Generic Schema Mappings for Composition and Query Answering

David Kensche[1], Christoph Quix[1], Xiang Li[1], Yong Li[1], Matthias Jarke[1,2]

[1]*RWTH Aachen University, Informatik 5 (Information Systems), 52056 Aachen, Germany*
[2]*Fraunhofer FIT, Schloss Birlinghoven, 53574 St. Augustin, Germany*
{*kensche,quix,lixiang,liyong,jarke*}*@i5.informatik.rwth-aachen.de*

Abstract

In this article we present extensional mappings, that are based on second order tuple generating dependencies between models in our Generic Role-based Metamodel *GeRoMe*. Our mappings support data translation between heterogeneous models, such as XML Schemas, relational schemas, or OWL ontologies. The mapping language provides grouping functionalities that allow for complete restructuring of data, which is necessary for handling object oriented models and nested data structures such as XML. Furthermore, we present algorithms 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. Furthermore, we address the question how generic schema mappings can be harnessed for answering queries against an integrated global schema.

Key words: Model Management, Schema Mappings, Mapping Composition, Executable Mappings, Data Integration

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 [1]. It should provide operators such as Match that computes a mapping between two models [2], ModelGen that transforms models between modeling languages [3], or Merge that integrates two models based on a mapping in between [4].

An important issue in a model management system is the representation of mappings which can be categorized as *intensional* and *extensional* mappings [5]. Intensional mappings deal with the intended semantics of a model and are used, for example, in schema integration [4]. These mappings interrelate model elements by set relationships such as equality and subset relationships. However, they do not refer explicitly to instances of models and therefore cannot be used for data translation. Extensional mappings define interschema constraints which must hold for all valid instances of the related schemas [6; 7]. If they are specified as source-to-target mappings, they can be used also for data translation, i.e. extracting data from a source and transforming it into a target schema. We denote such mappings as executable mappings. In this article, we will deal only with extensional mappings which are source-to-target as our goal is to have a generic representation for executable mappings.

1.1 Motivation

To motivate our approach for a generic mapping language, consider the data integration scenario of fig. 1. In this example, we want to integrate a relational data source and an XML document into another XML document. The relational source can be mapped via an existing OWL ontology with the integrated XML schema. The first challenge to be addressed in this scenario is the heterogeneity of modeling languages involved. Other approaches for heterogeneous data integration use some form of wrapper to translate the data of the source into a common format, such as relations or XML. The disadvantage of this approach is that it requires additional overhead for data translation in the wrapper and constraints specified in the source schema might be lost due to the limited expressivity of the modeling language for wrapped schemas. Therefore, a mapping language is required which can map between heterogeneous metamodels and which can be translated into executable queries for a specific modeling language.

Another problem indicated in the scenario is the restructuring of nested data. The

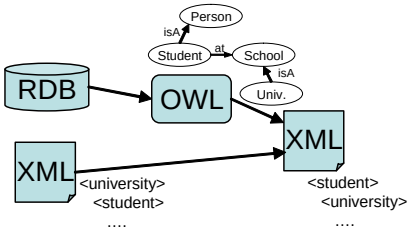

Fig. 1. A simple data integration scenario

XML source groups student elements in a university element, whereas the target reverses this nesting structure. Consequently, a mapping language must support nested data structures as well as such restructuring operations.

A third requirement is the composability of mappings: in the example, there are already mappings from the relational source to the OWL ontology, and from the OWL ontology to the integrated XML schema. If we could compose these two mappings, we could have a direct mapping from the RDB to the XML document.

An extensional mapping can be represented as two queries which are related by some operator (such as equivalent or subset) [7]. 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 [2]. Such mappings can be extended to path morphisms that are defined on a restricted form of relational schemas. Path morphisms can be translated into an executable form but have limited expressivity [8]. Most formal mapping representations also rely on the relational data model, e.g. source-to-target tuple-generating dependencies (s-t tgds, also known as GLAV mappings) [9] or second order tgds (SO tgds) [10]. Tuple generating dependencies (tgds) are dependencies of the form $\forall \mathbf{x}(\varphi(\mathbf{x}) \rightarrow \exists \mathbf{y}\psi(\mathbf{x}, \mathbf{y}))$ where φ and ψ are conjunctions of relation atoms and $\mathbf{x}$ and $\mathbf{y}$ are disjoint sets of variables. If φ contains only relation symbols from the source schema and ψ contains only relation symbols from the target schema, the dependency is called a source-to-target tgd (s-t tgd). Second order tgds (SO tgds) are in our context also source-to-target and have the form $\exists \mathbf{f}((\forall \mathbf{x_1}(\varphi_1 \rightarrow \psi_1) \wedge \ldots \wedge (\forall \mathbf{x_n}(\varphi_n \rightarrow \psi_n))$. Each φ_i is a conjunction of relation atoms based on $\mathbf{x_i}$ and equalities based on $\mathbf{x_i}$ and $\mathbf{f}$, and each ψ_i is a conjunction of relation atoms based on terms of $\mathbf{x_i}$ and $\mathbf{f}$. Thus, the existentially quantified variables in s-t tgds are replaced by existentially quantified function symbols. For a nested relational model, a nested mapping language has been proposed [11].

Each mapping representation has its strengths and weaknesses regarding the requirements for a mapping language mentioned above. Similar requirements have been also stated in [1]: (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; (iii) support for the nesting of mappings (to avoid redundant mapping specifications) and nested data structures should be provided; (iv) mappings should have a rich expressiveness while being generic across modeling languages; (v) mappings should support data translation between the instances of the connected models. While each of the mapping representations mentioned before fulfills some of these requirements for the (nested) relational model, they fail at being generic as they do not take into account other modeling languages.

1.2 Contributions

The main contribution of this article is the definition of 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, using a generic mapping representation, questions such as composability, invertability, decidability, and executability have to be addressed only *once* for the generic mapping representation and do not have to be reconsidered for each combination of mapping and modeling language. The mapping language presented in this paper is based on second order tgds for which a composition algorithm has been defined [10]. This algorithm has been adapted to our mapping language and complemented by optimization steps to deal with the exponential complexity of the algorithm.

A prerequisite for a generic representation of mappings is a generic representation of models. Our work is based on the role based generic metamodel *GeRoMe* [12]. 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* [13]. *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*, which can also translate mappings into a specific data manipulation language such as SQL. This generation of executable queries and update statements for SQL and XML is also presented in this article.

The generic metamodel allows us to represent arbitrary model structures, e.g. flat relational tables as well as nested XML documents. Thus, the generic mapping language supports also nested structures and provides grouping and nesting functionality. Another contribution of this paper is the application of our generic mappings in data integration by using them for query answering.

The contributions of our work define also the structure of the paper. After providing some background information on *GeRoMe* in section 2, we will define in section 3 a *generic mapping representation* based on the semantics of *GeRoMe*. 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. Section 7 then addresses *query answering* in data integration systems. In section 8, we discuss our approach and compare it with other related work. Section 9 concludes our paper and gives an outlook on future work.

This paper is an extended version of our previous paper [14]. Compared to the

previous version, this paper provides an updated version of the mapping definition, explains this in more detail using more examples, provides a proof of the composition algorithm, explains the query and update generation in more detail, updates the evaluation results, and adds the section about query answering.

2 Background

The Generic Role based Metamodel *GeRoMe* was presented in [12], including a formalization and a definition of the semantics of *GeRoMe*. As this representation forms the basis for our mapping representation presented in section 3, we briefly summarize it in this section by using an example.

2.1 The Generic Metamodel GeRoMe

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. Figure 3 shows an example of a *GeRoMe* model of the XML Schema shown in figure 2.

```
<xsd:schema>
  <xsd:element name="University" type="UniType"/>
  <xsd:complexType name="UniType">
    <xsd:sequence>
      <xsd:element name="Student" type="StudType"/>
    </xsd:sequence>
    <xsd:attribute name="uname" type="xsd:string"/>
  </xsd:complexType>
  <xsd:complexType name="StudType">
    <xsd:attribute name="sname" type="xsd:string"/>
    <xsd:attribute name="ID" type="xsd:integer"/>
  </xsd:complexType>
</xsd:schema>
```

Fig. 2. XML Schema for universities and students

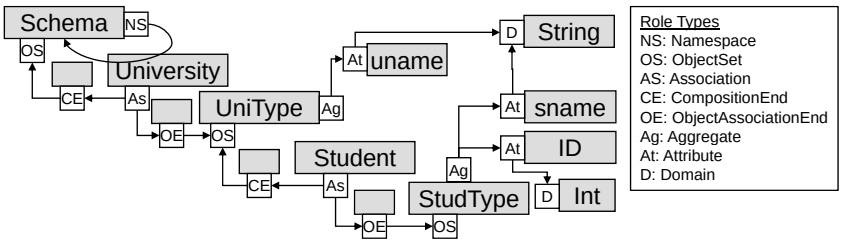

Fig. 3. *GeRoMe* representation of the XML schema in fig. 2

The gray boxes in figure 3 denote model elements, the attached white boxes represent the roles played by the model elements. These roles are instances of a fixed set of role types that may be used in *GeRoMe* models. These role types are actually the available modeling constructs. Rule based import operators [15] take care that model elements play roles according to the meaning of their native modeling constructs. 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 parent complex type and the nested type. This is true for all elements except those that are allowed as the root element of a document. In *GeRoMe*, the definition of a root element is an association between the schema node and the element's complex type, as there is no complex type in which the root element is nested. In the example, the element `University` is an association between the model element `Schema` and the complex type `UniType` [1]. 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 anonymous association ends are required for several reasons. Firstly, since *GeRoMe* is a *generic* metamodel, we must take care that comparable modeling constructs from different modeling languages are represented uniformly. For instance, in UML, associations may be of degree higher than two, whereas in OWL or XML, instances of types (or classes) are always only connected via binary associations. However, in order to allow uniform handling by model management operators we need to represent all these types of associations in the same way. Secondly, the specialized association end type *CompositionEnd* encodes the nesting structure. Without knowing, which type is composed of which elements we would not be able to tell whether `Student` is nested into `University` or vice versa. Another reason is that association ends can have properties themselves. For example, they may define cardinality constraints (which we omitted from the figure for sake of readability).

