A New Framework for Designing Schema Mappings

Bogdan Alexe[1] and Wang-Chiew Tan[2]

[1] IBM Research - Almaden, San Jose, CA
balexe@us.ibm.com
[2] University of California, Santa Cruz, CA
tan@cs.ucsc.edu

Abstract. One of the fundamental tasks in information integration is to specify the relationships, called *schema mappings*, between database schemas. Schema mappings specify how data structured under a source schema is to be transformed into data structured under a target schema. The design of schema mappings is usually a non-trivial and time-intensive process and the task of designing schema mappings is exacerbated by the fact that schemas that occur in real life tend to be large and heterogeneous. Traditional approaches for designing schema mappings are either manual or performed through a user interface from which a schema mapping is interpreted from correspondences between attributes of the source and target schemas. These correspondences are either specified by the user or automatically derived by applying schema matching on the two schemas.

In this paper, we examine an alternative approach that allows a user to follow the "divide-design-merge" paradigm for specifying a schema mapping. The user can choose to independently design schema mappings for smaller portions of the source and target schema. Afterwards, the user can interact with the system to refine and further design schema mappings through the use of data examples. Finally, in the merge phase, a global schema mapping is generated through the correlation of the individual schema mappings.

Keywords: Schema mappings, data examples, merge.

1 Introduction

The need to combine information that resides in heterogeneous, and typically independently created data sources often arises in enterprises. In today's information age, where vast amounts of (un)structured data is available on the Web, and where many data sources collected or curated by different organizations are made publicly available (e.g., [20, 34]), the demand for technology that can effectively combine disparate data sources goes well beyond enterprises. The process of combining different data sources into one is called *information integration*, which is a broad term that encompasses *data integration* and *data exchange*. The goal of *data integration* is to create a single virtual view of the underlying data sources and provide seamless and transparent access to these data sources through the virtual view. On the other hand, the goal of *data exchange* is to create a materialized view of the underlying data sources.

Systems such as Multibase [32] and EXPRESS [31] have pioneered the study of data integration and data exchange respectively and considerable research effort has been

V. Tannen et al. (Eds.): Buneman Festschrift, LNCS 8000, pp. 56–88, 2013.

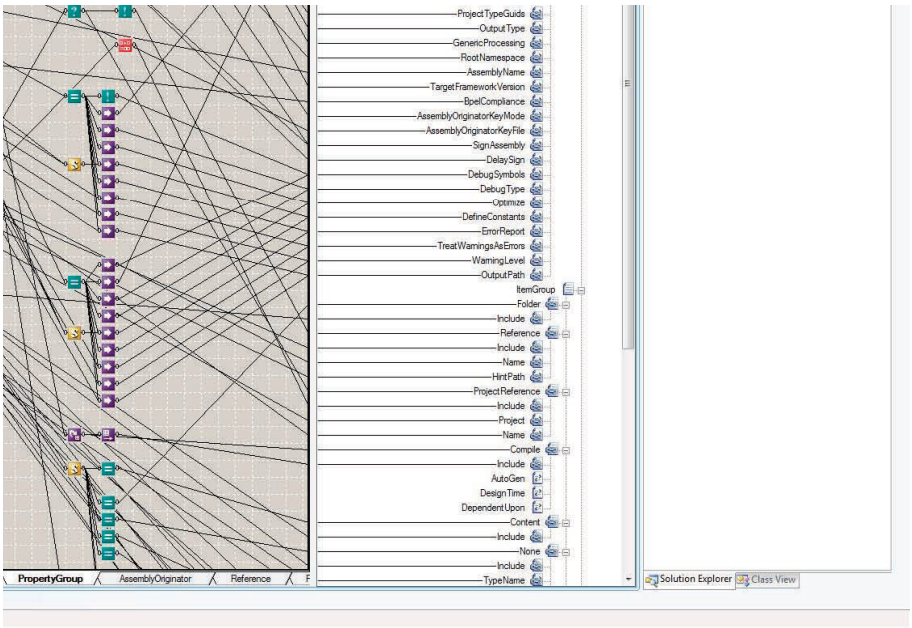


Fig. 1. Screenshot of a mapping design tool (from [12])

put into addressing information integration challenges since Multibase and EXPRESS. In practice, information integration is still a difficult and time-consuming process that incurs high costs in terms of money and human effort and recent reports provide strong evidence of this. For example, [12] stated that information integration is frequently "the biggest and most expensive challenge that information-technology shops face" and "information integration is thought to consume about 40% of their budget".

Even though data integration and data exchange differ in their goals, they share a common abstraction, called *schema mappings*, which describe the relationship between database schemas. In research prototypes such as Clio [16] and HePToX [15], the term *schema mappings* is used to refer to the high-level declarative specfication that specifies the semantics of translating data from the *source schema* to the *target schema*. However, commercial data transformation systems such as Altova Mapforce [25], Stylus Studio [33] and Microsoft BizTalk Mapper [13] often refer to schema mappings or *data mappings* as the executable script (e.g., XQuery or SQL) that can be used to translate data from the source schema to the target schema. Regardless of terminology, most of these tools work in two steps. First, a visual interface is used to solicit all known *attribute correspondences* between elements of the two schemas from the user. Such correspondences are usually depicted as arrows between the attributes of the source and target schemas. For illustration, Figure 1 presents a screenshot of a mapping design tool with a number of correspondences between attributes of a source schema on the left and a target schema on the right. Once the correspondences are established, systems such as Altova MapForce, Stylus Studio, and Microsoft Biztalk Mapper, interpret them directly

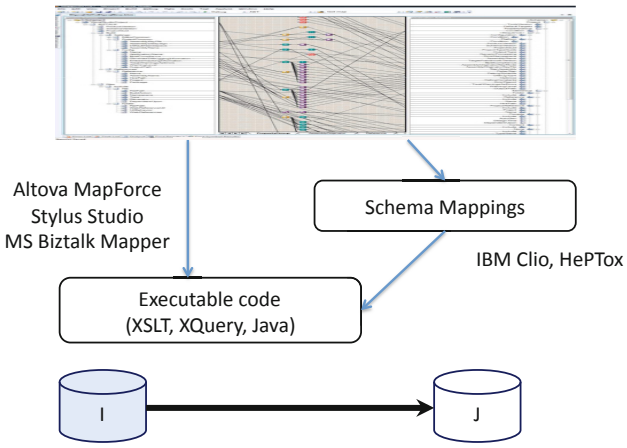


Fig. 2. Generic architecture of schema mapping design systems

into an executable script (e.g., XQuery or SQL query), which can be executed on an instance of the source schema to obtain an instance of the target schema. Other systems such as Clio or HePToX, interpret the correpondences into an internal representation (which we refer to as schema mappings in this article), and this representation can be compiled over different runtimes. Often, the user will need to refine the schema mapping (whether as an internal representation or an executable script) that is derived from such tools in order to achieve the desired transformation semantics.

The previously outlined two-step schema mapping design framework is illustrated in Figure 2. While this framework provides a method for end users to visually specify a schema mapping, it lacks support for reusability and for modularity in design; A schema mapping between two schemas must always be designed all-at-once. In particular, this methodology does not allow the design of a schema mapping to be divided up and designed modularly in different steps with intermediate schemas. Furthermore, the user must be familiar with the language of schema mappings in order to refine them. For the rest of this article, we will describe a new framework for designing schema mappings that will overcome some of the limitations of existing schema mapping design tools. Details of this framework can be found in the dissertation of Bogdan Alexe [8].

2 Our Divide-Design-Merge Framework

Our framework for designing schema mappings between two schemas follows three main steps: Divide, Design, and Merge, as outlined in Figure 3. This new framework overcomes some of the aforementioned limitations of the existing mapping design paradigm.

Since smaller mappings tend to be easier to create and understand, our framework allows a schema mapping between large source and target schemas to be divided up

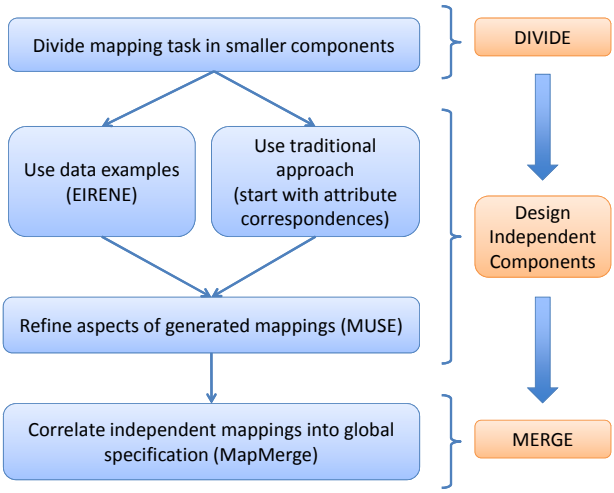


Fig. 3. Divide-Design-Merge Workflow

and designed through independent components. Furthermore, the design of each such schema mapping can be broken up into multiple smaller intermediate steps which involve intermediate schemas. Each of the schema mappings can either be designed with existing approaches (i.e., via attribute correspondences) or via our new approach (i.e., Eirene component system) that requires the user to specify *data examples*, which are pairs of source instance and expected target instance. After this, various components of a schema mapping can be refined through our Muse component system. Finally, in the merge phase, a global schema mapping is generated through the correlation of the individual mapping components (i.e., MapMerge component system). In this new framework, schema mappings that have been previously designed for some of the components can be saved, reused, and customized further at a later time.

We note that in the divide phase, the process of dividing or breaking up schema mappings into smaller "chunks" that are more amenable to design and understanding is entirely driven by the user. It will be interesting work to further design a component that will suggest strategies for such divisions.

2.1 An Example

As mentioned before, the user may choose to divide the design task into smaller components that can be designed independently. For instance, in Figure 4, the design of a schema mapping from schema $\mathbf{S}_1$ to schema $\mathbf{S}_4$ can be divided into a sequence of steps, involving the intermediate schemas $\mathbf{S}_2$ and $\mathbf{S}_3$. Existing schema mapping design tools would only allow designing a monolithic end-to-end schema mapping from $\mathbf{S}_1$ to $\mathbf{S}_4$. In our framework, the user can design smaller mappings independently and merge them together at the end. For instance, the user can start by designing the mapping, denoted by t_1, from *Group* in $\mathbf{S}_1$ to *Dept* in $\mathbf{S}_2$, then the mapping t_2 relating a join of *Works* and

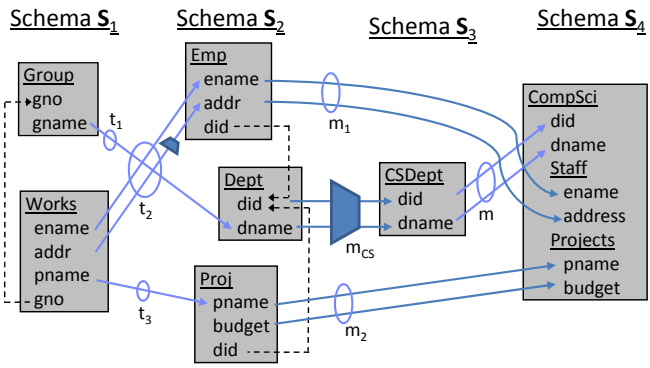


Fig. 4. Designing a schema mapping from the first schema $\mathbf{S}_1$ to the last schema $\mathbf{S}_4$

Group to *Emp* and *Dept*, and so on. For this toy example, it is conceivable that the user would successfully design a mapping directly from $\mathbf{S}_1$ to $\mathbf{S}_4$ (or even $\mathbf{S}_1$ to $\mathbf{S}_2$) with relatively little effort. However, in real-life scenarios, it is typically difficult to understand the entire schemas and to grasp the complexities of the desired global transformation all at once.

Eirene. The design of each component mapping can be driven by data examples. A data example is a pair of input and output instances. Intuitively, a data example specifies the expected output for a given input and represents a partial specification of the desired semantics. This is beneficial, since users may be familiar with their data and the use of data examples is akin to specifying test cases during program debugging to ensure that programs behave as intended.

