Generic Schema Mappings

David Kensche, Christoph Quix, Yong Li, Matthias Jarke

RWTH Aachen University, Informatik 5 (Information Systems), 52056 Aachen, Germany
{kensche,quix,liyong,jarke}@i5.informatik.rwth-aachen.de

Abstract. Schema mappings come in different flavors: simple correspondences are produced by schema matchers, intensional mappings are used for schema integration. However, the execution of mappings requires a formalization based on the extensional semantics of models. This problem is aggravated if multiple metamodels are involved. In this paper we present extensional mappings, that are based on second order tuple generating dependencies, between models in our Generic Role-based Metamodel *GeRoMe*. By using a generic metamodel, our mappings support data translation between heterogeneous metamodels. Our mapping representation provides grouping functionalities that allow for complete restructuring of data, which is necessary for handling nested data structures such as XML and object oriented models. Furthermore, we present an algorithm for mapping composition and optimization of the composition result. To verify the genericness, correctness, and composability of our approach we implemented a data translation tool and mapping export for several data manipulation languages.

1 Introduction

Information systems often contain components that are based on different models or schemas of the same or intersecting domains of discourse. These different models of related domains are described in modeling languages (or metamodels) that fit certain requirements of the components such as representation power or tractability. For instance, a database may use SQL or an object oriented modeling language. A web service described in XML Schema may be enriched with semantics by employing an ontology of the domain. All these different types of models have to be connected by mappings stating how the data represented in one model is related to the data represented in another model. Integrating these heterogeneous models requires different means of manipulation for models and mappings which is the goal of a *Model Management* system. [3]. It should provide operators such as Match that computes a mapping between two models [17], ModelGen that transforms models between modeling languages [1], or Merge that integrates two models based on a mapping in between [16].

An important issue in a model management system is the representation of mappings which can be categorized as *intensional* and *extensional* mappings. Intensional mappings deal with the intended semantics of a model and are used, for example, in schema integration [16]. If the task is data translation or data integration, extensional mappings have to be used [9]. In this paper, we will deal only with extensional mappings as our goal is to have a generic representation for executable mappings.

An extensional mapping can be represented as two queries which are related by some operator (such as equivalent or subset) [9]. As the query language depends on

the modeling language being used, the question of mapping representation is tightly connected to the question how models are represented. In schema matching systems, which often represent the models as directed labeled graphs, mappings are represented as pairs of model elements with a confidence value which indicates their similarity [17]. Such mappings can be extended to path morphisms on tree schemas which can be translated into an executable form but have limited expressivity [12]. Other existing mapping representation rely on the relational model, e.g. tuple generating dependencies (tgds), GLAV mappings [11] or second order tgds [4]. For a nested relational model, a nested mapping language has been proposed [5].

Each mapping representation has its strengths and weaknesses regarding the requirements for a mapping language [3]: (i) mappings should be able to connect models of different modeling languages; (ii) the mapping language should support complex expressions between sets of model elements (m:n mappings); (iii) support for the nesting of mappings (to avoid redundant mapping specificiations) and nested data structures should be provided; (iv) mappings should have a rich expressiveness while being generic across modeling languages; (v) mappings should support the data translation between the instances of the connected models. While the mapping representations mentioned above fulfill these requirements for the (nested) relational model, they fail at being generic as they do not take into account other modeling languages. The goal of this paper is to define a mapping representation which is generic across several modeling languages and still fulfills the requirements regarding expressiveness and executability. This allows for a generic implementation of model management operators which deal with these mappings. Furthermore, each mapping language has its own characteristics regarding questions such as composability, invertability, decidability, and ability to be executed. Using a generic mapping representation, such questions can be addressed *once* for the generic mapping representation and do not have to be reconsidered for each combination mapping and modeling language.

A prerequisite for a generic representation of mappings is a generic representation of models. We developed the role based generic metamodel *GeRoMe* [7]. It provides a generic, but yet detailed representation of data models originally represented in different metamodels and is the basis for our model management system *GeRoMeSuite* [8]. *GeRoMeSuite* provides a framework for *holistic* generic model management; unlike other model management systems it is neither limited by nature to certain modeling languages nor to certain model management operators. The generic mapping language shown here is the basis for the data translation component of *GeRoMeSuite* and can be translated into a specific data manipulation language such as SQL.

The main contributions of our work define also the structure of the paper. After reviewing related work in section 2, we will define in section 3 a *generic mapping representation* based on the semantics of *GeRoMe*. We adapt second order tuple generating dependencies (SO tgds, [4]) originally defined for relational models to mappings between *GeRoMe* models which also allow for complex grouping and nesting of data. To show the usefulness and applicability of our mapping representation, we will present in section 4 an *algorithm for mapping composition*, and, in section 5, algorithms to translate our generic mapping representation into *executable mappings*. The *evaluation* of our approach with several examples of the recent literature is shown in section 6.

2 Background

Mappings: Extensional mappings are defined as local-as-view (LAV), global-as-view (GAV), source-to-target tuple generating dependencies (s-t tgds) [9,12], second order tuple generating dependencies (SO tgds) [4], or similar formalisms.

Clio [6] defines mappings over a nested relational model to support mappings between relational databases and XML data. However, it would still be difficult to extend this mapping representation to express a mapping between other models, such as UML models, because there is simply no appropriate query language. On the other hand, it is always possible to compose these mappings, because the composition of such mappings is equivalent to the composition of queries [14].

Besides being not generic, another drawback of these *basic mappings* is pointed out: they do not reflect the nested structure of the data [5]. This leads to an inefficient execution of the mappings and redundant mapping specifications as parts of the mapping have to be repeated for different nesting levels. Furthermore, the desired grouping of the target data cannot be specified using basic mappings which leads to redundant data in the target. Fuxman et al. [5] proposed a nested mapping language which addresses these problems. The desired nesting and grouping of data can be expressed in the mapping specification. Another form of basic mappings based on a Prolog-like representation is used by Atzeni et al. [1]. These mappings are generic as they are based on a generic metamodel, but they require the data to be imported to the generic representation as well. This leads to an additional overhead during execution of the mappings.