The same structure of association and association end roles is used for the element `Student` which is an association between the complex types `UniType` and `StudType`. The two complex types have attributes; therefore, they also play *Aggregate* (Ag) roles which link these model elements to their attributes. The model elements representing attributes play *Attribute* (At) roles which refer also to the types of the attributes which are, in this example, simple domains denoted by the *Domain* (D) role.

[1] XML documents must 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.

Definition 1 (*GeRoMe* model) *Formally, the* GeRoMe *metamodel is defined by a set of role types* $\mathcal{R} = \{r_1, \ldots, r_n\}$ *and a set of property types* $\mathcal{P} = \{p_1, \ldots, p_m\}$ *which can be applied to role types.* $\mathcal{V}$ *denotes a set of atomic values which may be used as property values. A model* M *represented in* GeRoMe *is defined by a tuple* $M = \langle E, R, type, plays, property \rangle$*, where*

- $E = \{e_1, \ldots, e_k\}$ *is a set of model elements,*
- $R = \{o_1, \ldots, o_p\}$ *is a set of roles,*
- $type : R \rightarrow \mathcal{R}$ *is a total function that assigns exactly one role type to each role,*
- $plays \subseteq (E \cup R) \times R$ *represents the aforementioned relation between model elements (or roles) and roles,*
- $property \subseteq (R \times \mathcal{P}) \times (R \cup \mathcal{V} \cup E)$ *represents the property values of a role (i.e. properties may also refer to other roles and model elements).*

The sets sets $\mathcal{R}$ and $\mathcal{P}$ are actually the modeling constructs that can be used in a *GeRoMe* model. Consequently, they are fixed. $\mathcal{R}$ is the set of available role types (e.g. As, OE, CE, Ag, etc. as introduced in the example of fig. 3) and $\mathcal{P}$ is the set of properties of these role types which are either simple properties or connect roles to each other. Examples of property types are the *name* property of a named model element (as opposed to an anonymous model element) or the *participator* property of an *AssociationEnd* that connects the association end to the participating *ObjectSet* role. In EER models this could be any number of association ends, whereas XML elements always connect their own (nested) type to the containing complex type.

It is important to emphasize that the representation of models in *GeRoMe* is not to be used by end users for modeling schemas. Instead, it is a representation employed internally by *GeRoMeSuite*, with the goal to generically provide more information to model management operators than a simple graph based model.

2.2 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. Data instances are also used at the model level, e.g. as default values or boundaries of a type defined by an interval. However, the main goal of the formal semantics is the formal definition of executable mappings between models. Our mappings are second-order tuple generating dependencies (SO tgds), which require 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 *AssociationEnd* as

```
<University uname="RWTH">
  <Student sname="John"
           ID="123"/>
</University>
```

$inst(\#0, \texttt{Schema})$,
$inst(\#1, \texttt{UniType}), av(\#1, \texttt{uname}, \texttt{'RWTH'})$,
$inst(\#2, \texttt{StudType})$,
$av(\#2, \texttt{sname}, \texttt{'John'}), av(\#2, \texttt{ID}, 123)$,
$inst(\#3, \texttt{University}), inst(\#4, \texttt{Student})$,
$part(\#3, \texttt{parent}_U, \#0), part(\#3, \texttt{child}_U, \#1)$,
$part(\#4, \texttt{parent}_S, \#1), part(\#4, \texttt{child}_S, \#2)$

Fig. 4. XML document and its representation as *GeRoMe* instance

the roles which essentially express the structure of instances.

Definition 2 (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:*

- *for every object (represented by the abstract identifier id_i) which is an instance of model element e: $inst(id_i, e) \in \mathcal{D}_M$,*
- *for every element e playing a* Domain *role and for all values $v \in \mathcal{A}$ in this domain: $\{value(id_i, v), inst(id_i, e)\} \subseteq \mathcal{D}_M$ (id_i is an abstract ID of an object representing the value v).*
- *for every element e playing an* Aggregate *role, having the attribute a, and the instance id_i has the object id_v for that attribute as value: $attr(id_i, a, id_v) \in \mathcal{D}_M$.*
- *for every model element e playing an* Association *role in which the object with abstract identifier id_o participates for the association end ae: $part(id_i, ae, id_o) \in \mathcal{D}_M$.*
- *There are no other elements 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. 3, an instance containing a university and a student is defined as show in fig. 4. 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 label the association ends with "parent" and "child" to make clear which association end is referred to. However, in practice this information is encoded in the underlying model. 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 end 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. As we will see, the application of this representation to SO tgds can significantly improve the expressiveness of SO tgds.

3 Formal Definition of *GeRoMe* Mappings

Mapping language and modeling language are closely related. We use the formal representation of *GeRoMe* instances as defined in the previous section as basis for our mapping language. This will be applied to second order tgds, which gives a rich expressive mapping language which still maintains features such as composability and executability. Furthermore, the mapping language is generic across several modeling languages. This allows for a generic implementation of model management operators which deal with these mappings.

The main feature of our mapping language is that we use reification to describe data structures by introducing abstract individuals to denote the instances of model elements. In definition 2 and figure 4 these abstract individuals were denoted by abstract identifiers. As will be seen in the following definition and the subsequent example, our mapping language uses so-called abstract variables and abstract function terms to resemble the abstract individuals. Since each instance of a model element is represented by an abstract variable or abstract function term, we can define references between instances using associations and their association ends. This allows us to define also tree- and graph-structured schemas as source and target of mappings. This is not possible in strictly relational SO tgds. Since *GeRoMe* models associations as model elements (instead of properties), its formal semantics and the mapping language reifies them as well. Therefore, the generic mapping language can map associations of arbitrary degree. Besides providing flexibility in describing the structure of instances, abstract functions enable grouping functionality as will discuss in section 3.3. Therefore, we can formulate mappings between any two models that can be described as a *GeRoMe* model. However, although the reified model elements are represented as predicates, they cannot be simply interpreted as relations.

3.1 Definition

Based on the formal definition of *GeRoMe* instances, we extend the definition of SO tgds as a mapping between two relational schemas in [10] to the definition of a mapping between two *GeRoMe* models:

Definition 3 (*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 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*

(1) $\mathbf{f}$ *is a collection of function symbols,*
(2) each $\mathbf{x_i}$ *is a collection of variables,*

(3) *each φ_i is a conjunction of atomic predicates over constants defined in $\mathbf{S}$ and variables and/or equalities.*
(4) *each ψ_i is a conjunction of atomic predicates over constants defined on $\mathbf{T}$, variables, and function symbols.*
(5) *Valid atomic predicates are those defined in def. 2.*
(6) *We require that mappings are source-to-target, i.e. the second arguments of $inst$, $attr$ and $part$ predicates are constants referring only to*
   - *model elements in $\mathbf{S}$ (for predicates in φ_i)*
   - *model elements in $\mathbf{T}$ (for predicates in ψ_i)*
(7) *Each set $\mathbf{x_i}$ can be partitioned into a set of abstract variables $\mathbf{x_{i,a}}$ and a set of concrete variables $\mathbf{x_{i,c}}$.*
   - *Variables in $\mathbf{x_{i,a}}$ may occur only in φ_i, either as the first argument of any predicate, or as the third argument of $part$ predicates.*
   - *Variables in $\mathbf{x_{i,c}}$ may occur only as the second argument of $value$ predicates in φ_i or ψ_i, in equalities in φ_i, and in any function term in φ_i or ψ_i.*
(8) *Analogously, $\mathbf{f}$ can be partitioned into a set of abstract functions $\mathbf{f_a}$ and a set of concrete functions $\mathbf{f_c}$.*
   - *Function symbols in $\mathbf{f_a}$ may occur only in ψ_i, either as the first argument of any predicate, or as the third argument of $part$ predicates.*
   - *Function symbols in $\mathbf{f_c}$ may occur only as the second argument of $value$ predicates in ψ_i or in equalities in φ_i.*
(9) *The second component of a $value$ predicate may also be a constant.*
(10) *Equalities are of the form $t = t'$ where t and t' are terms over $\mathbf{x_i}$, $\mathbf{f}$ and constants. Furthermore, the same safety conditions apply to variables as in the original definition of SO tgds [10]: each variable in $\mathbf{x_i}$ appears in some predicate in φ_i.*

Please note that the definition requires the constants identifying model elements to be from $\mathbf{S}$ on the source side only, and to be from $\mathbf{T}$ only on the target side. Thus, we only consider source-to-target dependencies in this work.

The intuition behind the partition of variables is that variables in $\mathbf{x_{i,c}}$ play the role of distinguished variables in conventional SO tgds. Concrete variables may be shared on both sides of the implication whereas the abstract variables $\mathbf{x_{i,a}}$ occur only on the source side. When executing a mapping, values are bound to the variables in $\mathbf{x_{i,c}}$. Thereby, values are "transferred" from the source to the target.

On the other hand, abstract variables on the source side replace the abstract identifiers used in definition 2 and fig. 4. They represent the instances of model elements such as relations or XML elements. By using reified predicates, we can query for particular features of these elements. As abstract variables refer to instances of the source schema, it does not make sense to use them on the target side. The identification of instances at the target side is achieved by using the function symbols in $\mathbf{f_a}$, instead. Abstract functions are handled like Skolem functions during execution. That is, they cannot be executed but they are only instantiated as ground terms in order to

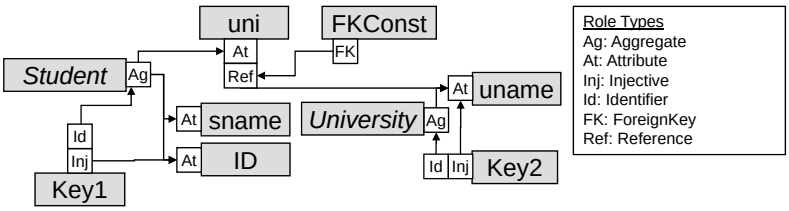

Fig. 5. *GeRoMe* representation of a relational schema

uniquely identify instances of model elements on the rhs. Consequently, the choice of arguments for an abstract function determines the identification of objects on the target side, and thereby the grouping behaviour defined by the mapping. In addition, since the same function symbols can be used in multiple implications, the results of these implications can be merged due to the interpretation as Skolem functions.

