

Model Transformation with Triple Graph Grammars

Alexander Königs

Merckstr. 25

Real-Time Systems Lab

University of Technology Darmstadt

D-64283 Darmstadt, Germany

koenigs@es.tu-darmstadt.de

September 23, 2005

Abstract

Model Driven Application Development (MDA) is OMG's vision of model-based software system development. MDA is based on the idea of automatically transforming abstract models into more specific models. In this paper we explain our model integration approach tackling a common case study. Our approach implements triple graph grammars which are declarative and brings them together with OMG's MOF standard. From a set of declarative triple graph grammar rules we (semi-)automatically derive operational graph grammar rules that can be used for consistency checking, consistency recovery, and model transformation using rule application mechanisms. Thus, triple graph grammar are suitable for model integration in general and model transformation in particular.

1 Introduction

Model Driven Application Development (MDA) is OMG's vision of developing software systems in the future [KWB03]. As the name implies, MDA is model-based. The basic idea is that the developer specifies rather abstract models of the desired system, and automatically transforms them into more specific models, resulting in the final system. These transformations are specified by sets of transformation rules that can be reused for a lot of similar software system development projects. Recently, this vision has been addressed by a number of approaches like *GReAT* [AKS03], *BOTL* [Bra04], *IMPROVE* [BW03], or the upcoming *QVT submission* from the *QVT Merge Group* [QVT05]

	QVT	GReAT	BOTL	IMPROVE	TGG
Declarative Specifications	(+)	-	+	(+)	+
Consistency Checking	+	-	(+)	+	+
Model Transformations	Unidirectional	Unidirectional	Bidirectional	Bidirectional	Bidirectional
Notation	Textual (Graphical)	Graphical	Graphical	Graphical	Graphical
Meta-Model Based	+	+	+	+	+
MOF 2.0 - compliant	+	-	-	-	+
Code Generation	+	+	+	-	JMI-compliant
Traceability Support	(+)	-	-	+	+
Change Propagation	?	-	-	(Incremental)	(Incremental)

Figure 1: Classification of different model integration approaches

for instance. Figure 1 classifies these approaches¹ with respect to the following properties² based on OMG's *Request for Proposal: MOF 2.0 Query/Views/Transformations RFP* [OMG02]:

- Support for declarative or operational specifications of model integration rules
- Support for checking whether models are consistent or not
- Support for uni- or bidirectional model transformations
- Textual or graphical notation
- Meta-model based
- Compliant to OMG's MOF 2.0 standard
- Support for code generation
- Traceability support
- Support for change propagation

¹The figure considers the initial version of the QVT submission from the QVT Partners [QVT03]. It is ongoing work to evaluate the revised submission.

²[CH03] provides another set of properties that allow for a classification taking more technical issues into account.

In order to allow for a detailed comparison of different approaches the workshop organizers have provided a common case study. This case study deals with the transformation of class diagrams into database schemas.

Currently, we implement our approach as part of the *Toolnet* project [A⁺03]. *Toolnet* is a model integration platform that provides uniform access to tools' data repositories by means of *JMI*-compliant [Mos02] tool adapters, and the infrastructure for inter-tool communication based on *SOAP* [W3C03]. Actually, *Toolnet* only supports the automatic creation of correspondence links between tools' data objects for traceability and consistency checking purposes. Consistency recovery and model transformation mechanism are ongoing work and will be added in future versions of *Toolnet*.

The specification of the triple graph schema as well as the generation of *JMI*-compliant Java code from them is done by using *MOFLON* [Rea05] which is a *MOF 2.0* [OMG03] diagram editor and code generator plug-in for the upcoming version of the *Fujaba* tool suite [NNZ00]. The specification of declarative triple graph grammar rules and the (semi-)automatic derivation of operational graph grammar rules is done by using an adopted version of *Fujaba*'s triple graph grammar plug-in [Wag01]. Since *Fujaba*'s code generator does not provide *JMI*-compliant code for graph transformations yet the translation of operational graph grammar rules has to be done by hand until an adopted code generator is in place.

We start in Section 2 by giving an overview of our approach. In Section 3 we tackle the common example by specifying a set of declarative triple graph grammar rules, (semi-)automatically deriving the needed transformation rules, and explaining how these rules are applied in order to perform the desired transformation. Section 4 concludes this paper and discusses the results and open issues.

2 Overview of our approach

In this section we present our approach which implements *triple graph grammars* [Sch94], and how we can use it for model transformations.

The basic idea of *triple graph grammars* is that there is a graph that evolves by applying graph grammar rules. This graph must comply to its graph schema at all times. The point is that this graph can be separated into three corresponding subgraphs. Hence, the rules are called triple graph grammar rules. Again, each subgraph must comply to its own graph schema. Two of these subgraphs evolve simultaneously while the third keeps track on correspondences between the other graphs. Merging this idea with OMG's *MOF* standard means that *triple graph grammars* can be used to specify the simultaneous evolution of models, whereas models coincide with graphs as well as meta-models coincide with the corresponding graph schemas.

