Towards Tracking "Guilty" Transformation Rules

A Requirements Perspective

Loli Burgueño
Universidad de Málaga
Spain
loli@lcc.uma.es

Manuel Wimmer
Universidad de Málaga
Spain
mw@lcc.uma.es

Antonio Vallecillo
Universidad de Málaga
Spain
av@lcc.uma.es

ABSTRACT

Several approaches for specifying the requirements for model transformations have been proposed. Most of them define constraints on source and target models as well as on the relationships between them. A major advantage of these approaches is their independence from transformation implementation languages and transformation implementations. However, when these constraints are used for testing, identifying the model transformation rules that violate the constraints is not possible. In this paper we present an approach for automatically aligning specifications of model transformations and their implementations. Matching functions establish these alignments based on the used metamodel elements in the constraints and rules. We present our first results and outline further use cases where an alignment between constraints and rules is beneficial.

1. INTRODUCTION

Model transformations are critical points in the Model-driven Engineering (MDE) development process. The quality of the resulting system is highly influenced by the quality of the employed model transformations to produce the systems. However, users of transformations have to deal with the problem that transformations are difficult to debug and test for correctness. Such tests require a specification which expresses what is correct and what is not, something that is currently not supported by most transformation languages.

A possible solution is to define with specification languages the requirements that a transformation has to fulfil. There are several approaches available for defining constraints on the input and output models as well as on the relationships between them (for an overview see [12]). These constraints are used as a blueprint for developing the model transformations employing implementation languages such as ATL, QVT, or graph transformations. Thus, the specification and the implementations are normally not coupled at all, which has several advantages but may also lead to disadvantages. In particular, when it comes to tracking errors, the missing traceability between specifications and implementations hampers the debugging process. Often the specifications are employed as oracles to check the transformation result. In case constraints are not fulfilled, the elements involved in the constraint evaluation may give a valuable information for the transformation engineer, but the link to the transformation rules is not available.

To tackle this limitation, we present in this paper a first solution for measuring the alignment between a constraint and a model transformation rule by applying an automated matching function. In particular, three different measures are introduced which provide different viewpoints on the alignment problem. We employ the approach for a case study and finally discuss how this general approach may be applied for specific use cases in model transformation engineering.

2. BACKGROUND

In this section, we shortly introduce the formalisms used in this paper for specifying and implementing model transformations. As we shall see, these formalisms are not integrated, and thus, the developed artifacts are completely independent of each other.

2.1 Specifying Transformations with Tracts

Tracts were introduced in [3] as a specification and black-box testing mechanism for model transformations. They provide modular pieces of specification, each one focusing on a particular transformation scenario. Thus every model transformation can be specified by means of a set of tracts, each one covering a particular use case—which is defined in terms of particular input and output models and how they should be related by the transformation. In this way, tracts allow partitioning the full input space of the transformation into smaller, more focused behavioural units, and to define specific tests for them. Basically, what we do with the tracts is to identify the scenarios of interest to the user of the transformation (each one defined by a tract) and check whether the transformation behaves as expected in these scenarios. Another characteristic of Tracts is that we do not require complete proofs, just to check that the transformation works for the tract test suites, hence providing a *light-weight* form of verification.

In a nutshell, a tract defines a set of constraints on the *source* and *target* metamodels, a set of *source-target* constraints, and a tract *test suite*, i.e., a collection of source models

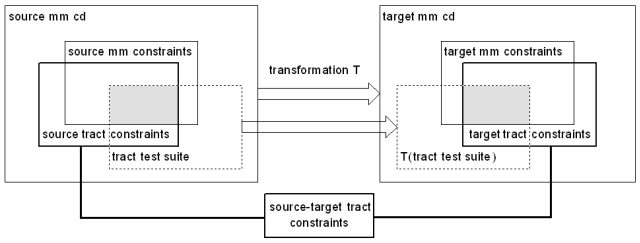

Figure 1: Building Blocks of a Tract.

