Muse: Mapping Understanding and deSign by Example

Bogdan Alexe #1, Laura Chiticariu #2, Renée J. Miller *3, Wang-Chiew Tan #4

University of California, Santa Cruz
[1]abogdan@cs.ucsc.edu
[2]laura@cs.ucsc.edu
[4]wctan@cs.ucsc.edu

* *University of Toronto*
[3]miller@cs.toronto.edu

***Abstract*— A fundamental problem in information integration is that of designing the relationships, called *schema mappings*, between two schemas. The specification of a semantically correct schema mapping is typically a complex task. Automated tools can suggest potential mappings, but few tools are available for helping a designer understand mappings and design alternative mappings.**

We describe Muse, a mapping design wizard that uses data examples to assist designers in understanding and refining a schema mapping towards the desired specification. We present novel algorithms behind Muse and show how Muse systematically guides the designer on two important components of a mapping design: the specification of the desired grouping semantics for sets of data and the choice among alternative interpretations for semantically ambiguous mappings. In every component, Muse infers the desired semantics based on the designer's actions on a short sequence of small examples. Whenever possible, Muse draws examples from a familiar database, thus facilitating the design process even further. We report our experience with Muse on some publicly available schemas.

I. Introduction

A fundamental problem in information integration is the specification of the relationships between a source schema and a target schema [1]. Such a specification is called *schema mappings*. In some systems such as Map Force[1], Stylus Studio[2] schema mappings are specified with transformation code (e.g., XSLT or Java code). In other systems, including Clio [2], [3], HePToX [4], Microsoft's mapping composer [5], and IBM's Rational Data Architect[3] schema mappings are specified using a declarative language based on a logical formalism. A benefit of such languages is that they facilitate the reuse of mappings for different integration tasks: they can be used to generate executable transformation code for data exchange [2], for query translation (reformulation) in data integration [6], [7], [8], to compose mappings in a peer network environment [5], [9], [10], [11], and numerous other model management tasks.

Tools, such as Clio and HePToX, for semi-automatically creating mappings make the use of declarative mappings viable by automating a large amount of the mapping design task. The well-known 80-20 rule applies in mapping design. Mapping creation tools can automate 80% of the work, covering common cases and creating a mapping that is close to correct. However, ensuring complete correctness can still require intricate manual work to perfect portions of the mapping. Previous research on mapping understanding and refinement [12] and anecdotal evidence from mapping designers suggest that this perfection process can be facilitated by using data examples to explain the mapping and alternative mappings. Mapping designers usually understand their data better than they understand mapping specifications and could therefore, leverage familiar data examples to illustrate nuances of how a small change to a mapping specification changes its semantics. The work of Yan et al. [12] is based on this observation. A designer can understand and refine a mapping specification, given by SQL queries, between two relational schemas by walking through examples of source data, and seeing how this data would be transformed by different choices of mapping specifications. As part of this work, they studied alternative interpretations of a semantically 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, a schema mapping could be ambiguous because it asserts that a project supervisor is a project manager or 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 a source database as the supervisor of a project in the target database since there are two alternative interpretations to this ambiguous mapping.

Our work is largely inspired by Yan et al. [12]. As in their work, Muse uses examples to differentiate between alternative mapping specifications and infer the desired mapping semantics based on the designer's actions. However, we go significantly beyond the techniques and space of alternative mappings supported by [12].

First, Muse is capable of helping a designer derive the desired grouping semantics for a mapping specification using examples. For example, to infer whether a designer wishes to group projects by a a company's name and location or only by a company's name, Muse automatically constructs a small number of, essentially, yes-or-no questions using

[1]http://www.altova.com
[2]http://www.stylusstudio.com/
[3]http://www.ibm.com/software/data/integration/rda

small examples. The designer's answers to these questions will allow Muse to infer the desired grouping semantics. For schemas without keys or functional dependencies, the number of questions we pose to the designer is the number of schema elements that the designer could use for designing the grouping semantics. The size of an example typically consists of at most two tuples in each (nested) relation. Furthermore, Muse exploits keys in the source schema (or more general functional dependencies when available) to reduce the number of questions a designer must consider.

Second, as in [12], Muse helps a designer choose among alternative interpretations of an ambiguous mapping. Muse constructs a small example that would illustrate, and differentiate, all interpretations. The example is at most as big as the size of the specification of the ambiguous mapping. Each (nested) relation in the source typically contains only a few tuples. In [12], the designer is asked to select among the target instances generated by the source example through each alternative interpretation of the mapping. There are as many target instances as the number of alternative interpretations, which can be overwhelming. In Muse, we show the designer one partial target instance and the designer is asked to select among a small set of data choices. Each choice is in fact a list of possible values for a target schema element, corresponding to all alternative interpretations. There are as many choices as the number of schema elements with more than one alternative interpretation, and this number is much smaller than the total number of alternative interpretations for the ambiguous mapping. The designer's actions on these choices translate into a unique interpretation of the ambiguous mapping.

Finally, unlike previous work which relies exclusively on a source instance to illustrate mappings, Muse can "fall back" to its own constructed example whenever a meaningful example cannot be drawn from the actual source instance. For the current source instance, two mappings may produce indistinguishable results. However, for mapping design it is important for a designer to understand the difference in the mappings over the space of possible source instances. Muse is able to automatically detect when an actual source instance is incapable of illustrating all design alternatives and if so, it is able to construct synthetic examples to illustrate differences in all design alternatives. We show experimentally that this feature of Muse is necessary to help design mappings for some real mapping settings and instances.

The design components of Muse have been shown to be important mapping design parameters. Grouping has been shown to be important in mappings for not only nested data, including XML, but also in designing mappings for such common tasks as schema evolution [3]. Furthermore, as described in [12], ambiguous mappings frequently arise in real schemas.

We describe our Muse wizard based on the schema mapping language [2] for relational and nested relational schemas that has been proposed for data exchange (Sec. II). This language is a generalization of commonly used mapping languages including source-to-target tuple generating dependencies [13], a common form of global-and-local-as-view (GLAV) mappings

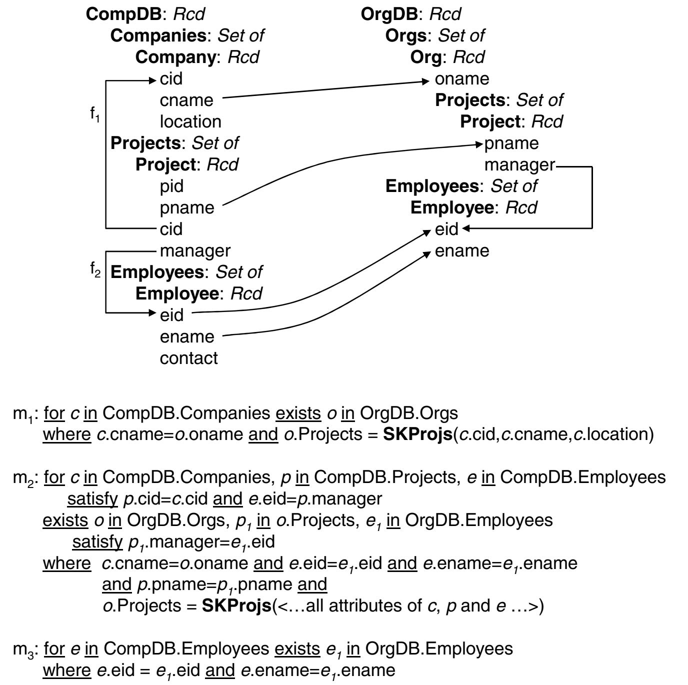

Fig. 1. A mapping scenario.

