Translating Web Data

Lucian Popa Yannis Velegrakis Renee Miller Mauricio Hernandez
Ronald Fagin

Technical Report CSRG-441
Deprtment of Computer Science
University of Toronto
February 2002

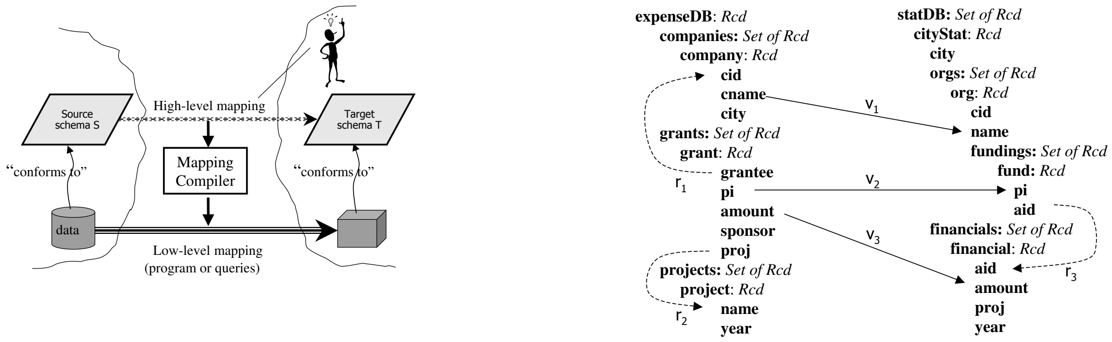
Abstract

Mapping and translating data stored in different formats continues to be an important problem in modern information systems. We present a novel framework for mapping among XML and relational schemas in which a high-level mapping is translated into semantically meaningful queries that transform source data into the target representation. Our approach works in two phases. In the first phase, a high-level mapping, expressed as a set of attribute-to-attribute correspondences, is processed and converted into a logical mapping that captures the design choices made in the source and target schemas (including their hierarchical organization and the grouping of attributes into nested tables and sets). The second phase translates the logical mapping into a query that can be executed over the source schemas and is guaranteed to produce data satisfying the constraints and structure of the target schema. To this end, target attribute values may need to be invented to ensure that the data respects the constraints (including nested referential constraints) and the (possibly nested) structure of the target schema. Our approach is unique in that 1) we consider not only relational schemas, but also XML schemas with (nested) constraints; 2) for this large class of schemas, the mapping algorithm is complete in that it produces **all** mappings that are consistent with the schema constraints; 3) our data translation algorithm correctly translates source data even if there is missing data in the target (attributes with no correspondence to the source). We have implemented the mapping algorithm in a high-level schema mapping tool.

1 Introduction

An important issue in modern information systems and e-commerce applications is providing support for interoperability of independent data sources. A broad variety of data is available on the Web in distinct heterogeneous sources, stored under different formats: database formats (relational), document formats (SGML/XML), browser formats (HTML), scientific data, etc. Integration of such data is an increasingly important problem. Nonetheless, the effort involved in such integration, in practice, is considerable: translation of data from one format (or schema, in database terminology) to another requires writing and managing complex data transformations programs or queries.

To shield users from manually performing this task for every translation problem at hand, we advocate the use of high-level schema mapping tools. In such a tool, a *high-level mapping* is specified using *correspondences*, which map attributes of a source schema to attributes of a target schema. This specification is independent of logical design choices such as the grouping of attributes into tables (normalization choices) or the nesting of records or tables (for



(a) (b)

Figure 1: (a) A high-level schema mapping tool. (b) A schema mapping scenario.

example, the hierarchical structure of an XML schema). In other words, one need not specify the logical access paths (join or navigation) that define the associations between attributes involved. Therefore even users that are unfamiliar with the complex structure of the schema can easily use such a tool. The difficulty is then to discover in an automatic way all the semantic associations that exist among the attributes of the schemas and, based on them, to generate a meaningful data translation program: a *low-level mapping*. The high-level schema mapping tool acts then as a *mapping compiler* (see Figure 1(a)) with the role of finding the correct low-level mapping.

