Semantic Adaptation of Schema Mappings when Schemas Evolve

Cong Yu[*]
Dept. of EECS, Univ. of Michigan
congy@umich.edu

Lucian Popa
IBM Almaden Research Center
lucian@almaden.ibm.com

Abstract

Schemas evolve over time to accommodate the changes in the information they represent. Such evolution causes invalidation of various artifacts depending on the schemas, such as *schema mappings*. In a heterogenous environment, where cooperation among data sources depends essentially upon them, schema mappings must be adapted to reflect schema evolution. In this study, we explore the *mapping composition* approach for addressing this *mapping adaptation* problem. We study the semantics of mapping composition in the context of mapping adaptation and compare our approach with the incremental approach of Velegrakis et al [21]. We show that our method is superior in terms of capturing the semantics of both the original mappings and the evolution. We design and implement a mapping adaptation system based on mapping composition as well as additional *mapping pruning* techniques that significantly speed up the adaptation. We conduct comprehensive experimental analysis and show that the composition approach is practical in various evolution scenarios. The mapping language that we consider is a nested relational extension of the second-order dependencies of Fagin et al [7]. Our work can also be seen as an implementation of the mapping composition operator of the model management framework.

1 Introduction

Independent data sources are often heterogeneous even when they cover the same information domain, as reflected in the adoption of different schemas for describing the data. Despite the difficulties of dealing with such heterogeneous data, cooperation among data sources, with the goal of providing a comprehensive and cohesive view of the information, has become more and more important, for many applications. Schemas and schema mappings are two fundamental metadata ingredients that are at the core of such cooperation. Schemas describe the structure and, to some extent, the semantics of data at the data sources, while schema mappings describe relationships between data sources. Schema mappings can be used to transform data between two different schemas (*data exchange or translation* [6, 20]), or to translate queries over one schema to queries over a different schema [13, 22]. In particular, they can be used in answering queries over a mediated schema that unifies multiple source schemas (*data integration* [8, 10]).

Schema mappings are often specified using high-level declarative formalisms that describe the correspondences between different schemas at a logical level. Such *logical* mappings can be viewed as abstractions for the more complex *physical* specifications (e.g., XQuery or XSLT scripts) that operate at the data transformation runtime. Such abstractions are easier to understand and to reason about, and still capture most of the information needed to generate the physical artifacts. This philosophy of abstract representation, which we follow here, is adopted by many recent data exchange and integration systems. In particular, GAV (global-as-view), LAV (local-as-view), and, more generally, GLAV (global-and-local-as-view) assertions have been used in data integration systems for query answering and rewriting (see [8, 10] for two surveys). Similarly, source-to-target tuple-generating dependencies (s-t tgds) have been used for specifying data exchange between relational schemas [6]; moreover, nested (XML-style) s-t tgds have been used in the *Clio* data exchange system [18] as the underlying representation for transformations between XML schemas. Schema mappings can often be derived semi-automatically [16, 18] based on the outcome of schema matching algorithms [19].

Schemas of data sources invariably evolve over time due to various reasons. For example, the incorporation of new data not captured by an existing schema will require the introduction of new schema structures. Once schemas change, the mappings between these schemas, together with the physical artifacts that were generated based on them, may become invalid. A typical solution is to regenerate the mappings and then regenerate the depending artifacts. However, even with the help of mapping generation tools, this process can be costly in terms of human effort and expertise, especially for complex schemas. Moreover, there is no guarantee that the regenerated mappings preserve the semantics of the original mappings. A better solution is to design algorithms that *reuse* the original mappings and (semi-)automatically *adapt* them into a set of mappings that are valid with respect to the new schemas and, moreover, reflect the semantics of the original mappings and the schema evolution. This process is called *mapping adaptation* under schema evolution.

A comprehensive method for *incremental* mapping adaptation was established in [21]. Its main idea is to incrementally change the mappings each time a primitive change occurs in the source or target schema. The method, however, has a few drawbacks. First, when drastic schema evolution occurs (i.e., significant restructuring in one of the original schemas) and the new schema version is directly given, it is unclear how feasible it is to extract the list of primitive changes that can describe the evolution. Such scenarios often occur in practice, especially in scientific fields (e.g., HL7 Clinical Document Architecture, Swissprot, MAGE-ML, and their many schema versions, used in health-care and

[*] Supported in part by NSF under grant IIS-0219513, by NIH under grant LM08106-01; work partially done while at IBM Almaden Research Center.

**Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005**

bioinformatics[1]). We also point out later that the set of primitive changes in [21] is not expressive enough to capture complex evolution. Second, even when such a list of changes can be obtained, applying the incremental algorithm for each change in this potentially very long list will be highly inefficient. Finally, there is no guarantee that after repeatedly applying the algorithm, the semantics of the resulting mappings will be the desired one.

A more general approach, which has been suggested in [2, 3, 21] is to describe the schema evolution itself as mappings and to employ mapping composition to derive the adapted mappings. While attractive in principle, there is so far no concrete study that demonstrates the practicality of such approach, due to the many challenges involved, including: how to represent schema evolution as mappings; what is the right mapping language; what is the semantics of such mapping composition; and how mappings can be efficiently composed. In this paper, we address these challenges and offer a systematic study and comprehensive evaluation of how mapping composition can be applied to solve the mapping adaptation problem.

Main contributions and paper outline I. We study the mapping-based representation of schema evolution and show that it is more flexible and expressive than the change-based representation (Section 2); II. We explore the semantics of mapping composition and its application to mapping adaptation (Sections 2 and 3); III. We show that our prior mapping-based query rewriting algorithm [22] can be applied to implement the hierarchical mapping composition that we need for mapping adaptation (Section 4); IV. We design *mapping pruning* algorithms to improve the performance of mapping adaptation (Section 5), and provide a comprehensive evaluation of the resulting system, showing its scalability and practicality (Section 6). Our work is part of a recent effort for building a meta-data management system, called *Criollo*, jointly started between IBM Almaden Research and IBM Software Group.

Related work The ubiquity of schemas and schema mappings has recently motivated the developing of frameworks for managing such metadata objects. Bernstein et al have introduced such a framework [2, 3], called *model management*. Our work can be seen as part of this model management framework. Recently, two studies show the limitations of existing mapping languages (GLAV and s-t tgds) for mapping composition, and provide composition algorithms within a limited context [12] or with a language extension [7]. Our work uses the results in [7] and applies them to the hierarchical model and the mapping adaptation context. A more detailed view on how our work relates to existing work on mapping composition and query rewriting is given in Section 2.2. Besides [21], which we discussed earlier, [1, 11] study schema evolution in the object-oriented databases context. However, the focus there is to recognize schema changes and produce transformations to keep the data up to date. Another closely related work is the EVE [9] system, which is the first to define and address the problem of adapting view definitions when the schema of base relations change. However, the supported changes are limited and evolution can only appear at the source side.

2 Motivation and Overview

In this section we motivate and give an overview of our composition-based approach to mapping adaptation. We start with an overly simplified relational example that is enough to show the motivation for mapping composition and its advantages over the related approaches (Sections 2.1 and 2.2). Motivated by the need to support XML data, we then present the nested relational model, nested schemas and mapping language, upon which our solutions are based (Section 2.3).

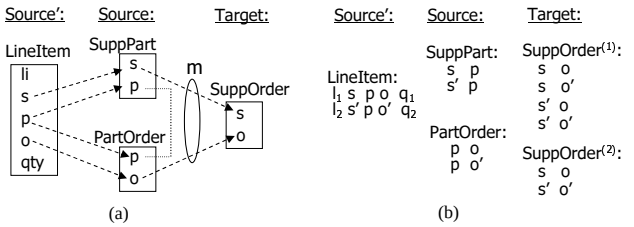


Figure 1: (a) Mapping scenario and evolution. (b) Data example.

2.1 The Mapping Adaptation Problem

Consider the following example, based on the TPC-H schema, where an e-commerce application stores information about suppliers and the parts they supply in a `SuppPart` relation and information about orders and the parts they order in a `PartOrder` relation. These two relations are part of the schema `Source` in Figure 1(a). The following constraint from schema `Source` to schema `Target` (also shown in Figure 1) expresses the fact that orders, together with all their potential suppliers (i.e., those that can supply the relevant parts), must be exported into the target relation `SuppOrder`:

$$\text{(m)}\quad \texttt{SuppPart}(s,p) \ \wedge\ \texttt{PartOrder}(p,o) \rightarrow \texttt{SuppOrder}(s,o)$$

In general, we allow the left-hand side of such constraint to have a conjunction of relational atoms over the source, and the right-hand side to have a conjunction of relational atoms over the target schema. All variables on the left are universally quantified. Although not in this example, there may be variables in the right-hand side that do not appear on the left. Such variables are existentially quantified and are typically used to handle those parts of the target schema that do not have a correspondence from the source schema. This class of assertions are known as sound (open-world) GLAV mappings [10] or s-t tgds [6]. A schema mapping is a set of such assertions.

Schema evolution Assume now that the raw data arrive from an external source in the form of tuples (li, s, p, o, qty), relating an order o and a part p with an available supplier s. Rather than splitting and inserting the data into the two relations, `SuppPart` and `PartOrder`, a decision is made by the application to store the incoming tuples, as they are, in a `LineItem` relation, which becomes the new schema `Source'`. This reorganization can save the cost of extra data processing, and the cost of maintaining the two original relations. However, the mapping m that depends on schema `Source` must now be changed (migrated) to use schema `Source'`.