The Eirene component of our system is a schema mapping design component that takes as input a set of data examples provided by the user. In turn, Eirene outputs a schema mapping that "fits" the set of data examples, if such schema mapping exists. Referring back to Figure 4, the design of t_2 can be achieved through Eirene by providing a data example that reflects the transformation semantics that the user expects from the mapping. In this case, the source instance of the data example may consist of a *Group* tuple and a *Works* tuple that agree on their *gno* attributes, while the target instance may consist of an *Emp* tuple and a *Dept* tuple that have the same *did* value. Furthermore, the tuples may be specified in such a way that the *gname* and *dname* values are the same across the *Group* and *Dept* tuple. In addition, the *ename* and *addr* of the *Works* tuple are identical, respectively, to the *ename* and *addr* of the *Emp* tuple in the target. This reflects that the desired transformation semantics is to migrate *gname, ename, addr* to the corresponding "locations" in the target. For this data example, the system will determine that a fitting schema mapping exists, and it will generate such mapping that will produce the desired target instance on the corresponding source instance of each data example.

Eirene can also be used to refine a schema mapping that already exists. To do this, Eirene will first generate a set of *canonical data examples* for the existing mapping. The user can then "tweak" the canonical data examples, and Eirene will generate a new

mapping that fits, if possible. Alternatively, a schema mapping can also be designed using the traditional methodology via attribute correspondences, imported from previous design work, and augmented with new attribute correspondences and additional customizations.

Muse. The Muse component of our system assists the user with refining the existing schema mappings. The focus of Muse is to use data examples to help the user refine two important mapping features: *grouping semantics* and *disambiguation*. The basic idea behind Muse is to present the user with different data examples, where each data example represents a specific (grouping/disambiguation) semantics of the underlying specification. The choices made by the user will allow the Muse system to automatically refine the underlying specification.

Referring to Figure 4 again, Muse can assist the user with specifying how the nested *Staff* set of tuples should be grouped under the *CompSci* root of schema $\mathbf{S}_4$. The semantics of grouping *Staff* is determined by its set identifier, which consists of a *Skolem function* parameterized by some of the attributes in schemas $\mathbf{S}_2$ and $\mathbf{S}_3$. By presenting differentiating examples that can be used to distinguish among alternative grouping semantics, Muse helps the user determine which attributes should be used to parameterize the nested set identifier of *Staff*.

In addition, Muse can also help the user understand the right interpretation of a visual specification. This part of Muse works with traditional schema mapping design systems, where the user specifies a set of attribute correspondences between a source and a target schema. (A *visual specification* consists of the source and target schema, and the attribute correspondences.) A visual specification is *ambiguous* if more than one schema mapping can be interpreted from the visual specification[1]. In case a visual specification is ambiguous, our Muse system will detect the ambiguity and present the user with a carefully constructed "data example" that essentially represents the transformation semantics of all alternative schema mappings. The target instance of the "data example" contains choices of data values on certain attributes of tuples. Each selection of a value from a choice by the user will prune away some schema mappings among the set of all possible schema mappings that can be interpreted from the visual specification. At the end, when all choices have been made, only one schema mapping will remain.

MapMerge. When all component schema mappings are designed, the *MapMerge* schema mapping operator [6] can be invoked to automatically generate a meaningful overall mapping between each pair of source and target schemas. MapMerge takes as input a set of schema mappings between the same source and target schema, and it returns a schema mapping that correlates the specifications given by the individual mapping components. As we shall show, this orchestration phase is necessary since simply considering the union of input mappings is inadequate in general; in the context of data exchange, simply taking the union of input schema mappings may result in the loss

[1] In systems such as Clio, a default schema mapping is generated when a visual specification is ambiguous. The user can choose among alternative mappings by manually inspecting the alternatives and picking one of the alternatives.

of certain data associations and also lead to a more "redundant" target instance. These deficiencies can be easily avoided if the relationships across input mappings are carefully considered in the context of source and target schemas. A schema mapping that results from a MapMerge of input mappings is experimentally shown to overcome these deficiencies when compared with a simple union of the input mappings [6].

Finally, the end-to-end mapping for flows of mappings, such as from the first schema $\mathbf{S}_1$ to the last schema $\mathbf{S}_4$ in Figure 4 can be obtained using a new algorithm that combines MapMerge with mapping composition [18] to correlate flows of schema mappings.

3 Background and Related Work

We define the basic concepts and terminology that will be used, as well as discuss prior approaches to schema mapping design.

Schemas and Instances. A *relational schema* $\mathbf{R}$ is a finite sequence $(P_1, \ldots, P_k)$ of relation symbols, each of a fixed arity. An *instance* K over $\mathbf{R}$ is a sequence $(P_1^K, \ldots, P_k^K)$, where each P_i^K is a relation of the same arity as P_i. We shall often write P_i to denote both the relation symbol and the relation P_i^K that interprets it. Here, we assume that all values occurring in relations belong to some fixed infinite set dom of values. A *fact* (or *tuple*) of an instance K over a schema $\mathbf{R}$ is an expression $P(a_1, \ldots, a_m)$ such that P is a relation symbol of $\mathbf{R}$ and $(a_1, \ldots, a_m) \in P^K$. We denote by $\mathrm{adom}(K)$ the *active domain* of an instance K, that is to say, the set of all values from dom occurring in facts of K. A relational schema can be associated with a set of key/foreign key constraints.

Referring back to Figure 4, schema $\mathbf{S}_1$ consists of two relation symbols *Group* and *Works*. The key/foreign key constraint associated with $\mathbf{S}_1$, denoted in the figure via the dashed line, requires that in each instance of $\mathbf{S}_1$, for each *Works* tuple, there must exist a unique *Group* tuple such that they agree on the value of the *gno* attribute. An example of a possible valid instance of $\mathbf{S}_1$ is shown below, where *John* works in group number 123 and the name of group 123 is *CS*.

$$\{\mathrm{Group}(123, \mathrm{CS}), \mathrm{Works}(\mathrm{John}, \mathrm{NY}, \mathrm{Web}, 123)\}$$

In Muse and MapMerge, we use an extension of the relational model that allows for the representation of nested data: the *nested relational* (NR) model [19, 28]. The NR model generalizes the relational model where tuples and relations are modeled as *records* and respectively, *sets* of records. In the NR model however, an element, such as a set of records, may be nested inside another element, such as a record, to form hierarchies. In the following we will use the terms record and tuple, as well as set and relation, interchangeably. To simplify our discussions, we assume that XML schemas are modeled using a single schema root of record type whose elements are all of set type. We also assume strict alternation of set and record types. As an example, consider schema $\mathbf{S}_4$ in Figure 4. This is a nested schema, where each root *CompSci* record contains nested *Staff* and *Projects* sets.

In a nested relational schema, nested sets have associated identifiers called *SetIDs*, also referred to as *grouping functions*. They are *Skolem functions*. In an instance of a nested relational schema, the parameters of each Skolem function serving as a grouping function are instantiated with actual data values, hence providing unique set identifiers

for each nested set in the instance. By convention, we use SKN to denote the SetID name of a nested set N in a schema. For example, the SetID name of the nested set *Projects* in the schema $\mathbf{S}_4$ mentioned above is SKProjects (or SKProjs, or simply SK when there is no ambiguity). We sometimes refer to a nested set N simply as SKN. We assume that every nested set in a schema has a different SetID name.

Schema Mappings. A *schema mapping* or *mapping* is a triple $(\mathbf{S},\mathbf{T},\Sigma)$ where $\mathbf{S}$ is a source schema, $\mathbf{T}$ is a target schema that is disjoint from $\mathbf{S}$, and Σ is a set of constraints. The largest class of constraints we consider is a subset of *second-order tuple generating dependencies (SO tgds)* [18]. One way to express this type of constraints is through the following logical formalism expressed in a query-like notation:

$$\underline{for}\ \mathbf{x}\ \underline{in}\ \bar{S}\ \underline{satisfying}\ B_1(\mathbf{x})\ \underline{exists}\ \mathbf{y}\ \underline{in}\ \bar{T}\ \underline{where}\ B_2(\mathbf{y})\ \underline{and}\ \mathrm{C}(\mathbf{x},\mathbf{y})$$

Here, the symbol $\bar{S}$ represents a vector of relation symbols (possibly repeated), while $\mathbf{x}$ represents the tuple variables that are bound, correspondingly, to these relations. A similar notation applies to the *exists* clause for the vector $\bar{T}$ of target relation symbols and $\mathbf{y}$ of tuple variables that are bound to these relations. The conditions $B_1(\mathbf{x})$ and $B_2(\mathbf{y})$ are conjunctions of equalities over the source and, respectively, target variables. Note that these conditions may equate variables with constants, allowing the definition of *user-defined filters*. The condition $C(\mathbf{x},\mathbf{y})$ is a conjunction of equalities that equate target expressions (e.g., y.A) with either source expressions (e.g., x.B) or *Skolem terms* of the form $F[x_1,\ldots,x_i]$, where F is a function symbol and $x_1,\ldots,x_i$ are source variables or other Skolem terms. Skolem terms are used to relate target expressions across different SO tgds.

Both Muse and MapMerge components of our system use the language of schema mappings specified by SO tgds over nested relational source and target schemas, while the Eirene component focuses on SO tgds without Skolem terms over relational source and target schemas. A constraint of this type may also be called, simply, a *tuple-generating dependency* or *tgd* [17]. In some situations we will refer to a tgd by the equivalent term *GLAV (Global-Local-As-View) constraint*. GLAV constraints have been extensively studied in the context of data exchange and data integration [21, 22]. In cases where $\bar{S}$ and $\bar{T}$ refer to source and, respectively, target relation symbols, then the tgd is referred to as *source-to-target tgds* or *s-t tgds* in short. They are also used in such systems as Clio [16] and HePToX [15].

Two examples of SO tgds that relate schemas $\mathbf{S}_1$ and $\mathbf{S}_2$ in Figure 4 are given below:

(t_1):

for	g *in* Group
exists	d *in* Dept
where	d.dname=g.gname

(t_2):

for	w *in* Works, g *in* Group
satisfying	w.gno = g.gno *and* w.addr="NY"
exists	e *in* Emp
where	e.ename=w.ename *and*
	e.addr=w.addr *and* e.did=$F[g]$

The constraint t_1 is a tgd that states that for every record g in the relation *Group*, there must be a record d in *Dept* where *dname* of d is the same as *gname* of g. Here,

g and d are record variables that range over records in *Group* and, respectively, *Dept*. The second assertion t_2 is an SO tgd that states that for every record g in *Group* and every record w in *Works*, where their *gno*s are identical and the *addr* value of the *Works* record is "NY", there must be a record e in *Emp* where the conditions in the *where* clause are satisfied. Note that "e.*did* = F[g]" states that the *did* value of e is dependent on g through the Skolem function F. Thus, $F[g]$ is a *Skolem term*.

Note that our SO tgds do not allow equalities between or with Skolem terms in the *satisfying* clause. While such equalities may be needed for more general purposes [18], they do not play a role for data exchange and can be eliminated, as observed in [36].

Solutions. Let $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ be a schema mapping. An instance I of $\mathbf{S}$ will be called a *source instance*, and an instance J of $\mathbf{T}$ will be called a target instance.

We say that J is a *solution of I w.r.t. $\mathcal{M}$* if $(I, J) \models \Sigma$, i.e., if (I, J) satisfies every constraint in Σ. In general, there are many possible solutions for a source instance I under a schema mapping $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$.

To illustrate, in line with the previous examples, suppose the source schema consists of the relation symbol $Group$, the target schema consists of the relation symbol $Dept$, and the schema mapping $\mathcal{M}$ is specified by the constraint t_1 given as an example above. Consider the source instance $I = \{\mathrm{Group}(123, \mathrm{CS}), \mathrm{Group}(456, \mathrm{EE})\}$ and the target instances

$$J_1 = \{\mathrm{Dept}(N1, \mathrm{CS}), \mathrm{Dept}(N2, \mathrm{EE})\}$$
$$J_2 = \{\mathrm{Dept}(N1, \mathrm{CS}), \mathrm{Dept}(456, \mathrm{EE})\}$$
$$J_3 = \{\mathrm{Dept}(N1, \mathrm{CS})\}.$$

Both J_1 and J_2 are solutions for I w.r.t. $\mathcal{M}$, but J_3 is not. Observe that the solutions J_1 and J_2 contain values (namely, $N1$ and $N2$) that do not occur in the active domain of the source instance I. Intuitively, these values can be thought of as labeled nulls.

As we shall describe later, a central concept in both Eirene and Muse is the concept of a *data example*. Given a source schema $\mathbf{S}$ and a target schema $\mathbf{T}$ respectively, a *data example* is a pair (I, J) such that I is an instance of $\mathbf{S}$ and J is an instance of $\mathbf{T}$.

