Schema Mapping Evolution through Composition and Inversion[⋆]

Ronald Fagin[1], Phokion G. Kolaitis[2,1], Lucian Popa[1], and Wang-Chiew Tan[1,2]

[1] IBM Almaden Research Center
[2] UC Santa Cruz
fagin@almaden.ibm.com, kolaitis@cs.ucsc.edu,
lucian@almaden.ibm.com, wctan@cs.ucsc.edu

Abstract. Mappings between different representations of data are the essential building blocks for many information integration tasks. A schema mapping is a high-level specification of the relationship between two schemas, and represents a useful abstraction that specifies how the data from a source format can be transformed into a target format. The development of schema mappings is laborious and time-consuming, even in the presence of tools that facilitate this development. At the same time, schema evolution inevitably causes the invalidation of the existing schema mappings (since their schemas change). Providing tools and methods that can facilitate the adaptation and reuse of the existing schema mappings in the context of the new schemas is an important research problem.

In this chapter, we show how two fundamental operators on schema mappings, namely composition and inversion, can be used to address the mapping adaptation problem in the context of schema evolution. We illustrate the applicability of the two operators in various concrete schema evolution scenarios, and we survey the most important developments on the semantics, algorithms and implementation of composition and inversion. We also discuss the main research questions that still remain to be addressed.

1 Introduction

Schemas and schema mappings are two fundamental metadata components that are at the core of heterogeneous data management. Schemas describe the structure of the various databases, while schema mappings describe the relationships between them. Schema mappings can be used either to transform data between two different schemas (a process typically called *data exchange* (Fagin et al, 2005a) or *data translation* (Shu et al, 1977)) or to support processing of queries formulated over one schema when the data is physically stored under some other schemas (a process typically encountered in *data integration* (Lenzerini, 2002) and also in *schema evolution* (Curino et al, 2008)).

[⋆] To appear in Z. Bellahsene, A. Bonifati, and E. Rahm, editors, *Schema Matching and Mapping*, Springer, 2011.

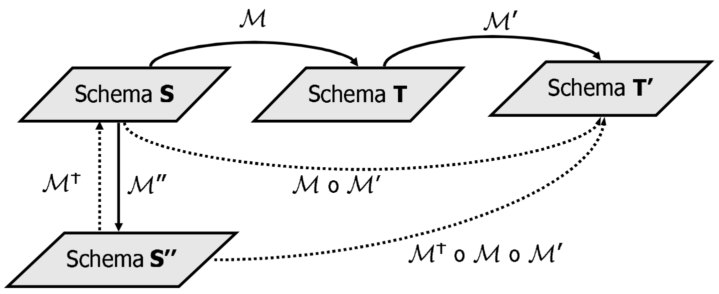

Fig. 1. Application of composition and inversion in schema evolution.

A schema mapping is typically formalized as a triple $(\mathbf{S}, \mathbf{T}, \Sigma)$ where $\mathbf{S}$ is a source schema, $\mathbf{T}$ is a target schema, and Σ is a set of dependencies (or constraints) that specify the relationship between the source schema and the target schema. Schema mappings are necessarily dependent on the schemas they relate. Once schemas change (and this inevitably happens over time), the mappings become invalid. A typical solution is to regenerate the mappings; however, this process can be expensive in terms of human effort and expertise, especially for complex schemas. Moreover, there is no guarantee that the regenerated mappings will reflect the original semantics of the mappings. A better solution is to provide principled solutions that reuse the original mappings and *adapt* them to the new schemas, while still incorporating the original semantics. This general process was first described in (Velegrakis et al, 2003), which called it *mapping adaptation* and also provided a solution that applied when schemas evolve in small, incremental changes. In this paper, we describe a more general formalization of the mapping adaptation problem where schema evolution can be specified by an arbitrary schema mapping. Under this formalization, which is in the spirit of model management (Bernstein, 2003), the new, adapted mapping is obtained from the original mapping through the use of schema mapping operators.

The two operators on schema mappings that we need to consider are *composition* (Bernstein et al, 2008; Fagin et al, 2005b; Madhavan and Halevy, 2003; Nash et al, 2005) and *inversion* (Arenas et al, 2008; Fagin, 2007; Fagin et al, 2008b, 2009b). These operators turn out to be quite fundamental, with many applications in metadata management (Bernstein, 2003) and for schema evolution. In particular, the two operators of composition and inversion provide a principled way to solving the problem of adapting a schema mapping when schemas evolve. We will use Figure 1 to describe, at a high-level, the operators of composition and inversion, and their application in the context of schema evolution. First, assume that we are given a schema mapping $\mathcal{M}$ (the "original" schema mapping) that describes a relationship or a transformation from a source schema $\mathbf{S}$ to a target schema $\mathbf{T}$. In order to "reuse" the original schema mapping $\mathcal{M}$ when schemas evolve, we need to handle changes in either the target schema or the source schema.

Target schema evolution. Assume that the target schema evolves to a new target schema $\mathbf{T}'$, and that we model this evolution as a schema mapping $\mathcal{M}'$ from $\mathbf{T}$ to $\mathbf{T}'$. Intuitively, $\mathcal{M}'$ is a new data transformation that converts instances of $\mathbf{T}$ to instances of $\mathbf{T}'$. Note that generating such $\mathcal{M}'$ is an instance of the general schema mapping creation problem and can be done manually or with the help of tools such as Clio, described elsewhere (Fagin et al, 2009a). Based on $\mathcal{M}$ and $\mathcal{M}'$, we can then obtain a new mapping from $\mathbf{S}$ to $\mathbf{T}'$ by applying the composition operator. Composition operates, in general, on two *consecutive* schema mappings $\mathcal{M}$ and $\mathcal{M}'$, where the target schema of $\mathcal{M}$ is the source schema of $\mathcal{M}'$. The result is a schema mapping $\mathcal{M} \circ \mathcal{M}'$ that has the same effect as applying first $\mathcal{M}$ and then $\mathcal{M}'$. For our schema evolution context, $\mathcal{M} \circ \mathcal{M}'$ combines the original transformation $\mathcal{M}$ with the evolution mapping $\mathcal{M}'$.

Source schema evolution. Assume now that the source schema evolves to a new source schema $\mathbf{S}''$, and that we model this evolution as a schema mapping $\mathcal{M}''$ from $\mathbf{S}$ to $\mathbf{S}''$. Intuitively, $\mathcal{M}''$ represents a data transformation that converts instances of $\mathbf{S}$ to instances of $\mathbf{S}''$. We need to obtain a new schema mapping that reflects the original schema mapping $\mathcal{M}$ (or rather $\mathcal{M} \circ \mathcal{M}'$ after target schema evolution) but uses $\mathbf{S}''$ as the source schema. Note that in this case we cannot directly combine $\mathcal{M}''$ with $\mathcal{M} \circ \mathcal{M}'$ via composition, since $\mathcal{M}''$ and $\mathcal{M} \circ \mathcal{M}'$ are not consecutive. In order to be able to apply composition, we need first to apply the inversion operator and obtain a schema mapping $\mathcal{M}^\dagger$ that "undoes" the effect of $\mathcal{M}''$. Once we obtain a suitable $\mathcal{M}^\dagger$, we can then apply the composition operator to produce $\mathcal{M}^\dagger \circ \mathcal{M} \circ \mathcal{M}'$. The resulting schema mapping, which is now from $\mathbf{S}''$ to $\mathbf{T}'$, is an adaptation of the original schema mapping $\mathcal{M}$ that works on the evolved schemas.

While the composition of two schema mappings (under a fairly natural semantics (Fagin et al, 2004; Melnik, 2004)) always exists and it is "only" a matter of expressing the composition in a suitable language for schema mappings, the situation is worse for inversion. In general, a schema mapping may lose information and, as a result, it may not be possible to revert the transformation in a way that recovers the original data. Hence, an exact inverse (Fagin, 2007) may not exist, and one needs to look beyond such exact inverses. As a result, there are several notions of "approximations" of inverses that have recently been developed: quasi-inverses (Fagin et al, 2008b), maximum recoveries (Arenas et al, 2008), maximum extended recoveries (Fagin et al, 2009b). In this paper, motivated by the applications to schema evolution, we take a more pragmatic approach to the treatment of the various notions of inverse and emphasize the *operational* aspect behind them. In particular, we focus on two types of inverses, which were first introduced in (Fagin et al, 2009b) and which have a clear operational semantics based on the notion of chase that makes them attractive from a practical point of view. The first type of such operational inverses, which are called *chase-inverses*, can be used to recover the original data (without loss) via the chase. In general, this recovery is up to homomorphic equivalence due to the presence of nulls; in the ideal case when the original source instance is recovered exactly, we call the chase-inverse an *exact* chase-inverse. The second type of operational inverses, which we call *relaxed chase-inverses*[3], are relaxations of chase-inverses that work in situations where there is information loss and, hence, chase-inverses do not exist. Intuitively, a relaxed chase-inverse recovers the original source data as well as possible.

In this chapter, we use various concrete examples of schema evolution to illustrate the main developments and challenges behind composition and inversion and their applications to schema evolution. We note that we are focused here on composition and inversion; a companion book chapter (Hartung et al, 2010) will give a separate overview of the schema evolution area in general. In our survey, we illustrate the concept of composition, and then discuss the two flavors of operational inverses mentioned above. At the same time, we discuss the languages in which such composition and inversion can be expressed. In the context of the schema evolution scenarios that we consider, these languages vary in complexity from GAV schema mappings to LAV and GLAV schema mappings (the latter are also known as source-to-target tuple-generating dependencies, or s-t tgds (Fagin et al, 2005a)) and then to mappings specified by second-order (SO) tgds (Fagin et al, 2005b). During the exposition, we will proceed from simpler, easier scenarios of schema evolution to more challenging scenarios, and illustrate how composition and inversion techniques can be put together into a framework that deals with schema evolution problems.

In a separate section, we examine in detail two systems that implement one or both of the above schema mapping operators in order to deal with aspects of schema evolution. The first one is an implementation of mapping composition (Yu and Popa, 2005) that is part of the Clio system, is based on the second-order tgds introduced in (Fagin et al, 2005b) and is specifically targeted at the problem of mapping adaptation in the context of schema evolution. The second system is the PRISM workbench (Curino et al, 2008) for query migration in the presence of schema evolution. This system is based on query rewriting under constraints and in particular on the chase and backchase framework (Deutsch et al, 1999). However, before it can apply such query rewriting, the PRISM system needs to implement both mapping composition and inversion. The notion chosen here for inversion is based on quasi-inverses (Fagin et al, 2008b).

We end the paper with a discussion of the main open research questions that still remain to be solved. Perhaps the most important open issue here is to find a unifying schema-mapping language that is closed under both composition and the various flavors of inverses, and, additionally, has good algorithmic properties.

[3] These were introduced in (Fagin et al, 2009b) under a different name: *universal-faithful inverses*. However, the term *relaxed chase-inverses*, which we use in this paper, is a more suggestive term that also reflects the relationship with the chase-inverses.

2 Preliminaries

A *schema* $\mathbf{R}$ is a finite sequence $\langle R_1, \ldots, R_k \rangle$ of relation symbols, where each R_i has a fixed arity. An *instance* I over $\mathbf{R}$ is a sequence $(R_1^I, \ldots, R_k^I)$, where each R_i^I is a finite relation of the same arity as R_i. We shall often use R_i to denote both the relation symbol and the relation R_i^I that instantiates it. We assume that we have a countably infinite set $\underline{\mathsf{Const}}$ of *constants* and a countably infinite set $\underline{\mathsf{Var}}$ of *labeled nulls* that is disjoint from $\underline{\mathsf{Const}}$. A *fact* of an instance I (over $\mathbf{R}$) is an expression $R_i^I(v_1, \ldots, v_m)$ (or simply $R_i(v_1, \ldots, v_m)$), where R_i is a relation symbol of $\mathbf{R}$ and $v_1, \ldots, v_m$ are constants or labeled nulls such that $(v_1, \ldots, v_m) \in R_i^I$. The expression $(v_1, \ldots, v_m)$ is also sometimes referred to as a *tuple* of R_i. An instance is often identified with its set of facts.

A *ground* instance over some schema is an instance such that all values occurring in its relations are constants. In general, however, instances over a schema may have individual values from $\underline{\mathsf{Const}} \cup \underline{\mathsf{Var}}$; thus, some of the values in the instances may be nulls representing unknown information. Such (non-ground) instances naturally arise in data integration, data exchange and also schema evolution. We will see examples of instances with nulls all throughout this paper.