As mentioned, one solution is to regenerate a new mapping based on `Source'` and `Target`, by re-establishing correspondences between the schemas (schema matching), and regenerating the mapping that embodies the actual semantics of the transformation (mapping generation). We refer to this method as the *blank-sheet approach*. Besides being costly for large schemas, this approach suffers in that the original mapping is not considered during the regeneration. Hence, the semantics embedded in the original mapping may be lost. A better approach is to *reuse* the original mapping and *adapt* it in a way that (1) preserves the

[1]http://www.hl7.org/; http://www.ebi.ac.uk/swissprot/; http://www.mged.org/Workgroups/MAGE/mage-ml.html

intention of the original mapping, and (2) takes into account all the schema changes, so that the adapted mapping is valid with respect to the new schema.

Incremental approach and change-based representation (CBR) One such method was established in [21]: the main idea is that schemas often evolve in small, primitive, steps[2]; after each such step, the schema mapping can be incrementally adapted by applying local modifications. We apply their incremental algorithm to our example by evolving Source into Source′ using one possible sequence of primitive changes. We refer to such a representation of schema evolution (i.e., a list of primitive changes) as *change-based representation (CBR)*. Based on the changes, we explain how mapping m is adapted according to [21] (without giving the full details of that algorithm, which also operates on a different syntax).

- Move SuppPart/s to become PartOrder/s. The mapping m has a reference to SuppPart/s in the target side. This will be changed to a reference to PartOrder/s. Also, the second source atom must be changed to reflect this new attribute in PartOrder. The resulting mapping is:
  (m) $\texttt{SuppPart}(p) \wedge \texttt{PartOrder}(s, p, o) \rightarrow \texttt{SuppOrder}(s, o)$
- Delete SuppPart/p and then Delete the relation SuppPart (we merge two steps into one here). The atom that involves SuppPart is dropped.
- Rename PartOrder to LineItem and Add LineItem/li and LineItem/qty (we merge three steps here). The only source atom is changed to refer to LineItem (with two additional variables). We obtain:
  (m^i) $\texttt{LineItem}(li, s, p, o, qty) \rightarrow \texttt{SuppOrder}(s, o)$

Deficiencies of the incremental approach The incremental approach, designed to handle a few primitive changes, is intuitive and efficient for such cases: the mapping is minimally changed so that it becomes syntactically valid with respect to the new schema and still reflects the schema correspondences (i.e., s↦s even when s moves from SuppPart to PartOrder). However, once we try to apply this algorithm to non-incremental evolution, we are faced with several issues (some of them already noted in [21]). First, the algorithm must be reapplied after each primitive change. This can become inefficient since we often need a long list of changes to represent non-incremental evolution scenarios. Even for our example, we need 6 primitive changes. Moreover, the list of changes may not be given and may need to be discovered (in the common case when we do not know how one schema evolved into another). If we do such discovery then there may be multiple lists of changes with the same effect of evolving the old schema into the new one. The work in [21] did not study whether their algorithm is confluent (i.e., whether the resulting mapping is independent of which list of changes is given).

Furthermore, somewhat surprisingly, the semantics of the above mapping m^i may not be the expected one. Consider the instances in Figure 1(b). The middle instance, under Source, includes two suppliers (s and s') that both supply a certain part (p) and two orders (o and o') that both ask for the same part (p). The left instance, under Source′, consists of two LineItem tuples that are consistent with Source data. The relation $\texttt{SuppOrder}^{(1)}$, under Target, includes all pairs that the original mapping m requires to exist in SuppOrder, based on Source data: each order is paired with all suppliers since they all supply the relevant part. In contrast, the relation $\texttt{SuppOrder}^{(2)}$ contains the pairs that the incrementally adapted mapping m^i requires to exist in SuppOrder, based on Source′ data. Notably, m^i loses the fact that o should also be related to s', and o' should also be related to s.

Thus, m^i does not quite capture the intention of the original mapping, given the new format of the incoming data. Part of the reason this happens is that the new Source′ data does not necessarily satisfy a join dependency that is explicitly encoded in the original mapping m. There are other examples where the incremental approach falls short in terms of preserving the semantics. Furthermore, the same goes for the blank-sheet approach. Indeed, on the previous example, if we just match the common attributes of Source′ and Target, and regenerate the mapping based on this matching, we would obtain the same mapping m^i as in the incremental approach. A systematic approach, with stronger semantic guarantees, is clearly needed.

2.2 Our Approach: Mapping Composition and Mapping-Based Representation (MBR)

The previous example showed one of the pain points of the incremental approach: the lack of a precise criterion under which the adapted mapping is indeed the "right" result. The adapted mapping was given by the algorithm itself and there was no *a priori* semantic definition to validate the implementation.

In this paper we take the approach where the schema evolution itself is quantified as a mapping rather than a list of changes. This *mapping-based representation (MBR)* of schema evolution is a more general, more declarative and more *precise* representation of how *data* at the new schema relates to *data* at the old schema. Arbitrary (non-incremental) schema evolution can be tackled as long as such evolution is expressed as a mapping. Moreover, this enables a precise and simple definition of mapping adaptation as the *composition* between the mapping describing the evolution and the original mapping.

Back to the example in Figure 1, the obvious mapping from Source′ to Source is the following set of constraints, describing how data in LineItem relates to data in SuppPart and PartOrder, respectively:

(e_1) $\texttt{LineItem}(l, s, p, o, q) \rightarrow \texttt{SuppPart}(s, p)$
(e_2) $\texttt{LineItem}(l, s, p, o, q) \rightarrow \texttt{PartOrder}(p, o)$

In order to obtain the new mapping from Source′ to Target, we compose $\{e_1, e_2\}$ with $\{m\}$ by simply substituting the references to SuppPart and PartOrder in m with corresponding references to LineItem:

(m^a) $\texttt{LineItem}(l_1, s, p, o_1, q_1) \wedge \texttt{LineItem}(l_2, s_2, p, o, q_2) \rightarrow \texttt{SuppOrder}(s, o)$

Mapping m^a has now the same semantics as the original mapping m, in the following precise sense: applying $\{m^a\}$ to a Source′ instance yields the same Target instance as applying first $\{e_1, e_2\}$ to the given Source′ instance, followed by applying $\{m\}$ to the resulting Source instance. This can be easily verified for the data example shown in Figure 1(b).

Obtaining such composition, for schema evolution, is the focus of this paper. Several remarks are in order now to better explain and delineate our current work from existing work on mapping composition and query rewriting:

- Further extensions to the above mapping language are necessary to deal with nested/XML schemas.
- While the above composition looks similar to query unfolding techniques, mapping composition poses its own set of challenges. A main difference is the incompleteness of mappings

[2]Including move/copy/rename/delete elements and add/delete constraints.

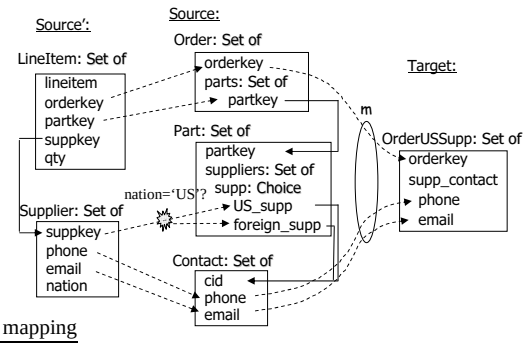
that can manifest via the existentially quantified variables that may occur in the target side. Because of this, GLAV mappings may not compose [12, 7]. On the other hand, an extension to this language that includes second-order functions suffices to provide composability [7]. We will consider such extension in here, too.

- There are two different semantics that can be associated with mappings and, consequently, with composition. The first, which we call *relationship semantics*, is the one under which mappings are constraints required to hold between pairs of instances over the source and the target schema. Under this semantics, multiple pairs of instances (I, J) and (I, J') may satisfy the relationship (i.e., we do not have functionality). The second semantics, which we call *transformation semantics*, is the one under which mappings are viewed as transformations from source to target: given a source instance I, there is a canonical way of computing (by chasing) a unique target instance J. This transformation semantics follows the data exchange line of work in [6, 7, 22].
- We argue that for schema evolution the transformation semantics is more useful and more intuitive. In contrast, the relationship semantics, which is the focus in [7, 15, 17], is often too general and the result of composition is syntactically complex. The transformation semantics, which is our focus, is more challenging to implement due to the need for minimization (as in conjunctive query minimization [4]).
- We show that our algorithm given in [22] to rewrite queries over a target schema based on a source-to-target mapping can be tailored to handle mappings (rather than target queries). We show that we obtain a mapping composition algorithm that is correct for the transformation semantics.
- Mapping composition poses increased scalability challenges when compared to usual query rewriting approaches. This is due to the fact that mappings between schemas must often cover the entire schema, while queries usually access only parts of a schema and typically produce simple output. Fortunately, the mapping composition that is needed for schema evolution can benefit from special optimizations. One of our main contributions is to formalize such optimizations and make the composition approach practical.
- While we concentrate here on mappings that have a direction (source-to-target), the recent work in [17] studies composition of schema mappings that are given by arbitrary embedded dependencies over two schemas. While more general, their framework also poses greater computational challenges: they show several negative results, and give a composition algorithm that terminates under certain conditions. Schema evolution is not specifically addressed in their work.