Since the simultaneous evolution of models does not apply in practice, *triple graph grammars* must

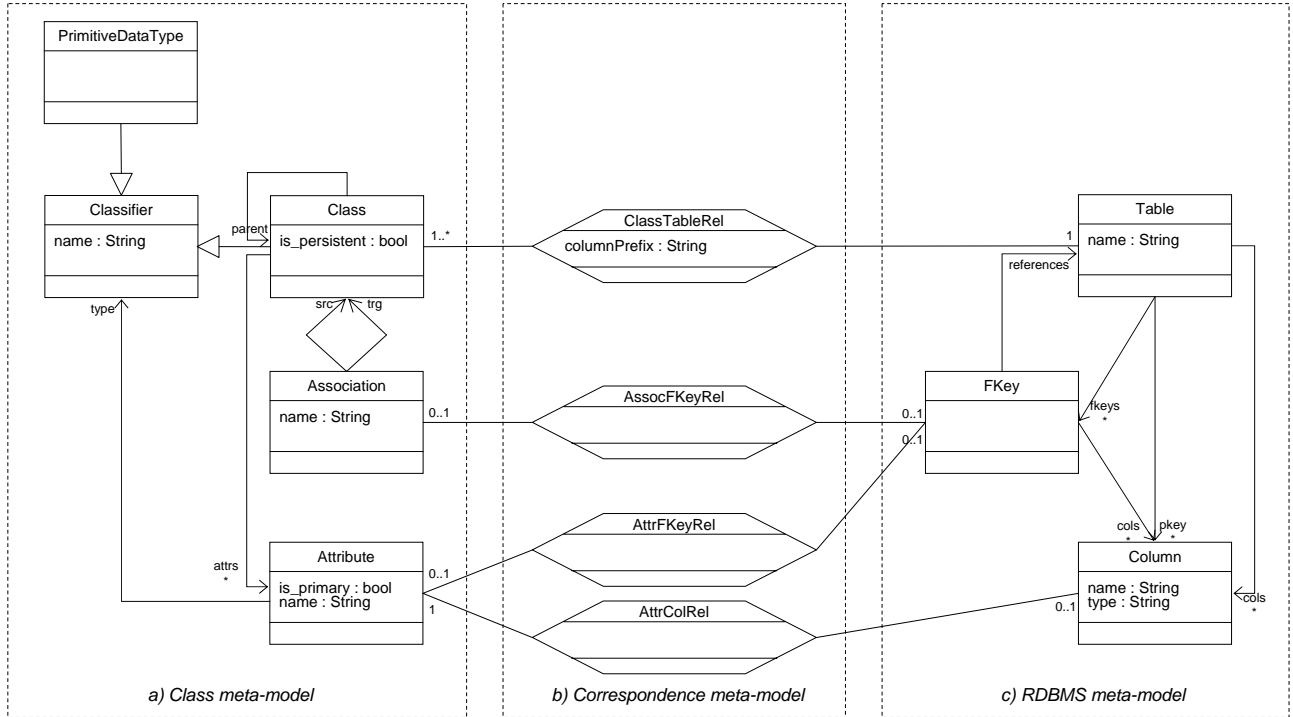


Figure 2: Example of a triple Graph Grammar Schema

be considered declarative. Therefore, we derive a set of operational graph grammar rules from each declarative triple graph grammar rule. These operational rules can then be applied in order to fulfill various model integration tasks as consistency checking, consistency recovery, and bidirectional model transformation.

2.1 Triple graph schemas

As we mentioned above, we need a triple graph schema / meta-model for each model integration scenario. Figure 2 shows the triple graph schema we use in the given case study. Figure 2a depicts the given meta-model for the to be transformed class diagrams. The meta-model states that a class diagram consists of **Classifiers** which can be **PrimitiveDataTypes** or **Classes**. **Classes** own **Attributes**, and can be associated with each other by directed **Associations**. Furthermore, **Attributes** have a type that is a **Classifier**. Figure 2c presents the given meta-model for the database schemas resulting from the transformation. In the case study database schemas consists of **Tables** which owns **Columns** and **FKeys** (foreign keys). Additionally, each **Table** contains a designated set of **Columns** which constitute the primary keys of the **Table**. Finally, each **FKey** refers to a **Table**

