A query structured approach to model transformation

Hamid Gholizadeh, Zinovy Diskin, Tom Maibaum

McMaster University, Computing and Software Department, Canada
{ mohammh | diskinz | maibaum }@mcmaster.ca

Abstract. Model Transformation (MT) is an important operation in the domain of Model-Driven Engineering (MDE). While MDE continues to be further adopted in the design and development of systems, MT programs are applied to more and more complex configurations of models and relationships between them and grow in complexity. Structured techniques have proven to be helpful in design and development of programming languages. In this paper, using an example, we explain an approach in which MT specifications are defined in a structured manner, by distinguishing queries as their main building blocks. We call the approach *Query Structured Transformation* (QueST). We demonstrate that the contents of individual queries used in QueST to define a transformation are dispersed all over the entire corresponding MT definition in ETL or QVT-R. Our claim is that the latter two languages are less supportive of a structured approach than QueST. Finally we discuss the promising advantage of QueST in MT definition, and possible obstacles towards using it.

Keywords: Model Transformation, Query Structured Model Transformation, Formal Methods, Model Driven Engineering.

1 Introduction

As the adaption of Model-Driven Engineering (MDE) in the design and development of systems increases in industry, the complexity of Model Transformation (MT) programs —basic MDE operations which transform models to other models— also increases. Structured approaches already have proven to be successful in managing the complexity of systems; for example, the shift from programming with *goto* statements to the structured programming paradigm has proven to be a good design decision, which undoubtedly improved the quality of the software produced. Following this principle, we propose a *Query Structured Transformation* (QueST[1]) approach for defining model transformations which are translating source models to target models. QueST approach is originated from [2, 1], and its mathematical foundation has been developing for the past few years [3, 4]. In this paper, we explain QueST by introducing its structural

[1] The acronym is suggested by Sahar Kokaly, our colleague in the MDE group.

components using a well-known class-to-table example. The structural components of QueST are declarative queries that are used to define target element generation. For each individual target metamodel element, there exists one distinct query in QueST used to define the part of the source metamodel used to generate it. We exhibit the definition of a number of these queries in relation to an example, and explain their execution in QueST. Then, we demonstrate how these queries are dispersed in the body of the corresponding MT programs written in the Epsilon Transformation Language (ETL) [7], and in QVT-Relational [9]. We discuss the reasons for this phenomenon, and finally discuss the benefits of QueST for MT definitions.

The paper is organized as follows: in the next section we introduce the metamodels for the class-to-table example, and informally describe the transformation rules. In Section 3, we explain the structural components of the class-to-table example in QueST, and provide a mathematical definition for a number of these components. Then, we explain how the QueST engine would execute the MT definition. In Section 4, we briefly discuss program building blocks in QVT-R and ETL, and demonstrate the dispersal of the contents of the QueST components (i.e., queries) in the MT definitions corresponding to the same class-to-tables example in these languages. In Section 5, we briefly discuss the reasons for the query dispersals, and also promising advantages of using QueST. Section 6 concludes the paper and mentions potential future research.

2 Class diagram to database schema MT

Metamodels specify valid instances of domains, and are the focal point of MT definitions in QueST. We first briefly describe the source and target metamodels of the class-to-table example and then define the MT in QueST based on them.

2.1 Metamodels

Metamodels of the class diagram (CD) and database schema (DBS) specify the valid model spaces of the CDs and DBSs, respectively. Fig. 1(a) exhibits the CD metamodel. Each `class` contains some `attribute`s associated to it by `atts`; each attribute has a `type` and a multiplicity (`Lbound` and `Ubound`). Each `class` might inherit at most one class (`parent` arrow). `Association`s have multiplicity like `Attributes` and connect `src` classes to `trg` classes.

Fig. 1(b) shows the metamodel of the DBS. Each `Table` has at least one `Column` (see `cols` in the figure). Some columns are primary keys for the table (`pKeys`). A table might also have some foreign keys (`Fkey`). `fKeys` associate these foreign keys to tables. Each `FKey` refers to one table (`refs`) and some columns (`fCols`).

The constraints associated with the metamodels are exhibited using red labels with enclosing brackets. The multiplicities on the arrows are also constraints and therefore they are in red and are included inside brackets. Constraints can be written in any suitable language by interpreting squares as sets and arrows as binary relations. `[isAbstract]` specifies that `NamedElement` is abstract.