2.3 Nested Mapping Framework

In this section, we introduce nested schemas and illustrate the mapping language extensions that allow us to handle evolution scenarios involving XML and hierarchical sources. These extensions follow closely the internal schema language and mapping language that we have developed in our previous work on Clio schema mapping generation [18], mapping-based query rewrite [22], and incremental mapping adaptation [21].

Consider the mapping scenario between schemas `Source` and `Target` in Figure 2. The schemas are shown in a nested relational representation (to be defined shortly) that can be used



Original mapping

m: foreach o in `Order`, p in o.parts,
p' in `Part`, s in p'.suppliers, i in case s.supp→US_supp,
c in `Contact`
where p.partkey = p'.partkey and i = c.cid
exists os in `OrderUSSupp`
with os.orderkey = o.orderkey and os.supp_contact.phone = c.phone
and os.supp_contact.email = c.email

Evolution mapping

e_1: foreach l in `LineItem`, s_0 in `Supplier`
where l.suppkey = s_0.suppkey and s_0.nation = 'US'
exists o in `Order`, p in o.parts,
p' in `Part`, s in p'.suppliers, i in case s.supp→US_supp,
c in `Contact`
where p.partkey = p'.partkey and i = c.cid
with o.orderkey = l.orderkey and p.partkey = l.partkey
and i = s_0.suppkey and c.phone = s_0.phone and c.email = s_0.email

e_2: similar to e_1, but with s_0.nation ≠'US' as source filter, and
i in case s.supp→foreign_supp as target choice selection

Figure 2: A more complex mapping scenario and evolution.

as a common data model for both relational schema and XML Schema. The mapping between `Source` and `Target` is a singleton set containing one *logical mapping* (m). In general, a schema mapping M between two schemas S and T will be a set $\{m_1, \ldots, m_k\}$ of logical assertions over S and T that are called logical mappings (or sometimes, mappings).

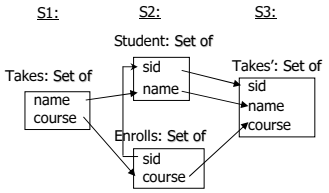
Expanded from the previous example, schema `Source` describes data about orders, parts, and suppliers. Each `Order` contains multiple `Part`s, which can be supplied by multiple `suppliers` that are from either 'US' or 'foreign' countries (denoted by the Choice type of the `supp` element). The `partkey` foreign key associates parts with orders, while the `US_supp` and `foreign_supp` foreign keys associate the `Contact` information with the suppliers.

Schemas and types In general, a *schema* is a set of labels (called roots), each with an associated *type* τ, defined by:

$$\tau ::= \mathsf{Str} \mid \mathsf{Int} \mid \mathsf{SetOf}\ \tau \mid \mathsf{Record}[l_1 : \tau_1, \ldots, l_n : \tau_n] \mid \mathsf{Choice}\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle$$

Types Int and Str are atomic types (we only list two here). SetOf types model the repeatable elements of XML Schema, and Record and Choice types represent the "all" and "choice" *model-groups*, respectively. In Figure 2, only SetOf and Choice types are shown explicitly. We do not consider order: SetOf represents unordered sets, and the "Sequence" model-groups of XML Schema are represented as (unordered) Record types.

Intuitively, a logical mapping implements a group of arrows between the "mapped" schema elements. Such arrows are also called *value correspondences (or VCs)* [16, 18]. The intention of the above logical mapping m is to map all orders and their parts, along with the contact information of the *US* (only) suppliers of those parts, into the target relation `OrderUSSupp`. This is made precise by the formula that is associated with m in Figure 2. We abandon the logic-based notation in favor of



m_1: foreach t in Takes
exists s in Student
with s.sid = F(t.name) and
s.name = t.name

m_2: foreach t in Takes
exists e in Enrolls
with e.sid = F(t.name) and
e.course = e.course

e: foreach s in Student, e in Enrolls where s.sid = e.sid
exists t in Takes′
with t.sid = s.sid and t.name = s.name and t.course = e.course

Figure 3: Schema mappings $\{m_1, m_2\}$ and $\{e\}$, using functions.

a more suitable XQuery-like notation that can easily express navigation through nested records via record projection (e.g., os.supp_contact.phone), navigation through set-type elements by using variables to explicitly iterate over the sets, as well as choice selection by explicit case bindings.

Mapping language An *expression* is defined by the grammar $e ::= S \mid x \mid e.l$, where x is a variable, S is a schema root, l is a label, and $e.l$ is record projection. A *term* is defined as $t ::= e \mid F(t)$, where F is a function symbol. Then a logical mapping has the following form:

$$\begin{array}{lll} m ::= \underline{\text{foreach}} & x_1 \ \underline{\text{in}}\ g_1, \ldots, x_n \ \underline{\text{in}}\ g_n & \underline{\text{where}}\ \ B_1 \\ \underline{\text{exists}} & y_1 \ \underline{\text{in}}\ g'_1, \ldots, y_m \ \underline{\text{in}}\ g'_m & \underline{\text{where}}\ \ B_2 \\ \underline{\text{with}} & e'_1 = t_1 \ \underline{\text{and}} \ldots \underline{\text{and}}\ e'_k = t_k & \end{array}$$

Each x_i in g_i (y_j in g'_j) is called a *generator*. Each g_i (g'_j) is either: (1) an expression e of type SetOf τ, in which case the variable x_i (y_j) binds to individual elements of type τ, or (2) case $e \rightarrow l$, where e is an expression of type Choice$\langle \ldots, l : \tau, \ldots \rangle$, in which case x_i (y_j) binds to the element (if any) of type τ under the choice l of e. The variable (if any) used in g_i (or g'_j) must be defined by a previous generator in the same clause. Any schema root used in the foreach or exists clause must be a source or target schema root, respectively. The two where clauses (B_1 and B_2) are conjunctions of equalities between source *terms* (in the case of B_1) or target *expressions* (in the case of B_2) over x_i or y_j, respectively. We also allow equalities or inequalities with constants (i.e., selections). In the with clause, each equality $e'_i = t_i$ involves a target *expression* e'_i and a source *term* t_i of the same *atomic* type.

In the above example, the schema mapping $\{m\}$ contains no function symbols. As a different example, consider the schema mapping $\{m_1, m_2\}$ shown in Figure 3. There, the function symbol F appears in both m_1 and m_2; the term F(t.name) is used to represent a student id that is associated, via F, with the student name t.name. It is this term that is "assigned" to the target `sid` element in the two logical mappings.

Let $M = \{m_1, \ldots, m_k\}$ be a schema mapping consisting of a set of logical mappings and let $\{F_1, \ldots, F_n\}$ be the set of function symbols that appear in M. Then M defines a constraint between instances of the two schemas that holds whenever one can find an interpretation for the functions $F_1, \ldots, F_n$, so that all of $m_1, \ldots, m_k$ hold, for the given interpretation. Thus, the meaning is that of a second-order constraint of the form $\exists F_1 \ldots \exists F_n [m_1 \wedge \ldots \wedge m_k]$. We note that the functions that we consider are *not* the traditional user-defined functions[3]; our functions are special and have a more basic role of explicitly denoting and linking "unknown" elements (e.g., encoding that the same `sid` value must exist in the two tables `Student` and `Enrolls`, for a source student `name`.) Oftentimes, the necessary function terms are system generated (via Skolemization algorithms and via composition itself).

Without functions, the above mapping language reduces to the language of GLAV mappings (or s-t tgds), in the relational case. This language was shown not expressive enough for composition: the composition of two sequential schema mappings may not expressible in this same language [12, 7]. However, [7] showed a natural extension, called second-order tgds (or SO tgds), which includes functions and is compositional. Our mapping language is a nested-relational *extension* of SO tgds and, in fact, the two languages coincide in the relational case.

Figure 2 also illustrates a source evolution scenario: the schema `Source` is evolved into `Source'`. In particular, the relation `Supplier` now stores the `nation` of each supplier and no longer employs the choice type. An evolution mapping consisting of two logical mappings (e_1 and e_2) can be established, to describe how data in `Source'` are related to data in `Source`. We point out that such evolution cannot be expressed as a list of schema changes (for example, the `suppkey` element in `Source'` corresponds to two elements in `Source`, depending on the value of `nation`). Thus, the mapping-based representation is strictly more expressive than the change-based representation of schema evolution.

Composing the evolution mapping $\{e_1, e_2\}$ with the original mapping $\{m\}$ will yield the following mapping m^a:

foreach l in `LineItem`, l' in `LineItem`, s_0 in `Supplier`
where l.partkey = l'.partkey and l'.suppkey = s_0.suppkey and s_0.nation = 'US'
exists os in `OrderUSSupp`
with os.orderkey = l.orderkey and os.supp_contact.phone = s_0.phone
and os.supp_contact.email = s_0.email

Coming up with such adaptation of m by hand would require a rather good understanding of the schemas and mappings. The task becomes increasingly challenging with increasingly complex schemas (which are common occurrences in practice). The rest of the paper will address the semantic, algorithmic and practical issues involved in the automatic derivation, by composition, of an adapted mapping such as m^a.