Data Exchange, Homomorphisms, and Universal Solutions. *Data exchange* is the following problem: given a schema mapping $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ and a source instance I, construct a solution J for I such that $(I, J) \models \Sigma$. As we just saw, a source instance may have more than one solution with respect to a given GLAV schema mapping. We will be interested in *universal solutions*, which were identified in [17] as the preferred solutions for data exchange purposes. Universal solutions are defined in terms of *homomorphisms*, as follows.

Let I_1 and I_2 be two instances over the same relational schema $\mathbf{R}$. A *homomorphism* $h : I_1 \to I_2$ is a function from $\mathrm{adom}(I_1)$ to $\mathrm{adom}(I_2)$ such that for every fact $P(a_1, \ldots, a_m)$ of I_1, we have that $P(h(a_1), \ldots, h(a_m))$ is a fact of I_2. We write $I_1 \to I_2$ to denote the existence of a homomorphism $h : I_1 \to I_2$. In our previous example, we have that $J_1 \to J_2$ since the function $\{\mathtt{N1} \to \mathtt{N1},\ \mathtt{CS} \to \mathtt{CS},\ \mathtt{N2} \to \mathtt{456},\ \mathtt{EE} \to \mathtt{EE}\}$ is a homomorphism from J_1 to J_2. We say that I_1 and I_2 are *homomorphically equivalent* if there is a homomorphism from I_1 to I_2 and a homomorphism from I_2 to I_1.

Let $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ be a schema mapping and let I be a source instance. A target instance J is a *universal solution* for I w.r.t. $\mathcal{M}$ if the following hold:

1. J is a solution for I w.r.t. $\mathcal{M}$.
2. For every solution J' of I w.r.t. $\mathcal{M}$, there is a homomorphism $h : J \rightarrow J'$ that is constant on $\mathrm{adom}(I) \cap \mathrm{adom}(J)$, that is to say, $h(a) = a$, for every value $a \in \mathrm{adom}(I) \cap \mathrm{adom}(J)$.

Intuitively, universal solutions are the "most general" solutions. Furthermore, in a precise sense, they represent the entire space of solutions (see [17]). For this reason, universal solutions have become the standard semantics for data exchange. Going back to our previous example, note that J_1 is a universal solution for I w.r.t the schema mapping $\mathcal{M}$ specified by the constraint t_1. In contrast, J_2 is not a universal solution for I w.r.t. $\mathcal{M}$, since there is no homomorphism from J_2 to J_1 that is constant on $\mathrm{adom}(I) \cap \mathrm{adom}(J_2)$.

Chase and Canonical Universal Solutions. For GLAV schema mappings $\mathcal{M}$ (and in fact for the wider class of SO tgds), a variant of the *chase procedure* can be used to compute, given a source instance I, a *canonical* universal solution for I w.r.t. $\mathcal{M}$ in time bounded by a polynomial in the size of I (see [17]).

Intuitively, the chase provides a way of populating the target instance J in a minimal way, by adding the tuples that are *required* by Σ. For every instantiation of the *for* clause of a dependency in Σ such that the *satisfying* clause is satisfied but the *exists* and *where* clauses are not, the chase adds corresponding tuples to the target relations. Fresh new values (also called *labeled nulls*) are used to give values for the target attributes for which the dependency does not provide a source expression. Additionally, Skolem terms are instantiated by nulls in a consistent way: a term $F[x_1, \ldots, x_i]$ is replaced by the same null every time $x_1, \ldots, x_i$ are instantiated with the same source tuples. Finally, to obtain a valid target instance, we must chase (if needed) with any target schema constraints. For our earlier example, the target instance J_1 is the result of chasing the source instance I with the constraint t_1. The tuple $\mathrm{Dept}(N_1, \mathrm{CS})$ appears in J_1 since it is asserted by the *exists* clause of t_1, when the *for* clause of t_1 is instantiated with the tuple $\mathrm{Group}(123, \mathrm{CS})$ from I. The *CS* value is propagated from the source *Group* tuple to the target *Dept* tuple because of the equality condition in the *where* clause of t_1. Furthermore, the fresh labeled null N_1 is introduced since t_1 does not provide a source expression for the *did* attribute of the target *Dept* tuple. The tuple $\mathrm{Dept}(N_2, \mathrm{EE})$ in J_1 is obtained in a similar fashion. Since J_1 is the result of chasing I with t_1, we have that J_1 is a *canonical universal solution* for I w.r.t. the schema mapping specified by the constraint t_1.

In practice, mapping systems such as Clio do not necessarily implement the chase with Σ, but generate queries to achieve a similar result [19, 28].

3.1 Prior Schema Mapping Design Systems

A *mapping system* is a graphical user interface that allows a user to visually specify a schema mapping (i.e., data transformation) that translates data from one schema into another. Mapping systems can be categorized as either *function-based* or *relationship-*

based [30]. In function-based mapping systems, schema mappings are specified operationally, as a workflow of operators, which is very similar to the way Extract-Transform-Load (ETL) processes are specified in ETL tools. These systems tend to be highly expressive since the user is allowed to define custom operators. At the same time, these systems are aimed at relatively advanced technical users, as users are required to specify and understand the workflow of operations that constitute the overall semantics of the data transformation at hand.

Relationship-Based Mapping Systems. In contrast, the only type of input required of users of *relationship-based* mapping systems is the specification of high-level relationships between elements (i.e., attributes or sets of attributes) of the source and target schemas. The design methodology of relationship-based mapping systems is shown in Figure 2. The user starts the mapping design process by providing, through a graphical interface, all known *attribute correspondences* (i.e., lines between elements) between elements of a source schema $\mathbf{S}$ (typically shown on the left of the graphical interface) and a target schema $\mathbf{T}$ (typically shown on the right of the graphical interface). An example of a graphical interface typical of a relationship-based mapping system was presented in Figure 1. Sometimes, a schema matching module [29] is used to suggest or derive attribute correspondences.

The source and target schemas, together with the attribute correspondences, form a *visual specification* of the schema mapping intended by the user. Since all that is required as input is the specification of attribute correspondences, this methodology is generally more accessible to non-technical users who may understand their data and the relationships between schema elements.

For commercial mapping systems (e.g., Altova Mapforce [25], Stylus Studio [33], and Microsoft BizTalk Mapper [13]), the visual specification is compiled directly into a runtime executable code (e.g., in XSLT or XQuery or SQL or Java) that implements the intended relationships that are captured by the visual specification. *Data exchange* can be achieved by applying the generated executable code on an instance I of the source schema $\mathbf{S}$ to derive an instance J of the target schema $\mathbf{T}$.

On the other hand, research prototypes such as Clio [16], HePToX [15], and Spicy++ [26] first compile the visual specification into SO tgds or GLAV constraints. To illustrate, consider schemas $\mathbf{S}_1$ and $\mathbf{S}_2$ in Figure 4, and the visual specifications represented by the groups of arrows denoted by t_1 and t_2, respectively. From the visual specification, the declarative schema mappings (t_1) and (t_2) which are expressed as constraints described earlier are first generated. These schema mappings (t_1) and (t_2) can then be compiled into runtime executable code.

One advantage of using schema mappings to specify the relationship between two schemas as an intermediate form is that they are more amenable to the formal study of data exchange and data integration. Many properties of data integration and data exchange, and rigorous studies of operators for manipulating schema mappings have been investigated as a consequence of such logical formalisms [21].

Limitations of Existing Schema Mapping Design Methodologies. Existing schema mapping design systems do not provide the capability for automatically combining pre-existing schema mappings that are independently designed over different and possibly

overlapping parts of a source and target schema. To derive the overall schema mapping between the two schemas, the pre-existing schema mappings are typically "integrated" manually or the overall schema mapping is re-designed from scratch.

The ability to automatically combine different schema mappings that are designed over the same source and target schema allows one to design a schema mapping between two schemas by focussing on smaller components of the schemas. Such a feature is especially useful when the schemas are large and far too complex for the entire mapping to be designed all-at-once. On a similar note, relationship-based mapping systems offer very little support for designing a schema mapping through designing a workflow of (smaller) schema mapping steps. In other words, the procedural methodology offered by function-based mapping systems is sometimes desirable when schemas are large and too complex to be designed in one step.

Finally, even though relationship-based mapping systems tend to be more user-friendly, they cannot be used to generate any arbitrary schema mapping. These systems derive a fixed set of possible schema mappings from a given visual specification, and the derived schema mappings may not correspond to what a user desires. It is typically the case that the user will have to manually tweak or create a schema mapping with the desired semantics.

For the rest of this article, we overview our new framework for designing schema mappings, which overcomes the limitations described earlier. Section 4 describes how data examples can be used to derive and refine a schema mapping interactively. Section 5 describes our MapMerge operator which correlates different schema mappings over the same source and target schema to produce an overall schema mapping which preserves "data associations". In the same section, we also describe how MapMerge together with the composition operator can be leveraged to allow one to design a schema mapping between a source and target schema by designing a workflow of small schema mapping steps. Details of these subsystems can be found in [1–3, 5, 10].

4 Interactive Mapping Design and Refinement via Data Examples

In our new framework, a schema mapping can be designed with existing approaches or interactively with our new approach through the Eirene component system. In Eirene, the user specifies data examples, which are pairs of source instance and expected target instance and the Eirene component system will provide a schema mapping that "fits" the given data examples, if possible. The user can continue to refine various components of a schema mapping through our Muse component system.

4.1 Eirene

The Eirene system supports the design of *GLAV (Global-and-Local-As-View)* schema mappings over a relational source and a relational target schema interactively via data examples. For the rest of this section, we shall use the term *schema mappings* to refer to GLAV schema mappings.

Recall that a *data example* is a pair (I, J) consisting of a source instance and a target instance that conform to a source and target relational schema. The Eirene workflow is

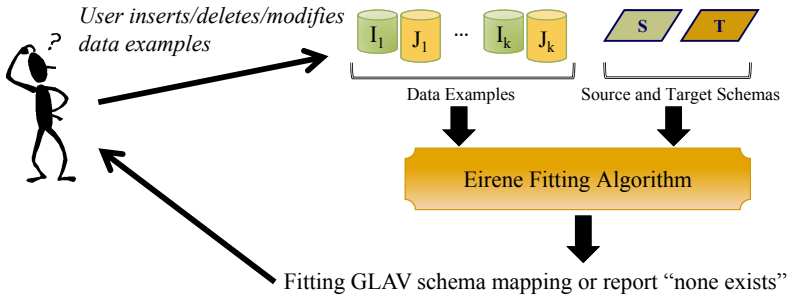


Fig. 5. Workflow for interactive design of schema mappings via data examples

depicted in Figure 5. The interaction between the user and Eirene begins with the user providing an initial finite set $\mathcal{E}$ of data examples, where each data example in $\mathcal{E}$ provides a partial specification of the semantics of the desired schema mapping. Furthermore, the user stipulates that, for each data example (I, J), the target instance J is a *universal solution for* I w.r.t. the desired schema mapping. Intuitively, the target instance J is a "most general" target instance that, together with I, satisfies the specifications of the desired schema mapping. Eirene responds by generating a schema mapping that *fits* the data examples in $\mathcal{E}$ or by reporting that no such schema mapping exists. Here, we say that a schema mapping $\mathcal{M}$ *fits a set* $\mathcal{E}$ of data examples if for every data example $(I, J) \in \mathcal{E}$, the target instance J is a universal solution of the source instance I w.r.t. $\mathcal{M}$. The *refinement process* can continue where the user may modify the data examples in $\mathcal{E}$ to arrive at another finite set $\mathcal{E}'$ of data examples. Again, Eirene responds by testing whether or not there is a schema mapping that fits $\mathcal{E}'$. Eirene reports a fitting schema mapping if one exists. Otherwise, it reports that no fitting schema mappings exist. The process of modifying data examples and generating fitting schema mappings can be repeated until the user is satisfied.

Data examples were considered in [3, 7, 35] as a means to illustrate and help understand schema mappings. In [9], several different notions of "fitting" were explored, including the just defined notion of fitting in terms of universal examples. However, universal solutions, being the most general solutions, are natural as data examples because they contain just the information needed to represent the desired outcome of migrating data from source to target. In particular, they contain no extraneous or overspecified facts, unlike arbitrary solutions. In addition, note that the alternative notion of "fitting" with solutions in place of universal solutions gives rise to a trivial "fitting" problem since, in this case, the schema mapping with an empty set of constraints would "fit" every data example (I, J). In fact, it would be the "most general fitting schema mapping".