In the same way as $\mathbf{x_{i,c}}$ are the distinguished variables of conventional SO tgds, $\mathbf{f_c}$ plays the role of functions available in these SO tgds. They have to be executed in order to transform source data into target data, such as concatenating first and last name, or at least they must be interpreted as Skolem functions in order to generate distinct null values depending on the values bound to distinguished variables.

The grouping functionality provided by abstract functions is not available in conventional SO tgds as their relational instances are not identified by Skolem functions. Also, on the source side of mappings the flexibility regarding the structure of instances, which is enabled by $part$ predicates and the reification style of our mappings, is not provided by conventional SO tgds. However, we will see that our mappings are still closed under composition. For nested mappings [11] on the other hand, which allow grouping in tree structures, it is not known whether they are closed under composition. This is because it is not known how to translate an SO tgd to a nested mapping.

3.2 Example

To show an example of a mapping between models originally represented in two different modeling languages, we define in fig. 5 a *GeRoMe* model representing a relational schema for the university domain. 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 *Injective* (Inj) roles to indicate that an attribute is unique, and *Identifier* (Id) roles to specify the aggregates for which the attributes are the keys. 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.

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

Fig. 6. Mapping from XML Schema in fig. 3 to the relational schema in fig. 5 ($\mathcal{M}_{12}$)

Now, we can define a mapping using the XML schema as source and the relational schema as target (cf. fig. 6). The predicates in the conditional part of the rule correspond to the instance predicates shown in fig. 4, now just with variables instead of constants. The variables o_0 to o_4 represent abstract identifiers (they are *abstract variables*), their function is to describe (implicitly) the structure of the source data that is queried for. The predicate $inst(o_0, \texttt{Schema})$ defines an abstract variable o_0 that must be bound to an instance of the `Schema`, $inst(o_2, \texttt{UniType})$ defines an abstract variable o_2 of type `UniType`, whereas $inst(o_1, \texttt{University})$ defines a variable o_1 that must be bound to an instance of the `University` element. The av predicates define variables for the attribute values of the complex type instances o_2 and o_4, whereas the $part$ predicates are used to constrain the structure of the elements. In the example the student element o_3 is nested under the university element o_1. The predicate $part(o_1, \texttt{child}_U, o_2)$ defines the complex type instance o_2 (of type `UniType`) to be the participator in the association end $\texttt{child}_U$. This association end in the underlying GeRoMe model denotes the link from the XML element `University` to its type. The variable o_2 is also defined as the participator in the association end $\texttt{parent}_S$ which links the variable o_3 (denoting an instance of the XML element `Student`) to its containing complex type instance. In other approaches for mapping representation (e.g. [11]) such structures are represented by nesting different sub-expressions of a query. Although nested mappings are easier to read, they are strictly less expressive than SO tgds [11]. 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 (f and g are *abstract functions* which generate abstract identifiers). 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 into the target.

The difference of our mappings from conventional SO tgds is that our mappings do not map relations to other relations but instead use reification to represent mappings between graph structures. Instead of mapping to the relation $Student(s, i, u)$ we use the *set* of predicates from the right hand side of the above mapping to make statements about an object $g(i)$ which is an instance of that relation. On the left hand side of this mapping, $part$ predicates define relationships between instances of XML complex types and XML elements. Whereas this example features an XML Schema, such reified statements about model elements can be utilized to describe arbitrary data structures, such as relationships between class instances in an object oriented model. Furthermore, by using the $part$ predicates, the mapping language allows us to define even *n-ary* relationships between model elements as there is no limitation to the number of association ends defined for an association. This could be employed for mappings between object oriented models, which allow relationships of degree higher than two. Additionally, the $part$ predicates together with the Skolem functions occurring on the right hand side of our reified mappings allow for arbitrary grouping based on the values bound to concrete variables. This is done by using the concrete variables as arguments of the Skolem functions and, in doing so, defining the instances of model elements.

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 concrete ("normal") functions for value conversions or other types of data transformation (e.g. concatenation of first and last names). While executing a mapping, these functions must be actually evaluated to get the value which has to be inserted into the target.

3.3 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 data translation, we use an example that transforms relational data into XML. The relational schema is as in fig. 5 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. In addition, the courses taken by a student at

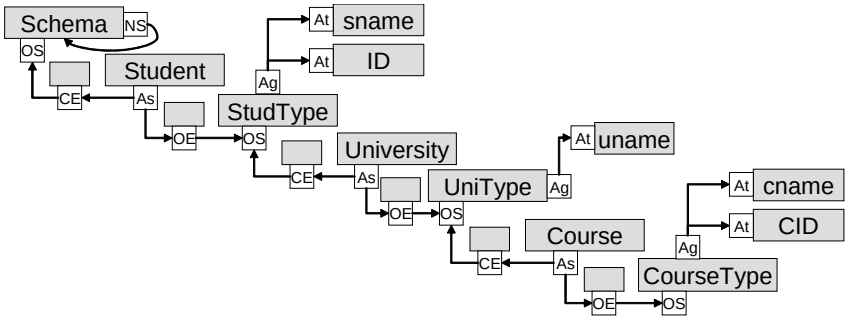

Fig. 7. XML Schema with reversed nesting structure

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

Fig. 8. Mapping from an extended version of the relational schema in fig. 5 to the XML Schema from fig. 7

a university can be stored in the XML document as the innermost XML element. The *GeRoMe* model of this XML schema is shown in fig. 7. The mapping between the updated relational schema and XML schema is shown in fig. 8. As the relational schema does not provide information about courses, the mapping does not reference the `Course` or `CourseType`.

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. $f'(i)$ represents an instance of `StudType` 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, whereas we could use just u as the sole argument of f in the case of the relational model to generate an instance of the `University` table (fig. 6), we have to include the additional argument i in the XML Schema case. This is because each `StudType` instance has its own nested `University` element. Therefore, we may have multiple `University` elements 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`.

3.4 Visualization and Editing of 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* [13], 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 [16]) 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 extensional mapping editor also visualizes models as trees. To overcome the problem of limited expressivity of trees we provide multiple ways to convert a model into a tree, which can be seen as different views on the same model. The editor must furthermore allow for definition of selection and join predicates as well as implementations of concrete functions for data conversion. Still, an appropriate visual representation of complex mappings is an active research area [17], and we have to evaluate whether our design will be accepted by users.

4 Mapping Composition

Composition of mappings is required for many model management tasks [18]. 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. This application will be also shown for our mappings in section 7. 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 schemas.

4.1 Semantics of Mapping Composition

In general, the problem of composing mappings has the following formulation: given a mapping $\mathcal{M}_{12}$ from model $\mathbf{S}_1$ to model $\mathbf{S}_2$, and a mapping $\mathcal{M}_{23}$ from model $\mathbf{S}_2$ to model $\mathbf{S}_3$, derive a mapping $\mathcal{M}_{13}$ from model $\mathbf{S}_1$ to model $\mathbf{S}_3$ that is equivalent to the successive application of $\mathcal{M}_{12}$ and $\mathcal{M}_{23}$ [10].

Mapping composition has been studied only for mappings which use the Relational Data Model as basis. Fagin et al. [10] proposed a semantics of the Compose operator that is defined over instance spaces of schema mappings. To this effect, $\mathcal{M}_{13}$ is the composition of $\mathcal{M}_{12}$ and $\mathcal{M}_{23}$ means that the instance space of $\mathcal{M}_{13}$ is the set-theoretical composition of the instance spaces of $\mathcal{M}_{12}$ and $\mathcal{M}_{23}$. Under this semantics, which we will also adopt in this article, the composition mapping $\mathcal{M}_{13}$ is unique up to logical equivalence.

According to [10], the composition of two mappings expressed as SO tgds can be also expressed as an SO tgd. In addition, the algorithm proposed in [10] 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.

However, due to the extensions made in the definition of our generic SO tgds we have to do some adaptations to the original composition algorithm. In the following, we will first show this adaptation of the algorithm of [10] 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 mappings or models.

4.2 Composition Algorithm

The composition algorithm shown in fig. 9 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 $P.c(\mathbf{y})$ on 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 $P.c(\mathbf{t}_i)$.

To illustrate the composition algorithm, consider the mapping given in fig. 10 which maps the simple relational schema from fig. 5 to the relational schema with the additional `Studies` relation. This mapping (called $\mathcal{M}_{23}$ in the following) will be composed with the mapping from fig. 6 to which we will refer as $\mathcal{M}_{12}$. As an example, we will replace the predicate $av(p_2, \texttt{uni}, x)$ in $\mathcal{M}_{23}$, which can be unified with $av(g(i), \texttt{uni}, u)$ from the rhs of $\mathcal{M}_{12}$. The equalities which have to be introduced are $p_2 = g(i) \wedge x = u$. Now, the predicate $av(p_2, \texttt{uni}, x)$ can be replaced with the lhs of $\mathcal{M}_{12}$. This leads to the following intermediate result: [2]

$$
\begin{aligned}
&\mathit{inst}(p_1, \texttt{University}_r) \wedge \mathit{inst}(p_2, \texttt{Student}_r) \wedge \\
&\mathit{av}(p_1, \texttt{name}, x) \wedge \mathit{av}(p_2, \texttt{name}, y) \wedge \mathit{av}(p_2, \texttt{ID}, z) \wedge \\
&\mathit{inst}(o_0, \texttt{Schema}) \wedge \\
&\mathit{inst}(o_1, \texttt{University}_x) \wedge \mathit{part}(o_1, \texttt{parent}_U, o_0) \wedge \mathit{part}(o_1, \texttt{child}_U, o_2) \wedge \\
&\mathit{inst}(o_3, \texttt{Student}_x) \wedge \mathit{part}(o_3, \texttt{parent}_S, o_2) \wedge \mathit{part}(o_3, \texttt{child}_S, o_4) \wedge \\
&\mathit{av}(o_2, \texttt{name}, u) \wedge \mathit{av}(o_4, \texttt{name}, s) \wedge \mathit{av}(o_4, \texttt{ID}, i) \\
&p_2 = g(i) \wedge x = u \rightarrow \phi
\end{aligned}
$$

The first two lines are the original predicates of $\mathcal{M}_{23}$ which have not been replaced, yet. The next four lines are the lhs of $\mathcal{M}_{12}$ and the last line contains the equalities which have to be introduced to make the "join" between these two parts. ϕ represents the unchanged rhs of $\mathcal{M}_{23}$.