Next, we define the concepts of *homomorphism* and *homomorphic equivalence*, which we use frequently throughout this paper. Let I_1 and I_2 be instances over a schema $\mathbf{R}$. A function h from $\underline{\mathsf{Const}} \cup \underline{\mathsf{Var}}$ to $\underline{\mathsf{Const}} \cup \underline{\mathsf{Var}}$ is a *homomorphism* from I_1 to I_2 if for every c in $\underline{\mathsf{Const}}$, we have that $h(c) = c$, and for every relation symbol R in $\mathbf{R}$ and every tuple $(a_1, \ldots, a_n) \in R^{I_1}$, we have that $(h(a_1), \ldots, h(a_n)) \in R^{I_2}$. We use the notation $I_1 \rightarrow I_2$ to denote that there is a homomorphism from I_1 to I_2. We say that I_1 is *homomorphically equivalent* to I_2 if $I_1 \rightarrow I_2$ and $I_2 \rightarrow I_1$, and we write this as $I_1 \leftrightarrow I_2$.

Schema mappings A *schema mapping* is a triple $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$, where $\mathbf{S}$ is a source schema, $\mathbf{T}$ is a target schema, and Σ is a set of constraints (typically, formulas in some logic) that describe the relationship between $\mathbf{S}$ and $\mathbf{T}$. We say that $\mathcal{M}$ is syntactically *specified by*, or, *expressed by* Σ. Furthermore, $\mathcal{M}$ is semantically identified with the binary relation:

$$\mathrm{Inst}(\mathcal{M}) = \{(I, J) \mid I \text{ is an } \mathbf{S}\text{-instance, } J \text{ is a } \mathbf{T}\text{-instance, } (I, J) \models \Sigma\}.$$

We will use the notation $(I, J) \in \mathcal{M}$ to denote that the ordered pair (I, J) satisfies the constraints of $\mathcal{M}$; furthermore, we will sometimes define schema mappings by simply defining the set of ordered pairs (I, J) that constitute $\mathcal{M}$ (instead of giving a set of constraints that specify $\mathcal{M}$). If $(I, J) \in \mathcal{M}$, we say that J is a *solution* of I (with respect to $\mathcal{M}$).

In general, the constraints in Σ are formulas in some logical formalism. In this chapter, we will focus on schema mappings specified by source-to-target tuple-generating dependencies.

An *atom* is an expression of the form $R(x_1, ..., x_n)$, where R is a relation symbol and $x_1, \ldots, x_n$ are variables that are not necessarily distinct. A *source-to-target tuple-generating dependency (s-t tgd)* is a first-order sentence φ of the form

$$\forall \mathbf{x}(\varphi(\mathbf{x}) \rightarrow \exists \mathbf{y}\psi(\mathbf{x}, \mathbf{y})),$$

where $\varphi(\mathbf{x})$ is a conjunction of atoms over $\mathbf{S}$, each variable in $\mathbf{x}$ occurs in at least one atom in $\varphi(\mathbf{x})$, and $\psi(\mathbf{x}, \mathbf{y})$ is a conjunction of atoms over $\mathbf{T}$ with variables in $\mathbf{x}$ and $\mathbf{y}$. For simplicity, we will often suppress writing the universal quantifiers $\forall \mathbf{x}$ in the above formula. Another name for s-t tgds is *global-and-local-as-view* (GLAV) constraints (see (Lenzerini, 2002)). They contain GAV and LAV constraints, which we now define, as important special cases.

A GAV *(global-as-view)* constraint is an s-t tgd in which the right-hand side is a single atom with no existentially quantified variables, that is, it is of the form

$$\forall \mathbf{x}(\varphi(\mathbf{x}) \rightarrow P(\mathbf{x})),$$

where $P(\mathbf{x})$ is an atom over the target schema. A LAV *(local-as-view)* constraint is an s-t tgd in which the left-hand side is a single atom, that is, it is of the form

$$\forall \mathbf{x}(Q(\mathbf{x}) \rightarrow \exists \mathbf{y}\psi(\mathbf{x}, \mathbf{y})),$$

where $Q(\mathbf{x})$ is an atom over the source schema[4].

We often write a *LAV schema mapping* to mean a schema mapping specified entirely by LAV s-t tgds. A strict LAV schema mapping is a LAV schema mapping where it is specified entirely by strict LAV s-t tgds. Similarly, a GAV schema mapping (respectively, GLAV schema mapping) is a schema mapping specified entirely by GAV s-t tgds (respectively, GLAV s-t tgds).

Chase The *chase procedure* has been used in a number of settings over the years. Close to our area of interest, the chase procedure has been used in (Fagin et al, 2005a) to give a natural, operational semantics for data exchange. Specifically, in data exchange, if $\mathcal{M}$ is a fixed schema mapping specified by s-t tgds, then the chase procedure can be used to compute, given a source instance I, a target instance $chase_{\mathcal{M}}(I)$ for I that has a number of desirable properties. First, $chase_{\mathcal{M}}(I)$ is a *universal solution* (Fagin et al, 2005a) of I with respect to the schema mapping $\mathcal{M}$. Universal solutions are the most general solutions that one can obtain for a given source instance I with respect to $\mathcal{M}$ in the sense that a universal solution has homomorphisms into every solution of I with respect to $\mathcal{M}$. Second, $chase_{\mathcal{M}}(I)$ is computed in time bounded by a polynomial in the size of I.

There are several variants of the chase procedure. Here, we will consider the variant of chase described in (Fagin et al, 2005a). The chase on I with a schema mapping $\mathcal{M}$ produces a target instance, denoted as $chase_{\mathcal{M}}(I)$, as follows: For every s-t tgd

$$\forall \mathbf{x}(\varphi(\mathbf{x}) \rightarrow \exists \mathbf{y}\psi(\mathbf{x}, \mathbf{y}))$$

[4] A stricter version of LAV s-t tgds, where no repeated variables in the left-hand side $Q(\mathbf{x})$ are allowed and all variables in $\mathbf{x}$ appear in the right-hand side, is also used in literature. We refer to this type of LAV s-t tgds as *strict* LAV s-t tgds.

in Σ and for every tuple $\mathbf{a}$ of constants from the active domain of I, such that $I \models \varphi(\mathbf{a})$, if there does not exists a tuple $\mathbf{b}$ of constants or labeled nulls, such that $\psi(\mathbf{a}, \mathbf{b})$ exists in $\mathit{chase}_{\mathcal{M}}(I)$, then we add to $\mathit{chase}_{\mathcal{M}}(I)$ all facts in $\psi(\mathbf{a}, \mathbf{N})$ where $\mathbf{N}$ is a tuple of new, distinct labeled nulls interpreting the existential quantified variables $\mathbf{y}$. We sometimes say that $\mathcal{M}$ *has been applied to I to produce $\mathit{chase}_{\mathcal{M}}(I)$* to mean that I has been chased with $\mathcal{M}$ to produce $\mathit{chase}_{\mathcal{M}}(I)$.

We end this section by giving two examples of the chase in action. Variations of the schemas and the mappings used in these examples will appear throughout the paper. First, let $\mathcal{M}_1$ be a LAV schema mapping specified by:

$$\mathtt{Takes}(n, m, co) \rightarrow \exists s(\mathtt{Student}(s, n, m) \wedge \mathtt{Enrolled}(s, co))$$

Here, we assume that the source schema has a ternary relation symbol `Takes` and the target schema has two binary relation symbols, `Student` and `Enrolled`. The mapping takes input tuples of the form (n, m, co) in `Takes`, where n represents a student name, m represents a major for the student and co represents a course that the student takes. For each such input tuple, the mapping asserts the existence of two target tuples: a tuple (s, n, m) in `Student`, and a tuple (s, co) in `Enrolled`. These tuples are related by the fact that the same student id s occurs in both.

Let I be the source instance consisting of the following two facts:

$$\begin{array}{l} \mathtt{Takes}(John, CS, CS101), \\ \mathtt{Takes}(Ann, Math, MATH203). \end{array}$$

The chase of I with $\mathcal{M}_1$ will then produce a target instance J that consists of the following four facts:

$$\begin{array}{l} \mathtt{Student}(N_1, John, CS), \mathtt{Enrolled}(N_1, CS101), \\ \mathtt{Student}(N_2, Ann, Math), \mathtt{Enrolled}(N_2, MATH203). \end{array}$$

In the above instance, N_1 and N_2 are nulls (representing student ids for $John$ and Ann, respectively). The chase of I with $\mathcal{M}_1$ works by exhaustively determining facts in the source instance that can "trigger" the s-t tgd in $\mathcal{M}_1$ to generate new target facts. The first fact in I, namely, $\mathtt{Takes}(John, CS, CS101)$, triggers the s-t tgd in $\mathcal{M}_1$, resulting in the addition of two target facts: $\mathtt{Student}(N_1, John, CS)$ and $\mathtt{Enrolled}(N_1, CS101)$. Observe that this *chase step* instantiates the existentially quantified variable s in the tgd with the null N_1, which effectively associates the newly created `Student` and `Enrolled` facts together. Similarly, the second source fact also triggers the s-t tgd in $\mathcal{M}_1$ to generate two target facts: $\mathtt{Student}(N_2, Ann, Math)$ and $\mathtt{Enrolled}(N_2, MATH203)$. After this, no other source facts could trigger the s-t tgd in $\mathcal{M}_1$ to generate new target facts. Hence, the chase terminates with the target instance that consists of the above four facts.

As another example, let $\mathcal{M}_2$ be a GAV schema mapping specified by:

$$\mathtt{Student}(s, n, m) \wedge \mathtt{Enrolled}(s, co) \rightarrow \mathtt{Takes}'(s, n, co)$$

This schema mapping combines information in `Student` and `Enrolled` into the `Takes′` relation. Observe that `Takes′` contains information about student ids, name, and courses (as opposed to name, major, and course in `Takes`). Suppose I consists of the following facts:

$$\texttt{Student}(111, John, CS), \texttt{Enrolled}(111, CS101),$$
$$\texttt{Student}(111, John, Math), \texttt{Enrolled}(111, MATH101).$$

The chase of I with $\mathcal{M}_2$ will produce the following target instance:

$$\texttt{Takes}'(111, John, CS101),$$
$$\texttt{Takes}'(111, John, MATH101).$$

The source facts $\texttt{Student}(111, John, CS)$ and $\texttt{Enrolled}(111, CS101)$ together trigger the s-t tgd in $\mathcal{M}_2$ to produce $\texttt{Takes}'(111, John, CS101)$. In addition, the source facts $\texttt{Student}(111, John, Math)$ and $\texttt{Enrolled}(111, MATH101)$ trigger the s-t tgd in $\mathcal{M}_2$ to produce $\texttt{Takes}'(111, John, MATH101)$ in the target. After this, even though the source facts $\texttt{Student}(111, John, CS)$ and `Enrolled` $(111, MATH101)$ also trigger the s-t tgd in $\mathcal{M}_2$, this chase step is not taken since the target fact `Takes` $(111,\ John,\ MATH101)$ already exists in the target instance. It is easy to observe that no other source facts would trigger the s-t tgd in $\mathcal{M}_2$, and hence J is the result of the chase. Also note that, as opposed to the previous example, there is no need to generate nulls in the target, since $\mathcal{M}_2$ has no existentially quantified variables (i.e., it is a GAV mapping).

3 An Ideal Scenario of Evolution

We start our exposition of the application of composition and inversion to schema evolution, by considering first a relatively "simple" example of schema evolution. For this section, we will refer to the schema evolution scenario that is graphically illustrated in Figure 2.

We first assume the existence of a schema mapping $\mathcal{M}$ from a source schema **S**, consisting of one relation `Takes`, to a target schema **T**, consisting of two relations `Student` and `Enrolled`. The `Takes` relation contains tuples relating student ids with their majors and the courses they take. According to the mapping $\mathcal{M}$, each tuple of `Takes` is split into two tuples, one in `Student` and the other in `Enrolled`, that share the same `sid` value. Formally, the schema mapping is given by the following two assertions:

$$\begin{aligned} \mathcal{M}:\ & \texttt{Takes}(s, m, co) \rightarrow \texttt{Student}(s, m) \\ & \texttt{Takes}(s, m, co) \rightarrow \texttt{Enrolled}(s, co) \end{aligned}$$

Note that $\mathcal{M}$ is an example of both a GAV mapping and a (strict) LAV mapping. Also note that in this example we have a variation of the earlier $\mathcal{M}_1$ (in Section 2); in this variation, the `sid` value in the target is not existentially quantified, but instead it is copied directly from the source relation `Takes`.