[7]. The mapping generation algorithms of mapping discovery tools [2], [3], [4] and some model management tools [14] make default decisions in creating mappings, which may not always be the desired ones. In the subsequent sections, we illustrate how Muse allows a designer to refine, understand, debug, and modify a mapping, using data examples, to create a mapping that corresponds to the designer's intended semantics.

II. Background on Mappings

Nested Relational Model. Fig. 1 shows a *mapping scenario* between two schemas, *CompDB* and *OrgDB* respectively, written in the nested relational (NR) representation of [2], [3]. 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 source schema, *Companies* is a set of *Comp* records where each record has three atomic elements: *cid* (company id), *cname* (company name) and *location*. Similarly, *Projects* and *Employees* are sets of *Proj* and respectively, *Emp* records. The two referential constraints f_1 and f_2 specify that for every *Proj* tuple p, there must exist a *Comp* tuple c and an *Emp* tuple e such that $p.cid = c.cid$ and $p.manager = e.eid$. The target schema is a slight reorganization of the source.

An NR schema is formally a set of labels $\{R_1,...,R_k\}$, called *roots*, where each root is associated with a type τ, defined by the following grammar: $\tau ::= \mathsf{String} \mid \mathsf{Int} \mid \mathsf{SetOf}\ \tau \mid \mathsf{Rcd}[l_1 : \tau_1, ..., l_n : \tau_n] \mid \mathsf{Choice}[l_1 : \tau_1, ..., l_n : \tau_n]$. The types String and Int are atomic types and Rcd and Choice are complex types. (The atomic types are not shown in Fig. 1.)

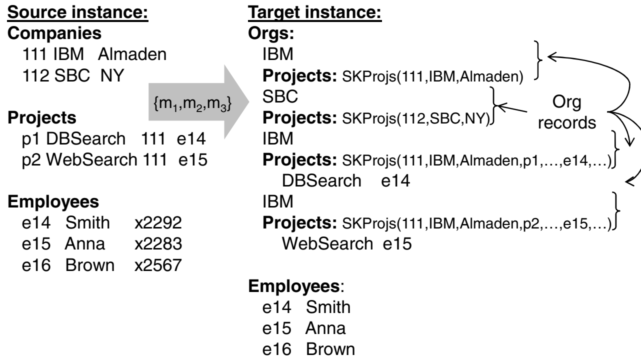

Fig. 2. The result of chasing the source with $\{m_1,m_2,m_3\}$.

A (record) value of type Rcd$[l_1 : \tau_1, ..., l_n : \tau_n]$ is a set of label-value pairs $[l_1 : a_1, ..., l_n : a_n]$, where $a_1, ..., a_n$ are of types $\tau_1, ..., \tau_n$, respectively. A value of type Choice$[l_1 : \tau_1, ..., l_n : \tau_n]$ is a single label-value pair $[l_k : a_k]$, where a_k is of type τ_k and $1 \leq k \leq n$. The set type SetOf τ (where τ is a complex type) is used to model repeatable elements. Order is not considered, hence SetOf represents unordered sets. A value of type SetOf τ is represented by a *SetID* and an associated (possibly empty) set of values $\{v_1, ..., v_m\}$, where each v_i is of type τ. Every type is associated with a path from the root to that type, and whenever we refer to a type, we assume that this path is implicit and uniquely determined (i.e., we cannot have two distinct types with the same path). We use the terms *set types* and *nested sets* interchangeably. For example, *Orgs*, *Projects*, *Employees* are all nested sets (or set types) in the target schema. We frequently use the term *tuple* to refer to a value of type record.

To simplify our discussions, we assume that XML schemas are modeled using a single schema root of record type whose elements are all of type SetOf. We also assume strict alternation of set and record types. In our implementation, however, we handle the NR model in its full generality.

Mappings. A schema mapping is a triple ($\mathbf{S}$, $\mathbf{T}$, Σ), where Σ is a set of *mappings* specified using the language of [3]. For ease of exposition, we use the mapping language of [2], which is a special sublanguage of [3], in this paper. Our implementation of Muse in fact handles the full generality of the language of [3] (see [15]). As an example, let $\mathbf{S}$ and $\mathbf{T}$ be the source and target schemas of Fig. 1. Then, ($\mathbf{S}$, $\mathbf{T}$, $\{m_1, m_2, m_3\}$) is a schema mapping, where m_1 to m_3 are shown in the figure. Intuitively, m_1 is a specification that maps *Comp* names to *Org* names. More precisely, it states that whenever a *Comp* tuple c exists in the source, there must be an *Org* tuple o in the set *OrgDB.Orgs* in the target such that $o.oname = c.cname$. The value of $o.Projects$ is the SetID (also called *grouping function* or *Skolem function*) SKProjs($c.cid$,$c.cname$,$c.location$). By convention, we use SKN to denote the SetID name of a nested set N in the target schema. For example, the SetID name of the nested set *Projects* is SKProjects, which we write SKProjs for short (or SK when there is no ambiguity). We sometimes refer to a nested set N simply as SKN. We assume that every nested set in the target schema has a different SetID name. The mapping m_2 states that for every *Comp* tuple c, *Proj* tuple p and *Emp* tuple e such that c, p and e satisfy the referential constraints f_1 and f_2, then there must be corresponding *Org*, *Proj* and *Emp* tuples (o, p_1 and e_1 respectively) in the target with the appropriate values extracted from c, p and e. The mapping m_3 migrates employee information to the target, independently of whether the employee is a manager of some project.