These steps have to be done for each predicate on the lhs of $\mathcal{M}_{23}$. As there can be multiple implications in $\mathcal{M}_{12}$ which match a predicate of $\mathcal{M}_{23}$, the size of the composed mapping may grow exponentially during this step. Please note also, that the predicates of the lhs of $\mathcal{M}_{12}$ will be repeated six times (once for each predicate on the lhs of $\mathcal{M}_{23}$) with different variables. This is due to the fact that the algorithm considers only one predicate at a time and does not replace a set of predicates. In the next section, we will explain how we can simplify the composition result, by using logical transformations which transform the mapping into an equivalent simpler mapping, and by using the constraints of the schema to remove redundant predicates.

[2] We added indices r and x to the model elements to indicate whether they refer to the relational schema or to the XML schema. Furthermore, we abstained from renaming the predicates in order to avoid introducing a new notation.

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:
for each predicate $P(x, c, y)$ (or $P(x, c)$) in Σ_{12} (and Σ_{23}, respectively)
where $P \in \{inst, attr, av, part\}$
replace $P(x, c, y)$ with $P.c(x, y)$ and replace $P(x, c)$ with $P.c(x)$)
endfor
for each implication of the form $\phi \rightarrow p_1 \wedge \ldots \wedge p_n$ in Σ_{12}
replace the implication with set of implications $\phi \rightarrow p_1, \ldots, \phi \rightarrow p_n$
endfor
put the resulting implications into $\mathcal{S}_{12}$ and $\mathcal{S}_{23}$, respectively.

Composition:
while there are implications of the form $\chi = \psi \rightarrow \sigma \in \mathcal{S}_{23}$
where ψ contains a predicate $P.c(\mathbf{y})$ which refers to $\mathbf{S}_2$:
for each implication $\phi(\mathbf{x}) \rightarrow P.c(\mathbf{t}) \in \mathcal{S}_{12}$
create a copy $\phi(\mathbf{x_i}) \rightarrow P.c(\mathbf{t_i})$ using new variable names
add a new implication $\chi_i = \psi \rightarrow \sigma$ to $\mathcal{S}_{23}$
in χ_i 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 $\mathbf{t_i}$
endfor
remove χ from $\mathcal{S}_{23}$

Optimization I:
for each implication $\chi \in \mathcal{S}_{23}$
repeat
(i) introduce equalities based on abstract functions
(ii) introduce equalities based on schema constraints
(iii) for each implication χ in $\mathcal{S}_{23}$,
select an equality $y = t$,
and replace all occurences of y in χ by t.
endfor
until no changes can be made to the implication
endfor

Optimization II:
for each pair of implications $A_1 \rightarrow C_1 \in \mathcal{S}_{23}$ and $A_2 \rightarrow C_2 \in \mathcal{S}_{23}$
if $A_1 \rightarrow A_2$ and $C_2 \rightarrow C_1$ (containment test)
remove the implication $A_1 \rightarrow C_1$ as it is subsumed by the other implication
endfor

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 in $\mathcal{S}_{23}$ and $\mathbf{z}_i$ being all the variables appearing in χ_i.

Fig. 9. Algorithm **Compose** for *GeRoMe* mappings based on [10]

As a first step towards a simpler result, we apply in the next step some optimizations to each implication. Optimizations at this stage just address single implications and are actually interleaved with the previous composition step. In the example, we

$$
\begin{aligned}
&\exists f', g', h' \quad \forall p_1, p_2, x, y, z \\
&inst(p_1, \texttt{University}) \wedge inst(p_2, \texttt{Student}), \\
&av(p_1, \texttt{name}, x) \wedge av(p_2, \texttt{name}, y) \wedge av(p_2, \texttt{ID}, z) \wedge av(p_2, \texttt{uni}, x) \rightarrow \\
&\quad inst(f'(x), \texttt{University}) \wedge inst(g'(z), \texttt{Student}), inst(h'(x, z), \texttt{Studies}) \wedge \\
&\quad av(f'(x), \texttt{name}, x) \wedge av(g'(z), \texttt{name}, y) \wedge av(g'(z), \texttt{ID}, z) \wedge \\
&\quad av(h'(x, z), \texttt{uni}, x) \wedge av(h'(x, z), \texttt{id}, z)
\end{aligned}
$$

Fig. 10. Mapping between two relational schemas ($\mathcal{M}_{23}$)

just can apply item (iii), thus, we remove the variables which were originally in $\mathcal{M}_{23}$. This reduces the number of equalities in the mapping. In the example shown above, this means that we replace all occurences of p_2 with $g(i)$ and of x with u. Please recall that this step must only be applied after all predicates from the lhs of $\mathcal{M}_{23}$ have been replaced. Consequently, it will replace abstract variables only in equality predicates. The result of such replacements are equalities between terms using abstract functions which can be used to derive further optimizations as we will explain in more detail in section 4.3.

Before we create the final result, we apply another optimization step on the level of mappings as we check for each pair of implications contained in the composed mapping, whether one implication is subsumed by the other implication. If this is the case, the subsumed implication can be removed.

The final step creates the composed mapping as one formula from $\mathcal{S}_{23}$. The following theorem states that the algorithm actually produces a correct result.

Theorem 4 *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 $\mathcal{M}_{13} = \mathcal{M}_{12} \circ \mathcal{M}_{23}$.*

Proof: The proof is based on the correctness of the composition algorithm in [10]. Predicate renaming and optimizations are the major difference between our and Fagin et al.'s composition algorithm. We rename all predicates in the two given mappings before we compose them, and we also rename all predicates in the output mapping after we compose them. For notational purposes, we use Σ'_{12} and Σ'_{23} to denote the formulas that are the results of renaming the predicates in Σ_{12} and Σ_{23} respectively, and we use Σ'_{13} to denote the formulas of Σ_{13} without renaming its predicates. Observe that Σ'_{12} and Σ'_{23} are still SO tgds. Because of the correctness of Fagin et al.'s composition algorithm, the output, which is Σ'_{13}, is an SO tgd and is the composition of Σ'_{12} and Σ'_{23}. Therefore, we only need to prove that we can always get a *GeRoMe* mapping after renaming predicates in Σ'_{13}, and that Σ_{13} is logically equivalent to the output of applying Fagin et al.'s algorithm directly on Σ_{12} and Σ_{23}.

Based on the result of Fagin et al.'s composition algorithm, all predicate names appearing on the lhs of Σ'_{13} come from the predicate names appearing on the lhs of

Σ'_{12}, and all predicate names appearing on the rhs of Σ'_{13} come from the predicate names appearing on the rhs of Σ'_{23}. Observe that, we always get Σ_{12} if we apply the renaming rules in step 4 of our algorithm on Σ'_{12}. Similarly, we always get Σ_{23} if we apply the same rules on Σ'_{23}. After renaming all predicates in Σ'_{13}, predicate names in Σ_{13} are predicate names from Σ_{12} and Σ_{23} only. That is, Σ_{13} is an SO tgd and contains only valid predicates for *GeRoMe* mappings. Thus, the output mapping $\mathcal{M}_{13}$ is always a valid *GeRoMe* mapping.

To prove the logical equivalence, we take the $inst$ predicates as an example and prove that renaming $inst$ predicates does not affect the logical equivalence. Consider the following two $inst$ predicates, $inst(x, const_x)$ on the rhs of an implication in Σ_{12} and $inst(y, const_y)$ on the lhs of an implication in Σ_{23}. After renaming, they become $inst.const_x(x)$ and $inst.const_y(y)$. In the second step of the composition algorithm, we replace $inst.const_y(y)$ with the lhs predicates of an implication whose rhs is $inst.const_x(x)$, only if $const_x$ is the same constant as $const_y$. If we apply the composition algorithm directly on Σ_{12} and Σ_{23}, we would then always replace $inst(y, const_y)$ with lhs predicates of an implication whose right hand side is $inst(x, const_x)$ even if $const_x$ and $const_y$ are not the same. However, in the latter case, we would add $const_x = const_y$ to the conjunction of predicates on the lhs of the result implication. The left hand side of the result implication is always evaluated to false if $const_x$ and $const_y$ are not the same constants. Therefore, we can safely remove all result implications where $const_x$ and $const_y$ are not the same. That is, only implications where $const_x$ and $const_y$ are the same constants remains in the result. This is exactly the same as in the case where we rename predicates before composing them. Similar results apply to $part$, $attr$, $value$ and oid predicates. So, our output mapping is logically equivalent to the output mapping of applying composition without renaming predicates.

Also, the optimizations do not affect the logical semantics of the mapping as they guarantee that the optimized mapping is logically equivalent to the unoptimized mapping as we will discuss in the next section. □

4.3 Semantic Optimization of the Composition Result

The proposed mapping language is capable of representing composable and executable mappings between models in arbitrary modeling languages. This degree of flexibility is due to the usage of reification by the meta predicates of the *GeRoMe* semantics, which allows us to specify the features (properties) of the model elements that are mapped representing arbitrary graph structures.

The composition algorithm has exponential time costs, which depend on the number of implications in the mappings and on the number of predicates in the implications. 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 be inserted multiple times with different variable names. Thus, reification increases both, flexibility of the mapping language and computational costs. Consequently, the composed mapping has on the lhs many similar sets of predicates. Although the result is logically correct, the predicates on the lhs of the composition seem to be duplicated.

Because the increased size of the resulting implications is due to duplicates of predicates using different variable names, we can compensate it by collapsing variables that must be bound to the same objects. This can be done based on the constraints encoded in the model. The optimization steps first consider only each implication of the mapping separately and then perform a containment test between the implications.

A first step in optimizing the composed mapping consists of deriving equalities of concrete variables in the mapping based on equalities of abstract functions. As abstract functions are interpreted only syntactically like Skolem functions, the following conditions hold 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. Therefore, we can remove implications which contain equality predicates 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 ($p(x) \wedge p(y) \wedge x = y \Leftrightarrow p(x)$).