We next address the issues of schema evolution, starting with the target schema first.

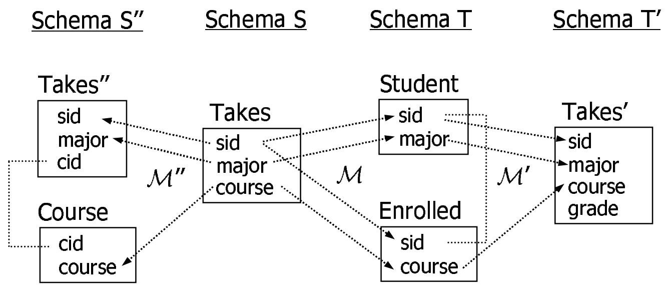

Fig. 2. Our first scenario of schema evolution.

3.1 Target Evolution: GAV-GLAV Composition

Let us assume that the target schema evolves to a new schema $\mathbf{T}'$ consisting of one relation `Takes′` that combines all the attributes in $\mathbf{T}$ (i.e., `sid`, `major`, `course`) and further includes an extra `grade` attribute. Moreover, assume that the evolution mapping from $\mathbf{T}$ to $\mathbf{T}'$ is:

$$\mathcal{M}' : \texttt{Student}(s, m) \wedge \texttt{Enrolled}(s, co) \rightarrow \exists G\ \texttt{Takes}'(s, m, co, G)$$

In contrast to the original mapping $\mathcal{M}$, the above $\mathcal{M}'$ is an example of a more general GLAV mapping: it is neither LAV (since there is more than one atom on the left-hand side) nor GAV (since there is an existential quantifier on the right-hand side).

Before we can show how to adapt the mapping $\mathcal{M}$ to the new target schema, we formally state what composition means.

Definition 1. (Composition of Schema Mappings (Fagin et al, 2005b)) 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 schema mappings such that the schemas $\mathbf{S}_1$, $\mathbf{S}_2$, and $\mathbf{S}_3$ have no relation symbol in common pairwise. A schema mapping $\mathcal{M}_{13} = (\mathbf{S}_1, \mathbf{S}_3, \Sigma_{13})$ is a *composition* of $\mathcal{M}_{12}$ and $\mathcal{M}_{23}$ (written $\mathcal{M}_{13} = \mathcal{M}_{12} \circ \mathcal{M}_{23}$) if $M_{13} = \{(I_1, I_3) \mid$ there exists I_2 such that $(I_1, I_2) \in \mathcal{M}_{12}$ and $(I_2, I_3) \in \mathcal{M}_{23}\}$.

The important computational problem associated to mapping composition is the following: Given two schema mappings $\mathcal{M}_{12}$ and $\mathcal{M}_{23}$ how do we compute, and in what language can we express, a set Σ_{13} of constraints that specifies the composition $\mathcal{M}_{13}$ of $\mathcal{M}_{12}$ and $\mathcal{M}_{23}$? The answer to the above question very much depends on the language in which the input schema mappings are specified.

For our running example, to adapt the above mapping $\mathcal{M}$ to the new target schema, we must compose $\mathcal{M}$ with the evolution mapping $\mathcal{M}'$. As it turns out, we are in an "easy" case where we can express the result of this composition as a GLAV mapping. This is essentially due to the fact that the first mapping is GAV. (The second mapping $\mathcal{M}'$ is a GLAV mapping.) We shall see that in cases where $\mathcal{M}$ is LAV or GLAV the composition need not be first-order and we need a more powerful language to express the composition. For the scenario in this section, the fact that the composition is a GLAV mapping follows from the next theorem.

Theorem 1 ((Fagin et al, 2005b)). *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two consecutive schema mappings. The following hold:*

1. *If $\mathcal{M}_1$ and $\mathcal{M}_2$ are GAV mappings then $\mathcal{M}_1 \circ \mathcal{M}_2$ can be expressed as a GAV mapping.*
2. *If $\mathcal{M}_1$ is a GAV mapping and $\mathcal{M}_2$ is a GLAV mapping then $\mathcal{M}_1 \circ \mathcal{M}_2$ can be expressed as a GLAV mapping.*

As a more general result, we obtain the following corollary that applies to a chain of GAV mappings followed by a GLAV mapping.

Corollary 1. *Let $\mathcal{M}_1, \ldots, \mathcal{M}_{k+1}, \mathcal{M}_k$ be consecutive schema mappings. If $\mathcal{M}_1$, $\ldots$, $\mathcal{M}_k$ are GAV mappings and M_{k+1} is a GLAV mapping then the composition $\mathcal{M}_1 \circ \ldots \circ M_k \circ \mathcal{M}_{k+1}$ can be expressed as a GLAV mapping.*

Concretely, for our scenario, it can be verified that the following GLAV mapping is the composition of $\mathcal{M}$ and $\mathcal{M}'$:

$$\mathcal{M} \circ \mathcal{M}' : \texttt{Takes}(s, m, co) \wedge \texttt{Takes}(s, m', co') \rightarrow \exists G\ \texttt{Takes}'(s, m, co', G)$$

Observe that the self-join on `Takes` in the above composition is needed. This can be traced to the fact that students can have multiple majors, in general. At the same time, the `Takes` relation need not list all combinations of `major` and `course` for a given `sid`. However, the evolution mapping $\mathcal{M}'$ requires all such combinations. The composition $\mathcal{M} \circ \mathcal{M}'$ correctly accounts for all these subtle semantic aspects.

To see a concrete example, consider the following instance of `Takes`:

$$\texttt{Takes}(007, Math, MA201)$$
$$\texttt{Takes}(007, CS, CS101)$$

In the above instance, 007 identifies a student (say, *Ann*) who has a double major (in $Math$ and CS) and takes two courses. Given the above instance, the composition $\mathcal{M} \circ \mathcal{M}'$ requires the existence of the following four `Takes′` facts, to account for all the combinations between *Ann*'s majors and the courses that *Ann* took.

$$\texttt{Takes}'(007, Math, MA201, G_1)$$
$$\texttt{Takes}'(007, Math, CS101, G_2)$$
$$\texttt{Takes}'(007, CS, MA201, G_3)$$
$$\texttt{Takes}'(007, CS, CS101, G_4)$$

In practice, we would also have an additional target constraint (a functional dependency) on `Takes′` specifying that `sid` together with `course` functionally determines `grade`. This functional dependency would then force the equality of G_1 and G_3, and also the equality of G_2 and G_4 in the above instance.

Composition Algorithm Next, we explain on our example how the composition algorithm of (Fagin et al, 2005b) arrives at the formula that specifies $\mathcal{M} \circ \mathcal{M}'$. We give an intuitive explanation of the algorithm rather than a complete and formal one. Recall that $\mathcal{M}$ is specified by the following GAV s-t tgds

$$\begin{aligned} \mathcal{M}: \ & \texttt{Takes}(s, m, co) \rightarrow \texttt{Student}(s, m) \\ & \texttt{Takes}(s, m, co) \rightarrow \texttt{Enrolled}(s, co) \end{aligned}$$

and that $\mathcal{M}'$ is specified by the following GLAV s-t tgd

$$\mathcal{M}' : \texttt{Student}(s, m) \wedge \texttt{Enrolled}(s, co) \rightarrow \exists G \ \texttt{Takes}'(s, m, co, G).$$

Intuitively, the composition algorithm will replace each relation symbol from $\mathbf{T}$ in $\mathcal{M}'$ by relation symbols from $\mathbf{S}$ using the GAV s-t tgds of $\mathcal{M}$. In this case, the fact $\texttt{Student}(s, m)$ that occurs on the left-hand side of $\mathcal{M}'$ can be replaced by a `Takes` fact, according to the first GAV s-t tgd of $\mathcal{M}$. Hence, we arrive at an intermediate tgd shown below:

$$\texttt{Takes}(s, m, co') \wedge \texttt{Enrolled}(s, co) \rightarrow \exists G \ \texttt{Takes}'(s, m, co, G)$$

Observe that a new variable co' in `Takes` is used instead of co. This avoids an otherwise unintended join with `Enrolled`, which also contains the variable co. (This is accomplished in the algorithm by a variable renaming step.)

Next, the composition algorithm will replace $\texttt{Enrolled}(s, co)$ with a `Takes` fact, based on the second GAV s-t tgd of $\mathcal{M}$. We then obtain the following GLAV s-t tgd from the source schema $\mathbf{S}$ to the new target schema $\mathbf{T}'$. This tgd[5] specifies the composition $\mathcal{M} \circ \mathcal{M}'$.

$$\texttt{Takes}(s, m, co') \wedge \texttt{Takes}(s, m', co) \rightarrow \exists G \ \texttt{Takes}'(s, m, co, G)$$

3.2 Source Evolution: The Case of a Lossless Mapping

Let us now assume that the source schema evolves to a new schema $\mathbf{S}''$ consisting of the two relations $\texttt{Takes}''$ and `Course` shown in Figure 2. Thus, in the new schema, courses are stored in a separate relation and are assigned ids (`cid`). The relation $\texttt{Takes}''$ is similar to `Takes` except that `course` is replaced by `cid`. Let us assume that the source evolution is described by the following LAV mapping:

$$\mathcal{M}'' : \texttt{Takes}(s, m, co) \rightarrow \exists C \ (\texttt{Takes}''(s, m, C) \wedge \texttt{Course}(C, co))$$

Note first that in the figure the direction of $\mathcal{M}''$ is the reverse of the direction of the original mapping $\mathcal{M}$. Intuitively, the assertions of $\mathcal{M}''$ imply a data flow

[5] Note that it is logically equivalent to the earlier way we expressed $\mathcal{M} \circ \mathcal{M}'$, and where the roles of co and co' were switched.

from the schema $\mathbf{S}$ to the schema $\mathbf{S}''$, where facts over $\mathbf{S}''$ are required to exist based on facts over $\mathbf{S}$. In order to enable the application of the same composition techniques as we used for target evolution, we first need to invert the mapping $\mathcal{M}''$. After inversion, we can then combine the result, via composition, with the previously obtained $\mathcal{M} \circ \mathcal{M}'$.

From a practical point of view, the important (and ideal) requirement that we need from an inverse is to be able to recover the original source instance. Concretely, if we apply the mapping $\mathcal{M}''$ on some source instance I and then we apply the candidate inverse on the result of $\mathcal{M}''$, we would like to obtain the original source instance I. Here, applying a schema mapping $\mathcal{M}$ to an instance I means generating the instance $\mathit{chase}_{\mathcal{M}}(I)$. The next definition captures the requirements of such an inverse.

Definition 2 (Exact chase-inverse). Let $\mathcal{M}$ be a GLAV schema mapping from a schema $\mathbf{S_1}$ to a schema $\mathbf{S_2}$. We say that $\mathcal{M}^*$ is an *exact chase-inverse* of $\mathcal{M}$ if $\mathcal{M}^*$ is a GLAV schema mapping from $\mathbf{S_2}$ to $\mathbf{S_1}$ with the following property: for every instance I over S_1, we have that $I = \mathit{chase}_{\mathcal{M}^*}(\mathit{chase}_{\mathcal{M}}(I))$.

For our example, consider the following candidate inverse of $\mathcal{M}''$:

$$\mathcal{M}^\dagger : \texttt{Takes}''(s,m,c) \wedge \texttt{Course}(c,co) \rightarrow \texttt{Takes}(s,m,co)$$

As it turns out, this candidate inverse satisfies the above requirement of being able to recover, exactly, the source instance. Indeed, it can be immediately verified that for every source instance I over $\mathbf{S}$, we have that $\mathit{chase}_{\mathcal{M}^\dagger}(\mathit{chase}_{\mathcal{M}''}(I))$ equals I. Thus, $\mathcal{M}^\dagger$ is an exact chase-inverse of $\mathcal{M}''$.