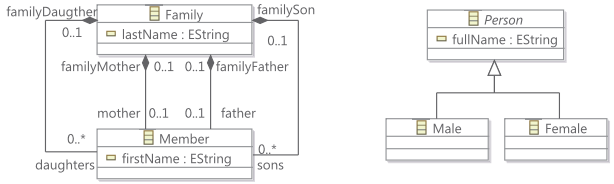

Figure 2: The Family and Person metamodels.

satisfying the source constraints. The constraints serve as "contracts" (in the sense of contract-based design [7]) for the transformation in some particular scenarios, and are expressed by means of OCL invariants. They provide the *specification* of the transformation. Figure 1 gives an overview on the used concepts and their connection.

For demonstrating how to use Tracts, we introduce the simple transformation example *Families2Persons*[1]. The source and target metamodels of this transformation are shown in Figure 2. For this example, a set of tracts is developed to consider only those families which count exactly four members (mother, father, daughter, son):

```
-- C1: SRC_oneDaughterOneSon
Family.allInstances->forAll(f|f.daughters->size=1 and
     f.sons->size=1)

-- C2: SRC_TRG_Mother2Female
Family.allInstances->forAll(fam|Female.allInstances->
     exists(f|fam.mother.firstName.concat(' ').concat(
     fam.lastName)=f.fullName))

-- C3: SRC_TRG_Daughter2Female
Family.allInstances->forAll(fam|Female.allInstances->
     exists(f|fam.daughters->exists(d|d.firstName.
     concat(' ').concat(fam.lastName)=f.fullName)))

-- C4: SRC_TRG_FatherSon2Male
Family.allInstances->forAll(fam|Male.allInstances->
     exists(m| fam.father.firstName.concat(' ').concat
     (fam.lastName)=m.fullName xor fam.sons->exists(s|
     m.firstName.concat(' ').concat(fam.lastName)=s.
     fullName))

-- C5: SRC_TRG_Female2MotherDaughter
Female.allInstances->forAll(f|Family.allInstances->
     exists(fam|fam.mother.firstName.concat(' ').
     concat(fam.lastName)=f.fullName xor fam.daughters
     ->exists(d|d.firstName.concat(' ').concat(fam.
     lastName)=f.fullName)))

-- C6: SRC_TRG_Male2FatherSon
Male.allInstances->forAll(m|Family.allInstances->
     exists(fam|fam.father.firstName.concat(' ').
     concat(fam.lastName)=m.fullName xor fam.sons->
     exists(s|s.firstName.concat(' ').concat(fam.
     lastName)=m.fullName)))

-- C7: SRC_TRG_MemberSize_EQ_PersonSize
Member.allInstances->size=Person.allInstances->size

-- C8: TRG_PersonHasName
Person.allInstances->forAll(p|p.fullName <> '' and
     not p.fullName.oclIsUndefined())
```

[1]The complete example is available at our project website `http://atenea.lcc.uma.es/index.php/Main_Page/Resources/Tracts-ATL`

2.2 Implementing Transformations with ATL

Given this specification, a model transformation language may be selected to implement the transformation. The ATLAS Transformation Language (ATL) [6] is a common choice. ATL is designed as a hybrid model transformation language containing a mixture of declarative and imperative constructs for defining uni-directional transformations. An ATL transformation is mainly composed by a set of *rules*. A rule describes how a subset of the target model should be generated from a subset of the source model. A rule consists of an *input* pattern (having an optional *filter* condition) which is matched on the source model and an *output* pattern which produces certain elements in the target model for each match of the input pattern. The values of the target model elements are assigned in *bindings* which calculate the values by OCL expressions. Given the metamodels in Figure 2 and the tracts, a possible implementation in ATL may be as follows:

```
module Families2Persons;
create OUT: Persons from IN: Families;

helper context Families!Member def: isFemale():
     Boolean = ...
helper context Families!Member def: familyName:
     String = ...

rule Member2Male { -- R1 for short
  from
    s: Families!Member (not s.isFemale())
  to
    t: Persons!Male (fullName <- s.firstName + ' ' +
         s.familyName )
}

rule Member2Female { -- R2 for short
  from
    s: Families!Member (s.isFemale())
  to
    t: Persons!Female (fullName <- s.firstName + ' '
         + s.familyName)
}
```