Mapping Composition: In general, the problem of composing mappings has the following formulation: given a mapping M_{12} from model S_1 to model S_2, and a mapping M_{23} from model S_2 to model S_3, derive a mapping M_{13} from model S_1 to model S_3 that is equivalent to the successive application of M_{12} and M_{23} [4].

Mapping composition has been studied only for mappings which use the Relational Data Model as basis. Fagin et al. [4] proposed a semantics of the Compose operator that is defined over instance spaces of schema mappings. To this effect, M_{13} is the composition of M_{12} and M_{23} means that the instance space of M_{13} is the set-theoretical composition of the instance spaces of M_{12} and M_{23}. Under this semantics, which we will also adopt in this paper, the mapping composition M_{13} is unique up to logical equivalence. Fagin et al. also explored the properties of the composition of schema mappings specified by a finite set of s-t tgds. They proved that the language of s-t tgds is not closed under composition. To ameliorate the problem, they introduced the class of SO tgds and proved that (i) SO tgds are closed under composition by showing a mapping composition algorithm; (ii) SO tgds form the smallest class of formulas (up to logical equivalence) for composing schema mappings given by finite sets of s-t tgds; and (iii) given a mapping M and an instance I over the source schema, it takes polynomial time to calculate the solution J which is an instance over the target schema and which satisfies M. Thus, SO tgds are a good formalization of mappings.

Another approach for mapping composition uses expressions of the relational algebra as mappings [2]. The approach uses an incremental algorithm which tries to replace as many symbols as possible from the "intermediate" model. As the result of mapping composition cannot be always expressed as relational algebra expressions, the algorithm may fail under certain conditions which is inline with the results of [4].

Executable mappings: Executable mappings are necessary in many meta-data intensive applications, such as database wrapper generation, message translation and data transformation [12]. While many model management systems were used to generate mappings that drive the above applications, few of them were implemented using executable mappings. Because executable mappings usually drive the transformation of instances of models, Melnik et al. [12] specified a semantics of each operator by relating the instances of the operator's input and output models. They also implemented two model management system prototypes to study two approaches to specifying and manipulating executable mappings. In the first implementation, they modified *Rondo*'s [13] language to define path morphisms and showed that it is possible to generate executable mappings in a form of relational algebra expressions. On the positive side, this system works correctly whenever the input is specified using path morphisms, and the input is also closed under operators which return a single mapping. However, the expressiveness of path morphisms is very limited. To overcome this limitation, they developed a new prototype called Moda [12] in which mappings are specified using embedded dependencies. The expressiveness is improved in the second implementation, but it suffers from the problem that embedded dependencies are not closed under composition. Because of this problem, the output of the Compose operator may not be representable as an embedded dependency and thus a sequence of model management operators may not be executable in the system. Although they further developed a script rewriting procedure to ameliorate this problem, it has not been completely solved.

3 Mapping Representation

Before we define the representation of mappings for *GeRoMe* models, we first present the main concepts of *GeRoMe* using an example (section 3.1). As mappings relate instances of models, we have to define how instances of a *GeRoMe* model can be represented, i.e. defining a formal semantics for *GeRoMe* as described in section 3.2. This representation forms the basis for our mapping representation presented in section 3.3.

3.1 The Generic Metamodel *GeRoMe*

Our representation of mappings is based on the generic role based metamodel *GeRoMe* [7], which we will introduce briefly here. *GeRoMe* provides a generic but detailed representation of models originally expressed in different modeling languages. In *GeRoMe* each model element of a native model (e.g. an XML schema or a relational schema) is represented as an object that plays a set of roles which decorate it with features and act as interfaces to the model element. Fig. 1 shows an example of a *GeRoMe* model representing an XML schema.

The grey boxes in fig. 1 denote model elements, the attached white boxes represent the roles played by the model elements. XML Schema is in several aspects different from "traditional" modeling languages such as EER or the Relational Metamodel. The main concept of XML Schema "element" represents actually an association between the nesting and the nested complex type. This is true for all elements except those which are allowed as root element of a document. In the *GeRoMe* representation of an XML

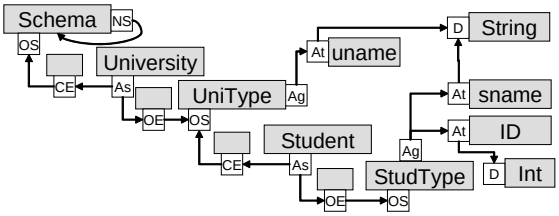

Fig. 1. *GeRoMe* representation of an XML schema

schema, the root element is an association between the schema node and the top-level complex type, as there is no complex type in which the root element is nested. In the example of fig. 1 [1], the element `University` is an association between the model element `Schema` and the complex type `UniType`. The fact that the `University` element is an association is described by the *Association* (As) role which connects the *ObjectSet* (OS) roles of `Schema` and `UniType` via two anonymous model elements playing a *CompositionEnd* (CE) and an *ObjectAssociationEnd* (OE) role, respectively. The same structure is used for the element `Student` which is an association between the complex types `UniType` and `StudType`. The two complex types have also attributes; therefore, they play also the *Aggregate* (Ag) role which links these model elements to their attributes. The model elements representing attributes play an *Attribute* (At) role which refers also to the type of the attributes which are, in this example, simple domains denoted by the *Domain* (D) role.

It is important to emphasize that this representation is not to be used by end users. Instead, it is a representation employed internally by model management applications, with the goal to generically provide more information to model management operators than a simple graph based model.

3.2 *GeRoMe* Semantics: Instances of a *GeRoMe* Model

Before we can formally define *GeRoMe* mappings, we first need to define the formal semantics of *GeRoMe* instances. Our mappings are second-order tuple generating dependencies (SO tgds) which requires that the instances are represented as a set of logical facts. In addition, the semantics should also capture all the structural information that is necessary to reflect the semantics of the model. To fulfill both requirements, the semantics should contain facts that record literal values of an instance of a model and also facts that describe the structure of that instance. To record the literal values of an instance, $value$ predicates are used to associate literal values with objects. To describe the structure of an instance, we identify *Attribute* and *Association* as the roles which essentially express the structure of instances.

[1] XML documents may have only one root element. Thus, the schema needs to have another element "Universities" to allow for a list of universities in the XML document. For reasons of simplicity, we omitted this extra element in our example and assume that XML documents may have multiple elements at the top-level.