The efficacy of supporting attribute-to-attribute correspondences is greatly increased by the fact that they need not be specified by a human user. They could be in fact the result of an automatic component that matches the attributes of the two schemas [DDH01, MBR01] (but does not associate any semantic meaning). In this paper, we study the design of a mapping compiler that takes a set of correspondences and schema constraints as input, and produces a semantically meaningful data transformation program.

Example 1.1 *Consider the two schemas in Figure 1(b). For convenience, both schemas are shown in a common nested relational representation*[1]*. The left-hand schema represents a source relational schema with three tables:* `company(cid, cname, city)`*,* `grant(grantee, pi, amount, sponsor, proj)`*, and* `project (name, year)`*. It describes information about companies, their projects and the funding (grants) given for those projects. Companies have a company id (*`cid`*), a name (*`cname`*), and reside in a city (*`city`*). Each grant is given to a company for a specific project. Therefore, each* `grant` *tuple has foreign keys (*`grantee` *and* `proj`*) referencing the associated* `company` *and* `project` *tuples. Foreign keys are shown as dashed lines. In addition, a grant has a principal investigator (*`pi`*), an amount (*`amount`*), and a sponsor (*`sponsor`*). The right-hand schema represents a target XML schema. While the information that the target contains is similar to that of the source, the data is structured in a different way. Organizations and projects are organized by city. For each different city, there is an element* `cityStat` *containing the organizations (*`org`*) and the grants (*`financials`*) in that city. The information about project funding is nested within* `org` *and related with the* `financial` *information through a foreign key based on the* `aid` *element.*

Given the two schemas, a user enters correspondences (such as v_1, v_2, and v_3 shown in the figure) indicating where data values from the source should appear in the target. Alternatively, we may use a tool that uses linguistic reasoning to match attributes based on their names or their place in a hierarchical structure [MBR01]. If some data is already stored under the target schema, we may also use a data mining technique that matches data values or their characteristics to find such correspondences [DDH01, NHT[+]*02].* ∎

[1]Which is in fact adopted by our mapping compiler and used throughout this paper.

Logical Mappings In our approach, the compilation has two main phases. The first one, **semantic translation**, generates an intermediate *logical mapping* representation that is a precise and faithful understanding of the high-level mapping. Since correspondences specify only attribute-to-attribute mappings, they are easy to create and manipulate by users (or by an automatic tool). However, to understand the data translation implied by the given correspondences, one must identify *all* the different ways in which attributes in each of the two schemas are related.

Example 1.2 *For the mapping scenario of Figure 1(b), we must understand how to associate a company name with a principal investigator of a grant. It is unlikely that all combinations of* `cname` *and* `pi` *values are semantically meaningful. Rather, we make use of the semantic information embedded in the source schema to determine which combinations of values are meaningful. The foreign key constraint between* `grant` *and* `company` *indicates that each* `grant` *(and its* `pi`*) is naturally associated with one* `company` *(through a foreign key join). To populate the target, that is to place values of* `cname` *and* `pi` *correctly in the target schema, we use the semantic information expressed in the target schema. In this example, the semantics is conveyed not through constraints, but through the nesting structure of the target. Specifically,* `fund` *tuples are nested within* `org` *tuples. Hence, for each* `org` *(that is, for each company name), we must group together all* `pi` *values into a separate* `fundings` *set.*

In general, there are many semantic associations in a schema, and even the same set of attributes could be associated in more than one way. In our example, there may be many ways of associating `cname` *and* `pi`*. Suppose that a* `sponsor` *is always a* `company`*, so* `sponsor` *is a foreign key for* `company`*. We could then associate* `cname` *of companies either with* `pi` *of grants they have been given (a join on* `grantee`*), or with* `pi` *of grants they sponsor (a join on* `sponsor`*). The choice depends on semantics that are not represented in the source schema and must instead be given by a user. Moreover, according to the source schema, data in the source may include companies that have no grants associated with them. Thus,* `cname` *can be part of an association by itself, and hence could give rise to a meaningful mapping that translates only company names to the target.* ∎

