Model Transformation in the Large

Felix Klar, Alexander Königs, Andy Schürr
Technische Universität Darmstadt
Real-Time Systems Lab
Merckstr. 25
D-64283 Darmstadt, Germany
[klar|koenigs|schuerr]@es.tu-darmstadt.de

ABSTRACT

Current rule-based model transformation approaches as the Query / View / Transformation (QVT) standard or Triple Graph Grammars (TGGs) disregard means for structuring model transformation specifications. As a result large scale model transformation specifications are hard to understand and to maintain. Furthermore, these specifications cannot utilize reusing mechanisms which would reduce the size of the specifications and improve their readability. In this paper we discuss how to transfer means for structuring huge metamodels and models as provided by common modeling languages to the world of model transformation languages. We focus on generalization issues as well as on package dependencies. As a result we come up with an extension to our TGG approach that enables the user to specify structured bidirectional model transformations in a declarative way.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features

General Terms: Languages

1. INTRODUCTION

Nowadays, the size of software systems increases dramatically. In order to develop and maintain such systems efficiently developers need appropriate modeling languages that allow for the modularization of the systems into smaller and more manageable subsystems as well as possibilities to reuse and refine already existing software components. To this end the current version of the infrastructure of the Unified Modeling Language (UML) [20] provides support for modularization by means of sophisticated model package dependencies and generalization concepts to allow for the specification of reusable and refineable software components. Moreover, approaches like the Model Driven Architecture (MDA®) [8] of the OMG envision to lift the development task to higher levels of abstractions from which the desired implementations can be generated automatically.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.
Copyright 2007 ACM 978-1-59593-811-4/07/0009 ...$5.00.

In order to specify model transformation that translate more abstract models to more specific models the OMG asked in 2002 for proposals for a language that supports queries and transformations on models as well as creating views on metamodels [16]. Currently, the finalization of such a Query / View / Transformation (QVT) [18] language is pending and will presumably be adopted this year. QVT suffers from the fact that its semantics is only informally given. Additionally, most common model transformation approaches in general and QVT in particular disregard means for the modularization and reusability of their model transformation specifications. Since such model transformation specifications themselves tend to be huge as well they are hard to develop, maintain, and understand. This problem currently is recognized and addressed by workshops like GaMMa 2006 (1st International Workshop on Global Integrated Model Management)[1] and approaches like [12]. These approaches introduce the term *Megamodeling* for all kinds of activities dealing with problems that arise when the MDA approach is used in really large projects, where thousands of models, metamodels, and model transformation specifications are under development. Right now Megamodeling activities mainly address the development of "x-in-the-large" concepts for models and metamodels, whereas most publications still neglect the needs for modularization, composition, specialization, parametrization, and reuse concepts for model transformations.

Therefore, this paper makes a first proposal how to adapt package import and refinement (merge) concepts originally developed for UML models and MOF metamodels to create reusable and refineable model transformations. These "model transformation in the large concepts" are added to our Triple Graph Grammar (TGG) approach. TGGs combine the world of precisely defined graph transformations with a QVT-like functionality for the declarative visual definition of bidirectional model transformations. Thereby, the term *schema* from the world of graphs corresponds to the term *metamodel* from the world of metamodeling. Moreover, the term *graph* matches the term *model*. *Node types* and *edge types* correspond to *classes* and *associations*. Finally, *nodes* and *edges* match *objects* and *links*. Throughout this paper we will use these terms interchangeably according to the given context.

Due to lack of space we can only rely on the tedious toy example of integrating class diagrams with database schemas as a case study. Nevertheless, this case study allows us to sufficiently demonstrate the usage of our proposed concepts.

[1] `http://planetmde.org/gamma2006/`

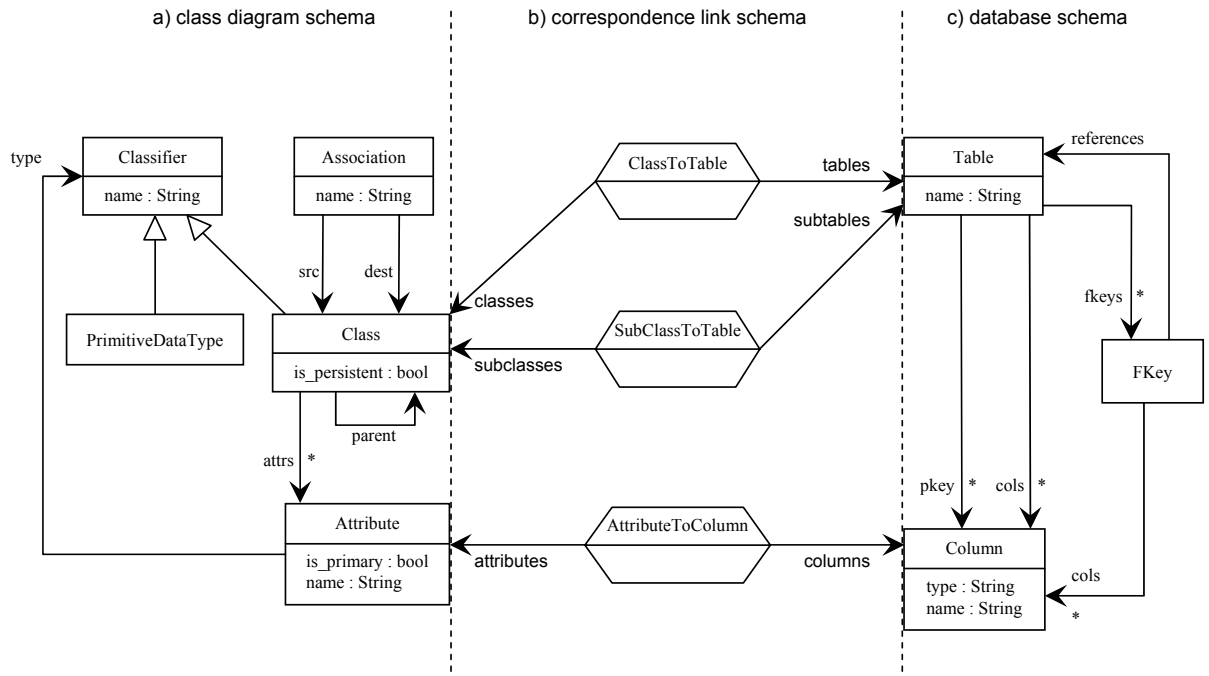

Figure 1: Example of a TGG schema

However, various real-world case studies (e.g. integrating textual use case specifications in the requirements engineering tool Doors[2] from Telelogic with use case diagrams in Enterprise Architect[3] from Sparx Systems) from our industrial partners prove the need for dealing with large scale model transformation specifications. Moreover, in the ToolNet project [1] in cooperation with DaimlerChrysler we are facing the task of integrating a number of COTS (commercial off the shelf) tools for embedded system development. Due to their size the metamodels of these tools as well as the specification of the needed model transformation rules require means for modularization and reuse.

