Clio: Schema Mapping Creation and Data Exchange

Ronald Fagin[1], Laura M. Haas[1], Mauricio Hernández[1],
Renée J. Miller[2], Lucian Popa[1], and Yannis Velegrakis[3]

[1] IBM Almaden Research Center, San Jose, CA 95120, USA
[2] University of Toronto, Toronto ON M5S2E4, Canada
[3] University of Trento, 38100 Trento, Italy
fagin@almaden.ibm.com, laura@almaden.ibm.com, mauricio@almaden.ibm.com,
miller@cs.toronto.edu, lucian@almaden.ibm.com, velgias@disi.unitn.eu

Abstract. The Clio project provides tools that vastly simplify information integration. Information integration requires data conversions to bring data in different representations into a common form. Key contributions of Clio are the definition of non-procedural *schema mappings* to describe the relationship between data in heterogeneous schemas, a new paradigm in which we view the mapping creation process as one of query discovery, and algorithms for automatically generating queries for data transformation from the mappings. Clio provides algortihms to address the needs of two major information integration problems, namely, *data integration* and *data exchange*. In this chapter, we present our algorithms for both schema mapping creation via query discovery, and for query generation for data exchange. These algorithms can be used in pure relational, pure XML, nested relational, or mixed relational and nested contexts.

1 Introduction

We present a retrospective on key contributions of the Clio project, a joint project between the IBM Almaden Research Center and the University of Toronto begun in 1999. Clio's goal is to radically simplify information integration, by providing tools that help in automating and managing one challenging piece of that problem: the conversion of data between representations. Clio pioneered the use of *schema mappings*, specifications that describe the relationship between data in two heterogeneous schemas. From this high-level, non-procedural representation, it can automatically generate either a view, to reformulate queries against one schema into queries on another for *data integration*, or code, to transform data from one representation to the other for *data exchange*. In this chapter, we focus on two key components of Clio: the creation of mappings between heterogeneous schemas, and their use for the implementation of data exchange.

1.1 Schema Mapping

Schema mappings are fundamental for a number of important information integration problems [9] including data integration, data exchange, peer-to-peer data sharing, schema integration and schema evolution. Applications are typically limited to handling information with a specific schema, so they rely on systems that can create and use mappings to transform data from one representation to another.

A fundamental requirement for Clio is that it make no assumption about the relationship between the schemas or how they were created. In particular, we do not assume that either of the schemas is a global or mediator schema, nor that one schema is a view (global or otherwise) over the other. This implies that both schemas may contain data not represented in the other, and that both may have their own constraints.

This requirement to map independently created schemas has a strong impact on our mapping language, as we need one that is more general than those used in traditional schema integration [6] or in mediator systems such as TSIMMIS [16] or Information Manifold [34].

A second requirement is that we be able to map between relational schemas and nested schemas (for example, XML schemas). As XML emerged as a common standard for exchanging data, an early motivating application for our work was publishing legacy relational data in XML. This often requires relational data to be placed into a predefined XML schema (defined, e.g., by a standards committee to permit meaningful exchange within a specific domain). However, for other applications including schema evolution, data warehousing, and data federation, we also need to be able to map data between different relational schemas and between any combination of nested and relational schemas.

A third requirement is that we be able to create and use mappings at different levels of granularity. For some applications and some schemas, it may be sufficient to create fine-grained mappings between individual components (for example, between attributes or elements to translate gross salary in francs to net salary in dollars). For others, mappings between broader concepts are required (for example, between the order concept in one billing application with that used by another). And for other applications, we may need to map full documents (for example, map every company's carbon emission data expressed in a schema suitable for the European Union Emission Trading Scheme to a schema designed for the Portable Emissions Measurement Systems standard).

Finally, we want our mapping creation algorithms to be incremental. There are many motivations for this. First, for many tasks, complete mapping of one entire schema to another is not the goal. It may suffice to map a single concept to achieve the desired interoperability. Second, we want a tool that gives users (or systems) with only partial knowledge of the schemas, or limited resources, useful (and usable) mappings despite their incomplete knowledge or resources. We hope that incomplete mappings can help them in understanding the schemas and data better, and that the mappings can be refined over time as need arises, for example, as new data appears, or the application needs change. This particular

aspect of our approach was explored in more detail in our work on data-driven mapping refinement [51] and in work on mapping debugging [3], but will not be emphasized in this chapter. This ability to evolve mappings incrementally has more recently been coined *pay-as-you-go* [23].

Clio mappings assume that we are given two schemas and that we would like to map data from the first to the second. We refer to the first schema as a *source* schema, and the second as a *target schema*. In practice, this meets most application needs, as most require only a uni-directional flow of data. For example, one common use of mappings is in query reformulation, commonly referred to as *data integration* [33], where queries on a target schema are reformulated, using the mappings, into queries on a source schema. For applications requiring bi-directional mapping, mappings are created in both directions.

1.2 Implementing Data Exchange

Another common use of mappings is for *data exchange* where the goal is to create a target instance that reflects the source instance as accurately as possible [19]. Since the target is materialized, queries on the target schema can be answered directly without query reformulation. At the time we started Clio, data integration had been widely studied, but work on data exchange was quite dated. Foundational systems like Express [46, 47] did data exchange for mappings which were much less expressive than those needed to map arbitrary schemas. There were no systems that performed data exchange for the general mappings we strove to create.

For independent schemas, because the schemas may represent overlapping but distinct sets of concepts, a schema mapping may relate a source instance with many *possible* target instances. As a result, we have a fundamentally new problem: given a schema mapping, determine which possible target instance is the best one to use for data exchange. At the time of our first results [38, 43], this problem had not yet been formalized. Hence, in Clio, we made some intuitive decisions that were later formalized into a theory for data exchange [19]. In this chapter, we discuss this intuition, and how it corresponds to the later theory. We also discuss some important systems issues not covered by the theory. Specifically, we consider how to create a *data exchange program* from a schema mapping. Due to the requirements for schema mapping laid out above, we choose to produce executable queries for data exchange. A schema mapping is a declarative specification of how an instance of a source schema corresponds to possibly (infinitely) many target instances, and from this we choose a best target instance to materialize. Hence, our data exchange queries, when executed on a source instance, will generate this one chosen target instance.

The origins of this chapter first appeared in Miller et al. [38] and Popa et al. [43]. This chapter also includes details, originally from Velegrakis [48], of our data exchange implementation. The requirements outlined above force us to accommodate various runtime environments. In this chapter, we discuss how to generate data exchange queries in SQL, XQuery or XSLT.

The chapter is organized as follows. Section 2 introduces a motivating example and describes the problem of schema mapping generation and the problem of data exchange. Section 3 presents the data model we will use for representing both relational and nested relational schemas along with our schema mapping formalism. Section 4 presents our algorithm for generating schema mappings. Section 5 presents our algorithm for data exchange (mapping code generation). We present a discussion and analysis of our algorithms in Section 6, describe the related work in Section 7 and then conclude.

2 A Motivating Example

To motivate our approach, we first walk through an example explaining the intuition behind our mapping creation algorithm, and highlighting some of its features. We then extend our example to illustrate how schema mappings can be used to generate queries for data exchange, and describe the key innovations in that algorithm.

2.1 Schema Mapping Creation

Schema. Consider a data source with information about companies and grants. The structure of its data is described by the Schema `S`, illustrated in Figure 1. It is a relational schema containing three tables, `companies`, `grants`, and `contacts`, presented in a nested relational representation that we use to model both relational and XML schemas. It contains a set of grants (`grants`), each consisting of a grant identifier (`gid`), a recipient (`recipient`), its amount (`amount`), its supervisor (`supervisor`) and its manager (`manager`). The `recipient` is actually the name of the company that received the grant. For each company, the database stores its name (`name`), address (`address`) and the year it was founded (`year`). Similarly, the supervisor and manager are references to some contact information, which consists of an identifier (`cid`), an email (`email`) and a phone number (`phone`). The curved lines f_1, f_2 and f_3 in the figure represent referential constraints specified as part of the schema. For example, f_1 may be a foreign key, or simply an inclusion dependency, stating that values in `grants.recipient` must also appear in `companies.name`.

Consider a second schema T, as illustrated on the right-hand side of Figure 1. It records the funding (`fundings`) that an organization (`organizations`) receives, nested within the organization record. The amount of each funding (`budget`) is kept in the `finances` record along with a contact phone number (`phone`). The target may be an XML schema containing a referential constraint in the form of a *keyref* definition (f_4).

Correspondences. To begin to understand the relationship between the schemas, we may invoke a schema matcher to generate a set of element correspondences (or matchings). Alternatively, we could ask a user (for example, a data designer or administrator familiar with the schemas) to draw lines between

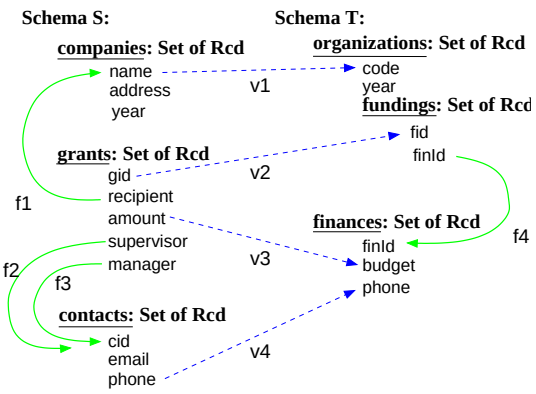

Fig. 1. A source and a target schema in a mapping scenario

elements that should contain related data. In our example, the dashed arrows between the elements of the two schemas in Figure 1 represent a set of matchings or correspondences. The line v_1 indicates (informally) that what is called a company `name` in the first schema, is referred to as an organization `code` in the second. In contrast, both schemas have an element `year`, but the data administrator (or matching tool) has specified no arrow between them. That may be because there is reason to believe that these elements do not represent the same concept. For instance, element `year` in the first schema may represent the time the company was founded, while in the second it may represent the time the company had its initial public offer (IPO).

Our approach is agnostic to how correspondences are created, whether manually or (semi-)automatically, but is cognizant that matchings are often incomplete, and sometimes incorrect. Hence, we have developed techniques for incrementally modifying mappings as correspondences change [49]. To keep our example simple, we assume that correspondences v_1, v_2, v_3, v_4 are correct.

Mappings. One way a correspondence can be interpreted is that the target schema element should be populated with values from the source schema element. This can be formally expressed using an inter-schema inclusion dependency or more generally through a source-to-target *tuple generating dependency*, (*tgd*) [7]. A tgd representing correspondence v_1 of Figure 1 is shown below (the nested set `fundings` which is represented by the variable F inside `organizations` will be explained below).

$$\forall \underline{n}, d, y, \texttt{companies}(\underline{n}, d, y) \;\rightarrow\; \exists y', F\; \texttt{organizations}(\underline{n}, y', F) \qquad (1)$$

This simple mapping states that for each `companies` tuple, there must be an `organizations` tuple whose code is the same as the `companies.name`; this is represented by the shared variable n, which carries source data to the target. As a convention, when writing tgds, we underline all the variables that appear in both

the left-hand side and the right-hand side of the implication. This target tuple must have a value for the `year` attribute, but the mapping does not specify what this value should be (this is represented by the existential variable y'). Similarly, we could write simple tgds to express the other three correspondences.

If our application only requires information about organizations (and not about `fundings` or `finances`), then we can stop the mapping generation here. This simple element-to-element mapping can be used for data exchange or for data integration [52]. However, if the user is interested in mapping more data, we can continue the mapping generation using the other correspondences from Figure 1 to map additional concepts.

Associations. Correspondences alone do not specify how individual data values should be connected in the target. For example, in the target, funding information is nested inside organizations. This nesting indicates that there is a semantic association between organization and funding records. This may represent organizations and the funding that the organization has received, or possibly organizations and the funding that they have given to others. The semantics is not specified precisely in the schema, but it is clear that some real-world association is being represented. Given this, we will look for associations between organization information and funding information in the source to see if one of these can be used to associate data in the target. For our example, `organizations.code` corresponds to `companies.name`, while `fundings.fid` corresponds to `grants.gid`. Hence, our algorithm will search for associations between these source elements within the source schema. In our example, the referential constraint f_1 indicates that each grant is associated with a company, thus this constraint can be used to associate each company with a set of grants. In general, there may be many ways to associate elements within a schema. Our algorithm will use logical inference to find all associations represented by referential constraints and a schema's relational and nesting structure.