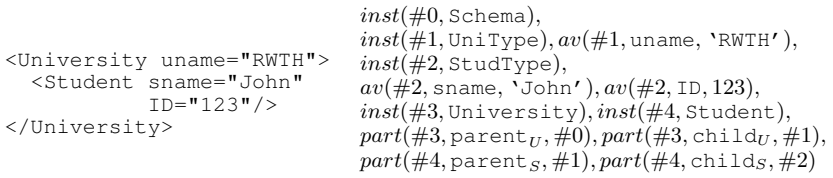

Fig. 2. XML document and its representation as *GeRoMe* instance

Definition 1 (Interpretation of a *GeRoMe* model) *Let M be a* GeRoMe *model with $\mathcal{A}$ being the set of all literal values, and $\mathcal{T}$ the set of all abstract identifiers $\{id_1, \ldots, id_n\}$. An interpretation $\mathcal{I}$ of M is a set of facts $\mathcal{D}_M$, where:*

- $\forall$ *objects (represented by the abstract identifier id_i) which are an instance of model element m: $inst(id_i, m) \in \mathcal{D}_M$,*
- $\forall$ *elements m playing a* Domain *role and $\forall$ values v in this domain: $\{value(id_i, v), inst(id_i, m)\} \subseteq \mathcal{D}_M$ (id_i is an abstract ID of an object representing the value v).*
- $\forall$ *elements m playing an* Aggregate *role and having the attribute a, and the instance id_i has the value v for that attribute: $\{attr(id_i, a, id_v), value(id_v, v)\} \subseteq \mathcal{D}_M$.*
- $\forall$ *model elements m playing an* Association *role in which the object with identifier o participates for the association end ae: $part(id_i, ae, o) \in \mathcal{D}_M$.*

Thus, each "feature" of an instance object is represented by a separate fact. The abstract IDs connect these features so that the complete object can be reconstructed. For the example from fig. 1, an instance containing a university and a student is defined as show in fig. 2. As the predicates $attr$ and $value$ often occur in combination, we use the predicate av as a simplification: $av(id_1, a, v) \Leftrightarrow \exists id_2 attr(id_1, a, id_2) \wedge value(id_2, v)$. In addition, we labeled the association ends with "parent" and "child" to make clear which association end is referred to. The first $inst$-predicate defines an instance of the schema element which represents the XML document itself. Then, two instances of the complex types and their attributes are defined. The last three lines define the associations and the relationships between the objects defined before.

As the example shows, association roles and attribute roles are not only able to define flat structures, e.g. tables in relational schemas, but also hierarchical structures, e.g. element hierarchies in XML schemas. Compared to the original definition of SO tgds, which were only used to represent tuples of relational tables, our extension to the original SO tgds significantly improves the expressiveness of SO tgds.

3.3 Formal Definition of GeRoMe Mappings

Based on the formal definition of *GeRoMe* instances, the definition of *GeRoMe* mappings as SO tgds is straightforward. We extend the definition of a mapping between two relational schemas in [4] to the definition of a mapping between two *GeRoMe* models:

Definition 2 (*GeRoMe* Mapping) *A* GeRoMe *model mapping is a triple $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$, where $\mathbf{S}$ and $\mathbf{T}$ are the source model and the target model respectively, and where Σ is a*

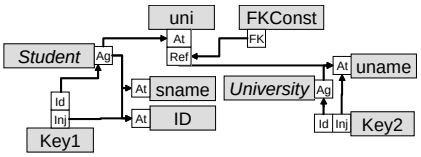

Fig. 3. *GeRoMe* representation of a relational schema

$$
\begin{aligned}
&\exists f, g \quad \forall o_0, o_1, o_2, o_3, o_4, u, s, i \\
&inst(o_1, \texttt{University}) \wedge part(o_1, \texttt{parent}_U, o_0) \wedge part(o_1, \texttt{child}_U, o_2) \wedge \\
&inst(o_3, \texttt{Student}) \wedge part(o_3, \texttt{parent}_S, o_2) \wedge part(o_3, \texttt{child}_S, o_4) \wedge \\
&av(o_2, \texttt{uname}, u) \wedge av(o_4, \texttt{sname}, s) \wedge av(o_4, \texttt{ID}, i) \rightarrow \\
&\qquad inst(f(u), \texttt{University}) \wedge inst(g(i), \texttt{Student}), \\
&\qquad av(f(u), \texttt{uname}, u) \wedge av(g(i), \texttt{sname}, s) \wedge av(g(i), \texttt{ID}, i) \wedge av(g(i), \texttt{uni}, u)
\end{aligned}
$$

Fig. 4. Mapping between XML and relational schema

set of formulas of the form: $\exists \mathbf{f}((\forall \mathbf{x_1}(\varphi_1 \rightarrow \psi_1)) \wedge \ldots \wedge (\forall \mathbf{x_n}(\varphi_n \rightarrow \psi_n)))$ *where* $\mathbf{f}$ *is a collection of function symbols, and where each* φ_i *is a conjunction of atomic predicates and/or equalities over constants defined in* $\mathbf{S}$ *and variables, and* ψ_i *is a conjunction of atomic predicates over constants defined on* $\mathbf{T}$*, variables, and function symbols. Valid atomic predicates are those defined in def. 1. Furthermore, we require that every element name in these atomic predicates is a constant, i.e. the second arguments of* $inst$*,* $attr$ *and* $part$ *predicates are constants.*

To show an example of a mapping between models originally represented in two different modeling languages, we define in fig. 3 a *GeRoMe* model representing a relational schema that corresponds to the XML schema in fig. 1. The schema contains two relations `University(uname)` and `Student(id,sname,uni)`. The keys `uname` and `id` are defined in *GeRoMe* using separate model elements representing the key constraint. These model elements play an *Injective* (Inj) role to indicate that an attribute is unique, and an *Identifier* (Id) role to specify the aggregate for which the attribute is the key. The foreign key constraint between `Student` and `University` is also represented by a separate model element which plays a *Foreign Key* (FK) role. The FK role points to a *Reference* (Ref) role which is played by the attribute that references the key of the other relation.