Logical Formalism for Schema Mappings. We will often express GLAV constraints using a logical formalism, which is syntactically different, but equivalent to the query-like notation described in Section 3. In this logical formalism, a constraint is a first-order sentence 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 the source schema $\mathbf{S}$, 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 the

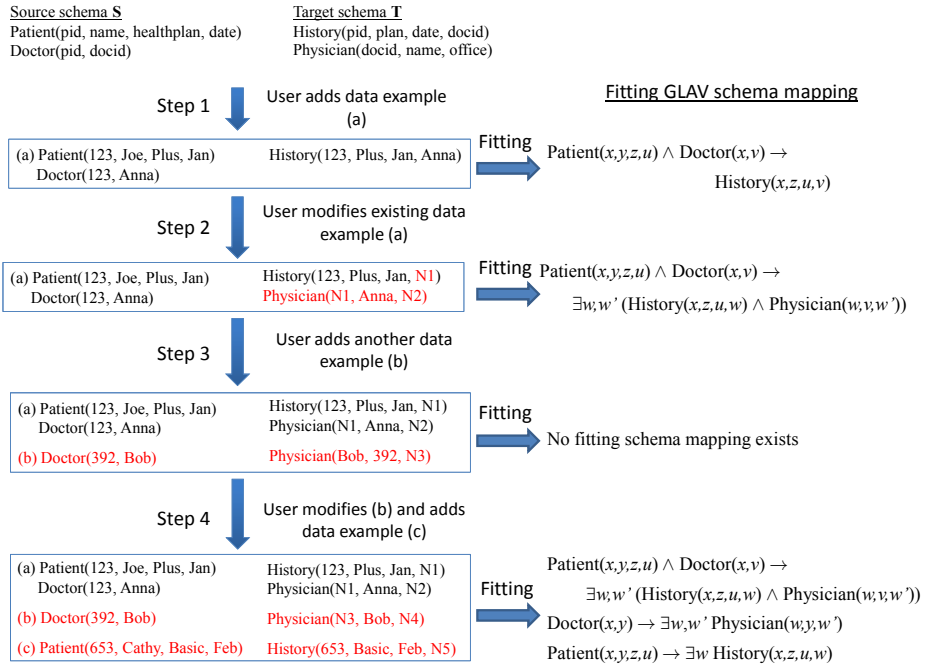


Fig. 6. An example of the workflow in Figure 5

target schema $\mathbf{T}$ with variables from $\mathbf{x}$ and $\mathbf{y}$. By an *atom* over a schema $\mathbf{R}$, we mean a formula $P(x_1, \ldots, x_m)$, where $P \in \mathbf{R}$ and $x_1, \ldots, x_m$ are variables, not necessarily distinct. For notational simplicity, we will often drop the universal quantifiers $\forall \mathbf{x}$ in the front of GLAV constraints. To draw an analogy to the query-like notation introduced in Section 3, the atoms in the $\varphi(\mathbf{x})$ conjunction correspond to the atoms in the *for* clause, while repeated appearances of a variable from $\mathbf{x}$ correspond to equalities specified in the *satisfying* clause. A similar analogy holds between the $\psi(\mathbf{x}, \mathbf{y})$ formula and the *exists* and *where* clauses.

An Example Run of Eirene. Suppose a user wishes to design a schema mapping between the source schema and target schema shown in the top-left corner of Figure 6. The source schema has two relations: Patient and Doctor, and the target schema has two relations: History and Physician.

Step 1. The user adds a single data example, shown in the first box, which essentially states that *Anna* is the doctor of *Joe*, whose health plan is *Plus*, and date-of-visit is *Jan*. In the target relation, there is a single fact that consolidates this information, omitting the patient name. Based on this single data example, Eirene will infer the schema mapping shown on the right of the box. This schema mapping states that whenever a Patient tuple and Doctor tuple agree on the *pid* value (i.e., a natural join between Patient and Doctor), create a target tuple with the *pid*, *healthplan*, *date*, and *docid* values from Patient and Doctor.

Step 2. The user may choose to refine the data example further, perhaps after a realization that there was a typographical error in the data example that is just entered. The modified data example is shown in the second box. For this data example, the source instance remains unchanged, but the user has now modified the target instance to consist of two tuples: a History tuple and a Physician tuple which are "connected" through the value *N1*. Observe that the values *N1* and *N2* in the target instance do not occur among the values of the source instance and they, intuitively, represent unknown and possibly different values. Based on this single data example, our system infers the desired schema mapping shown on the right. The new schema mapping asserts that information from the inner join of Patient and Doctor should be migrated to the target relations, with appropriate nulls to represent unknown and possibly different values.

Step 3. In the third box of Figure 6, the user adds a second data example (b). Eirene now reports that no schema mapping can fit the two data examples (a) and (b). This is because the pattern of data migration in data examples (a) and (b) are inconsistent. According to (b), every Doctor(pid,docid) fact in the source must have a corresponding Physician(docid,pid,office) fact in the target. Observe that the *pid* value is copied to the second column of the corresponding Physician fact. However, this is inconsistent with what (a) states: a Doctor(pid, docid) has a corresponding Physician(_,docid,_) fact in the target, and *docid* gets copied to the second column of the corresponding Physician fact instead.

Step 4. In the fourth box, the user modifies data example (b) and adds a third data example (c). Based on these data examples, Eirene reports the schema mapping shown to the right of the fourth box. Essentially, the schema mapping migrates information from the outer join of Doctor and Patient to the corresponding relations in the target.

Our algorithm that underlies Eirene is shown in Figure 7. It solves the *fitting generation problem* and relies on a *homomorphism extension test* that is a necessary and sufficient condition for the *fitting decision problem*.

Given a source schema $\mathbf{S}$, a target schema $\mathbf{T}$, and a finite set $\mathcal{E}$ of data examples that conform to the schemas, the *GLAV Fitting Decision Problem* asks to tell whether or not there is a GLAV schema mapping $\mathcal{M}$ that fits $\mathcal{E}$. The *GLAV Fitting Generation Problem* asks to construct a GLAV schema mapping $\mathcal{M}$ that fits $\mathcal{E}$, if such a schema mapping exists, or to report that "None exists", otherwise.

The GLAV Fitting Algorithm. As seen in Figure 7, our algorithm has two main steps. Given a finite set $\mathcal{E}$ of data examples, the first step of the algorithm uses the homomorphism extension test to check whether there exists a GLAV schema mapping that fits $\mathcal{E}$. If no such fitting GLAV schema mapping exists, then the algorithm simply reports that none exists. Otherwise, the second step of the algorithm proceeds to construct a GLAV schema mapping that fits the set $\mathcal{E}$.

Homomorphism Extension Test Let (I, J) and (I', J') be two data examples. We say that a homomorphism $h : I \rightarrow I'$ *extends to a homomorphism* $\widehat{h} : J \rightarrow J'$ if for all $a \in adom(I) \cap adom(J)$, we have that $\widehat{h}(a) = h(a)$. The homomorphism extension test checks the following: for every pair of data examples from the given set $\mathcal{E}$, test whether every homomorphism between the source instances of the two examples extends to a homomorphism between the corresponding target instances. If this homomorphism

Algorithm: GLAV Fitting
Input: *A source schema* $\mathbf{S}$*, a target schema* $\mathbf{T}$*, and a finite set* $\mathcal{E}$ *of data examples* $(I_1, J_1) \ldots (I_n, J_n)$ *over* $\mathbf{S}, \mathbf{T}$.
Output: *Either a fitting GLAV schema mapping or 'None exists'*

// Homomorphism Extension Test: Test for existence of a fitting GLAV schema mapping

for all $i, j \leq n$ **do**
  for all homomorphisms $h : I_i \rightarrow I_j$ **do**
    if not(h extends to a homomorphism $\widehat{h} : J_i \rightarrow J_j$) **then**
      fail('None exists')

// Construct a fitting canonical GLAV schema mapping

$\Sigma := \emptyset$
for all $i \leq n$ **do**
  add to Σ the canonical GLAV constraint of (I_i, J_i)
return ($\mathbf{S}$, $\mathbf{T}$, Σ)

Fig. 7. The GLAV Fitting Generation Algorithm

extension test fails, the algorithm immediately reports that no GLAV schema mapping can fit the set $\mathcal{E}$ of data examples.

To illustrate the failure of the homomorphism extension test, we refer back to Figure 6 and the set of data examples resulting after Step 3 of the depicted workflow. The homomorphism $\{\texttt{392} \rightarrow \texttt{123},\ \texttt{Bob} \rightarrow \texttt{Anna}\}$ from the source instance of data example (b) to the source instance of data example (a) cannot be extended to a homomorphism between the corresponding target instances. Any such homomorphism would necessarily map the value `Bob` to `N1`, as well as `392` to `Anna`. Consequently, in this case, the homomorphism extension test fails, and the algorithm terminates. If the homomorphism extension test succeeds, the fitting algorithm proceeds to construct the fitting schema mapping.

Constructing a Fitting Canonical GLAV Schema Mapping In this step, the algorithm proceeds to construct the *canonical GLAV schema mapping* of $\mathcal{E}$. The concept of a canonical GLAV schema mapping is similar to that of a *canonical conjunctive query*. If (I, J) is a data example, then the *canonical GLAV constraint of* (I, J) is the GLAV constraint $\forall \mathbf{x}(q_I(\mathbf{x}) \rightarrow \exists \mathbf{y} q_J(\mathbf{x}, \mathbf{y}))$, where $q_I(\mathbf{x})$ is the conjunction of all facts of I (with each value from the active domain of I replaced by a universally quantified variable from $\mathbf{x}$) and $q_J(\mathbf{x}, \mathbf{y})$ is the conjunction of all facts of J (with each value from $\mathrm{adom}(J) \setminus \mathrm{adom}(I)$ replaced by an existentially quantified variable from $\mathbf{y}$). The *canonical GLAV schema mapping of* $\mathcal{E}$ is the schema mapping $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$, where Σ consists of the canonical GLAV constraints of each data example in $\mathcal{E}$. For example, the canonical GLAV schema mapping for the set of data examples resulting after Step 4 of the workflow in Figure 6 is specified by the three GLAV constraints depicted on the right of the box containing the data examples. Notice that this step takes time linear in the size of the given set $\mathcal{E}$ of data examples.

It is important to point out that the canonical GLAV schema mapping of a given set of data examples need *not* fit this set of examples. In fact, this is what makes the GLAV fitting generation problem interesting and nontrivial. Consider the set $\mathcal{E}$ consisting of the data examples

$$(\{S(a,b)\},\{T(a)\}) \text{ and } (\{S(c,c)\},\{U(c,d)\}).$$

The canonical GLAV schema mapping of $\mathcal{E}$ is specified by the GLAV constraints

$$\forall xy(S(x,y) \rightarrow T(x))$$
$$\forall x(S(x,x) \rightarrow \exists z U(x,z))$$

This schema mapping does not fit $\mathcal{E}$, as the second data example violates the first constraint. Note also that our homomorphism extension test in the first step of the algorithm would detect this: the homomorphism h that maps $S(a,b)$ to $S(c,c)$ does not extend to any target homomorphism from $T(a)$ to $U(c,d)$. Hence, in this case, our algorithm will terminate after the first step and report that "None exists".

Next, we report results that show the correctness of our algorithm, that our algorithm returns the "most general" fitting schema mapping, if a fitting schema mapping exists, that our algorithm is complete for GLAV schema mapping design, the complexity of our algorithm, and our implementation.

Correctness. The correctness of the GLAV fitting generation algorithm is given by the following result.

Theorem 1. *Let $\mathcal{E}$ be a finite set of data examples. The following are equivalent:*

1. *The canonical GLAV schema mapping of $\mathcal{E}$ fits $\mathcal{E}$.*
2. *There is a GLAV schema mapping that fits $\mathcal{E}$.*
3. *(Homomorphism Extension Test) For all $(I,J),(I',J') \in \mathcal{E}$, every homomorphism $h : I \rightarrow I'$ extends to a homomorphism $\widehat{h} : J \rightarrow J'$.*

Theorem 1 shows that the homomorphism extension test is a necessary and sufficient condition for determining whether GLAV schema mapping fitting $\mathcal{E}$ exists. Furthermore, this condition is also a necessary and sufficient condition for determining whether the canonical GLAV schema mapping of $\mathcal{E}$ fits $\mathcal{E}$.