Further optimization steps take the constraints of the models into account. For example, if we know than an attribute $\mathtt{a}$ has maximum cardinality of 1, then we can conclude from $av(x, \mathtt{a}, v_1) \wedge av(x, \mathtt{a}, v_2)$ that $v_1 = v_2$. Another possibility for optimization are uniqueness constraints: if the attribute $\mathtt{a}$ is unique, we can conclude from $av(x, \mathtt{a}, v) \wedge av(y, \mathtt{a}, v)$ that $x = y$. The same ideas can be applied also to associations and $part$ predicates. These steps can be applied iteratively until no changes can be made anymore to an implication.

Finally, a pairwise containment check is applied on all implications that were produced by the composition algorithm. Given two implications $I_1 = A_1 \rightarrow C_1$ and $I_2 = A_2 \rightarrow C_2$ the implication I_1 is contained in I_2 if and only if $A_1 \rightarrow A_2$ and $C_2 \rightarrow C_1$. In this case the implication I_1 can be safely removed from the mapping without changing its set of solutions. The containment test is based on the procedure [19]; as function symbols are used only in a restricted way, it is sufficient to construct one canonical database which might be a counter example for containment.

Furthermore, it is particularly important that this optimization step is interleaved with the composition algorithm. Because the set of implications produced for each predicate replaced by the algorithm serves as input for the next iteration, the number

of implications, which is crucial for the computational costs, grows exponentially depending on the numbers of predicates and implications. Keeping the number of implications low during the composition already, speeds up the composition procedure and reduces the space requirements dramatically. However, in order to speed up the containment check, the preceding steps of collapsing variables have also been implemented interleaved with the composition.

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. 11 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 executer produces variable assignments from the evaluation of the queries against the source data. The update executer 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 executer components have to fit to each other.

5.1 Generating XQuery Queries from Generic Schema Mappings

We now introduce our algorithm for generating XQueries from our generic mappings (cf. fig. 12). However, our tool transforms data arbitrarily between relational and

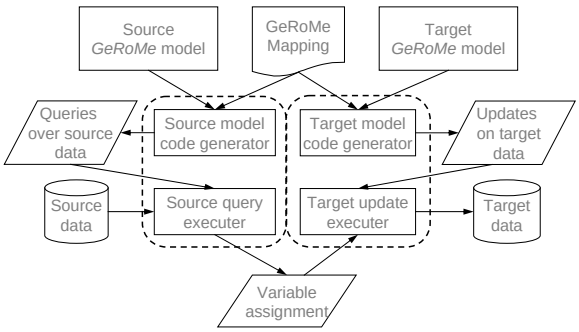

Fig. 11. The architecture of the data translation tool

Input: The lhs of implication χ in an SO tgd Σ with source *GeRoMe* model $\mathbf{S}$ and corresponding XML Schema $\mathbf{S}_{XS}$
Output: An XQuery query over $\mathbf{S}_{XS}$.
Initialization: $T = Open = Close = R = P = \emptyset$
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 $(S, \text{“/”})$ to $Open$ and set $T = (S, \text{“/”}, null)[]$ where $[]$ is an empty list.
Construct element hierarchy T:
  repeat
    let $(X, path) \in Open$ and
    let $t = (X, path, label)[C]$ be the subtree in T with $path$ as its second component
    for each subformula $inst(Id, name) \wedge part(Id, ae_1, X) \wedge part(Id, ae_2, Y)$ in χ
        where Id, X, and Y are variable symbols, $name$, ae_1 and ae_2 are model elements,
        and there is no $path'$ with $(Y, path') \in Close$,
      add $(Y, path + name/)$ to $Open$
      set $t = (X, path, label)[C|(Y, path + name/, name)[]]$
      remove $(X, path)$ from $Open$ and add it to $Close$
    endfor
  until $Open = \emptyset$
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
      $(X, path, label) \in T$,
    add $(V, \text{“\$}X/@a\text{”})$ to R.
  endfor
  for each term $value(X, V)$ on the lhs of χ,
      where X, V are variable symbols and $(X, path, label) \in T$,
    add $(V, \text{“\$}X/\text{text()”})$ to R.
  endfor
Construct condition set P:
  for each predicate $V_1 = V_2$ on the lhs of χ with $(V_1, path_1) \in R \wedge (V_2, path_2) \in R$ (explicit join condition)
    add “$path_1$ **eq** $path_2$” to P
  endfor
  for each $(V, path_1) \in R \wedge (V, path_2) \in R$ (specifies an implicit join condition)
    add “$path_1$ **eq** $path_2$” to P
  endfor
  for each term $value(V, c)$ on the lhs of χ, where c is a constant
    add “V **eq** c” to P
  endfor
Produce XQuery:
  let $T = (doc, \text{“/”}, null)[(e_{1,1}, p_{1,1}, l_{1,1})[(e_{2,1}, p_{2,1}, l_{2,1})[\ldots], \ldots, (e_{2,k_2}, p_{2,k_2}, l_{2,k_2})[\ldots]]]$
  let $(v_1, path_1), (v_2, path_2), \ldots, (v_n, path_n) \in R$
  let $p_1, p_2, \ldots, p_n \in P$.
  Then produce the following XQuery query for χ:
  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. 12. Algorithm **XQueryGen**

XML schemas; these generation and execution components can also be replaced by components that handle other metamodels (e.g. OWL or UML). The construction of SQL queries and updates will be explained later in this section.

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. 6 that can be used to query the document in fig. 4. We first have to identify the variable which refers to the document element. As the lhs of the mapping contains a term $inst(o_0, \texttt{Schema})$,

o_0 is the variable we are looking for. Therefore, we add $(o_0, /)$ to $Open$ and put $(o_0, /, null)[]$ as the root into T (where [] is a yet empty list denoting the children of the root node).

Now, we construct the element hierarchy T. For $(o_0, /)$ in $Open$ the required pattern is satisfied by the subformula $inst(o_1, \texttt{University}) \wedge part(o_1, \texttt{parent}_U, o_0) \wedge part(o_1, \texttt{child}_U, o_2)$. We add $(o_2, /\texttt{University})$ to $Open$ and add $(o_2, /\texttt{University}, \texttt{University})$ to T as a child of $(o_0, /, null)$. As no other subformula satisfies the pattern, we remove $(o_0, /)$ from $Open$ and add it to $Close$. We get $Open = \{(o_2, /\texttt{University})\}, T = (o_0, /, null)[(o_2, /\texttt{University}, \texttt{University})]$ and $Close = \{(o_0, /)\}$. We repeat the step for $(o_2, /\texttt{University})$. The result for T after this step is $(o_0, /, null)[(o_2, /\texttt{University}, \texttt{University})[(o_4, /\texttt{University/Student}, \texttt{Student})]]$. No elements are added in the last step.