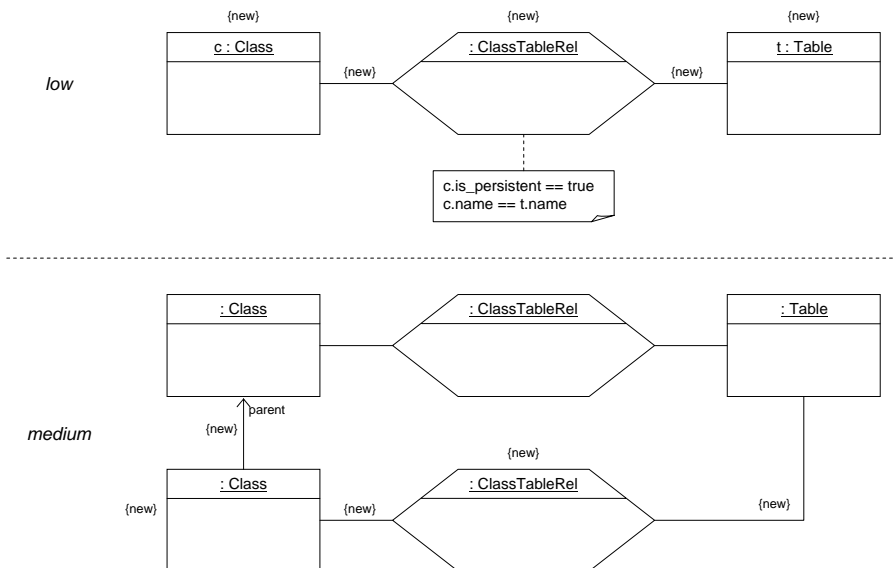


Figure 3: Example of triple graph grammar rules

and denotes a number of `Columns`. Figure 2b presents the meta-model of the link objects which are used by the *triple graph grammar* to keep track of the correspondences between object of the class diagrams and objects of the resulting database schemas. The hexagonal shape indicates that the classes carry a tag `integration`. As we describe in [KS05] in more detail we use this tag for code generation purposes as well as for an abbreviation of multiplicities. The class `ClassTableRel` states that on the one hand each `Class` corresponds to one `Table`³. On the other hand each `Table` corresponds to at least one `Class`. The class `AttrColRel` links one `Attribute` with at most one `Column`, whereas each `Column` is linked with exactly one `Attribute`. Accordingly, the class `AttrFKeyRel` keeps track of the correspondence between an `Attribute` and up to one `FKey` on the one hand, and one `FKey` and at most one `Attribute` on the other hand. Similarly, the class `AssocFKeyRel` links one `Association` with at most one `FKey`, and one `FKey` with at most one `Association`.

2.2 Triple graph grammar rules

Besides a schema a *triple graph grammar* provides a set of triple graph grammar rules. These rules describe how all subgraphs / models evolve simultaneously. Figure 3 shows two examples for such rules. The first rule means that every time a new `Class` `c` is added a new `Table` `t` is added simultaneously. Additionally, a correspondence link with the type `ClassTableRel` which links `c` and `t` is added. The

³This does not imply that each `Class` will be transformed into an own `Table`.

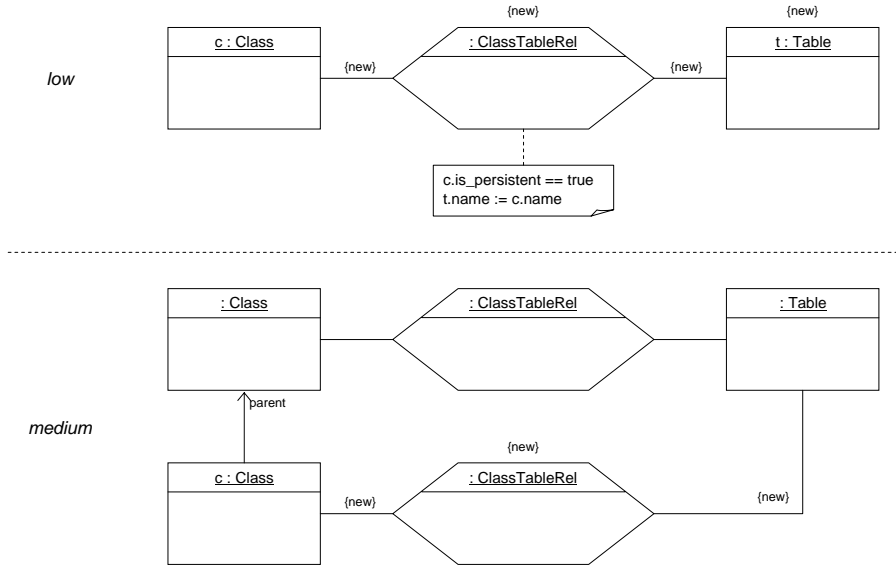


Figure 4: Example of operational graph grammar rules

constraint states that this rule is only applicable if the value of the attribute `is_primary` of `c` equals `true`. Furthermore, the name of `c` must match the name of `t`. Finally, `low` denotes the priority of this rule. The priority of a rule is used by the rule application mechanism as described below.

The second rule means that every time a new `Class` `c` is added as a child of another `Class`, `c` will be linked to the already existing `Table` by a new correspondence link with the type `ClassTableRel`. Furthermore, the already existing `Table` must have been linked to the parent `Class` by a correspondence link with the type `ClassTableRel` before. The constraints states that this rule is only applicable if `c` is persistent. Finally, the priority of this rule is `medium`.

2.3 Operational graph grammar rules

Since the simultaneous evolution of models does not apply in practice, the triple graph grammar rules are not useful as they are. Therefore, we (semi-)automatically derive operational graph grammar rules from them, which can be applied for different model integration tasks. Figure 4 shows the operational forward transformation graph grammar rules which have been derived from the rules depicted in Figure 3. As we point out in [KS05], we derive seven additional rules for consistency checking, consistency recovery, and bidirectional transformations. The first rule shown in Figure 4 means that a persistent `Class` `c` will be transformed into a new `Table` `t`. Furthermore, `c` will be linked to `t` by a new correspondence link of the type `ClassTableRel`. The constraint that deals with created

model elements has been transformed into an attribute assignment that guarantees that the name of the new **Table** matches the name of the to be transformed **Class**. Actually, the transformation of the constraint into an attribute assignment has to be done manually, but we plan to incorporate some kind of constraint solving mechanism in the future. The priority of this rule has been transferred from the corresponding triple graph grammar rule.

The second rule means that a **Class** c which has a parent will just be linked to the same already existing **Table** which is linked to the parent **Class** by a new correspondence link of the type **TableColumnRel**. Again, the priority of this rule has been transferred from the corresponding triple graph grammar rule.

2.4 Rule application

In order to perform a model integration task we need a rule application mechanism. On the one hand in our approach the rule application mechanism of each integration tasks consists of a hard wired algorithm which visits the model elements in the correct order and applies operational graph grammar rules where possible. Each algorithm is specific for a general integration task as consistency checking, consistency recovery, or model transformation, but not for a specific model integration scenario as transforming a class diagram into a database schema for instance. On the other hand we generate code from each operational graph grammar rule which is responsible for matching and applying the pattern described by the rule. The algorithm for the desired forward transformation looks as follows.