These mappings are expressed in the "query-like" notation of [2], [3] where each variable in the for and exists clauses binds to tuples in a source and respectively, a target nested set. The type of each variable is, hence, a record. The *correspondences* between atomic schema elements (e.g., $cname$ to $oname$), which are shown in Fig. 1 as arrows, are expressed as equalities (e.g., $c.cname = o.oname$) in the where clause of the mapping. In addition, the satisfy clauses following the for and respectively, the exists clause may contain equalities to express source and respectively, target referential constraints. In Muse, we will be constructing source and target instances that satisfy proposed mappings and any constraints on the schemas. Hence, to simplify our discussion in this paper, we assume that mappings are *closed under source and target referential constraints*. For example, m_1, m_2 and m_3 in Fig. 1 are mappings that are closed under the source and target referential constraints. However, the following mapping m does not satisfy the source referential constraint f_1 because "c in CompDB.Companies" and "$p.cid = c.cid$" are missing from the for and corresponding satisfy clauses respectively.

for p in CompDB.Projects, e in CompDB.Employees
  satisfy e.eid = p.manager
exists e_1 in OrgDB.Employees
where e.eid = e_1.eid and e.ename = e_1.ename

A mapping that is not closed under referential constraints can always be transformed into an equivalent one that is closed under referential constraints by chasing [16]. Mappings that are generated by tools such as Clio are always closed under *acyclic* referential constraints. The acyclicity condition can in fact be weakened in ways that have been studied by others.

Solutions, chase and homomorphisms. Fig. 2 shows a source instance I and a *solution* for I with the schema mapping in Fig. 1. A target instance J is a *solution* for I under the schema mapping if I and J together satisfy Σ.

The chase procedure has been used to generate solutions in data exchange [13]. The target instance in Fig. 2 is in fact the the result of *chasing* the source instance I with Σ = $\{m_1$, m_2, $m_3\}$. We describe intuitively the chase process and refer the interested reader to [2], [3], [13] for details of the chase procedure. The instance I is chased with each member of Σ. Suppose I is first chased with m_1. Due to m_1 and the two *Comp* records in I, two *Org* records are constructed in the target: *Org*(IBM, SKProjs(111,IBM,Almaden)) and *Org*(SBC, SKProjs(112,SBC,NY)). Due to m_2, the first *Comp* record and the two *Proj* records, along with their corresponding managers in I, two more *Comp* records, two *Proj* records (i.e.,

(DBSearch,e14) and (WebSearch,e15)) and their corresponding Employee records (i.e., *Emp*(e14,Smith) and *Emp*(e15, Anna)) are constructed in the target. The two *Proj* records that are constructed belong to distinct *Projects* sets whose SetIDs are SKProjs(111,IBM,Almaden,p1,...) and respectively, SKProjs(111,IBM,Almaden,p2,...). Next, I is chased with m_3 and three Employee tuples are constructed in the target, two of which (Smith and Anna) have already been generated by m_2. The result of chasing I with Σ is formed by taking the set union of all tuples that have been constructed. Special values, called *labeled nulls*, may be created during the chase. For example, suppose there is an extra attribute *address* in *Org* record, which does not correspond to any element in the source schema. The chase of I with m_1 will generate two *Org* tuples: (IBM, N_1) and (SBC, N_2), where N_1 and N_2 are labeled nulls used to represent, possible different, unknown *address* values.

In general, there are many possible solutions for a source instance I under a schema mapping $\mathcal{M} = \{\mathbf{S}, \mathbf{T}, \Sigma\}$. The space of all solutions for I under $\mathcal{M}$ is denoted as $Sol(\mathcal{M}, I)$, or $Sol(\Sigma, I)$ when $\mathbf{S}$ and $\mathbf{T}$ are understood from the context. It was shown in [13] that chasing I with Σ produces a *universal solution* for I under $\mathcal{M}$. Intuitively, a universal solution J for I under $\mathcal{M}$ is a most general solution in the space of all solutions for I in that there is a homomorphism from J to every solution for I under $\mathcal{M}$. We say that h is a *homomorphism* from an instance J to an instance J', denoted as $h : J \rightarrow J'$, if for every tuple $R(c_1, ..., c_n)$ in J, where R is a relation symbol, we have that $R(h(c_1), ..., h(c_n))$ is a tuple in J', and for every tuple $D(c_1, ..., c_n)$ in J, where D is a SetID, we have $h(D)(h(c_1), ..., h(c_n))$ is a tuple in J'. Furthermore, h has the following properties: (i) $h(c) = c$ if c is a constant, (ii) $h(D) = D'$ if D is a SetID and D' has the same set type as D, and (iii) $h(N)$ is a constant or labeled null if N is a labeled null. In other words, h is the identity on constants but not necessarily on SetIDs or labeled nulls. We say that J and J' are *homomorphically equivalent* if there is a homomorphism from J to J' and a homomorphism from J' to J. We say that J and J' are *isomorphic* if there is a one-to-one homomorphism from J to J' and vice versa.

III. Designing Grouping Functions

Grouping or combining related data together is an essential functionality of many integration systems. In this section, we describe the grouping design wizard of Muse, called Muse-G. We show how Muse-G infers a desired grouping function through the actions taken by the designer on a short sequence of small data examples (or questions).

The mappings generated by mapping generation tools [2], [3], [4] and some model management tools [14] define a *default grouping function* for every nested set in the target schema. The grouping functions are a restricted form of *Skolem functions*, where the arguments consist of only atomic attributes. For example, in [3], the default grouping function for *Projects* in m_2 is SKProjs($c.cid$,$c.cname$,$c.location$,$p.pid$,$p.pname$, $p.cid$,$p.manager$,$e.eid$,$e.ename$,$e.contact$). In other words, *Proj* records are grouped according to the values of all attributes of the *Comp*, *Proj* and *Emp* records. If SKProjs(*cname*) is the grouping function instead, then *Proj* records are grouped according to *cname* of *Comp* records (i.e., *oname* of *Org* records). (We write SKProjs(*cname*) instead of SKProjs(*c.cname*) when there is no ambiguity.) By default, there are no grouping functions for topmost-level sets. Hence, in Fig. 1, there are no grouping functions for *Orgs* and *Employees* in the target. Most tools (Mapforce, Stylus Studio and [2], [3], [4]) only support the manual specification or modification of grouping functions, where the arguments of the grouping function have to be explicitly specified. This can prove to be a difficult task if the schemas are large or the number of possible arguments for a grouping function is large. 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 designer, what the n possible grouping attributes are (see [2], [3]).

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 designer has in mind. As the examples illustrate the different possibilities of grouping, Muse-G can also be very useful when the designer only has a partial understanding of the desired grouping semantics. Naturally, an advanced designer can always choose to specify the desired grouping function explicitly without using Muse-G.

If there is at most one key per nested set in the source schema (a very common case) and there are n attributes that a designer can group by, then Muse-G asks at most n questions to infer the desired grouping function. All source schemas we have encountered in Sec. VI fall into this category. Moreover, each question makes use of a small (hence amenable) example, where each nested set in the source typically has two tuples. Our experimental results justify that for these schemas, the number of questions posed remains small for a natural class of desired grouping functions. In the following, we keep the discussion informal and illustrate the ideas behind Muse-G with examples. All algorithms and proofs of our technical results can be found in [15].

A. The Basic Algorithm behind Muse-G

We first describe the algorithm behind Muse-G when there are no functional dependencies (FDs) in the source schema. Extensions to handle keys (and FDs in general) in the source schema are described in Sec. III-B (and [15] respectively).