The 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 $(u, \$o_2/@uname)$, $(s, \$o_4/@sname)$ and $(i, \$o_4/@ID)$ 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     $o2 in fn:doc(_fname)/University
        for $o4 in $o2/Student
return <result>
          <u>$o2/@uname</u> <s>$o4/@sname</s> <i>$o4/@ID</i>
       </result>
```

5.2 Generating and Executing SQL Queries and Updates

Generating queries and updates in SQL are both very similar. As update generation also includes the phase in which the values retrieved from the source side have to be inserted into the generated SQL insert commands, we focus on this part. The algorithm in fig. 13 receives as input the rhs of an implication whose target model is a SQL Schema. From this it generates a set of parametrized SQL update statements.

For a given implication it first computes the set of tables for which records are generated. By handling each predicate $inst(f(\mathbf{x}), a)$ individually, it is possible to generate multiple records of one table. Then it fetches the set of triples $av(f(\mathbf{x}), attr, term)$ from the implication where $term$ specifies the concrete value of column $attr$ for record $f(\mathbf{x})$. In the last step this information is used to construct the insert statement. The statements generated are of the form "**insert into** ... **on duplicate key update** ...". This is a variant of the insert operation for the MySQL database management system which inserts records, or if a record with the given key already exists, the existing record will be updated. Using this syntax we can allow multiple implications to contribute to one record in the database. Please note that we need to assume that

Input: The rhs of an implication $\chi \in \Sigma$ taken from a *GeRoMe* mapping $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ where $\mathbf{T}_{SQL}$ is the SQL Schema corresponding to the *GeRoMe* model $\mathbf{T}$.

Output: A set of parameterized SQL update statements over $\mathbf{T}_{SQL}$

Initialization: $T = C = K = \emptyset$

Construct the table set T:
for each predicate $inst(f(\mathbf{x}), a)$ on the rhs of χ, where
f is an abstract function,
$\mathbf{x}$ is a vector of concrete variables, and
a is a model element in $\mathbf{T}$ that plays an Aggregate role,
add $(f(\mathbf{x}), a)$ to T
endfor

Construct column set C and key column set K:
for each $av(f(\mathbf{x}), attr, term)$ predicate on the rhs of χ, where
f is an abstract function,
$\mathbf{x}$ is a vector of concrete variables,
$attr$ is a model element in $\mathbf{T}$ that plays the role of an Attribute of table t, and
$term$ is a term over $\mathbf{x}_{i,c}$ and $\mathbf{f}_c$.
if $\mathbf{T}$ declares $attr$ to be a key component for table t
add $(f(\mathbf{x}), attr, term)$ to K.
else
add $(f(\mathbf{x}), attr, term)$ to C
endif
endfor

Construct the update:
for each $(f(\mathbf{x}), a) \in T$:
let $K_{f(\mathbf{x})} = \{t | t = (f(\mathbf{x}), attr, term) \in K\} =$
$\{(f(\mathbf{x}), attr_1, term_1), \ldots, (f(\mathbf{x}), attr_m, term_m)\}$
let $C_{f(\mathbf{x})} = \{t | t = (f(\mathbf{x}), attr, term) \in C\} =$
$\{(f(\mathbf{x}), attr_{1_a}, term_{1_a}), \ldots, (f(\mathbf{x}), attr_{n_a}, term_{n_a})\}$
Then construct the following SQL insert statement:
insert into $a(attr_1, \ldots, attr_m, attr_{1_a}, \ldots, attr_{n_a})$
values $(?term_1, \ldots, ?term_m, ?term_{1_a}, \ldots, ?term_{n_a})$
on duplicate key update $attr_{1_a} = ?term_{1_a}, \ldots, attr_{n_a} = ?term_{n_a}$
endfor

Fig. 13. Algorithm **SQLUpdateGen**

each implication generates values for all key components of generated tuples to do so. This is a reasonable assumption because there must be a way to identify tuples on the target side if multiple tuples have to be merged.

Fig. 14 depicts the execution procedure for the SQL update statements. Given a set of update statements, this procedure interprets the terms used in the implication in the place of concrete values. It uses the implementations assigned to function symbols in $\mathbf{f}_c$ and the variable assignments produced by executing the query for the lhs of the implication. Given this information the denotation of each term on the rhs of the implication can be computed. Replacing the corresponding placeholders in the

Input: A set of parameterized SQL insert / update statements $\{U_1, \ldots, U_n\}$ generated from the rhs of implication $\chi \in \Sigma$ by algorithm 13
A set of variable assignments for all concrete variables in χ
An interpretation $\mathcal{I}$ that defines the semantics of all functions $f \in \mathbf{f}_c$

Output: A set of SQL update statements U without parameters over $\mathbf{T}_{SQL}$

Set update parameters:
for each update statement U_i and for each variable assignment μ
for each term $term$ in U_i
let $d = ||term||_{\mathcal{I},\mu}$ be the denotation of that term
replace $term$ in U_i with its denotation d
endfor
add the 'ground' update statement to U
endfor

Fig. 14. Algorithm **SQLUpdateExec**

update statements gives a set of "ground" update statements that can be executed.

6 Evaluation of Mapping Composition and Execution

To evaluate mapping composition we used fourteen composition problems which were taken from recent literature [10], or which were manually defined by us. The mappings from literature had to be formulated manually in our representation before composing them. The results of composition were logically equivalent to the results documented in the literature. All tests were run on a Windows XP machine with a Pentium M processor with 1.86GHz CPU and 700MB heap space.

In fig. 15, we list, for each test case, the number of implications in Σ_{12}, Σ_{23} (I_{12} and I_{23}) and the average numbers of predicates in the lhs and rhs of the implications ($P_{l,12}$, $P_{l,23}$, $P_{r,12}$ and $P_{r,23}$). All these properties of the input mappings influence the computation time for composition and the quality of the result which we captured with the number of implications in Σ_{13} (I_{13}) and the average number of predicates in the lhs of its implication ($P_{l,13}$).

The upper bound of the number of implications in the unoptimized composition is $O(\sum_i(I^{P_i}))$, where I is the number of implications in the normalized Σ_{12} ($I = I_{12} \cdot P_{r,12}$) and P_i is the number of predicates in the lhs of implication i 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 matching implications in $\mathcal{S}_{23}$. Since 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. Repeating the same reasoning for every source predicate in Σ_{23} will lead to the stated upper bound.

Thus, with increasing numbers of implications in mapping $\mathcal{M}_{12}$ and increasing num-

bers of predicates in the mappings, the number of generated implications increases dramatically for the unoptimized case. This is because the composition algorithm generates a new implication for each possible replacement of a predicate in the lhs of $\mathcal{M}_{23}$. For the conferences example used in [13] (example 8) the result contained 972 implications in the unoptimized case and only one implication in the optimized case. For composition task 14, the unoptimized composition mapping contained 9216 implications as opposed to 24 implications in the optimized case. The number of implications is crucial as a high number of implications in an intermediate result also produces a higher number of implications after the next predicate replacement. Thus, the optimization algorithm improves the computation time by keeping the number of implications low during the composition. Furthermore, although a mapping with thousands of implications may be logically correct, it is practically unusable. Thus, the optimization procedure is needed to improve the computation time of more complex composition tasks, but also to produce an actually useful result. Furthermore, the computation needs about 18 seconds in the unoptimized case, whereas doing the same composition with optimization returns the composed mapping after 1.2 seconds.

Similar to the number of implications, the number of predicates per implication in the composition mapping influences the practical usability of the mapping. In the unoptimized case, the result of composing the conferences example yields implications with about 200 predicates each whereas the optimized composition algorithm produces a mapping with one single implication containing 15 predicates. The result was the same mapping that would have been handcrafted for the two schemas. Thus, the optimization steps performed in the loop of the algorithm significantly reduce the number of predicates in the composition result. Furthermore,

Example	I_{12}	$P_{l,12}$	$P_{r,12}$	I_{23}	$P_{l,23}$	I_{13}	$P_{l,13}$	time(ms)
1	1	12	6	1	6	1	12	172
2	2	4	6	1	3	2	4	156
3	2	3	3	1	6	1	3	31
4	2	4	6	1	6	1	4	63
5	2	4	6	1	9	1	8	125
6	2	8	18	2	16	2	8	781
7	3	4	6	1	9	3	8	375
8	3	7.33	28.33	1	39	1	15	1281
9	4	4	6	1	9	3	8	250
10	5	4	6	1	18	2	16	625
11	6	4	6	1	21	2	18	735
12	7	4	6	1	24	4	24	1828
13	8	4	6	1	30	8	33	7563
14	9	4	6	1	30	24	33.33	25284

Fig. 15. Time performance of the composition algorithm with optimization

a reduced number of predicates in the implications of course also improves the performance of the containment test that is necessary for the optimization.

The evaluation also confirms the exponential time costs of the composition algorithm which is inline with the results of [10] where it was proven that the computation time of composition is exponential in the size of the input mappings.

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 composed mapping 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.

7 Answering Queries using Generic Schema Mappings

Data integration requires the definition of mappings between source schemas and the global integrated schema. These mappings can be defined in various ways [7]. The basic form of a mapping is $q_{\mathbf{S}} \theta q_{\mathbf{G}}$ in which $q_{\mathbf{S}}$ is a query over a source schema $\mathbf{S}$, $q_{\mathbf{G}}$ is a query over the global schema $\mathbf{G}$, and θ is a comparison operator for sets, such as $\subseteq$, $=$, or $\supseteq$. The usual semantics of such a mapping is that the query $q_{\mathbf{S}}$ evaluated over a valid instance of $\mathbf{S}$ delivers an equivalent, sub-, or super-set of the answers of $q_{\mathbf{G}}$ evaluated over the corresponding instance of $\mathbf{G}$.

Such a mapping in this general form, with queries on both sides of the mapping, is also called a "GLAV mapping" (global-local-as view) [7], as it is a combination of the more frequently used GAV (global-as-view) and LAV (local-as-view) mappings. In GAV mappings, each element of the global schema is described by a query over the source schema (i.e. the query $q_{\mathbf{G}}$ is a single predicate) whereas in LAV mappings, each element of the local schemas is described by a query over the global schema (i.e. the query $q_{\mathbf{S}}$ is a single predicate).

For data integration, the problem of query answering is important. Query answering is the process of rewriting a query defined over the global schema into queries defined over the source schemas. Unless constraints for the global schema are given [20], the rewriting in the case of GAV mappings can be done by unfolding the query definition with mappings. In the case of LAV mappings, more complex algorithms for answering queries using views have to be used [21]. As the mappings presented in this paper are strictly source-to-target, we can use the composition algorithm to rewrite a query over the global, integrated schema into queries over the source

schemas. Recall, that a mapping expression is called "source-to-target if its left hand side contains only elements from the source model and its right hand side references only elements from the target model. Please note that the definition requires the constants identifying model elements to be from $\mathbf{S}$ on the lhs only, and to be from $\mathbf{T}$ only on the rhs. Thus, we only consider source-to-target dependencies in this work.

The details of this rewriting procedure will be explained in the following section.

7.1 Query Answering with Source-To-Target Mappings

We are considering a virtual data integration scenario, in which a query posed to the integrated schema has to be rewritten into queries against the sources in order to compute the query result from the sources. We first define a *query* to a *GeRoMe* model in our data integration scenario.

Definition 5 (*GeRoMe* query) *A query q to a* GeRoMe *model* $\mathbf{M}$ *is defined by an expression of the form* $q(\mathbf{x}) \leftarrow p_1(\mathbf{y_1}) \wedge \ldots \wedge p_n(\mathbf{y_n})$. *The predicates* $p_1, \ldots, p_n$ *can be either equalities or are those atomic predicates defined in definition 2. The lists of arguments of these predicates are similar to definition 3: all predicates must have abstract variables as their first argument;* $inst$, $attr$ *and* $part$ *predicates have constants referring to model elements of* $\mathbf{M}$ *as second argument;* $part$ *predicates have also abstract variables as third argument;* $value$ *predicates have concrete variables or constants as second argument.*

The set $\mathbf{x}$ *is a set of concrete variables, in which each variable occurs at least once as the second argument of a* $value$ *predicate in* $p_1, \ldots, p_n$.

Thus, a *GeRoMe* query is basically a conjunctive query, similar to the left-hand side of a *GeRoMe* mapping. As queries should return only concrete values and not abstract identifiers, the variables in $\mathbf{x}$ must refer to concrete values. Furthermore, in order to have a safe expression, each of the variables in the query head must appear at least once in the query body; as these variables refer to values, they may only appear as second argument of a $value$ predicate.

As an example, we consider as integrated schema the XML schema in fig. 7 with students, universities, and courses. A query for students with their ID and name, and the names of courses which they take, can be expressed in the following way: [3]

$$\begin{aligned} q(i,s,c) \leftarrow\ & inst(x, \texttt{Schema}) \wedge inst(se, \texttt{Student}) \wedge part(se, \texttt{parent}_S, x) \wedge \\ & part(se, \texttt{child}_S, st) \wedge av(st, \texttt{sname}, s) \wedge av(st, \texttt{id}, i) \wedge \\ & inst(ue, \texttt{University}) \wedge part(ue, \texttt{parent}_U, st) \wedge part(ue, \texttt{child}_U, ut) \wedge \\ & inst(ce, \texttt{Course}) \wedge part(ce, \texttt{parent}_C, ut) \wedge part(ce, \texttt{child}_C, ct) \wedge \end{aligned}$$

[3] For reasons of simplicity, we omit in this query redundant $inst$ predicates as in the previous mapping examples.