3. MATCHING CONSTRAINTS AND RULES

As we have seen in the previous section, Tracts allow defining constraints for transformations, while ATL uses rules to express model transformations. Having independent artifacts for the specification and implementation allows for freedom which formalisms to choose for both levels and how to implement the specifications. However, the following questions cannot be answered without through reasoning on both artifact types: (*a*) Which transformation rule(s) implement(s) which constraint(s)? (*b*) Are all constraints covered by transformation rules? and (*c*) Are all transformation rules covered by constraints?

3.1 Challenges

In general, we there are two possibilities to compute alignments between rules and constraints to answer the previous questions. First, there is *static alignment* by reasoning only on the constraints and transformation rules without executing them, and second, there is the possibility to *dynamically* explore the relationships by running the transformations as well as checking the constraints to find overlaps on accessed model elements.

While the second approach may lead to more precise alignments, the alignments are always specific to a given input model. If a more general alignment should be derived, the static approach would more beneficial. However, static alignment seems to be more challenging, because there is a complete paradigm mismatch of the specification language and the implementation language. While in Tracts general OCL expressions are used, in ATL the prime elements are rules. Thus, current generic model matching frameworks cannot be employed, because they produce matches based on structural equivalences. But in our case we have two different languages following different programming paradigms. Thus no structural equivalences are identifiable in a generic manner and other means for comparison have to be found.

The common denominator of constraints and rules are the metamodel elements used, which may give an indication of the relatedness. Therefore, we describe next how this information can be obtained from constraints and rules, compare the extracted information, and present the results to the user.

3.2 Matching Tables: 3 different Viewpoints

For representing the alignments between constraints and rules, we use tabular representations which we call *matching tables*. Our aim is to automatically compute such tabular representations by employing matching functions and to provide different viewpoints on the alignments found. Using different viewpoints on alignments supports answering different questions as outlined in [15].

Given a set of constraints and a set of rules, the corresponding matching tables are computed based on the types of the elements, i.e., the classes from the metamodels, that they contain. In these tables, each cell links a constraint and a rule with a specific value between 0 and 1. Let C_i be the set of types extracted for constraint i and R_j for rule j. In the following, three different metrics are introduced that provide different viewpoints on the types overlaps.

The *constraint coverage* (CC) metric states the coverage for constraint i by a given rule j. For this metric, the value for the cell $[i, j]$ is given by the following formula.

$$CC_{i,j} = \frac{|C_i \cap R_j|}{|C_i|} \tag{1}$$

As the denominator is the number of types in C_i, the result is relative to constraint i and we interpret this value for rule traceability, i.e., to find rules which are related to the given constraint.

The *rule coverage* (RC) metric states the coverage for rule j by a given constraint i. We use this value to express constraint traceability, i.e., to find the constraints most closely related to a given rule. The following formula is used to compute the values for this metric.

$$RC_{i,j} = \frac{|C_i \cap R_j|}{|R_j|} \tag{2}$$

The last metric is relative to both constraints and rules. Thus, it gives a statement of the relatedness of both without defining a direction for interpreting the values. The *relatedness* of *constraints* and *rules* (RCR) metric is as follows.

$$RCR_{i,j} = \frac{|C_i \cap R_j|}{|C_i \cup R_j|} \tag{3}$$

After extracting the types for constraints and rules, there exist five possible cases, as depicted in Figure 3 using Venn diagrams. Let us study each one and comment some of the particular properties of these metrics.

In case (a), every constraint type is contained by the set of rule types, $C_i \subseteq R_j$, thus the value for the CC metric is 1 and it means that the constraint is fully covered by the rule. The other metrics have a value lower than 1.

Case (b) is the opposite to case (a), $R_j \subseteq C_i$, and here the RC metric is always 1. One possible interpretation follows. If after the transformation execution and constraint verification we detect that C_i fails, we know that the failure probably comes from R_j or a part of it and the bigger the value of RC is, the most likely it is that the failure comes from R_j.