1. Start with the set S of model elements of the source model which have no parents by means of a composition relationship.
2. For each element e of S do:
 - (a) Append the children of e by means of composition relationships to S .
 - (b) Identify the set G of operational graph grammar rules which deal with the transformation of e .
 - (c) For each rule r of G test whether r is potentially applicable by means of its pattern. If the pattern requires that another element must be handled before, e is removed from S and appended to the end of S . This guarantees that the elements are visited in the correct order. If r is potentially applicable keep it in a set R of potentially applicable rules.
 - (d) Only keep the rules of R that have the highest priority. Thus, the priorities allow for the application of multiple but not necessarily all matching rules.
 - (e) Apply each rule that remains in R for *every* matching unless stated otherwise.

3 Tackling the example

In this section we show how we tackle the common example using our approach as presented in Section 2. The example deals with the model-based transformation of *class diagrams* into *database schemas*. Figure 2a presents the given meta-model of *class diagrams*. The meta-model of *database schemas* is depicted in Figure 2c. As mentioned above, Figure 2b adds the meta-model of correspondence links which we need for our approach. The task is to map the given description of the desired transformation to our approach. This means that we have to provide a set of triple graph grammar rules from which we derive a set of forward transformation rules that allows for the desired transformation. In order to save space we just present the triple graph grammar rules. We get the corresponding forward transformation rules by omitting the italicized `{new}` tags and considering the constraints which deal with created model elements as attribute assignments. The model transformation task consists of the transformation of **Classes** and inheritance hierarchies on the one hand and of the transformation of **Attributes** and **Associations** on the other hand. Finally, we demonstrate the application of our approach by transforming a given example input.

3.1 Transformation of classes

The case study demands that **Classes** are transformed as follows:

- Classes that are marked as persistent in the source model should be transformed into a single table of the same name in the target model.
- Classes that are marked as non-persistent should not be transformed at the top level.
- In inheritance hierarchies, only the top-most parent class should be converted into a table; the resultant table should however contain the merged columns from all of its subclasses.

The corresponding triple graph grammar rules are shown in Figure 3. The forward transformation rule derived from the first triple graph grammar rule transforms a given persistent **Class** into a **Table** which has the same name as the **Class**. The forward transformation rule derived from the second triple graph grammar rule just links a given **Class** which is a child of an already transformed parent **Class** to the already existing **Table** to which the parent **Class** has been linked before. Since the priority of the first rule is lower than the priority of the second rule, the first rule will only be applied in cases where the second rule is not applicable. Thus, the first rule will only be applied to top-most parent classes. Observe that the second rule can be applied recursively to transform all levels of an inheritance hierarchy. This ensures that **Columns** will be associated with the correct **Tables** later on.