3 Semantics of Mapping Composition

There are two possible semantics that can be associated with mappings and mapping composition: the *relationship semantics* and the *transformation semantics*.

The first semantics [7, 15, 17] is a general notion that can be used when mappings describe arbitrary relationships between schemas. More concretely, a mapping M between a schema S and a schema T represents a binary relation between instances over S and instances over T: Inst(M) = $\{(I, J) \mid (I, J)$ satisfies $M\}$. Then, given a mapping M_{12} from S_1 to S_2, a mapping M_{23} from S_2 to S_3 and a mapping M_{13} from S_1 to S_3, we say that *M_{13} is the composition of M_{12} and M_{23}, with respect to the relationship semantics*, if M_{13} induces the *same* binary relation Inst(M_{13}) between instances of S_1 and S_3 as the binary relation that is the composition of Inst(M_{12}) and Inst(M_{23}). (The composition of binary relations P and Q is the binary relation $\{(x, z) \mid \exists y\ (x, y) \in P \wedge (y, z) \in Q\}$.)

The second semantics is applicable when mappings describe transformations. More concretely, a mapping M between a schema S and a schema T represents a transformation that, given

[3]Which we did not include here, for simplicity of presentation.

a source instance I, generates a canonical target instance $M(I)$. The process of generating $M(I)$ involves two steps. First, each logical mapping m in M is *skolemized*: all the atomic type elements that (1) are reachable via record projection from the target (i.e., exists) variables of m, and (2) are not (explicitly) assigned a source term by the with clause, are now assigned a term of the form $F(t_1, \ldots, t_k)$, where F is a new function symbol, and $t_1, \ldots, t_k$ are the source terms that appear in the with clause of m. Care must be taken so that the atomic elements that are required to be equal by the target where clause of m are assigned the same term. All the new functions are added to the already existing functions of M. The result of skolemization is a new set M' of logical mappings.

The target is then populated from the source instance I in a minimal way, by adding all the elements that are *required* by M'. The atomic components of the generated elements are either source values or ground function terms (formed with actual source values). Moreover, we perform PNF-based merging as in [22]. For an example, consider the set `Part` in Figure 2: all of its tuples that have the *same* `partkey` value will be merged into one, by combining all their supplier sets into one set. Essentially, we perform grouping by `partkey`. In general, such grouping is done based on all the atomic components of a tuple, and at every level (recursively). Moreover, this grouping is independent of the mappings, that is, we could be merging sets of suppliers coming from different mappings (and sources) as long as `partkey` is the same.

At the end, every ground function term is uniformly replaced, throughout the target instance, with a distinct *null*. The final result $M(I)$ is uniquely determined (up to renaming of nulls). The construction that we sketched here is similar to the process of chasing with second-order tgds described in [7], with the additional PNF-based merging.

The transformation semantics of composition is as follows. Given a mapping M_{12} from S_1 to S_2, a mapping M_{23} from S_2 to S_3 and a mapping M_{13} from S_1 to S_3, we say that M_{13} *is the composition of* M_{12} *and* M_{23}*, with respect to the transformation semantics,* if for every instance I_1 over S_1, we have that $M_{13}(I_1)$ is the same (up to renaming of nulls) as $M_{23}(M_{12}(I_1))$. Although less general, the transformation semantics has two main advantages: (1) it is better suited when mappings are intended to describe transformations, and (2) the resulting composition mapping can be greatly simplified. As a byproduct, the adapted mapping, in the case of schema evolution, can be much closer in syntax to the original mapping. This is a design issue that is quite important in practice.

Mapping reduction under transformation semantics Consider again the evolution scenario in Figure 3. There, the target S_2 of the mapping $\{m_1, m_2\}$ is evolved into a new target S_3, consisting of one relation `Takes'`, that is similar to the relation `Takes` in S_1 but has the extra `sid` element. The evolution mapping from S_2 to S_3 is $\{e\}$. There, `sid` is used to join students with enrollments to obtain the related course information. It can be shown that the composition of the mappings, with respect to the relationship semantics, is given by:

m: foreach t_1 in Takes, t_2 in Takes where F(t_1.name) = F(t_2.name)
exists t' in Takes'
with t'.sid = F(t_1.name) and t'.name = t_1.name and t'.course = t_2.course

This mapping is surprisingly complex, still correct. To understand this, consider an instance I_1 with tuples $[Alice, Math]$ and $[Mary, Art]$, and an instance I_3 with tuples $[X, Alice, Math]$, $[X, Alice, Art]$, $[X, Mary, Art]$, $[X, Mary, Math]$. Then, the pair (I_1, I_3) is in the composition of the binary relations associated with the two schema mappings. This is because there is an instance I_2, namely the one consisting of tuples $[X, Alice]$, $[X, Mary]$ in `Student`, and tuples $[X, Math]$, $[X, Art]$ in `Enrolls`, such that (I_1, I_2) satisfies $\{m_1, m_2\}$ and (I_2, I_3) satisfies $\{e\}$. The first satisfaction is true under an interpretation of F for which $F(Alice) = X$ and $F(Mary) = X$. (The second satisfaction is obvious.) In contrast, the pair (I_1, I_3'), where I_3' contains just the tuples $[X, Alice, Math]$ and $[X, Mary, Art]$, is not in the composition relation. Otherwise, the only way for (I_1, I_3') to be in the composition relation would be to pick F such that $F(Alice) = F(Mary) = X$ and then the other two tuples $[X, Alice, Art]$ and $[X, Mary, Math]$ would also have to be part of I_3', which is not the case. The above mapping m correctly distinguishes between the case of (I_1, I_3) and the case of (I_1, I_3') by explicitly requiring the existence of the two extra tuples (since $F(Alice) = F(Mary)$).

However, for data transformation purposes, mapping m is unnecessarily complex and unintuitive. A typical user will often expect to see an "identity" mapping between S_1 and S_3. In fact, the canonical transformation from S_1 to S_3 based on m will never assign the same id (via F) to two different names. (In the canonical target instance $m(I_1)$, with I_1 as above, $F(Alice)$ and $F(Mary)$ will be two *distinct* ground terms.) Thus, for canonical transformation purposes, F is not arbitrary but one-to-one. This enables the reduction of m to:

m': foreach t_2 in Takes
exists t' in Takes'
with t'.sid = F(t_2.name) and t'.name = t_2.name and t'.course = t_2.course

In a nutshell, since F is one-to-one, the equality F(t_1.name) = F(t_2.name) in m can be replaced by t_1.name = t_2.name. Then the first generator (t_1) over `Takes` becomes redundant and can be eliminated. This step is akin to minimization of conjunctive queries ([4], see also [5] for the nested case). We note that m and m' are *not* equivalent in general, but they are equivalent under the transformation semantics.

The simplicity of the resulting mapping becomes a stronger argument when schemas and mappings are complex and the reduction accomplished by such minimization can be significant. Although minimization is expensive, its usefulness forces us to investigate techniques for making the resulting method as efficient as possible (Section 5).

Finally, we note that for the example in Figure 1, the schema mapping $\{m^a\}$ given in Section 2.2 is the composition with respect to both semantics (and the same happens for the example in Figure 2 and the corresponding $\{m^a\}$). From now on, we focus on the transformation semantics.

4 Mapping Composition Algorithm

In this section we articulate the observation that the query rewriting algorithm given in [22] can be used to compose mappings. Given the syntactic similarities between mappings and queries, it is not surprising that we are able to reuse that algorithm. There, to rewrite a query q_2 over a schema S_2 in terms of a schema S_1, based on a schema mapping M_{12}, a set of rules R_{12} is generated to compute, for each source instance I_1, a canonical target instance I_2, according to the transformation semantics. Then q_2 can be rewritten into a source query q_1 by essentially performing substitution with R_{12}. The same machinery works here, except that instead of rewriting a query over S_2 we rewrite a schema mapping M_{23} from S_2 to a third schema S_3. We then show that

the resulting algorithm serves our purpose, that is, the algorithm yields a correct composition M_{13}, with respect to the transformation semantics. (For query rewriting, the semantics of the rewritten q_1 is that it computes the "right" answers of q_2; in particular, if q_2 is conjunctive, q_1 can be used to retrieve the *certain answers*, which is often assumed to be the standard semantics in LAV/GLAV systems).

We now give a brief review of the rewriting algorithm in [22], adapted for mapping composition. We assume that we need to compose two sequential mappings M_{12}, from schema S_1 to schema S_2, and M_{23}, from schema S_2 to schema S_3. The algorithm works in three phases.

Phase I: Rule generation We first skolemize M_{12} into M'_{12} by using the method described in the previous section. We then generate a set R_{12} of mapping rules for all the *set* elements of S_2 that are referred to in M'_{12}. Considering Figure 2 and e_1 only, the following rules for `Part` and `Part/suppliers` of `Source` are created:

Part = for l_0 in LineItem, s_0 in Supplier
where l_0.suppkey = s_0.suppkey and s_0.nation = 'US'
return [partkey = l_0.partkey, suppliers = $SK_s(l_0.\text{partkey})$]

$SK_s(p)$ = for l_1 in LineItem, s_1 in Supplier
where l_1.suppkey = s_1.suppkey and s_1.nation = 'US'
and $p = l_1$.partkey
return [supp = ⟨ US_supp = s_1.suppkey ⟩]

In general, a rule defining a set element comprises a *union* of for-where-return queries (depending on how many logical mappings refer to that element). Moreover, a set element that is not top-level (e.g., `Part/suppliers`) is defined as a *parameterized* view (e.g., $SK_s(p)$, where SK_s is a new Skolem function symbol that is uniquely associated to `Part/suppliers`, and p is the parameter). In our first rule, for each l_0.`partkey` we must create an instance of `Part/suppliers` (which will contain the group of suppliers for the given `partkey`). The creation of such instance is then expressed by "invoking" SK_s with the actual parameter l_0.`partkey`. Note how this achieves the PNF-based merging described earlier. Finally, for each atomic element that must be output in the return clause, we use the corresponding source term that appears in the with clause of the logical mapping.