The remaining paper is structured as follows. In Section 2 we briefly introduce our model integration approach called Triple Graph Grammars (TGGs) and provide a running example that deals with the integration of class diagrams with database schemas. After that we discuss means for modularizing model transformations at package level in Section 3. Means for supporting reuse and refinement of model transformations by generalization is described in Section 4. Section 5 illustrates the integration of the proposed features into our existing TGG implementation as part of the meta-CASE tool MOFLON[4]. We compare our approach with related ones in Section 6. Finally, Section 7 concludes this paper and discusses open issues as well as future work.

2. TRIPLE GRAPH GRAMMARS

Triple Graph Grammars (TGGs) have been introduced in 1994 by Schürr [22]. TGGs as an extension of Pratt's pair grammar approach from 1971 [21] aim at the declarative specification of model to model integration rules. Pratt's approach implicitly couples two (graph or string) grammars with each other in order to express simultaneous application of grammar rules. The application of such a pair grammar results in two graphs or strings that are said to be consistent to each other. In addition TGGs explicitly maintain the correspondence of two graphs by means of correspondence links. These correspondence links play the role of traceability links that map elements of one graph to elements of the other graph and vice versa. Furthermore, each correspondence link may carry additional information calculated during rule application. As a running example we consider the integration of a class diagram with a corresponding database schema. This example is described in [18] in an extended version and, thus, can be considered as a benchmark for model integration approaches.

TGGs consist of a schema and a set of graph rewriting rules. The schema declares the node and edge types of the integrated graphs as well as of the correspondence graph. In other words, a TGG schema consists of a pair of metamodels of integrated models and the types of correspondence links between them. Furthermore, each correspondence link type owns a TGG rule. The TGG rules declaratively specify the simultaneous evolution of the integrated graphs and the creation of the correspondence links. For a detailed description of the formal semantics of TGGs the reader is referred to [10].

Figure 1 gives an example of a TGG schema. The left-hand side of the figure depicts a simplified graph schema (or metamodel) of class diagrams. Basically, class diagrams consist of *Classes* that are related to each other by *Associations*. Furthermore, *Classes* can inherit from each other by means of the *parent* relationship. *Classes* may own *Attributes*. *Attributes* carry a *type* which is a *Classifier* (i.e., a

[2] `http://www.telelogic.de/products/doors/index.cfm`
[3] `http://www.sparxsystems.com/products/ea.html`
[4] `http://www.moflon.org`