`[noLoop]` specifies that there is no loop in the inheritance hierarchy of classes. `[pKeysInCols]` states that the primary keys are columns of the same tables. `[FKeyColIsValid]` states that the columns to which each foreign key refers are subset of table columns of which they are a part (i.e., `fKeys;fCols` $\subset$ `cols`).

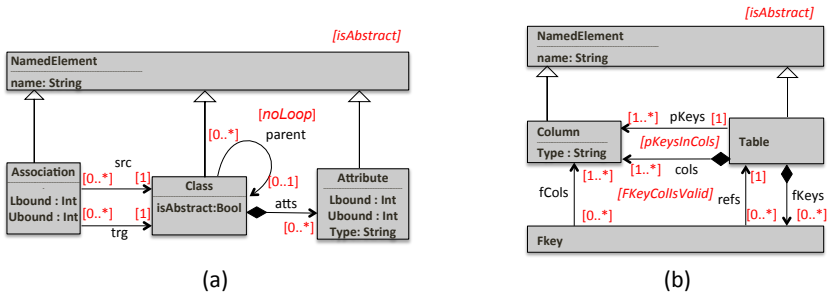

Fig. 1. Class Diagram and DB schema metamodels

2.2 Transformation rules

There are different ways to translate a CD to a corresponding DBS [5]. We propose the following rules as the transformation specification of the example in this paper. For each class we generate a table. Single-valued attributes (svAtts) —those with multiplicity of one or zero— of a class are translated to columns of the corresponding table. A table is generated for each multi-valued attribute —those with the multiplicity greater than one. These tables have two columns: one for keeping the attribute values and another for a foreign key referring to the table corresponding to the attribute containment class. Single-valued associations (svAssoci) are handled by foreign keys; for each svAssoci we create a column in the table corresponding to the source of the association. For each multi-valued association, we create a table with two foreign keys which refer to the source and target of the association, respectively. Inheritance is handled in a way similar to the single-valued associations.

Fig.2(a) shows a class diagram and Fig.2(b) shows its corresponding DB schema, following the rules specified above. In Fig.2(b), FK in parenthesis in front of a column indicates that the column is a foreign key and its outgoing arrow is referring to the table that this foreign key is referring to. As exhibited in the figure, the two tables `takes` and `teaches` are created due to the corresponding two multi-valued associations and the table `telephone` is created due to the corresponding multi-valued attribute.

3 QueST: Query Structured Transformation approach

In a typical MT language, like ETL or even QVT-R, the MT designer thinks in the following way: *how does each pattern in the source model produce a collection of elements in the target model?* But QueST assumes a different manner of thinking. The question that the MT designer is required to think about when

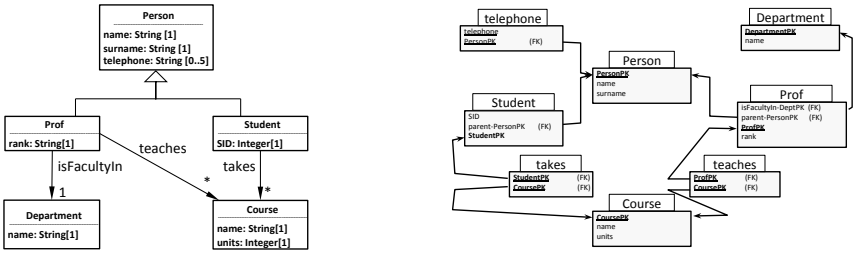

Fig. 2. A class diagram and its corresponding DB schema

starting to specify an MT in QueST is the following: *from which elements of the source model is each element of the target model generated?* This arises from the observation that the target model information is somehow hidden inside the source model and the model transformation program functionality's purpose is to reveal this information by manipulating the information inside the source model and creating the target elements out of the resulting information; for example, according to the MT rules specified in Sec. 2.2, tables in DBS are generated in three different cases: 1) for each class 2) for each multi-valued attribute and 3) for each multi-valued association. This can be specified by defining a query on the CD metamodel. The blue square with a diagonal corner named QTable in the right hand side of Fig.4 represents this query and Fig.3(a) exhibits this query definition in mathematical notation. The plus notation in the query means disjoint union. After this query definition, we associate the `Table` element in the target to this query (the green arrow from `Table` to `QTable` in Fig. 4).