Now, we can define a mapping using the XML schema as source and the relational schema as target (cf. fig. 4). The predicates in the conditional part of the rule correspond to the instance predicates shown in fig. 2, now just with variables instead of constants. A remark has to be made about the variables o_0 to o_4: these variables represent abstract identifiers (to be named *abstract variables* in the following), their main function is to describe (implicitly) the structure of the source data; in the example we can see, that the student element o_3 is nested under the university element o_1. In other approaches for mapping representation (e.g. [5]), this is done by nesting different sub-expressions of a query. Although nested mappings are easier to understand, they are less expressive

than SO tgds [5]. In addition, several tasks dealing with mappings such as composition, inverting, optimization, and reasoning have to be reconsidered for nested mappings (e.g. it is not clear how to compose nested mappings and whether the result composing two nested mappings can be expressed as a nested mapping). As our approach is based on SO tgds, we can leverage the results for SO tgds for our generic mapping representation.

Similarly to the abstract variables on the source side, the functions f and g represent abstract identifiers on the target side and therefore describe the structure of the generated data in the target. We will call such functions (which generate abstract identifiers) in the following *abstract functions*. Abstract functions can be understood as Skolem functions which do not have an explicit semantics; they are interpreted by syntactical representation as term. Please note that abstract variables and abstract functions just specify the structure of data, there will be no values assigned to abstract variables or evaluation of abstract functions during the execution of a mapping. Instead, as we will present in section 5, abstract identifiers and functions determine the structure of the generated code to query the source and to insert the data in the target.

To describe the structure of the target data, it is important to know which values are used to identify an object. According to the definition of the relational schema, universities are identified by their name (u) and students by their ID (i); that is why we use u and i as arguments of the abstract functions f and g. We will explain below that for nested data these functions will usually have more than one argument.

In addition to abstract functions, a mapping can also contain "normal" functions for value conversions or some other kind of data transformation (e.g. concatenation of first and last name). While executing a mapping, these functions must be actually evaluated to get the value which has to be inserted into the target.

The example shows also that only variables representing values occur on both sides of the implication. Abstract variables will be used only on the source side of a mapping as they refer only to source objects, abstract functions will appear only on the target side as they refer only to objects in the target. This implies that for the execution of the mapping, we need to maintain a table with the values extracted from the source, and then generate the target data using these values according to the mapping.

Grouping and Nesting: The generation of complex data structures which can be arbitrarily nested is an important requirement for a mapping representation. In order to show that our mapping language is able to express complex restructuring operations in a data transformation, we use an example that transforms relational data into XML. The relational schema is as in fig. 3 with the exception that we now assume that students may study at multiple universities. To have a schema in 3NF, we add a relation `Studies` with two foreign keys `uni` and `id`. The foreign key from the `Student` relation is removed. On the target side, the data should be organized with students at the top level, and the list of universities nested under each student. The mapping between the updated relational and XML schemas is shown in fig. 5.

The source side is almost identical with the target side of the previous mapping: the abstract functions f and g have been replaced with the abstract variables o_1 and o_2; a variable o_3 for the `Studies` relation and the corresponding av predicates have been added. On the target side, we first generate an instance of the `Student` element; as students are identified by their ID, the abstract function f has only i as argument. $f'(i)$

$$
\begin{aligned}
&\exists f, f', g, g', d \quad \forall o_1, o_2, o_3, u, s, i \\
&inst(o_1, \texttt{University}) \wedge inst(o_2, \texttt{Student}), inst(o_3, \texttt{Studies}) \\
&av(o_1, \texttt{uname}, u) \wedge av(o_2, \texttt{sname}, s) \wedge av(o_2, \texttt{ID}, i) \wedge av(o_3, \texttt{uni}, u) \wedge av(o_3, \texttt{id}, i) \rightarrow \\
&\quad inst(f(i), \texttt{Student}) \wedge part(f(i), \texttt{parent}_S, d()) \wedge part(f(i), \texttt{child}_S, f'(i)) \wedge \\
&\quad av(f'(i), \texttt{sname}, s) \wedge av(f'(i), \texttt{ID}, i) \wedge \\
&\quad inst(g(i,u), \texttt{University}) \wedge part(g(i,u), \texttt{parent}_U, f'(i)) \wedge \\
&\quad part(g(i,u), \texttt{child}_U, g'(i,u)) \wedge av(g'(i,u), \texttt{uname}, u)
\end{aligned}
$$

Fig. 5. Mapping from relational data to XML

represents an instance of `StudType`[2] for which we also define the attribute values of `sname` and `ID`. Thus, if a student studies at more than one university and therefore occurs multiple times in the result set of the source, only one element will be created for that student and all universities will be correctly grouped under the `Student` element.

On the other hand, we must make sure that there is more than one `University` element for each university, as the universities have to be repeated for each student. This is guaranteed by using both identifiers (of the nesting element `Student` and the nested element `University`, i and u) as arguments of the abstract function g. Finally, we assign a value to the attribute `uname` of the instance $g'(i, u)$ of `UniType`, similarly as before for the instance of `StudType`.

Constructing Mappings: Our mappings have a rich expressivity, but are hard to understand in their formal representation, even for an information system developer who is used to working with modeling and query languages. As mentioned above, *GeRoMe* should not replace existing modeling languages, users will still use the modeling language that fits best their needs. *GeRoMe* is intended as an internal metamodel for model management applications. This applies also to the *GeRoMe* mappings, users will not define mappings using the SO tgds as defined above, rather they will use a user interface in which they can define the mappings graphically.

As part of our model management system *GeRoMeSuite* [8], we are currently developing mapping editors for the various forms of mappings. In these mapping editors, the models are visualized as trees (based on the hierarchy of associations and aggregations), and the mapping can be defined by connecting elements of the trees. However, such a visualization of models and mappings has limited expressivity (it roughly corresponds to the path morphisms and tree schemas used in Rondo [13]) as not every model can be easily visualized as a tree. Even an XML schema can break up the tree structure by having references between complex types.

Our current design for an editor for extensional mappings also visualizes models as trees. In addition, mappings as nested structures to represent their possibly complex grouping and nesting conditions. Still, an appropriate representation of complex mappings is an active research area [18], and we have to evaluate whether our design will be accepted by users.