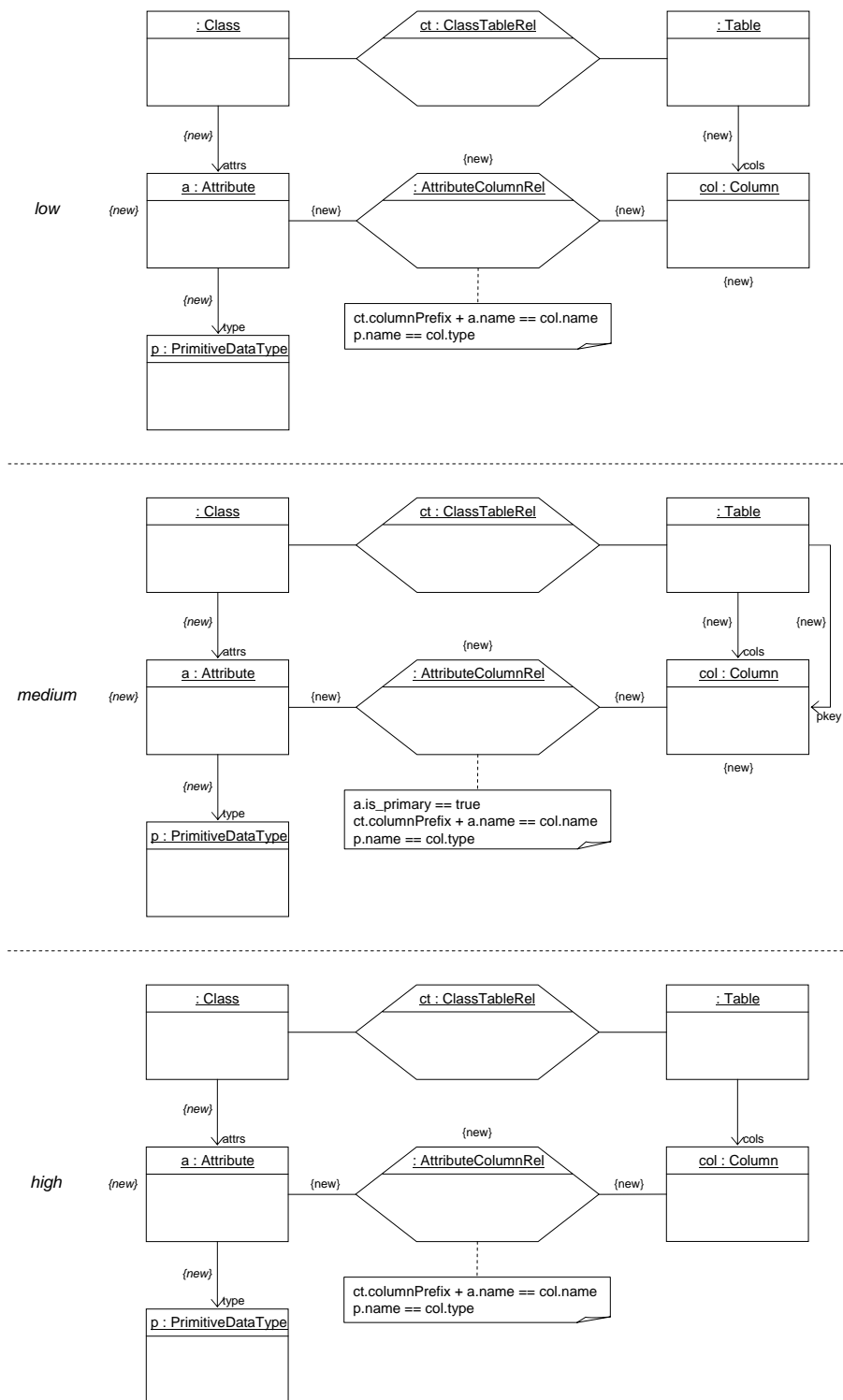


Figure 5: TGG rules concerning attributes with primitive data type

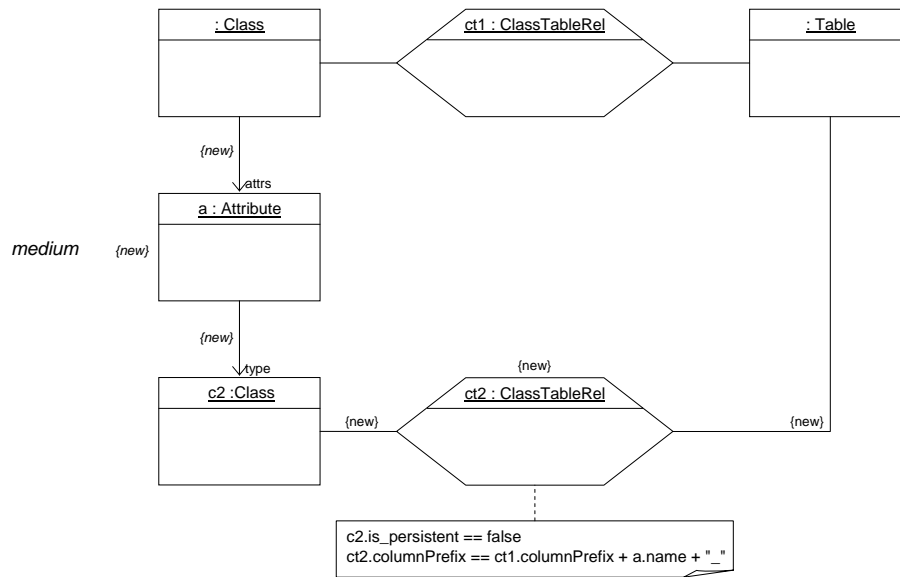


Figure 6: TGG rule concerning attributes with non-persistent class type

3.2 Transformation of attributes and associations

- Attributes whose type is a primitive data type (e.g. String, Int) should be transformed to a single column whose type is the same as the primitive data type.
- (...) primary keys (...) should point to the correct model elements – transformations which create duplicate elements with the same names are not considered to provide an adequate solution.

Figure 5 depicts the corresponding triple graph grammar rules. The first rule transforms an **Attribute** into a **Column** with the same name. The **Column** will be added to the **Table** which has been linked to the **Class** that contains the regarded **Attribute** before. The second rule does the same. Additionally, the rule tests if the given **Attribute** has been marked as **primary** and creates a **pkey** link accordingly. The last rule tests whether there is already a **Column** with the same name as the regarded **Attribute**. If this is the case the rule does not create a new **Column**. It just links the **Attribute** to the **Column**. Observe that the given priorities ensure that primary keys point to the correct model elements, and that no duplicate **Columns** are created.

- Attributes whose type is a non-persistent class should be transformed to one or more columns. (...) Note that the primary and foreign keys of the translated non-persistent class need to be merged in appropriately, taking into consideration that the non-persistent class may contain primary and foreign keys from an arbitrary number of translated classes.

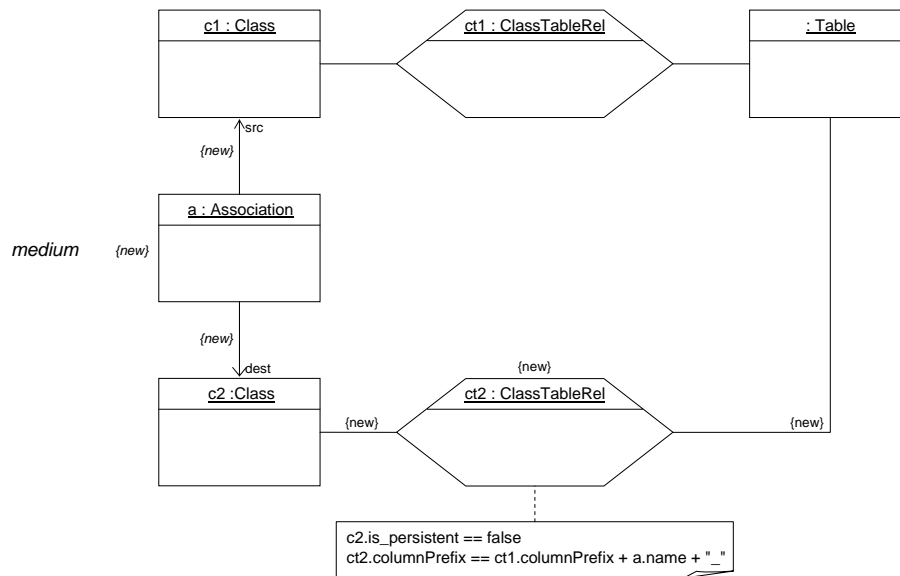


Figure 7: TGG rule concerning associations with non-persistent class destination

- (...) The columns should be named *name_transformed_attr* where *name* is the name of the attribute or association in question, and *transformed_attr* is a transformed attribute, the two being separated by an underscore character. The columns will be placed in tables created from persistent classes.