For our example, we have only one way of associating company names and grant gids in the source, so we will use this association to associate fundings with organizations in the target. A mapping reflecting this association is represented by the following formula.

$$\forall \underline{n}, d, y, \underline{g}, a, s, m \ \texttt{companies}(\underline{n}, d, y), \texttt{grants}(\underline{g}, \underline{n}, a, s, m) \ \rightarrow$$

$$\exists y', \mathrm{F}, f \ \texttt{organizations}(\underline{n}, y', \mathrm{F}), \ \mathrm{F}(\underline{g}, f) \qquad (2)$$

The variable F in formula (2) does not represent an atomic value, but rather a set identifier, and is also used as a term in the formula. This variable represents the set of `fundings` that an `organizations` tuple has.

Notice that Mapping (2) specifies what must be true of the target data, given that a certain pattern holds in the source data. In this example, it says that if a grant joins with a company in the source, then there must be an organization in the target with the name of the company as its code, and with a fundings record nested inside of the organization record that has the grant's gid as its `fundings.fid`. No other constraints are placed on what is in this set. So the

mapping is specifying that the association between grants and companies should be preserved in the target.

Now let us consider the correspondence v_3. Considered in isolation, this correspondence could be represented by the following mapping.

$$\forall g, r, \underline{a}, s, m \ \texttt{grants}(g, r, \underline{a}, s, m) \ \rightarrow \ \exists f, p \ \texttt{finances}(f, \underline{a}, p) \tag{3}$$

However, this mapping does not recognize that grant amounts are associated with specific grant gids (in the source) and that `fundings.fid` and `finances.budget` are associated in the target (through the referential constraint f_4). If these two associations represent the same semantic association, then a better mapping can be constructed by using the source and target associations.

$$\forall \underline{n}, d, y, \underline{g}, \underline{a}, s, m \ \texttt{companies}(\underline{n}, d, y), \texttt{grants}(\underline{g}, \underline{n}, \underline{a}, s, m) \ \rightarrow$$

$$\exists y', \mathrm{F}, f, p \ \texttt{organizations}(\underline{n}, y', \mathrm{F}), \ \mathrm{F}(\underline{g}, f), \texttt{finances}(f, \underline{a}, p), \tag{4}$$

Notice that a company and grant tuple that join in the source will create three tuples in the target: an `organizations` tuple, a `fundings` tuple (which is nested inside the `organizations` tuple), and a `finances` tuple. The mapping specifies that the `fundings` and `finances` tuples must share the same value (f) in their `finId` attributes. It does not, however, specify what this value must be (that is, the variable f is existentially quantified and is not bound to source data).

Now to complete our example, let us consider the final correspondence v_4. In the target, a phone is associated with a budget because they are represented in the same relation `finances`. In the source, there are two ways to associate a `grants.amount` (the source for `finances.budget`) and a `contacts.phone` (the source for `finances.phone`). These are represented by the two referential constraints f_2 (which associates a grant with its supervisor's phone) and f_3 (which associates a grant with its manager's phone).

It is not clear which, if either, of these associations should be used to create `finances` tuples. Clio will create two mappings, one using f_2 (Mapping (5)) which uses a join on `supervisor`, and one using f_3 (Mapping (6)) which uses a join on `manager`. To help a user decide which mapping to use, Clio provides a data viewer which allows users to see (and compare) sample target data created by each mapping [51].

$$\forall \underline{n}, d, y, \underline{g}, \underline{a}, s, m, e, \underline{p} \ \texttt{companies}(\underline{n}, d, y), \texttt{grants}(\underline{g}, \underline{n}, \underline{a}, s, m), \texttt{contacts}(s, e, \underline{p})$$

$$\rightarrow \ \exists y', \mathrm{F}, f \ \texttt{organizations}(\underline{n}, y', \mathrm{F}), \ \mathrm{F}(\underline{g}, f), \texttt{finances}(f, \underline{a}, \underline{p}) \tag{5}$$

$$\forall \underline{n}, d, y, \underline{g}, \underline{a}, s, m, e, \underline{p} \ \texttt{companies}(\underline{n}, d, y), \texttt{grants}(\underline{g}, \underline{n}, \underline{a}, s, m), \texttt{contacts}(m, e, \underline{p})$$

$$\rightarrow \ \exists y', \mathrm{F}, f \ \texttt{organizations}(\underline{n}, y', \mathrm{F}), \ \mathrm{F}(\underline{g}, f), \texttt{finances}(f, \underline{a}, \underline{p}) \tag{6}$$

We have illustrated a few of the issues involved with generating schema mappings. We now highlight some of the features of the Clio mapping algorithm.

Mapping Formalism. As our example illustrated, we use source-to-target tgds, a generalization of the relational tgds of Fagin et al. [19] to the nested relational model, to represent schema mappings. Our mappings are a form of what has been called sound GLAV (global-and-local-as-view) mappings [33]. In general, a GLAV mapping asserts a relationship between a query over the source and a query over the target. We use sound mappings, where the relationship between queries is a containment relationship (the result of the source query is contained in the target query) as is common in data integration. Such mappings do not restrict what data can be in the target; hence, we have the freedom to map multiple sources into a single target.

Mapping Generation. Clio exploits the schema and its constraints to generate a set of alternative mappings. Our approach uses the chase technique [36] to generate all possible associations between source elements (and all possible associations between target elements). The mappings are formed using these associations or other associations given by a user.

Multiple Mappings. As we create mappings, each query may omit information that the user may have wanted included. Consider Mapping (2). This mapping takes grants that have an associated company and creates target data from this information. Notice, however, that companies that have no grants would not be included in the mapping. We may want to include all companies, not just companies with grants, in the target. To allow for this, Clio generates a set of mappings that would map all source data (in this example, both companies with and without grants). A user can then choose among these mappings. If she only wishes to map a subset of the source data, she can do so by selecting an appropriate subset of the mappings Clio generates.

2.2 Query Generation for Data Exchange

For data exchange, Clio can generate code that, given a source instance, will produce an instance of the target schema that satisfies the mapping and that represents the source data as accurately as possible. In general, given a source instance there may be many target instances that satisfy the mapping (or many *solutions* in data exchange parlance [19]). Hence, to perform data exchange, we must choose a single "best" target instance, i.e., a single *solution* to materialize.

Let us assume, for example, that the instance of the source schema of Figure 1 is the one illustrated in Figure 2, and that a user has indicated that Mapping (5) is correct and would like to create a target instance. For each `companies` tuple that joins with `grants` and `contacts`, the mapping indicates that there should be an `organizations` tuple containing as organization `code` the company `name`. Furthermore, there must also be a nested `fundings` element containing the `gid` of the grant from the source. Finally, a `finances` element must also exist with

Companies

name	address	year
MS	Redmond, SA	1975
AT&T	Dallas, TX	1983
IBM	Armonk, NY	1911

Grants

gid	recipient	amount	supervisor	manager
g1	MS	1M	Rice	Gates
g2	MS	2M	Bama	Gates
g4	AT&T	3M	Greer	Dorman

Contacts

cid	email	phone
Rice	rice@microsoft	7062838
Gates	gates@microsoft	7069273
Bama	bama@microsoft	7066252
Greer	rxga@att	3607270
Dorman	dorman@att	3600102

Fig. 2. An instance for the source schema in Figure 1

the same value for its `finId` as the value of `finId` in this `fundings` record. Moreover, the `finances` element must contain the grant amount in the `budget` element and the supervisor phone number as `phone`. An instance that satisfies the mapping, i.e., a solution, can be seen in Figure 3. In this solution, `fundings` tuple `g1` can correctly be joined with the first `finances` tuple to associate it with its correct budget (`1M`). However, in Figure 3 all `fundings` tuples are associated with all `finances` tuples, which was not true in the source. In data exchange, we would like the target instance to represent *only* the data associations in the source. Clearly the instance of Figure 3 does not fullfill that desire. Furthermore, the last tuple in the Finances table does not correspond to any source data, yet its inclusion in the instance does not violate Mapping (5). So while this instance is a solution, it is not minimal.

Fagin et al. [19] proposed the use of universal solutions for data exchange. A *universal solution* is an instance of the target schema that contains no more and no less than what the mapping specification requires. A universal solution for Mapping (5) is given in Figure 4. Note that the values of the `finId` attribute have been created to associate each `fundings` with *all* and *only* the `finances` tuples that contain the correct budget amount. The instance is also minimal in that it does not contain any tuples that were not required to be in the target. To compute a universal solution, Fagin et al. [19] present an algorithm based on the chase [36]. However, in Clio, we use queries to perform the data exchange. Here we present algorithms that, given a schema mapping, generate a set of executable queries to perform the desired data exchange. The queries can be in SQL, XQuery or XSLT depending on the desired runtime environment. In the case of a pure relational setting (relational source and target schemas), these queries generate a universal solution. In our algorithm, we make use of Skolem functions (one-to-one functions) that generate values based on a set of source values. We will discuss later how to determine the correct arguments for these

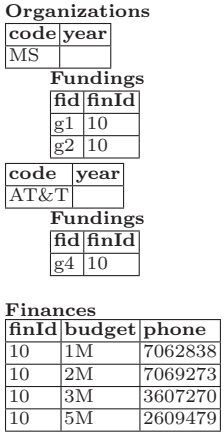

Fig. 3. A non-universal solution target instance for Mapping (5)

Skolem functions and when one Skolem function should be reused in different schema elements. For example, in Section 5, we show why the Skolem function for `finId` needs to depend on four source attributes (`name`, `gid`, `amount`, and `phone`).

In the case of a nested target schema, Clio applies additional grouping and nesting to produce a target instance that is in PNF (Partitioned Normal Form) [1]. This is done to minimize the redundancy in the target instance. Consider the universal solution given in Figure 4 which contains a different `organizations` tuple with a singleton `fundings` set for each `companies` and `grants` pair in the source, even if multiple pairs share the same company name. Clio avoids this redundancy, by producing a single `organizations` tuple for each source `name` and grouping all the `fundings` that belong to the same organization together under one single organization element (Figure 5). As shown in the figure, we use Skolem functions to represent set identifiers for each `fundings` set. Our algorithms determine the correct arguments for these Skolem functions to achieve PNF grouping. In more recent work which will not be covered in this chapter [24], we have considered how to declaratively represent PNF grouping semantics in the mapping specification along with other types of grouping. In this chapter, we will assume PNF is the desired semantics, and we present our solutions for generating PNF target instances.

There are two main innovations in our data exchange algorithm. The first is a new technique for generating Skolem terms to represent existential values and for achieving grouping in the target instance. Second, our algorithm can identify and merge data that are generated by different mappings, but represent the same target entities. Assume, for instance, that the user wanted to populate the target with all the companies, independently of whether they have funding or not. Mapping (5) can generate only companies with grants (i.e., funding), due to the join it performs on the source data. Mapping (1) on the other hand, generates all

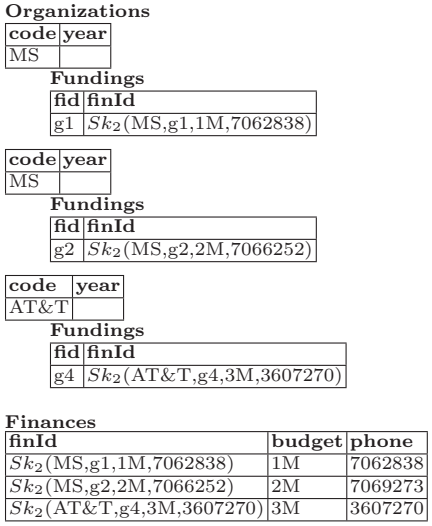

Fig. 4. A universal solution target instance for Mapping (5)

the companies, but without their potential funding. The desired target instance can be achieved by using both mappings (5) and (1). The resulting instance would be the same as Figure 5 but with an additional `organizations` element for IBM having an empty `fundings` subelement. The MS and AT&T tuples would not be impacted, even though they are produced by both mappings.

3 Mapping Language and Schema Constraints

Schemas and Types. We use a nested relational model to model both relational and XML Schemas. In general, a *schema* is a sequence of labels (called roots), each with an associated *type* τ, defined by the grammar:

$\tau ::= \mathsf{String} \mid \mathsf{Integer} \mid \mathsf{Set\ of}\ \tau \mid \mathsf{Rcd}[l_1 : \tau_1, \ldots, l_n : \tau_n] \mid \mathsf{Choice}[l_1 : \tau_1, \ldots, l_n : \tau_n]$

Types Integer and String are called atomic types, Set of is a set type, and Rcd and Choice are complex types. With respect to XML Schema, we use Set of to model repeatable elements (or repeatable groups of elements), while Rcd and Choice are used to represent the "all" and "choice" *model-groups*. For each set type Set of τ, τ must be an atomic (String or Integer) type or a Rcd type. We do not consider order, that is, Set of represents unordered sets. "Sequence" model-groups of XML Schema are also represented as (unordered) Rcd types.

