

# Muse: A System for Understanding and Designing Mappings

Bogdan Alexe,  
Laura Chiticariu  
UC Santa Cruz  
abogdan@cs.ucsc.edu,  
laura@cs.ucsc.edu

Renée J. Miller  
U. of Toronto  
miller@cs.toronto.edu

Daniel Pepper,  
Wang-Chiew Tan  
UC Santa Cruz  
dpepper@ucsc.edu,  
wctan@cs.ucsc.edu

## ABSTRACT

Schema mappings are logical assertions that specify the relationships between a source and a target schema in a declarative way. The specification of such mappings is a fundamental problem in information integration. Mappings can be generated by existing mapping systems (semi-)automatically from a visual specification between two schemas. In general, the well-known 80-20 rule applies for mapping generation tools. They 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 and anecdotal evidence from mapping designers suggest that the mapping design process can be perfected by using data examples to explain the mapping and alternative mappings. We demonstrate Muse, a data example driven mapping design tool currently implemented on top of the Clio schema mapping system. Muse leverages data examples that are familiar to a designer to illustrate nuances of how a small change to a mapping specification changes its semantics. We demonstrate how Muse can differentiate between alternative mapping specifications and infer the desired mapping semantics based on the designer's actions on a short sequence of simple data examples.

### Categories and Subject Descriptors:

H.2.1 [Logical Design]: Schema and subschema, D.2.2 [Design Tools and Techniques], H.2.5 [Heterogeneous Databases]

**General Terms:** Design, Algorithms, Languages

**Keywords:** schema mappings, data exchange, data translation, data examples, design, refinement

## 1. INTRODUCTION

*Schema mappings*, or *mappings* in short, are logical assertions that specify the relationships between a source and a target schema in a declarative way. The specification of such mappings is a fundamental problem in information integration. Existing mapping systems such as Clio [6], HePToX [2], and IBM's Rational Data Architect [5], can (semi-)automatically generate mappings from a visual specification between two schemas. In general, the well-known 80-20 rule applies for mapping generation tools. They 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.

As described in [1], the mapping design process can be perfected by using data examples to explain the mapping and alternative mappings. Mapping designers usually understand their data better than they understand mapping specifications. Hence, familiar data examples could be leveraged to illustrate nuances of how a small change to a mapping specification changes its semantics.

Motivated by the observations above, we have built Muse, a data example driven mapping design tool currently implemented on top of Clio [6]. Muse can differentiate between alternative mapping specifications and infer the desired mapping semantics based on the designer's actions on a *short sequence of simple* data examples.

**Summary of demonstration features.** We demonstrate Muse-G, the component of Muse which helps a designer derive the desired grouping semantics for a mapping specification using examples. For instance, through examples, Muse-G can infer whether the designer intends to group projects by a company's name and location or only by a company's name. Grouping or combining related data together is an essential functionality of many integration systems. However, tools such as [2, 5, 6] define a *default grouping function* for every target nested set in a mapping, which can only be manually modified. This can prove to be a difficult task, if the schemas or the number of possible arguments for a grouping function are large. Indeed, if there are  $n$  possible attributes to group by, then there are  $2^n$  choices of grouping functions. Furthermore, it may not be obvious to a designer, what the  $n$  possible grouping attributes are [4, 6]. To illustrate this point, our demo will take users through example mapping scenarios, and give them an opportunity to deduce the possible grouping attributes themselves. We will demonstrate that this is not always an easy cognitive task. We demonstrate how Muse-G infers the desired grouping semantics through the actions taken by the designer on a *short sequence of simple* data examples. We also demonstrate how Muse-G exploits source schema constraints (keys and functional dependencies in general), when available, to reduce the number of examples presented to a designer. The interactive nature of the demonstration will allow us to show how the examples illustrate design alternatives in a much more natural way than having the designer deduce the proper semantics and edit the mapping specification directly. In addition, we demonstrate how Muse-G supports the incremental design of a grouping function without restarting the process from scratch.