The corresponding rule is depicted in Figure 6. The rule deals with **Attributes** whose type are non-persistent **Classes**. To this end the rule links the non-persistent **Class** with the **Table** which has been linked to the **Class** containing the **Attribute** before. This ensures that **Columns** resulting from **Attributes** of the non-persistent **Class** will appear in the correct **Tables**. The name of each **Column** will be calculated using the technical attribute `columnPrefix` of the correspondence link class `ClassTableRel`. Observe, that the presented rule can be applied recursively. The rule that deals with **Associations** whose destination is a non-persistent **Class** looks similar and is shown in Figure 7.

- Attributes whose type is a persistent class should be transformed to one or more columns, which should be created from the persistent classes' primary key attributes. The columns should be named *name_transformed_attr* where *name* is the attributes' name. The resultant columns should be marked as constituting a foreign key; the *FKey* element created should refer to the table created from the persistent class.

These requirements are satisfied by the rule shown in Figure 8. The rule links an attribute whose type is a persistent **Class** to a *FKey* element. The dashed shape of the *FKey* element in combination

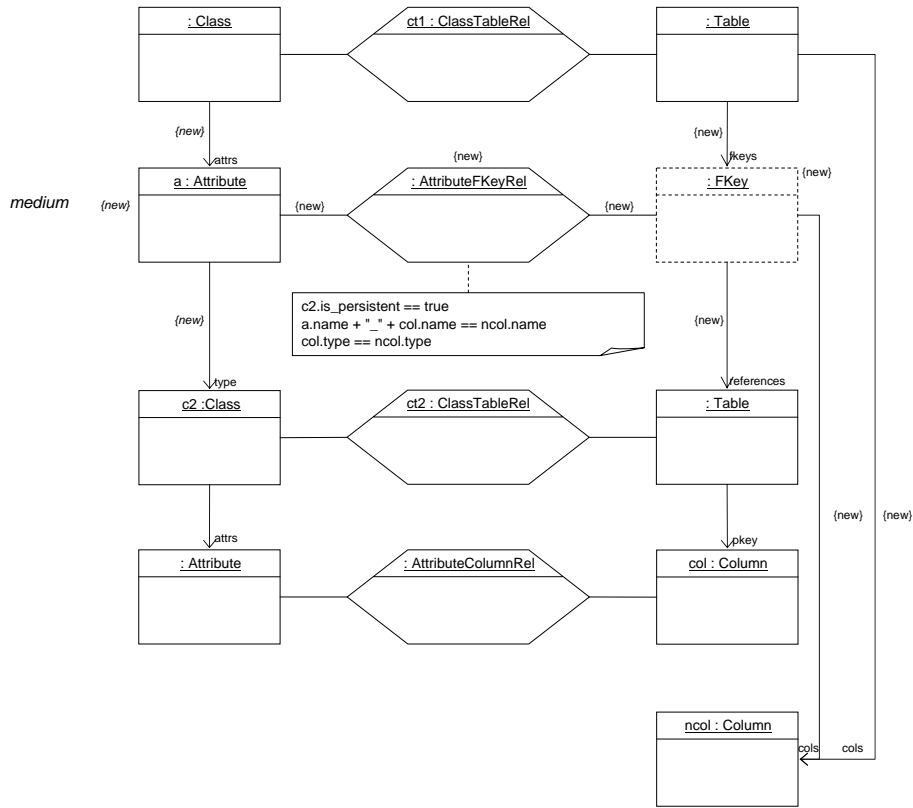


Figure 8: TGG rule concerning attributes with persistent class type

with the `{new}` tag means that this elements will only be created if it does not already exists. Thereby, we ensure that only one `FKKey` element will be created. The `FKKey` element is added as a foreign key to the `Table` which is linked to the `Class` that contains the `Attribute`. The `FKKey` refers to the `Table` which has been created from the persistent `Class` before. Furthermore, the rule adds a new `Column` to the first `Table` for each `Column` which is marked as a primary key in the second `Table`. The created `FKKey` element points to these new `Columns`. Again, the name of the new `Columns` is calculated using the technical attribute `columnPrefix` of the correspondence link class `ClassTableRel1`. Finally, the rule dealing with `Associations` whose destination is a persistent `Class` is depicted in Figure 9.

- When transforming a class, all attributes of its parent classes (which must be recursively calculated), and all associations which have such classes as a *src*, should be considered. (...)