For case (c), C_i and R_j are disjoint sets. Thus all the metrics are 0 which means that the given constraint and the given rule are completely independent.

In case (d), every metric will have a value between 0 and 1. The exact value will be dependent on the size of the sets and

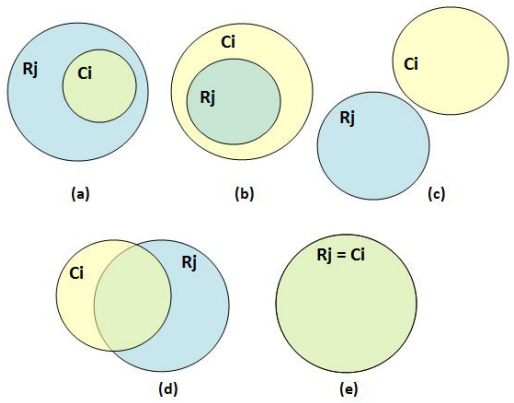

Figure 3: Possible situations for C_i and R_j.

Table 1: Used types for Families2Person example.

Constraint/Rule	Involved Types
C1	Member, Family
C2	Member, Family, Female
C3	Member, Family, Female
C4	Member, Family, Male
C5	Member, Family, Female
C6	Member, Family, Male
C7	Member, Person
C8	Person
R1	Member, Male
R2	Member, Female

Table 2: Families2Person matching tables.

	CC		RC		RCR	
	R1	R2	R1	R2	R1	R2
C1	0.5	0.5	0.5	0.5	0.33	0.33
C2	0.33	0.66	0.5	1	0.25	0.66
C3	0.33	0.66	0.5	1	0.25	0.66
C4	0.66	0.33	1	0.5	0.66	0.25
C5	0.33	0.66	0.5	1	0.25	0.66
C6	0.66	0.33	1	0.5	0.66	0.25
C7	0.5	0.5	0.5	0.5	0.33	0.33
C8	0.0	0.0	0.0	0.0	0.33	0.33

the number of common elements. For example, the bigger the common part for C_i is, the closer the value for metric CC will be to 1. The lower the common part is, the closer CC will be to 0. It is the same with R_j and metric RC. Considering the third metric in case (d), its value depends only on the size of the common part. Thus, the bigger it is, the closer the value is to 1.

In case (e), both types of constraints and rules are the same set, consequently each metric is 1. It is the situation where a constraint and a rule are totally covered by each other.

The *Families2Persons* example presented in the previous section counts on two rules and eight constraints. The types used in the constraints and rules are summarized in Table 1. According to the types extracted for this example, the corresponding matching tables are shown in Table 2. The second and third columns express the constraint coverage, the fourth and fifth the rule coverage, and the sixth and seventh the relatedness. Please note that this is a small example with the only intention of showing how the metrics are computed. Section 4 shows matching tables for a larger example as well as their interpretation.

3.3 Implementation

In order to obtain the result shown in the previous subsection, it is beneficial to have automation support for the matching process. Figure 4 depicts each step of the matching process. The initial input for this processes are the constraints and the transformation rules. The output are the matching tables as explained before.

Starting with the constraint branch, the first step is to extract the types for each constraint. This is achieved by

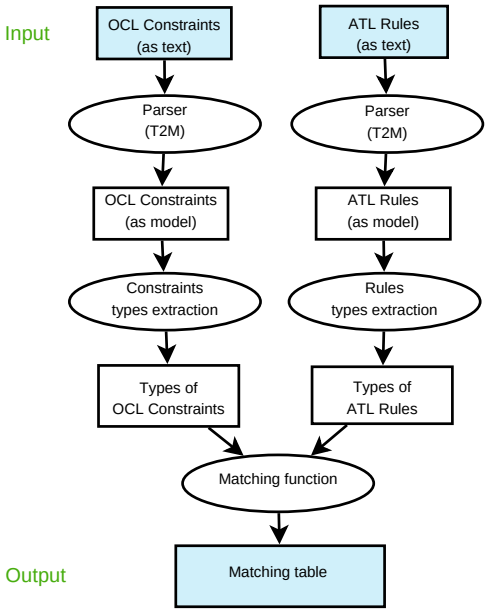

Figure 4: Matching process at a glance.