Most General Fitting GLAV Schema Mapping. Given a finite set $\mathcal{E}$ of data examples, there may be many GLAV schema mappings that fit $\mathcal{E}$. If there is a GLAV schema mapping that fits $\mathcal{E}$, we showed that the canonical GLAV schema mapping is the most general GLAV schema mapping that fits $\mathcal{E}$.

Let $\mathcal{M} = (\mathbf{S},\mathbf{T},\Sigma)$ and $\mathcal{M}' = (\mathbf{S},\mathbf{T},\Sigma')$ be two schema mappings over the same source and target schemas. We say that $\mathcal{M}$ is *more general than* $\mathcal{M}'$ if Σ' logically implies Σ, i.e., if for every data example (I,J) such that (I,J) satisfies Σ', we have that (I,J) also satisfies Σ. For example, both $R(x,y) \rightarrow P(x,y)$ and $R(x,x) \rightarrow P(x,x)$ fit the data example $(\{R(a,a)\},\{P(a,a)\})$ with the latter mapping being more general. In this case, the GLAV fitting algorithm will return the latter mapping $R(x,x) \rightarrow P(x,x)$.

This result, along with the correctness of the GLAV fitting algorithm, imply that if a fitting GLAV schema mapping exists for a given set $\mathcal{E}$ of data examples, then our GLAV fitting algorithm returns the most general GLAV schema mapping that fits $\mathcal{E}$. Note that this most general schema mapping is unique up to logical equivalence.

Completeness for Design. Our method of designing schema mappings via data examples is complete for schema-mapping design.

Theorem 2. For every GLAV schema mapping $\mathcal{M}$, there is a finite set of data examples $\mathcal{E}_{\mathcal{M}}$, such that, when given $\mathcal{E}_{\mathcal{M}}$ as input, the GLAV fitting algorithm returns a schema mapping that is logically equivalent to $\mathcal{M}$.

In other words, every GLAV schema mapping can be produced (up to logical equivalence) by our GLAV fitting algorithm with an appropriate set of data examples.

Complexity. The most general schema mapping produced by our GLAV fitting generation algorithm has size linear in the size of the input set of data examples. We showed that this linear bound on the size of the most general schema mapping cannot be improved in general. In contrast, the first step of the GLAV fitting algorithm can be exponential, since the number of homomorphisms between two database instances can be exponential. Hence, the GLAV fitting algorithm runs in exponential time in the worst case. We showed that the GLAV fitting decision problem is complete for the second level Π_2^{p} of the polynomial hierarchy, hence, in all likelihood, it is harder than NP-complete.

Implementation. We implemented our approach as a prototype in Java 6, with IBM DB2 Express-C v9.7 as the underlying database engine, running on a Dual Intel Xeon 3.4GHz Linux workstation with 4GB RAM. Eirene stores data examples in the IBM DB2 database system and implements the homomorphism extension test as a set of DB2 user-defined functions. Intuitively, each function is associated with a data example and it tries to find a witness to the failure of the homomorphism extension.

The high worst-case complexity of the GLAV fitting problem notwithstanding, the experimental results that we have obtained demonstrate the feasibility of interactively designing schema mappings using data examples. In particular, our experiments show that our system achieves very good performance in real-life scenarios. For more details, we refer the interested reader to the experimental evaluation presented in [1].

4.2 Muse

Muse allows a user to refine various aspects of an existing schema mapping specification, based on the choices made by users on a series of data examples that are presented by the system. The Muse workflow is shown in Figure 8. In contrast, the Eirene system derives schema mappings from data examples provided by the user.

The Muse system is largely inspired by the work of Yan *et al.* [35], which was the first to present data examples to users so that users' feedback can be used for refining schema mappings. Like [35], Muse uses data examples to differentiate between alternative mapping specifications and infer the desired mapping semantics based on a user's actions. However, we go significantly beyond the techniques and space of alternative mappings supported by [35].

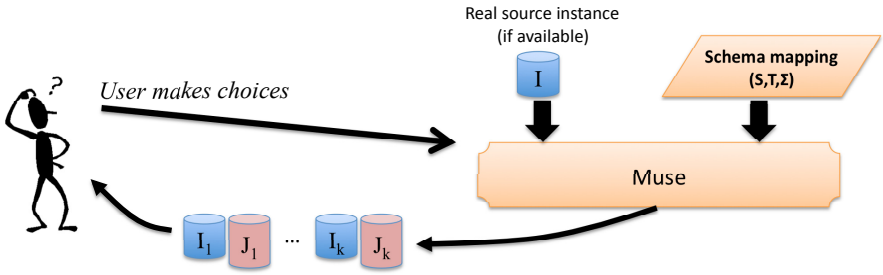
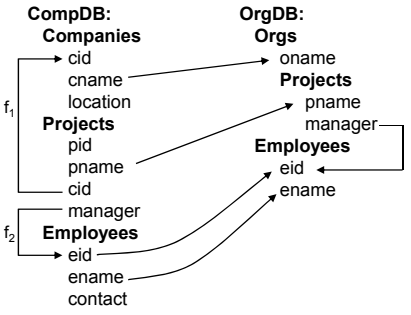


Fig. 8. Interactive refinement of various aspects of schema mappings via data examples

First, Muse is capable of helping a user derive the desired grouping semantics for a mapping specification through choices made on data examples. For instance, to infer whether a user wishes to group projects by a a company's name and location or only by a company's name, Muse will construct a sequence of choice questions with data examples. The selection of data examples made by the user allows Muse to infer the desired grouping semantics. The number of choice questions and the size of each data example are usually small. They correspond roughly to the number schema elements that could be used for grouping and each data example consists of at most two tuples per (nested) relation.

Second, as in [35], Muse helps a user choose among alternative interpretations of an ambiguous mapping. Intuitively, a schema mapping is *ambiguous* if it specifies, in more than one way, how an atomic target schema element (or attribute) is to be obtained. For example, the schema mapping that is generated from the visual specification could be ambiguous because the visual specification may assert (through attribute correspondences) that a project supervisor is a project manager and a project tech-lead at the same time. In other words, it is not clear whether to extract the manager's name or the tech-lead's name (or both) from the source database as the supervisor of a project in the target database and hence the ambiguity. When this happens, the user is asked to select among a small set of data choices to fill in the target instance of a data example that is constructed by Muse. The data example and choices are carefully chosen so that they reflect all possible interpretations of the ambiguous mapping. Furthermore, the user's actions on these choices translate into a unique interpretation. Apart from our ability to handle nested XML-like data, Muse is also different from [35] in that we show all possible interpretations of an ambiguous schema mapping in *one compact representation* (i.e., the data example together with data choices in the target instance of the data example). In contrast, all different target instances are shown to the user in [35]. The discussion of ambigious mappings will be omitted from this article. However, details can be found in [8].

Finally, unlike previous work which relies exclusively on an available source instance to illustrate mappings, Muse can construct its own synthetic data example whenever a meaningful data example cannot be drawn from the actual source instance or when the source instance is unavailable. It is important to note that for a given source instance, schema mappings that are logically inequivalent may produce the same target instance



m_1: for *c* in CompDB.Companies exists *o* in OrgDB.Orgs
where *c*.cname=*o*.oname and
o.Projects = **SKProjs**(*c*.cid,*c*.cname,*c*.location)

m_2: for *c* in CompDB.Companies, *p* in CompDB.Projects,
e in CompDB.Employees
satisfying *p*.cid=*c*.cid and *e*.eid=*p*.manager
exists *o* in OrgDB.Orgs, p_1 in *o*.Projects,
e_1 in OrgDB.Employees
where p_1.manager=e_1.eid and
c.cname=*o*.oname and *e*.eid=e_1.eid and
e.ename=e_1.ename and *p*.pname=p_1.pname and
o.Projects = **SKProjs**(<all attributes of *c*, *p* and *e*>)

m_3: for *e* in CompDB.Employees
exists e_1 in OrgDB.Employees
where *e*.eid = e_1.eid and *e*.ename=e_1.ename

Fig. 9. A mapping scenario

on the given source instance. Muse is able to automatically detect such situations and construct a synthetic source instance that will illustrate differences in all design alternatives as needed. In fact, our experiments justify that this feature of Muse is necessary to help design mappings for some real mapping settings and instances.

Naturally, an advanced user can always choose to tweak or specify the desired schema mapping function directly without using Muse. Muse is useful for cases where such direct manipulation of code is not preferred.

Design of Grouping Functions. Grouping or combining related data together is an essential functionality of many integration systems. We now describe how the grouping design wizard Muse-G of Muse can be used to infer the desired grouping function through a sequence of choices made by the user on data examples.

The Muse-G wizard is always able to infer a grouping function that has the same grouping semantics as the actual grouping function that the user has in mind. As the data examples illustrate the different possibilities of grouping, Muse-G can also be very useful when the user only has a partial understanding of the desired grouping semantics.

In what follows, we overview the basic algorithm behind Muse-G when there are no functional dependencies (FDs) in the source schema. Details of this algorithm and extensions to handle keys (and FDs in general) in the source schema, as well as our experimental results can be found in [8].

Except for topmost-level sets, every nested set in the target schema of mapping generation tools (e.g., [14, 19, 28]) has a *default grouping function*, where the arguments consist of only atomic attributes. For example, there are no grouping functions for *Orgs* and *Employees* in the target schema of Figure 9. However, the default grouping function for *Projects* in m_2 according to [19] is

$$\mathrm{SKProjs}(c.cid, c.cname, c.location, p.pid, p.pname, p.cid, p.manager, e.eid, e.ename, e.contact)$$

This means that *Projects* records are grouped according to the values of all attributes of the *Companies*, *Projects* and *Employees* source records. If SKProjs(*cname*) is the grouping function instead, then *Projects* records are grouped according to *cname* of

Companies records (i.e., *oname* of *Orgs* records). (We write SKProjs(*cname*) instead of SKProjs(c.*cname*) when there is no ambiguity.)

In tools such as Mapforce, Stylus Studio and [14, 19, 28] the arguments of the grouping function have to be explicitly modified or specified. This can be difficult when schemas are large and the number of possible arguments for a grouping function tends to be large as a consequence. Indeed, if there are n possible attributes to group by, then there are in fact 2^n choices of grouping functions. Furthermore, it may not be obvious to a user, what the n possible grouping attributes are (see [19, 28]).

Muse-G takes as input a schema mapping ($\mathbf{S}$, $\mathbf{T}$, Σ). The user can choose to design any grouping function that occurs in Σ. We assume that there is a real source instance I from which Muse-G can draw real data examples whenever possible, and show how Muse-G constructs its own examples otherwise. To illustrate our algorithm, we use the schema mapping ($\mathbf{S}$, $\mathbf{T}$, $\{m_2\}$), where $\mathbf{S}$, $\mathbf{T}$ and m_2 are the source and target schemas and respectively, mapping, of Figure 9.

Step 1. The first step is to determine an order to the set of grouping functions that the user wishes to (re)design in a mapping in Σ by performing a breadth-first traversal of $\mathbf{T}$ starting from the root. This yields, for our example, the order *Orgs*, *Employees*, and *Projects*. Since *Orgs* and *Employees* are top-level sets without grouping functions, Muse-G will only prompt the design of grouping functions for *Projects* (i.e., SKProjs) in m_2.

Step 2. Next, we determine the set *poss*(m_2, SKProjs) of all possible arguments for SKProjs according to m_2. According to the schema of *OrgDB*, a *Projects* SetID is nested inside an *Orgs* tuple. According to the *for* clause of m_2, the existence of an *Orgs* tuple is dependent on the existence of a *Companies* tuple, an *Employees* tuple, and a *Projects* tuple which agrees with the *Companies* and *Employees* tuples on the values of *pid* and *manager*, respectively.

This means that *poss*(m_2,SKProjs) consists of the set of attributes in the *Companies*, *Projects* and *Employees* records, which is {*cid*, *cname*, *location*, *pid*, *pname*, *pid*, *manager*, *eid*, *ename*, *contact*}. Note that the sets *poss*(m,SK) are in fact identical for all nested sets SK occurring in m. In other mapping formalisms, however, they may be different (see [19] for details). However, to simplify our subsequent discussion, we will assume that *poss*(m_2,SKProjs)={*cid*,*cname*,*location*}.