We also demonstrate Muse-D, the component of Muse which helps a designer choose among alternative interpretations of an ambiguous mapping. Intuitively, a mapping is ambiguous if it specifies, in more than one way, how an atomic target schema element 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. We demonstrate how Muse-D can help a mapping designer understand and differentiate

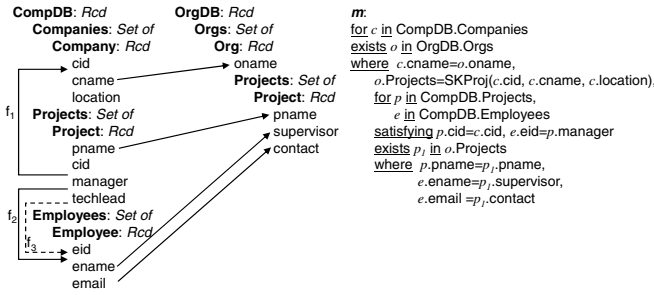


Figure 1: An example relational-to-XML schema mapping.

among all interpretations of an ambiguous mapping through a *single small* data example of the source. The designer is presented with a partial target instance derived from the source data example and asked to select among a small set of data choices. We show how the designer’s actions on these choices translate into a unique interpretation of the ambiguous mapping.

Finally, we demonstrate how Muse leverages the designer’s familiarity with the actual source instance by attempting to extract “real” examples from the source instance first. Sometimes, the actual source instance is insufficient to illustrate all design alternatives. When this happens, Muse “falls back” to its own synthetically constructed example. Moreover, we show how the visual interface of Muse is able to highlight “related” source and target tuples in a data example, thus facilitating the designer’s understanding of the mapping even further through these relationships.

After testing our system on several real life scenarios, we have found that it scales well with the complexity of the mappings. On the average, Muse takes less than a second to obtain the example instances that are presented to the mapping designer at each step. For a thorough experimental evaluation of Muse, we refer the interested reader to [1].

**Technical contribution.** Our main technical contribution is an implementation of a mapping designer with the above mentioned features. To the best of our knowledge, Muse is the first data example driven mapping design tool that implements these functionalities. While our work is largely inspired by Yan et al. [7], we go significantly beyond the techniques and space of alternative mappings supported by [7]. Our work is complementary to the work of [3] which allows a designer to understand and debug schema mappings by showing the relationships, called routes, between source or target data. In their system, the mapping designer is not guided with examples in creating or refining a schema mapping, but instead must manually change the mapping, once a problem has been identified by analyzing the routes. We refer the interested reader to [1] for complete discussion of related work.

Although we have built Muse on top of Clio, our techniques are independent of Clio and can be deployed with any mapping system based on a similar mapping formalism. Furthermore, Muse works for any combination of relational and XML source and target schemas.

## 2. DEMONSTRATION OVERVIEW

We illustrate the main features of Muse through an example mapping kept simple, for ease of exposition. We emphasize that in general, real life mappings are much more complex than our illustrative example, and thus harder to perfect. We will use a more complex schema mapping in our actual system demonstration.

**Our schema mapping example.** We use the nested relational model and nested mapping language of [4] for describing our mapping (see Figure 1). Nested mappings generalize the mapping lan-

guage of [6] which is equivalent to *source-to-target (s-t) tuple generating dependencies* or *global-local-as-view (GLAV)* assertions and widely used in research on data exchange and integration between relational schemas.

Figure 1 shows a *mapping scenario* between a relational source schema *CompDB* and a nested target schema *OrgDB*. (Ignore the dashed referential constraint  $f_3$  for now.) Such a mapping scenario is interpreted by a mapping-generation tool such as Clio [4, 6] into a set of s-t constraints that *declaratively* specify the relationship between the two schemas. The mapping  $m$  in Figure 1 is an example of one of the constraints generated. The outer part of  $m$  asserts that for every *Company* tuple  $c$  (refer to the first **for** clause of  $m$ ) there must exist an *Org* tuple  $o$  in the target (the first **exists** clause of  $m$ ) with the value for *oname* extracted from  $c.cname$  (first **where** clause of  $m$ ). Furthermore, the value of  $o.Projects$  is the SetID (also called *grouping function* or *Skolem function*)  $SKProj(c.cid, c.cname, c.location)$ . The inner part of  $m$  asserts that for every *Project* tuple  $p$  and *Employee* tuple  $e$  that together with the *Company* tuple  $c$  satisfy the referential constraints  $f_1$  and  $f_2$ , there must exist a *Project* tuple  $p_1$  within the set  $o.Projects$  with the corresponding values for *pname*, *supervisor* and *contact*.

