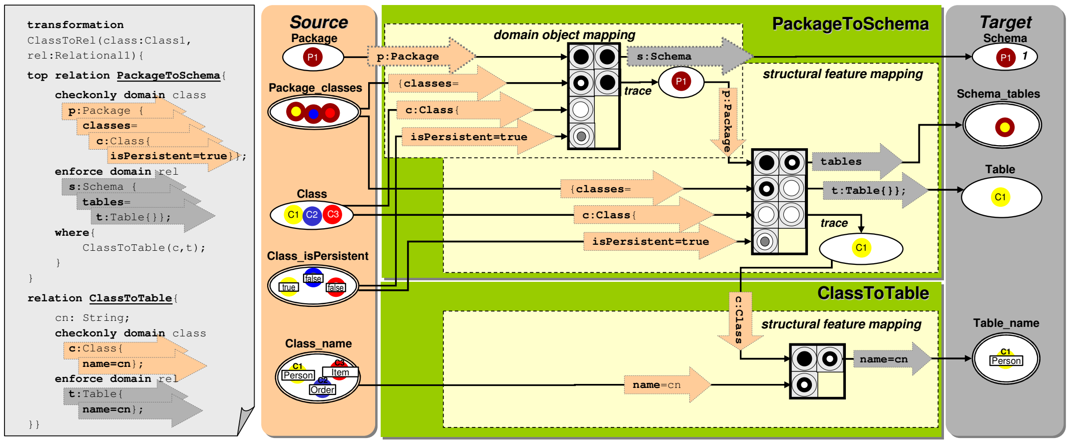
Wrong pattern condition. One basic pitfall is, that the source domain pattern specifies a wrong condition, i.e., it is either too weak resulting in too many object matches or it is too restrictive resulting in too few object matches (cf. eye-catcher 4 of realization I in Figure 2).

Detection in TROPIC: Inspect the trace places of the respective relations and compare the number of tokens residing in



```
transformation
ClassToRel(class:Class1,
rel:Relational1){
top relation PackageToSchema{
    checkonly domain class
      p:Package {
        classes=
          c:Class{}};
    enforce domain rel
      s:Schema {
        tables=
          t:Table{}};
    where{
        ClassToTable(c,t);
    }
}
relation ClassToTable{
    cn: String;
    checkonly domain class
      c:Class{
        name=cn,
        isPersistent=true};
    enforce domain rel
      t:Table{
        name=cn};
}}
```

Figure 3. Potential Realization II and the Translation in the Corresponding TROPIC View



```
transformation
ClassToRel(class:Class1,
rel:Relational1){
top relation PackageToSchema{
    checkonly domain class
      p:Package {
        classes=
          c:Class{
            isPersistent=true}};
    enforce domain rel
      s:Schema {
        tables=
          t:Table{}};
    where{
        ClassToTable(c,t);
    }
}
relation ClassToTable{
    cn: String;
    checkonly domain class
      c:Class{
        name=cn};
    enforce domain rel
      t:Table{
        name=cn};
}}
```

Figure 4. Potential Realization III and the Translation in the Corresponding TROPIC View

it with the number of expected tokens. If it is too high, you have an indication that the condition might be too weak and if it is too low, that the condition might be too strict.

Wrong pattern granularity. Another pitfall concerns the pattern granularity, i.e., the number of 1:n relationships occurring in the pattern. Starting from the domain object, only a simple 1:n relationship is permitted (e.g. between `Package` and `Class`), since a further 1:n relationship (e.g. between `Class` and `Attribute`) would lead to too many matches which is very likely not intended.

Detection in TROPIC: Inspect the trace places of the respective relations and search for duplicates, i.e., tokens with the same color. If you find duplicates, there is a strong indication that there were too many matches.

3.1.2. Target domain pattern.

Wrong connection to source domain pattern. A further pitfall may occur if the target domain pattern is wrongly connected to the source domain pattern by missing or incorrect variable assignments (cf. eye-catcher 2 in Figure 2).

Detection in TROPIC: Compare the colors of the tokens on the left side with the colors of the tokens on the right side of a transition. If you detect additional colors on the right side, then these objects have not been bound by variables.

Wrong pattern granularity. As on the source domain pattern side, also on the target domain pattern side more than one 1:n relationship leads to too many matches since the correlation between the source and target is lost.

Detection in TROPIC: Inspect the trace places of the respective relations and search for duplicates, i.e., tokens with the same color. If you find duplicates, there is a strong indication that there were too many matches.

3.2. Inter-Relational

3.2.1. Source domain patterns.

Calling relation is too weak with respect to the called relation. If the source domain pattern of the calling relation specifies a less restrictive condition than the one of the called relation, then this can lead to objects in the output model that lack structural features, i.e., attributes and references, as created by the called relation (cf. realization II in Figure 3, where only class C1 has a name).

Detection in TROPIC: Inspect the trace place, holding the tokens for the domain object and compare the number of tokens to the number of tokens in the target place of a structural feature produced by the called relation. If tokens in the place of the structural feature are missing (e.g., you have three `Class` tokens but only one `Class_name` token), then you have an indication that the condition of the called relation is too restrictive.