Step 3. Suppose the user has SKProjs(Z) in mind, where $Z \subseteq$ *poss*(m_2, SKProjs). In what follows, we show how Muse-G proceeds to construct data examples to present choices to the user in order to infer the desired grouping function.

Construct Data Examples. To determine whether or not an attribute A from *poss*(m_2, SKProjs) is to be included in the grouping function of SKProjs, Muse-G carefully constructs a small source instance I_e such that two differentiating target instances are obtained: regardless of what the rest of the grouping attributes might be, one is the result of including the attribute A as part of SKProjs in m_2, and the other omits it.

Suppose the attribute under consideration is *cid*. An example source instance I_e with two tuples, as shown below, will be constructed:

$$I_e : \{\textit{Companies}(c_1, n_1, l_1), \textit{Projects}(p_1, pn_1, c_1, e_1), \textit{Employees}(e_1, en_1, cn_1),$$
$$\textit{Companies}(c_2, n_1, l_1), \textit{Projects}(p_2, pn_2, c_2, e_2), \textit{Employees}(e_2, en_2, cn_2)\}$$

Observe that each relation in I_e has two tuples. Furthermore, every attribute value of every tuple is distinct, except for *cname* and *location* values of *Companies* tuples. The reason for this is so that the target instances generated by m_2 with SKProjs(*cid*,$\mathbf{y}$), where $\mathbf{y} \subseteq \{$*cname*,*location*$\}$, versus m_2 with SKProjs($\mathbf{y}$) will be non-isomorphic. Indeed, the former target instance will contain two distinct *Projects* sets, while the latter consists of only one *Projects* set.

To obtain a real source instance, Muse-G generates the following query that will be executed against the actual source instance, if available, to retrieve real tuples for the example instance I_e.

$$Q^{I_e} : \textit{Companies}(c_1, n_1, l_1) \wedge \textit{Companies}(c_2, n_1, l_1) \wedge \\ \textit{Projects}(p_1, pn_1, c_1, e_1) \wedge \textit{Projects}(p_2, pn_2, c_2, e_2) \wedge \\ \textit{Employees}(e_1, en_1, cn_1) \wedge \textit{Employees}(e_2, en_2, cn_2) \wedge c_1 \neq c_2$$

All variables of Q^{I_e} are universally-quantified. The two *Companies* tuples must disagree on *cid* (the probed attribute) and agree on *cname* and *location* as explained earlier.

If $Q^{I_e}(I)$ returns an empty result, Muse-G will present the user with the synthetic instance I_e, shown earlier. Alternatively, a "semi-real" I_e may also be constructed by putting together various real values drawn from I (e.g., use *cid*, *cname* and *location* values drawn from the corresponding columns of the *Companies* relation to create a *Companies* tuple in I_e, regardless of whether these values participate in a real *Companies* tuple). However, this may lead to combinations that are misleading to the user. On the other hand, if $Q^{I_e}(I)$ returns a non-empty result, Muse-G constructs a real example based on the returned values. A possible real example constructed in this way is shown in Figure 10(a), where each tuple in *Companies*, *Projects* and *Employees* exists in I.

Next, Muse-G obtains two differentiating target instances shown in Scenarios 1 and 2 in Figure 10(a), by chasing I_e with mappings d_1 and respectively, d_2. Here, d_1 and d_2 are identical to m_2 except they have SKProjs(*cid*) and respectively, SKProjs() as grouping functions for *Projects*. Now, Muse-G asks the user "which target instance looks correct"?

Note that the instance I_e has been carefully crafted so that the chase of I_e with d_1 is isomorphic to the chase of I_e with d_1', where d_1' is a mapping obtained from m_2 by replacing SKProjs with SKProjs($\{$*cid*$\} \cup Y$), where $Y \subseteq \{$*cname*, *location*$\}$. Since *cname* and *location* values are identical for the two *Comp* tuples in I_e, the mapping d_1 has the same effect as d_1' on I_e. Similarly, d_2 has the same effect as d_2' on I_e, where d_2' is obtained from d_2 by replacing SKProjs with SKProjs(Y). Hence, based on the user's choice of Scenario 1 or 2, Muse-G correctly determines whether *cid* is part of the user's desired grouping function. So with one question, we either eliminate all mappings using *cid* (not only SKProjs(*cid*), but SKProjs(*cid*, *cname*), SKProjs(*cid*, *location*), and SKProjs(*cid*, *cname*, *location*)), or we eliminate all mappings that do not use *cid* in the skolem function for *Projects*.

Continuing with our example, suppose the user has the grouping function SKProjs(*cname*) in mind. She would select Scenario 2 in Figure 10(a). We now repeat the process for the other attributes *cname* and *location*. Figure 10(b) shows the example source instance and the two scenarios obtained by considering the attribute *cname*. The two source *Companies* tuples must differ on the values of *cname* and agree on the values of *location*. Note that the *cid* values of the two *Companies* tuples are not required

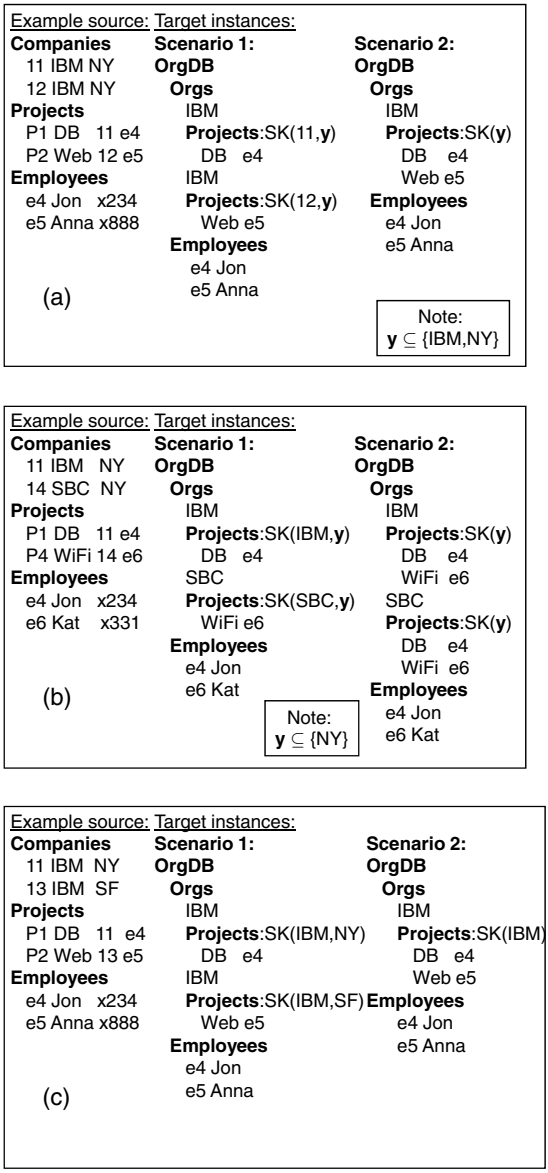


Fig. 10. Probing on (a) *cid*, (b) *cname*, and (c) *location* when the user has SKProjs(*cname*) in mind

to be identical, since *cid* is not an argument of SKProjs. The user will pick Scenario 1 in Figure 10(b), since she wants to group *Projects* by *cname*, and Muse-G infers that *cname* is an argument to SKProjs. Figure 10(c) shows the data examples that are presented to the user when the attribute *location* is under consideration. The user will pick Scenario 2. Since *cname* is part of the grouping, the *Companies* tuples must agree on the *cname* values, otherwise, Muse-G would not be able to infer whether *location* is part of the groping from the user's choice in Figure 10(c). At this point, Muse-G concludes and returns SKProjs(*cname*).

Recall that we have assumed above that *poss*(m_2, SKProjs) is {*cid*,*cname*,*location*} for simplicity, when in fact it consists of all attributes of *Companies*, *Projects* and *Employees* records. In this case, Muse-G concludes only after subsequently probing all the attributes of *Projects* and *Employees* records (the user will choose Scenario 2 in each case). Note also that it is conceivable for Muse-G to generate homomorphically equivalent target instances (i.e., target instances with a homormophisms into each other) for Scenarios 1 and 2 (e.g., Figure 10(b)). However, it is always possible for the user to distinguish between such instances, as they are non-isomorphic.

Muse-G infers the desired grouping function by presenting the user a *small* number of choice questions, where each choice question consists of a *small* source instance with two target instances that correspond to the two possible choices in this question.

Small Number of Choices, Small Data Examples. For each nested set SK in a mapping m, there are 2^n different grouping functions where $n = |poss(m, \text{SK})|$. However, Muse-G determines the desired grouping function by asking the user only $|poss(m, \text{SK})|$ questions. In fact, if there is at most one key per nested set, then Muse-G performs a careful reordering of the questions posed to the user. The questions pertaining to the attributes in the key are asked first. In general, using this strategy, at most n questions are needed to infer the desired grouping function. If the user decides to include the key attributes in the grouping function, then the number of questions is equal to the number of key attributes. It is also important to note that all real source schemas that we have encountered in our experimental evaluation fall into this category.

Furthermore, for each choice, Muse-G constructs a small source example. The size of the source example is twice the number of "$x \in X$" clauses in *for* clauses of m. This typically means that there are at most two tuples in each nested set.

We refer the interested reader to [2] for a report on our experience with Muse on publicly available mapping scenarios.

5 Modular Design of Schema Mappings

5.1 Overview

As outlined in Section 2, in our Divide-Design-Merge methodology, the user can choose to design a schema mapping by focusing on designing smaller and easier to understand mappings, using data examples as much as possible. In the previous section, we have presented our techniques for designing and refining schema mappings via data examples. However, simply taking the independently designed schema mapping components and using them as the specification for the global schema mapping may not achieve the

desired semantics. This may lead, as it will be explained later, to problems such as data redundancies and loss of data associations. Hence, the design workflow is not complete without a mechanism for correlating the set of independent schema mappings resulting after the previous phase into a meaningful global schema mapping (see Figure 3). This is the role of the MapMerge schema mapping operator, presented in this section. This operator allows for the modular construction of complex and larger schema mappings from multiple "smaller" schema mappings between the same source and target schemas into an arguably better overall schema mapping.

Since the mappings given as input to MapMerge can be as simple as individual attribute correspondences, MapMerge supersedes previous mapping generation algorithms such as the ones in Clio [16]. In addition, as we will show later, MapMerge can be used in conjunction with the schema mapping composition operator [18, 23, 27] to correlate flows of schema mappings in a meaningful way.

5.2 Motivating Example

To illustrate the ideas behind MapMerge, consider first a mapping scenario between the schemas $\mathbf{S_1}$ and $\mathbf{S_2}$ shown in the left part of Figure 4. The goal is data restructuring from two source relations, *Group* and *Works*, to three target relations, *Emp*, *Dept*, and *Proj*. In this example, *Group* (similar to *Dept*) represents groups of scientists sharing a common area (e.g., a database group, a CS group, etc.) The dotted arrows represent foreign key constraints in the schemas.

Independent Mappings. Assume the existence of the following (independent) schema mappings from $\mathbf{S_1}$ to $\mathbf{S_2}$. The first mapping is the constraint t_1 in Figure 11(a), and corresponds to the arrow t_1 in Figure 4. This constraint requires every tuple in *Group* to be mapped to a tuple in *Dept* such that the group name (*gname*) becomes department name (*dname*). The second mapping is more complex and corresponds to the group of arrows t_2 in Figure 4. This constraint involves a custom filter condition; every pair of joining tuples of *Works* and *Group* for which the *addr* value is "NY" must be mapped into two tuples of *Emp* and *Dept*, sharing the same *did* value, and with corresponding *ename*, *addr* and *dname* values. (Note that *did* is a target-specific field that must exist and plays the role of key / foreign key). Intuitively, t_2 illustrates a pre-existing mapping that a user may have spent time in the past to create, possibly using the techniques based on data examples from Section 4. Finally, the third constraint in Figure 11(a) corresponds to the arrow t_3 and maps *pname* from *Works* to *Proj*. This is an example of a correspondence that is introduced by a user after loading t_1 and the pre-existing mapping t_2 into the mapping tool.

The goal of the system is now to (re)generate a "good" overall schema mapping from $\mathbf{S_1}$ to $\mathbf{S_2}$ based on its input mappings. We note first that the input mappings, when considered in isolation, do not generate an ideal target instance.