```
<course cid="1" cname="Databases" uni="RWTH">
  <enrolled sid="123456"/>
  <enrolled sid="234567"/>
  ...
</course>
```

Fig. 16. Structure of the XML document with courses and enrolled students

$$
\begin{aligned}
&\exists k, k', g, g', h, h', d \quad \forall o_1, o_2, o_3, o_4, u, cn, i, c \\
&inst(o_1, \texttt{Course}) \wedge part(o_1, \texttt{parent}_C, o_0) \wedge part(o_1, \texttt{child}_C, o_2) \wedge \\
&inst(o_3, \texttt{enrolled}) \wedge part(o_3, \texttt{parent}_E, o_2) \wedge part(o_3, \texttt{child}_E, o_4) \wedge \\
&av(o_2, \texttt{cid}, c) \wedge av(o_2, \texttt{cname}, cn) \wedge av(o_2, \texttt{uni}, u) \wedge av(o_4, \texttt{sid}, i) \rightarrow \\
&\quad inst(k(i), \texttt{Student}) \wedge part(k(i), \texttt{parent}_S, d()) \wedge part(k(i), \texttt{child}_S, k'(i)) \wedge \\
&\quad inst(g(i,u), \texttt{University}) \wedge part(g(i,u), \texttt{parent}_U, k'(i)) \wedge \\
&\quad part(g(i,u), \texttt{child}_U, g'(i,u)) \wedge inst(h(i,u,c), \texttt{Course}) \wedge \\
&\quad part(h(i,u,c), \texttt{parent}_C, g'(i,u)) \wedge part(h(i,u,c), \texttt{child}_C, h'(i,u,c)) \wedge \\
&\quad av(k'(i), \texttt{ID}, i) \wedge av(g'(i,u), \texttt{uname}, u) \wedge av(h'(i,u,c), \texttt{cid}, c) \wedge \\
&\quad av(h'(i,u,c), \texttt{cname}, cn)
\end{aligned}
$$

Fig. 17. Mapping between courses in XML as in fig. 16 and the integrated schema

$$av(ct, \texttt{cname}, c)$$

The query navigates through the structure of the XML document by starting at the document root x (which is an instance of the `Schema` element) and following the associations from student over university to course. As we are only interested in the id and name of the student, and the name of the course, we just retrieve these attribute values using av predicates; as we are not interested in the name of the universities, we do not have an av predicate for the `uname` attribute of the `UniType`.

It is obvious that such a query language is not intended to be used by end users. For our envisioned integration system, we plan to use a modified variant of SQL with SELECT-FROM-WHERE clauses. On the one hand, this language will be a subset of the original SQL language as we do not plan to support extended features such as aggregation or grouping. On the other hand, the language will extend SQL with path expressions to be able to navigate through nested structures or associations.

To show the basic idea of our rewriting algorithm, we use as an example two mappings which have the integrated schema of fig. 7 as a target:

- $\mathcal{M}_1$ is the mapping shown in fig. 8 from the updated relational schema to the XML schema.
- $\mathcal{M}_2$ is a new mapping from an XML document with courses and students enrolled in these courses (shown in fig. 16) to the integrated XML schema. The mapping is shown in fig. 17.

Suppose now, we have to rewrite a *GeRoMe* query. As the body has the same structure as the left-hand side of a *GeRoMe* mapping, we can use the query as input mapping $\mathcal{M}_{23}$ of the composition algorithm. The conjunction of the mappings

between all sources and the integrated schema is the input mapping $\mathcal{M}_{12}$.

When building the conjunction of mappings, we have to take care of the function symbols used in the mappings. Symbols for concrete functions are associated with a specific implementation of a function; this does not change if we combine several mappings in a conjunction. Abstract functions however are just interpreted as Skolem functions (i.e. as syntactic terms). If different function symbols are used for abstract functions that represent the abstract identifiers of the same model element in two different mappings, then the instances of this model element cannot be matched across these two mappings. To illustrate this problem, we used the function symbol k in the mapping of fig. 17 to identify instances of `Student`, instead of the symbol f that was used in the other mapping in fig. 8. This change is trivial, but it should show that different function symbols could be used in different mappings to identify the instances of the same model element.

Therefore, after constructing the conjunction of the mappings between sources and global schema, we have to check for the mappings that for each model element of the global schema, the same function symbols are used. We assume that, if different function symbols are used for the same model element, then they still have the same set of input arguments. This is not a strong limitation, as also in practice we must have a unique way to identify identical instances in order to be able to merge the information from different sources.

Formally, we have to check the mapping for the occurence of two predicates of the form $inst(f(\mathbf{x}), m)$ and $inst(g(\mathbf{y}), m)$ with $f \neq g$. As said before, we assume that the functions still have the same set of input arguments. As function symbols are existentially quantified, we can replace in the whole mapping all occurences of g with f, without changing the logical semantics of the mapping.

If we have done this transformation, we can apply the composition algorithm to the conjoined mappings and the query. As a result, we will get a set of implications which might refer on the lhs to multiple different source schemas. In the example, each condition of an implication in the resulting mapping will refer to both sources, as information about the student name (`sname`) can be only retrieved from the source in mapping $\mathcal{M}_1$, and information about courses is only given in the source of $\mathcal{M}_2$. In order to be able to generate an executable mapping, we have to partition the predicates in the condition according to the source schemas. If two predicates referring to two different sources share the same variable x, we replace the variable in all predicates of one source with a new variable x' and introduce an additional equality predicate $x = x'$. As equality predicates do not refer to any source schema, they will be handled separatedly.

If this partitioning of the predicates is done, we can pass a set of predicates referring to one source to the query generator component of the data translation tool, execute the generated query and keep the variable assignment until the queries for all sources

have been executed. In a final step, we can then evaluate the equality predicates based on the variable assignment, which basically corresponds to a join operation between two sources. The result of this step is the result of the query.

7.2 Optimization of the Rewritten Query

As we are using the composition algorithm presented in section 4, the same optimization steps will be applied to the rewritten query to get more efficient source queries. In addition, further optimizations can be applied to the query which were not considered for the composition. To explain, we use again our running example: the query asks for the id and name of students together with their course names. The sources provide additional information such as university name and course id, which is not directly relevant for the query result. As predicates about these attributes are contained in the mappings between sources and global schema, these predicates will be also contained in the rewritten query. However, these predicates can be removed from the query if the cardinality constraints of the schema guarantee that such an attribute exists for each instance of the corresponding type.

If we generate an executable SQL query from a *GeRoMe* query, the removal of an av predicate in the query just corresponds to the removal of a term in the SELECT clause of the query, which will probably not have a big impact on query performance. However, for XML, we can omit one path expression to retrieve the attribute value, which might have a bigger impact on query performance.

8 Discussion and Related Work

In this section, we review related work and compare it with our approach for generic schema mappings. We will first present approaches for mapping representation, then discuss the mapping composition in section 8.2, and address the execution of mappings in section 8.3. Section 8.4 presents recent approaches for query answering.

8.1 Mappings

Many different mapping representations are used in various areas of data management, such as data translation, query rewriting and schema integration. Each mapping representation has certain advantages and disadvantages for specific application areas. The simplest form of mappings are binary correspondences (also called morphisms [16]), which are usually the result of schema matching [2].

Morphisms only state a similarity between schema elements but do not provide any detailed semantics of the relationship. More formal mappings are required for tasks such as schema integration and data translation. Schema integration requires *intensional* mappings, which formalize the relationship of model elements with respect to their intensional semantics [4; 22]. In contrast to extensional mappings, intensional mappings are not directly related to a particular instance of the schema.

The topic of this article are *extensional* mappings which can be used for data translation or data exchange. Extensional mappings are defined as local-as-view (LAV), global-as-view (GAV), source-to-target tuple generating dependencies (s-t tgds) [7; 8], second order tuple generating dependencies (SO tgds) [10], or similar formalisms. Source-to-target tuple-generating-dependencies (s-t tgds) or GLAV assertions are used to specify mappings between relational schemas. Therefore, 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 cannot.

Path-conjunctive constraints [23] are an extension of s-t tdgs for dealing with nested schemas. However, they may suffer from several problems [11]. 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 [11], which are used in *Clio* [24], 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. The language of SO tgds is a strict superset of the language of nested mappings [11]. 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 SO tgds, our generic mappings are closed under composition. Furthermore, like nested mappings they are also able to handle nested data and specify arbitrary grouping conditions for elements.

Like path-conjunctive constraints, a *GeRoMe* mapping cannot be nested into another *GeRoMe* mapping. Thus, 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 [11] to allow nested mapping definitions.

Another form of mappings based on a Datalog-like representation is used by Atzeni et al. [3]. 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. In our approach, we are able to translate the generic mapping representation into a specific query language

of a metamodel and thereby avoid this overhead.

8.2 Mapping Composition

Mapping composition has been studied for various mapping formalisms [9; 10; 18]. A semantics for mapping composition has been proposed firstly in [9]. The authors defined the semantics of the Compose operator relative to a class $\mathcal{Q}$ of queries over the model $\mathbf{S}_3$ where the "equivalence" means that, for every query q in $\mathcal{Q}$, the certain answers for q wrt. $\mathcal{M}_{13}$ are the same as the certain answers for q wrt. $\mathcal{M}_{12}$ and $\mathcal{M}_{23}$. The main drawback of this definition is that the semantics of the composition is relative to the class of queries $\mathcal{Q}$.