Instances. An instance of a schema associates each schema root l of type τ with a value v of type τ. For the atomic types, the allowed values are the expected ones (i.e., strings, integers). A value of type $\mathsf{Rcd}[l_1 : \tau_1, \ldots, l_n : \tau_n]$ is an unordered

Organizations $Sk_0()$

code	year
MS	

Fundings $Sk_1(MS)$

fid	finId
g1	Sk_2(MS,g1,1M,7062838)
g2	Sk_2(MS,g2,2M,7066252)

code	year
AT&T	

Fundings $Sk_1(AT\&T)$

fid	finId
g4	Sk_2(AT&T,g4,3M,3607270)

Finances $Sk3()$

finId	budget	phone
Sk_2(MS,g1,1M,7062838)	1M	7062838
Sk_2(MS,g2,2M,7066252)	2M	7069273
Sk_2(AT&T,g4,3M,3607270)	3M	3607270

Fig. 5. A target instances for Mapping (5) in PNF

tuple of pairs $[l_1 : v_1, \ldots, l_n : v_n]$ where v_i is a value of type τ_i with $1 \leq i \leq n$. A value of type $\mathsf{Choice}[l_1 : \tau_1, \ldots, l_n : \tau_n]$ on the other hand, is a pair $\langle l_i : v_i \rangle$ where v_i is a value of type τ_i with $1 \leq i \leq n$. With respect to XML, the labels $l_1, \ldots, l_n$ model element names or attribute names, while the values $v_1, \ldots, v_n$ represent the associated contents or value. In our model, we do not distinguish between XML elements and attributes.

A value of type $\mathsf{Set\ of}\ \tau$ is actually an identifier (*set ID*). This identifier is associated to an unordered set $\{v_1, v_2, \ldots, v_n\}$ of values, each being of type τ.[4] This id-based representation of sets faithfully captures the graph-based data models of XML. In a constraint or expression in general, we shall always interpret the equality of two expressions of set type to mean the equality of their set ids.

Mapping Language. Our mapping language is based on the source-to-target tgds [19], extended to support nested relations. When restricted to the relational model, our mapping formulas coincide with source-to-target tgds. However, we permit variables occurring inside atoms in a tgd to represent relations as well as atomic values to allow the representation of nested sets.

In this paper, as in other Clio work [43, 49, 52], we will use a more verbose form of the mapping language to make the algorithms and ideas easier to follow. Let an expression be defined by the grammar $e ::= S \mid x \mid e.l$, where x is a variable, S is a schema root, l is a label, and $e.l$ is record projection. Then a *mapping* is a statement (constraint) of the form:

$$\begin{aligned} \mathcal{M} ::= \ & \underline{\mathsf{foreach}}\ \ x_1\ \underline{\mathsf{in}}\ g_1, \ldots, x_n\ \underline{\mathsf{in}}\ g_n \\ & \quad \underline{\mathsf{where}}\ \ B_1 \\ & \underline{\mathsf{exists}}\ \ \ y_1\ \underline{\mathsf{in}}\ g'_1, \ldots, y_m\ \underline{\mathsf{in}}\ g'_m \\ & \quad \underline{\mathsf{where}}\ \ B_2 \end{aligned}$$

[4] The reason will become clear when we talk about how data generated by different mappings is merged to form one single instance of the target schema.

$$\underline{\text{with}} \quad e_1 = e'_1 \ \underline{\text{and}} \ \ldots \underline{\text{and}} \ e_k = e'_k$$

where each x_i $\underline{\text{in}}$ g_i (y_j $\underline{\text{in}}$ g'_j) is called a *generator* and each g_i (g'_j) is one of the following two cases:

1. An expression e with type Set of τ; in this case, the variable x_i (y_j) binds to individual elements of the set e.
2. A choice selection $e \rightarrow l$ (where e is an expression with a type Choice $[\ldots, l : \tau, \ldots]$) representing the selection of attribute l of the expression e; in this case, the variable x_i (y_j) will bind to the element of type τ under the choice l of e.

The mapping is well-formed if the variable (if any) used in g_i (g'_j) is defined by a previous generator within the same clause. Every schema root used in the foreach must be a root of the source schema. Similarly, every schema root used in the exists clause must be a target schema root. The two where clauses (B_1 and B_2) are conjunctions of equalities between expressions over $x_1, \ldots, x_n$, or $y_1, \ldots, y_m$, respectively. They can also include equalities with constants (i.e., selections). Finally, each equality $e_i = e'_i$ in the with clause involves a source expression e_i (over $x_1, \ldots, x_n$) and a target expression e'_i (over $y_1, \ldots, y_m$), with the requirement that e_i and e'_i are of the same atomic type.

For simplicity of presentation, we shall ignore for the remainder of this paper the Choice types and their associated expressions. We note, however, that these are an important part of XML Schema and XML mappings, and they are fully supported in the Clio system.

Example 1. Recall the schemas in Figure 1. The following mapping is an interpretation of the correspondences v_1, v_2, v_3, given the structure of the schemas and the constraints. It is equivalent to the tgd for Mapping (4) in Section 2.

$$\begin{array}{ll}
\underline{\text{foreach}} & c \ \underline{\text{in}} \ \texttt{companies}, g \ \underline{\text{in}} \ \texttt{grants} \\
\quad \underline{\text{where}} & c.\texttt{name} = g.\texttt{recipient} \\
\underline{\text{exists}} & o \ \underline{\text{in}} \ \texttt{organizations}, f \ \underline{\text{in}} \ \texttt{fundings}, f' \ \underline{\text{in}} \ \texttt{finances} \\
\quad \underline{\text{where}} & f.\texttt{finId} = f'.\texttt{finId} \\
\underline{\text{with}} & c.\texttt{name} = o.\texttt{code} \ \underline{\text{and}} \ g.\texttt{gid} = f.\texttt{fid} \ \underline{\text{and}} \ g.\texttt{amount} = f'.\texttt{budget}
\end{array}$$

Each mapping is, essentially, a source-to-target constraint of the form $Q^S \rightsquigarrow Q^T$, where Q^S (the foreach clause and its associated where clause) is a query over the source and Q^T (the exists clause and its associated where clause) is a query over the target. The mapping specifies a containment assertion: for each tuple returned by Q^S, there must exist a corresponding tuple in Q^T. The with clause makes explicit how the values selected by Q^S relate to the values in Q^T.

Schema Constraints (NRIs). Before defining the schema constraints that we use in our mapping generation algorithm, we need to introduce *primary paths* and *relative paths*.

Definition 1. *A* primary path *with respect to a schema root R is a well-formed sequence P of generators x_1 in $g_1, \ldots, x_n$ in g_n where the first generator g_1 is an expression that depends only on R and where each generator g_i with $i \geq 2$ is an expression that depends only on the variable x_{i-1}. Given a variable x of type τ, a* relative path *with respect to x is a well-formed sequence $P(x)$ of generators x_1 in $g_1, \ldots, x_n$ in g_n where the first generator g_1 is an expression that depends only on x and where each generator g_i with $i \geq 2$ is an expression that depends only on x_{i-1}.*

The following are examples of primary paths and relative paths:

$$\begin{aligned} P_1^S: &\quad c \text{ in } \texttt{companies} \\ P_1^T: &\quad o \text{ in } \texttt{organizations} \\ P_2^T: &\quad o \text{ in } \texttt{organizations}, f \text{ in } o.\texttt{fundings} \\ P_r^T: &\quad f \text{ in } o.\texttt{fundings} \end{aligned}$$

The first two paths, P_1^S and P_1^T, are primary paths corresponding to top-level tables in the two schemas in our example. The third one is a primary path corresponding to the nested set of `fundings` under the top-level `organizations`. Finally, the fourth one is a relative path with respect to $o : \tau$ where τ is the record type under `organizations`.

Primary paths will play an important role in the algorithm for mapping generation, as they will be the building blocks for larger associations between data elements. Primary paths, together with relative paths, are also useful in defininig the schema constraints that we support in our nested relational model. Schema constraints use a similar formalism as mappings, but they are defined within a single schema. Hence, variables in both the foreach and the exists clauses are defined on the same schema.

Definition 2. *A* nested referential integrity constraint (NRI) *on a schema is an expression of the form*

$$\text{foreach } P_1 \text{ exists } P_2 \text{ where } B,$$

with the following requirements: (1) P_1 is a primary path of the schema, (2) P_2 is either a primary path of the schema or a relative path with respect to one of the variables of P_1, and (3) B is a conjunction of equalities of the form $e_1 = e_2$ where e_1 is an expression depending on one of the variables of P_1 and e_2 is an expression depending on one of the variables of P_2.

As an example, the following NRI captures the constraint f_4 informally described in Section 2:

$$\begin{aligned} &\text{foreach} \quad o \text{ in } \texttt{organizations}, f \text{ in } o.\texttt{fundings} \\ &\text{exists} \quad f' \text{ in } \texttt{finances} \\ &\quad \text{where} \quad f.\texttt{finId} = f'.\texttt{finId} \end{aligned}$$

Note that NRIs, which capture relational foreign key and referential constraints as well as XML keyref constraints, are a special case of (intra-schema) tgds and moreover such that the foreach and the exists clauses form paths. In general, the source-to-target tgds expressing our mappings may *join* paths in various ways, by using equalities in the where clause on the source and/or the target side. As an example, the mapping in Example 1 in this section joins the primary path for `companies` with the primary path for `grants` in the source, and joins the primary path for `fundings` with the primary path for `finances` in the target. The next section is focused on deriving such mappings.

4 Schema Mapping

Our mapping generation algorithm makes use of associations between atomic elements within the source and target schemas. To describe our approach, we first present an algorithm for finding natural associations within a schema. We then present an algorithm that given a set of correspondences, and a set of source and target associations, creates a set of schema mappings.

4.1 Associations

We begin by considering how the atomic elements within a schema can be related to each other. We define the notion of *association* to describe a set of associated atomic type schema elements. Formally, an association is a form of query (with no return or select clause) over a single schema; intuitively, all the atomic type elements reachable from the query variables (without using additional generators) are considered to be "associated".

Definition 3 (Association). *An* **association** *is a statement of the form:*

$$\begin{array}{l} \underline{\mathsf{from}} \quad x_1 \ \underline{\mathsf{in}} \ g_1, \ldots, x_n \ \underline{\mathsf{in}} \ g_n \\ \underline{\mathsf{where}} \quad B \end{array}$$

where each x_i $\underline{\mathsf{in}}$ g_i is a generator (as defined above). We require that the variable (if any) used in g_i is defined by a previous generator within the same clause. The where *clause (B) is a conjunction of equalities between expressions over $x_1, \ldots, x_n$. This clause may also include equalities with constants (i.e., selection conditions).*

An association implicitly defines a relationship among all the atomic elements defined by expressions over the variables $x_1, \ldots, x_n$. As a very simple example, consider the following association:

$$\underline{\mathsf{from}} \ c \ \underline{\mathsf{in}} \ \texttt{contacts}$$

The atomic type elements that are implicitly part of this association are c.`cid`, c.`email` and c.`phone`.

To understand and reason about mappings and rewritings of mappings, we must understand (and be able to represent) relationships between associations. We use renamings (1-1 functions) to express a form of subsumption between associations.

Definition 4. *An association A_1 is* **dominated** *by an association A_2 (denoted as $A_1 \dot{\preceq} A_2$) if there is a renaming function h from the variables of A_1 to the variables of A_2 such that the* from *and* where *clauses of $h(A_1)$ are subsets, respectively, of the* from *and* where *clauses of A_2. (Here we assume that the* where *clause of A_2 is closed under transitivity of equality.)*

Definition 5. *The* **union** *of two associations A_1 and A_2 (denoted as $A_1 \sqcup A_2$) is an association whose* from *and* where *clause consist of the contents of the respective clauses of A_1 and A_2 taken together (with an appropriate renaming of variables if they overlap).*

If B is a set of equalities $e{=}e'$, we will abuse notation a bit and use $A \sqcup B$ to denote the association A with the equalities in B appended to its where clause.

Structural Associations. An important way to specify semantically meaningful relationships between schema elements is through the structural organization of the schema elements. Associations that are based on this kind of relationship will be referred to as *structural associations*.

Definition 6 (Structural Association). *A* **structural association** *is an association defined by a primary path P and having no where clause:* from *P.*

Example 2. Figure 6 indicates all the primary paths of the two schemas of Figure 1. There is one structural association for each primary path. Note that where more than one relation is used, the relations are related through nesting. For example, $P_2^{\mathcal{T}}$ represents fundings (which will each necessarily have an associated organization).