Semantic translation works in two steps. First, we enumerate the *logical access paths* that exist in each schema. Such paths are defined by using two basic forms of relationships that exist in relational and nested schemas: 1) parent/child (specific to nested/XML schemas) and 2) foreign key/key (for both relational and XML schemas). Moreover, the generated paths ensure that no integrity or structural constraints of the schema are violated. For example, if a logical path includes a relation that has a (non-nullable) foreign key into another relation then the second relation is also included in the path. Also, a path that includes an element that is a child of another element, according to a hierarchical schema, includes the second element (the parent) as well. Semantic *associations* (we also call them *relationships*) among attributes are discovered based on whether the attributes lie within a same logical path. For example, `cname` of `company` and `pi` of `grant` belong to the same source relationship because they are both part of the foreign key join of `grant` and `company`. A second relationship between them is discovered if the additional foreign key constraint on `sponsor` is given. The second step of semantic translation generates, based on the given correspondences, all possible ways of mapping source relationships to target relationships. The result is a *set* of logical mappings. Each of them is a meaningful and *different* mapping.

Enumeration of all such mappings is an essential ingredient of our approach. As we have seen in the example, any *one*, any *subset* or *all* of the mappings could correspond to the user's intentions for a given pair of schemas and their correspondences. The entire process of semantic translation is therefore a semi-automatic process. The system generates *all* logical mappings consistent with the specification and the user chooses a subset of them. Our experience with real schemas has shown that the number of such mappings is typically not large (six or more often less) and that there are real situations in which the intended semantics includes all these mappings. Hence, a heuristic approach that prunes some consistent mappings will not work in general. To reduce the burden on the user, in work reported elsewhere, we have developed innovative techniques for both explaining each alternative mapping in a natural way and for ordering mappings so that users can focus on the most likely mapping [YMHF01].

To uniformly represent schemas from both the relational and XML worlds we use a variation of the nested relational model. On this data model, we are able to represent both relational and XML foreign key/key constraints, which are essential for defining associations. The logical mappings themselves are represented by a nested extension of the relational data dependencies[FV86]. Dependencies are both simple and precise. They are declarative assertions specifying how data instances of a source association correspond to data instances of a target association. As an internal representation of the mapping, dependencies are simpler than queries and therefore easier to manipulate and optimize.

Data Translation The second phase of our mapping compiler is concerned with *data translation*, i.e., implementing the specification given by the logical mappings. The result of this phase is a query (or rather, set of queries, one for each logical mapping) expressed in a specific data transformation language (e.g., SQL or XQuery). For each logical mapping, the generated query has two roles: *retrieve* and *insert*. First it retrieves the data instances of the source association by performing the required join and unnest operations (i.e. following the source logical path). The second part is the hard one: we must correctly insert values for the attributes of the target association in their actual places in the target schema. This requires operations that are inverse in nature to the ones used in the retrieve phase: nest operations, and partitioning of data into multiple tables or sets (the inverse of join!)

To correctly generate such a query, we must be able to reconcile and translate logical design choices made in the two schemas as well as differences in the data content. We do not assume the source and target schema represent the same data. Hence, there may be data in the target that is not represented in the source. In some cases, values must be produced for undetermined attributes in the target schema (i.e., target attributes for which no correspondence was given). Values may be needed if the target attribute cannot be null and no default is given. More importantly, the creation of new values for such target attributes may be essential for ensuring the consistency of the target data (e.g., we create any foreign keys and keys in the target that are required to ensure source data is correctly mapped). Our query generation algorithm provides a new solution for automatically generating such target values (ids), based on Skolem functions.