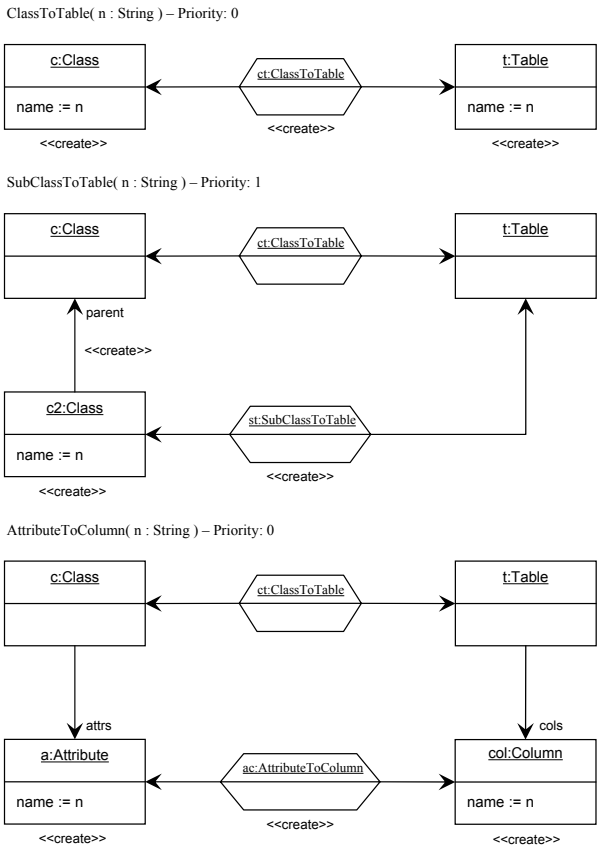

Figure 2: Examples of TGG rules

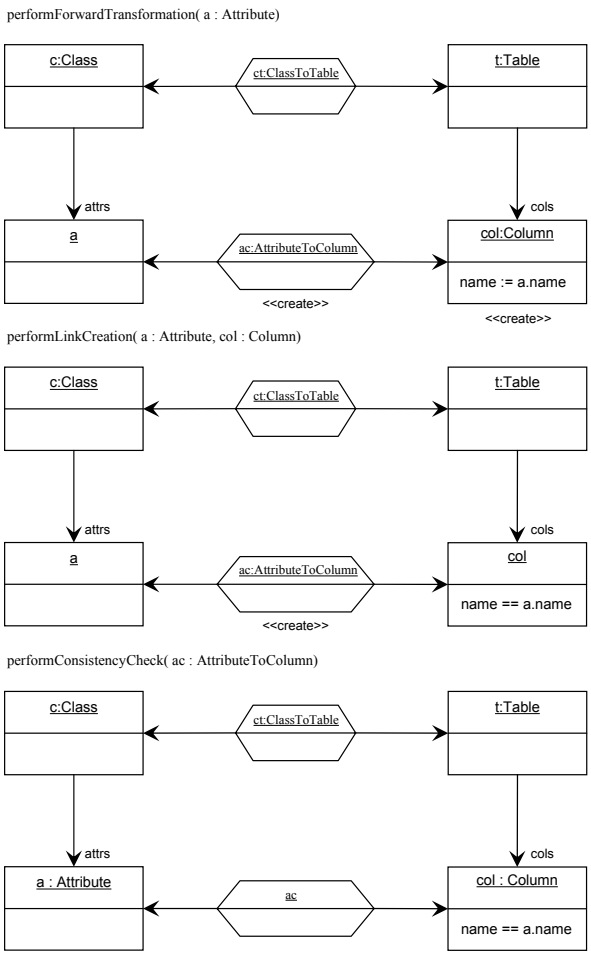

Figure 3: Examples of operational rules

PrimitiveDatatype or a *Class*). Correspondingly, the right-hand side of the figure shows a simplified graph schema of database schemas. A database schema consists of *Tables* that own *Columns*. Some *Columns* may be marked as the *primary keys* of the owning *Table*. Furthermore, a *Table* may have *Foreign Keys (FKeys)* that *refer* to *Columns* of a different *Table*. Finally, the center of Figure 1 introduces the correspondence link types used in our example. Basically, *Classes* from class diagrams correspond to *Tables* in database schemas. We keep track of these correspondences by links of type *ClassToTable*. A special case are *Classes* that inherit from a different *Class*. We link these *Classes* to *Tables* using the link type *SubClassToTable*. Furthermore, *Attributes* from class diagrams correspond to *Columns* in database schemas. We represent these correspondences by links of type *AttributeToColumn*. A more complete discussion of this running example can be found in [9].

Besides a schema a TGG provides a set of TGG rules. In our approach each correspondence link type is provided with at most one TGG rule. Figure 2 shows some examples of such rules. Each rule has a name and may be provided with an arbitrary number of parameters which may be used in the body of the rule. Furthermore, TGG rules additionally carry priorities.

Priorities are used to resolve conflicts that occur in the case that more than one rule is applicable at a certain point of model integration. If multiple rules are applicable only the rules with the highest priority are applied. That means that during model integration the rule *ClassToTable* will only be applied to a given *Class* or *Table* object if the rule *SubClassToTable* is not applicable.

The rule *ClassToTable* simultaneously creates[5] a *Class* and a corresponding *Table*. The *name* attributes of both objects are assigned to the parameter *n* of the rule. Finally, both objects are linked by a new correspondence link of type *ClassToTable*. The rule *SubClassToTable* creates a new *Class c2* and links it to a *parent Class*. Furthermore, the rule links *c2* to the same *Table* to which its *parent Class c* already is linked to. Observe that the rule requires *c* to be linked to a *Table* by a link of type *ClassToTable*. Since the link type *SubClassToTable* currently does not inherit from the link type *ClassToTable* the *parent Class c* cannot be a *Class* that itself has a *parent Class*. In common TGG approaches we, therefore, need an additional rule that exactly looks like the rule *SubClassToTable* and replaces the link type *ClassToTable* by *SubClassToTable*. In Section 4 we introduce generalization on link types which allows for a more elegant specification of TGG rules. Finally, rule *AttributeToColumn* simultaneously adds a new *Attribute* to an already existing *Class* and a new *Column* to an already ex-

[5]Elements that are created by a rule are annotated with `create` in contrast to elements that are matched.

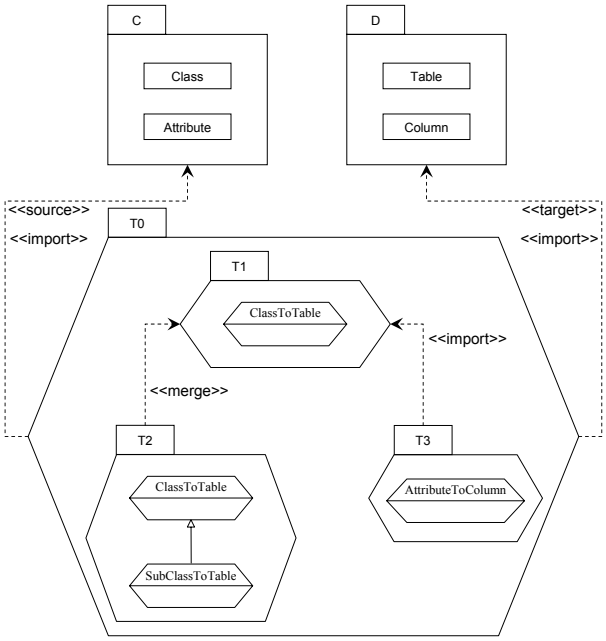

Figure 4: Package Dependency Example

isting *Table* that has been linked to the *Class* beforehand.

Usually, TGG rules are not intended to be applied as they are. The simultaneous evolution of two graphs is rarely useful in practice. Rather, one graph is given and the user wants to transform it into a corresponding second graph. Another scenario is that two graphs are given and the user wants to calculate the correspondences between both graphs. To this end TGG rules are regarded as merely a declarative specification from which a number of operational graph rewriting rules will be derived from. Figure 3 gives some examples of such derived rules. For a more complete set of possible rule derivations the reader is referred to [10]. Rule *performForwardTransformation* transforms a given *Attribute a* to a corresponding *Column* and creates a correspondence link. The *Column* will be attached to *Table* which corresponds to the *Class* that owns *a*. Rule *performLinkCreation* links a given *Attribute* with a given *Column*. The rule tests whether the given *Column* is attached to the correct *Table* and has the correct *name*. Finally, rule *performLinkCreation* checks whether or not a given *AttributeToColumnLink* still is consistent.

3. PACKAGE DEPENDENCIES

When specifying large models and transformations between these models, a packaging concept that allows for structuring and reuse of existing elements in a model transformation language is crucial. Applying these concepts one may create rule libraries and reuse existing transformation rules. The OMG has standardized a packaging concept in the UML 2.0 Infrastructure Library [20]. In the following we will adapt the concepts *package import* and *package merge* described in this standard to our TGG approach and shortly discuss *nested namespaces*. These concepts will be explained on the basis of our running example. Therefore, we take the TGG schema shown in Figure 1 and create a structured TGG schema (Figure 4) from it using the concepts introduced in this chapter. As packing concepts start to make sense when dealing with larger specifications, it is obvious that in our small example the application of these concepts is of conceptual nature. For purposes of clarity Figure 4 does not contain references between correspondence link types and referenced classes (see Figure 1 for more details).

3.1 Nested Namespaces

Regarding the visibility of elements in nested namespaces (TGG packages and integration link types are namespaces) we adapt the semantics defined in [20]:

1. *TGG elements defined in an enclosing namespace are available using their unqualified names in the enclosed namespaces.*

This means, that all elements contained in a TGG package will be visible to correspondence link types and rules contained in a nested TGG package. As correspondence link types are namespaces, rules that are declared in a link type will see everything that is visible to the link type.