The rules presented do not satisfy this requirement. We have to investigate whether path expressions can help us and come up with a more sophisticated set of rules. One practical but inelegant

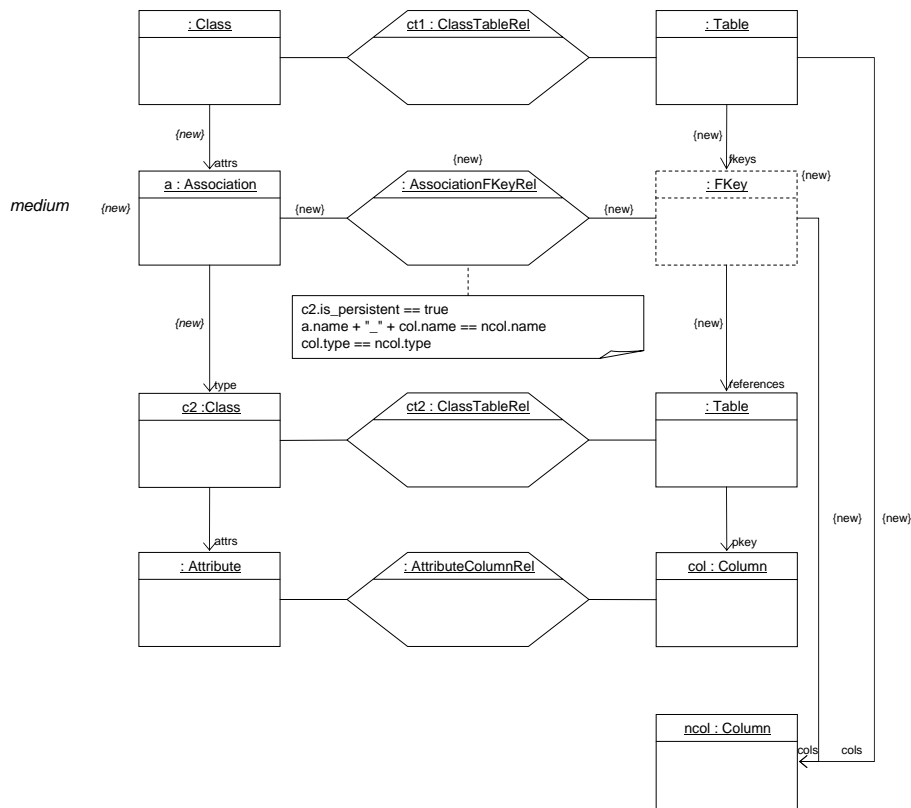


Figure 9: TGG rule concerning associations with persistent class destination

solution is to use some kind of preprocessor which recursively propagates the attributes of a class to its subclasses beforehand - as all other approaches do.

3.3 Example execution

We now demonstrate the application of our approach by transforming a given example input. The to be transformed class diagram is shown in Figure 10a. The rule application mechanism presented in Section 2 starts with an arbitrary model element because there is no given hierarchy by means of a composition relationship. Nevertheless, the application mechanism guarantees that the model elements are transformed in a proper order. Assume that the transformation starts with the `Attribute att1`. The rule application mechanism tries to apply the rule from Figure 5. This rule requires that `Class c1` must be transformed beforehand. Thus, the rule application mechanism postpones the transformation of `Attribute att1`. In the following we transform the class diagram in a proper order. Therefore, we start with the transformation of the `Classes c1, c2, and c3`. For readability reasons we omit