[2] The $inst$ predicate has been omitted as it is redundant: $f'(i)$ is declared as child of a `Student` element; thus, it is an instance of `StudType` according to the schema definition.

4 Mapping Composition

Composition of mappings is required for many model management tasks [2]. In a data integration system using the global-as-view (GAV) approach, a query posed to the integrated schema is rewritten by composing it with the mapping from the sources to the integrated schema. Schema evolution is another application scenario: if a schema evolves, the mappings to the schema can be maintained by composing them with an "evolution" mapping between the old and the new schema.

According to [4], the composition of two mappings expressed as SO tgds can be also expressed as SO tgd. In addition, the algorithm proposed in [4] guarantees, that predicates in the composed SO tgd must appear in the two composing mappings. Thus, the composition of two *GeRoMe* mappings is always definable by a *GeRoMe* mapping. It is important that *GeRoMe* mappings are closed under composition, because otherwise the Compose operator may not return a valid *GeRoMe* mapping.

In the following, we will first show the adaptation of the algorithm of [4] to *GeRoMe*, which enables mappings between heterogeneous metamodels. In the second part of this section, we address an inherent problem of the composition algorithm, namely that the size of the composed mapping is exponential in the size of the input mappings. We have developed some optimization techniques which reduce the size of the composed mapping using the semantic information given in the mapping or model.

Composition Algorithm: The composition algorithm shown in fig. 6 takes two *GeRoMe* mappings $\mathcal{M}_{12}$ and $\mathcal{M}_{23}$ as input. The aim is to replace predicates on the left hand side (lhs) of Σ_{23}, which refer to elements in $\mathbf{S}_2$, with predicates of the lhs of Σ_{12}, which refer only to elements in $\mathbf{S}_1$. As the first step, we rename the predicates in such a way that the second argument (which is always a constant) becomes part of the predicate name. This lets us avoid considering the constant arguments of a predicate when we are looking for a "matching" predicate, we can just focus on the predicate name. Then, we replace each implication in Σ_{12} with a set of implications which just have one predicate on the right hand side (rhs). We put the normalized implications from Σ_{12} with the updated predicate names into $\mathcal{S}_{12}$. For the implications in Σ_{23}, we just need to change the predicate names, and then we insert them into $\mathcal{S}_{23}$.

The next step performs the actual composition of the mappings. As long as we have an implication in $\mathcal{S}_{23}$ with a predicate $P.c(\mathbf{y})$ in the lhs that refers to $\mathbf{S}_2$, we replace it with every lhs of a matching implication from $\mathcal{S}_{12}$. Moreover, we have to add a set of equalities which reflect the unification of the predicates $P.c(\mathbf{y})$ and $P.c(\mathbf{t}_i)$.

During the composition step, the size of the resulting mapping may grow exponentially. As a first step towards a simpler result, we remove in the next step the variables which were originally in $\mathcal{M}_{23}$. This reduces the number of equalities in the mapping. The final step creates the composed mapping as one formula from the set of implications $\mathcal{S}_{23}$. The following theorem states that the algorithm produces actually a correct result. Due to space restrictions, we cannot show the proof of the theorem (the full proof is given in [10]), it is based on the correctness of the composition algorithm in [4].

Theorem 3 *Let $\mathcal{M}_{12} = (\mathbf{S}_1, \mathbf{S}_2, \Sigma_{12})$ and $\mathcal{M}_{23} = (\mathbf{S}_2, \mathbf{S}_3, \Sigma_{23})$ be two GeRoMe mappings. Then the algorithm Compose($\mathcal{M}_{12}$, $\mathcal{M}_{23}$) returns a GeRoMe mapping $\mathcal{M}_{13} = (\mathbf{S}_1, \mathbf{S}_3, \Sigma_{13})$ such that $\mathcal{M}_{13} = \mathcal{M}_{12} \circ \mathcal{M}_{23}$.*

Input: Two *GeRoMe* mappings $\mathcal{M}_{12} = (\mathbf{S}_1, \mathbf{S}_2, \Sigma_{12})$ and $\mathcal{M}_{23} = (\mathbf{S}_2, \mathbf{S}_3, \Sigma_{23})$
Output: A *GeRoMe* mapping $\mathcal{M}_{13} = (\mathbf{S}_1, \mathbf{S}_3, \Sigma_{13})$

Initialization: Initialize $\mathcal{S}_{12}$ and $\mathcal{S}_{23}$ to be empty sets.

Normalization: Replace in Σ_{12} and Σ_{23} predicates $P(x, c, y)$ (or $P(x, c)$) where $P \in \{inst, attr, av, part\}$ with $P.c(x, y)$ (or $P.c(x)$); replace implications in Σ_{12} of the form $\phi \rightarrow p_1 \wedge \ldots \wedge p_n$ with a set of implications $\phi \rightarrow p_1, \ldots, \phi \rightarrow p_n$. Put the resulting implications into $\mathcal{S}_{12}$ and $\mathcal{S}_{23}$, respectively.

Composition: Repeat until all predicates on the lhs of implications in $\mathcal{S}_{23}$ do not refer to $\mathbf{S}_2$: Let χ be an implication of the form $\psi \rightarrow \phi \in \mathcal{S}_{23}$ with a predicate $P.c(\mathbf{y})$ in ψ and $\phi_1(\mathbf{x}_1) \rightarrow P.c(\mathbf{t}_1), \ldots, \phi_n(\mathbf{x}_n) \rightarrow P.c(\mathbf{t}_n)$ be all the implications in $\mathcal{S}_{12}$ with predicate $P.c$ on the rhs ($\mathbf{x}$, $\mathbf{y}$ and $\mathbf{t}_i$ being vectors of variables and terms, respectively). If there is no such implication, remove χ from $\mathcal{S}_{23}$ and consider the next predicate. Remove χ from $\mathcal{S}_{23}$. For each implication $\phi_i(\mathbf{x}_i) \rightarrow P.c(\mathbf{t}_i)$, create a copy of this implication using new variable names, and replace $P.c(\mathbf{y})$ in ψ with $\phi_i(\mathbf{x}_i) \wedge \theta_i$ where θ_i are the component-wise equalities of $\mathbf{y}$ and $\mathbf{t}_i$ and add the new implication to $\mathcal{S}_{23}$.