3.2 Package Import

Package import makes all elements of an imported package visible in the namespace of the importing package. So it is possible to reference elements of the imported package without using the full qualified name of elements of the imported package.

In our TGG approach package imports may be defined between TGG packages and packages of the integrated metamodels, as well as between two different TGG packages. Importing packages from one of the integrated metamodels and of the imported TGG packages respectively makes elements of these namespaces visible in the TGG package. Visible classes from one of the integrated metamodels may be used at schema level to create correspondence link types between classes of the source and target metamodels. At rule level those classes may be used to define how source and target model are simultaneously changed and how correspondence links are established between instances of these classes.

At schema level, correspondence link types that become visible as a result of a package import may be used to establish generalization dependencies between link types that are owned by the package in which the link types are visible and the visible link types (see Section 4 for the description of generalization dependencies between link types). Creation of generalization dependencies between link types that become visible due to a package import and link types of the importing package are forbidden, as well as the creation of generalization dependencies between two link types that become visible due to a package import. At TGG rule level imported link types may be used as context element in rules of link types contained in the importing package. Whereas creation of correspondence links of imported correspondence link types in a TGG rule is forbidden. As shown in Figure 2, rule *AttributeToColumn* uses link type *ClassToTable* as context element. Thus the package import between T3 and T1 is necessary (cf. Figure 4).

[20] states that package import has either visibility of "public" or "private". In our example a "public" package import is defined between T3 and T1 ("private" imports would be denoted by an <<access>> label). Visibility determines whether imported elements will be visible to packages that

use the importing package as imported package. If the visibility of a package import is "public" the elements will be visible to the importing package and packages that import the importing package. If it is "private" they will only be visible to the importing package but not to other packages that import the importing package. Looking at our example, a package that would import T3 would also see elements contained in T1, because of the transitive nature of the "public" import between T3 and T1. In our extended TGG approach we offer both visibility options, as this gives more flexibility specifiying a TGG.

We come up with the following statements, that are directly adopted from [20]:

2. *A package import may be defined between a TGG package and a package of the integrated source or target metamodel.*

3. *A package import may be defined between a TGG package and another TGG package.*

4. *A package import may have visibility "public" or "private".*

In Figure 4 all elements contained in Packages C and D will be visible to T0, because of the package imports defined between T0 and C and D. In conjunction with the statement of Section 3.1 all elements contained in Packages C and D will also be visible to T1, T2 and T3, because they are enclosed namespaces of T0 (T0 is the enclosing namespace). The package import between TGG packages T3 and T1 is required, because *AttributeToColumn* uses *ClassToTable* in its TGG rule (see Figure 2). Without the package import *AttributeToColumn* would only see T1 but not the elements contained in T1.

3.3 Package Merge

A *package merge* is defined between a merged package and a receiving package. The package merge allows to extend elements of the merged package in the receiving package. In our TGG approach we restrict the merged package and the receiving package to be TGG packages and not packages from the integrated metamodels. Additionally, we use a slightly different semantics of package merge in contrast to the semantics defined in [20].

In our approach we use the semantics of an older version of the UML Infrastructure (cf. [17]), where the resulting package is constructed as follows (see Figure 5):

- the package merge is resolved to a package import
- a generalization dependency is constructed between matching correspondence link types of the merged and the receiving package
- two correspondence link types match, if their name is equal
- a package merge is invalid, if there are matching correspondence link types for which the merged link type is invalid as the parent of the receiving link type according to the rules defined in Section 4

In our example shown in Figure 4 the package merge produces a generalization dependency that is established between *T2::ClassToTable* and *T1::ClassToTable*.

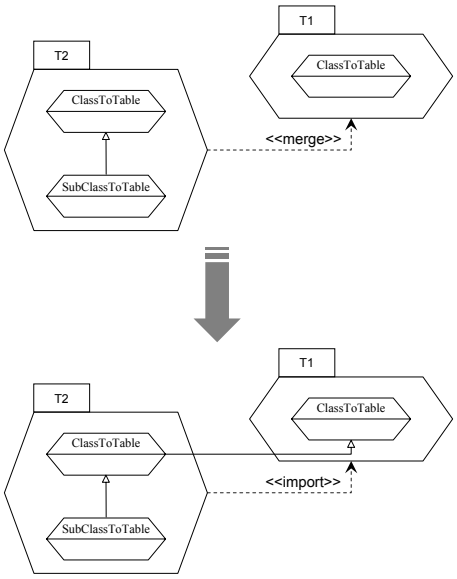

Figure 5: Merge semantic by example

If we would adopt the new package merge semantics described in [20], we would have a disadvantage which will be explained using our example (Figure 4). With the new package merge semantics defined in [20] *T1::ClassToTable* and *T2::ClassToTable* would be merged into the resulting correspondence link type *T1::ClassToTable@T2::ClassToTable* without having a generalization dependency between the resulting link type and *T1::ClassToTable*. An integration rule (*AttributeToColumn(n:String)*) that uses an integration rule type from a merged package (*T1::ClassToTable*) would not operate as desired, if the correspondence link type is merged into another package (*T2*). The resulting correspondence link type (*ClassToTable* in the resulting package) would reside in the resulting package (*T2*), but there would be no generalization dependency between the resulting correspondence link type and the merged link type (*ClassToTable* in package T1). So the integration rule *AttributeToColumn(n:String) (Figure 2)* would never match instances of *ClassToTable* of the resulting package as it expects instances of *ClassToTable* of the merged package T1.

We demand for package merge in the TGG schema:

5. *A package merge may only be created between TGG packages.*

4. GENERALIZATION

According to [13] *Generalization* provides another means for reusability in (object-oriented) software development. In this section we transfer that concept to the area of model transformation in general and our Triple Graph Grammar approach in particular. To this end we have to investigate in which way *Generalization* affects TGGs at schema level as well as at rule level.