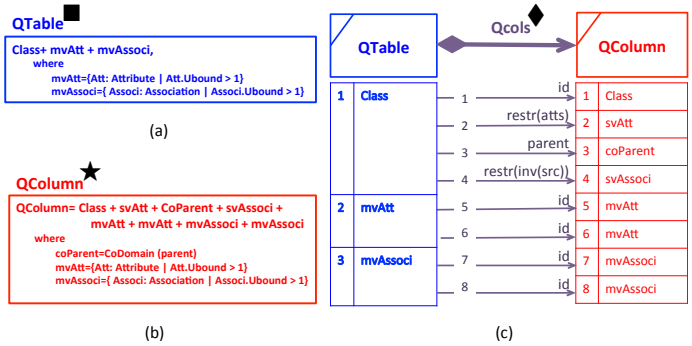

Fig. 3. Query definitions

From the transformation specification, we need to figure out how the columns are generated, and continue to answer similar questions for all the other entities (including squares and arrows) in the target metamodel. This method of thinking is the cornerstone in writing an MT in QueST. Therefore, based on the MT specification, we write a query like the one shown in Fig. 3(b) for the `Column` entity and name it `QColumn`. This query is simply specifying all the possible ways leading to the generation of a column based on the MT specification rules. Then we associate the `Column` in the target metamodel to this query (see Fig. 4).

We apply the same method for the associations between the squares in the target metamodel. Hence we draw an arrow between the `QTable` and `QColumn`

(see purple arrow called `Qcols` in Fig.4) and associate a query to this arrow. The query definition for this arrow is defining the relation between the elements of the `QTable` and `QColumn`. That is the disjoint union of all the arrows which are indexed from one to eight in Fig.3(c); `restr(atts)` is restriction of the `atts` relation over the `svAtt` co-domain; `inv(src)` is the inverse of the `src` relation, and `id` arrows are the identity relations over their domains. The ■ , ♦ and ★ signs on the `Qtable`, `Qcol` and `QColumn` in this figure will be used for comparison purposes in Section 4.

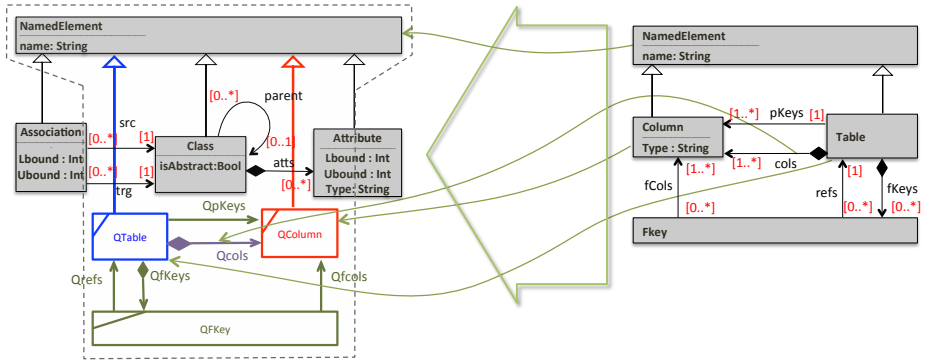

Fig. 4. CD to DBS MT definition in QueST

If we continue the above process of query definition and linking, for the other elements of the DBS metamodel, we will arrive at the structure shown in Fig.4. The entire figure shows the definition of the class-to-table MT in QueST. The new green elements called `QFKey`, `QpKeys`, `Qrefs`, `QfKeys`, and `Qfcols` are all new queries. Their definitions are not shown in the figure due to space limitations, but they are defined in a similar way to the queries defined in Fig. 3. We call the source metamodel with query annotations an *augmented* source metamodel. The augmented CD metamodel is shown in the left hand side of the Fig. 4. As it is shown by the dotted enclosing polygon in the figure, the structure of the target metamodel is somehow replicated in the source metamodel. The association links from the DBS to the CD metamodel show how the DBS metamodel is associated to the augmented CD metamodel. Not all the individual links are shown in the figure; instead a large blank green arrow is used to represent all the links. This linking from the target to the source is total.

From the MT definition in Fig. 4, it is seen that the building blocks of an MT definition in QueST are the queries on the source metamodel, and the links which associate the elements of the target metamodel to these queries (like `QTable`) or the source metamodel elements (like `NamedElement`). This provides a well-formed structure for the model transformation definition; the query definitions are encapsulated inside the squares and arrows, and are independent of each other, and tracing back (by the association links) from the target elements to the augmented metamodel shows how the transformation definition for each entity in the target metamodel is defined. In Section 4 we show how these queries are represented in ETL and QVT-R.