$P_1^{\mathcal{S}}$: p in `companies`
$P_2^{\mathcal{S}}$: g in `grants`
$P_3^{\mathcal{S}}$: c in `contacts`

$P_1^{\mathcal{T}}$: o in `organizations`
$P_2^{\mathcal{T}}$: o in `organizations`, f in o.`fundings`
$P_3^{\mathcal{T}}$: f in `finances`

Fig. 6. Source and target primary paths

All primary paths (and therefore all structural associations) in a schema can be computed by a one time traversal over the schema [43].

User Associations. The semantic relationships between atomic elements described by structural associations are relationships that the data administrators

have encoded in the schema during schema design. However, there are relationships that can exist between schema elements, that are not encoded in the schema, but can be either explicitly specified by a user, or identified through other means such as examining the queries in a given query workload. Associations of this kind are referred to as *user associations*.

Definition 7. *A* **user association** *is any association specified explicitly by a user or implicitly though user actions.*

Example 3. If the `grants.gid` contains as its first five letters, the name of the company who *gave* a grant, then we may find in the workload queries that perform this join frequently. Hence, we may define the following user association:

from g in `grants`, c in `companies`
where Substr(g.`gid`,1,5) = c.`name`

Logical Associations. Apart from the schema structure or a user query, an additional way database designers may specify semantic relationships between schema elements is by using constraints.

Example 4. Every record in `grants` in an instance of the source schema of Figure 1 is related to one or more records in `companies` through the referential constraint f_1 from the `recipient` element to `name`. From that, it is natural to conclude that the elements of a grant are semantically related to the elements of a company. Thus, they can be combined together to form an association. Similarly, we can conclude that the elements of a grant are also related to the elements of `contacts` through both of the referential constraints on `supervisor` and `manager`. Formally, the resulting association is:

from g in `grants` , c in `companies`, s in `contacts`, m in `contacts`
where g.`recipient` = c.`name` and g.`supervisor` = s.`cid`
and g.`manager` = m.`cid`

There are no other schema elements that can be added to the association by following the same reasoning (based on constraints). In that sense, the association that has just been formed is maximal.

Associations that are maximal like the one described in the previous example are called *logical associations*. Logical associations play an important role in the current work, since they specify possible join paths, encoded by constraints, through which schema elements in different relations or different primary paths can be related. We shall make use of this information in order to understand whether (and how) two correspondences in a schema mapping scenario should be considered together when forming a mapping.

Logical associations are computed by using the chase [36]. The chase is a classical method that was originally introduced to test the implication of functional dependencies. The chase was introduced for the relational model and later extended [42] so that it can be applied on schemas with nested structures.

The chase consists of a series of *chase steps*. A chase step of association A with an NRI f: foreach X exists Y with B, can be applied if, by definition, the association A contains the path X (up to a renaming α of the variables in X) but does not satisfy the constraint, that is, A does not contain the path Y (up to a renaming β of the variables in Y) such that B is satisfied (under the corresponding renamings α and β). The result of the chase step is an association A' with the Y clause and the B conditions (under the respective renamings) added to it. The chase can be used to enumerate logical join paths, based on the set of referential constraints in a schema. We use an extension of a nested chase [42] that can handle choice types and NRIs [48].

Let Σ denote a set of NRIs, and A an association. We denote by $\texttt{Chase}_\Sigma(A)$ the final association that is produced by a sequence of chase steps starting from A as follows. In the case where no constraints in Σ can be applied to A, then $\texttt{Chase}_\Sigma(A)$ is A. If there are constraints that can be applied to A, then $\texttt{Chase}_\Sigma(A)$ is constructed as follows: a first step is applied on A; each subsequent step is applied on the output of the previous one; each step uses an NRI from the set Σ; and the sequence of chase steps continues until no more can be applied using NRIs from Σ. In general, if the constraints are cyclic, the chase process may not terminate. However, if we restrict Σ to certain classes of constraints such as *weakly-acyclic* sets of NRIs, termination is guaranteed. (See also the later discussion in Section 6.) Furthermore, it is known (see [19] for example) that when the chase terminates, then it terminates with a unique result (up to homomorphic equivalence[5]). Thus, the result $\texttt{Chase}_\Sigma(A)$ is independent, logically, of the particular order in which the constraints are applied.

Of particular interest to us is the chase applied to structural associations, which is used to compute logical relationships that exist in the schema. A logical association can then be formally defined as the result of such chase.

Definition 8 (Logical Association). *A* **logical association** *is an association* $\texttt{Chase}_\Sigma(A)$*, where A is a structural association or a user association, and Σ is the set of NRIs defined on the schema.*

Example 5. Consider the structural association defined using $P_1^{\mathcal{S}}$ shown in Figure 6. A chase step with the referential constraint

$$f_1: \text{ foreach } g \text{ in } \texttt{grants} \text{ exists } c \text{ in } \texttt{companies} \text{ where } g.\texttt{recipient} = c.\texttt{name}$$

cannot be applied since f_1 uses `grants` in its foreach clause. A chase step with f_1 can, however, be applied on the association defined by $P_2^{\mathcal{S}}$, since there is a renaming function (in this case the identity mapping) from the variables in the foreach clause of the constraint to the variables of the association. Applying the chase step will lead to an association that is augmented with the exists and where clauses of the constraint. Thus, the association becomes:

[5] In our context, two associations are homomorphic equivalent if there are homomorphisms in both directions; this also implies that the associations have a unique *minimal* form.

from g in grants, c in companies
where g.recipient = c.name

Performing a subsequent chase step with the referential constraint f_2 creates the association:

from g in grants , c in companies, s in contacts
where g.recipient = c.name and g.supervisor = s.cid

A subsequent chase step with constraint f_3 will create the association:

from g in grants , c in companies, s in contacts, m in contacts
where g.recipient = c.name and g.supervisor = s.cid
and g.manager = m.cid

No additional chase steps can be applied to this association, since all the constraints are "satisfied" by it. Thus, the last association is maximal, and it is a logical association. Note how in this logical association the relation contacts appears twice. This is due to the fact that there are two different join paths through which one can reach contacts from grants. One is through the referential constraint f_2 and one through f_3.

Figure 7 indicates the logical associations that can be generated using all the structural associations of our two example schemas of Figure 1.

$A_1^{\mathcal{S}}$: from c in companies
$A_2^{\mathcal{S}}$: from g in grants , c in companies, s in contacts, m in contacts
where g.recipient = c.name and g.supervisor = s.cid and g.manager = m.cid
$A_3^{\mathcal{S}}$: from c in contacts

$A_1^{\mathcal{T}}$: from o in organizations
$A_2^{\mathcal{T}}$: from o in organizations, f in o.fundings, i in finances
where f.finId = i.finId
$A_3^{\mathcal{T}}$: from f in finances

Fig. 7. Source and target logical associations.

4.2 Mapping Generation

Logical associations provide a way to meaningfully combine correspondences. Intuitively, a set of correspondences whose source elements all occur in the same source logical association, and whose target elements all occur in the same target

logical association can be interpreted together. Such a set of correspondences maps a set of related elements in the source to a set of related elements in the target data.

The algorithm for generating schema mappings finds maximal sets of correspondences that can be interpreted together by testing whether the elements they match belong in the same logical association, in the source and in the target schema. As seen in the previous section, logical associations are not necessarily disjoint. For example, A_1^S and A_2^S of Figure 7 both include elements of `companies`, although A_2^S also includes elements of `grants` and `contacts`. Thus, a correspondence can use elements that may occur in several logical associations (in both source and target). Rather than looking at each individual correspondence, the mapping algorithm looks at each pair of source and target logical associations. For each such pair, we can compute a candidate mapping that includes all correspondences that use only elements within these logical associations. (As we shall see, not all candidate mappings are actually generated, since some are subsumed by other mappings).

We begin by formally defining correspondences as mappings. First, we define an intensional notion of a *schema element* that we shall use in the subsequent definitions and algorithm.

Definition 9. *Given a schema, a* schema element *is a pair* $\langle P; e\rangle$ *where* P *is a primary path in the schema and* e *is an expression depending on the last variable of* P*.*

Intuitively, the pair $\langle P; e\rangle$ encodes the navigation pattern needed to "reach" all instances of the schema element of interest. Note that a pair $\langle P; e\rangle$ can be turned into a query $\underline{\mathsf{select}}\ e\ \underline{\mathsf{from}}\ P$ that actually retrieves all such instances.

For our example, $\langle c\ \underline{\mathsf{in}}\ \texttt{companies}; c.\texttt{name}\rangle$ represents the schema element `name` in the `companies` table of our source schema, while $\langle o\ \underline{\mathsf{in}}\ \texttt{organizations}, f\ \underline{\mathsf{in}}\ o.\texttt{fundings}; f.\texttt{fid}\rangle$ identifies the schema element `fid` under `fundings` of `organizations` in our target schema.

Definition 10 (Correspondence). *A* correspondence *from an element* $\langle P^S; e_S\rangle$ *of a source schema to an element* $\langle P^T; e_T\rangle$ *of a target schema is defined by the mapping:*

$$v ::= \underline{\mathsf{foreach}}\ P^S \underline{\mathsf{exists}}\ P^T \underline{\mathsf{with}}\ e_S = e_T$$

In practice, a correspondence need not be restricted to an exact equality; we may allow the application of a function f to e_S. In such case the $\underline{\mathsf{with}}$ clause would be $f(e_S) = e_T$. Clio does not discover such functions, but does provide a library of common type conversion functions that a user may specify on a correspondence. The system also permits the use of user-defined functions. Similarly, we could have a function applied to several source elements to generate a single target element (for example, `concat(fname,lname) = name`). This type of N:1 correspondence could be created by some matchers [11] and used by Clio

in mapping generation. To keep the notation for our algorithms simple, we will assume that correspondences are of the form given in Definition 10.

Given a pair of source and target logical associations, we would like to define when (and how) a correspondence v is relevant to this pair. In general, a correspondence may be used in multiple ways with a pair of logical associations. For example, a correspondence for the `phone` element under `contacts` can be used in two ways with the pair of associations $\mathcal{A}_2^S$ and $\mathcal{A}_2^T$ in Figure 7 (to map either the `supervisor` or `manager` phone). In our algorithm, we identify all possible ways of using a correspondence. The following definition formalizes the notion of a single use (which we call *coverage*) of a correspondence by a pair of associations.

Definition 11. *A correspondence* v : foreach P^S exists P^T with $e_S = e_T$ *is* covered *by a pair of associations* $<A^{\mathcal{S}}, A^{\mathcal{T}}>$ *if* $P^S \doteq A^{\mathcal{S}}$ *(with some renaming function* h*) and* $P^T \doteq A^{\mathcal{T}}$ *(with some renaming function* h'*). We say in this case that there is a* coverage *of* v *by* $<A^{\mathcal{S}}, A^{\mathcal{T}}>$ *via* $<h, h'>$*. We also say that the* result *of the coverage is the expression* $h(e_S)$=$h'(e_T)$*.*

Our algorithm will consider each pair of source and target associations that cover by at least one correspondence. For each such pair, we will consider all correspondences that are covered and pick one coverage for each. For each such choice (of coverage), the algorithm will then construct what we call a *Clio mapping*.

Definition 12. *Let* $\mathcal{S}$ *and* $\mathcal{T}$ *be a pair of source and target schemas and* $\mathcal{C}$ *a set of correspondences between them. A* Clio mapping *is a mapping* foreach A^S exists A^T with E*, where* A^S *and* A^T *are logical associations in the source and the target schema, respectively, and* E *is a conjunction of equalities constructed as follows. For each correspondence* v *in* $\mathcal{C}$ *that is covered by* $<A^{\mathcal{S}}, A^{\mathcal{T}}>$*, we choose one coverage of* v *by* $<A^{\mathcal{S}}, A^{\mathcal{T}}>$ *via* $<h, h'>$ *and add the equality* $h(e_S) = h'(e_T)$ *that is the result of this coverage.*

Note that, in the above definition, only one coverage for each correspondence is considered when constructing a Clio mapping. Different coverages will yield different Clio mappings. The following two examples illustrate the process for two different pairs of associations.