According to [4], the default SetID or *grouping function* for *Projects* in  $m$  is  $SKProj(c.cid, c.cname, c.location)$  or  $SKProj(cid, cname, location)$ , when  $c$  is understood from the context. In other words, *Project* records are grouped according to the values of all attributes of the *Company* record. To illustrate, consider the following source and target instances satisfying the specification given by  $m$ .

Source instance:	Target instance:
<b>Companies:</b>	<b>Orgs:</b>
111 IBM Almaden	IBM
112 IBM Watson	<b>Projects:</b> SKProj(111, IBM, Almaden)
<b>Projects:</b>	DBSearch Anna anna@ibm
DBSearch 111 e14 e15	IBM
WebSearch 112 e14 e15	<b>Projects:</b> SKProj(112, IBM, Watson)
<b>Employees:</b>	WebSearch Anna anna@ibm
e14 Anna anna@ibm	

The two *Project* records in the target appear in distinct groups identified by SetIDs  $SKProj(111, IBM, Almaden)$  and  $SKProj(112, IBM, Watson)$ , respectively. If  $SKProj(cname)$  is the grouping function instead, then *Project* records are grouped according to *cname* of *Company* records. Hence, the two *Project* records would appear together under the same *Projects* record identified by  $SKProj(IBM)$ .

### 2.1 Muse-G: Designing Grouping Functions

We demonstrate the Muse-G wizard which infers a grouping function that has the same grouping semantics as the actual grouping function that the designer has in mind, through a sequence of “questions” posed to the designer. Here, we only demonstrate how Muse-G designs the grouping function  $SKProj$  of *Projects*. The algorithm behind Muse-G is described in detail in [1]. Intuitively, Muse-G examines every attribute in the set  $\{cid, cname, location\}$  of all possible arguments for  $SKProj$ . With each attribute, a *small* example source instance  $I_e$  is constructed, from which two differentiating target instances are obtained: one is the result of including the attribute as part of  $SKProj$  in  $m$ , and the other omits it.

For example, suppose *cid* is examined first. Muse-G will construct an example source instance  $I_e$  shown on top of Figure 2 and show two different target instances (Scenarios 1 and 2 in Figure 2) that are obtained by executing mappings  $m_1$  and respectively,  $m_2$  on  $I_e$ . Here,  $m_1$  and  $m_2$  are identical to  $m$  except that  $m_1$  groups *Projects* by *cid* (i.e.,  $SKProj(cid)$ ) and  $m_2$  groups all projects together, regardless of *cid*, *cname*, and *location* values (i.e.,  $SKProj()$ ). Next, Muse-G asks the designer whether scenario 1 or 2 looks correct. Assuming that the designer has the grouping function  $SKProj(cid)$  in mind, she would select Scenario 1 in Figure 2.