Muse-G takes as input a schema mapping ($\mathbf{S}$, $\mathbf{T}$, Σ). The designer 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 Fig. 1.

Step 1. The first step is to determine an order to the set of grouping functions that the designer 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 *Org*, *Emp*, and *Proj*. Since SKOrgs and SKEmps are top-level sets without grouping functions, Muse-G will only design the grouping function for *Projects* (i.e., SKProjs) in m_2. If there were another nested set *Grants* under *Projects* in $\mathbf{T}$ and m_2 would be a mapping that maps to both *Projects* and *Grants* in the target, then Muse-G would design SKProjs before SKGrants. When designing SKGrants, Muse-G will make use of the grouping function already designed for SKProjs.

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 *Org* tuple in *Orgs*. According to the for clause of m_2, the existence of an *Org* tuple is dependent on the existence of a *Comp* tuple in *CompDB.Companies*, an *Emp* tuple in *CompDB.Employees* and a *Proj* tuple in *CompDB.Projects* which agrees with the *Comp* and *Emp* tuples on the values of *pid* and *manager*, respectively. This means that *poss*(m_2,SKProjs) consists of the set of attributes in the *Comp*, *Proj* and *Emp* records. However, to simplify our subsequent discussion, we shall assume that *poss*(m_2,SKProjs)=$\{$*cid*,*cname*,*location*$\}$.

Step 3. Suppose the designer has SKProjs(Z) in mind, where $Z \subseteq$ *poss*(m_2, SKProjs). Muse-G now proceeds to probe and construct examples to infer the desired grouping function.

Probe and construct examples. Muse-G probes every attribute in the set *poss*(m_2,SKProjs)=$\{$*cid*,*cname*,*location*$\}$. The goal of each probe is to carefully construct a small example source instance I_e, from which two differentiating target instances are obtained: one is the result of including the probed attribute as part of SKProjs in m_2, and the other omits it. Suppose we probe on *cid* first. Muse-G first constructs its own example instance I_e, as shown below.

$$I_e\text{:}\{\textit{Comp}(c_1, n_1, l_1),\ \textit{Proj}(p_1, pn_1, c_1, e_1),\ \textit{Emp}(e_1, en_1, cn_1),\\ \textit{Comp}(c_2, n_1, l_1),\ \textit{Proj}(p_2, pn_2, c_2, e_2),\ \textit{Emp}(e_2, en_2, cn_2)\}$$

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 *Comp* 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 SKProj($\mathbf{y}$) will be non-isomorphic. Indeed, the former target instance will contain two distinct *Proj* sets, while the latter consists of only one *Proj* set. Next, Muse-G executes the following query against the actual source instance I in order to retrieve real tuples for the example instance I_e.

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

All variables of Q^{I_e} are universally-quantified. The two Company 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 designer 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. However, this may lead to combinations that are misleading to the designer. 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 Fig. 3(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 Fig. 3(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 designer "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 designer's choice of Scenario 1 or 2, Muse-G correctly determines whether *cid* is part of the designer'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 designer has the grouping function SKProjs(*cname*) in mind. She would select Scenario 2 in Fig. 3(a). We now repeat the process for the other attributes *cname* and *location*. Fig. 3(b) shows the example source instance and the two scenarios obtained by probing on *cname*. The two source *Comp* tuples must differ on the values of *cname* and agree on the values of *location*. Note that the *cid* values of the two *Comp* tuples are not required to be identical, since *cid* is not an argument of SKProjs. The designer will pick Scenario 1 in Fig. 3(b), since she wants to group *Projects* by *cname*, and Muse-G infers that *cname* is an argument to SKProjs. Fig. 3(c) shows the result of probing on *location*, where the designer will pick Scenario 2. Since *cname* is part of the grouping, the *Comp* 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 designer's choice in Fig. 3(c). At this point, Muse-G concludes and returns SKProjs(*cname*).

For simplicity, we have assumed above that *poss*(m_2, SKProjs) is $\{$*cid*,*cname*,*location*$\}$, when in fact it consists of all attributes of *Comp*, *Proj* and *Emp* records. In this case, Muse-G concludes only after subsequently probing all the attributes of *Proj* and *Emp* records (the designer will choose Scenario 2 in each case).

Note that it is conceivable for Muse-G to generate homomorphically equivalent target instances for Scenarios 1 and 2 (e.g., Fig. 3(b)). However, it is always possible to distinguish between such instances, as they are non-isomorphic.

Properties of Basic Muse-G. There are 2^n different grouping functions for each nested set SK in a mapping m, where n

(a)
Example source: Companies: 11 IBM NY; 12 IBM NY. Projects: P1 DB 11 e4; P2 Web 12 e5. Employees: e4 Jon x234; e5 Anna x888.
Target instances: Scenario 1: OrgDB / Orgs / IBM / Projects:SK(11,**y**) / DB e4 / IBM / Projects:SK(12,**y**) / Web e5 / Employees / e4 Jon / e5 Anna
Scenario 2: OrgDB / Orgs / IBM / Projects:SK(**y**) / DB e4 / Web e5 / Employees / e4 Jon / e5 Anna
Note: $\mathbf{y} \subseteq$ {IBM,NY}

(b)
Example source: Companies: 11 IBM NY; 14 SBC NY. Projects: P1 DB 11 e4; P4 WiFi 14 e6. Employees: e4 Jon x234; e6 Kat x331.
Target instances: Scenario 1: OrgDB / Orgs / IBM / Projects:SK(IBM,**y**) / DB e4 / SBC / Projects:SK(SBC,**y**) / WiFi e6 / Employees / e4 Jon / e6 Kat
Scenario 2: OrgDB / Orgs / IBM / Projects:SK(**y**) / DB e4 / WiFi e6 / SBC / Projects:SK(**y**) / DB e4 / WiFi e6 / Employees / e4 Jon / e6 Kat
Note: $\mathbf{y} \subseteq$ {NY}

(c)
Example source: Companies: 11 IBM NY; 13 IBM SF. Projects: P1 DB 11 e4; P2 Web 13 e5. Employees: e4 Jon x234; e5 Anna x888.
Target instances: Scenario 1: OrgDB / Orgs / IBM / Projects:SK(IBM,NY) / DB e4 / IBM / Projects:SK(IBM,SF) / Web e5 / Employees / e4 Jon / e5 Anna
Scenario 2: OrgDB / Orgs / IBM / Projects:SK(IBM) / DB e4 / Web e5 / Employees / e4 Jon / e5 Anna

Fig. 3. Probing on (a) *cid*, (b) *cname*, and (c) *location* when the designer has SKProjs(*cname*) in mind.

= $|poss(m, \mathrm{SK})|$. However, Muse-G determines the desired grouping function by asking the designer only $|poss(m, \mathrm{SK})|$ questions. Furthermore, Muse-G constructs a small source example at each probe. The size of the source example is twice the number of "$x \in X$" clauses in <u>for</u> clauses of m. This typically means there are at most two tuples in each nested set.