Calling relation is too restrictive with respect to the called relation. If the condition of the calling relation is too restrictive (cf. wrong pattern condition) then the structural features as created by the called relations are absent too in the result.

Detection in TROPIC: Inspect the trace place, holding the tokens for the domain object and compare the number of tokens to the number of expected tokens. If it is too low, you know that the condition is too strict and that also the called relations are affected by this pitfall.

Missing specification of parts. If none of the source domain patterns of the relations match a certain part, i.e., certain metamodel elements, then this part will not participate in the transformation process and therefore will not result in any model elements on the target side.

Detection in TROPIC: Search for source places that have no arc to any transition. If you find any, then you know, that these parts do not participate in the transformation process.

Redundant specification of parts. If more than one source domain pattern matches a certain part, then this can lead to redundant parts on the target side unless the conditions match disjoint subsets.

Detection in TROPIC: Search the target places for duplicates, i.e., same-coloured tokens, and search for source places that have more than one arc originating from it. If you find such a situation, then you have an indication, that there are parts specified redundantly.

3.2.2. Target domain patterns.

Missing specification of parts. As on the source side, also on the target side some parts may be missing, i.e., they are not specified by any relation.

Detection in TROPIC: Search for target places that have no arc from any transition. If you find any, then you know, that these places do not participate in the transformation process.

Redundant specification of parts. Again as on the source side, if some parts are specified more than once by the target domain patterns, then these parts will be produced several times (cf. eye-catcher 3 of realization I in Figure 2).

Detection in TROPIC: Search the target places for duplicates (same-coloured tokens) and search for target places that have more than one incoming arc. If you find such a situation, then you know, that there are parts specified redundantly.

3.2.3. Coherence between relations.

Multiple unrelated top relations. If several unrelated top relations are specified, i.e., without any `when` clause between them, then they work entirely independently of each other, resulting in unconnected parts in the output model. This can be intended, if the input model consists of unconnected parts too, but normally this won't be the case (cf. realization I in Figure 2).

Detection in TROPIC: Search for TROPIC units that have no connection to a trace place of another TROPIC unit. If you find such a situation, then you know, that there are relations that work independently of each other.

Wrong connection style. A question that inevitably arises is deciding between calling a relation in a `when` clause or a `where` clause, respectively, and deciding when to mark a relation as being top.

Detection in TROPIC: Inspect the trace places of the relations and compare the number of tokens residing in it with the number of expected tokens. If it is incorrect, then there is a strong indication that the relations are wrongly connected.

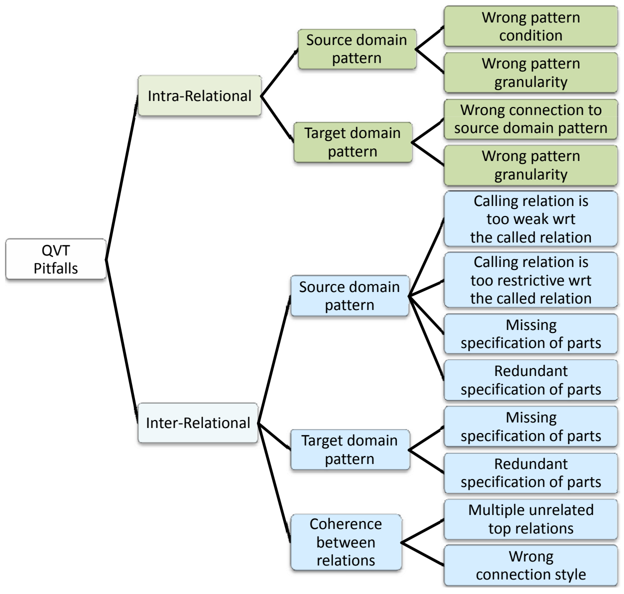

QVT Pitfalls
Intra-Relational
Source domain pattern
Wrong pattern condition
Wrong pattern granularity
Target domain pattern
Wrong connection to source domain pattern
Wrong pattern granularity
Inter-Relational
Source domain pattern
Calling relation is too weak wrt the called relation
Calling relation is too restrictive wrt the called relation
Missing specification of parts
Redundant specification of parts
Target domain pattern
Missing specification of parts
Redundant specification of parts
Coherence between relations
Multiple unrelated top relations
Wrong connection style

Figure 5. A Taxonomy of Common QVT Pitfalls

4. Related Work

We distinguish between three kinds of related work: firstly, related work concerning QVT Relation tool support, secondly, related work concerning our methodology using a translational approach, i.e., transforming QVT Relations into other representations, and thirdly, related work concerning methods for debugging model transformations.

QVT Relation environments. QVT Relation tool support has been developed but as mentioned in [3], it is still in its infancy. The most advanced tool is Medini QVT (http://projects.ikv.de/qvt) which provides debugging features. However, these features are based on the typical Eclipse debugger and are therefore considerably limited. In particular, the debugger only allows to inspect variables in a certain execution state and logging messages from the execution engine are shown. This makes it hard to recognize what is really going on during a transformation, because neither the output model nor the trace model can be accessed before the transformation has been finished. Other QVT Relation tools are ModelMorf (http://www.tcs-trddc.com/ModelMorf) and MOMENT-QVT (http://moment.dsic.upv.es/content/view/34/75/), but no information about debugging features could be found.