Indeed, consider the source instance I in Figure 12. The target instance that is obtained by minimally enforcing the constraints $\{t_1, t_2, t_3\}$ is the instance J_1 also shown in the figure. The first *Dept* tuple is obtained by applying t_1 on the *Group* tuple $(123, CS)$. There, $D1$ represents some did value that must be associated with CS in this tuple. Similarly, the *Proj* tuple, with some unspecified value B for $budget$ and a did

Input mappings from $\mathbf{S_1}$ to $\mathbf{S_2}$:

(t_1) *for* g in Group *exists* d in Dept
where d.dname = g.gname

(t_2) *for* w in Works, g in Group
satisfying w.gno = g.gno, w.addr = "NY"
exists e in Emp, d in Dept
where e.did = d.did,
e.ename = w.ename, e.addr = w.addr,
d.dname = g.gname

(t_3) *for* w in Works *exists* p in Proj
where p.pname = w.pname

(a)

Output of MapMerge($\mathbf{S_1}$, $\mathbf{S_2}$, {t_1, t_2, t_3}):

for g in Group *exists* d in Dept
where d.dname = g.gname, **d.did = F[g]**

for w in Works, g in Group
satisfying w.gno = g.gno, w.addr = "NY"
exists e in Emp
where e.ename = w.ename, e.addr = w.addr,
e.did = F[g]

for w in Works, g in Group
satisfying w.gno = g.gno, w.addr = "NY"
exists p in Proj
where p.pname = w.pname, p.budget = H_1[w],
p.did = F[g]

(b)

Fig. 11. (a) Schema mappings from $\mathbf{S_1}$ to $\mathbf{S_2}$ in the scenario of Figure 4. (b) Output of MapMerge.

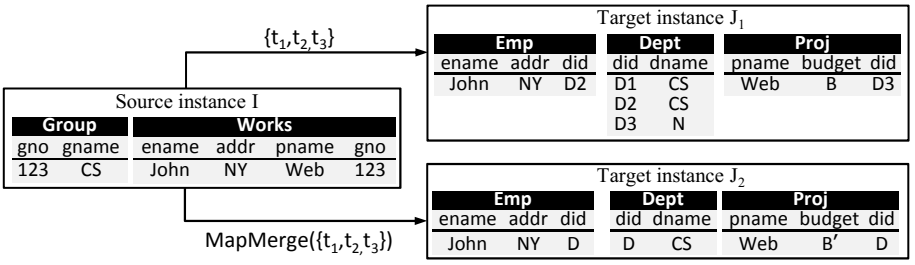


Fig. 12. An instance of $\mathbf{S}_1$ and two instances of $\mathbf{S}_2$

value of $D3$ is obtained via t_3. The *Emp* tuple together with the second *Dept* tuple are obtained based on t_2. As required by t_2, these tuples are linked via the same did value $D2$. Finally, to obtain a target instance that satisfies all the foreign key constraints, we must also have a third tuple in *Dept* that includes $D3$ together with some unspecified department name N.

Since the three mapping constraints are not correlated, the three did values ($D1$, $D2$, $D3$) are distinct. (There is no requirement that they must be equal.) As a result, the target instance J_1 exhibits the typical problems that arise when uncorrelated mappings are used to transform data: (1) *duplication of data* (e.g., multiple *Dept* tuples for CS with different did values), and (2) *loss of associations* where tuples are not linked correctly to each other (e.g., we have lost the association between project name Web and department name CS that existed in the source).

Correlated Mappings via MapMerge. Consider now the schema mappings that are shown in Figure 11(b) and that are the result of MapMerge applied on $\{t_1, t_2, t_3\}$. The notable difference from the input mappings is that all mappings consistently use the same expression, namely the Skolem term $F[g]$ where g denotes a *Group* tuple, to give values for the did field. The first mapping is the same as t_1 but makes explicit the fact that did is $F[g]$. This mapping creates a unique *Dept* tuple for each distinct *Group* tuple. The second mapping is (almost) like t_2 with the additional use of the same Skolem term $F[g]$. Moreover, it also drops the existence requirement for *Dept* (since this is now implied by the first mapping). Finally, the third mapping differs from t_3 by incorporating a join with *Group* before it can actually use the Skolem term $F[g]$. Furthermore, it inherits the filter on the $addr$ field, which applies to all such *Works* tuples according to t_2. As an additional artifact of MapMerge, it also includes a Skolem term $H_1[w]$ that assigns values to the $budget$ attribute, which was initially left unspecified. The target instance that is obtained by applying the result of MapMerge is the instance J_2 shown in Figure 12. The data associations that exist in the source are now correctly preserved in the target. For example, Web is linked to the CS tuple (via D) and also $John$ is linked to the CS tuple (via the same D). Furthermore, there is no duplication of *Dept* tuples.

Flows of Mappings. Taking the idea of mapping reuse and modularity one step further, an even more compelling use case for MapMerge in conjunction with mapping composition [18, 23, 27], is the *flow-of-mappings* scenario [4]. The key idea here is that to design a data transformation from the source to the target, one can decompose the process, in line with the Divide-Design-Merge approach, into several simpler stages, where each stage maps from or into some intermediate, possibly simpler schema. Moreover, the simpler mappings and schemas play the role of reusable components that can be applied to build other flows. Such abstraction is directly motivated by the development of real-life, large-scale ETL flows such as those typically developed with IBM Information Server (Datastage), Oracle Warehouse Builder and others.

To illustrate, suppose the goal is to transform data from the schema $\mathbf{S}_1$ to the nested schema $\mathbf{S_4}$ of Figure 4, where *Staff* and *Projects* information are grouped under *CompSci*. The mapping or ETL designer, following the divide-and-merge methodology, may find it easier to first construct the mapping between $\mathbf{S}_1$ and $\mathbf{S}_2$ (it may also be that this mapping may have been derived in a prior design). Furthermore, the schema $\mathbf{S_2}$ is a normalized representation of the data, where *Dept*, *Emp* and *Proj* correspond directly to the main concepts (or types of data) that are being manipulated. Based on this schema, the designer can then produce a mapping m_{CS} from *Dept* to a schema $\mathbf{S}_3$ containing a more specialized object *CSDept*, by applying some customized filter condition (e.g., based on the name of the department). The next step is to create the mapping m from *CSDept* to the target schema $\mathbf{S}_4$. Other independent mappings are similarly defined for *Emp* and *Proj* (see m_1 and m_2).

Once these individual mappings are established, the same problem of correlating the mappings arises. In particular, one has to correlate $m_{\mathsf{CS}} \circ m$, which is the result of applying mapping composition to m_{CS} and m, with the mappings m_1 for *Emp* and m_2 for *Proj*. This correlation will ensure that all employees and projects of computer

science departments will be correctly mapped under their correct departments, in the target schema.

In this example, composition itself gives another source of mappings to be correlated by MapMerge. While similar with composition in that it is an operator on schema mappings, MapMerge is fundamentally different in that it correlates mappings that share the same source schema and the same target schema. In contrast, composition takes two sequential mappings where the target of the first mapping is the source of the second mapping. Nevertheless, the two operators are complementary and together they can play a fundamental role in building data flows. In Section 5.4 we will give an overview of an algorithm that can be used to correlate flows of mappings.

5.3 Correlating Mappings: Key Ideas

How do we achieve the systematic and, moreover, *correct* construction of correlated mappings? After all, we do not want arbitrary correlations between mappings, but rather only to the extent that the *natural* data associations in the source are preserved and no extra associations are introduced.

There are two key ideas behind MapMerge. The first idea is to exploit the structure and the constraints in the schemas in order to define what natural associations are (for the purpose of the algorithm). Two data elements are considered associated if they are in the same tuple or in two different tuples that are linked via constraints. This idea has been used before in Clio [28], and provides the first (conceptual) step towards MapMerge. For our example, the input mapping t_3 in Figure 11(a) is equivalent, in the presence of the source and target constraints, to the following enriched mapping:

t_3': *for* w *in* Works, g *in* Group *satisfying* w.gno = g.gno
exists p *in* Proj, d *in* Dept *where* p.pname = w.pname *and* p.did = d.did

Intuitively, if we have a w tuple in *Works*, we also have a joining tuple g in *Group*, since gno is a foreign key from *Works* to *Group*. Similarly, a tuple p in *Proj* implies the existence of a joining tuple in *Dept*, since did is a foreign key from *Proj* to *Dept*.

Formally, the above rewriting from t_3 to t_3' is captured by the well-known chase procedure [11, 24]. The chase is a convenient tool to group together, syntactically, elements of the schema that are associated. The chase by itself, however, does not change the semantics of the mapping. In particular, the above t_3' does not include any additional mapping behavior from *Group* to *Dept*.

The second key idea behind MapMerge is that of *reusing* or borrowing mapping behavior from a more general mapping to a more specific mapping. This is a heuristic that changes the semantics of the entire schema mapping and produces an arguably better one, with consolidated semantics.

To illustrate, consider the first mapping constraint in Figure 11(b). This constraint (obtained by skolemizing the input t_1) specifies a general mapping behavior from *Group* to *Dept*. In particular, it specifies how to create $dname$ and did from the input record. On the other hand, the above t_3' can be seen as a more *specific* mapping from a *subset* of *Group* (i.e., those groups that have associated *Works* tuples) to a *subset* of *Dept* (i.e., those departments that have associated *Proj* tuples). At the same time, t_3' does not specify any concrete mapping for the $dname$ and did fields of *Dept*. We can then

borrow the mapping behavior that is already specified by the more general mapping. Thus, t'_3 can be enriched to:

t''_3: *for* w *in* Works, g *in* Group *satisfying* w.gno = g.gno
exists p *in* Proj, d *in* Dept
where p.pname = w.pname *and* p.did = d.did *and*
d.dname = g.gname *and* d.did = $F[g]$ *and* p.did = $F[g]$

where two of the last three equalities represent the "borrowed" behavior, while the last equality is obtained automatically by transitivity. The other borrowed behavior that we will add to t''_3 is the user-defined filter on *addr*. This filter already applies, according to t_2, to all tuples in *Works* that join with *Group* tuples, and are mapped to *Emp* and *Dept* tuples. The resulting constraint t'''_3 has the following form:

t'''_3: *for* w *in* Works, g *in* Group *satisfying* w.gno = g.gno *and* w.addr = "NY"
exists p *in* Proj, d *in* Dept
where p.pname = w.pname *and* p.did = d.did *and*
d.dname = g.gname *and* d.did = $F[g]$ *and* p.did = $F[g]$

Finally, we can drop the existence of d in *Dept* with the two conditions for $dname$ and did, since this is repeated behavior that is already captured by the more general mapping from *Group* to *Dept*. The resulting constraint is identical[2] to the third constraint in Figure 11(b), now correlated with the first one via $F[g]$. A similar explanation applies for the second constraint in Figure 11(b).

The MapMerge Algorithm. MapMerge takes as input a set $\{(\mathbf{S}, \mathbf{T}, \Sigma_1), ..., (\mathbf{S}, \mathbf{T}, \Sigma_n)\}$ of schema mappings over the same source and target schemas, which is equivalent to taking a single schema mapping ($\mathbf{S}$, $\mathbf{T}$, $\Sigma_1 \cup ... \cup \Sigma_n$) as input. The algorithm is divided into four phases. The first phase decomposes each input mapping assertion into basic components that are, intuitively, easier to merge. In Phase 2, we apply the chase algorithm to compute associations (which we call *tableaux*), from the source and target schemas, as well as from the source and target assertions of the input mappings. The latter type of tableaux is necessary to support user defined joins that may not follow foreign key constraints. By pairing source and target tableaux, we obtain all the possible *skeletons* of mappings. The actual work of constructing correlated mappings takes place in Phase 3, where for each skeleton, we take the union of all the basic components generated in Phase 1 that "match" the skeleton. Phase 4 is a simplification phase that also flags conflicts that may arise and that need to be addressed by the user. These conflicts occur when multiple mappings that map to the same portion of the target schema contribute with different, irreconcilable behaviors. For a complete presentation of the MapMerge algorithm, we refer the interested reader to [5].

Evaluation. To evaluate the quality of the data generated based on MapMerge, we introduced a measure that captures the similarity between a source and target instance

[2] Modulo the absence of $H_1[w]$, which is introduced to ensure that no target attributes are left unassigned.

by measuring the amount of data associations that are preserved by the transformation from the source to the target instance. We used this similarity measure in our experiments, on a mix of real-life and synthetic mapping scenarios, to show that the mappings derived by MapMerge are better than the input mappings. Our experiemental results are presented in [5].