Phase II: Substitution The set R_{12} of rules is then used to *translate* all references to S_2 that occur in M_{23} to references to S_1. This is accomplished by iteratively substituting set elements (`Part`, `Part/suppliers`, etc.) in the logical mapping (m) with their corresponding rule expressions. The set type Skolem functions introduced in the rules play a role only during translation and do not appear in the final logical mappings. Because a rule usually has a union of queries, each m in M_{23} will be translated into several logical mappings. We take M_{13} to be the union of all these logical mappings.

Phase III: Reduction For each of the logical mappings in M_{13} we now perform mapping reduction, along the lines discussed in Section 3. This reduction looks at the equalities in the source where clause of the logical mapping. All equalities between function terms are replaced (recursively) by the equalities of their arguments, whenever the function symbols are the same. Furthermore, any equality between terms with non-matching function symbols is regarded as unsatisfiable (since such non-matching function symbols will always produce distinct ground terms, under the transformation semantics) and the respective logical mapping in M_{13} is dropped. After all such equalities are processed, a logical mapping that is not dropped undergoes minimization to find its minimal equivalent form.

We remark that it is possible that the above composition algorithm starts with schema mappings containing no functions, but the resulting composition contains functions. This can happen because functions can be introduced during the skolemization step in Phase I. Even though our algorithm makes no attempt to de-skolemize the resulting mappings, in general a complete de-skolemization may not be possible (i.e., the functions strictly increase the expressive power of mappings).

The following theorem asserts the correctness of the composition algorithm. The proof uses the fact that the evaluation of the generated rules, R_{12}, on an instance I_1, coincides (up to null renaming) with $M_{12}(I_1)$ (as defined by the transformation semantics). Hence, $M_{23}(M_{12}(I_1))$ is the same as $M_{23}(R_{12}(I_1))$, for every I_1. Next, we use the fact that every step used in the substitution phase preserves the equivalence of mappings, when viewed as constraints. Finally, we use the fact that reduction preserves mapping equivalence under the transformation semantics.

Theorem 4.1 *If M_{13} is the result of applying the composition algorithm to M_{12} and M_{23}, then M_{13} is the composition of M_{12} and M_{23} with respect to the transformation semantics.*

5 Implementing a Practical Composition-Based Mapping Adaptation System

Although adaptation with the composition algorithm described in Section 4 (referred to as the full adaptation) captures better semantics, it is of little use unless we can have an efficient system implementation. We now describe the issues involved in building a practical mapping adaptation system.

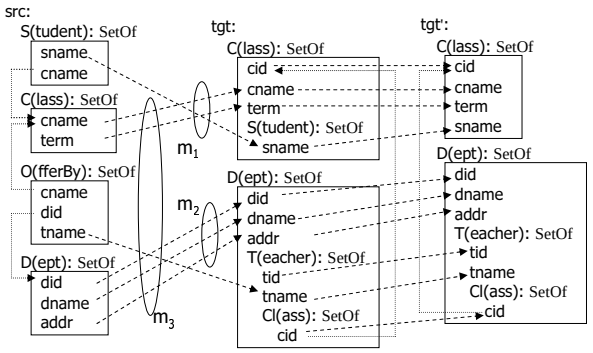
Figure 4 shows a more complex example of target evolution[4] that we will use throughout this section. The original mapping involves two schemas, `src` and `tgt`, and three logical mappings $\{m_1, m_2, m_3\}$. Both schemas describe information about departments and classes: `src` uses a relational structure while `tgt` uses a more hierarchical structure. The logical mappings are as follows: m_1 maps classes along with associated students; m_2 maps departments alone; and the more complex m_3 maps both department and classes, associated via the intermediate relation `OfferBy`. The `tgt` schema is then evolved into `tgt`′ where the original hierarchy of classes and students becomes a flat structure.

5.1 Deriving Evolution Mappings

Before mapping composition can be applied, an *evolution mapping* must be established. However, schema evolution is rarely represented as a mapping in practice. Instead, it is either represented as a list of changes (CBR) or, more often, implicitly embedded in the new version of the schema. Still, one can derive an evolution mapping from the change list or by comparing the two versions of the schema. Next, we sketch the algorithm that we implemented for the semi-automatic generation of an evolution mapping, given two schema versions.

We start with a simple schema matching phase, where the schemas are simultaneously navigated and VCs between elements with the same *absolute* path in both schemas are established (e.g., $\langle /C/cid \Rightarrow /C/cid \rangle$). This default set of VCs then undergo inspection by the user, who may drop VCs between elements that no longer correspond to each other and

[4] Although we focus on target evolution in this section, all the techniques described will apply with minor adjustments to source evolution.



m_1: foreach s in src.S, c in src.C where s.cname = c.cname
exists c' in tgt.C, s' in c'.S
with s'.sname = s.sname and c'.cname = c.cname and c'.term = c.term

m_2: foreach d in src.D exists d' in tgt.D
with d'.did = d.did and d'.dname = d.dname and d'.addr = d.addr

m_3: foreach c in src.C, o in src.O, d in src.D
where c.cname = o.cname and o.did = d.did
exists c' in tgt.C, d' in tgt.D, t' in d'.T, cl' in t'.Cl
where c'.cid = cl'.cid
with c'.cname = c.cname and c'.term = c.term and d'.did = d.did and d'.dname = d.dname and d'.addr = d.addr and t'.tname = o.tname

Figure 4: A more complex evolution scenario

add VCs between elements that correspond to each other (e.g., $\langle /C/S/sname \Rightarrow /C/sname \rangle$). The user may also specify arbitrary elements in the two schemas as starting points of matching. In this way, VCs between elements with the same *relative* path (from the two respective starting points) can be established automatically. Finally, a default set of logical mappings (which could be further modified by a user) is automatically generated from the final set of VCs, based on an existing generation algorithm [18]. The following logical mappings are created for our example (the with clauses can be inferred from the VCs between `tgt` and `tgt'` and are omitted):

e_1: foreach c in tgt.C exists c' in tgt′.C with ...

e_2: foreach c in tgt.C, s in c.S exists c' in tgt′.C with ...

e_3: foreach d in tgt.D exists d' in tgt′.D with ...

e_4: foreach d in tgt.D, t in d.T exists d' in tgt′.D, t' in d'.T with ...

e_5: foreach c in tgt.C, d in tgt.D, t in d.T, cl in t.Cl where cl.cid = c.cid
exists c' in tgt′.C, d' in tgt′.D, t' in d'.T, cl' in t'.Cl where cl'.cid = c'.cid
with ...

These mappings enumerate the basic ways in which `tgt` data can map to `tgt'` data, based on the structure and constraints of the schemas. None of these formulas can be removed, although some are seemingly redundant. For example, e_1 maps `Class` while e_2 maps both `Class` and `Student`. However, e_1 is required since only it can map classes without associated students. With no further information, both mappings are needed for the adaptation, to avoid losing valid adapted mappings.

5.2 Mapping Pruning

Full adaptation (i.e., composition of the original mapping with the evolution mapping) can become inefficient when the schemas involved are complex, even when the schema changes are relatively small. The inefficiency can be easily understood: the full adaptation adapts all of the original mappings using all of the evolution mappings even though some of the original mappings are not affected by the changes and therefore require no adaptation. In this subsection, we present a mapping pruning method that filters out not only the *unaffected* original mappings, but also the evolution mappings that do not need to participate in the adaptation. As a result, it significantly reduces the workload (i.e., mappings to be composed).

5.2.1 Unaffected Mappings

We first define the notion of (un)changed schema elements, based on the VCs that are established between the original and the evolved schemas (see Section 5.1). An **unchanged element** is an original schema element e for which a corresponding element e' with the same path exists in the evolved schema, and for which the only VC involving e is the "identity" VC between e and e'. All other elements are **changed elements**, which are original schema elements whose paths are no longer valid or whose semantics have been dropped, altered, or shared. For example, `/C/S/sname` is a changed element because the path is no longer valid in the evolved schema `tgt'`.

In our algorithm, instead of identifying all changed elements, it is enough to identify only the **changed set elements**. (We ignore *choice* elements in this discussion; they are treated similarly to the way set elements are treated.) A changed set element is defined as an original element of set type that is either (1) a changed element itself, or (2) has at least one directly reachable[5] non-set element that is a changed element. For example, `/C/S` is a changed set element. An **affected mapping** is simply an original mapping which uses at least one changed set element. For our example, m_1, which uses the changed element `/C/S` in the exists clause, is the only mapping affected.