Example 6. Consider the pair of logical associations $<A_1^{\mathcal{S}}, A_1^{\mathcal{T}}>$. We check each of the correspondences in $\{v_1, v_2, v_3, v_4\}$ to see if it is covered by these associations. It is easy to see that the correspondence v_1, which relates the source element $\langle c$ in `companies`$; c.$`name`$\rangle$ with the target element $\langle o$ in `organizations`$; o.$`code`$\rangle$, is covered by our pair of associations. There is only one coverage in this case, given by the identity renaming functions. Since no other correspondence is covered, we can form a Clio mapping based on $A_1^{\mathcal{S}}$ and $A_1^{\mathcal{T}}$, with the sole equality that results from v_1 added to the with clause:

m_{v1}: foreach c in `companies`
exists o in `organizations`
with $c.$`name` $= o.$`code`

In this simple example, the mapping happens to be the same as the original correspondence. This is because both of the primary paths of this correspondence are logical associations themselves, and also no other correspondence is covered; hence, mapping m_{v1} is v_1. (Notice that this is the same as Mapping (1) from Section (2), where it was represented in traditional s-t tgd notation).

Example 7. As a more complex example of mapping generation, consider the association pair $<A_2^{\mathcal{S}}, A_2^{\mathcal{T}}>$. Following the same steps, we can determine that the correspondences v_1, v_2 and v_3 are covered, and the following mapping using only these three correspondences can be generated:

foreach g in grants, c in companies, s in contacts, m in contacts
  where g.recipient = c.name and g.supervisor = s.cid
    and g.manager = m.cid
exists o in organizations, f in o.fundings, i in finances
  where f.finId = i.finId
with c.name = o.code and g.gid = f.fid and g.amount = i.budget

Consider now our final correspondence:

v_4: foreach c in contacts
  exists f in finances
  with c.phone = f.phone

which is also covered by the above two associations. However, since contacts appears twice in $A_2^{\mathcal{S}}$, there are two different renaming functions for the foreach clause of v_4; hence, there are two different coverages. In the first, the variable c (of v_4) maps to s, while in the second, the variable c maps to m. This will lead to the generation of the following two mappings.

M_1:

foreach g in grants , c in companies, s in contacts, m in contacts
  where g.recipient = c.name and g.supervisor = s.cid
    and g.manager = m.cid
exists o in organizations, f in o.fundings, i in finances
  where f.finId = i.finId
with c.name = o.code and g.gid = f.fid and g.amount = i.budget
  and s.phone = i.phone

M_2:

foreach g in grants , c in companies, s in contacts, m in contacts
  where g.recipient = c.name and g.supervisor = s.cid

```
            and g.manager = m.cid
exists  o in organizations, f in o.fundings, i in finances
        where f.finId = i.finId
with    c.name = o.code and g.gid = f.fid and g.amount = i.budget
            and m.phone = i.phone
```

Notice that M_1 and M_2 have two copies of `contacts` in their source query, only one of which is used in the target query. A minimization algorithm (similar to tableau minimization [2]) can be applied to remove the occurrence of `contacts` that is not used in the target. So Mapping M_1 is equivalent to the Mapping (5) of Section 2 which is written in the more common s-t tgd notation, and M_2 is equivalent to Mapping (6) of the same section.

Our mapping generation algorithm is summarized in Algorithm 1. If the source schema has N logical associations and the target schema has M logical associations, there will be $N \times M$ pairs of associations that have to be considered by the algorithm. However, not all of these pairs will generate actual mappings. Some pairs may not cover any correspondences and are discarded. Additionally, some pairs of associations are *subsumed* by other pairs and they are also discarded. More precisely, a pair $<A^S, A^T>$ of associations is *subsumed* by another pair $<X, Y>$ of associations if: (1) $X \dot{\preceq} A^S$ or $Y \dot{\preceq} A^T$, and at least one of these two dominances is strict (i.e., X or Y have strictly smaller number of variables), and (2) the set of correspondences covered by $<X, Y>$ is the same as the set of correspondences covered by $<A^S, A^T>$. Intuitively, all the correspondences that are covered by $<A^S, A^T>$ are also covered by a "strictly smaller" pair of associations. The heuristic that we apply is to discard the "larger" mapping (based on A^S and A^T) since it does not make use of the "extra" schema components. This heuristic can eliminate a large number of unlikely mappings in practice and is a key ingredient for the efficiency of the algorithm. Additional details regarding the data structures needed to efficiently implement this heuristic are given in Haas et al. [27].

5 Query Generation for Data Exchange

The schema mappings we generate specify how the data of the two schemas relate to each other. For data exchange, a source instance must be restructured and transformed into an instance of the target schema. Fagin et al. [19] have defined the problem as follows:

Definition 13. *Given a source schema $\mathcal{S}$, a target schema $\mathcal{T}$, a set Σ_{st} of source-to-target constraints (i.e., the mappings), and a set Σ_t of target constraints, the* data exchange problem *is the following problem: given a finite source instance $\mathcal{I}$, find a finite target instance $\mathcal{J}$ such that $(\mathcal{I}, \mathcal{J})$ satisfies Σ_{st} and $\mathcal{J}$ satisfies Σ_t. Such an instance $\mathcal{J}$ is called a* solution *to the data exchange problem.*

Algorithm 1: Schema Mapping Generation

Input: A source schema $\mathcal{S}$
A target schema $\mathcal{T}$
A set of correspondences $\mathcal{C}$
Output: The set of all Clio mappings $\mathcal{M}$

GenerateMappings($\mathcal{S}$, $\mathcal{T}$, $\mathcal{C}$)
(1) $\mathcal{M} \leftarrow \emptyset$
(2) $\mathcal{A}^{\mathcal{S}} \leftarrow$ Logical Associations of $\mathcal{S}$
(3) $\mathcal{A}^{\mathcal{T}} \leftarrow$ Logical Associations of $\mathcal{T}$
(4) **foreach** pair $<A^{\mathcal{S}}, A^{\mathcal{T}}>$ of $\mathcal{A}^{\mathcal{S}} \times \mathcal{A}^{\mathcal{T}}$
(5) $V \leftarrow \{v \mid v \in \mathcal{V} \wedge v$ is covered by $<A^{\mathcal{S}}, A^{\mathcal{T}}> \}$
(6) *// If no correspondences are covered*
(7) **if** $V = \emptyset$
(8) continue;
(9) *// Check if subsumed*
(10) **if** $\exists$ <X,Y> with $X \dot{\preceq} A^{\mathcal{S}}$ or $Y \dot{\preceq} A^{\mathcal{T}}$, and at least one dominance is strict
(11) $V' \leftarrow \{v \mid v \in \mathcal{C} \wedge v$ covered by $<X, Y> \}$
(12) **if** $V' = V$
(13) continue;
(14) let V be $\{v_1, \ldots, v_m\}$
(15) for every v_i: let Δ_{v_i} be $\{<h, h'> \mid v_i$ covered by $<A^{\mathcal{S}}, A^{\mathcal{T}}>$ via $<h, h'>\}$
(16) *// For every combination of correspondence coverages*
(17) **foreach** $(\delta_1, \ldots, \delta_m) \in \Delta_{v_1} \times \ldots \times \Delta_{v_m}$
(18) $W \leftarrow \emptyset$
(19) **foreach** $v_i \in V$
(20) let $e = e'$ be the equality in v_i
(21) let δ_i be $<h, h'>$
(22) add equality $h(e) = h'(e')$ to W
(23) form Clio mapping M: foreach $A^{\mathcal{S}}$ exists $A^{\mathcal{T}}$ with W
(24) $\mathcal{M} \leftarrow \mathcal{M} \cup \{M\}$
(25) **return** $\mathcal{M}$

This section will describe one approach to finding such a *solution*. We will discuss in Section 6 how this solution relates to the *universal solution* of Fagin et al. [19].

Notice that a schema mapping (1) does not specify all target values and (2) does not specify the grouping or nesting semantics for target data. Thus, in generating a solution we will have to address these issues. Furthermore, in generating a target instance, we will typically be using many mappings, as we may have one or more mappings per concept in the schemas. Hence, we will need to merge data produced from multiple mappings.

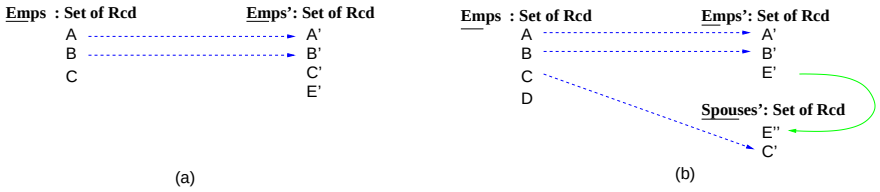

Fig. 8. Creation of new values in the target.

5.1 Intuition: What Are the Challenges

To begin, we will explore a bit further the challenges we face in building our data exchange queries, building up intuition for the algorithm itself.

Creation of New Values in the Target. Consider the simple mapping scenario of Figure 8(a). The source (Emps) and the target (Emps′) are sets containing employee records. An employee record in the source has atomic elements A, B and C, while an employee record in the target has elements A′, B′, C′ along with an extra atomic element E′. For the purpose of this discussion, we choose to use the abstract names A, etc., so that we can associate several different semantics with these elements for illustration. In the mapping, the two source elements A and B are mapped into the target elements A′ and B′, while C′ and E′ in the target are left unmapped. Consider the following schema mapping which does not specify any value for C′ or E′.

$$\begin{array}{ll} \underline{\text{foreach}} & e \ \ \underline{\text{in}}\ \texttt{Emps} \\ \underline{\text{exists}} & e' \ \ \underline{\text{in}}\ \texttt{Emps}' \\ \underline{\text{with}} & e'.A' = e.A \ \underline{\text{and}} \ \ e'.B' = e.B \end{array}$$

To populate the target we need to decide what values (if any) to use for unmapped elements. A frequent case in practice is one in which an unmapped element does not play a crucial role for the integrity of the target. For example, A and B could be employee name and salary, while C and E could be spouse and date of birth, respectively, where neither is used in any schema constraint. Creating a null value for C′ and E′ is then sufficient. If the unmapped element is optional in a target XML schema, then we can omit this element in the exchange.

Alternatively, the element E′ may be a key in the target relation, e.g., E′ could be employee id. The intention of the mapping would be, in this case, to copy employee data from the source, and assign a new id for each employee in the target. Thus, a non-null (and distinct) value for E′ is needed for the integrity of the target. In general, a target element is **needed** if it is (part of) a key or referential constraint (such as a foreign key) or is both not nullable and not optional.

If E′ is a key, we create a *different* but *unique* value for E′, for *each* combination of the source values for A and B using a one-to-one Skolem function. In this example, values for E′ are created using the function $Sk_{\texttt{E}'}(\texttt{A}, \texttt{B})$. We can

then augment the schema mapping with explicit conditions in the with clause to provide an appropriate value for all unmapped attributes.

foreach e in Emps
exists e' in Emps′
with $e'.A' = e.A$ and $e'.B' = e.B$ and $e'.C' = null$ and $e'.E' = Sk_{\mathtt{E}'}(e.\mathtt{A}, e.\mathtt{B})$

Notice that we choose to make E′ depend only on A and B, not on the (unmapped) source element C. Thus, even if in the source there may exist two tuples with the same combination for A and B, but with two different values for C (e.g., if C is spouse as above, and an employee has two spouses), in the target there will be only one tuple for the given combination of A and B (with one, unknown, spouse). Thus, the semantics of the target is given solely by the values of the source elements that are *mapped*. Of course, a new correspondence from C to C′ will change the mapping: an employee with two spouses will appear twice in the target and the value for E′ will be $Sk_{\mathtt{E}'}(\mathtt{A}, \mathtt{B}, \mathtt{C})$.

Now consider an unmapped target element that is used in a *referential constraint*. In Figure 8(b), the (mapped) target element C′ is stored in a different location (the set Spouses) than that of elements A′ and B′. However, the association between A′, B′ values and C′ values is meant to be preserved by a referential constraint (E′ plays the role of a reference in this case). The schema mapping created by Clio is the following.

foreach e in Emps
exists e' in Emps′, s' in Spouses′
where $e'.\mathtt{E}' = s'.\mathtt{E}''$
with $e'.\mathtt{A}' = e.\mathtt{A}$ and $e'.\mathtt{B}' = e.\mathtt{B}$ and $s'.\mathtt{C}' = e.\mathtt{C}$

Note that this mapping does not give a value for the required element Emps′.E′ or Spouses′.E″. We can provide values for these two unmapped elements using a Skolem function $Sk_{\mathtt{E}'}(\mathtt{A}, \mathtt{B}, \mathtt{C})$ to create values for E′ and E″.

foreach e in Emps
exists e' in Emps′, s' in Spouses′
where $e'.\mathtt{E}' = s'.\mathtt{E}''$
with $e'.\mathtt{A}' = e.\mathtt{A}$ and $e'.\mathtt{B}' = e.\mathtt{B}$ and $s'.\mathtt{C}' = e.\mathtt{C}$ and $e'.\mathtt{E}' = Sk_{\mathtt{E}'}(e.\mathtt{A}, e.\mathtt{B}, e.\mathtt{C})$

In the above example, the same Skolem function will populate both E' and E'', since E' and E'' are required to be equal by the where clause of the mapping. In general, however, different target attributes will use different Skolem functions.