Remove Variables: Repeat until all variables originally from Σ_{23} are removed: For each implication χ in $\mathcal{S}_{23}$, select an equality $y = t$ introduced in the previous step and replace all occurences of y in χ by t.

Create Result: Let $\mathcal{S}_{23} = \{\chi_1, \ldots, \chi_r\}$. Replace the predicates with their original form (e.g. $P.c(x, y)$ with $P(x, c, y)$). Then, $\Sigma_{13} = \exists \mathbf{g}(\forall \mathbf{z}_1 \chi_1 \wedge \ldots \wedge \forall \mathbf{z}_r \chi_r)$ with $\mathbf{g}$ being the set of function symbols appearing in $\mathcal{S}_{23}$ and $\mathbf{z}_i$ being all the variables appearing in χ_i.

Fig. 6. Algorithm **Compose**

Semantic Optimization of the Composition Result: We realized that the composed mapping has on the lhs many similar sets of predicates. The reason for this is that we replace a predicate in $\mathcal{S}_{23}$ with a conjunction of predicates in $\mathcal{S}_{12}$ and the same set of predicates in $\mathcal{S}_{12}$ may appear multiple times. Although the result is logically correct, the predicates on the lhs of the composition seems to be duplicated. We show in the following that both undesired implications and duplicated predicates can be removed.

A detailed inspection of our mapping definition reveals that only variables representing values correspond to values in an instance of the underlying model. All other arguments are either constants which correspond to names in a model or terms which correspond to abstract identifiers that identify *GeRoMe* objects. These abstract identifiers and the functions that return an abstract identifier are interpreted only syntactically. Thus, we are able to formulate the following conditions for *abstract functions*:

$$\forall f \forall g \forall \mathbf{x} \forall \mathbf{y}(f \neq g) \rightarrow f(\mathbf{x}) \neq g(\mathbf{y}),\ f, g \text{ are abstract functions}$$
$$\forall \mathbf{x} \forall \mathbf{y}(f(\mathbf{x}) = f(\mathbf{y}) \rightarrow \mathbf{x} = \mathbf{y}),\ f \text{ is an abstract function}$$

The first statement says that different abstract functions have different ranges. Using this statement, we can remove implications which have equalities of the form $f(\mathbf{x}) = g(\mathbf{y})$ on the lhs, because they never can become true. The second statement says that an abstract function is a bijection, i.e. whenever two results of an abstract function are equal, then the inputs are equal, too. This statement can be used to reduce the number of predicates in the composed mapping, e.g. if $f(o) = f(p)$ is contained in the lhs of an implication, then we can infer that $o = p$ and therefore replace all occurences of o with p (or vice versa). This will produce identical predicates in the conjunction, duplicates can then be removed without changing the logical meaning of the formula. Other optimizations use the constraint information of the model to reduce the complexity of the composed mapping, e.g. keys or cardinality constraints.

5 Mapping Execution

In this section we first describe the architecture of our data translation tool before we explain how we generate queries from a set of generic mappings and how we use these queries to produce target data from source data.

Fig. 7 shows the architecture of our data translation tool. Given the mapping and the source model as input, the code generator produces queries against the source schema. An implementation of this component must be chosen, so that it produces queries in the desired data manipulation language. In the same way, the target model code generator produces updates from the mapping and the target *GeRoMe* model.

Given the generated queries and updates the query executor produces variable assignments from the evaluation of the queries against the source data. The update executor then receives these generic variable assignments as input and produces the target data. Hence, components related to source and target respectively are only loosely coupled to each other by the variable assignments whereas the query/update generator and the executor components have to fit to each other.

Our query generation is based on the model transformations between native metamodels and *GeRoMe*. We now exemplarily introduce our algorithm for generating XQueries from our generic mappings (cf. fig. 8). However, our tool transforms data arbitrarily between relational and XML schemas; these generation and execution components can also be replaced by components that handle other metamodels (e.g. OWL or UML).

The element hierarchy T describes the structure that is queried for, the condition set P contains select and join conditions and the return set R assigns XQuery variables for values of attributes and simple typed elements in the source side of the mapping. The last step uses the computed data to produce the actual XQuery where _fname will be replaced with the actual XML file name when the query is executed.

We now generate an XQuery from the mapping in fig. 4 that can be used to query the document in fig. 2. In fig. 4, we omitted the term specifying the document element for brevity and simplicity. Assume the lhs of the mapping contains a term $inst(o_0, \texttt{Schema})$. Then o_0 is the variable referencing the document element. Therefore, we put $(o_0, /)$ as the root into T and also into $Open$.

Now, we construct the element hierarchy T. For $(o_0, /)$ in $Open$ the required pattern is satisfied by the subformula $inst(o_1, \texttt{University}) \wedge part(o_1, \texttt{parent}_U, o_0) \wedge$

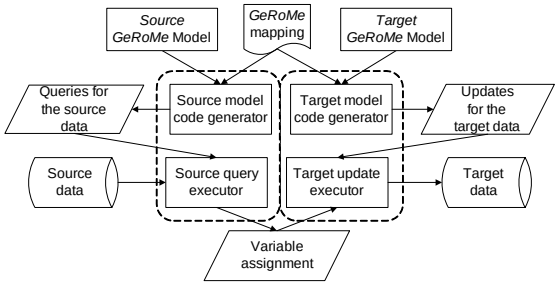

Fig. 7. The architecture of the data translation tool

Input: An implication χ in an SO tgd Σ with source schema σ
Output: A set of XQuery queries over σ.