Since $\mathcal{M}^\dagger$ is a GAV mapping, we can now apply Corollary 1 and compose $\mathcal{M}^\dagger$ with $\mathcal{M} \circ \mathcal{M}'$ to obtain a schema mapping from $\mathbf{S}''$ to $\mathbf{T}'$. The result of this composition is the following (GLAV) schema mapping:

$$\begin{aligned}\mathcal{M}^\dagger \circ \mathcal{M} \circ \mathcal{M}' : \quad & \texttt{Takes}''(s,m,c) \wedge \texttt{Course}(c,co) \wedge \\ & \texttt{Takes}''(s,m',c') \wedge \texttt{Course}(c',co') \\ & \quad \rightarrow \exists G\ \texttt{Takes}'(s,m',co,G)\end{aligned}$$

3.3 A More General Notion of Chase-Inverses

The schema mapping $\mathcal{M}^\dagger$ used in Section 3.2 is an exact chase-inverse in the sense that it can recover the original source instance I exactly. In general, however, equality with I is too strong of a requirement, and all we need is a more relaxed form of equivalence of instances, where intuitively the equivalence is modulo nulls. In this section, we start with a concrete example to show the need for such relaxation. We then give the general definition of a chase-inverse (Fagin et al, 2009b) and discuss its properties and its application in the context of schema evolution.

We observe that the schema mapping $\mathcal{M}''$ in Section 3.2 is similar to the following general pattern:

$$P(x,y) \rightarrow \exists z\ (Q(x,z) \wedge Q'(z,y))$$

Here, for simplicity, we focus on schema mappings on binary relations. (In particular, $\mathcal{M}''$ can be forced into this pattern if we ignore the major field in the two relations `Takes` and `Takes`$''$.) The important point about this type of mappings is that they always have an exact chase-inverse. Consider now a variation on the above pattern, where Q' is the same as Q. Thus, let $\mathcal{M}$ be the following schema mapping:

$$\mathcal{M}: \quad P(x,y) \rightarrow \exists z\ (Q(x,z) \wedge Q(z,y)).$$

The following schema mapping $\mathcal{M}^*$ is a natural candidate inverse of $\mathcal{M}$:

$$\mathcal{M}^*: \quad Q(x,z) \wedge Q(z,y) \rightarrow P(x,y).$$

Consider now the source instance $I = \{P(1,2), P(2,3)\}$. Then the result of applying $\mathcal{M}$ to I is

$$\mathit{chase}_{\mathcal{M}}(I) = \{Q(1,n_1), Q(n_1,2), Q(2,n_2), Q(n_2,3)\},$$

where n_1 and n_2 are two nulls introduced by the chase (for the existentially quantified variable z). Furthermore, the result of applying $\mathcal{M}^*$ to the previous instance is

$$\mathit{chase}_{\mathcal{M}^*}(\mathit{chase}_{\mathcal{M}}(I)) = \{P(1,2), P(2,3), P(n_1,n_2)\}.$$

Thus, we recovered the two original facts of I but also the additional fact $P(n_1,n_2)$ (via joining $Q(n_1,2)$ and $Q(2,n_2)$). Therefore, $\mathcal{M}^*$ is not an exact chase-inverse of $\mathcal{M}$. Nevertheless, since n_1 and n_2 are nulls, the extra fact $P(n_1,n_2)$ does not add any new information that is not subsumed by the other two facts. Intuitively, the last instance is equivalent (although not equal) to the original source instance I.

The above type of equivalence between instances with nulls is captured, in general, by the notion of *homomorphic equivalence*. Recall that two instances I_1 and I_2 are homomorphically equivalent, with notation $I_1 \leftrightarrow I_2$, if there exist homomorphisms in both directions between I_1 and I_2.

We are now ready for the main definition in this section.

Definition 3 (Chase-inverse). Let $\mathcal{M}$ be a GLAV schema mapping from a schema $\mathbf{S_1}$ to a schema $\mathbf{S_2}$. We say that $\mathcal{M}^*$ is a *chase-inverse* of $\mathcal{M}$ if $\mathcal{M}^*$ is a GLAV schema mapping from $\mathbf{S_2}$ to $\mathbf{S_1}$ with the following property: for every instance I over S_1, we have that $I \leftrightarrow \mathit{chase}_{\mathcal{M}^*}(\mathit{chase}_{\mathcal{M}}(I))$.

Intuitively, the above definition uses homomorphic equivalence as a replacement for the usual equality between instances. This is consistent with the fact that, in the presence of nulls, the notion of homomorphism itself becomes a replacement for the usual containment between instances. Note that when I_1 and I_2 are ground, $I_1 \rightarrow I_2$ is the same as $I_1 \subseteq I_2$. However, when I_1 has nulls, these nulls are allowed to be homomorphically mapped to other values (constants or nulls) inside I_2. This reflects the fact that nulls represent unknown information.

The existence of a chase-inverse for $\mathcal{M}$ implies that $\mathcal{M}$ has no information loss, since we can recover an instance that is the same modulo homomorphic equivalence as the original source instance. At the same time, a chase-inverse is a relaxation of the notion of an exact chase-inverse; hence, it may exist even when an exact chase-inverse does not exist.

Both examples of chase-inverses that we have given, namely $\mathcal{M}^\dagger$ in Section 3.2 and $\mathcal{M}^*$ in this section, are GAV mappings. This is not by accident. As the following theorem shows, we do not need the full power of GLAV mappings to express a chase-inverse: whenever there is a chase-inverse, there is a GAV chase-inverse. The main benefit of this theorem is that it may keep composition simpler. In particular, we may still be able to apply Corollary 1 as opposed to the more complex composition techniques of Section 4.

Theorem 2 ((Fagin et al, 2010)). *Let $\mathcal{M}$ be a GLAV schema mapping. If $\mathcal{M}$ has a chase-inverse, then $\mathcal{M}$ has a GAV chase-inverse.*

We remark that other, more general notions of inverses exist that are not based on the chase. The first notion of an "exact" inverse, capturing the case of no loss of information, was introduced by Fagin (Fagin, 2007). An exact inverse $\mathcal{M}^*$ of $\mathcal{M}$ is a schema mapping $\mathcal{M}^*$ satisfying the equation $\mathcal{M} \circ \mathcal{M}^* = \text{Id}$ where Id is the "identity" GLAV schema mapping that maps each relation in a schema to a copy of it. Subsequently, extended inverses (Fagin et al, 2009b) were introduced as an extension of exact inverses that is able to handle instances with nulls (i.e., non-ground instances). Without giving the exact definition of extended inverses here, we point out that chase-inverses coincide with the extended inverses that are specified by GLAV constraints. Thus, from a practical point of view, chase-inverses are important special cases of extended inverses, with good algorithmic properties.

We conclude this section with a corollary that summarizes the applications of chase-inverses together with the earlier Corollary 1 to our schema evolution context.

Corollary 2. *Let $\mathcal{M}$, $\mathcal{M}'$ and $\mathcal{M}''$ be schema mappings as in Figure 1 such that $\mathcal{M}$ is a GAV mapping and $\mathcal{M}'$ and $\mathcal{M}''$ are GLAV mappings. Assume that $\mathcal{M}''$ has a chase-inverse, and let $\mathcal{M}^\dagger$ be a GAV chase-inverse of $\mathcal{M}''$. Then the mapping $\mathcal{M}^\dagger \circ \mathcal{M} \circ \mathcal{M}'$ can be expressed as a GLAV mapping.*

We note that a chase-inverse may not exist in general, since a schema mapping may lose information and hence it may not be possible to find a chase-inverse. The above corollary depends on the fact that the schema mapping $\mathcal{M}''$ has a chase-inverse. In Section 5 we shall address the more general case where $\mathcal{M}''$ has no chase-inverse.

The other important restriction in the above corollary is that the original schema mapping $\mathcal{M}$ must be GAV and not GLAV. We shall lift this restriction in the next section.

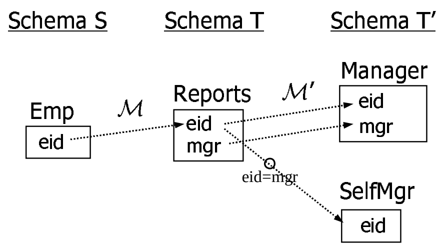

Fig. 3. A target evolution scenario that needs SO tgds.

4 Composition: The Need For Second-order TGDs

In this section, we discuss a more general schema mapping language as well as a more general composition result that enables us, in particular, to handle the general case of composing GLAV mappings. In particular, in our schema evolution context, we show how to handle the case where $\mathcal{M}$ is a GLAV mapping instead of a GAV mapping. We start by showing first that the composition $\mathcal{M} \circ \mathcal{M}'$ becomes challenging in such a case. We then illustrate the necessity of second-order tgds (SO tgds) (Fagin et al, 2005b) as a more powerful language needed to express such a composition.

For this section, we shall consider a very simple scenario (Fagin et al, 2005b) that is graphically illustrated in Figure 3. In this scenario, the source schema **S** consists of one relation `Emp` with a single attribute for employee id (`eid`). The target schema **T** consists of one relation `Reports` that associates each employee with his/her manager. In the target relation, `mgr` is itself an employee id (the employee id of the manager). Assume that we have the following schema mapping that describes the relationship between a database over **S** and a database over **T**:

$$\mathcal{M} : \texttt{Emp}(e) \rightarrow \exists M\ \texttt{Reports}(s, M)$$

Note that the above mapping is a very simple example of a LAV mapping that is not a GAV mapping.

Let us assume that the target schema evolves to a new schema $\mathbf{T}'$ consisting of the two relations `Manager` and `SelfMgr` shown in Figure 3. Moreover, assume that the evolution mapping from **T** to $\mathbf{T}'$ is given by:

$$\begin{aligned} \mathcal{M}' : \texttt{Reports}(e, m) &\rightarrow \texttt{Manager}(e, m) \\ \texttt{Reports}(e, e) &\rightarrow \texttt{SelfMgr}(e) \end{aligned}$$

Thus, in the new schema, the relation `Manager` of $\mathbf{T}'$ is intended to be a copy of the relation `Reports` of **T**, while the relation `SelfMgr` is intended to contain all employees that are their own managers, that is, employees for which the `eid` field equals the `mgr` field in the relation `Reports` of **T**. Note that the evolution mapping $\mathcal{M}'$ is a GAV mapping.

In order to express the composition $\mathcal{M} \circ \mathcal{M}'$ for this example, it turns out that we cannot use GLAV constraints. It is shown in (Fagin et al, 2005b) that there is no (finite or infinite) set of GLAV constraints that specifies $\mathcal{M} \circ \mathcal{M}'$. However, the following *second-order tgd (SO tgd)* specifies the composition $\mathcal{M} \circ \mathcal{M}'$:

$$\begin{aligned} \exists f(\ & \forall e(\texttt{Emp}(e) \rightarrow \texttt{Manager}(e, f(e))) \\ & \wedge \forall e(\texttt{Emp}(e) \wedge (e = f(e)) \rightarrow \texttt{SelfMgr}(e))) \end{aligned}$$

We will formally define SO tgds shortly. For now, we note that SO tgds strictly include GLAV constraints and make essential use of function symbols. In particular, the above SO tgd uses a function symbol f and an equality $e = f(e)$. The use of both equalities and function symbols is, in general, necessary. As it can be seen, the above SO tgd consists of two inner implications, $\forall e(\texttt{Emp}(e) \rightarrow \texttt{Manager}(e, f(e)))$ and $\forall e(\texttt{Emp}(e) \wedge (e = f(e)) \rightarrow \texttt{SelfMgr}(e))$, which share a universally-quantified unary function symbol f. Intuitively, the first part of the SO tgd states that every employee in `Emp` has a manager who is given by the value $f(e)$. The second part of the SO tgd states that if an employee e in `Emp` has a manager equal to itself (i.e., $e = f(e)$), then this employee must appear in the `SelfMgr` relation in the target.

Next, we provide the precise definition of an SO tgd and give an informal description of the composition algorithm of (Fagin et al, 2005b) that derives SO tgds such as the above one. The definition of an SO tgd makes use of the concept of a *term*, which we define first.

Given a collection $\mathbf{x}$ of variables and a collection $\mathbf{f}$ of function symbols, a *term (based on* $\mathbf{x}$ *and* $\mathbf{f}$*)* is defined inductively as follows:

1. Every variable in $\mathbf{x}$ is a term.
2. If f is a k-ary function symbol in $\mathbf{f}$ and $t_1, ..., t_k$ are terms, then $f(t_1, ..., t_k)$ is a term.