Translational Approaches. Several translational approaches have been proposed for executing QVT Relations on top of existing technologies. Jouault and Kurtev [8] propose to execute QVT Relations within the ATL Virtual Machine (ATL VM), by transforming QVT Relations into ATL VM code. Romeikat et al. [9] propose to transform QVT Relations into QVT Operational Mappings and execute the result with QVT Operational tools such as SmartQVT (http://smartqvt.elibel.tm.fr/). These two approaches transform QVT Relations into code which is on a lower level of abstraction and are therefore not suitable for debugging. Greenyer and Kindler [10] propose to transform QVT Relations into Triple Graph Grammars (TGGs) which can be executed in TGGs tools such as Fujaba (http://wwwcs.upb.de/cs/fujaba). Because QVT Relations and TGGs are conceptually and also syntactically similar, one can remain on the same abstraction level. However, the debugging problem is only shifted, because TGGs are not directly executable within existing tools. Again, TGGs have to be translated into executable instructions which are not suitable for debugging.

Debugging transformations. To the best of our knowledge, there is only one work regarding the debugging of model transformations. Hibberd et al. [11] present forensic debugging techniques for model transformations by utilizing the trace models of model transformation executions for determining the relationship between source elements, target elements, and the involved transformation rules. With the help of such trace models, they are able to answer debugging questions implemented as queries which are important for localizing bugs. In addition, they present a technique based on program slicing for further narrowing the area where a bug might be located. The work of Hibberd et al. is orthogonal to our approach, because we are using live debugging techniques instead of forensic. However, with our approach it is also possible to answer most of the debugging questions they raise based on the visualization of the path a source token has taken to become a target token.

5. Conclusion and Future Work

In this paper, we have proposed graphical debugging for QVT Relations based on TROPIC. By accomplishing the debugging on the TROPIC level, one gains several advantages, being, firstly, the high level of abstraction, secondly, the explicit operational semantics and thirdly, the homogenous representation of all transformation artifacts. This was also the basis for a deeper understanding of the operational semantics of QVT as well as of its pragmatics which led to a classification of common pitfalls.

Several further steps of future work remain, comprising:

Prototypical Implementation. After having formulated the mapping of QVT Relations to TROPIC on a conceptual level, a first prototypical implementation is currently established in order to experiment with a set of testcases.

Integration of OCL. One part of QVT Relations, that has been neglected so far, is the OCL part thereof used especially for describing queries. Further investigations should clarify, how to incorporate this part, i.e., as black-box or white-box.

References

[1] Object Management Group, "MDA Guide Version 1.0.1," http://www.omg.org/docs/omg/03-06-01.pdf, 2003.

[2] ——, "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification," www.omg.org/docs/ptc/07-07-07.pdf, 2007.

[3] I. Kurtev, "State of the Art of QVT: A Model Transformation Language Standard," *3rd Int. Symposium on Applications of Graph Transformation with Industrial Relevance*, 2008.

[4] T. Reiter, M. Wimmer, and H. Kargl, "Towards a runtime model based on colored Petri-nets for the execution of model transformations," *3rd Workshop on Models and Aspects @ ECOOP'07*, 2007.

[5] G. Kappel, H. Kargl, T. Reiter, W. Retschitzegger, W. Schwinger, M. Strommer, and M. Wimmer, "A Framework for Building Mapping Operators Resolving Structural Heterogeneities," *7th Int. Conf. on Information Systems Technology and its Applications*, 2008.

[6] M. Wimmer, A. Kusel, T. Reiter, W. Retschitzegger, W. Schwinger, and G. Kappel, "Lost in Translation? Transformation Nets to the Rescue!" *8th Int. Conf. on Information Systems Technology and its Applications*, 2009.

[7] G. Kappel, E. Kapsammer, H. Kargl, G. Kramler, T. Reiter, W. Retschitzegger, W. Schwinger, and M. Wimmer, "Lifting Metamodels to Ontologies - A Step to the Semantic Integration of Modeling Languages," *9th Int. Conf. on Model Driven Engineering Languages and Systems*, 2006.

[8] F. Jouault and I. Kurtev, "On the architectural alignment of ATL and QVT," *ACM Symposium on Applied Computing*, 2006.

[9] R. Romeikat, S. Roser, P. Müllender, and B. Bauer, "Translation of QVT Relations into QVT Operational Mappings," *1st Int. Conf. on Theory and Practice of Model Transformations*, 2008.

[10] J. Greenyer and E. Kindler, "Reconciling TGGs with QVT," *10th Int. Conf. on Model Driven Engineering Languages and Systems*, 2007.

[11] M. Hibberd, M. Lawley, and K. Raymond, "Forensic debugging of model transformations," *10th Int. Conf. on Model Driven Engineering Languages and Systems*, 2007.