Next, we describe how we have extended the basic algorithm to potentially reduce the number of questions posed to the designer when keys are present in the source.

B. Muse-G with Keys

In this section, we assume that key constraints may be specified on nested sets in the source schema. A key of a nested set N is a *minimal* set of attributes in N that functionally determines all attributes of N. We say that an instance I is a *valid instance for a set* $\mathcal{F}$ *of keys* if I satisfies every key in $\mathcal{F}$. In the presence of keys, the example I_e constructed when probing an attribute may not be valid with respect to the keys. To see this, suppose *cid* is the key for *Companies*. Consider SKProjs in m_2 and suppose we probe on *cname* first. Two *Comp* tuples (c_1,n_1,l_1) and (c_1,n_2,l_1) are created, which clearly do not satisfy the key. Even if we had probed on *cid* before *cname*, we may still construct an instance that does not satisfy the key. For example, assume the designer's desired grouping function is SKProjs(*cid*,*cname*) and Muse-G first probes on *cid*. The source instance and two scenarios that are constructed are as shown in Fig. 3(a). Since the designer has SKProjs(*cid*,*cname*) in mind, she picks Scenario 1, and hence Muse-G infers that *cid* is part of SKProjs. Subsequently, when probing on *cname*, two *Comp* tuples, (c_1,n_1,l_1) and (c_1,n_2,l_1), are constructed. Clearly, they do not satisfy the key.

It turns out that if $\mathcal{F}$ is such that every nested set has at most one key (of any arity), then there is a natural order of attributes to probe such that a valid instance for $\mathcal{F}$ is always constructed by Muse-G. The procedure for computing this natural order is based on the following result which implies that if K is a key of *poss*(m,SK), then the inclusion of K as arguments of SK makes the inclusion of other attributes of *poss*(m,SK) as arguments of SK inconsequential (Thm 3.2). For example, if *cid* is the key for *Companies*, then m_2 with SKProjs(*cid*) has the same effect as m_2 with SKProjs(*cid*,*cname*) or SKProjs(*cid*,*location*) or SKProjs(*cid*,*cname*,*location*), *for all instances*.

Definition 3.1: Let m_1 and m_2 be two mappings between a source schema $\mathbf{S}$ and a target schema $\mathbf{T}$. We say that m_1 *has the same effect as* m_2 if for every instance I over $\mathbf{S}$ we have that $Sol(\{m_1\}, I) = Sol(\{m_2\}, I)$.

This relation "has the same effect" is reflexive, symmetric and transitive. Note that two mappings have the same space of solutions if an only if their corresponding universal solutions are homomorphically equivalent [13]. We already took advantage of a weaker form of this property in Sec. 3.1 where we constructed instances I_e on which a set of mappings would produce isomorphic results. But the property above is much stronger in that it must hold for all instances. In this paper, we are interested in comparing m_1 and m_2 when they differ only in one grouping function (e.g., SKProjs(X_1) vs. SKProjs(X_2) with $X_1 \neq X_2$)).

Theorem 3.2: Let m be a mapping and SK be a grouping function that is defined in m. Let K be a key of *poss*(m,SK), and let W be a set of attributes in *poss*(m,SK). Then m with SK(K) has the same effect as m with SK($K \cup W$).

Given this result, a potentially rewarding order of attributes to probe in the presence of a key K would be K first, followed by the rest of attributes in *poss*(m,SK) if necessary (i.e., only if K is not chosen).

Continuing with our example, where *cid* is the key for *Companies*, suppose Muse-G is in the process of determining SKProjs by first probing on *cid*. If the designer picks Scenario 1 in Fig. 3(a), then Muse-G can immediately conclude SKProjs(*cid*), since any combination of grouping attributes that includes *cid* will have the same effect. Hence, Muse-G has avoided two probes on *cname* and *location* and consequently, avoided two unnecessary questions.

A technical difficulty that arises in the presence of multiple keys is that the technique used to construct the illustrative example I_e, described in the previous section, may not always be valid with respect to the keys. For example, if both *cid* and *cname* are keys for Companies, then probing on *cid* will construct the instance I_e from Sec. III-A. Clearly, this instance does not satisfy the key *cname* for *Companies*. However, if *cid*

is the only key, then probing on *cid* first does not result in an invalid instance.

Based on the observations above, we have extended the algorithm behind Muse-G to handle the case when the source schema has key constraints. Whenever there is only a single key, Muse-G avoids creating invalid instances by first probing attributes that belong to the key. In this case, Muse-G asks k questions, where k is the number of attributes in the key, before deciding on which attributes to probe next. In fact, if every nested set in the source schema has at most one key, we show that the number of questions asked by Muse-G is *at most* $|poss(m, \text{SK})|$. This is the case for all real schemas that we have encountered in Sec. VI.

Corollary 3.3: Let m be a mapping between a source schema $\mathbf{S}$ and a target schema. Let SK be a grouping function defined in m. If every nested set in $\mathbf{S}$ has at most one key, then Muse-G on SK asks the designer at most $|poss(\text{m}, \text{SK})|$ questions.

If there are multiple keys per nested set in the source schema, Muse-G takes a different approach to infer the desired grouping function in order to avoid creating invalid instances. If the designer intends to group by only one of the keys, then Muse-G determines the desired grouping function by asking only one question. This is possible through exploiting the fact that grouping by one key has the same effect as grouping by any superset of the key (including all keys). Otherwise, Muse-G will attempt to understand which subset of non-key attributes is the designer's desired grouping function. The complete details of this part of Muse-G can be found in full version of our paper [15].

C. Extensions to Muse-G

We briefly mention other extensions to Muse-G given in [15].

Muse-G with Functional Dependencies. In [15], we detail how Muse-G constructs examples in the presence of FDs. We give a generalization of Theorem 3.2 for FDs. We show that if $P \rightarrow Q$ holds in *poss*(m,SK), then the inclusion of P as arguments of SK makes the inclusion of Q as arguments of SK inconsequential. We also give a necessary and sufficient condition that characterizes when a set of FDs $\mathcal{F}$ is *single-keyed*. This allows us to generalize the Muse-G algorithm outlined in Sec. III-B to arbitrary functional dependencies.

Incremental Muse-G. Even after all grouping functions for a mapping m have been designed, a designer may wish to return to refine her design sometime later. Incremental Muse-G helps a designer refine an existing grouping function SK of m, without restarting the Muse-G algorithm from scratch, by choosing to "group more" (i.e., merge multiple nested sets into a bigger nested set) or "group less" (i.e., split a nested set into multiple smaller nested sets) on SK.

Designing grouping functions only for the instance I. Muse-G correctly designs grouping functions for a mapping m so that m produces the desired grouping effect on *any* source instance. If the designer is only interested in designing mappings for a specific source instance I, we have modified Muse-G to first identify attributes whose inclusion or exclusion as arguments of SKN is inconsequential for the grouping semantics of N records for the instance I. Muse-G will avoid some questions to the designer by not probing these attributes.