4.1 Schema level

At schema level TGGs basically declare types of correspondence links between types of the integrated metamodels. Therefore, we have to investigate how to transfer the

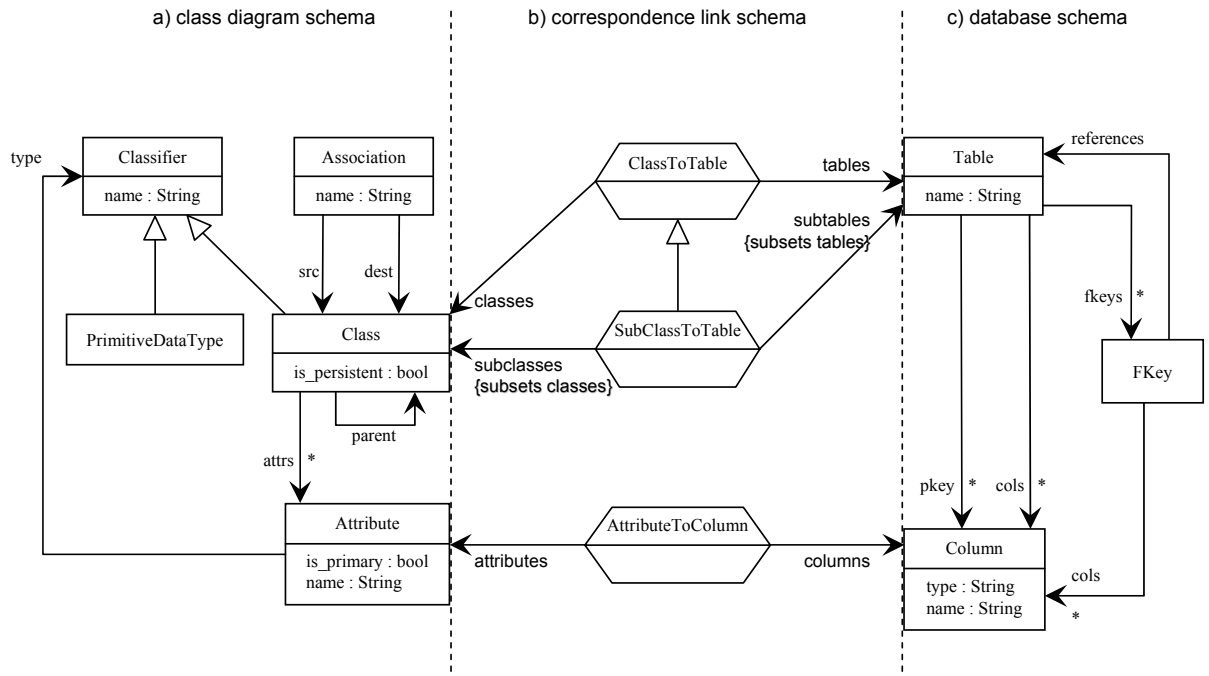

Figure 6: Using generalization on link types

concept of generalization to link types. Conceptually, in our approach a link type is a combination of a class and an association as defined by [20]. Classes as well as associations are classifiers. Classifiers may inherit from each other by means of generalization. A generalization is owned by the more specific classifier and references the more general classifier. [20] demands that the classifiers referred to by the ends of a more specific association must be more specific than or equal to the classifiers referred to by the ends of the more general association. Furthermore, this constraint is also postulated by [13] which states that a more specific element must refine the properties of a more general element. Consequently, we demand this constraint for link types in our approach as well:

6. *Classifiers referred to by the ends of a more specific link type must be more specific (or the same) than the classifiers referred to by the ends of the more general link type.*

Additionally, [20] allows to mark association ends as *subsets*, *union*, or *redefines* with roughly the following semantics. The set of instances linked by an association end marked as *subsets* of a more specific association is a subset of the set of instances linked by the subsetted association end of the more general association. The set of instances linked by an association end marked as *union* of a more general association is entirely composed of the sets of all subsetting association ends of all more specific associations, i.e. the more general association is merely abstract. An association end of a more specific association marked as *redefines* restricts the type of linked elements to a more specific type than the corresponding type of the redefined association end of the more general association. We transfer these markings to our link types as well:

7. *The ends of link types in our TGG approach may be marked as subsets, union, or redefines using the same semantics as provided by [20].*

8. *Consequently, link types in our TGG approach may be marked as abstract. An abstract link type still may provide a TGG rule but cannot be instantiated directly. (i.e., rules will never be applied, but may only be inherited and extended)*

Moreover, using generalization as well as the markings of association ends has the following consequences:

9. *Correspondence link types constitute relations. A more specialized link type constitutes a subrelation of the more general link type. Thus, the ends of a more specialized link type must at least subset or redefine the ends of the more general link type.*

10. *The priority of a more specialized rule must be higher than the priority of the more general rule. Otherwise the more specialized rule will never be applied as we will see in the next subsection.*

11. *According to [20] redefinition means that the element of a more specialized type at the redefined end of a more specialized association may only be linked by this association rather than by the more general association. Therefore, in our approach the redefinition of link type ends prevents the more general rule for being applied if the more specialized rule fails to be applied.*

Finally, [20] allows to provide *multiplicities* for association ends. The multiplicity of an association end determines how many instances of the classifier referred to by the regarded association end has to be linked to an instance of the classifier referred to by the opposite association end. The lower

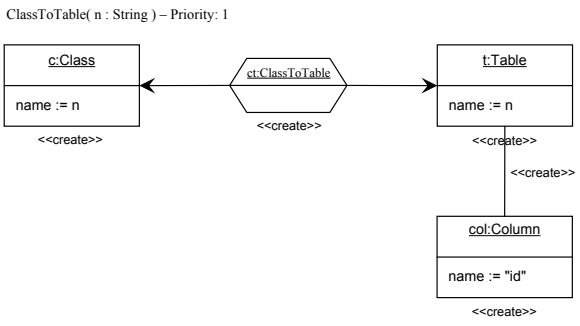

Figure 7: Refined TGG rule *ClassToTable*

bound of a more specific association end must be greater or equal than the corresponding association end of the more general association. The upper bound of a more specific association end must be less or equal than the corresponding association end of the more general association. Therefore, we demand for TGG link types:

12. *Ends of TGG link types may be provided with multiplicities as defined for associations in [20].*

13. *Multiplicities of TGG link type ends must adhere to the same constraints as multiplicities of association ends in [20].*

Using our proposed extensions to TGG schemas we can now come up with a more sophisticated schema (cf. Figure 6) concerning our running example of integrating class diagrams with database schemas from Section 2. This time link type *SubClassToTable* inherits from link type *ClassToTable*. Consequently, the link end *subclasses* from *SubClassToTable* subsets the link end *classes* from link type *ClassToTable*. Accordingly, the link end *subtables* subsets the link end *tables*. This means that every correspondence link that will be established between a *Class* that has a *parentClass* and a *Table* will automatically be propagated to the set of links between any *Classes* and *Tables*. We will realize the resulting benefit at rule level.