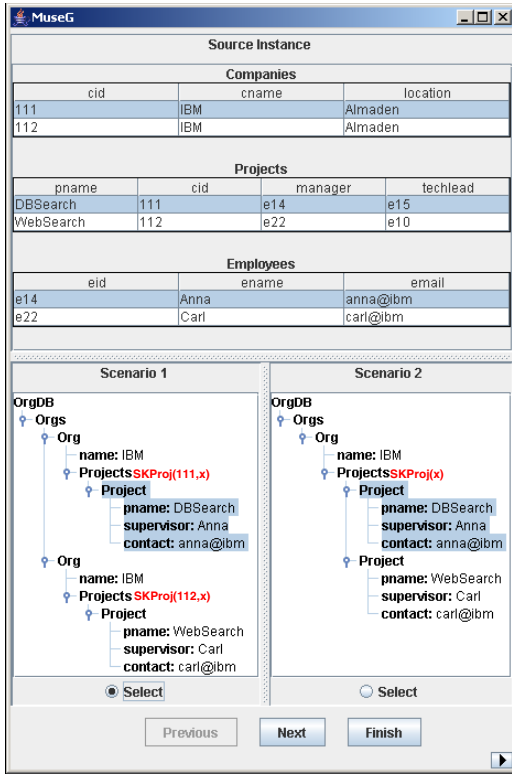


Figure 2: Screenshot of Muse-G, where  $x \subseteq \{\text{IBM, Almaden}\}$ .

Then, Muse-G repeats a similar process with the next attributes  $cname$  and  $location$  to determine whether  $cname$  and  $location$  are part of the grouping function  $SKProj$ .

Now we examine in more detail what happens when Muse-G examines each attribute in the set  $\{cid, cname, location\}$ . At each step, the goal 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 under examination as part of  $SKProjs$  in  $m$ , and the other omits it.

Suppose we examine  $cid$  first. Muse-G first constructs its own example synthetic instance  $I_{synth}$ , as shown below.

$$I_{synth}: \{Company(c_1, n_1, l_1), Project(p_1, c_1, e_1, t_1), \\ Employee(e_1, en_1, cn_1), \\ Company(c_2, n_1, l_1), Project(p_2, c_2, e_2, t_2), \\ Employee(e_2, en_2, cn_2)\}$$

Observe that each relation in  $I_{synth}$  has two tuples. Furthermore, every attribute value of every tuple is distinct, except for  $cname$  and  $location$  values of  $Company$  tuples. The reason for this is so that the target instances generated by  $m$  with  $SKProj(cid, y)$ , where  $y \subseteq \{cname, location\}$ , versus  $m$  with  $SKProj(y)$  will be non-isomorphic. Indeed, the former target instance will contain two distinct  $Project$  sets, while the latter consists of only one  $Project$  set. Next, Muse-G executes the following query against the actual source instance  $I$  in order to retrieve real tuples for the example instance  $I_e$ .

$$Q^{I_e}: Company(c_1, n_1, l_1) \wedge Company(c_2, n_1, l_1) \wedge \\ Project(p_1, c_1, e_1, t_1) \wedge Project(p_2, c_2, e_2, t_2) \wedge \\ Employee(e_1, en_1, cn_1) \wedge Employee(e_2, en_2, cn_2) \wedge c_1 \neq c_2$$

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  $I_{synth}$ , shown earlier. Alterna-

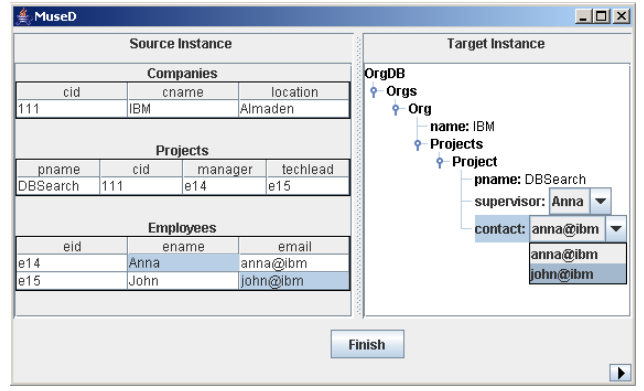


Figure 3: Screenshot of Muse-D.

tively, 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 at the top of Figure 2, 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 2, starting from  $I_e$  with mappings  $m_1$  and respectively,  $m_2$ . Here,  $m_1$  and  $m_2$  are identical to  $m$  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 result of applying  $m_1$  on  $I_e$  is isomorphic to the result of applying  $m'_1$  on  $I_e$ , where  $m'_1$  is a mapping obtained from  $m$  by replacing  $SKProjs$  with  $SKProjs(\{cid\} \cup Y)$ , where  $Y \subseteq \{cname, location\}$ . Since  $cname$  and  $location$  values are identical for the two  $Company$  tuples in  $I_e$ , the mapping  $m_1$  has the same effect as  $m'_1$  on  $I_e$ . Similarly,  $m_2$  has the same effect as  $m'_2$  on  $I_e$ , where  $m'_2$  is obtained from  $m_2$  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  $Projects$ .