Definition 4. (Second-Order Tuple Generating Dependencies (Fagin et al, 2005b)) Let $\mathbf{S}$ be a source schema and $\mathbf{T}$ a target schema. A *second-order tuple-generating dependency (SO tgd)* is a formula of the form:

$$\exists \mathbf{f}((\forall \mathbf{x}_1(\phi_1 \rightarrow \psi_1) \wedge \ldots \wedge \forall \mathbf{x}_n(\phi_n \rightarrow \psi_n)))$$

where

1. Each member of $\mathbf{f}$ is a function symbol.
2. Each ϕ_i is a conjunction of
   – atomic formulas of the form $S(y_1, ..., y_k)$, where S is a k-ary relation symbol of schema $\mathbf{S}$ and $y_1, ..., y_k$ are variables in $\mathbf{x}_i$, not necessarily distinct, and
   – equalities of the form $t = t'$ where t and t' are terms based on $\mathbf{x}_i$ and $\mathbf{f}$.
3. Each ψ_i is a conjunction of atomic formulas $T(t_1, ..., t_l)$, where T is an l-ary relation symbol of schema $\mathbf{T}$ and $t_1, ..., t_l$ are terms based on $\mathbf{x}_i$ and $\mathbf{f}$.
4. Each variable in $\mathbf{x}_i$ appears in some atomic formula of ϕ_i.

Composition Algorithm for SO tgds We now illustrate the steps of the composition algorithm using the schema mappings $\mathcal{M}$ and $\mathcal{M}'$ in this section. For the complete details of the algorithm, we refer the reader to (Fagin et al, 2005b). The first step of the algorithm is to transform $\mathcal{M}$ and $\mathcal{M}'$ into schema mappings that are specified by SO tgds (if they are not already given as SO tgds). Each GLAV constraint can be transformed into an SO tgd by skolemization, that is, by replacing each existentially quantified variable by a Skolem term. For our example, we transform $\mathcal{M}$ into a schema mapping specified by the following SO tgd:

$$\exists f (\forall e (\texttt{Emp}(e) \to \texttt{Reports}(e, f(e)))).$$

Here, f is an existentially quantified function and the term $f(e)$ replaces the earlier existentially quantified variable M. The second mapping $\mathcal{M}'$ needs no skolemization since there are no existentially quantified variables. The corresponding SO tgd for $\mathcal{M}'$ is simply one with no existentially quantified functions and consisting of the conjunction of the two constraints that specify $\mathcal{M}'$.

After this, we initialize two sets, $\mathcal{S}$ and $\mathcal{S}'$, to consist of all the implications of the SO tgds in $\mathcal{M}$ and, respectively, $\mathcal{M}'$.

$$\begin{aligned}&\mathcal{S} : \texttt{Emp}(e_0) \to \texttt{Reports}(e_0, f(e_0))\\&\mathcal{S}' : \texttt{Reports}(e, m) \to \texttt{Manager}(e, m),\ \texttt{Reports}(e, e) \to \texttt{SelfMgr}(e)\end{aligned}$$

Observe that the existential quantifiers of function symbols as well as the universal quantifiers in front of the implications are omitted, for convenience. Additionally, we have renamed the variables in $\mathcal{S}$ so that they are disjoint from the variables used in $\mathcal{S}'$.

Next, for each implication in $\mathcal{S}'$, we consider each relational atom on the left-hand side of the implication and replace that atom based on all the implications in $\mathcal{S}$ whose right-hand side have an atom with the same relation symbol. For our example, we will replace $\texttt{Reports}(e,m)$ of the first implication in $\mathcal{S}'$ using the sole implication in $\mathcal{S}$, whose right-hand side also has a `Reports` atom. Replacement proceeds by equating the terms in corresponding positions of $\texttt{Reports}(e_0, f(e_0))$ and $\texttt{Reports}(e,m)$, and then adding the left-hand side of the implication in $\mathcal{S}$. In this case, we obtain the equalities $e_0 = e$ and $f(e_0) = m$ and we add the relational atom $\texttt{Emp}(e_0)$. Hence, the first implication of $\mathcal{S}'$ becomes:

$$\chi_1 : \texttt{Emp}(e_0) \wedge (e_0 = e) \wedge (f(e_0) = m) \to \texttt{Manager}(e, m).$$

Similarly, the second implication of $\mathcal{S}_{23}$ becomes:

$$\chi_2 : \texttt{Emp}(e_0) \wedge (e_0 = e) \wedge (f(e_0) = e) \to \texttt{SelfMgr}(e).$$

The implications χ_1 and χ_2 can be simplified by replacing every occurrence of e_0 with e (according to the equality $e_0 = e$). In addition, χ_1 can be further simplified by replacing m with $f(e)$. We obtain:

$$\begin{aligned}&\chi_1 : \texttt{Emp}(e) \to \texttt{Manager}(e, f(e))\\&\chi_2 : \texttt{Emp}(e) \wedge (f(e) = e) \to \texttt{SelfMgr}(e)\end{aligned}$$

At this point, the resulting implications describe a relationship between relation symbols of $\mathbf{S}$ and relation symbols of $\mathbf{T}'$. The final SO tgd that describes the composition $\mathcal{M} \circ \mathcal{M}'$ is obtained by adding all the needed universal quantifiers in front of each implication and then by adding in all the existentially quantified functions (at the beginning of the formula). For our example, we obtain:

$$\exists f (\forall e\, \chi_1 \wedge \forall e\, \chi_2).$$

The following theorem states that SO tgds suffice for composition of GLAV mappings. Moreover, SO tgds are closed under composition. Thus, we do not need to go beyond SO tgds for purposes of composition.

Theorem 3 ((Fagin et al, 2005b)). *Let $\mathcal{M}$ and $\mathcal{M}'$ be two consecutive schema mappings.*

1. *If $\mathcal{M}$ and $\mathcal{M}'$ are GLAV, then $\mathcal{M} \circ \mathcal{M}'$ can be expressed by an SO tgd.*
2. *If $\mathcal{M}$ and $\mathcal{M}'$ are SO tgds, then $\mathcal{M} \circ \mathcal{M}'$ can be expressed by an SO tgd.*

Moreover, it is shown in (Fagin et al, 2005b) that SO tgds form a minimal language for the composition of GLAV mappings, in the sense that every schema mapping specified by an SO tgd is the composition of a finite number of GLAV schema mappings.

The above theorem has an immediate consequence in the context of target schema evolution. As long as the original schema mapping $\mathcal{M}$ is GLAV or given by an SO tgd, and as long as we represent the target evolution $\mathcal{M}'$ by a similar type of mapping, then the new adapted mapping can be obtained by composition and can be expressed as an SO tgd.

Additionally, the above theorem also applies in the context of source schema evolution, provided that the source evolution mapping $\mathcal{M}''$ has a chase-inverse. We summarize the applicability of Theorem 3 to the context of schema evolution as follows.

Corollary 3. *Let $\mathcal{M}$, $\mathcal{M}'$ and $\mathcal{M}''$ be schema mappings as in Figure 1 such that $\mathcal{M}$ and $\mathcal{M}'$ are SO tgds (or, in particular GLAV mappings) and $\mathcal{M}''$ is a GLAV mapping. If $\mathcal{M}''$ has a chase-inverse $\mathcal{M}^\dagger$, then the mapping $\mathcal{M}^\dagger \circ \mathcal{M} \circ \mathcal{M}'$ can be expressed as an SO tgd.*

The important remaining restriction in the above corollary is that the source evolution mapping $\mathcal{M}''$ must have a chase-inverse and, in particular, that $\mathcal{M}''$ is a lossless mapping. We address next the case where $\mathcal{M}''$ is lossy and, hence, a chase-inverse does not exist.

5 The Case of Lossy Mappings

We have seen earlier that *chase-inverses*, when they exist, can be used to recover the original source data either *exactly*, in the case of exact chase-inverses, or *modulo homomorphic equivalence*, in general. However, chase-inverses do not

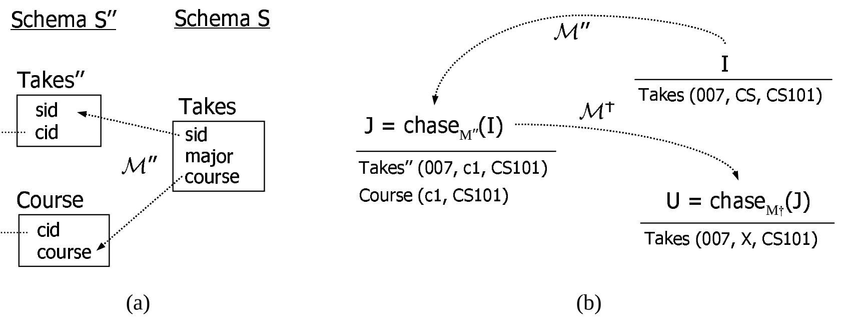

Fig. 4. (a) A case where $\mathcal{M}''$ is a lossy mapping. (b) Recovery of an instance U such that $U \leftrightarrow_{\mathcal{M}''} I$.

always exist. Intuitively, a schema mapping may drop some of the source information, by either projecting or filtering the data, and hence it is not possible to recover the same amount of information. In this section, we look at relaxations of chase-inverses, which we call *relaxed chase-inverses* (Fagin et al, 2009b), and which are intended for situations where there is information loss. Intuitively, a relaxed chase-inverse recovers the original source data as well as possible.

5.1 Relaxed Chase-Inverses

We consider a variation of the scenario described in Figure 2. In this variation, the evolved source schema $\mathbf{S}''$ is changed so that it no longer contains the `major` field. The new source evolution scenario is illustrated graphically in Figure 4(a). The source evolution mapping $\mathcal{M}''$ is now given as:

$$\mathcal{M}'' : \texttt{Takes}(s, m, co) \rightarrow \exists C\ (\texttt{Takes}''(s, C) \wedge \texttt{Course}(C, co))$$

The natural "inverse" that one would expect here is the following mapping:

$$\mathcal{M}^\dagger : \texttt{Takes}''(s, c) \wedge \texttt{Course}(c, co) \rightarrow \exists M\ \texttt{Takes}(s, M, co)$$

First of all, it can be verified that $\mathcal{M}^\dagger$ is not a chase-inverse for $\mathcal{M}''$. In particular, if we start with a source instance I for `Takes` where the source tuples contain some constant values for the `major` field, and then apply the chase with $\mathcal{M}''$ and then the reverse chase with $\mathcal{M}^\dagger$, we obtain another source instance U for `Takes` where the tuples have nulls in the `major` position. Consequently, the resulting source instance U cannot be homomorphically equivalent to the original source instance I. To give a concrete example, consider the source instance I over the schema $\mathbf{S}$ that is shown in Figure 4(b). If we apply the chase with $\mathcal{M}''$ on I we obtain the instance J shown in the same figure. Here, c_1 is a null that is assigned as the course id for $CS101$. If we now apply $\mathcal{M}^\dagger$ to J we obtain another source instance U, where a null X is used in place of a major.

As it can be seen, the recovered source instance U is not homomorphically equivalent to the original source instance: there is a homomorphism from U to

I, but no homomorphism can map the constant CS in I to the null X in U. Intuitively, there is information loss in the evolution mapping $\mathcal{M}''$, which does not export the `major` field. Later on, in Section 5.2, we will show that in fact $\mathcal{M}''$ has no chase-inverse; thus, we cannot recover a homomorphically equivalent source instance.

At the same time, it can be argued, intuitively, that the source instance U that is recovered by $\mathcal{M}^\dagger$ in this example is the "best" source instance that can be recovered, given the circumstances. We will make this notion precise in the next paragraphs, leading to the definition of a relaxed chase-inverse. In particular, we will show that $\mathcal{M}^\dagger$ is a relaxed chase-inverse.

Data Exchange Equivalence. First, we observe that the source instance U that is recovered by $\mathcal{M}^\dagger$ contains all the information that has been present in the original source instance I *and* has been exported by $\mathcal{M}''$. Indeed, if we now apply the mapping $\mathcal{M}''$ on U, we obtain via the chase an instance that is the same as J modulo null renaming (i.e., the chase may generate a different null c_2 instead of c_1). Thus, the following holds:

$$\mathit{chase}_{\mathcal{M}''}(U) \leftrightarrow \mathit{chase}_{\mathcal{M}''}(I),$$

where recall that $\leftrightarrow$ denotes homomorphic equivalence of instances. Intuitively, the above equivalence says that U is as good as I from the point of view of the data they export via $\mathcal{M}''$. Thus, intuitively, U and I are also equivalent, although in a weaker sense. This weaker notion of equivalence is captured by the following definition, which was first given in (Fagin et al, 2008b).

Definition 5. Let $\mathcal{M}$ be a GLAV schema mapping from $\mathbf{S_1}$ to $\mathbf{S_2}$. Let I and I' be two instances over $\mathbf{S_1}$. We say that I and I' are *data exchange equivalent with respect to* $\mathcal{M}$ if $\mathit{chase}_{\mathcal{M}}(I) \leftrightarrow \mathit{chase}_{\mathcal{M}}(I')$. We also write in such case that $I \leftrightarrow_{\mathcal{M}} I'$.