Initialization: $T = Open = Close = R = \emptyset$
Find document variable: This is the only variable symbol S that occurs in a term of the form $inst(S, \texttt{Schema})$ on the lhs of χ where `Schema` is the name of the schema element. Add $(S, \text{“}/\text{”})$ as the root to T and add it to $Open$.
Construct element hierarchy T: Repeat the following until $Open = \emptyset$. Let $(X, path) \in Open$. For each subformula $inst(Id, name) \wedge part(Id, ae_1, X) \wedge part(Id, ae_2, Y)$ on the lhs of χ where $Id, X,$ and Y are variable symbols, $name$, ae_1 and ae_2 are *GeRoMe* element names, and there is no $path'$ with $(Y, path') \in Close$, add $(Y, path/name)$ to $Open$ and add it as a child to $(X, path)$ in T with label $name$. Finally, remove $(X, path)$ from $Open$ and add it to $Close$.
Construct return set R: For each term $av(X, a, V)$ on the lhs of χ, where X, V are variable symbols, a is a constant name of an attribute, and $(X, path) \in T$, add (V, “$\$X$/@$a$”) to R. For each term $value(X, V)$ on the lhs of χ, where X, V are variable symbols and $(X, path) \in T$, add (V, “$\$X$/text()”) to R.
Construct condition set P: $V_1 = V_2$ on the lhs of χ with $(V_1, path_1) \in R \wedge (V_2, path_2) \in R$ specifies an explicit join condition. Add “$path_1$ **eq** $path_2$” to P. If $(V, path_1) \in R \wedge (V, path_2) \in R$ this specifies an implicit join condition. Add “$path_1$ **eq** $path_2$” to P. For each term $value(V, c)$ on the lhs of χ, where c is a constant, add “V **eq** c” to P.
Produce XQuery: Let $T = (doc, \text{“}/\text{”})[(e_{1,1}, p_{1,1}, l_{1,1})[(e_{2,1}, p_{2,1}, l_{2,1})[\ldots], \ldots, (e_{2,k_2}, p_{2,k_2}, l_{2,k_2})]]$ be the computed element tree, where $(e, p, l)[\ldots, (e', p', l'), \ldots]$ means (e', p') is a child of (e, p) with label l' in T. Let $(v_1, path_1), (v_2, path_2), \ldots, (v_n, path_n) \in R$, and $p_{1,1}, p_2, \ldots, p_n \in P$. The XQuery query for χ is:

for $\$e_{1,1}$ **in** fn:doc(_fname)/l_1
  for $\$e_{2,1}$ **in** $\$e_{1,1}/l_{2,1}$ **...**
  for $\$e_{2,k_2}$ **in** $\$e_{1,1}/l_{2,k_2}$
    for $\$e_{3,1}$ **in** $\$e_{2,i_{3,1}}/l_{3,1}$ **...**
where p_1 **and** p_2 **and** $\ldots$ **and** p_n
return <result> $< v_1 > path_1 </ v_1 > \ldots < v_n > path_n </ v_n >$ </result>

Fig. 8. Algorithm **XQueryGen**

$part(o_1, \texttt{child}_U, o_2)$ We add $(o_2, /\texttt{University})$ to $Open$ and also add it to T as a child of $(o_0, /)$ with label `University`. As no other subformula satisfies the pattern, we remove $(o_0, /)$ from $Open$ and add it to $Close$. We get $Open = \{(o_2, /\texttt{University})\}$, $T = (o_0, /)[(o_2, /\texttt{University}, \texttt{University})]$ and $Close = \{(o_0, /)\}$. We repeat the step for $(o_2, /\texttt{University})$. The result for T after this step is $(o_0, /)[(o_2, /\texttt{University}, \texttt{University})[(o_4, /\texttt{University/Student}, \texttt{Student})]]$. The last iteration does not add any elements.

Three variables on the lhs of χ are assigned by the query, u, s and i. According to the rules described in the algorithm, we add $(u, /\texttt{University}/@uname)$, $(s, /\texttt{University/Student}/@sname)$ and $(i, /\texttt{University/Student}/@ID)$ to the return set R. There are no join or select conditions in the mapping, therefore, the condition set for this mapping remains empty. The assignments to the variables u, s and i that are returned by the query are used as input when executing the rhs of the mapping. The XQuery generated from χ is:

for $\$o_2$ **in** fn:doc(_fname)/University
  for $\$o_4$ **in** $\$o_2$/Student

return <result>
<u>o_2/@uname</u> <s>o_4/@sname</s> o_4/@ID
</result>

6 Evaluation

Correctness and Performance: To evaluate mapping composition we used nine composition problems drawn from recent literature [4,12,14]. The original mappings had to be translated manually into our representation before composing. The results of composition were logically equivalent to the documented results. The same set of problems had been used to evaluate the performance of our implementation. As was proven in [4] the computation time of composition may be exponential in the size of the input mappings. Fig. 9 displays the time performance of our composition implementation.

NI12	MP23	IC	TT(ms)
3	3	2	200
6	4	3	420
6	4	4	605
3	3	1	372
5	5	1	391
5	5	1	453
7	7	4	1324
15	6	99	16122
15	9	288	100170

Note: *NI12*: the number of implications in the normalized $\mathcal{S}_{12}$
MP23: the maximum number of predicates in each implication in Σ_{23}
IC: the number of implications in the un-optimized composition
TT: the total run time of running the composition algorithm 200 times.

Fig. 9. Time performance of the composition algorithm

The upper bound of the number of implications in the non-optimized composition is $O(\sum_i(I^{P_i}))$, where I is the number of implications in the normalized Σ_{12} and P_i is the number of source predicates in each implication in Σ_{23}. In the second step of our composition algorithm, a predicate on the left hand side of χ can have at most I matched implications in $\mathcal{S}_{23}$. Since only one implication is generated for each matched implication, after replacing the predicate, the number of implications in $\mathcal{S}_{23}$ will increase at most at the factor of I. Repeat the same reasoning for every source predicate in Σ_{23} will lead to the stated upper bound. In table 9, we listed, for each test case, the I, the maximum of P_i, the size of the un-optimized composition and its running time. It can be seen that the execution time may indeed be exponential in the size of input mappings. Even though composing mappings may be expensive, the performance is still reasonable. It takes about 80 milliseconds to run a test that generates in total 99 implications and about half a second for a test which generates 288 implications.

To evaluate mapping execution we defined seven test cases between relational databases and XML documents. The performance was linear in the size of the output and thus, our framework does not impose a significant overhead to data exchange tasks. These tests included also executing the composition of two mappings from a relational to an XML Schema and back. The result was an identity mapping and execution of the

optimized result was about twice as fast as subsequent execution of the mappings. Our tests showed that our mapping execution yields the desired results satisfying both, the logical formalisms and the grouping semantics specified in the mappings. All tests were run on a Windows XP machine with a P4 2.4GHz CPU and 512MB memory.