Unaffected mappings provide the starting point for mapping pruning: they can be removed from the adaptation workload. Furthermore, their removal may in turn render certain evolution mappings unnecessary (because the participation of those evolution mappings in the adaptation does not give rise to any valid or non-redundant adapted mapping). However, those unnecessary mappings can not be detected just by checking for changed elements. To correctly detect which mappings are necessary, we must first understand the role that each mapping plays during composition, through the concepts of *composability links* and *conditional mapping containment*.

5.2.2 Composability Links and Conditional Containment

Let us consider performing the full adaptation, without mapping pruning, on the evolution example in Figure 4. For each evolution mapping, the composition algorithm iterates through all generators and substitutes each set expression with the rule defining that set. Consider the evolution mapping e_1. Its sole generator involves the set element `C`, for which a rule with two terms exists, one for each of the two mappings m_1 and m_3 that "map" into `C`. Hence, the composition algorithm replaces e_1 with two logical mappings, one for each choice (m_1 or m_3). In general, the algorithm continues until all generators are translated. The translation process can thus be visualized as a tree, where each root-to-leaf path (branch) corresponds to one of the final (fully translated) logical mappings. For each such branch, we record the set of all of the original mappings that are chosen at some substitution step along the path.

Definition 5.1 *A* **composability link** *(CL) is a pair: from $\{m_1, \ldots, m_n\}$ to m_e, such that m_e is an evolution mapping, and there is a branch in the translation of m_e for which $m_1, \ldots, m_n$ are all the original mappings that were used.*

Generating all the possible CLs can be done efficiently by simply analyzing the set elements involved in mappings, without performing the actual translation itself. For our example, we can establish the following CLs:

[5] An element `/.../B/`C_1`/`C_2`/.../`C_n`/A` is *directly reachable* from `B` if all C_i are record type elements.

Input: original and evolution mappings $\mathcal{M}_{ori}, \mathcal{M}_{evo}$,
the set of changed set elements E, the set of composability links CLS,
the set of conditional containments CCS,
1. Initialize M_o = {}, m_e = {}; // final adaptation workloads
2. Initialize $M_u = \mathcal{M}_{ori}$;
3. **foreach** $m \in \mathcal{M}_{ori}$: // remove affected original mappings from M_u
4. **if** m.exists uses $e \in E$: remove m from M_u; **break**;
5. **for each** $l \in CLS$, **each** $m \in l$.from:
6. **if** $m \in M_u$: remove m from l.from;
7. **foreach** $l_1 \in CLS$:
8. **if** l_1.from is empty: remove l_1 from CLS; **continue**;
9. **if** l_1.to can not be fully translated with l_1.from:
10. remove l_1 from CLS; **continue**;
11. **foreach** $l_2 \in CLS$: **if** $l_1 = l_2$: remove l_2 from CLS;
12. **foreach** $\langle m_1, m_2, C\rangle \in CCS$: // m_2 contains m_1 under C
13. **foreach** $(l_1, l_2), l_1, l_2 \in CLS$ **and** l_1.to $= m_1$ **and** l_2.to $= m_2$:
14. contained = **true**;
15. **foreach** $m \in l_1$.from: **if** m.exists **not satisfy** C: contained = **false**; break;
16. **if** contained: remove l_1 from CLS;
17. **foreach** $l \in CLS$:
18. $M_o = M_o \cup l$.from; $M_e = M_e \cup l$.to;
19. **return** M_o, M_e (subsets of $\mathcal{M}_{ori}, \mathcal{M}_{evo}$, respectively).

Figure 5: Algorithm MappingPruning

e_1: L_1 from $\{m_1\}$ to e_1; L_2 from $\{m_3\}$ to e_1
e_2: L_3 from $\{m_1\}$ to e_2; L_4 from $\{m_1, m_3\}$ to e_2
e_3: L_5 from $\{m_2\}$ to e_3; L_6 from $\{m_3\}$ to e_3
e_4: L_7 from $\{m_3\}$ to e_4; L_8 from $\{m_2, m_3\}$ to e_4
e_5: L_9 from $\{m_3\}$ to e_5; L_{10} from $\{m_1, m_3\}$ to e_5
L_{11} from $\{m_2, m_3\}$ to e_5; L_{12} from $\{m_1, m_2, m_3\}$ to e_5

While CLs represent dependencies between original and evolution mappings, they do not capture dependencies among evolution mappings. Consider e_1 and e_2. Without any condition, both mappings are necessary since e_1 maps classes that do not satisfy the condition imposed in e_2 (i.e., having at least one enrolled student). However, if the original mappings participating in the adaptation guarantee that no class will be empty, e_1 becomes redundant and can be eliminated.

Definition 5.2 *A* **conditional mapping containment** *is a triplet $\langle m_1, m_2, C\rangle$ satisfying: for every instance I satisfying constraint C, $m_1(I)$ is a sub-instance of $m_2(I)$. We also say that m_1 is contained in m_2 under C.*

We have developed an algorithm that can detect potential containment relationships between evolution mappings along with the condition under which the containment is valid. The main idea is to detect mappings whose generators and conditions can all be matched against another mapping, and the extra condition in the other mapping naturally becomes the containment condition. Applying the algorithm to our example, we obtain the following conditional containments:

$\langle e_1, e_2,\ \forall (c{\in}tgt.\text{C})\ \exists (s{\in}c.\text{S})\ \rangle$;
$\langle e_4, e_5,\ \forall (d{\in}tgt.\text{D})\ \forall (t{\in}d.\text{T})\ \exists (cl{\in}t.\text{Cl})\ \exists (c{\in}tgt.\text{C})\ cl.\text{cid} = c.\text{cid}\rangle$;
$\langle e_3, e_4,\ \forall (d{\in}tgt.\text{D})\ \exists (t{\in}d.\text{T})\ \rangle$;
$\langle e_3, e_5,\ \forall (d{\in}tgt.\text{D})\ \exists (t{\in}d.\text{T})\ \exists (cl{\in}t.\text{Cl})\ \exists (c{\in}tgt.\text{C})\ cl.\text{cid} = c.\text{cid}\rangle$.

The algorithm does not guarantee to detect all possible conditional containments (i.e., it is not complete). However, it does detect most of the containments that can be utilized by the mapping pruning algorithm, which we discuss next.

5.2.3 Mapping Pruning Algorithm

The mapping pruning algorithm (Figure 5) first identifies all the original mappings that are unaffected by the change (lines 3-4) and then removes them from the workload as well as from all the composability links (lines 5-6). For our example, m_2 and m_3 will be detected and removed from multiple composability links (e.g., L_2, L_{12}, etc.). Next, all composability links are examined for the impact of the previous removal (lines 7-11). In our example, after the removal of m_2 and m_3, among the 12 composability links: the from clauses of $L_{2,5-9,11}$ become empty; L_4 becomes a duplicate of L_3; and the from clauses of $L_{10,12}$ become insufficient to translate the respective evolution mappings (e.g., translating e_5 requires a rule for /D/T, which can not be provided by m_1, the only mapping left in the from clause of L_{10}). All these composability links are removed, which leaves us with L_1 and L_3, involving one original mapping, m_1, and two evolution mappings, e_1 and e_2. Next, L_1 and L_3 are checked for redundancy according to the conditional containments (lines 12-16). We have previously identified that e_2 contains e_1 under the condition: $\forall (c{\in}tgt.\text{C})\ \exists (s{\in}c.\text{S})$. The condition is satisfied since the only mapping m_1 in from clauses of L_1 and L_3 ensures that all the classes in tgt will have an associated student (see m_1.exists in Figure 4). As a result, L_1 is removed. The detection of the condition satisfaction is standard: we check whether the canonical database that can be associated with m_1.exists (together with the corresponding where clause, when it exists) satisfies the constraint.

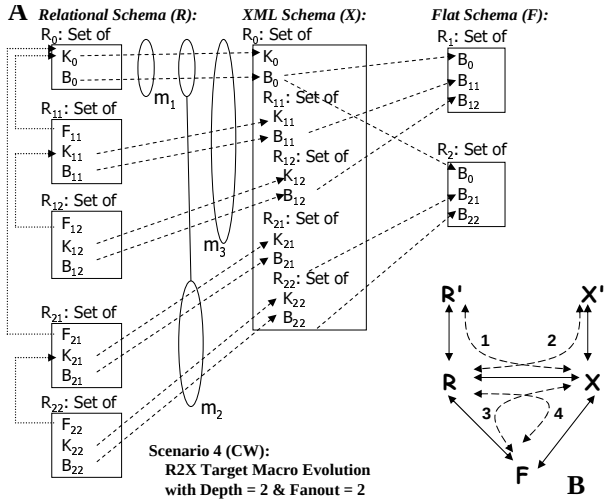
Finally, the algorithm collects the original and evolution mappings from the from and to clauses of the remaining composability links: those mappings will be the final workload for adaptation. Because only L_3 remains in our example, *the adaptation only needs to compose m_1 with e_2*, a significant reduction from the original workload (three original and five evolution mappings). Although pruned from the workload of adaptation, the removed original mappings will still be part of the final adapted mappings: they are simply unchanged.

6 Experimental Evaluation

To evaluate the performance of our approach, we implemented the MACES (Mapping Adaptation, using Composition, for Evolving Schemas) system and tested it with a comprehensive set of synthetic and real life experiments. We show that when the mapping pruning techniques are incorporated, the system scales well with increasing mapping complexity in most of the synthetic scenarios and can efficiently adapt mappings in the two real application scenarios. The system is implemented using Java and all experiments were performed on a Windows XP SP2 machine (2.0GHz P4 CPU, JRE 1.4.2, 384MB VM).