**Exploiting source schema constraints.** We also demonstrate how Muse-G exploits source constraints such as keys (or functional dependencies in general) to reduce the number of examples shown to the designer. For example, if  $cid$  is the key for  $Companies$ , then for all source instances that satisfy the key, we know that a potentially rewarding sequence of attributes to examine would be  $cid$  first, followed by either  $cname$  or  $location$ . If the designer chooses Scenario 1, then Muse-G can immediately conclude  $SKProj(cid)$  without examining the rest of the attributes. As we show in [1], grouping with  $SKProj(cid)$  is the same as grouping with  $SKProj(cid, y)$ , where  $y \subseteq \{cname, location\}$ . Consequently, there is no need to examine the attributes  $cname$  and  $location$ , thereby reducing the number of examples shown to the designer by two.

**Incremental design.** We also demonstrate how grouping functions can be designed incrementally. This feature is useful for allowing a designer to return to refine her design at a later time. We show how an existing grouping function (e.g.,  $SKProj$ ) may be refined without restarting Muse-G from scratch, by choosing to “group more” (i.e., merge multiple groups into a bigger group) or “group less” (i.e., split a group into multiple smaller groups) on  $SKProj$ .

## 2.2 Muse-D: Disambiguating Mappings

We demonstrate how Muse-D can be used to select among multiple interpretations of an *ambiguous* mapping through examples. Consider the additional referential constraint  $f_3$  in Figure 1. Note that there are now two referential constraints from *Project* to *Employee*. This new mapping scenario can be interpreted in several ways, four of which are condensed into the mapping  $m_a$  shown below.

$m_a$ : for  $c$  in CompDB.Companies  
exists  $o$  in OrgDB.Orgs  
where  $c.name=o.name$ ,  $o.Projects=SKProj(c.cid, c.name, c.location)$ ,  
for  $p$  in CompDB.Projects,  
 $e_1$  in CompDB.Employees,  $e_2$  in CompDB.Employees  
satisfying  $p.cid=c.cid$ ,  $e_1.eid=p.manager$ ,  $e_2.eid=p.techlead$   
exists  $p_1$  in  $o.Projects$   
where  $p.pname=p_1.pname$ ,  
 $(e_1.ename=p_1.supervisor$  or  $e_2.ename=p_1.supervisor)$ ,  
 $(e_1.email=p_1.contact$  or  $e_2.email=p_1.contact)$

The mapping  $m_a$  is *ambiguous*: It contains or predicates to illustrate alternative interpretations. The non-bold parts are common to all four interpretations and each of the bold conjuncts represents two alternative ways of associating a *supervisor* (and *email*, respectively) with a *Project.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.

To disambiguate the four interpretations of  $m_a$ , Muse-D constructs a *single small* example source instance  $I_e$  shown in Figure 3 with the property that four pairwise distinct target instances can be generated, one from each interpretation of  $m_a$ . The designer’s selection of one of these target instances is thus a selection of one of the interpretations of  $m_a$ . In our system, Muse-D does not display all four distinct target instances. Instead, it compactly represents them in one “instance” by factoring common parts and displaying the alternatives for each ambiguous schema element according to  $m_a$  (see the target instance in Figure 3). If the designer picks the values Anna for *supervisor* and Jon@ibm for *contact*, it means that the desired mapping is one that uses “ $e_2.ename = p_1.supervisor$ ” and “ $e_1.email = p_1.contact$ ” in the where clause of  $m_a$ .