4.2 Rule level

We now investigate to which extent generalizations specified between correspondence link types in the schema have consequences for their underlying TGG rules. Basically, correspondence link types similar to QVT constitute relations (i.e., sets of links between model elements). Each TGG rule declaratively specifies the creation of new links of the link type the rule is attached to. Since a generalization usually means that a member of a more specialized type also is a member of the more general type the set of links of a more specialized link type should be a subset of the links of the more general link type. In order to guarantee this, we must ensure that each time a TGG rule of a more specialized type is applicable, the TGG rule of the more general link type would be applicable as well. Therefore, we demand that:

14. *The TGG rule of the more specialized link type basically contains a copy of the TGG rule of the more general link type.*

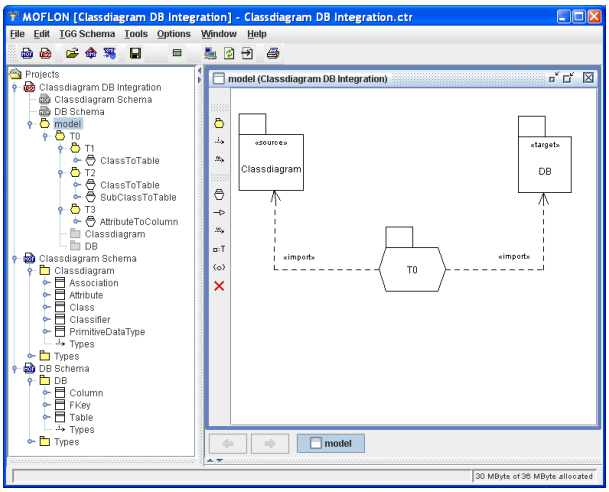

Figure 8: Screenshot of the TGG-Editor

15. *The user may now replace types of elements on the left-hand side of the rule by specialized ones (i.e., subclasses for nodes, subsets or redefines on association ends).*

16. *The user may choose to move elements from the right-hand side of the rule to the left-hand side of the rule. Thereby, the more specific rule requires the existence of graph elements which would have been created by the more general rule.*

17. *The user may add entirely new elements to the left-hand side as well as to the right-hand side.*

18. *The user may add further attribute conditions and assignments.*

As a consequence, a more general rule is always applicable when the more specific rule can be applied and it always creates a subset of the graph elements that do exist after the more specific rule has been applied; i.e., the observable effects of a more specific rule always imply the observable effects of the more general rule. A formal proof of these properties has to be omitted here due to lack of space.

Concerning our running example rule *SubClassToTable* inheriting from rule *ClassToTable* from Figure 2 satisfies our demands. Moreover, rule *SubClassToTable* is applicable if *c* itself is a *Class* that has a *parent Class* which has been linked to *Table t* by a link of type *SubClassToTable* which conforms to *ClassToTable* beforehand.

The rule *ClassToTable* from the merging package *T2* (cf. Figure 4) inherits from rule *ClassToTable* from the merged package *T1*. This rule might look as depicted in Figure 7. The rule basically looks like the inherited rule from package *T1*. Besides the *Table* this rule additionally creates a *Column* in order to hold *ids* for each row of the *Table*.

5. IMPLEMENTATION

Currently, we are implementing the proposed concepts as presented in this paper as part of our MOFLON TGG-Editor plug-in (cf. Figure 8). The architecture of our TGG-Editor is shown in Figure 9. Bascially, the TGG-Editor consists of

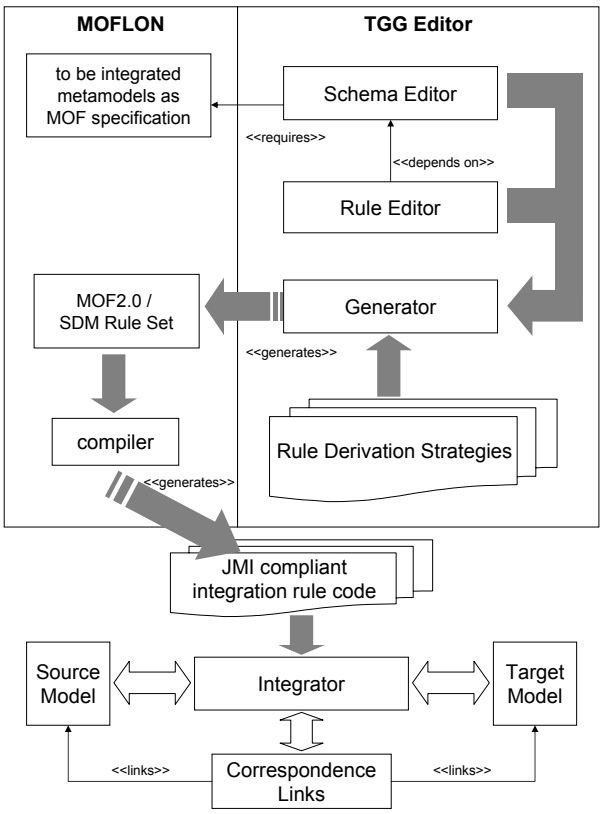

Figure 9: Architecture of the TGG-Editor

a TGG schema editor, a TGG rule editor, and a generator module. The TGG schema editor allows for the specification of a package hierarchy that contains the declaration of correspondence link types. To this end the TGG-Editor refers to two MOFLON MOF 2.0 projects that constitute the metamodels of the integrated models. These metamodels can be specified by using MOFLON's MOF 2.0-Editor plug-in or can be imported from external tools (e.g. Rational Rose, Magic Draw) using the XMI file format. The TGG rules that are attached to the declared correspondence link types can be specified by using the TGG rule editor. From the declarative TGG model integration specification the generator module automatically generates a plain MOFLON MOF 2.0 project by converting the TGG schema into a MOF 2.0-compliant [19] metamodel (cf. Figure 10). The generator maps TGG packages to MOF packages. Import and merge relationships between TGG packages are mapped to import and merge relationships between MOF packages. TGG link types are mapped to MOF classes. Generalizations between link types are mapped to generalizations between MOF classes. Finally, operational graph rewriting rules are derived from the declarative TGG rules as explained in Section 2 and attached to the mapping of the owning TGG link type. From the resulting MOFLON MOF 2.0 project we can automatically generate JMI-compliant[6] Java code using MOFLON's code generation facility.