Example 1.3 *In the scenario of Figure 1(b),* `pi` *and* `amount` *of* `grant` *are mapped, via* v_2 *and* v_3*, to* `pi` *of* `fund` *and* `amount` *of* `financial`*. The foreign key from* `aid` *of* `fund` *to* `aid` *of* `financial` *indicates that* `pi` *values are associated with* `amount`*. Thus, the semantic translation algorithm will generate a logical mapping that includes both* v_2 *and* v_3 *(*v_1 *as well, in the example, but let us focus on* v_2 *and* v_3*). This logical mapping specifies a data partitioning operation. However, to populate the target, we must have values for the two* `aid` *attributes. To maintain the proper association in the target, these values may not be arbitrary and cannot be null either. However, as is often the case with elements that carry structural information but no real data, there is no correspondence that maps into* `aid` *from the source. Our solution is to invent id values in a way that maintains the association. The invented value should be the same for the* `aid` *of the* `financial` *and the* `aid` *of the* `fund`*, but different for each such pair. Consider, for example, two* `grant` *tuples in the source:* $[\texttt{pi} = \textit{Gerstner}, \texttt{cid} = \textit{IBM}, \texttt{proj} = \textit{DB2}, \texttt{sponsor} = \textit{NSF}, \texttt{amount} = \textit{\$100K}]$ *and* $[\texttt{pi} = \textit{Gates}, \texttt{cid} = \textit{MSFT}, \texttt{proj} = \textit{SQLServer}, \texttt{sponsor} = \textit{NSF}, \texttt{amount} = \textit{\$200K}]$*. When the first tuple is mapped to the target, it will generate the* `fund` *tuple* $[\texttt{pi} = \textit{Gerstner}, \texttt{aid} = G_1]$ *and the* `financial` *tuple* $[\texttt{aid} = G_1, \texttt{amount} = \textit{\$100K}, \texttt{proj} = \textit{null}, \texttt{year} = \textit{null}]$*. Similarly for the second* `grant` *tuple, the mapping will generate the* `fund` *tuple* $[\texttt{pi} = \textit{Gates}, \texttt{aid} = G_2]$ *and the* `financial` *tuple* $[\texttt{aid} = G_2, \texttt{amount} = \textit{\$200K}, \texttt{proj} = \textit{null}, \texttt{year} = \textit{null}]$*. If the generated value* G_2 *was not different than* G_1*, then the target database would assert, wrongly, that Gerstner and Gates are both principal investigators for two funds, one of* $\$200K$ *and one of* $\$100K$*.* ∎

The result of the compilation is a set of queries on the source schema that are guaranteed to produce data satisfying the constraints and structure of the target schema. Our solutions have been implemented in an interactive high-level schema mapping tool for creating and managing mappings between complex heterogeneous schemas.

There are two main contributions in this paper. In Section 4, we present a semantic translation algorithm (Algorithm 4.15) that interprets attribute-to-attribute correspondences as logical mappings. Each logical mapping transforms one semantic relationship of the source into one semantic relationship of the target. Thus, our algorithm preserves[2] semantic relationships during translation from one schema to the other. Preservation of semantic relationships is something unique to our approach: previous work in the area of semi-automatic generation of data transformations ([MHH00, YMHF01]) was concerned only with discovering source semantic relationships, thus ignoring half of the more general problem. In their case, the target schema can consist only of relations with no connections (foreign key/key constraints) among them. Then, in Section 6.1, we present a query generation algorithm that compiles each logical mapping into a query enhanced with rich restructuring capabilities (resembling ILOG [HY90], WOL [DK97] and XML-QL [DFF+99]). Most of the previous work in the area of (semi-)automatic mapping of schemas was mostly concerned with *schema matching*. When it did consider data translation, it mostly ignored the complexity of the target schemas (target constraints and/or nesting); thus it was focused on the *retrieve* part. Our solution correctly maps partial information from the source and uses id invention to ensure no loss of information.

Our approach generates *only* mappings between (any) one semantic relationship of the source and (any) one semantic relationship of the target. We believe that semantic relationships (as implied by the constraints of a schema) are the fundamental relationships in a schema we hence focus on *their* correct translation. Moreover, an automatic tool can enumerate only a finite space of mappings of possible interest. The more "sophisticated" cases of data transformations (of which there are infinitely many, in general) must be provided by a human user with expert knowledge in the translation problem of interest. Although rare, we argue that such transformations are typically restrictions (or selections) on semantic relationships. In our system such restrictions can be added, using filters (additional selection and join predicates), with minimal effort by a user.