3.1 MT execution in QueST

For a given class diagram, like the one in Fig.2(a), the QueST engine first starts executing the queries of the augmented CD metamodel over the given class diagram. The order in which the queries are executed does not matter semantically, so the execution of the queries can be scheduled in any order by the QueST execution engine. This enables the engine implementation with the possibility of applying any appropriate optimization mechanism over the execution of the queries at the implementation level.

The collected elements from the execution of each query are then typed over that query. For example, Fig 5 shows the elements generated by execution of the `QTable` query that are typed over it (see `:QTable` square in Fig 5). The incoming dotted arrows to the `:QTable` square show from where each element of the square is coming. After all queries are executed, the next step for the QueST engine is to produce the taget elements. This is done by replicating the elements collected by the queries and changing their types according to the association links in the MT definition; for example, for the `QTable` query, the engine first duplicates all the elements collected inside the `:QTable` square and changes their types to `Table`, since the `Table` entity is associated to the `QTable` by a link in the MT definition, as in the previous section. The outgoing dotted arrows from `:QTable` connect the elements to their replicated versions which are all typed over the `Table` element (or simply are tables in DBS).

The process described for the `QTable` query in the previous paragraph continues for all other queries and its completion leads to the generation of the target model shown in Fig 2(b) .

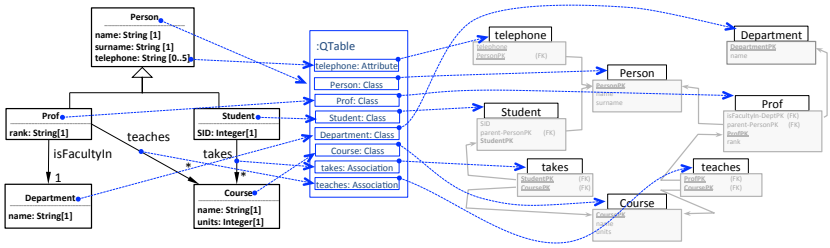

Fig. 5. QueST engine executing QTable query and generating Tables

4 Dispersal of queries in ETL and QVT-R

We have defined the very same MT transformation described in Section 2.2 with ETL, and also with QVT-R. In this section, we briefly explain the structures of the MT definitions is each language, and by marking each transformation code, we demonstrate the wide dispersal of the contents of structural components (i.e., queries) of QueST in these definitions.

4.1 Query dispersal in ETL

An MT in ETL is specified by a set of rules. Each rule defines how an element of a specific type is translated to one or more elements (not all necessarily having

the same type) in the target model. The rules might have guard expressions that constrain their application.

Fig. 6 shows the code for defining the class-to-table example in ETL. Intentionally, the code font size used is very small in the figure, since we will not discuss in detail each part of the definition, and instead, we only take into account the entire transformation definition to show the dispersal of the QueST queries all over it. The parts in Fig. 6 that correspond to the queries in Fig.3 are marked: 1) ■ marking the parts corresponding to the generation of tables (`QTable` query); 2) ★ marking the parts corresponding to the generation of columns (`QColumn` query) and 3)⧫ marking the parts which associate the columns to the tables (`Qcols` Query). The numbers above the markings correspond to the numbers in Fig. 3(c) for each marking; for example, $\overset{2}{\blacksquare}$ refers to the `mvAtt` component of the `QTable`, and $\overset{3}{\blacklozenge}$ and $\overset{3}{\bigstar}$ refer to the `parent` arrow of `Qcols`, and the `coParent` component of `QColumn`, respectively.

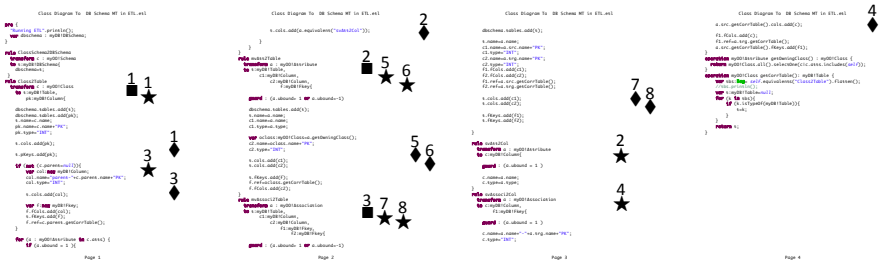

Fig. 6. ETL code for the CD to DBS transformation

As the figure shows, each marking sign is dispersed over the entire code. This means that different components of the queries in Fig 3 are dispersed all over the code in ETL; for example the eight component of the `Qcols` which are indexed from one to eight are scattered throughout the entire definition and among the different rules.