IV. Disambiguating Mappings

We use the scenario in Fig. 4(a) to illustrate Muse-D, the component wizard of Muse that disambiguates mappings. Observe that atomic elements from two different record types *Proj* and *Emp* in the source are associated together in the same *Proj* record in the target. Moreover, there are two referential constraints in the source, from *Proj* to *Emp*.

The mapping scenario can be interpreted in several ways, four of which we have condensed into the mapping m_a shown in Fig. 4(a), which has been extended with or predicates to illustrate alternative interpretations. The non-bold parts are common to all four interpretations that we have, while each of the bold conjuncts represents two alternative ways of associating a *supervisor* (and *email*, respectively) with a *Proj.pname*. For example, the first set of or conditions specifies that one can extract either the *manager*'s name or the *tech-lead*'s name as the *supervisor* of a project.

We say that a mapping m is *ambiguous* if there exists at least one or predicate in its where clause. We assume that every group G of or conditions in an ambiguous mapping m are *alternatives for an atomic target element* A and is of the form ($s_1.A_1 = A$ or ... or $s_n.A_n = A$). We say m *is ambiguous for* A and there are n *alternatives for* A *according to* m.

A. The Muse-D Algorithm

The Muse-D algorithm takes as input a schema mapping ($\mathbf{S}$, $\mathbf{T}$, Σ), where Σ is a set of possibly ambiguous mappings, and a real source instance I, if available. For each ambiguous mapping $m \in \Sigma$, Muse-D constructs an example source instance I_e that differentiates among the underlying set of alternative (unambiguous) mappings that m encodes. In other words, if m represents l alternative mappings, the chase of I_e with each of the l unambiguous mappings results in l target instances that are pairwise different. Each target instance corresponds to one of the unambiguous mappings that m encodes. Hence, the designer's selection of one of these target instances can be translated into a selection of one of the underlying mappings. The example source instance that Muse-D generates for the schema mapping of Fig. 4(a) is shown in Fig. 4(b). Observe that Muse-D does not display four target instances. Instead, it compactly represents all target instances in one "instance" by factoring common parts (corresponding to chasing the non-bold part of mapping m_a in Fig. 4(a)) and displaying the alternatives for each ambiguous schema element according to m_a. If the designer picks the values Anna for *supervisor* and Jon@ibm for *email*, this means that the desired mapping is one that uses "e_2.*ename* = p_1.*supervisor*" and "e_1.*contact* = p_1.*email*" in the where clause of m_a.

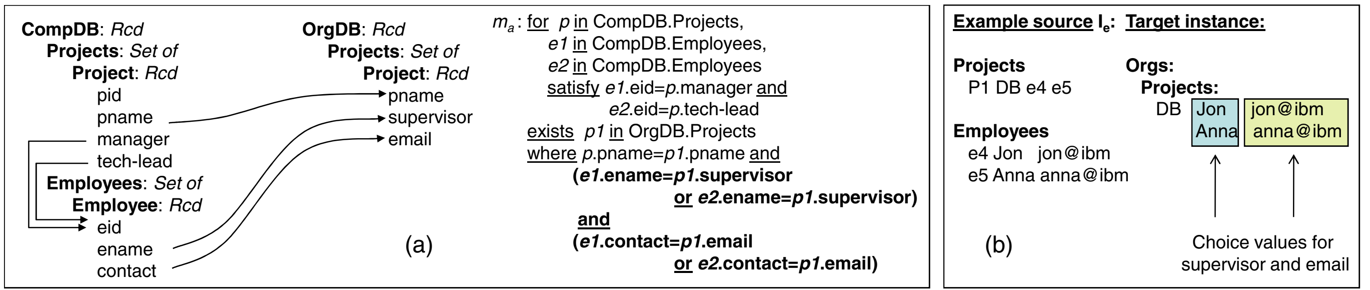

Fig. 4. (a) A mapping scenario and an ambiguous mapping; (b) Muse-D on the schema mapping in (a).

Next, we briefly illustrate how Muse-D constructs an example source instance that differentiates among all alternative mappings of the ambiguous mapping m_a in Fig. 4(a). Muse-D first constructs an example I_e which consists of a *Proj* tuple (p_1,pn_1,e_1,e_2) and two *Emp* tuples (e_1,en_1,cn_1) and (e_2,en_2,cn_2), corresponding to the manager and respectively, the technical leader of the project p_1. The query below is executed to replace I_e with real tuples from I:

$$Q^{I_e} : \textit{Proj}(p_1, pn_1, e_1, e_2) \wedge \textit{Emp}(e_1, en_1, cn_1) \wedge \textit{Emp}(e_2, en_2, cn_2) \wedge en_1 \neq en_2 \wedge cn_1 \neq cn_2$$

All variables of Q^{I_e} are universally quantified. Since *supervisor* and *email* are ambiguous elements according to m_a, we add the inequalities $en_1 \neq en_2$ and $cn_1 \neq cn_2$ to ensure that one can disambiguate mappings according to the designer's selection on these values. A possible real example constructed from $Q^{I_e}(I)$ is shown Fig. 4(b). If $Q^{I_e}(I)$ returns an empty result, then the synthetic instance I_e shown above would be presented to the designer instead.

Finally, Muse-D chases I_e to generate the target instance with "choices" shown in Fig. 4(b). Intuitively, the non-choice part of the target instance is generated by chasing I_e with the non-ambiguous part of m_a. The choices for an atomic target element are obtained by taking the union of values extracted from each alternative. After this, the designer "fills-in-the-choices" in the target instance. The completed target instance translates into an underlying mapping that m_a encodes.

Properties of Muse-D. For each ambiguous mapping m, Muse-D presents the designer with a single pair of source and target instances. The number of tuples in the source instance is the number of "$x \in X$" clauses in the for clause of m. The number of choice values the designer has to select in the target instance is the number of ambiguous elements in m.

More options. A designer may choose a subset of the four mappings as the desired interpretation in general. Muse-D allows the selection of multiple mappings by allowing the designer to select more than one value in each choice.

Note that the for clause of m_a in Fig. 4(a) expresses an inner join between *Employees* and *Projects*. Therefore, only employees that are both managers and technical leads are exchanged in the target. A designer can choose between inner or outer joins (e.g., exchange employees that are neither managers nor tech leads) in Muse-D. Here, we rely on the technique of Yan et al. [12] for constructing examples to differentiate between inner and outer joins.

Detecting ambiguities. So far we have assumed that ambiguities are specified as or predicates in the mapping. Techniques for detecting ambiguities when given a set of mappings (without or predicates) is an interesting subject for further investigation. However, we observe that Muse-D could work directly from mapping tools such as Clio, because ambiguities can be detected during mapping generation.