For our example, we have that $U \leftrightarrow_{\mathcal{M}''} I$. At this point, we could take such a condition (i.e., the recovery of an instance U that is data exchange equivalent to I) to be the requirement for a relaxation of a chase-inverse. Such relaxation would be consistent with the earlier notion of chase-inverse and lead into a natural hierarchy of inverses. More precisely, if $\mathcal{M}$ is a GLAV schema mapping, then we could have three types of chase-based inverses $\mathcal{M}^*$, that increasingly relax the equivalence requirement between I and $\mathit{chase}_{\mathcal{M}^*}(\mathit{chase}_{\mathcal{M}}(I))$:

1. $I = \mathit{chase}_{\mathcal{M}^*}(\mathit{chase}_{\mathcal{M}}(I))$ (exact chase-inverse)
2. $I \leftrightarrow \mathit{chase}_{\mathcal{M}^*}(\mathit{chase}_{\mathcal{M}}(I))$ (chase-inverse)
3. $I \leftrightarrow_{\mathcal{M}} \mathit{chase}_{\mathcal{M}^*}(\mathit{chase}_{\mathcal{M}}(I))$

Somewhat surprisingly, having just the third condition is too loose of a requirement for a good notion of a relaxation of a chase-inverse. As we show next, we need to add an additional requirement of homomorphic containment.

Relaxed Chase-Inverse: Stronger Requirement. We illustrate the need for the extra condition by using our example. Refer again to the schema mapping

$\mathcal{M}''$ in Figure 4(a) and the natural candidate inverse $\mathcal{M}^\dagger$ introduced earlier. As shown in Figure 4(b), given the source instance I, the mapping $\mathcal{M}^\dagger$ recovers an instance U such that U and I are data exchange equivalent with respect to $\mathcal{M}''$. However, there can be many other instances that are data exchange equivalent to I but intuitively are incorrect. Consider, for example, the following instance:

$$U' = \{\texttt{Takes}(007, 007, CS101)\}$$

Like U, the instance U' is data exchange equivalent to I with respect to $\mathcal{M}''$. (The only difference from U is in the *major* field, which is not used by the chase with $\mathcal{M}''$.) Furthermore, such instance U' would be obtained if we use the following "inverse" instead of $\mathcal{M}^\dagger$:

$$\mathcal{M}_1^\dagger : \texttt{Takes}''(s, c) \wedge \texttt{Course}(c, co) \rightarrow \texttt{Takes}(s, s, co)$$

Intuitively, the instance U' and the mapping $\mathcal{M}_1^\dagger$ are not what we would expect from a natural inverse. In the instance U', the `sid` value 007 is artificially copied into the `major` field, and the resulting `Takes` fact represents *extra* information that did not appear in the original source instance I. We can rule out bad "inverses" such as $\mathcal{M}_1^\dagger$ by requiring any recovered instance to also have a homomorphism into I. Intuitively, this is a soundness condition saying that the recovered instance does not have extra facts that were not present in I. Note that the earlier instance U does have a homomorphism into I.

Putting it all together, we now formally capture the two desiderata discussed above (data exchange equivalence and homomorphic containment) into the following definition of a *relaxed chase-inverse*.

Definition 6 (Relaxed Chase-Inverse). Let $\mathcal{M}$ be a GLAV schema mapping from a schema $\mathbf{S_1}$ to a schema $\mathbf{S_2}$. We say that $\mathcal{M}^*$ is a *relaxed chase-inverse* of $\mathcal{M}$ if $\mathcal{M}^*$ is a GLAV schema mapping from $\mathbf{S_2}$ to $\mathbf{S_1}$ such that, for every instance I over S_1, the following properties hold for the instance $U = chase_{\mathcal{M}^*}(chase_{\mathcal{M}}(I))$:

(a) $U \leftrightarrow_{\mathcal{M}} I$ (data exchange equivalence w.r.t. $\mathcal{M}$),
(b) $U \rightarrow I$ (homomorphic containment).

The notion of relaxed chase-inverse originated in (Fagin et al, 2009b), under the name of *universal-faithful inverse*. The definition given in (Fagin et al, 2009b) had, however, a third condition called *universality*, which turned out to be redundant (and equivalent to homomorphic containment). Thus, the formulation given here for a relaxed chase-inverse is simpler.

Coming back to our example, it can be verified that the above $\mathcal{M}^\dagger$ satisfies the conditions of being a relaxed chase-inverse of $\mathcal{M}''$, thus reflecting the intuition that $\mathcal{M}^\dagger$ is a good "approximation" of an inverse in our scenario.

Since $\mathcal{M}^\dagger$ is a GLAV mapping, we can now apply the composition of $\mathcal{M}^\dagger$ with $\mathcal{M} \circ \mathcal{M}'$, to obtain an SO tgd that specifies $\mathcal{M}^\dagger \circ \mathcal{M} \circ \mathcal{M}'$. This SO tgd is the result of adapting the original schema mapping $\mathcal{M}$ to the new schemas $\mathbf{S}''$ and $\mathbf{T}'$. We leave the full details to the reader.

5.2 More on Relaxed Chase-Inverses

It is fairly straightforward to see that every chase-inverse is also a relaxed chase-inverse. This follows from a well-known property of the chase that implies that whenever $U \leftrightarrow I$ we also have that $U \leftrightarrow_{\mathcal{M}} I$. Thus, the notion of relaxed chase-inverse is a generalization of the notion of chase-inverse; in fact, it is a strict generalization, since the schema mapping $\mathcal{M}^{\dagger}$ in Section 5.1 is a relaxed chase-inverse of $\mathcal{M}''$ but not a chase-inverse of $\mathcal{M}''$. However, for schema mappings that have a chase-inverse, the notions of a chase-inverse and of a relaxed chase-inverse coincide, as stated in the following theorem, which can be derived from results in (Fagin et al, 2009b).

Theorem 4. *Let $\mathcal{M}$ be a GLAV schema mapping from a schema $\mathbf{S_1}$ to a schema $\mathbf{S_2}$ that has a chase-inverse. Then the following statements are equivalent for every GLAV schema mapping $\mathcal{M}^*$ from $\mathbf{S_2}$ to $\mathbf{S_1}$:*

(i) $\mathcal{M}^*$ is a chase-inverse of $\mathcal{M}$.
(ii) $\mathcal{M}^*$ is a relaxed chase-inverse of $\mathcal{M}$.

As an immediate application of the preceding theorem, we conclude that the schema mapping $\mathcal{M}''$ in Section 5.1 has no chase-inverse, because $\mathcal{M}^{\dagger}$ is a relaxed chase-inverse of $\mathcal{M}''$ but not a chase-inverse of $\mathcal{M}''$.

In Section 3.3, we pointed out that chase-inverses coincide with the extended inverses that are specified by GLAV constraints. For schema mappings that have no extended inverses, a further relaxation of the concept of an extended inverse has been considered, namely, the concept of a *maximum extended recovery* (Fagin et al, 2009b). It follows from results established in (Fagin et al, 2009b) that relaxed chase-inverses coincide with the maximum extended recoveries that are specified by GLAV constraints.

6 Implementations and Systems

In this section we examine systems that implement composition and inversion and apply them to the context of schema evolution. We do not attempt to give here a complete survey of all the existing systems and implementations but rather focus on two systems that are directly related to the concepts described earlier and also targeted at schema evolution.

The first system that we will discuss is an implementation of mapping composition that is reported in (Yu and Popa, 2005) and is targeted at the mapping adaptation problem in the context of schema evolution. This implementation is part of the Clio system (Fagin et al, 2009a) and builds on the schema mapping framework of Clio. In particular, it is focused on schema mappings that are expressed as second-order tgds (Fagin et al, 2005b). A different implementation of mapping composition that is worth noting, but which we do not discuss in detail in here, is the one reported in (Bernstein et al, 2008). This system allows a schema mapping to contain not only source-to-target constraints, but also

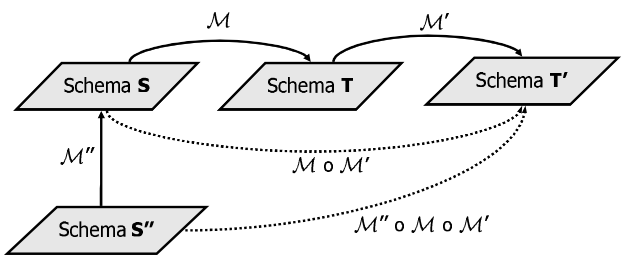

Fig. 5. Using composition (only) in schema evolution.

target constraints, source constraints and target-to-source constraints. Furthermore, the focus is on expressing the composition as a first-order formula (when possible). In this approach, a significant effort is spent on eliminating second-order features (via deskolemization). As a result, the composition algorithm is inherently complex and may not always succeed in finding a first-order formula, even when one exists.

The second system that we will discuss in this section is a more recent one, reported in (Curino et al, 2008), and includes both composition and inversion as part of a framework for schema evolution. This system is focused on the query migration (or adaptation) problem in the context of schema evolution.

6.1 Mapping Composition and Evolution in Clio

The system described in (Yu and Popa, 2005) is part of the larger Clio system (Fagin et al, 2009a) and is the first reported implementation of mapping composition in the context of schema evolution. In this system, both source schema evolution and target schema evolution are described through mappings, which are given in the same language as the original schema mapping (that is to be adapted). However, differently from the earlier diagram shown in Figure 1, the source evolution is required to be given as a schema mapping from $\mathbf{S}''$ to $\mathbf{S}$, and not from $\mathbf{S}$ to $\mathbf{S}''$. (The latter would, intuitively, be a more natural way to describe an evolution of $\mathbf{S}$ into $\mathbf{S}''$.) The main reason for this requirement is that the system described in (Yu and Popa, 2005) preceded the work on mapping inversion. Thus, the only way to apply mapping composition techniques was to require that all mappings form a chain, as seen in Figure 5.

In the system implemented in (Yu and Popa, 2005), the schema mapping language that is used to specify the input mappings (i.e., the original mapping $\mathcal{M}$ and the evolution mappings $\mathcal{M}'$ and $\mathcal{M}''$) are based on second-order (SO) tgds (Fagin et al, 2005b). One reason for this choice is that, as discussed earlier, GLAV mappings are not closed under composition, while SO tgds form a more expressive language that includes GLAV mappings and, moreover, is closed under composition. Another reason is that SO tgds, independently of mapping composition, include features that are desirable for any schema mapping language. In particular, the Skolem terms that can be used in SO tgds enable a

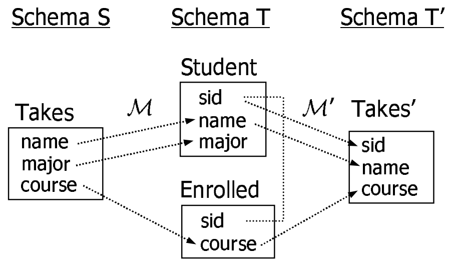

Fig. 6. Example to illustrate SO tgd-based composition and minimization.

much finer control over the creation of new data values (e.g., ids) in the process of data exchange. We shall give such example shortly. A related point is that the language used in (Yu and Popa, 2005) (and also in the larger Clio system), is actually a nested relational extension of SO tgds that can handle XML schemas and can be compiled into XQuery and XSLT. We will not elaborate on the XML aspect here and refer the interested readers to either (Haas et al, 2005) or (Yu and Popa, 2005).

Another main ingredient of the system described in (Yu and Popa, 2005) is the use of an operational semantics of mapping composition that is based on the chase. Under this semantics, the composition algorithm needs to find an expression that is chase-equivalent only, rather than logically equivalent, to the composition of the two input mappings. (We define shortly what chase-equivalence means.) In turn, the use of this chase-based semantics of composition enables *syntactic minimization* of the outcome of mapping composition. For schema evolution, such minimization is shown to be essential in making the outcome of mapping adaptation intuitive (and presentable) from a user point of view. This is especially true for the larger schemas that arise in practice, where the outcome of mapping composition (under the general semantics) is complex, contains many self-joins and it is generally hard to understand.