Comparison with other Mapping Representations: Source-to-target tuple-generating-dependencies (s-t tgds) and GLAV assertions are used to specify mappings between relational schemas. They are strict subsets of our adaptation of SO tgds. Every s-t tgd has a corresponding *GeRoMe* mapping but not vice versa. *GeRoMe* mappings can express nested data structures, e.g. XML data, while s-t tgds can not.

Path-conjunctive constraints [15] are an extension of s-t tdgs for dealing with nested schemas. However, they may suffer from several problems [5]. First, the same set of paths may be duplicated in many formulas which induces an extra overhead on mapping execution. Second, grouping conditions cannot be specified, leading to incorrect grouping of data. Nested mappings [5] extend path-conjunctive constraints to address the above problems. Nested mappings merge formulas sharing the same set of high level paths into one formula, which causes mapping execution to generate less redundancy in the target. In addition, nested mappings provide a syntax to specify grouping conditions. Compared to nested mappings, *GeRoMe* mappings are also able to handle nested data and specify arbitrary grouping conditions for elements. Furthermore, the language of SO tgds is a superset of the language of nested mappings [5]. Since every SO tgd specified for relational schemas can be transformed into a corresponding *GeRoMe* mapping, our mapping language is more expressive than the nested mapping language.

Like path-conjunctive constraints, a *GeRoMe* mapping cannot be nested into another *GeRoMe* mapping. Therefore, a common high-level context has to be repeated in different formulas of a *GeRoMe* mapping. Again, this leads to less efficient execution. However, duplication in target data is overcome by grouping conditions. We may also borrow from the syntax of nested mappings to allow nested mapping definitions.

7 Conclusion

In this paper we introduced a rich mapping representation for mappings between models given in our Generic Role-based Metamodel *GeRoMe* [7]. Our mapping language is closed under composition as it is based on second order tuple-generating dependencies [4]. The mapping language is generic as it can be used to specify mappings between any two models represented in our generic metamodel. Moreover, mappings can be formulated between semistructured models such as XML schemas and are not restricted to flat models like relational schemas. Another feature of the proposed language is that it allows for grouping conditions that enable intensive restructuring of data, a feature also supported by nested mappings [5] which are not as expressive as SO tgds.

To verify the correctness and usefulness of our mapping representation we implemented an adapted version of the composition algorithm for second order tuple-generating dependencies [4]. Furthermore, we developed a tool that exports our mappings to queries and updates in the required data manipulation language and then uses them for data translation. As an example, we showed an algorithm that translates the lhs of a generic mapping to a query in XQuery. The components for mapping export and

execution can be arbitrarily replaced by implementations for the required metamodel. The evaluation showed that both, mapping composition and mapping execution, yield the desired results with a reasonable time performance.

In the future we will develop techniques for visualizing our mappings with the goal to implement a graphical editor for generic, composable, structured extensional mappings. This editor will be integrated into our holistic model management system *GeRoMeSuite* [8]. We will also investigate the relationship between our extensional mappings and intensional mappings that are used for schema integration [16].

Acknowledgements: The work is supported by the Research Cluster on Ultra High-Speed Mobile Information and Communcation UMIC (`www.umic.rwth-aachen.de`).

References

1. P. Atzeni, P. Cappellari, and P. A. Bernstein. Model-independent schema and data translation. In *EDBT*, volume 3896 of *LNCS*, pages 368–385. Springer, 2006.
2. P. A. Bernstein, T. J. Green, S. Melnik, and A. Nash. Implementing mapping composition. In *Proc. VLDB'06*, pages 55–66, Seoul, 2006.
3. P. A. Bernstein, A. Y. Halevy, and R. Pottinger. A vision for management of complex models. *SIGMOD Record*, 29(4):55–63, 2000.
4. R. Fagin, P. G. Kolaitis, L. Popa, and W. C. Tan. Composing schema mappings: Second-order dependencies to the rescue. *ACM Trans. Database Syst.*, 30(4):994–1055, 2005.
5. A. Fuxman, M. A. Hernández, C. T. H. Ho, R. J. Miller, P. Papotti, and L. Popa. Nested mappings: Schema mapping reloaded. In *Proc. VLDB'06*, pages 67–78, Seoul, 2006.
6. M. A. Hernández, R. J. Miller, and L. M. Haas. Clio: A semi-automatic tool for schema mapping. In *Proc. ACM SIGMOD*, page 607, 2001.
7. D. Kensche, C. Quix, M. A. Chatti, and M. Jarke. *GeRoMe*: A generic role based metamodel for model management. *Journal on Data Semantics*, VIII:82–117, 2007.
8. D. Kensche, C. Quix, X. Li, and Y. Li. *GeRoMeSuite*: A system for holistic generic model management. In *Proc. 33rd Int. Conf. on Very Large Data Bases*, 2007. to appear.
9. M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.
10. Y. Li. Composition of mappings for a generic meta model. Master's thesis, RWTH Aachen University, 2007.
11. J. Madhavan and A. Y. Halevy. Composing mappings among data sources. In *Proc. VLDB*, pages 572–583. Morgan Kaufmann, 2003.
12. S. Melnik, P. A. Bernstein, A. Y. Halevy, and E. Rahm. Supporting executable mappings in model management. In *Proc. SIGMOD Conf.*, pages 167–178. ACM Press, 2005.
13. S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A programming platform for generic model management. In *Proc. SIGMOD*, pages 193–204. ACM, 2003.
14. A. Nash, P. A. Bernstein, and S. Melnik. Composition of mappings given by embedded dependencies. In C. Li, editor, *PODS*, pages 172–183. ACM, 2005.
15. L. Popa and V. Tannen. An equational chase for path-conjunctive queries, constraints, and views. In *ICDT '99*, pages 39–57, London, UK, 1999. Springer-Verlag.
16. C. Quix, D. Kensche, and X. Li. Generic schema merging. In *Proc. CAiSE'07*, LNCS, pages 127–141. Springer-Verlag, 2007.
17. E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.
18. G. G. Robertson, M. P. Czerwinski, and J. E. Churchill. Visualization of mappings between schemas. In *Proc. SIGCHI*, pages 431–439, 2005.