We avoided marking all the queries that are generating the target elements in the definition of Fig. 6 to keep the figure simple. However, by repeating the marking procedure for all of the other queries, we would get a similar dispersal of their content over the entire code. One immediate consequence of these dispersals is the difficulty that occurs during the debugging process in the development of the model transformation; since the user needs to check different parts of the code, if he gets some errors regarding generation of specific element in the target.

4.2 Query dispersal in QVT-R

An MT definition in QVT-R is composed of a set of *top* and *non-top* relations. The difference is that the latter relations are invoked by the former ones. Each relation definition specifies how some elements in the source model are related to some elements in the target model. There are *when* and *where* clauses which act as pre- and post-conditions for the execution of the corresponding relation [8]. At the time QVT-R engine executes an MT definition, it enforces that all the defined top relations hold true, by creating missing elements in the target.

We wrote a transformation in QVT-R for the same class-to-table example, and perform the same analysis over the code as we did in the previous section for the ETL definition. Fig. 7 shows the definition and the marking; the semantics of the markings are identical to what we described before. It is seen from the figure that, similar to the ETL code, the queries for the generation of the target model elements are distributed all over the code.

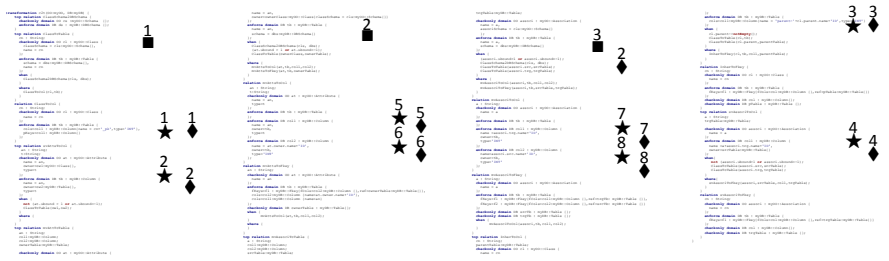

Fig. 7. QVT-R code for the CD to DBS transformation

5 Discussion

This section is divided into two parts: we first discuss the reasons that cause the scattering of each individual query in QueST over the entire MT definitions by ETL and by QVT-R; then, we discuss the promising advantages of QueST from different perspectives.

5.1 Why query dispersal happens in ETL and QVT-R

It might be argued that what are presented in Section 4, as the definitions of the class-to-table example in QVT-R, and in ETL are subjective, in the sense that there might be different implementations for the example by these languages, such that the demonstrated query dispersal are prevented. We believe that the scattering of the QueST queries would happen in any implementation, because of the three reasons briefly discussed below:

One to many and many to many associations. In ETL, each rule associates one source metamodel element to many target metamodel elements. In QVT-R, each relation associates many source metamodel elements to many target metamodel elements. Therefore one target element can be referenced in many rules/relations in a transformation definition in these languages; this means that the queries generating the elements of a specific type are spread between different rules/relations.

Arrows are secondary elements. References to arrows in MT definitions in ETL and QVT-R happen by means of nodes; queries only define nodes, and arrow definitions are implicit inside these queries. More concretely, it is not possible to define an arrow as a target of an ETL rule (i.e., as a rule header parameter), or as a domain of a relation in QVT-R. This means that if the queries generating nodes are dispersed, then the queries for generating the arrows which are referenced by these nodes will also be dispersed. The ⧫ marking signs appearing

everywhere close to the ★ signs in Fig. 7 show this phenomenon.
Flattening the graphical structure. ETL and QVT-R are textual languages, while as it may be seen from the Fig. 4, MT definitions contain graphical constructs. A representation of a graphical construct in a textual format causes a scattering of references to the graphical elements, inside the representation; for example a node with several incoming edges in a graph would be inevitably referenced in different places in the graph's textual representation.