employing the API of the USE (UML based Specification Environment) tool [10]. This API allows to parse an OCL expression and provides the parsing results in a model-based representation. Using this representation, we are able to extract all the types used within an OCL expression. This is actually provided by having the parse tree representing each subexpression by an explicit node which also provides the return type for each subexpression.

The types extraction for ATL transformations is more challenging compared to the OCL part, because currently there is no support offered by the ATL implementation. However, the textual ATL transformations can be automatically injected to model-based representations. This model-based representation allows to extract the needed information from an ATL transformation by applying another ATL transformation (a so-called higher-order transformation) which generates a model stating for each ATL rule all used types of the input and output pattern elements. Currently, we only support to extract the explicitly given types. The extraction of implicit types used in filter and binding expressions is subject to future work.

Having the used types of all constraints and rules, we may apply the matching functions—which coincide with the metrics described in the previous subsection. The matching functions are implemented in Java and the output of the computation is either represented as an Excel file as well as can be exported as an EMF-based model for further computations, e.g., by applying further model transformations for analysing the matching tables. The *Tracts2ATL* Matcher prototype can be downloaded from our project website.

4. CASE STUDY

Tracts also provide mechanisms for testing model transformations. But when a tract fails, it is useful to know which rule or rules are responsible for the failure. This need origi-

nated the work we are presenting in this paper. Finding the "guilty" rules is supported by rule traceability (CC metric), and complemented by the RCR metric that reflects additional information. As we presented in the previous section, we count on several metrics which allows us to reason on more complicated cases.

4.1 Transformation Example: UML2ER

The *Families2Persons* example is a rather small example, but sufficient for demonstrating the basic process of computing the different metrics. Of course, the usefulness of the process described above is intended to show up for larger examples having a comprehensive set of constraints and rules. In comparison with the *Families2Persons* example, the following *UML2ER* example is larger. This transformation scenario considers the metamodels for a simplified version of UML class diagrams and Entity-Relationship diagrams. The artifacts for this case study can be found at our project website.

The specification of the transformation comprises eight source-target constraints where two kinds of constraints are used. One kind is comparing the amount of instances for a given source and target class, while the other kind is checking for equivalency of elements based on containment relationships and value correspondences. The transformation contains eight transformation rules, whereas three of the rules are abstract rules and a multitude of inheritance relationships between the rules exists: R8, R7 < R6; R6, R5 < R4; R4, R3 < R2.

4.2 Results

We now report on the results we got when applying our approach for the *UML2ER* example. Tables 3-5 illustrate the corresponding matching tables.

Now assume that the transformation is executed, the constraints are checked, and C6 fails. By looking at Table 3, we find a complete coverage of C6 by R3. Thus, it is more likely that the failure is due to R3, instead of coming from R4, R5, R6, R7 or R8. In contrast, the cells R1/C6 and R2/C6 indicate that C6 is completely independent from R1 and R2. Thus, it seems more appropriate in the error tracking process to start with R3 and then continuing with R4-R8.

Let us now suppose that C7 fails. As we can easily identify, the value in R4/C7 of Table 3 is 1, while the value of that cell in the other two tables is different. In this case, R4 is a good candidate for review. If there were more rules with 1 for C7 in Table 3 and lower values for C7 in the other two tables, we should chose the rule which has the higher value for RCR metric (Table 5).

Another situation happens when a rule is fully covered by a constraint. Consequently, the value of the corresponding cell in Table 4 is 1, but not 1 in the rest of the tables. For instance, in Table 4, C8 has value 1 for R1, R3, R4 and R5 but less than 1 in Tables 3 and 5. Again, in this situation it is better to chose the rule with a higher value in Table 5. Another interesting information that we can extract from Table 4 is whether each rule is fully covered by a set of Tracts or if further Tracts are needed to enhance the test coverage.