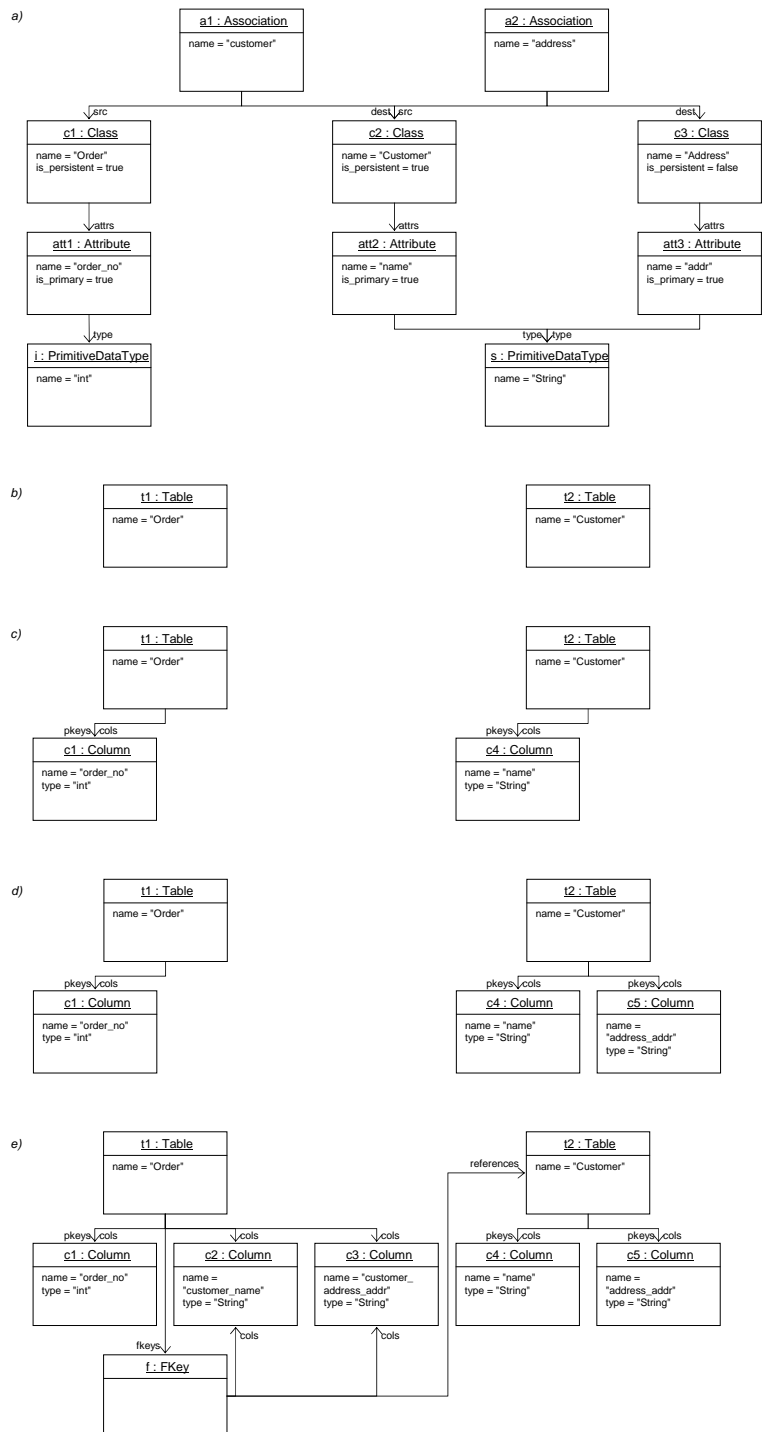


Figure 10: Application of our approach

the correspondence links although they contain important information on the transformation. The rule application mechanism tests whether the second rule from Figure 3 is applicable for the model elements `c1` and `c2`. Since there is no `parent` relationship given in the example this rule does not apply. Therefore, the mechanism test whether the first rule is applicable. Since this is the case the mechanism applies this rule for the model elements `c1` and `c2`. `Class c3` is non-persistent and therefore is not transformed at this level. The current result of the transformation is depicted in Figure 10b. The transformation of the `Attributes att1` and `att2` is done by applying the rule from Figure 5 resulting in the situation described in Figure 10c. By applying the rules from Figure 7 and Figure 5 to the model elements `a2` and `att3` we get the result presented in Figure 10d. The final result from Figure 10e is reached by transforming the `Association a1` by applying the rule from Figure 9.

4 Conclusion

In this paper we have given an overview of our declarative model integration approach focusing on (unidirectional) model transformation. Tackling a common example we have shown that we are able to perform the desired transformation in principle. We point out that we will have to spend more effort in adopting additional concepts from graph grammars to improve expressiveness. Path expressions for instance might allow us to cope with the inheritance of attributes in the presented use case. Furthermore, we have to ensure that improvements to the expressive power do not weaken the declarative nature of triple graph grammars. Finally, we plan to adapt our approach with the upcoming revised *QVT*-submission from the *QVT Merge Group* [QVT05]. This submission is the most promising answer to OMG's *QVT-RFP* and looks similar to our own approach.

References

- [A⁺03] Altheide et al. An Architecture for a Sustainable Tool Integration. In Dörr and Schürr, editors, *TIS 2003 Workshop on Tool Integration in System Development*, pages 29–32, 2003. <http://www.es.tu-darmstadt.de/english/events/tis/documentation/Proceedings.pdf>.
- [AKS03] Agrawal, Karsai, and Shi. Graph Transformations on Domain-Specific Models. Technical report, Institute for Software Integrated Systems at Vanderbilt University, 2003.
- [Bra04] Braun. *Metamodellbasierte Kopplung von Werkzeugen in der Softwareentwicklung*. Logos Verlag, 2004. German, PhD thesis.

- [BW03] Becker and Westfechtel. Incremental Integration Tools for Chemical Engineering: An Industrial Application of Triple Graph Grammars. In *29th Intl. Workshop Graph-Theoretic Concepts in Computer Science*, volume 2880 of *LNCS*, pages 46–57, 2003.
- [CH03] Czarnecki and Helsen. Classification Of Model Transformation Approaches. In *2nd OOP-SLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003. <http://www.softmetaware.com/oopsla2003/czarnecki.pdf>.
- [KS05] Königs and Schürr. Tool Integration with Triple Graph Grammars - A Survey. *Electronic Notes in Theoretical Computer Science*, 2005. submitted for publication.
- [KWB03] Kleppe, Warmer, and Bast. *MDA Explained*. Addison-Wesley, 2003.
- [Mos02] Mosher. *A New Specification for Managing Metadata*. Sun Microsystems, 2002. <http://java.sun.com/developer/technicalArticles/J2EE/JMI/>.
- [NNZ00] Nickel, Niere, and Zündorf. The FUJABA Environment. In *Proc. of the 22nd International Conference on Software Engineering*, pages 742–745, 2000.
- [OMG02] OMG. *Request for Proposal: MOF 2.0 Query/Views/Transformations RFP*, 2002. <http://www.omg.org/cgi-bin/doc?ad/2002-04-10>.
- [OMG03] OMG. *MOF 2.0 Specification*, 2003. <http://www.omg.org/cgi-bin/doc?ad/2003-04-07>.
- [QVT03] QVT Partners. *Initial submission for MOF 2.0 Query/View/Transformation RFP*, 2003. <http://qvtp.org/downloads/1.0/qvtpartners1.0.pdf>.
- [QVT05] QVT Merge Group. *Revised submission for MOF 2.0 Query/View/Transformation RFP (ad/2002-04-10)*, 2005. <http://www.omg.org/cgi-bin/doc?ad/2005-03-02>.
- [Rea05] Real-Time Systems Lab, Darmstadt University of Technology. *MOFLON*, 2005. <http://www.mofflon.org>.
- [Sch94] Schürr. Specification of graph translators with triple graph grammars. In Mayr and Schmidt, editors, *Proc. WG'94 Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163, 1994.
- [W3C03] W3C. *SOAP Version 1.2 Specification*, 2003. <http://www.w3.org/TR/soap12/>.
- [Wag01] Wagner. Realisierung eines diagrammübergreifenden Konsistenzmanagement-Systems für UML-Spezifikationen. Master's thesis, Universität Paderborn, 2001. German.