Finally, the generated Java code can be executed by our

[6]http://java.sun.com/products/jmi/

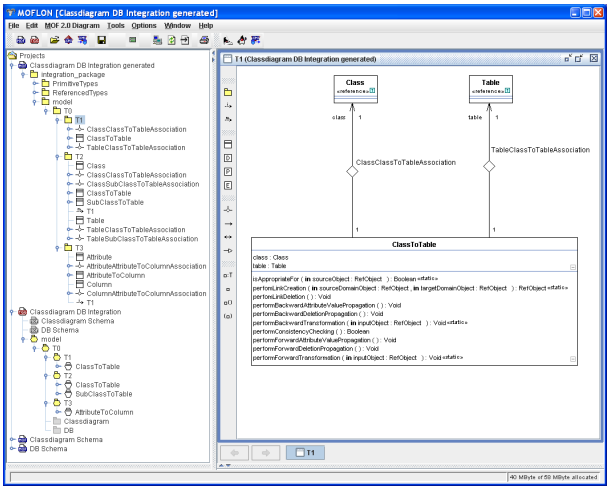

Figure 10: Generated MOFLON project

Integrator[7]. The Integrator is an application which is independent from MOFLON. Furthermore, the Integrator is independent from the metamodels of the integrated models and the generated model integration Java code. Rather, the Integrator relies on the reflective interfaces provided by the JMI standard. In order to perform a model integration the user of the Integrator must provide the metamodels of the integrated models as well as the metamodel of the correspondence link types. After that, the user must specify the integrated models. The user can choose to access models by means of JMI-compliant tool adapters that directly access models residing in tools through JMI-compliant interfaces or to provide XMI files that contain the models. Finally, the user must choose which model integration task (e.g. Forward Transformation, Consistency Checking, and so on) should be performed.

6. RELATED WORK

In this section we compare our extended TGG approach with current model transformation approaches. Thereby, we focus on the support for modularization and reuseability. As we will see most approaches spend little regard to and support for these issues.

OMG's Query / View / Transformation (QVT) [18] standard has many similarities to TGGs in general [5] and our approach in particular. As our approach does, QVT complies to OMG's MOF 2.0 [19] standard as demanded by OMG's corresponding request for proposal [16]. In fact the metamodel of QVT is based on a part of MOF 2.0 called Essential MOF (EMOF). In EMOF there is no concept *Association*. Consequently, QVT does not support *subsets*, *redefines*, and *union* for its transformations. Furthermore, in EMOF packages can only be nested but not be *imported* or *merged*. One basic concept of QVT is *Transformation*. A transformation defines the mapping between two models. To this end a transformation owns a set *Rules*. Furthermore, a transformation syntactically inherits from *Class* and *Package* as defined in EMOF. As a package, a transformation

[7]http://gforge.echtzeitsysteme.org/projects/integrator/

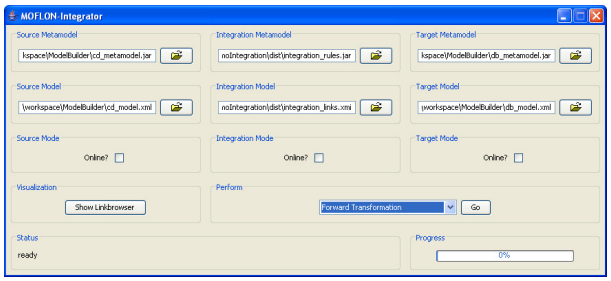

Figure 11: Screenshot of the Integrator

provides a namespace for its rules. As a class, a transformation allows for the definition of configuration values and utility functions. A transformation can be extended by another transformation. Thereby, the rules of the extended transformation are transitively included in the extending transformation. Since a transformation inherits from package it is possible to nest transformations in each other. To the best of our knowledge [18] disregards this possibility and provides no semantics for that. A rule in QVT roughly corresponds to a rule in our TGG approach. In QVT a rule may override other rules if certain overriding constraints are satisfied. To conclude the QVT standard provides some limited not very well-specified means for reuseability and modularization.

[12] discusses rule-based modularization support in model transformation languages in general and the ATLAS Transformation Language (ATL) [7] in particular. Although this paper identifies and addresses the same issues than we do the presented approach proposes an entirely different solution. In order to support reuseable model transformation specifications [12] demands to tweak the metamodels of the integrated models in a way that allows for the application of more generic model transformations. The authors argue that commonly metamodels do not properly separate different dimensions of concerns from each other. Therefore, model transformations tend to mix up these different dimensions, too. [12] concludes that this fact hinders the reuse of model transformation rules and demands the decomposition of the metamodels as well as the model transformation rules along the different dimensions of concerns. Although we could tweak the metamodels of integrated models as proposed as well we usually assume that source and target metamodels are immutable (e.g. metamodels of commercial-off-the-shelf (COTS) tools).

According to [23] the Programming with Graph Rewriting System (PROGRES) provides a package concept for modularizing graph rewriting specifications. The package concept supports nesting of packages as well as import dependencies. An import dependency enables the importing package to refer to any visible element from the imported package. Thereby, an import dependency may reduce the visibility of the imported elements. Additionally, PROGRES defines generalization on packages which is required in PROGRES when a class from one package inherits from a class of another package. PROGRES does not support more sophisticated package concepts as *package merge*. Moreover, PROGRES supports generalization on graph rewriting rules that represent methods of node classes and types [14]. PROGRES demands that a graph rewriting rule of a more specialized class or type that overrides a graph rewriting rule of a more general class must maintain the parameter type profile of the overridden rule but may change parameter names. There are no further restrictions concerning the shape of the specified graph transformations.

The TGG approach [4] as implemented as a plug-in for the FUJABA toolsuite [24] allows for the specification of a dedicated TGG schema and a set of declarative TGG rules as our approach does. On the one hand FUJABA relies on version 1.3 of the UML [15]. This version does not provide the sophisticated *merge* package dependency. On the other hand FUJABA itself provides little support for the specification of package hierarchies at all. Furthermore, FUJABA's TGG schema editor is in fact FUJABA's class diagram editor and does not properly separate the integrated metamodels as well as the link type metamodel from each other. The editor allows for the specification of multiplicities for link types as we do but does not support concepts like *subsets*,*union*, and *redefines* since they are not included in [15]. The TGG rules in FUJABA are not attached to the link types declared in the TGG schema. Rather, TGG rules exist more or less separately from the link types and, thus, do not regard inheritance on link types by means of the shape of the specified graph transformations. The same drawbacks apply to the TGG approach as implemented in the IMPROVE project [2].