V. Using Muse

So far, we have described the Muse-G and Muse-D component wizards of Muse in isolation. These components may be used independently to refine mappings that are hand-coded or automatically generated. They may also be put together to form a complete mapping design wizard that would guide the designer, with examples, to the desired mapping specification, starting from mappings generated by tools such as Clio [2] or HePToX [4]. To exemplify, consider the Clio tool which helps a designer create a mapping scenario (e.g., those shown in Figs. 1 and 4(a)). Clio interprets such a mapping scenario into a set of (possibly ambiguous) mappings. If the default mappings generated are found to be unsatisfactory, Muse-D could be used to select the desired interpretation. The output of Muse-D is a set of unambiguous mappings. Muse-G can then be used to guide the designer towards the desired grouping semantics, if the default are unsatisfactory.

VI. Experience

To evaluate Muse, we use four pairs of source and target schemas as input to Clio, from which we design four mapping scenarios. The input schemas are (1) the relational and DTD schemas of the Mondial geographical database[4] (2) two nested schemas for the DBLP bibliography obtained from the DBLP website and respectively, the Clio schemas repository[5] (3) the relational TPCH schema [17] and a nested version of this schema which we created, and (4) the first relational schema in the Amalgam data integration benchmark [18] and a nested schema which we created based on the third Amalgam schema. As source instances, we used the Mondial database,

[4] http://www.dbis.informatik.uni-goettingen.de/Mondial/
[5] http://www.cs.toronto.edu/db/clio/testSchemas.html

Schema mapping	Average size of $poss(m, SK)$	Number of questions (average)	% times found real I_e	Average time to obtain I_e (s)	Group strat.
Mondial	13.1	2.6	38%	0.014	G_1
		8.5	41%	0.187	G_2
		2.9	40%	0.015	G_3
DBLP	11	1.5	17%	0.450	G_1
		11	11%	0.337	G_2
		1.5	17%	0.454	G_3
TPCH	26.7	1.5	0%	0.785	G_1
		17	12%	0.893	G_2
		1.5	0%	0.782	G_3
Amalgam	14.1	2	29%	0.013	G_1
		3	52%	0.043	G_2
		3	52%	0.030	G_3

Fig. 5. Experimental results with Muse-G.

scaled down versions of the DBLP bibliography and the TPCH database, and data for the first Amalgam schema. All schemas have key and foreign key constraints. The table below shows some characteristics of the mapping scenarios we designed. In the Mondial scenario, for example, the target schema has 8 nested sets with grouping functions. Clio generates 26 mappings of which 7 are ambiguous.

Mapping Scenarios	Size of I	Target sets w/ grouping	Number of mappings	Ambiguous mappings
Mondial	1MB	8	26	7
DBLP	2.6MB	6	4	0
TPCH	10MB	4	5	1
Amalgam	2MB	2	14	0

Muse-G. We considered three types of grouping functions, denoted as G_1, G_2 and G_3, respectively. Under G_1, every set is grouped by all possible attributes. Hence, G_1 produces the largest number of possible groups. For our example in Fig. 1, under G_1, SKProjs($\langle$all attributes of c, p, $e\rangle$) is the grouping function for *Projects*. Under G_2, every set SK is grouped by all source atomic elements that are exported to records that appear on the path from the root of the target schema to SK. For example, under G_2, the grouping function for *Projects* is SKProjs(c.*cname*). A slight variation of G_2 is given by G_3: Every set SK is grouped by all atomic elements in *poss*(m,SK) that are exported to the target schema. Under G_3, the grouping function for *Projects* is SKProjs(c.*cname*, p.*pname*, p.*manager*, e.*eid*, e.*ename*).

Fig. 5 summarizes our experience with Muse-G. We use DB2 v9 (with buffer pool size of 10 MB) and respectively, the Saxon-B implementation of XQuery to retrieve real tuples from relational and respectively, XML source instances. In all source schemas, there is at most one key for each nested set. Fig. 5 shows the average size of *poss*(m, SK) (i.e., the total size of *poss*(m,SK), over all m and SK, divided by the number of grouping functions in all mappings) and the average number of questions posed to the designer (i.e., the average number of attributes probed) over all sets SK. Muse-G was able to reduce the number of questions posed to the designer in most cases, in the presence of keys. For example, Muse-G asked, on average, only 1.5 questions per nested set under G_1 and G_3 in the DBLP scenario, where the average size of *poss*(m, SK) is 11. Recall that all attributes in *poss*(m, SK) are probed in the absence of keys. Hence, Muse-G avoided 9.5 questions per nested set in these cases, on average. However, Muse-G is unable to reduce the number of questions posed to the designer when she has G_2 in mind because it happens that the attributes for G_2 do not contain the key of *poss*(m, SK), for any mapping m and set SK of m in this scenario.

Fig. 5 shows that in all scenarios, Muse-G was able to extract real tuples from the source instance and present the designer with a "real" example I_e up to 52% of the time. Note that in all the schema mappings we have used in our experiments, it is not possible to extract real source examples all the time. Hence, the ability of Muse-G to construct appropriate examples is important. The average time required to construct and retrieve I_e from the source instance was subsecond in all cases. Note that the performance of Muse-G mainly depends on the performance of queries that extract I_e from the source instance I. We have implemented various strategies to avoid having the designer wait a long time for real examples from Muse-G in cases when I is large. For example, we exploit the "think time" of the designer on one example to precompute other examples ahead of time in the background. Muse-G also falls back to its own artificially constructed example if a real example was not found after a fixed amount of time.

Muse-D. We used Muse-D to disambiguate among alternative mappings in the Mondial and TPCH scenarios. (There are no ambiguous mappings for the DBLP and Amalgam scenarios.) Below, we show the number of alternative mappings that are encoded by the ambiguous mappings that Clio generates. For example, for the Mondial scenario, the 7 ambiguous mappings encode 208 mapping alternatives in total. The number of pairs of source and target examples generated in each scenario (i.e., the number of questions posed to the designer) is also shown. This number is equal to the number of ambiguous mappings. In all cases, we were able to extract real examples from the actual source instance to illustrate the ambiguities.

Schema Mapping	Alternatives encoded	Num. questions	Size of I_e (# tuples)	# Ambiguous vals. in target inst.
Mondial	208	7	3–4	4–5
TPCH	16	1	9	4

It is important to observe that the sizes of the example source instances and the number of ambiguous values Muse-D shows in the target instances are small compared to the number of mapping alternatives. In Mondial, for example, Muse-D disambiguates among 208 mapping alternatives by showing only 7 examples, where each source example consists of 3 to 4 tuples and the corresponding target instances have 4 to 5 ambiguous elements, each with two choice values.

VII. Related Work and Conclusion

Grouping functions (or Skolem functions) have been used for schema translation and schema augmentation of object-based data models [19], as well as in many tools for managing and creating schemas and mappings [2], [3], [5], [14], [20], [21]. In all cases, grouping functions are automatically generated and may be manually modified. To the best of our knowledge, Muse-G is the first design wizard for grouping functions.