Mapping composition using s-t tgds was explored in [10]. It was proven that the language of s-t tgds is not closed under composition. To ameliorate the problem, the authors 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 $\mathcal{M}$ and an instance $\mathcal{I}$ over the source schema, it takes polynomial time to calculate the solution $\mathcal{J}$ which is an instance over the target schema and which satisfies $\mathcal{M}$. Thus, SO tgds are a good formalization of mappings and therefore, we have chosen them as the basis for our mapping representation. We adapted the composition algorithm of [10] to our generic mapping representation and thereby allow composition of mappings between heterogeneous modeling languages. Furthermore, we implemented the composition algorithm and made the observation, that the exponential complexity of the algorithm and the result size is not applicable in a pratical solution. Therefore, we added several optimization steps to the composition algorithm which reduces on the one hand the runtime of the algorithm and, on the other hand, the size of the result. Therefore, the results of our composition algorithm can be executed efficiently.

Another approach for mapping composition uses relational algebra expressions as mappings [18]. 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 [10].

8.3 Executable mappings

Recall that an executable mapping is a mapping that has formal semantics and can be used for translating data or queries between different schemas of overlapping data. Executable mappings are necessary in many metadata intensive applications, such as database wrapper generation, message translation and data transformation [8].

While many model management systems were used to generate mappings that drive the above applications, few of them were implemented using executable mappings. The reason is probably due to the lack of rigorous semantics of model management operators if the mappings are executable mappings.

Because executable mappings usually drive the transformation of instances of models, Melnik et al. [8] specified a semantics of each operator by relating the instances of the operator's input and output models. The authors 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 [16] language to define path morphisms. 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 [8] 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. Although they further developed a script rewriting procedure to ameliorate this problem, it has not been completely solved.

In this article we modified the language of SO tgds such that we are able to represent generic mappings between schemas in heterogeneous modeling languages, as opposed to mappings between relational schemas only. Despite these changes, we have shown that our mapping language remains executable while at the same time being closed under composition and allowing for restructuring of data.

8.4 Query Answering

Many approaches dealt with the problem of query rewriting for data integration or data exchange scenarios [21; 7; 25; 26]. Since our mappings are strictly source-to-target mappings, we can use mapping composition for query rewriting.

Our work on query answering and query rewriting with our mapping representation is still work in progress. One limitation in our approach is that we assume that the sources are sound and complete, and therefore do not consider the constraints in the target model. Constraints on the global schema and the assumption of incomplete sources complicate the process of query answering also for the case of source-to-target mappings [20]. If the constraints include foreign keys which have cyclic dependencies, query answering becomes more difficult. In order to get the certain answers [27] to a query, the query has to be rewritten to take into account dependencies which are defined by the foreign key relationships. Similar work, but for a data exchange setting, is done in [26].

Our goal is to use our mappings in a peer-to-peer data integration scenario [25].

However, in such a system, a peer that serves as target in one mapping serves as source in another mapping. In this case, query unfolding cannot be used for rewriting the query. Therefore, we have to consider additional query rewriting techniques such as query answering using views [21]. The results have to be combined with queries rewritten by mapping composition. The expressiveness of our generic mapping language and the generic metamodel might complicate the adaptation of existing algorithms, but it provides on the other hand a good basis for a useful and efficient peer-to-peer system.

9 Conclusion

In this paper we presented a rich language for schema mappings between models given in our Generic Role-based Metamodel *GeRoMe* [12]. Our mapping language is closed under composition as it is based on second order tuple-generating dependencies [10]. 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, object-oriented models, or OWL ontologies and are not restricted to flat schemas 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 [11] which are not as expressive as SO tgds. However, such grouping functionality is not supported by conventional SO tgds.

We implemented an adapted version of the composition algorithm for second order tuple-generating dependencies [10]. Furthermore, we showed that the mapping language is still executable by developing a tool that exports our mappings to queries and updates in the required data manipulation language and then uses them for data translation. Exemplarily, we introduced algorithms that translate the source side of a generic mapping to a query in XQuery as well as algorithms for translating the target side of a generic mapping into SQL updates and for executing these updates. The algorithm for generating SQL updates harnesses the opportunity to specify mappings in which multiple implications contribute to the same generated tuples. The components for mapping export and execution can be arbitrarily replaced by implementations for the required metamodels. The evaluation showed that both, mapping composition and mapping execution, yield the desired results with a reasonable time performance. Furthermore, we discussed how generic schema mappings can be used for query rewriting and pointed out some problems that may appear in this context.

We are currently developing 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* [13]. We will also investigate the relationship between our extensional

mappings and intensional mappings that are used for schema integration [4]. Furthermore, we are investigating techniques to generate executable generic schema mappings from simple morphisms. As a starting point we adapt the techniques employed by Clio [28] to our generic metamodel and mapping language.

Acknowledgements:
This work is supported by the DFG Research Cluster on Ultra High-Speed Mobile Information and Communication UMIC at RWTH Aachen University, Germany (`http://www.umic.rwth-aachen.de`).

References

[1] P. A. Bernstein, A. Y. Halevy, R. Pottinger, A vision for management of complex models, SIGMOD Record 29 (4) (2000) 55–63.

[2] E. Rahm, P. A. Bernstein, A survey of approaches to automatic schema matching, VLDB Journal 10 (4) (2001) 334–350.

[3] P. Atzeni, P. Cappellari, P. A. Bernstein, Model-independent schema and data translation, in: Y. E. Ioannidis, M. H. Scholl, J. W. Schmidt, F. Matthes, M. Hatzopoulos, K. Böhm, A. Kemper, T. Grust, C. Böhm (Eds.), Proc. 10th International Conference on Extending Database Technology (EDBT), Vol. 3896 of Lecture Notes in Computer Science, Springer, Munich, Germany, 2006, pp. 368–385.

[4] C. Quix, D. Kensche, X. Li, Generic schema merging, in: J. Krogstie, A. Opdahl, G. Sindre (Eds.), Proc. 19th Intl. Conf. on Advanced Information Systems Engineering (CAiSE'07), LNCS, Springer-Verlag, 2007, pp. 127–141.

[5] T. Catarci, M. Lenzerini, Representing and using interschema knowledge in cooperative information systems, Intl. Journal of Cooperative Information Systems 2 (4) (1993) 375–398.

[6] P. A. Bernstein, S. Melnik, Model management 2.0: Manipulating richer mappings, in: L. Zhou, T. W. Ling, B. C. Ooi (Eds.), Proc. ACM SIGMOD Intl. Conf. on Management of Data, ACM Press, Beijing, China, 2007, pp. 1–12.

[7] M. Lenzerini, Data integration: A theoretical perspective, in: L. Popa (Ed.), Proc. 21st ACM Symposium on Principles of Database Systems (PODS), ACM Press, Madison, Wisconsin, 2002, pp. 233–246.

[8] S. Melnik, P. A. Bernstein, A. Y. Halevy, E. Rahm, Supporting executable mappings in model management., in: F. Özcan (Ed.), Proceedings of the ACM SIGMOD International Conference on Management of Data, ACM, Baltimore, Maryland, USA, 2005, pp. 167–178.

[9] J. Madhavan, A. Y. Halevy, Composing mappings among data sources, in: J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, A. Heuer (Eds.), Proc. of 29th Intl. Conference on Very Large Data Bases (VLDB), Morgan Kaufmann, Berlin, Germany, 2003, pp. 572–583.

[10] R. Fagin, P. G. Kolaitis, L. Popa, W. C. Tan, Composing schema mappings:

Second-order dependencies to the rescue., ACM Trans. Database Syst. 30 (4) (2005) 994–1055.

[11] A. Fuxman, M. A. Hernández, C. T. H. Ho, R. J. Miller, P. Papotti, L. Popa, Nested mappings: Schema mapping reloaded., in: U. Dayal, K.-Y. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, Y.-K. Kim (Eds.), Proc. 32nd Intl. Conference on Very Large Data Bases (VLDB), ACM Press, 2006, pp. 67–78.

[12] D. Kensche, C. Quix, M. A. Chatti, M. Jarke, *GeRoMe*: A generic role based metamodel for model management, Journal on Data Semantics VIII (2007) 82–117.

[13] D. Kensche, C. Quix, X. Li, Y. Li, *GeRoMeSuite*: A system for holistic generic model management, in: C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C.-C. Kanne, W. Klas, E. J. Neuhold (Eds.), Proceedings 33rd Intl. Conf. on Very Large Data Bases (VLDB), Vienna, Austria, 2007, pp. 1322–1325.

[14] D. Kensche, C. Quix, Y. Li, M. Jarke, Generic schema mappings, in: Proc. 26th Intl. Conf. on Conceptual Modeling (ER'07), 2007, pp. 132–148.

[15] D. Kensche, C. Quix, Transformation of models in(to) a generic metamodel, in: Proc. BTW Workshop on Model and Metadata Management, 2007, pp. 4–15.

[16] S. Melnik, E. Rahm, P. A. Bernstein, Rondo: A programming platform for generic model management, in: Proc. ACM SIGMOD Intl. Conference on Management of Data, ACM, San Diego, CA, 2003, pp. 193–204.

[17] G. G. Robertson, M. P. Czerwinski, J. E. Churchill, Visualization of mappings between schemas, in: CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems, ACM Press, New York, NY, USA, 2005, pp. 431–439.

[18] P. A. Bernstein, T. J. Green, S. Melnik, A. Nash, Implementing mapping composition, VLDB Journal 17 (2) (2008) 333–353.

[19] J. D. Ullman, Information integration using logical views, in: F. N. Afrati, P. G. Kolaitis (Eds.), Proceedings of the 6th International Conference on Database Theory (ICDT), Vol. 1186 of Lecture Notes in Computer Science (LNCS), Springer, Delphi, Greece, 1997, pp. 19–40.

[20] A. Calì, D. Calvanese, G. D. Giacomo, M. Lenzerini, Data integration under integrity constraints, Information Systems 29 (2) (2004) 147–163.

[21] A. Y. Halevy, Answering queries using views: A survey, VLDB Journal 10 (4) (2001) 270–294.

[22] S. Spaccapietra, C. Parent, Y. Dupont, Model independent assertions for integration of heterogeneous schemas, VLDB Journal 1 (1) (1992) 81–126.

[23] L. Popa, V. Tannen, An equational chase for path-conjunctive queries, constraints, and views, in: ICDT '99, Springer-Verlag, London, UK, 1999, pp. 39–57.

[24] M. A. Hernández, R. J. Miller, L. M. Haas, Clio: A semi-automatic tool for schema mapping, in: Proc. ACM SIGMOD Intl. Conference on the Management of Data, ACM Press, Santa Barbara, CA, 2001, p. 607.

[25] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, I. Tatarinov, The

piazza peer data management system, IEEE Transactions on Knowledge and Data Engineering 16 (7) (2004) 787–798.

[26] R. Fagin, P. Kolaitis, R. J. Miller, L. Popa, Data exchange: Semantics and query answering, Theoretical Computer Science 336 (2005) 89–124.

[27] S. Abiteboul, O. M. Duschka, Complexity of answering queries using materialized views, in: Proceedings of the Seventeenth ACM Symposium on Principles of Database Systems (PODS), ACM Press, Seattle, Washington, 1998, pp. 254–263.

[28] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernandez, R. Fagin, Translating web data, in: Proc. 28th Intl. Conference on Very Large Data Bases (VLDB), Morgan Kaufmann, Hong Kong, China, 2002, pp. 598–609.