Table 3: Matching table using CC metric.

	R1	R2	R3	R4	R5	R6	R7	R8
C1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
C2	0.5	0.0	0.5	0.25	0.25	0.25	0.25	0.25
C3	0.33	0.0	0.33	0.5	0.33	0.33	0.33	0.33
C4	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
C5	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
C6	0.0	0.0	1.0	0.5	0.5	0.5	0.5	0.5
C7	0.0	0.0	0.0	1.0	0.5	0.5	0.5	0.5
C8	0.28	0.0	0.28	0.42	0.42	0.28	0.28	0.28
C9	0.25	0.0	0.25	0.37	0.25	0.37	0.37	0.25
C10	0.25	0.0	0.25	0.37	0.25	0.25	0.25	0.37

Table 4: Matching table using RC metric.

	R1	R2	R3	R4	R5	R6	R7	R8
C1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
C2	1.0	0.0	1.0	0.33	0.33	0.33	0.33	0.33
C3	1.0	0.0	1.0	1.0	0.66	0.66	0.66	0.66
C4	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
C5	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
C6	0.0	0.0	1.0	0.33	0.33	0.33	0.33	0.33
C7	0.0	0.0	0.0	0.66	0.33	0.33	0.33	0.33
C8	1.0	0.0	1.0	1.0	1.0	0.66	0.66	0.66
C9	1.0	0.0	1.0	1.0	0.66	1.0	1.0	0.66
C10	1.0	0.0	1.0	1.0	0.66	0.66	0.66	1.0

Table 5: Matching table using RCR metric.

	R1	R2	R3	R4	R5	R6	R7	R8
C1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
C2	0.5	0.0	0.5	0.16	0.16	0.16	0.16	0.16
C3	0.33	0.0	0.33	0.5	0.28	0.28	0.28	0.28
C4	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
C5	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
C6	0.0	0.0	1.0	0.25	0.25	0.25	0.25	0.25
C7	0.0	0.0	0.0	0.66	0.25	0.25	0.25	0.25
C8	0.28	0.0	0.28	0.42	0.42	0.25	0.25	0.25
C9	0.25	0.0	0.25	0.37	0.22	0.37	0.37	0.22
C10	0.25	0.0	0.25	0.37	0.22	0.22	0.22	0.37

In the last case (although not shown in our example), a constraint which fails has values in all tables different from 1. In that case, one more time, we must review the rule with the higher value for Table 5.

To sum up, what is the possibility of finding a rule for a failed constraint? As mentioned before, we have two kinds of constraints. For constraints checking that the amount of instances for source and target classes should be equal (cf. C4 and C5), unambiguous alignments can be found. For the other kind, the situation is different. Depending on the size of the constraints and the amount of used types, we may find several rules having similar alignment ratings.

5. RELATED WORK

With respect to the contribution of this paper, two threads of related work are discussed: first, general traceability approaches in software engineering, and second, specific approaches for tracking "guilty" transformation rules, i.e., those

whose behaviour violate the transformation specifications.

IEEE [4] defines traceability as the degree to which a relationship between two or more artifacts can be established. Most tracing approaches are dedicated to establish traceability links between artifacts that are in a predecessor/-successor relationship with respect to their creation time, e.g., between requirements, features, design, architecture, and code. Our approach for automatically finding the alignments between constraints and transformation rules are in the spirit of traceability rules as presented in [9, 8]. A survey dedicated to traceability in the field of MDE is presented in [2], where the possibilities of using trace links established by model transformations are discussed. However, this survey does not report on tracing approaches between transformation specifications and implementations.