Next, we briefly illustrate the way Muse-D constructs a differentiating example source instance. Muse-D first constructs a synthetic example  $I_{synth}$  which consists of a *Company* tuple ( $c_1, cn_1, l_1$ ), a *Project* tuple ( $p_1, c_1, e_1, e_2$ ) and two *Employee* 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  $I_{synth}$  with  $I_e$ , which consists of real tuples from an existing source instance  $I$ :

$$Q^{I_e} : Company(c_1, cn_1, l_1) \wedge Project(p_1, c_1, e_1, e_2) \wedge \\ Employee(e_1, en_1, cn_1) \wedge Employee(e_2, en_2, cn_2) \wedge \\ en_1 \neq en_2 \wedge cn_1 \neq cn_2$$

All variables of  $Q^{I_e}$  are universally quantified. Since *supervisor* and *contact* 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 in Figure 3. If  $Q^{I_e}(I)$  returns an empty result, then the synthetic instance  $I_{synth}$  shown above would be presented to the designer instead.

Finally, Muse-D applies  $m_a$  on  $I_e$  to generate the target instance with “choices” shown also in Figure 3. Intuitively, the non-choice part of the target instance is generated by applying the non-ambiguous part of  $m_a$  on  $I_e$ . 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.

## 2.3 Constructing and Visualizing Examples

We demonstrate how Muse extracts, whenever possible, example instances from a real source instance provided by the designer, falling back to “artificial” examples only if a real example is not found. The example instances are obtained by executing appropriate SQL/XQuery queries on the real source instance. Furthermore, the time that the designer has to wait for a real example from Muse is minimized as much as possible: Muse exploits the “think” time of the designer on the current example, to precompute other examples ahead of time in the background.

We also demonstrate how the visual interface facilitates the designer’s understanding of nuances in the mappings even further, by automatically highlighting how source and target data values are related via mappings in an example. Consider the example shown in Figure 2. When the designer selects the *Project* tuple under SKProj(111,x) in Scenario 1 (i.e., the DBSearch tuple), the source tuples that are related to the DBSearch tuple by the underlying mapping are highlighted. In addition, the DBSearch tuple is also highlighted in Scenario 2 (under SKProj(x)) because the same source tuples contribute to it. However, the DBSearch tuple is not in the same group as the WebSearch tuple in Scenario 1, but they are in the same group in Scenario 2. The highlighting thus illustrates the difference in the semantics of  $m$  with grouping functions SKProj(cid,y) and respectively, SKProj(y). Similarly, in Figure 3 the values currently picked by the designer in the target instance are automatically highlighted by Muse in the example source instance, indicating where the values are extracted from.

## 3. SYSTEM ARCHITECTURE

Muse is implemented in Java 5 on top of the Clio system. Clio presents the designer with a list of ambiguous mappings, if any, and Muse-D is used in disambiguating them. Once a mapping is disambiguated, Muse-G is employed in designing its grouping functions.

## 4. CONCLUSION

We have described the functionalities we demonstrate for 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).

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

## 5. REFERENCES

- [1] B. Alexe, L. Chiticariu, R. J. Miller, and W. Tan. Muse: Mapping Understanding and deSign by Example. In *ICDE*, 2008.
- [2] A. Bonifati, E. Q. Chang, T. Ho, and L. V. S. Lakshmanan. HepToX: Heterogeneous Peer to Peer XML Databases, 2005.
- [3] L. Chiticariu and W. Tan. Debugging Schema Mappings with Routes. In *VLDB*, pages 79–90, 2006.
- [4] A. Fuxman, M. A. Hernández, H. Ho, R. J. Miller, P. Papotti, and L. Popa. Nested Mappings: Schema Mapping Reloaded. In *VLDB*, pages 67–78, 2006.
- [5] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio Grows Up: From Research Prototype to Industrial Tool. In *SIGMOD*, pages 805–810, 2005.
- [6] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating Web Data. In *VLDB*, pages 598–609, 2002.
- [7] L. Yan, R. Miller, L. Haas, and R. Fagin. Data-Driven Understanding and Refinement of Schema Mappings. In *SIGMOD*, pages 485–496, 2001.