5.4 Correlating Flows of Schema Mappings with MapMerge and Composition

As discussed in the introduction, we can bring modular design of mappings beyond sets of parallel mappings between the same pair of schemas, towards assembling general flows of mappings. To generate meaningful end-to-end transformation specifications for such flows, we have to bring along into the picture the sequential mapping composition operator [18]. This operator can be used to obtain end-to-end mappings from chains of successive mappings. In contrast, MapMerge assembles sets of "parallel" mappings. These two operators can be leveraged in conjunction to correlate flows of mappings.

Recall the example of the flow of mappings in Figure 4. The individual mappings can be assembled into an end-to-end mapping from the schema $\mathbf{S}_1$ to the schema $\mathbf{S}_4$ through repeated applications of the MapMerge and composition operators. To exemplify, the specialized mapping for *Dept* records between $\mathbf{S}_2$ and $\mathbf{S}_4$ is a result of composing the m_{CS} and m mappings. Furthermore, the right correlations among the *Dept*, *Emp*, and *Proj* records that are migrated into $\mathbf{S}_4$ can be achieved by applying MapMerge on m_1, m_2, and the result $m_{\mathsf{CS}} \circ m$ of the previous composition.

Flow Correlation Algorithm. We provide here an overview of our flow correlation algorithm. The complete details of this algorithm can be found in [5]. A flow of mappings can be modeled as a multigraph whose nodes are the schemas and whose edges are the mappings between the schemas. Recall that a mapping consists of a pair of source and target schemas as well as a set of constraints specified by SO tgds. In this algorithm, a mapping between a source and a target schema is either part of the input, or a consequence of applying MapMerge or mapping composition. Our algorithm assumes that the graph of mappings is acyclic. In addition, for the purposes of this algorithm, we assume that the MapMerge operator does not lead to outstanding residual equality constraints. Integrating such constraints with the mapping composition operator is a problem we plan to investigate in future work.

The flow correlation algorithm, which is shown in Figure 13, proceeds through alternative phases of applying the MapMerge and mapping composition operators, and terminates when no further progress can be made. In a MapMerge phase, the multigraph modeling the flow is essentially transformed into a regular graph. For any pair of schemas $\mathbf{S}_i, \mathbf{S}_j$, the set of mappings $\mathcal{M}_{ij}$ going from $\mathbf{S}_i$ to $\mathbf{S}_j$ is replaced by the result of applying MapMerge on $\mathcal{M}_{ij}$. In a mapping composition phase, for any distinct schemas $\mathbf{S}_i, \mathbf{S}_j, \mathbf{S}_k$ in the flow such that $\mathbf{M}_1$ is a mapping from $\mathbf{S}_i$ to $\mathbf{S}_j$ and $\mathbf{M}_2$ is a mapping from $\mathbf{S}_j$ to $\mathbf{S}_k$, the result $\mathbf{M} = \mathbf{M}_1 \circ \mathbf{M}_2$ of composing $\mathbf{M}_1$ and $\mathbf{M}_2$ is added to the flow. We use here the mapping composition algorithm in [18], since it applies to schema mappings specified by SO tgds.

Our correlation algorithm keeps track, via the set $\mathcal{C}$, of the mappings being added to the flow in the composition phase. As a result, a mapping is not re-added to the flow

Algorithm CorrelateFlow($\mathcal{M}$)
Input: A set of schema mappings $\mathcal{M}$.
Output: The set of schema mappings $\mathcal{M}$ after correlation.

Let $\mathcal{S}$ be the set of schemas that are either source or target schemas for the mappings in $\mathcal{M}$.
Initialize $\mathcal{C} = \emptyset$
Repeat
    [Phase 1] MapMerge
        For every pair $(\mathbf{S}_i, \mathbf{S}_j)$ of distinct schemas in $\mathcal{S}$
            Let $\mathcal{M}_{ij}$ be the set of mappings from $\mathbf{S}_i$ to $\mathbf{S}_j$ in $\mathcal{M}$.
            Remove the mappings in $\mathcal{M}_{ij}$ from $\mathcal{M}$
            Add MapMerge($\mathcal{M}_{ij}$) to $\mathcal{M}$
    [Phase 2] Composition
        Initialize $\mathcal{N} = \emptyset$
        For every triple $(\mathbf{S}_i, \mathbf{S}_j, \mathbf{S}_k)$ of distinct schemas in $\mathcal{S}$
            where there exist in $\mathcal{M}$ a mapping $\mathbf{M}_1$ from $\mathbf{S}_i$ to $\mathbf{S}_j$ and
                a mapping $\mathbf{M}_2$ from $\mathbf{S}_j$ to $\mathbf{S}_k$
            Let $\mathbf{M} = \mathbf{M}_1 \circ \mathbf{M}_2$
            If $\mathbf{M} \notin \mathcal{C}$ (this composition was not considered before), add $\mathbf{M}$ to $\mathcal{N}$
        Add the mappings in $\mathcal{N}$ to $\mathcal{M}$, and to $\mathcal{C}$
Until $\mathcal{N}$ is empty.
Return $\mathcal{M}$.

Fig. 13. The mapping flow correlation algorithm

if the result of composing the same mappings was computed and added to the flow previously in the execution of the algorithm. The algorithm terminates when no new mappings can be added to the flow in the composition phase, and returns the correlated flow of mappings $\mathcal{M}$. After executing this algorithm, the flow of mappings will contain at most one mapping between each pair of schemas (with each mapping typically being a set of correlated formulas).

6 Conclusion

This article presents a new framework for designing schema mappings between large schemas. This new framework allows a user to divide-and-conquer the design of large schema mappings by designing the schema mappings between smaller portions of the participating schemas. These smaller schema mappings can be designed independently of the rest through the specification of data examples or through the use of traditional schema mapping design tools. Such individually designed schema mappings can then be correlated and merged into one that better represents the associations in the source data, whenever possible.

Acknowledgements. This article presents an overview of work that has been investigated in the dissertation of Bogdan Alexe, whose academic genealogy can be traced back to Peter Buneman.

The authors are grateful to Balder ten Cate, Laura Chiticariu, Mauricio A. Hernández, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa for their collaboration on various aspects of this work. This work is supported by NSF Grant IIS-0905276 and a Google Faculty Award. Part of this work was done while Tan was at IBM Research - Almaden.

References

1. Alexe, B., ten Cate, B., Kolaitis, P.G., Tan, W.C.: Designing and Refining Schema Mappings via Data Examples. In: SIGMOD Conference (2011)
2. Alexe, B., Chiticariu, L., Miller, R.J., Pepper, D., Tan, W.C.: Muse: a System for Understanding and Designing Mappings. In: SIGMOD Conference, pp. 1281–1284 (2008)
3. Alexe, B., Chiticariu, L., Miller, R.J., Tan, W.C.: Muse: Mapping Understanding and deSign by Example. In: ICDE, pp. 10–19 (2008)
4. Alexe, B., et al.: Simplifying Information Integration: Object-Based Flow-of-Mappings Framework for Integration. In: Castellanos, M., Dayal, U., Sellis, T. (eds.) BIRTE 2008. LNBIP, vol. 27, pp. 108–121. Springer, Heidelberg (2009)
5. Alexe, B., Hernández, M.A., Popa, L., Tan, W.C.: MapMerge: Correlating Independent Schema Mappings. PVLDB 3(1), 81–92 (2010)
6. Alexe, B., Hernández, M.A., Popa, L., Tan, W.C.: MapMerge: Correlating Independent Schema Mappings. VLDB Journal 21(1), 1–21 (2012)
7. Alexe, B., Kolaitis, P.G., Tan, W.C.: Characterizing Schema Mappings via Data Examples. In: ACM PODS, pp. 261–272 (2010)
8. Alexe, B.: Interactive and Modular Design of Schema Mappings. Ph.D. thesis, University of California, Santa Cruz (2011)
9. Alexe, B., ten Cate, B., Kolaitis, P.G., Tan, W.C.: Characterizing schema mappings via data examples. ACM TODS 36(4) (2011)
10. Alexe, B., ten Cate, B., Kolaitis, P.G., Tan, W.C.: Eirene: Interactive design and refinement of schema mappings via data examples. PVLDB (Demonstration Track) (2011)
11. Beeri, C., Vardi, M.Y.: A Proof Procedure for Data Dependencies. JACM 31(4), 718–741 (1984)
12. Bernstein, P.A., Haas, L.M.: Information Integration in the Enterprise. Commun. ACM 51(9), 72–79 (2008)
13. Microsoft BizTalk Server, `http://www.microsoft.com/biztalk`
14. Bonifati, A., Chang, E.Q., Ho, T., Lakshmanan, L.V.S.: HepToX: Heterogeneous Peer to Peer XML Databases (2005), `http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0506002`
15. Bonifati, A., Chang, E.Q., Ho, T., Lakshmanan, V.S., Pottinger, R.: HePToX: Marrying XML and Heterogeneity in Your P2P Databases. In: VLDB, pp. 1267–1270 (2005)
16. Fagin, R., Haas, L.M., Hernández, M., Miller, R.J., Popa, L., Velegrakis, Y.: Clio: Schema Mapping Creation and Data Exchange. In: Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S. (eds.) Conceptual Modeling: Foundations and Applications. LNCS, vol. 5600, pp. 198–236. Springer, Heidelberg (2009)
17. Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L.: Data Exchange: Semantics and Query Answering. TCS 336(1), 89–124 (2005)
18. Fagin, R., Kolaitis, P.G., Popa, L., Tan, W.C.: Composing Schema Mappings: Second-Order Dependencies to the Rescue. TODS 30(4), 994–1055 (2005)
19. Fuxman, A., Hernández, M.A., Ho, H., Miller, R.J., Papotti, P., Popa, L.: Nested Mappings: Schema Mapping Reloaded. In: VLDB, pp. 67–78 (2006)
20. International Nucleotide Sequence Database Collection, `http://www.insdc.org`

21. Kolaitis, P.G.: Schema Mappings, Data Exchange, and Metadata Management. In: PODS, pp. 61–75 (2005)
22. Lenzerini, M.: Data Integration: A Theoretical Perspective. In: PODS, pp. 233–246 (2002)
23. Madhavan, J., Halevy, A.Y.: Composing Mappings Among Data Sources. In: VLDB, pp. 572–583 (2003)
24. Maier, D., Mendelzon, A.O., Sagiv, Y.: Testing Implications of Data Dependencies. TODS 4(4), 455–469 (1979)
25. Altova MapForce, http://www.altova.com
26. Marnette, B., Mecca, G., Papotti, P., Raunich, S., Santoro, D.: ++spicy: an opensource tool for second-generation schema mapping and data exchange. PVLDB 4(12), 1438–1441 (2011)
27. Nash, A., Bernstein, P.A., Melnik, S.: Composition of Mappings Given by Embedded Dependencies. In: PODS, pp. 172–183 (2005)
28. Popa, L., Velegrakis, Y., Miller, R.J., Hernández, M.A., Fagin, R.: Translating Web Data. In: VLDB, pp. 598–609 (2002)
29. Rahm, E., Bernstein, P.A.: A Survey of Approaches to Automatic Schema Matching. VLDB Journal 10(4), 334–350 (2001)
30. Roth, M., Hernández, M.A., Coulthard, P., Yan, L., Popa, L., Ho, H.C.T., Salter, C.C.: XML Mapping Technology: Making Connections in an XML-centric World. IBM Sys. Journal 45(2), 389–410 (2006)
31. Shu, N.C., Housel, B.C., Taylor, R.W., Ghosh, S.P., Lum, V.Y.: EXPRESS: A Data EXtraction, Processing, and REStructuring System. ACM Trans. Database Syst. 2(2), 134–174 (1977)
32. Smith, J.M., Bernstein, P.A., Dayal, U., Goodman, N., Landers, T.A., Lin, K.W.T., Wong, E.: Multibase: Integrating Heterogeneous Distributed Database Systems. In: AFIPS National Computer Conference, pp. 487–499 (1981)
33. Stylus Studio, http://www.stylusstudio.com
34. U.S. Census Bureau, http://www.census.gov
35. Yan, L., Miller, R., Haas, L., Fagin, R.: Data-Driven Understanding and Refinement of Schema Mappings. In: SIGMOD, pp. 485–496 (2001)
36. Yu, C., Popa, L.: Semantic Adaptation of Schema Mappings when Schemas Evolve. In: VLDB, pp. 1006–1017 (2005)