Note also that if duplicate $[a, b, c]$ triples occur in the source (perhaps with different D values, where D is some other attribute) only one element is generated in each of Emps′ and Spouses′. Thus, we eliminate duplicates in the target based on the *mapped* source data.

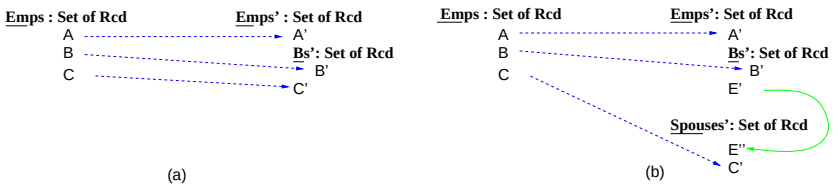

Fig. 9. Grouping of elements in the target.

Grouping of Nested Elements. Consider now Figure 9(a), in which the target schema contains two levels of nesting: elements $\mathtt{A}'$ and $\mathtt{C}'$ are at the top level, while there are multiple $\mathtt{B}'$ elements ($\mathtt{Bs}'$ is of set type). Elements $\mathtt{A}$, $\mathtt{B}$, and $\mathtt{C}$ of the source $\mathtt{Emps}$ are mapped, via correspondences, into the respective elements of the target $\mathtt{Emps}'$. The mapping that Clio generates:

$$\begin{array}{ll} \underline{\text{foreach}} & e \ \ \underline{\text{in}}\ \mathtt{Emps} \\ \underline{\text{exists}} & e' \ \ \underline{\text{in}}\ \mathtt{Emps}',\ b' \ \ \underline{\text{in}}\ \ e'.Bs' \\ \underline{\text{with}} & e'.\mathtt{A}' = e.\mathtt{A} \ \ \underline{\text{and}}\ \ b'.\mathtt{B}' = e.\mathtt{B} \ \ \underline{\text{and}}\ \ e'.\mathtt{C}' = e.\mathtt{C} \end{array}$$

requires that all $(\mathtt{A}, \mathtt{B}, \mathtt{C})$ values found in the source appear in the target. In addition, a natural interpretation of the target schema is that all B values sharing the same $\mathtt{A}$ and $\mathtt{C}$ be grouped together in one set. For illustration, if $\mathtt{A}$, $\mathtt{B}$, and $\mathtt{C}$ are employee, child, and spouse names, respectively, Clio will choose to group all children with the same employee and spouse in a single set. Note that this behavior is *not* part of the mapping specification itself. (A solution of the above mapping can be obtained by creating a target tuple with a singleton Bs'-set for each source tuple.) However, for data exchange, we choose to add this grouping semantics that is implicitly specified by the target schema, and produce a target instance that is in Partitioned Normal Form (PNF) [1].

PNF: *In any target nested relation, there cannot exist two distinct tuples that coincide on all the atomic elements but have different set-valued elements.*

To achieve this grouping behavior, we use Skolemization as well. If a target element has a set type, then its identifier (recall that each set is identified in the data model by a set id) is created via a Skolem function. This function *does not* depend on any of the atomic elements that are mapped under (in terms of nesting) the respective set type, in the schema hierarchy. Instead it depends on all the atomic elements at the same level or above (up to the root of the schema). The same Skolem function (for a given set type of the target schema) is used across all mappings. Intuitively, we perform a deep union of all data in the target independently of their source. For the example of Figure 9(a), we modify the schema mapping with a Skolem function for $\mathtt{Bs}$.

$$\begin{array}{ll} \underline{\text{foreach}} & e \ \ \underline{\text{in}}\ \mathtt{Emps} \\ \underline{\text{exists}} & e' \ \ \underline{\text{in}}\ \mathtt{Emps}',\ b' \ \ \underline{\text{in}}\ \ e'.Bs' \\ \underline{\text{with}} & e'.\mathtt{A}' = e.\mathtt{A} \ \ \underline{\text{and}}\ \ b'.\mathtt{B}' = e.\mathtt{B} \ \ \underline{\text{and}}\ \ e'.\mathtt{C}' = e.\mathtt{C} \ \ \underline{\text{and}}\ \ e'.Bs' = Sk_{Bs'}(e.\mathtt{A}, e.\mathtt{C}) \end{array}$$

The meaning of the above rule is the following: for each (a, b, c) triple of values from the source, create a record in $\texttt{Emps}'$, with the appropriate $\texttt{A}'$ and $\texttt{C}'$ attributes, and also a $\texttt{Bs}'$ attribute, the value of which is the set id $Sk_{\texttt{Bs}'}(a, c)$. Thus, the Skolem function $Sk_{\texttt{Bs}'}$ is used here to create a set type element. Also, we create an element $\texttt{B}'$, with value b, under $Sk_{\texttt{Bs}'}(a, c)$. Another tuple (a, b', c) will lead to the value b' being nested in the same set as b.

Hence, we achieve desired grouping of $\texttt{B}'$ elements for fixed $\texttt{A}'$ and $\texttt{C}'$ values.

Value Creation Interacts with Grouping. To create a nested target instance, we will need to consider referential constraints together with our desired PNF grouping. We again explain our technique with an example. Consider Figure 9(b), where the elements $\texttt{A}'$ and $\texttt{C}'$ are stored in separate target sets. The association between $\texttt{A}'$ (e.g., employee name) and $\texttt{C}'$ (e.g., spouse name) is preserved via the foreign key $\texttt{E}'$ (e.g., spouse id). Thus, $\texttt{E}'$ is a required element and must be created. However, in this case, it is rather intuitive that the value of $\texttt{E}'$ should not depend on the value of $\texttt{B}'$, but only on the $\texttt{A}'$ and $\texttt{C}'$ value. This, combined with the PNF requirement, means that all the $\texttt{B}'$ (child) values are grouped together if the employee and spouse names are the same. We achieve therefore the same effect that the mapping of Figure 9(a) achieves. In contrast, if $\texttt{E}'$ were to be assigned a different value for different $\texttt{B}'$ values, then each child will end up in its own singleton set. For data exchange, we choose the first alternative, because we believe that this is the more practical interpretation. Thus, we adjust the earlier Skolemization scheme for atomic type elements as follows.

The function used for creation of an atomic element $\texttt{E}$ *does not depend on any of the atomic elements that occur at a lower level of nesting in the target schema.*

For the example of Figure 9(b) we create the rule:

$$
\begin{array}{l}
\underline{\text{foreach}}\ \ e\ \ \underline{\text{in}}\ \texttt{Emps} \\
\underline{\text{exists}}\ \ \ e'\ \ \underline{\text{in}}\ \texttt{Emps}',\ b'\ \ \underline{\text{in}}\ \ e'.Bs',\ s'\ \ \underline{\text{in}}\ \texttt{Spouses}' \\
\quad \underline{\text{where}}\ \ e'.\texttt{E}' = s'.\texttt{E}'' \\
\underline{\text{with}}\ \ \ \ e'.\texttt{A}' = e.\texttt{A}\ \ \underline{\text{and}}\ \ b'.\texttt{B}' = e.\texttt{B}\ \ \underline{\text{and}}\ \ s'.\texttt{C}' = e.\texttt{C}\ \ \underline{\text{and}} \\
\qquad\qquad e'.\texttt{E}' = Sk_{\texttt{E}'}(e.\texttt{A}, e.\texttt{C})\ \ \underline{\text{and}}\ \ e'.Bs' = Sk_{Bs'}(e.\texttt{A}, Sk_{\texttt{E}'}(e.\texttt{A}, e.\texttt{C}))
\end{array}
$$

Note that in this case, the Skolem function for $\texttt{Bs}'$ is (as before) a function that depends on all the atomic type elements that are at the same level or above it (in the schema hierarchy). Thus, it is a function of the values that are assigned to $\texttt{A}'$ and $\texttt{E}'$ (the latter being a Skolem function itself, in this case).

As an extreme but instructive case, suppose that in Figure 9(b) we remove the correspondences that involve $\texttt{A}'$ and $\texttt{C}'$, but keep the one that involves $\texttt{B}'$. If the target elements $\texttt{A}'$ and $\texttt{C}'$ are not optional, then they will be created by (unique) Skolem functions with no arguments. This is the same as saying that they will each be assigned a single (distinct) null. Consequently, the two target sets $\texttt{Emps}'$ and $\texttt{Spouses}'$ will each contain a single element: some unknown

employee, and some unknown spouse, respectively. In contrast, the nested set Bs′ will contain all the B values (all the children listed in Emps). Thus, the top-level part of the schema plays only a structural role: it is *minimally* created in order to satisfy the schema requirements, but the respective values are irrelevant. Later on, as correspondences may be added that involve A′ and C′, the children will be separated into different sets, depending on the mapped values.

We now describe in some detail the algorithm for generating the data exchange queries. The algorithm is applied to every Clio mapping that has been generated, and works in three steps. First, given a Clio mapping, we construct a graph that represents the key portions of the query to be generated. Second, we annotate that graph to generate the Skolem terms for the query. These two steps are discussed in the next section. Finally, we walk the graph, producing the actual query, as described in Section 5.3.

5.2 The Query Graph

Given a mapping m, a graph is constructed, called the *query graph*. The graph will include a node for each target expression that can be constructed over the variables in the exists clause of the mapping. Source expressions that provide values for the target will also have nodes. Furthermore, the graph will include links between nodes to reflect parent-child relationships and equality predicates, respectively. This structure is then further refined to support Skolem function generation; at that point, it contains all the information needed to generate the query.

In the query generation algorithm, all the variables that appear in the input mapping are assumed to be named consistently as: $x_0, x_1, \ldots$ for the source side, and $y_0, y_1, \ldots$ for the target side. Let us revisit one of our earlier mappings (M_1 in Section 4.2), using the above naming convention:

```
foreach x0 in companies, x1 in grants , x2 in contacts, x3 in contacts
      where x0.name = x1.recipient and x1.supervisor = x2.cid
            and x1.manager = x3.cid
exists y0 in organizations, y1 in y0.fundings, y2 in finances
      where y1.finId = y2.finId
with  x0.name = y0.code and x1.gid = y1.fid and x1.amount = y2.budget
            and x2.phone = y2.phone
```

The query graph for this mapping is illustrated in Figure 10. The query graph is constructed by adding a node for each variable (and its generator) in the exists clause of the mapping. There are three such nodes in our example query graph. Furthermore, we add nodes for all the atomic type elements that can be reached from the above variables via record projection. For our example, we include nodes for y_0.code, y_0.year, and so on. (If we have multiple levels of records, then there will be several intermediate nodes that have to be introduced.) We then add structural edges (the directed full arcs in the figure) to reflect the

obvious relationships between nodes. Moreover, we add *source* nodes for all the source expressions that are actually used in the with clause of the mapping. For our example, we add nodes for x_0.`name`, x_1.`gid`, and the other two source expressions. We then attach these source nodes to the target nodes to which they are "equal". This is illustrated via the dotted, directed arcs in the figure (e.g., the arc from y_0.`code` to x_0.`name`).

Finally, we use the equalities in the where clause on the target side and add "equality" edges between target nodes. For our example, we add the dotted, undirected, edge connecting the node for y_1.`finId` and the node for y_2.`finId`.

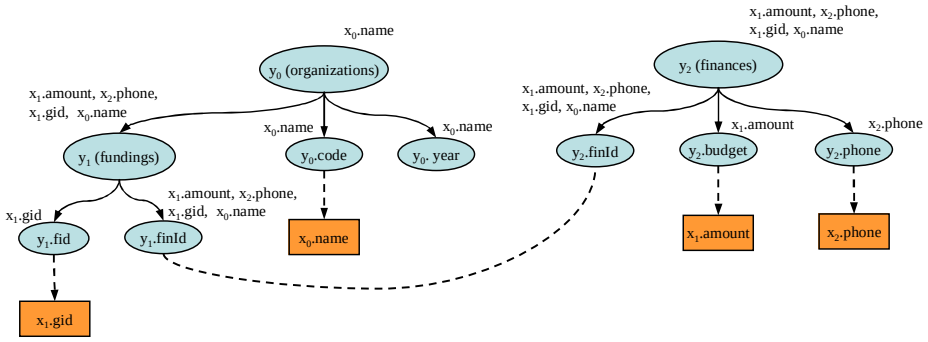

Fig. 10. An annotated query graph.