6.1 System Scalability

The synthetic evolution test case examines the scalability of MACES. It covers a range of complexity levels (measured by schema depth and fanout, explained shortly), where each level consists of eight different scenarios. Shown in Figure 6(B), the scenarios are created from five schemas: R and X, representing the original relational and XML schemas respectively; R′ and X′, representing evolved (via a single renaming operation) versions of R and X respectively; and F, representing a schema that has evolved significantly from R and X. Ten mappings (directed solid lines) are established: R→X and X→R represent the two original mappings and the rest represent evolution mappings (e.g., R→R′ and F→X). Each scenario (directed dashed lines) involves three schemas and two mappings. For example, following dashed line #4 in the clock-wise direction, we obtain a target macro evolution scenario where R is mapped into X, and X is subsequently evolved into F. This scenario is also detailed in Figure 6(A): R contains one central element (relation R_0), to which a number of "chains" of elements (R_{11}-R_{12}, R_{21}-R_{22}) are associated, and elements in each chain are linked via referential constraints; X consists of one root element (R_0), which contains a number of child elements (R_{11}, R_{21}), and each child element itself leads a chain



1: (CW) X2R target micro evolution (CCW) R2X source micro evolution
2: (CW) X2R source micro evolution (CCW) R2X target micro evolution
3: (CW) R2X source macro evolution (CCW) X2R target macro evolution
4: (CW) R2X target macro evolution (CCW) X2R source macro evolution
(CW): clock wise direction; (CCW): counter clock wise direction

Figure 6: Details (A) and overview (B) of the synthetic scenarios.

of elements connected via parent/child relationships; F flattens each chain into a single relation, while maintaining the number of chains. The original mapping (R→X) contains manually generated logical mappings that together fully map the corresponding elements in both schemas. The evolution mapping (X→F) contains logical mappings that are automatically generated as described in Section 5.1. The *complexity level* is determined by the depth and fanout of the schemas. Depth is the number of set elements within each chain, while fanout is the number of chains in the schema. In each scenario, R and X have the same depth and fanout; and the evolved schema has comparable size with the original schemas (F has a proportionally smaller size, while R′ and X′ is as complex as R and X respectively).

Increasing fanout leads to increased number of logical mappings and VCs in both original and evolution mappings, while increasing depth leads to not only increased number of logical mappings but also increased complexity of each logical mapping (larger foreach and exists clauses). Table 1 summarizes the statistics of synthetic scenarios at median complexity levels. For all micro evolution scenarios, a single leaf level VC is affected, affecting a single original logical mapping. For all macro evolution scenarios, all VCs are affected (the worst case scenario), affecting all original logical mappings.

Performance of adaptation Figure 7 evaluates MACES (with mapping pruning) on a series of evolution scenarios with increasing schema depth or fanout. We show the performance of adapting X→R in the case of source evolution (left y-axis, linear scale) and adapting R→X in the case of target evolution (right y-axis, log scale). The symmetric cases (i.e., source evolution of R→X and target evolution of X→R) will be discussed shortly.

Scenario (Depth × Fanout)	Schema Size	Mappings [#VCs]
8 × 3	87	13 [50]
3 × 25	265	51 [152]

Table 1: Statistics for the synthetic scenarios. Schema size is the average number of elements in both original schemas. Mappings and #VCs are the number of logical mappings and VCs in the original mapping. The statistics are independent of the mapping direction (R→X or X→R) or the evolution type (source or target).

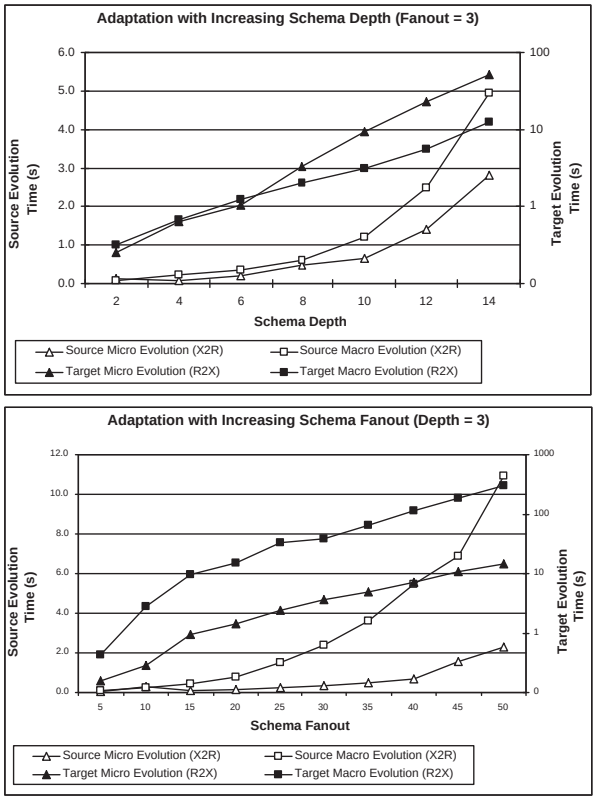


Figure 7: Evaluation of mapping adaptation.

As shown, the system scales reasonably well with the increasing complexity. For a rather complex target macro evolution scenario with schema fanout at 50 (101 logical mappings and 302 VCs in the original mapping), MACES finishes the adaptation in less than 5 minutes.

The results also show that, despite being more complex in nature, adapting macro evolution is not a lot worse than adapting micro evolution. In the case of target macro evolution with deep schemas, the performance is even better than that of the corresponding micro evolution. This is due to an optimization technique, *rule simplification*, employed by MACES to simplify mapping rules that include union. The simplification is done by removing the terms in the union that are contained in (subsumed by) other terms in the union. The terms that are eliminated are the larger ones (with more generators and conditions, thus more restrictive); this results in faster subsequent processing. For target micro evolution, most of the logical mappings in the original mapping are pruned out, yielding mapping rules that contain fewer (but larger) terms in the union, with fewer opportunities for rule simplification. Rule simplification also explains why adapting source evolution can be faster than adapting target evolution (because there are usually more mappings in the evolution mapping and rule simplification has more opportunities to eliminate larger terms). This irregularity shows that mapping pruning has to be intelligently tuned to act in synergy with other optimization techniques, which is one of our future research interests.

Minimization cost We consider now the symmetric measurements: target evolution of X→R and source evolution of R→X. For most part, the results are similar to those in Figure 7. There is one exception: the performance of adapting X→R in target evolution with increasing schema depth is worse than that of adapting the R→X counterparts. After careful analyses, we found

Mapping Scenario (Depth × Fanout)	Full Adaptation	Mapping Pruning	Saving %
Source Micro X2R (4 × 3)	132.14	0.09	99.93%
Target Micro R2X (4 × 3)	21.19	0.64	96.98%
Source Micro X2R (3 × 10)	267.59	0.31	99.88%
Target Micro R2X (3 × 10)	62.66	0.28	99.55%
Source Macro X2R (4 × 3)	12.32	0.23	98.13%
Target Macro R2X (4 × 3)	28.47	0.68	97.61%
Source Macro X2R (3 × 10)	4.90	0.26	94.69%
Target Macro R2X (3 × 10)	85.75	2.79	96.75%

Table 2: Costs (seconds) of full adaptation and mapping pruning.

two main reasons for this. First, unlike increasing fanout, which mainly leads to increased number of logical mappings to be adapted, increasing depth leads to both increased number of logical mappings and, more significantly, increased complexity of each logical mapping. Second, the logical mappings that result after composing the logical mappings in $\mathsf{X}\rightarrow\mathsf{R}$ with the corresponding target evolution logical mappings involve deeply nested elements in the source schema. Hence, these logical mappings contain long chains of source generator dependencies (i.e., one generator depends on another). We found that, for our current implementation of minimization[6], the time to minimize such logical mappings is consistently worse than the time to minimize logical mappings with no dependencies between generators. As a result, adapting $\mathsf{X}\rightarrow\mathsf{R}$ target evolution scenarios at deep schemas becomes the worst case scenario (worse than source evolution due to reduced applicability of rule simplification). In terms of actual numbers, at depth 6, the cost of minimization dominates the overall cost and exceeds the 60 minutes threshold. An alternative is to leave these mappings un-minimized. But this is not acceptable since the number of redundant generators that result after composition is often high. Improving the performance of minimization remains an interesting research issue.

Impact of mapping pruning To understand the impact of mapping pruning on the adaptation, we compared the time cost of adaptation with and without pruning (full adaptation). As expected, pruning significantly affects scalability. In evolution scenarios of medium to high complexity (e.g., over 6×3 or 3×20), full adaptation failed to finish within the 60 minutes time limit we set. Most of the time spent by full adaptation is on minimization of intermediate logical mappings that result after composition. Mapping pruning reduces the number of logical mappings to be composed and to be minimized. Table 2 lists the time costs for both full adaptation and adaptation with pruning for some of the scenarios where full adaptation is able to finish. (The full adaptation costs for source macro evolution are significantly better than the corresponding costs for source micro evolution; this is because, in the former case, the macro evolved schema (F) has less set elements than the micro evolved schema (X'), hence the evolution mappings have less generators and are therefore easier to process.) As shown, mapping pruning can significantly improve the performance of adaptation, often by several orders of magnitude.