To make the above ideas more concrete, consider the following schema evolution scenario depicted in Figure 6. This scenario is a variation on the earlier schema evolution scenario described in Figure 2. In the new scenario, we focus on the target schema evolution alone. Furthermore, there are several changes in the schemas as well as the mappings. We assume that the source schema **S** consists of one relation `Takes` where instead of a student id (`sid`) we are given a student name (`name`). However, the target schema **T**, consisting of the two relations `Student` and `Enrolled`, still requires a student id that must relate the two relations. The schema mapping that relates the two schemas is now given as the following SO tgd:

$$\begin{aligned}\mathcal{M}: \quad \exists f(\; & \texttt{Takes}(n, m, co) \rightarrow \texttt{Student}(f(n), n, m) \\ \wedge \; & \texttt{Takes}(n, m, co) \rightarrow \texttt{Enrolled}(f(n), co) \;)\end{aligned}$$

In the above SO tgd, f is an existentially quantified Skolem function and, for each student name n, the Skolem term $f(n)$ represents the associated student id that is used to populate both `Student` and `Enrolled` tuples. The use of such Skolem terms offer fine control over the creation of target values. By changing the parameters given to the Skolem function, one can change how the target values are populated. For example, if we know that a student name does not uniquely identify a student, but the student name together with the major does, then we can change $f(n)$ to $f(n, m)$ to reflect such dependency.

Assume now that the target schema evolves to a new schema $\mathbf{T}'$ that consists of a single relation `Takes′` that keeps the association between `sid`, `name` and `course`, while dropping the major. The target evolution can be described by the following mapping:

$$\mathcal{M}' : \quad \texttt{Student}(s, n, m) \wedge \texttt{Enrolled}(s, co) \rightarrow \texttt{Takes}'(s, n, co)$$

It can be verified that the composition of $\mathcal{M}$ and $\mathcal{M}'$ is expressed by the following SO tgd:

$$\begin{aligned}\sigma : \quad \exists f(\; & \texttt{Takes}(n, m, co) \wedge \texttt{Takes}(n', m', co') \wedge (f(n) = f(n')) \\ & \rightarrow \texttt{Takes}'(f(n'), n', co) \;)\end{aligned}$$

This mapping is surprisingly complex, but still correct (i.e., σ expresses $\mathcal{M} \circ \mathcal{M}'$). It accounts for the fact that, given a source instance I over $\mathbf{S}$ and a target instance J over $\mathbf{T}$, two different student names n and n' occuring in different tuples of I may relate to the same `sid` in J. In other words, the function f that is existentially quantified by the original mapping $\mathcal{M}$ may have the property that $f(n) = f(n')$ for some distinct names n and n'. In order to account for such possibility, the composition σ includes a self-join on `Takes` and the test $f(n) = f(n')$.

Minimization of SO tgds under chase-equivalence. If we now take the operational view behind schema mappings, the above σ can be drastically simplified. Under the operational view, a mapping $\mathcal{M}$ does not describe an arbitrary relationship between instances I and J over two schemas but rather a transformation which, given a source instance I, *generates* the target instance $J = \mathit{chase}_{\mathcal{M}}(I)$. We refer the reader to (Fagin et al, 2005b) for the definition of the chase with SO tgds. Here, we point out that an important property of this chase is that it always generates different values (nulls) for different arguments to the Skolem functions. Hence, for our example, the equality $f(n) = f(n')$ can happen only if $n = n'$. As a result, the above σ reduces to the following SO tgd:

$$\sigma_0 : \quad \exists f(\; \texttt{Takes}(n, m, co) \rightarrow \texttt{Takes}'(f(n), n, co) \;)$$

The above SO tgd is much simpler and more intuitive than the earlier σ. Just by looking at the diagram in Figure 6, one would expect the overall adapted mapping from $\mathbf{S}$ to $\mathbf{T}'$ to be as close as possible to an identity schema mapping. The SO tgd σ_0 accomplishes this desideratum while still incorporating the id generation behavior via $f(n)$ that is given in the original mapping $\mathcal{M}$.

The reduction algorithm implemented in (Yu and Popa, 2005) systematically replaces every equality between two Skolem terms with the same function symbol by the equalities of their arguments, until all equalities that involve such Skolem terms are eliminated. The algorithm also eliminates every implication where the left-hand side contains an equality between two Skolem terms that use different Skolem functions. Intuitively, such equalities cannot be satisfied during the chase; hence, the implications that contain them can be dropped. Finally, the algorithm uses conjunctive-query minimization (Chandra and Merlin, 1977) type of techniques to eliminate any redundant relational atoms in the resulting mappings. For example, in the above σ, once we replace $f(n) = f(n')$ with $n = n'$, the second `Takes` atom becomes $\texttt{Takes}(n, m', co')$; it can then be eliminated, since it is subsumed by the first `Takes` atom, and neither m' nor co' are used in the right-hand side of the implication.

The main observation behind this reduction algorithm is that its output SO tgd (e.g., σ_0) is *chase-equivalent* to the input SO tgd (e.g., σ).

Definition 7. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two schema mappings from $\mathbf{S}$ to $\mathbf{T}$ that are specified by SO tgds (or in particular by GLAV mappings). We say that $\mathcal{M}_1$ and $\mathcal{M}_2$ are *chase-equivalent* if for every source instance I, we have that $chase_{\mathcal{M}_1}(I) \leftrightarrow chase_{\mathcal{M}_2}(I)$.

Theorem 5 ((Yu and Popa, 2005)). *Every SO tgd σ is chase-equivalent to its reduced form σ_0.*

We note that the above σ_0 is not logically equivalent to the input σ. In general, the notion of chase-equivalence is a relaxation of the concept of logical equivalence. A systematic study of relaxed notions of equivalence of schema mappings appeared later in (Fagin et al, 2008a). For schema mappings specified by GLAV mappings or, more generally, by SO tgds, the above notion of chase-equivalence turns out to be the same as the notion of *CQ-equivalence* of schema mappings studied in (Fagin et al, 2008a). There, two schema mappings $\mathcal{M}_1$ and $\mathcal{M}_2$ are CQ-equivalent if for every source instance I, the *certain answers* of a conjunctive query q are the same under both $\mathcal{M}_1$ and $\mathcal{M}_2$. For our example, the CQ-equivalence of σ_0 and σ is another argument of why we can use σ_0 instead of σ.

We also note that σ_0 represents a relaxation of the composition $\mathcal{M} \circ \mathcal{M}'$ (since σ_0 is chase-equivalent but not logically equivalent to σ, which expresses $\mathcal{M} \circ \mathcal{M}'$). Such relaxation of composition appears early in the work of Madhavan and Halevy (Madhavan and Halevy, 2003).[6] The concept used there is based, implicitly, on CQ-equivalence; however, their results are limited to GLAV mappings, which, in general, are not powerful enough to express composition (even under the relaxed form) (Fagin et al, 2005b).

Since schemas can be quite large in practice, mapping composition as well as mapping reduction can be expensive. Therefore, a great deal of the work in (Yu and Popa, 2005) is spent on developing pruning techniques that identify the parts

[6] In fact, that is how Madhavan and Halevy defined composition of schema mappings.

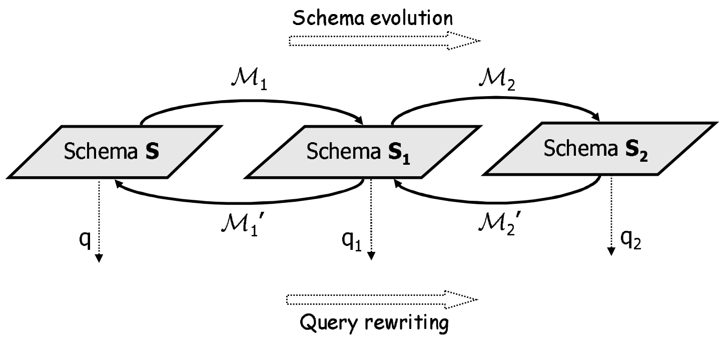

Fig. 7. Schema evolution and query rewriting in PRISM.

of a schema mapping that are not affected by the changes in the schemas, and hence do not need to be involved in the process of composition and reduction. We refer the interested reader to (Yu and Popa, 2005) for more details on this.

6.2 The PRISM Workbench: Query Adaptation

The PRISM project, described in (Curino et al, 2008), has the overall goal of automating as much as possible the database administration work that is needed when schemas evolve. Under this general umbrella, one of the main concrete goals in PRISM is to support migration (or adaptation) of queries from old (legacy) schemas to the new evolved schemas. Similar to the Clio-based schema evolution system in (Yu and Popa, 2005), PRISM also uses schema mappings (although, in a restricted form) to describe the evolution of schemas. However, differently from the Clio-based system, the focus in PRISM is not on mapping adaptation but on query adaptation. More concretely, in the Clio-based system, we are given a schema mapping from **S** to **T** and the goal is to adapt it when either **S** or **T** changes, while in PRISM we are given a query q over a schema **S** and the goal is to adapt it when **S** changes. Because it is targeted at queries, PRISM makes prominent use of query rewriting. In particular, it applies the chase and backchase algorithm introduced in (Deutsch et al, 1999) for query rewriting under constraints. Additionally, PRISM also makes use of the schema mapping operations that we described earlier (i.e., composition and inversion), in order to enable the application of the query rewriting algorithm and in order to optimize its application.

We use Figure 7 to illustrate the type of functionality that PRISM aims to achieve. There, schema **S** represents an initial (legacy) schema that goes through several steps of change, forming a schema evolution chain: from **S** to $\mathbf{S_1}$, then to S_2, and so on. Each of the evolution steps can be described by a mapping. However, these mappings are not arbitrary and must correspond to a set of predefined *Schema Modification Operations* that allow only for certain type of schema modifications. Examples of such modifications are: copying of a table, renaming of a table or of a column, taking the union of two tables into one, decomposing a table into two, and others. These operations are chosen carefully so that they represent the most common forms of schema evolution that arise in practice, but also to allow for invertibility. More precisely, each of the evolution mappings that are allowed in PRISM is guaranteed to have a quasi-inverse (Fagin et al, 2008b). Thus, in Figure 7, $\mathcal{M}'_1$ is a quasi-inverse of $\mathcal{M}_1$, and $\mathcal{M}'_2$ is a quasi-inverse of $\mathcal{M}_2$. The main reason for why each evolution mapping must have a reverse mapping is that the presence of mappings in both directions (i.e., from $\mathbf{S}$ to $\mathbf{S_1}$, and from $\mathbf{S_1}$ to $\mathbf{S}$) is essential for the application of query reformulation algorithms, as we explain next.

More concretely, query reformulation in PRISM can be phrased as follows. We are given a query q over the original schema $\mathbf{S}$. We assume one step of evolution, with mapping $\mathcal{M}_1$ from $\mathbf{S}$ to $\mathbf{S_1}$ and reverse mapping $\mathcal{M}'_1$ from $\mathbf{S_1}$ to $\mathbf{S}$. The problem is to find a query q_1 over the schema $\mathbf{S_1}$ such that q_1 is equivalent to q, where equivalence is interpreted over the union $\mathbf{S} \cup \mathbf{S_1}$ of the two schemas and where $\mathcal{M}_1$ and $\mathcal{M}'_1$ form constraints on the larger schema. In other words, we are looking for a query q_1 to satisfy $q(K) = q_1(K)$, for every instance K over $\mathbf{S} \cup \mathbf{S_1}$ such that K satisfies the union of the constraints in $\mathcal{M}_1$ and $\mathcal{M}'_1$. In turn this is an instance of the general problem of query reformulation under constraints (Deutsch et al, 2006), which can be solved by the chase and backchase method (Deutsch et al, 1999). The application of the chase and backchase method in this context consists of, first, applying the chase on q with the constraints in $\mathcal{M}_1$, and then on applying the (back) chase with the reverse constraints in $\mathcal{M}'_1$, in order to find equivalent rewritings of q.

Before we concretely illustrate on an example the application of the chase and backchase in the PRISM context, we need to point out that for multiple evolution steps, the query reformulation problem needs to take into account all the direct and reverse mappings alongs the chain (e.g., $\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{M}'_1$ and $\mathcal{M}'_2$ for two evolution steps). Thus, as the evolution chain becomes longer, the number of constraints involved in query reformulation becomes larger. To reduce the number of constraints needed for rewriting, PRISM makes repeated use of composition to replace two consecutive schema mappings by one schema mapping. Since PRISM restricts mappings such as $\mathcal{M}_1$ and $\mathcal{M}_2$ to always be GAV schema mappings, the composition $\mathcal{M}_1 \circ \mathcal{M}_2$ can also be expressed as a GAV mapping (see our earlier Theorem 1, part 1). The same cannot be done for the reverse schema mappings (e.g., $\mathcal{M}'_1$ and $\mathcal{M}'_2$), which are quasi-inverses of the direct mappings, and require in general a more complex language that includes disjunction (see (Fagin et al, 2008b)). The exact language in which to express composition of such schema mappings (i.e., with disjunction) is an open research problem.