Next, we annotate each node in the query graph to facilitate the subsequent generation of Skolem functions. Each node in the graph will be annotated with a set of source expressions. These expressions will represent the *arguments* of a potential Skolem function for that node. However, only the nodes that are not mapped from the source and that are needed (as discussed in the previous section) will receive an actual Skolem function.

The annotations are computed by a propagation algorithm. First, every target node that is adjacent to a source schema node through an "equality" (dotted) edge, is annotated with the expression of that source schema node and only that. Next, we start propagating these expressions through the rest of the graph according to the following rules.

- **Upward propagation:** Every expression that a node acquires is propagated to its parent node, unless the (aquiring) node is a variable.
- **Downward propagation:** Every expression that a node acquires is propagated to its children if they do not already have it and if they are not equal to any of the source nodes.
- **Eq propagation:** Every expression that a node acquires is propagated to the nodes related to it through equality edges.

The propagation rules are applied repeatedly until no more rules can be applied. Figure 10 illustrates the result of propagation for our example. The resulting annotated graph is now ready for use to generate a data exchange query.

5.3 Generation of Transformation Queries

Once the query graph of a mapping has been generated, it can be converted to an actual query. The language of the query depends on the model of the source and target. When both source and target are relational schemas, the query will be a relational SQL query. When one schema is XML, Clio can generate a data exchange query in XQuery or XSLT. We describe here how to generate a data exchange query in XQuery. The XSLT and SQL approaches are similar, though for SQL, the query will not have nested elements.

The complete XQuery that is generated for our mapping M_1 is shown in Figure 11. An XSLT version of such query can be generated in a similar way, with only syntactic differences.

To begin, the foreach clause of the mapping is converted to a query fragment on the source schema. This is straightforward: Clio simply binds one variable to each term of the mapping and adds the respective conditions in the where clause. For instance, for our mapping M_1, we create the following XQuery fragment:

```
FOR $x0 IN $doc0/companies, $x1 IN $doc0/grants,
    $x2 IN $doc0/contacts, $x3 IN $doc0/contacts,
WHERE $x0.name=$x1.recipient and $x1.supervisor=$x2.cid
and $x1.manager=$x3.cid
```

Let us denote this query fragment by $Q^S_{M_1}$. Note that this is not a complete query since it does not have a result yet. This query fragment will be used (repeatedly) in a larger query that we generate based on the query graph. The algorithm starts at the target schema root in the query graph and performs a depth-first traversal.

- If the node being visited corresponds to a complex type schema element, then a complex element is generated by visiting the children and enclosing their corresponding elements.
- If a node corresponding to an atomic type element is visited, then: (i) if the node is linked to a source node (directly, or via any number of equality edges), then a simple element is generated with the value equal to the expression represented by the source node, or (ii) if the node corresponds to an optional element, nothing is generated, or (iii) if the node corresponds to a nullable schema element, the null value is generated; or finally, (iv) if none of the above applies, a value must be generated for that element via a Skolem function. In this case, we generate a fresh new Skolem function name and add all arguments that annotate the node. We take care so that all the nodes that are "equal" to the current node will receive the same Skolem function name.

– If the node visited is a variable, then a **FOR**-**WHERE**-**RETURN** query is produced, by first taking a "copy" of the source query fragment (e.g., $Q^S_{M_1}$) (where we rename, in a consistent way, all the variables in the query fragment). In addition, we inspect the expressions that annotate the node and compare with its parent variable (if any). For each *common* expression e, we then add an extra equality condition that equates the expression e at the parent query with (the renaming of) e in the current query. This creates a correlated subquery.
For our example, the subquery that is generated for the node y_1 in our query graph is based on a copy of the query fragment $Q^S_{M_1}$, with an extra equality requiring that (the renaming of) x_0.`name` in the subquery is equal to to x_0.`name` in the parent (e.g., "$x0L1/name/text() = $x0/name/text()").
The traversal continues and any new elements, generated based on the descendants of the current node (variable), will be placed inside the return clause of the subquery that has just been created.

One issue that we need to address in the actual implementation is the fact that Skolem functions are not first-class citizens in typical query languages, including XQuery. However, this issue can be dealt with in practice by simulating a Skolem function as a string that concatenates the name of the Skolem function with the string representations of the values of the arguments.

The above query generation algorithm, as presented, is not complete in the sense that it may not generate a PNF instance. If we consider each Clio mapping individually, then the generated query does produce a data fragment that is in PNF. However, in the case of multiple mappings, we still need to merge the multiple fragments into one PNF instance.

The merging required by PNF can be achieved either via post-processing, or by generating a more complex, two-phase query as follows. The first query transforms the source data, based on the mappings, into a set of "flat" views that encode (in a relational way) the nested target instance in PNF. Nested sets are encoded in this flat view by using Skolem functions (as described in Section 5.1). These Skolem functions represent the identifiers of sets, and each tuple records the id of the set where it belongs. The second query can then take this result and merge the data into the right hierarchy, by joining on the set identifier information produced by the first query. The resulting instance is then guaranteed to be in PNF.

The full details for the two-phase query generation can be found in [27] and in [24].

6 Analysis

In this section we discuss the general conditions under which our method for mapping generation is applicable, and we make precise the connection between our query generation algorithm and the data exchange framework of Fagin et al. [19].

```
LET $doc0 := document("input XML file goes here")
RETURN
 <T>
   {distinct-values (
   FOR $x0 IN $doc0/companies, $x1 IN $doc0/grants,
       $x2 IN $doc0/contacts,  $x3 IN $doc0/contacts
   WHERE
       $x0/name/text() = $x1/recipient/text() AND $x1/supervisor/text() = $x2/cid/text() AND
       $x1/manager/text() = $x3/cid/text()
   RETURN
      <organization>
        <code>  {  $x0/name/text() }  </code>
        <year>  {  "Sk3(", $x0/name/text(), ")" }  </year>
        {distinct-values (
        FOR $x0L1 IN $doc0/companies, $x1L1 IN $doc0/grants,
            $x2L1 IN $doc0/contacts,  $x3L1 IN $doc0/contacts
        WHERE
            $x0L1/name/text() = $x1L1/recipient/text() AND $x1L1/supervisor/text() = $x2L1/cid/text()
        AND $x1L1/manager/text() = $x3L1/cid/text() AND $x0L1/name/text() = $x0/name/text()
        RETURN
           <funding>
              <fid> {$x0L1/gid/text()} </fid>
              <finId>{"Sk5(", $x0L1/name/text(), ", ", $x2L1/phone/text(), ", ",
                      $x1L1/amount/text(), ", ", $x1L1/gid/text(), ")"}</finId>
           </funding> ) }
     </organization> ) }
   {distinct-values (
   FOR $x0 IN $doc0/companies, $x1 IN $doc0/grants,
       $x2 IN $doc0/contacts,  $x3 IN $doc0/contacts
   WHERE
       $x0/name/text() = $x1/recipient/text() AND $x1/supervisor/text() = $x2/cid/text() AND
       $x1/manager/text() = $x3/cid/text()
   RETURN
       <financial>
          <finId>{"Sk5(", $x0/name/text(), ", ", $x2/phone/text(), ", ",
                       $x1/amount/text(), ", ", $x1/gid/text(), ")"}</finId>
          <budget>  {  $x1/amount/text() }  </budget>
          <phone>   {  $x2/phone/text() }  </budget>
      </financial> ) }
</T>
```

Fig. 11. The data exchange XQuery for Mapping M_1.

6.1 Complexity and Termination of the Chase

The chase is the main process used to construct logical associations. In general, it is known that the chase with general dependencies may not terminate. The chase that we use is a special case in which the dependencies (the schema constraints) are NRIs. Even in this special case, the chase may not terminate: one can construct simple examples of cyclic inclusion dependencies (even for the relational model) for which the chase does not terminate.

To guarantee termination, we restrict the NRIs within each schema to form a *weakly-acyclic* set. The notion of a weakly-acyclic set of tgds has been studied [19] and, with a slight variation, in [17]; it was shown that weakly-acyclic sets of tgds strictly generalize the previous notions of acyclic dependencies and it was proven that the chase with a weakly-acyclic set of tgds always terminates in polynomially many chase steps. The definition of weak-acyclicity and the argument for termination apply immediately when we consider tgds over the nested relational model. Hence they apply to our NRIs.

While the number of chase steps required to complete the chase is polynomial (in the weak-acyclic case), a chase step itself (of an association with an NRI) can take an exponential amount of time in the worst case. This is due to the fact that the paths in an NRI require matching (on the association) to determine the applicability of the NRI. In general, there could be multiple ways of matching a variable in the path with a variable in an association. Hence, matching a path means exploring an exponential number of assignments of path variables in the worst case. However, the exponential is in the size of the path (i.e., the number of variables), which in practice is often small. Furthermore, in most cases, a variable in a path has only one way of matching due to schema restrictions (e.g., a variable required to be in `companies` can only match with a variable in `companies`). Thus, the exponential worst case rarely materializes.

Another challenge that is often found in XML schemas is type recursion. Type recursion occurs when a complex type is used in its own definition. Type recursion leads to infinitely many structural associations that may result in infinitely many possible interpretations of a given set of correspondences. Figure 12 illustrates how type recursion can lead to an infinite number of mappings. In this figure, the element `Father` is of type `Persons`. In order to partially cope with this limitation we allow recursive types but we require the user to specify the recursion level, or we choose a specific constant value as a bound. For example, in the mapping scenario of Figure 12, if we bind the recursion level to one, then only the first two mappings shown in the figure will be generated by the system, and the user will be left to select the one that better describes the intended semantics of the correspondences.

6.2 Characterization of Data Exchange

Consider a data exchange setting, that is, a source schema $\mathcal{S}$, a target schema $\mathcal{T}$, a set of mappings $\mathcal{M}$ from $\mathcal{S}$ to $\mathcal{T}$, a set of target constraints Σ_T, and an instance I of schema $\mathcal{S}$. The problem in data exchange is to find a *solution*, i.e., an instance

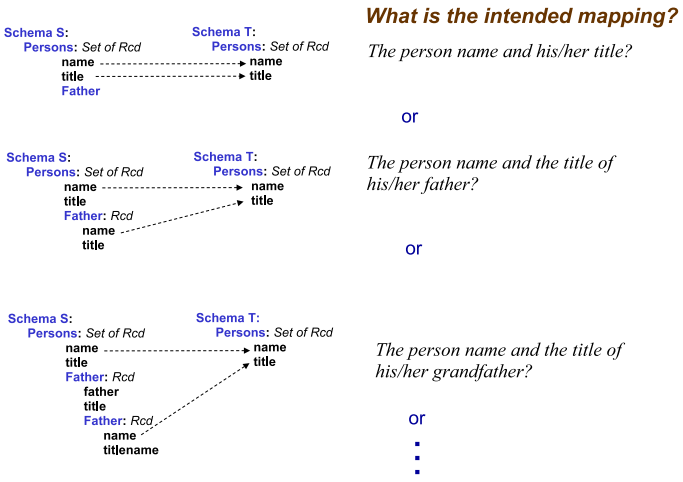

Fig. 12. Unlimited candidate mappings due to recursive types.

J of the target schema $\mathcal{T}$ that is consistent with the mappings in $\mathcal{M}$. If we assume that we are in the relational model, this allows us to use some of the existing results in the research literature. In a data exchange setting, there may be more than one solution. Fagin et al. [19] have provided an algebraic specification that selects among all solutions of a data exchange setting, a special solution that is referred to as a *universal solution*. A universal solution intuitively has no more and no less data than required for data exchange, and represents the entire space of possible solutions. A universal solution J_u is a solution for which there is a homomorphism h to every other solution J. Fagin et al. provided an algorithm, based on the chase, for computing a universal solution. That algorithm starts with the source instance I and chases it with the mappings, and also with the target schema constraints. The result of the chase gives a target instance that is a universal solution.

There are two main differences in our approach. First, the process of generating logical associations compiles the schema constraints (NRIs) into the associations and, hence, into the mappings we generate. As a consequence, the resulting set of mappings $\mathcal{M}$ "includes" the effect of the source and target schema constraints. Since in the algorithms presented in this chapter, we do not consider other target constraints (e.g., key constraints or functional dependencies), we reduce the data exchange problem to one based on $\mathcal{M}$ alone.

The other main difference is that we perform data exchange by generating queries that "implement" $\mathcal{M}$. The main property of our query generation algorithm is that for relational schemas the instance J_u generated by our data exchange queries is always a *universal solution*. Indeed, any other solution J will agree with the instance J_u on the elements for which correspondences have been defined (the elements determined by the mapping). For all the other elements,

the data exchange queries have generated Skolem values that can be mapped to the corresponding values of instance J. Thus, there is a homomorphism from J_u to any other solution J, which means that J_u is a universal solution.