6.2 Mapping Adaptation with Real Cases

We test MACES on two real life mapping cases. The first case is Mondial [14], a geographical database with both a relational schema and an XML schema. The original mapping is established between the XML schema (source) and the relational schema (target). The changes are introduced into the XML schema in two ways: renaming an element to create the micro-evolved XML schema and adding a new `terrain` element serving as the parent of all geological elements (e.g., `river`) to create the macro-evolved schema. We use this real case to represent scenarios where the data are ingested from XML sources and subsequently stored into a target relational database: when the XML sources evolve, the mappings between the sources and the relational target must be adapted.

[6]A necessarily exponential-time procedure, in general (unless P = NP).

Scenario	Mondial		Faculty	
	Micro	Macro	Micro	Macro
Size [Depth, Fanout]	123 [5, 19]		57 [4, 7]	
Original Mappings [#VCs]	15 [53]		11 [61]	
Affected VCs [%]	1 [2%]	37 [70%]	1 [2%]	36 [59%]
Full Adaptation	DNF	DNF	1.33	2.09
Mapping Pruning	1.89	9.19	0.35	1.23
Saving %	>99%	>99%	74%	41%
Blank-Sheet Mappings	14	14	11	13
Missed Mappings	1	1	0	10
Unintended Mappings	0	0	0	5
Unchanged Mappings	14	10	10	7
Adapted Mappings	1	5	1	11
Benefits	93%	67%	91%	52%

Table 3: Statistics for Mondial and Faculty evolution scenarios, and performance (seconds) and benefits of adaptation. Size, depth, and fanout are the average of both original schemas (except for Mondial, where the depth is the depth of source XML schema since the target schema is relational and has depth of 1 only). DNF indicates the adaptation was not finished within 60 minutes time limit.

The second case is Faculty, containing a list of schemas about CS faculty members of six major universities[7]. The schemas, which are constructed strictly based on the webpages, differ in terms of both schema size (from less than 5 elements to more than 20 elements) and how the information is represented (e.g, some have detailed research profile, others only a short description). A simple unioned schema and a deeply merged schema of the six are manually created, and the original mapping is established between the unioned schema (source) and the merged schema (target). The merged schema is then evolved via restructuring operations (macro evolution) or renaming operations (micro evolution). This represents common data integration scenarios where multiple data sources are mapped into a target: when the target evolves due to data reorganization or the addition of new sources, mappings between the old sources and the target must be adapted.

Performance As shown in Table 3, all adaptations with mapping pruning finish within a reasonable amount of time, with the Mondial macro evolution taking the most time at 9.19 seconds. The performance difference between the Mondial and Faculty scenarios is largely due to the fact that logical mappings in Mondial are more complex than those in Faculty. The improvements from mapping pruning are again clear in Mondial, saving at least 99% in both micro and macro evolutions. In fact, full adaptations of the scenario failed to finish within the time limit (60 minutes). The improvements are not so significant in Faculty because all the schemas are relatively simple, leading to reduced impact of pruning. In general, we expect many real schemas to be more complex than Mondial–the cost of full adaptation will be very high if not too high.

Benefits We measured the benefits of mapping adaptation over reconstructing the mappings using blank-sheet approach. We consider the fact that logical mappings created by the blank-

[7]The six universities are Berkeley, Michigan, North Carolina, UIUC, Washington, and Wisconsin. Snapshots were taken as of October 2004.

sheet approach all require user examination (to discard unintended mappings[8]) with extra effort devoted to search for mappings not automatically generated (missed mappings). In contrast, mappings generated by MACES fall into two categories: 1) unchanged, which do not need to be examined; 2) adapted, which may need to be examined if the user so chooses. Note that our mapping adaptation approach does not miss potentially valid mappings because it captures the semantics of both the original mappings and the evolution. Although the mapping generation tool being used in the comparison is *Clio* [18], we believe the trade-off between reducing the number of unintended mappings and reducing the number of missed mappings is universal for all mapping generation tools. As a result, the benefit analysis here can be similarly applied to other mapping systems. We also believe that the cost of the schema matching phase (Section 5.1) for mapping adaptation will be similar to the cost of manually creating the change list for the blank-sheet approach; both costs are therefore ignored in our calculation. Hence, the benefit of adaptation (effort saved using adaptation compared to using blank-sheet approach) can be defined as:

$$1 - \frac{mappings_{adapted}}{mappings_{blank-sheet} + mappings_{missed}}$$

As shown in Table 3, for the two real cases, the benefits of adaptation are around 90% and at least 50% for micro evolution and macro evolution scenarios, respectively. There are several interesting points regarding the benefits of adaptation. (1) The regeneration algorithm often produces some of the same exact mappings that the adaptation approach yields. This is true especially at small schema changes. However, there are no guarantees in general and, when schema changes become more drastic, the difference between the two approaches becomes larger (see the last point). (2) In most cases, the user is safe to choose not to examine the adapted mappings since those mappings capture the semantics of both the original mapping and the schema evolution. In general, we expect the users to do so and the benefit of adaptation will become 100% for those users. (3) The benefit formula underestimates the cost associated with searching for missed mappings in the blank-sheet approach, which can be a very expensive operation. (4) When schema changes are small (the micro evolution case), the benefit is high (close to 100%). This is due to the fact that there is a large number of mappings that do not need to be changed; the mapping adaptation process detects them and hence saves the cost of regenerating and re-examining them. (5) When the schema structure is drastically changed (e.g., Faculty macro evolution), the number of missed/unintended mappings increases greatly in the blank-sheet approach. This further validates that the blank-sheet approach can not capture the semantics of both the original mapping and the schema evolution, while the composition based adaptation is able to.

7 Conclusion

We addressed the problem of adapting schema mappings when schemas evolve, by using a mapping composition approach. We showed that our method is superior to a previous incremental approach by capturing the semantics of both the original mapping and the evolution. Within the context of mapping adaptation, we studied the *relationship* and *transformation* semantics of mapping composition. We implemented the MACES mapping adaptation system based on mapping composition under the transformation semantics, and designed *mapping pruning* techniques to improve the performance. Experimental analysis showed the overall approach to be scalable and practical in various evolution scenarios. One of the important challenges remaining is improving the performance of the minimization that is needed during composition.

[8]Those do not reflect the semantics of the original mappings but rather represent new mapping semantics, given the new schema structure.

References

[1] J. Banerjee, W. Kim, H.-J. Kim, and H. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *SIGMOD*, 1987.

[2] P. Bernstein. Applying Model Management to Classical Meta Data Problems. In *CIDR*, 2003.

[3] P. A. Bernstein and E. Rahm. Data Warehouse Scenarios for Model Management. In *ER*, 2003.

[4] A. K. Chandra and P. M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *STOC*, pages 77–90, 1977.

[5] A. Deutsch, L. Popa, and V. Tannen. Physical data independence, constraints, and optimization with universal plans. In *VLDB*, 1999.

[6] R. Fagin, P. Kolaitis, R. Miller, and L. Popa. Data exchange: Semantics and query answering. In *ICDT*, 2003.

[7] R. Fagin, P. Kolaitis, L. Popa, and W.-C. Tan. Composing Schema Mappings: Second-Order Dependencies to the Rescue. In *PODS*, 2004.

[8] A. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10:270–294, 2001.

[9] A. Lee, A. Nica, and E. Rundensteiner. The EVE Approach: View Synchronization in Dynamic Distributed Environments. *TKDE*, 14(5):931–954, 2002.

[10] M. Lenzerini. Data Integration: A Theoretical Perspective. In *PODS*, 2002.

[11] B. Lerner. A Model for Compound Type Changes Encountered in Schema Evolution. *TODS*, 25(1):83–127, 2000.

[12] J. Madhavan and A. Halevy. Composing mappings among data sources. In *VLDB*, 2003.

[13] I. Manolescu, D. Florescu, and D. Kossman. Answering XML queries over heterogeneous data sources. In *VLDB*, 2001.

[14] W. May. Information extraction and integration with FLORID: The MONDIAL case study. Technical report, Universität Freiburg, Institut für Informatik, 1999.

[15] S. Melnik. *Generic Model Management: Concepts and Algorithms*. PhD Thesis, University of Leipzig, Springer LNCS 2967, 2004.

[16] R. Miller, L. M. Haas, and M. Hernández. Schema mapping as query discovery. In *VLDB*, 2000.

[17] A. Nash, P. A. Bernstein, and S. Melnik. Composition of Mappings Given by Embedded Dependencies. In *PODS*, 2005.

[18] L. Popa, Y. Velegrakis, R. Miller, M. Hernández, and R. Fagin. Translating web data. In *VLDB*, 2002.

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

[20] N. Shu, B. Housel, R. Taylor, S. Ghosh, and V. Lum. EXPRESS: A Data EXtraction, Processing, and REStructuring System. *TODS*, 2(2):134–174, 1977.

[21] Y. Velegrakis, R. Miller, and L. Popa. Mapping Adaptation under Evolving Schemas. In *VLDB*, 2003.

[22] C. Yu and L. Popa. Constraint-Based XML Query Rewriting for Data Integration. In *SIGMOD*, 2004.