To make the above ideas more concrete, consider the following schema evolution example shown in Figure 8. This example is based on two of our earlier schemas (see $\mathbf{S}$ and $\mathbf{T}$ in Figure 2). Here, the schema $\mathbf{S}$ represents the “old” schema, which then evolves into a “new” schema $\mathbf{S_1}$. The evolution step from $\mathbf{S}$ to $\mathbf{S_1}$ can be described by one of the Schema Modification Operations (SMOs) that the PRISM workbench allows. In particular, this evolution step is an appli-

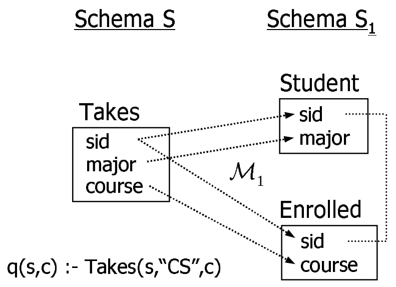

Fig. 8. Example of schema evolution with a query to be rewritten.

cation of the `Decompose` operator where the table `Takes` is split into two tables `Student` and `Enrolled` that share the common attribute `sid`. The application of the `Decompose operator` in this case can be represented by the following GAV mapping (this is the same as the earlier $\mathcal{M}$ in Section 3):

$$\begin{aligned}\mathcal{M}_1: \ & \texttt{Takes}(s,m,co) \rightarrow \texttt{Student}(s,m)\\ & \texttt{Takes}(s,m,co) \rightarrow \texttt{Enrolled}(s,co)\end{aligned}$$

Assume now that we have a legacy query q that is formulated in terms of the old schema. This query, shown in Figure 8, retrieves all pairs of student id and course where the major is "CS". The goal is to adapt, via query rewriting, the query q into a new query q_1 that is formulated in terms of the new schema $\mathbf{S_1}$ and is equivalent to q.

The first step is to retrieve a quasi-inverse $\mathcal{M}_1'$ of $\mathcal{M}_1$. As mentioned earlier, each evolution step in PRISM is an instance of one of the predetermined Schema Modification Operations. Thus, a quasi-inverse always exists and can be chosen by the system or by the user. In this case, the following is a quasi-inverse of $\mathcal{M}_1$:

$$\mathcal{M}_1' : \texttt{Student}(s,m) \wedge \texttt{Enrolled}(s,co) \rightarrow \texttt{Takes}(s,m,co)$$

The next step is to apply the chase and backchase algorithm to find rewritings of q that are equivalent given the union of the constraints in $\mathcal{M}_1$ and $\mathcal{M}_1'$. The following query over schema $\mathbf{S_1}$ is such an equivalent rewriting and will be returned by the chase and backchase algorithm.

$$q_1(s,c) \ :- \ \texttt{Student}(s, \text{“}CS\text{”}) \wedge \texttt{Enrolled}(s,c)$$

The above quasi-inverse $\mathcal{M}_1'$ also happens to be a chase-inverse of $\mathcal{M}_1$. In general, however, quasi-inverses differ from chase-inverses (or relaxed chase-inverses), and one may find quasi-inverses with non-intuitive behavior (e.g., a quasi-inverse that is not a chase-inverse, even when a chase-inverse exists). We note that the PRISM development preceded the development of chase-inverses or relaxed chase-inverses.

We also remark that the language needed to express quasi-inverses requires disjunction. As a result, PRISM uses an extension of the chase and backchase algorithm that is able to handle disjunctive dependencies; this extension was developed as part of MARS (Deutsch and Tannen, 2003). Finally, we note that we may not always succeed in finding equivalent reformulations, depending on the input query, the evolution mappings and also on the the quasi-inverses that are chosen. Hence, PRISM must still rely on a human DBA to solve exceptions.

7 Other Related Work

We have emphasized in this paper the operational view of schema evolution, where a schema mapping $\mathcal{M}$ is viewed as a transformation, which given an instance I produces $\mathit{chase}_{\mathcal{M}}(I)$. Under this view, we have emphasized two types of operational inverses: the chase-inverse (with its exact variation), which corresponds to the absence of information loss, and the relaxed chase-inverse, which is designed for the case of information loss. However, there is quite a lot of additional (and related) work on mapping inversion that studies more general, non-operational notions of inverses. These notions can be categorized into three main notions: inverses (Fagin, 2007), quasi-inverses (Fagin et al, 2008b) and maximum recoveries (Arenas et al, 2008).

Most of the technical development on inverses, quasi-inverses, and maximum recoveries was originally focused on the case when the source instances were assumed to contain no nulls, that is, they were assumed to be ground. However, in practice, such an assumption is not realistic, since an instance with nulls can easily arise as the result of another schema mapping. This is especially true in schema evolution scenarios, where we can have chains of mappings describing the various evolution steps. To uniformly deal with the case where instances can have nulls, the notions of inverses and of maximum recoveries were extended in (Fagin et al, 2009b) by systematically making use of the notion of homomorphism between instances with nulls as a replacement for the more standard containment of instances. In addition to their benefit in dealing with non-ground instances, it turns out that the two extended notions, namely extended inverses and maximum extended recoveries, have the operational counterpart that we want. More concretely, when $\mathcal{M}$ is a GLAV mapping, we have that: (1) extended inverses that are also expressed as GLAV mappings coincide with chase-inverses, and (2) maximum extended recoveries that are also expressed as GLAV mappings coincide with relaxed chase-inverses. (Note that extended inverses and maximum extended recoveries, or any of the other semantic notions of inverses, need not be expressible as GLAV mappings, in general). This correspondence between two very general semantic notions, on one hand, and two procedural and practical notions of inverses, on the other hand, is interesting in itself.

Finally, we note that there are certain limitations to what composition and inversion can achieve in the context of schema evolution. For example, if we refer back to Figure 1, it is conceivable that the composition $\mathcal{M} \circ \mathcal{M}'$ does not always give the "complete" mapping from $\mathbf{S}$ to $\mathbf{T}'$. Instead, the "complete" mapping from $\mathbf{S}$ to $\mathbf{T}'$ may require *merging* the schema mapping $\mathcal{M} \circ \mathcal{M}'$ with an additional mapping that relates directly $\mathbf{S}$ to $\mathbf{T}'$. Such additional mapping may be defined separately by a user to account for, say, a schema element that occurs in both $\mathbf{S}$ and $\mathbf{T}'$ but does not occur in $\mathbf{T}$. The operation of merging two schema mappings appears in the model management framework (Melnik et al, 2005) under the term *Confluence*; a more refined version of merge, together with an algorithm for it, appears in (Alexe et al, 2010).

8 Concluding Remarks

In this chapter, we illustrated how the composition operator and the inverse operator on schema mappings can be applied to schema evolution. The techniques presented here rely on the existence of chase-inverses or relaxed chase-inverses, which, in particular, are required to be specified by GLAV constraints. Much more remains to be done in the study of schema mappings for which no relaxed chase-inverse exists. In this direction, research issues include: (1) What is the exact language for expressing maximum extended recoveries? (2) How does this language compose with SO tgds? (3) What do inverses of SO tgds look like? More broadly, is there a unifying schema-mapping language that is closed under both composition and the various flavors of inverses, and, additionally, has good algorithmic properties?

Acknowledgements The authors wish to thank Erhard Rahm for reading an earlier version of this chapter and providing valuable feedback. The research of Kolaitis and Tan is supported by NSF grant IIS-0430994 and NSF grant IIS-0905276. Tan is also supported by NSF CAREER award IIS-0347065.

Bibliography

Alexe B, Hernández MA, Popa L, Tan WC (2010) MapMerge: Correlating Independent Schema Mappings. PVLDB 3(1)

Arenas M, Pérez J, Riveros C (2008) The Recovery of a Schema Mapping: Bringing Exchanged Data Back. In: PODS, pp 13–22

Bernstein PA (2003) Applying Model Management to Classical Meta-Data Problems. In: Conference on Innovative Data Systems Research (CIDR), pp 209–220

Bernstein PA, Green TJ, Melnik S, Nash A (2008) Implementing Mapping Composition. VLDB Journal 17(2):333–353

Chandra AK, Merlin PM (1977) Optimal Implementation of Conjunctive Queries in Relational Data Bases. In: ACM Symposium on Theory of Computing (STOC), pp 77–90

Curino C, Moon HJ, Zaniolo C (2008) Graceful Database Schema Evolution: The PRISM Workbench. PVLDB 1(1):761–772

Deutsch A, Tannen V (2003) MARS: A System for Publishing XML from Mixed and Redundant Storage. In: International Conference on Very Large Data Bases (VLDB), pp 201–212

Deutsch A, Popa L, Tannen V (2006) Query Reformulation with Constraints. SIGMOD Record 35(1):65–73

Deutsch A, Popa L, Tannen V (1999) Physical Data Independence, Constraints and Optimization with Universal Plans. In: International Conference on Very Large Data Bases (VLDB), pp 459–470

Fagin R (2007) Inverting Schema Mappings. ACM Transactions on Database Systems (TODS) 32(4)

Fagin R, Kolaitis PG, Popa L, Tan WC (2004) Composing Schema Mappings: Second-Order Dependencies to the Rescue. In: ACM Symposium on Principles of Database Systems (PODS), pp 83–94

Fagin R, Kolaitis PG, Miller RJ, Popa L (2005a) Data Exchange: Semantics and Query Answering. Theoretical Computer Science (TCS) 336(1):89–124

Fagin R, Kolaitis PG, Popa L, Tan WC (2005b) Composing Schema Mappings: Second-order Dependencies to the Rescue. ACM Transactions on Database Systems (TODS) 30(4):994–1055

Fagin R, Kolaitis PG, Nash A, Popa L (2008a) Towards a Theory of Schema-Mapping Optimization. In: ACM Symposium on Principles of Database Systems (PODS), pp 33–42

Fagin R, Kolaitis PG, Popa L, Tan WC (2008b) Quasi-Inverses of Schema Mappings. ACM Transactions on Database Systems (TODS) 33(2)

Fagin R, Haas LM, Hernández MA, Miller RJ, Popa L, Velegrakis Y (2009a) Clio: Schema Mapping Creation and Data Exchange. In: Conceptual Modeling: Foundations and Applications, Essays in Honor of John Mylopoulos, Springer, pp 198–236

Fagin R, Kolaitis PG, Popa L, Tan WC (2009b) Reverse Data Exchange: Coping with Nulls. In: ACM Symposium on Principles of Database Systems (PODS), pp 23–32

Fagin R, Kolaitis PG, Popa L, Tan WC (2010) Reverse Data Exchange: Coping with Nulls. Full version of Fagin et al (2009b)

Haas LM, Hernández MA, Ho H, Popa L, Roth M (2005) Clio Grows Up: From Research Prototype to Industrial Tool. In: SIGMOD, pp 805–810

Hartung M, Terwilliger J, Rahm E (2010) Recent Advances in Schema and Ontology Matching. In: Advances in Schema Matching and Mapping, Data-Centric Systems and Applications 5258, Springer

Lenzerini M (2002) Data Integration: A Theoretical Perspective. In: ACM Symposium on Principles of Database Systems (PODS), pp 233–246

Madhavan J, Halevy AY (2003) Composing Mappings Among Data Sources. In: International Conference on Very Large Data Bases (VLDB), pp 572–583

Melnik S (2004) Generic Model Management: Concepts and Algorithms, Lecture Notes in Computer Science, vol 2967. Springer

Melnik S, Bernstein PA, Halevy A, Rahm E (2005) Applying Model Management to Executable Mappings. In: SIGMOD, pp 167–178

Nash A, Bernstein PA, Melnik S (2005) Composition of Mappings Given by Embedded Dependencies. In: ACM Symposium on Principles of Database Systems (PODS), pp 172–183

Shu NC, Housel BC, Taylor RW, Ghosh SP, Lum VY (1977) EXPRESS: A Data EXtraction, Processing, amd REStructuring System. ACM Transactions on Database Systems (TODS) 2(2):134–174

Velegrakis Y, Miller RJ, Popa L (2003) Mapping Adaptation under Evolving Schemas. In: International Conference on Very Large Data Bases (VLDB), pp 584–595

Yu C, Popa L (2005) Semantic Adaptation of Schema Mappings when Schemas Evolve. In: VLDB, pp 1006–1017