Finally, AToM^3 [3] also relies on an older version of UML and, thus, does not provide *merge*, *subsets*, *union*, and *redefines*. Again, the graph rewriting rules in AToM^3 are not attached to link types and are, therefore, not subject to generalization issues. There are lots of further tools dealing with model transformation each of which providing only limited if any support for modularization and reuseability. To the best of our knowledge no related approach provides both sophisticated modularization support by means of package dependencies as defined by [20] and reuseability and refinement support by means of generalization.

7. CONCLUSION

In this paper we have motivated the needs to lift modularization and refinement concepts developed for (meta)models to model transformations. We have emphasized that most common model integration approaches currently disregard these features and provide little to no support. In particular we have discussed how to transfer package dependencies and generalization concepts as defined by the UML infrastructure to our TGG approach which is implemented as part of the MOFLON meta-CASE tool. It would be possible for the QVT standard to easily adopt our proposed concepts as well. The price is that the QVT standard which currently relies on EMOF, consequently would have to switch to complete MOF (CMOF).

Currently, our approach supports only single inheritance on correspondence link types. To improve the expressiveness of our approach we plan to adopt multiple inheritance on link types as well. Since this will require sophisticated analysis and merging algorithms for TGG rules of the more general link types, this is ongoing work.

The creation of views on metamodels would allow to specify more readable and maintainable model integration specifications. Furthermore, the QVT-RFP of the OMG demands the possibility for view creation as well. The current QVT standard intentionally disregards this demand. Therefore, we plan to provide means for view creation on metamodels.

We have already presented ideas for the specification of updateable model views with a specific variant of TGGs that avoid the materialization of model views in favor of translating transformations on views into transformations on the underlying models directly in [6].

Furthermore, we are interested in the development of new model transformation composition concepts that lift rule regulation mechanisms developed for controlling the application of in-place model transformations on one model to model-to-model translations that have m models as input and n models as output [11].

Finally, we want to adopt parametrized polymorphism to our TGG approach as realized in PROGRES [14] or by the generics concept of Java for instance. This enables the user to specify abstract model transformations with abstract type parameters which will be provided at design- or even at runtime with concrete types.

8. REFERENCES

[1] F. Altheide et al. An Architecture for a Sustainable Tool Integration. In A. Schürr and H. Dörr, editors, *Workshop on Tool Integration in System Development*, pages 29–32, 2003. http://www.es.tu-darmstadt.de/english/events/tis/.

[2] S. Becker, T. Haase, and B. Westfechtel. Model-Based A-Posteriori Integration of Engineering Tools for Incremental Development Processes. *Journal of Software and Systems Modeling*, 4(2):123–140, 2005. Springer Verlag.

[3] J. de Lara and H. Vangheluwe. AToM^3: A Tool for Multi-formalism and Meta-modelling. In *FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, pages 174–188, London, UK, 2002. Springer-Verlag.

[4] H. Giese and R. Wagner. Incremental Model Synchronization with Triple Graph Grammars. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Genova, Italy*, volume 4199 of *Lecture Notes in Computer Science (LNCS)*, pages 543–557. Springer Verlag, October 2006.

[5] J. Greenyer. A Study of Model Transformation Technologies: Reconciling TGGs with QVT. Master's thesis, Universität Paderborn, Germany, 2006. http://wwwcs.uni-paderborn.de/cs/ag-schaefer/Veroeffentlichungen/Quellen/Diplom/2006/DiplomarbeitJGreenyer.pdf.

[6] J. Jakob, A. Königs, and A. Schürr. Non-materialized Model View Specification with Triple Graph Grammars. In A. Corradini, editor, *International Conference on Graph Transformations*, volume 4178 of *Lecture Notes in Computer Science (LNCS)*, pages 321–335, Heidelberg, 2006. Springer Verlag.

[7] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like transformation language. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 719–720, New York, NY, USA, 2006. ACM Press.

[8] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained*. Addison-Wesley, 2003.

[9] A. Königs. Model Transformation with Triple Graph Grammars. In *Model Transformations in Practice Satellite Workshop of MODELS 2005, Montego Bay, Jamaica*, 2005.

[10] A. Königs and A. Schürr. Tool Integration with Triple Graph Grammars - A Survey. In R. Heckel, editor, *Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 113–150. Elsevier Science Publ., 2006.

[11] H. Kreowski, S. Kuske, and A. Schürr. Nested Graph Transformation Units. *Int. Journal on Software and Knowledge Engineering and Special Issue on Graph Grammar-based Specifications*, 7(4):479–502, 1997.

[12] I. Kurtev, K. van den Berg, and F. Jouault. Evaluation of Rule-based Modularization in Model Transformation Languages illustrated with ATL. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1202–1209, New York, NY, USA, 2006. ACM Press.

[13] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 2. edition, 1997.

[14] M. Münch. *Generic Modelling with Graph Rewriting Systems*. Berichte aus der Informatik. Shaker Verlag, Aachen, 2003. PhD thesis (RWTH Aachen).

[15] OMG. *Unified Modeling Language version 1.3*, 2000. http://www.omg.org/cgi-bin/doc?formal/00-03-01.

[16] OMG. *Request for Proposal: MOF 2.0 Query / Views / Transformations RFP*, 2002. http://www.omg.org/cgi-bin/doc?ad/02-04-10.

[17] OMG. *UML 2.0 Infrastructure Specification*, 2003. http://www.omg.org/docs/ptc/03-09-15.pdf.

[18] OMG. *MOF QVT Final Adopted Specification*, 2005. http://www.omg.org/cgi-bin/doc?ptc/2005-11-01.

[19] OMG. *Meta Object Facility (MOF) Core Specification version 2.0*, 2006. http://www.omg.org/cgi-bin/doc?formal/2006-01-01.

[20] OMG. *Unified Modeling Language: Infrastructure version 2.0*, 2006. http://www.omg.org./docs/formal/05-07-05.

[21] T. W. Pratt. Pair Grammars, Graph Languages and String-to-Graph Translations. *Journal of Computer and System Sciences*, 5:560–595, 1971. Academic Press.

[22] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In G. Tinhofer, editor, *WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science (LNCS)*, pages 151–163, Heidelberg, 1994. Springer Verlag.

[23] A. J. Winter. *Visuelles Programmieren mit Graphtransformationen*, volume 27 of *Aachener Beiträge zur Informatik*. Wissenschaftsverlag Mainz in Aachen, 2000. PhD thesis in German (RWTH Aachen).

[24] A. Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, 2001. Habilitation Thesis.