5.2 Promising advantages of QueST

MT design and development. As is shown in Section 3, QueST provides well structured definitions by encapsulating queries inside squares and arrows, i.e., the first class elements of the metamodeling language. All the target elements of a specific type (type could be an arrow or a square) are generated by one, and only one, query. We believe that this is helpful in development and debugging of MT definitions, because it follows the well-known principle of *separation of concerns*, where the concern is the production of elements of specific type. Further, each query definition is *fairly* independent in QueST, i.e., square queries are independent, and arrow queries are only dependent on their corresponding source and target queries; Hence, QueST's structural construct, the query, suitably provides a pattern to decompose complex MT definition tasks into small fairly independent definitions.
Semantic foundations. The mathematical foundation of the QueST approach is already discussed in some other work [2–4, 6]. This ensures that there is a clear and formal understanding of QueST. This would prevent some ambiguity and semantic issues like the ones investigated for the QVT-R specification [10]. Furthermore, its formal foundation provides a context to validate and formally verify MT definitions.
Flexibility in query language. Theoretically, any query language can be used for defining the queries in QueST. The expressive power of QueST depends on the expressive power of the chosen query language. We insist that the chosen query language should consider the arrows as equally as important as the nodes; the experiments of using QueST for MT definitions have shown that defining the square queries are easy in some query languages like OCL, but defining arrow queries are fairly complicated, because they necessarily include many references to the source and target of the arrow.
Declarative vs. imperative. QueST is declarative: the definitions of its structural building blocks (i.e., queries) provide specifications rather than implementations for the generation of the target elements. Further, the queries could be executed in any order in QueST. The advantages of declarative approaches in programming languages is less debatable now; even though the traditional challenges of training MT programers to think declaratively might still be an obstacle, considering the number of people who are trained to write queries declaratively, in the database community, it might be plausible enough to follow a similar pathway in the MT community as well.

6 Conclusion

From one perspective, the intent of MT programs is to collect information from the source model by executing some queries, and, then, to build up the target model elements. Formalizing this procedure suggests a structural pattern for model transformation which we call QueST. In QueST, the MT definer should think in a target-oriented manner, in the sense that he should define a query for each individual element in the target metamodel, during the MT development process. These queries define the generation of target model elements, and are encapsulated inside the fairly independent components which make up the structural building blocks of an MT definition in QueST. We used a well-known class-to-table example to explain these structural components (i.e., queries) and their definitions. We also demonstrated that the contents of these queries are necessarily dispersed all over the entire MT definitions written with ETL and QVT-R. Subsequently, we discussed how the dispersal of each query is not a specific property of the implemented examples and could be generalized to any programs written in QVT-R and ETL. Finally, we discussed the advantages of QueST from different perspectives. A deeper examination and evaluation of QueST, regarding the aspects discussed in the previous section, is the subject for our future work.

References

1. Z. Diskin and J. Dingel. A metamodel independent framework for model transformation: Towards generic model management patterns in reverse engineering. In *ATEM*, 2006.
2. Zinovy Diskin. Mathematics of generic specifications for model management. In *Encyclopedia of Database Technologies and Applications*, pages 351–366. Idea Group, 2005.
3. Zinovy Diskin. Model synchronization: Mappings, tiles, and categories. In *Generative and Transformational Techniques in Software Engineering III*, pages 92–165. Springer, 2011.
4. Zinovy Diskin, Tom Maibaum, and Krzysztof Czarnecki. Intermodeling, Queries, and Kleisli Categories. In Juan de Lara and Andrea Zisman, editors, *FASE*, volume 7212 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2012.
5. David W Embley and Bernhard Thalheim. *Handbook of conceptual modeling: theory, practice, and research challenges*. Springer, 2012.
6. Hamid Gholizadeh. Towards declarative and incremental model transformation. In *DocSymp@ MoDELS*, pages 32–39, 2013.
7. Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The epsilon transformation language. In *Theory and practice of model transformations*, pages 46–60. Springer, 2008.
8. Nuno Macedo and Alcino Cunha. Implementing qvt-r bidirectional model transformations using alloy. In *Fundamental Approaches to Software Engineering*, pages 297–311. Springer, 2013.
9. OMG, http://www.omg.org. *OMG. MOF2.0 query/view/transformation (QVT) version 1.1*, 2009.
10. Perdita Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *Software and System Modeling*, 9(1):7–20, 2010.