Tracking guilty transformation rules using a dynamic approach, i.e., by executing the model transformation under test, has been subject to investigations. In [14], we used OCL-based queries to backwards debugging of model transformations using an explicit runtime model based on the trace model between the source and target models. Aranega et al. [1] present an approach for situating transformations errors by exploiting also the traces between the source and target models. The dynamic approach is also used by [11] to build slices of model transformations. While these approaches are all tracking transformation rules using specific test input models, our aim is to statically build more general traceability models between transformations' specifications and their implementations. In [5], model footprints of operations are statically computed by the use of metamodel footprints. We pursue the idea of computing metamodel footprints from transformation specifications and implementations for establishing traceability links instead of reasoning on model footprints.

6. NEXT STEPS

The main motivation for this work was the need to track transformation rules that can be considered "guilty" for violating parts of the transformation specifications. Due to the generic nature of the matching tables, a multitude of further use cases emerge.

Properties of Alignments. Based on the matching tables, we are able to reason on the degree of tangling and scattering between constraints and rules. Scattering occurs when a single constraint is scattered across multiple rules, while tangling occurs when a single transformation rule is implementing multiple constraints at once. The work of Berg et al. [13] may be valid input to reason about design guidelines of transformation specifications and implementations based on matching tables.

Refinement of Alignments. More information may be extracted from constraints and transformation rules. For example, from the ATL transformations, inheritance between transformation rules, lazy rule calls, and types used in filters and bindings may be extracted. From the constraints, the accessed metamodel features may be extracted, too. Based on this additional information, more refined alignments may be explored. Furthermore, as we have mentioned in the evaluation, some constraints are using a multitude of types. To distinguish between types, e.g., types only required to navigate to the most relevant information in a model, types occurring more often in constraints may have less impact on the alignments as types that do not as frequently occur.

Alignment-based Slicing. Another direction for future work is to slice model transformations, metamodels, and models based on constraints. This is of course useful for debugging model transformations, however, using slicing techniques may be also beneficial for maintenance tasks. Imagine the requirements are changed by modifying a specific constraint. Adapting the transformation implementation to this change may be easier by reasoning only on a particular slice of the transformation problem referring to a subset of the transformation, metamodel, and models.

7. ACKNOWLEDGMENTS

This work is supported by the research project TIN2011-23795 and by the Austrian Science Fund (FWF) under grant J 3159-N23.

8. REFERENCES

[1] V. Aranega, J.-M. Mottu, A. Etien, and J.-L. Dekeyser. Traceability mechanism for error localization in model transformation. In *ICSOFT*, 2009.

[2] I. Galvão and A. Goknil. Survey of traceability approaches in model-driven engineering. In *EDOC*, 2007.

[3] M. Gogolla and A. Vallecillo. Tractable model transformation testing. In *ECMFA*, 2011.

[4] IEEE. Standard Glossary of Software Engineering Terminology. Technical report, IEEE, 1990.

[5] C. Jeanneret, M. Glinz, and B. Baudry. Estimating footprints of model operations. In *ICSE*, 2011.

[6] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *SCP*, 72(1-2):31–39, 2008.

[7] B. Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, 1992.

[8] F. A. C. Pinheiro and J. A. Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, 13(2):52–64, 1996.

[9] B. Ramesh and V. Dhar. Supporting systems development by capturing deliberations during requirements engineering. *TSE*, 18(6):498–510, 1992.

[10] M. Richters and M. Gogolla. OCL: Syntax, semantics, and tools. In *Object Modeling with the OCL*, 2002.

[11] Z. Ujhelyi, Á. Horváth, and D. Varró. Dynamic backward slicing of model transformations. In *ICST*, 2012.

[12] A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer, and L. Hamann. Formal Specification and Testing of Model Transformations. In *SFM*, 2012.

[13] K. van den Berg, J. M. Conejero, and J. Hernández. Analysis of crosscutting in early software development phases based on traceability. *TAOSD*, 3:73–104, 2007.

[14] M. Wimmer, G. Kappel, J. Schönböck, A. Kusel, W. Retschitzegger, and W. Schwinger. A Petri Net Based Debugging Environment for QVT Relations. In *ASE*, 2009.

[15] W. E. Wong, S. S. Gokhale, and J. R. Horgan. Quantifying the closeness between program components and features. *JSS*, 54(2):87–98, 2000.