For nested relational schemas, we additionally took the view that the mapping $\mathcal{M}$ also specifies some implicit grouping conditions on the target data. In particular, we required that the target instance must be in partitioned normal form (PNF): in any set we cannot have two distinct tuples that agree on all the atomic valued components but do not agree on the set-valued elements. Our query generation algorithm has then the property that the data exchange queries we produce will always generate a target instance that is a PNF version of a universal solution. Alternatively, the generated target instance is a PNF solution that is universal with respect to all PNF solutions.

7 Related Work

Declarative schema mapping formalisms have been used to provide formal semantics for data exchange [19], data integration [33], peer data management [28, 13, 25], pay-as-you-go integration systems [45], and model management operators [8]. A whole area of model management has focused on such issues as mapping composition [35, 21, 40] and mapping inverse [18, 22].

Schema mappings are so important in information integration that many mapping formalisms have been proposed for different tasks. Here we mention only a few. The important role of Skolem functions for merging data has been recognized in a number of approaches [30, 41]. HePToX [13] uses a datalog-like language that supports nested data and allows Skolem functions. Extensions to the languages used for schema mapping include nested mappings [24], which permit the declarative specification of correlations between mappings and grouping semantics, including the PNF grouping semantics used in Clio. Clip [44] provides a powerful visual language for specifying mappings between nested schemas. And second-order tgds [21] provide a mapping language that is closed under composition.

Perhaps the only work on mapping discovery that predates Clio is the TranSem system [39] which used matching to select among a pre-specified set of local schema transformations that could help transform one schema into another. Work on mapping discovery has certainly continued. Fuxman et al. [24] consider how to create nested mappings. An et al. [5] consider how to use conceptual schemas to further automate mapping discovery. Yan et al. [51] and Alexe et al. [3] consider how to use data examples to help a user interactively design and refine mappings for relational and nested schemas respectively. Hernández et al. [29] consider the generation of mappings that use data-metadata translations [50]. Also, Bohannon et al. [12] consider the generation of information preserving mappings (based on path mappings).

Analysis of mappings has become an important new topic with work on verification of mappings [14], mapping quality [15, 32], and mapping optimization [20], to name just a few.

Many industry tools such as BizTalk Mapper, IBM WebSphere Data Stage TX, and Stylus Studio's XML Mapper support the development (by a programmer) of mappings. Although these tools do not automate the mapping discovery process, they do provide useful programming environments for developing mappings. The Clio algorithms for mapping discovery are used in several IBM products, including IBM Rational Data Architect, and IBM InfoSphere FastTrack. STBenchmark [4] presents a new benchmark for schema mapping systems.

Clio was the first system to generate code (in our case queries) for data exchange. The generation of efficient queries for data exchange is not considered in work like Piazza [28] and HePToX [13] which instead focus on query generation for data integration. More recently, in model management [37, 10], query or code generation for data exchange has been considered for embedded dependencies. Hernández et al. [29] generate data exchange queries for richer mappings that include data to metadata conversion. And specialized engines for efficiently executing data exchange queries have been proposed [31].

8 Conclusions

We have presented a retrospective on key contributions of the Clio schema mapping system. These innovations include a new paradigm, in which we view the mapping creation process as one of query discovery. Clio provides a principled algorithm for discovering queries over the source, queries over the target, and a precise specification of their relationship. In Clio, we pioneered the use of schema mapping queries to capture the relationship between data in two heterogeneous schemas. Many uses have been found for such schema mappings, but Clio was the first system to exploit them to perform data exchange between independently created schemas, leading to a new theory of data exchange. In this chapter, we have presented our algorithms for both schema mapping creation via query discovery, and for query generation for data exchange. Our algorithms apply equally in pure relational, pure XML (or any nested relational), and mixed relational and nested contexts.

Clio set out to radically simplify information integration, by providing tools that help users convert data between representations – a core capability for integration. Today, most information integration systems, whether federation engines that do data integration, or ETL engines that enable data exchange, include a suite of tools to help users understand their data and to create mappings, though only a few leverage the power of Clio's algorithms. In the next generation of integration tools, we need to leverage data and semantic metadata more effectively in the integration process, combining data-driven, metadata-driven and schema-driven reasoning. Further, we need to provide users with a higher level of abstraction for the entire integration process, from identification of the data of interest through returning the results. Ideally, users would not have to decide *a priori* whether they wanted data integration or data exchange; instead, the system should understand the user's intentions and construct the integration

plan accordingly [26]. These challenges are natural next steps along the trail of increased automation and radical simplification blazed by Clio.

References

1. S. Abiteboul and N. Bidoit. Non-first Normal Form Relations: An Algebra Allowing Data Restructuring. *J. Comput. Syst. Sci.*, 33:361–393, Dec. 1986.
2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
3. B. Alexe, L. Chiticariu, R. J. Miller, and W.-C. Tan. Muse: Mapping understanding and design by example. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 10–19, 2008.
4. B. Alexe, W.-C. Tan, and Y. Velegrakis. STBenchmark: towards a benchmark for mapping systems. *Proceedings of the VLDB Endowment*, 1(1):230–244, 2008.
5. Y. An, A. Borgida, R. J. Miller, and J. Mylopoulos. A Semantic Approach to Discovering Schema Mapping Expressions. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 206–215, 2007.
6. C. Batini, M. Lenzerini, and S. B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4):323–364, Dec. 1986.
7. C. Beeri and M. Y. Vardi. A proof procedure for data dependencies. *J. ACM*, 31(4):718–741, 1984.
8. P. Bernstein, A. Halevy, and R. Pottinger. A Vision for Management of Complex Models. *SIGMOD Record*, 29(4):55–63, Dec. 2000.
9. P. A. Bernstein and L. M. Haas. Information Integration in the Enterprise. *Commun. ACM*, 51(9):72–79, 2008.
10. P. A. Bernstein, S. Melnik, and P. Mork. Interactive Schema Translation with Instance-Level Mapping. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1283–1286, 2005.
11. P. Bohannon, E. Elnahrawy, W. Fan, and M. Flaster. Putting Context into Schema Matching. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 307–318, 2006.
12. P. Bohannon, W. Fan, M. Flaster, and P. P. S. Narayan. Information Preserving XML Schema Embedding. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 85–96, 2005.
13. A. Bonifati, E. Q. Chang, T. Ho, V. S. Lakshmanan, and R. Pottinger. HePToX: Marrying XML and Heterogeneity in Your P2P Databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1267–1270, 2005.
14. A. Bonifati, G. Mecca, A. Pappalardo, S. Raunich, and G. S. G. Schema mapping verification: the spicy way. In *International Conference on Extending Database Technology (EDBT)*, pages 85–96, 2008.
15. A. Bonifati, G. Mecca, A. Pappalardo, S. Raunich, and G. S. G. The spicy system: towards a notion of mapping quality. In *ACM SIGMOD Conference*, pages 1289–1294, 2008.
16. S. Chawathe, H. GarciaMolina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *Proc. of the 100th Anniversary Meeting of the Information Processing Society of Japan(IPSJ)*, pages 7–18, Tokyo, Japan, Oct. 1994.

17. A. Deutsch and V. Tannen. Xml queries and constraints, containment and reformulation. *Theoretical Comput. Sci.*, 336(1):57–87, May 2005.
18. R. Fagin. Inverting schema mappings. *ACM Transactions on Database Systems (TODS)*, 32(4):25, 2007.
19. R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *Theoretical Comput. Sci.*, 336(1):89–124, May 2005.
20. R. Fagin, P. G. Kolaitis, A. Nash, and L. Popa. Towards a theory of schema-mapping optimization. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 33–42, 2008.
21. R. Fagin, P. G. Kolaitis, L. Popa, and W. Tan. Composing schema mappings: Second-order dependencies to the rescue. *ACM Transactions on Database Systems (TODS)*, 30(4):994–1055, 2005.
22. R. Fagin, P. G. Kolaitis, L. Popa, and W.-C. Tan. Quasi-inverses of schema mappings. *ACM Transactions on Database Systems (TODS)*, 33(2):1–52, 2008.
23. M. J. Franklin, A. Y. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Record*, 34(4):27–33, 2005.
24. A. Fuxman, M. A. Hernández, H. Ho, R. J. Miller, P. Papotti, and L. Popa. Nested Mappings: Schema Mapping Reloaded. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 67–78, 2006.
25. A. Fuxman, P. G. Kolaitis, R. Miller, and W.-C. Tan. Peer Data Exchange. *ACM Transactions on Database Systems (TODS)*, 31(4):1454–1498, 2006.
26. L. M. Haas. Beauty and the beast: The theory and practice of information integration. In *Proceedings of the International Conference in Database Theory (ICDT)*, pages 28–43, 2007.
27. L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Tork Roth. Clio grows up: From research prototype to industrial tool. In *ACM SIGMOD Conference*, pages 805–810, 2005.
28. A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The piazza peer data management system. *IEEE Transactions On Knowledge and Data Engineering*, 16(7):787–798, 2004.
29. M. A. Hernández, P. Papotti, and W.-C. Tan. Data exchange with data-metadata translations. *Proceedings of the VLDB Endowment*, 1(1):260–273, 2008.
30. R. Hull and M. Yoshikawa. ILOG: Declarative Creation and Manipulation of Object Identifiers. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 455–468, 1990.
31. H. Jiang, H. Ho, L. Popa, and W.-S. Han. Mapping-driven xml transformation. In *Proceedings of the International WWW Conference*, pages 1063–1072, 2007.
32. L. Jiang, A. Borgida, and J. Mylopoulos. Towards a compositional semantic account of data quality attributes. In *Proceedings of the International Conference on Conceptual Modeling (ER)*, pages 55–68, 2008.
33. M. Lenzerini. Data Integration: A Theoretical Perspective. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 233–246, 2002.
34. A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 251–262, 1996.
35. J. Madhavan and A. Y. Halevy. Composing Mappings Among Data Sources. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 572–558, 2003.
36. D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing Implications of Data Dependencies. *ACM Transactions on Database Systems (TODS)*, 4(4):455–469, 1979.

37. S. Melnik, P. A. Bernstein, A. Halevy, and E. Rahm. Applying model management to executable mappings. In *ACM SIGMOD Conference*, pages 167–178, 2005.
38. R. J. Miller, L. M. Haas, and M. Hernández. Schema Mapping as Query Discovery. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 77–88, 2000.
39. T. Milo and S. Zohar. Using Schema Matching to Simplify Heterogeneous Data Translation. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 122–133, 1998.
40. A. Nash, P. A. Bernstein, and S. Melnik. Composition of mappings given by embedded dependencies. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 172–183, 2005.
41. Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 413–424, 1996.
42. L. Popa and V. Tannen. An Equational Chase for Path-Conjunctive Queries, Constraints, and Views. In *Proceedings of the International Conference in Database Theory (ICDT)*, pages 39–57, 1999.
43. L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating Web Data. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 598–609, Aug. 2002.
44. A. Raffio, D. Braga, S. Ceri, P. Papotti, and M. A. Hernández. Clip: a Visual Language for Explicit Schema Mappings. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 30–39, 2008.
45. M. A. V. Salles, J.-P. Dittrich, S. K. Karakashian, O. R. Girard, and L. Blunschi. itrails: Pay-as-you-go information integration in dataspaces. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 663–674, 2007.
46. N. C. Shu, B. C. Housel, and V. Y. Lum. Convert: A high level translation definition language for data conversion. *Commun. ACM*, 18(10):557–567, 1975.
47. N. C. Shu, B. C. Housel, R. W. Taylor, S. P. Ghosh, and V. Y. Lum. EXPRESS: A Data EXtraction, Processing and REstructuring System. *ACM Transactions on Database Systems (TODS)*, 2(2):134–174, 1977.
48. Y. Velegrakis. *Managing Schema Mappings in Highly Heterogeneous Environments*. PhD thesis, Department of Computer Science, University of Toronto, 2004.
49. Y. Velegrakis, R. J. Miller, and L. Popa. On Preserving Mapping Consistency under Schema Changes. *International Journal on Very Large Data Bases*, 13(3):274–293, 2004.
50. C. M. Wyss. and E. L. Robertson. Relational languages for metadata integration. *ACM Transactions on Database Systems (TODS)*, 30(2):624–660, 2005.
51. L.-L. Yan, R. J. Miller, L. Haas, and R. Fagin. Data-Driven Understanding and Refinement of Schema Mappings. *ACM SIGMOD Conference*, 30(2):485–496, 2001.
52. C. Yu and L. Popa. Constraint-Based XML Query Rewriting For Data Integration. *ACM SIGMOD Conference*, 33(2):371–382, 2004.