Muse-D is inspired by the vision of Yan et al. [12], but greatly extends its functionality (as described in Section I). Muse is fundamentally different from previous work on form filling [22], Query-By-Example (QBE) paradigms (perhaps the most notable one is [23]), and browsing and querying paradigms (e.g., [24]). Muse uses data examples to illustrate nuances in small changes to an *existing* mapping, while all the work mentioned above is about assisting a designer to build a (valid) query. The work of Rowe [22] is similar to Muse-D in that it requires the user to fill-in some values in a form with empty fields. A form corresponds to the schema of a view of the underlying database. Arbitrary values are allowed in each field, and each entered value translates to a selection predicate on the underlying query from which more tuples may be retrieved. In Muse-D, each "blank" in the target instance contains a list of alternatives (no arbitrary choices are allowed). A completion of the target instance corresponds to a selection of a unique underlying mapping.

Examples have been used to illustrate constraints, such as functional and inclusion dependencies [25]. An example database (called an Armstrong database [26]) is a database that satisfies exactly a given set of constraints and their logical consequences, and no other constraints. Since such databases illustrate constraints that hold or do not hold, it is useful for alerting the designer of possible extra or missing constraints. Muse uses data examples for illustrating the differences in semantics resulting from small changes to a mapping. It may not show which mappings are missing.

The work of [27] allows a designer to understand and debug schema mappings by showing the relationships, called routes, between selected source or target data. Our work is complementary to [27]. Their system does not automatically "guide" the designer, with examples, in creating or refining the schema mapping. Rather, the designer must manually change a mapping, once a problem has been identified by analyzing the routes. Moreover, the design of grouping functions is not considered in [27].

Conclusion. We have described Muse, a mapping design wizard that uses data examples to help designers understand, design, and refine schema mappings. Muse permits a designer to work with data rather than with complex specifications to understand a mapping's semantics. Muse works on two important components of a mapping specification, corresponding to the design of desired grouping semantics for mappings (Muse-G) and the desired interpretation of ambiguous mappings (Muse-D). Muse explores a large and comprehensive design space of alternative mappings to ensure a designer can efficiently arrive at her desired mapping semantics.

Acknowledgements. Alexe, Chiticariu and Tan are partly supported by NSF CAREER Award IIS-0347065 and NSF grant IIS-0430994. Work partially done while Tan was visiting the IBM Almaden Research Center.

References

[1] P. Bernstein and S. Melnik, "Model Management 2.0: Manipulating Richer Mappings," in *SIGMOD*, 2007, pp. 1–12.
[2] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin, "Translating Web Data," in *VLDB*, 2002, pp. 598–609.
[3] A. Fuxman, M. A. Hernández, H. Ho, R. J. Miller, P. Papotti, and L. Popa, "Nested Mappings: Schema Mapping Reloaded," in *VLDB*, 2006, pp. 67–78.
[4] A. Bonifati, E. Q. Chang, T. Ho, and L. V. S. Lakshmanan, "HepToX: Heterogeneous Peer to Peer XML Databases," 2005. [Online]. Available: http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0506002
[5] P. A. Bernstein, T. J. Green, S. Melnik, and A. Nash, "Implementing Mapping Composition," in *VLDB*, 2006, pp. 55–66.
[6] A. Y. Halevy, "Answering Queries Using Views," *VLDB Journal*, vol. 10, no. 4, pp. 270–294, 2001.
[7] M. Lenzerini, "Data Integration: A Theoretical Perspective," in *PODS*, 2002, pp. 233–246.
[8] C. Yu and L. Popa, "Constraint-Based XML Query Rewriting For Data Integration." in *SIGMOD*, 2004, pp. 371–382.
[9] J. Madhavan and A. Y. Halevy, "Composing Mappings Among Data Sources," in *VLDB*, 2003, pp. 572–583.
[10] R. Fagin, P. G. Kolaitis, L. Popa, and W. Tan, "Composing Schema Mappings: Second-Order Dependencies to the Rescue," *TODS*, vol. 30, no. 4, pp. 994–1055, 2005.
[11] A. Nash, P. A. Bernstein, and S. Melnik, "Composition of Mappings given by Embedded Dependencies," in *PODS*, 2005, pp. 172–183.
[12] L. Yan, R. Miller, L. Haas, and R. Fagin, "Data-Driven Understanding and Refinement of Schema Mappings," in *SIGMOD*, 2001, pp. 485–496.
[13] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa, "Data Exchange: Semantics and Query Answering," *TCS*, vol. 336, no. 1, pp. 89–124, 2005.
[14] P. Atzeni, P. Cappellari, and P. A. Bernstein, "Model-Independent Schema and Data Translation," in *EDBT*, 2006, pp. 368–385.
[15] B. Alexe, L. Chiticariu, R. J. Miller, and W. Tan, "Muse: Mapping Undestanding and deSign by Example," UC Santa Cruz, Tech. Rep. UCSC-CRL-07-10, 2007.
[16] L. Popa and V. Tannen, "An Equational Chase for Path-Conjunctive Queries, Constraints, and Views," in *ICDT*, 1999, pp. 39–57.
[17] "TPC Transaction Processing Performance Council," http://tpc.org.
[18] R. J. Miller, D. Fisla, M. Huang, D. Kymlicka, F. Ku, and V. Lee, "The Amalgam schema and data integration test suite," 2001, www.cs.toronto.edu/˜ miller/amalgam.
[19] R. Hull and M. Yoshikawa, "ILOG: Declarative Creation and Manipulation of Object Identifiers," in *VLDB*, 1990, pp. 455–468.
[20] S. Melnik, P. A. Bernstein, A. Halevy, and E. Rahm, "Supporting Executable Mappings in Model Management," in *SIGMOD*, 2005, pp. 167–178.
[21] P. A. Bernstein, S. Melnik, and P. Mork, "Interactive schema translation with instance-level mappings," in *VLDB (demo)*, 2005, pp. 1283–1286.
[22] L. A. Rowe, ""Fill-in-the-Form" Programming," in *VLDB*, 1985, pp. 394–404.
[23] M. Zloof, "Query-By-Example: A Data Base Language," *IBM Sys. Journal*, vol. 16, no. 4, pp. 324–343, 1977.
[24] M. J. Carey, L. M. Haas, V. Maganty, and J. H. Williams, "PESTO : An Integrated Query/Browser for Object Databases," in *VLDB*, 1996, pp. 203–214.
[25] A. M. Silva and M. A. Melkanoff, "A Method for Helping Discover the Dependencies of a Relation," in *Adv. in Data Base Theory*, 1979, pp. 115–133.
[26] C. Beeri, M. Dowd, R. Fagin, and R. Statman, "On the Structure of Armstrong Relations for Functional Dependencies," *JACM*, vol. 31, no. 1, pp. 30–46, 1984.
[27] L. Chiticariu and W. Tan, "Debugging Schema Mappings with Routes," in *VLDB*, 2006, pp. 79–90.