Other Contributions: In Section 5, we present a completeness guarantee that all semantic relationships[3] that exist between attributes in a schema are discovered by the semantic translation algorithm. In addition, we prove that the logical mappings we produce will satisfy the structure of the target schema, including the target referential integrity constraints. In Section 6, we present a correctness guarantee that the data translation query we produce preserves information. Hence, all the source data covered by the high-level mapping is transformed without loss into the target. We also show that the query result (the target instance) is in partition normal form [AB86, ÖY87, RK87] thus the redundancy in the target is minimal. In Section 7, we describe our experience mapping real schemas using our prototype. We discuss how the design and principles underlying our mapping algorithm have been effective in managing and controlling the complexity of the mapping task in a real Life Science application.

2 Related Work

In *schema integration*, an integrated (global) schema is created from a set of source (local) schemas. Either this integration is expressed as a view on the local schemas in the *Global As View* approach or the local schemas are expressed as views on the global schema in the *Local As View* approach (see [RR99] for a survey of schema integration). Since the integration is defined from the source schemas, its constraints and scope (data content) match those of the sources. For example, there will not be any data in the integration that is not created from some source. The mappings (views) between the schemas are typically given (either by a user or as an artifact of

[2]With respect to the given correspondences.

[3]Defined in a precise way and independently of our algorithm.

the integration process) and the goal of much data integration research is to express, optimize and evaluate queries on the global schema using the local schemas. In contrast, in *schema mapping* (which is the approach we follow) the view definitions (mappings) are not given in advance, and the main goal is not query translation but rather mapping (query) creation [MHH00]. Furthermore, we have no design freedom in choosing a target schema that matches the source. Hence, we must be concerned with differences in both constraints and the scope (data content) between the source and target schemas. Previous work on schema mapping considered the creation of mappings between only relational schemas and ignored target constraints [MHH00, YMHF01]. Hence, the data translation was not guaranteed to be a valid instance of the target schema. In addition, if the target contained attributes with no correspondence to the source, the translation was not necessarily information preserving. We take a more principled approach and present a mapping algorithm that is complete in that it generates all consistent mappings, rather than heuristically choosing among possible mappings.

Research on *data translation* has been mostly focussed on translation languages rather than on automating the creation of programs or queries expressed in these languages. Data translation languages, including rule-based correspondence languages, have powerful pattern matching primitives that can facilitate data restructuring and translation [ACM97, CDSS98, AT97, DK97, HY90]. Of these approaches, only WOL [DK97] takes into consideration integrity constraints in the language design. We make use of restructuring primitives inspired by these languages, including most notably Skolem functions, similar to those of ILOG [HY90]. One of our contributions is that we *automatically* determine the correct set of Skolem functions to use in the mapping in order to respect source and target constraints and preserve information; a problem that has not been considered in other data translation work.

In addition there has been considerable work on translating data between data models, including many proposals for XML to relational translations (and vice versa). That work creates the translated schema and does not consider translation into a fixed target that may not match the source.

The TranScm [MZ98] system was one of the first to considered the use of schema information to help automate data translation, and is perhaps the closest to our approach. TranScm identifies matches between two schemas and derives the translation functions. The approach is based on a set of predefined matching rules that describe the most commonly used transformations. The rules are mostly structural, ignoring constraints. In addition, the expressive power of the rules is limited in that they can detect only local structural differences. More advanced mappings that include joining disparate portions of the schema cannot be created (and indeed were not the focus of that work). In contrast, our approach can generate arbitrary join queries with nesting and unnesting.

Other work has considered *schema matching*, the automatic detection of parts of two schemas that may model the same real world entities (see [RB01] for a survey of this work). The output of schema matching is typically not a set of queries, but rather a high-level indication of how portions of the schemas are related. This work complements our results by providing a method for producing an initial set of attribute correspondences (which can later be enhanced or modified by a user). Our techniques generate an executable data translation program from these schema level matches.

Finally, work on *logical data independence* also considered the problem of specifying queries in a high-level form that is independent of specific logical design decisions. Work on *Universal Relation* interfaces is probably the best known [MUV84, FMU82] in this area. Our high-level attribute-to-attribute correspondences very much resemble Universal Relation queries. However, because of our application, we cannot assume anything about the design of either the source or target schema. Hence, we do not assume the *one flavor* assumption (also known as the *universal relation assumption*). Our solutions extend those developed for universal relation systems to include nested schemas and constraints (extending even results for the Nested Universal Relation Model [LL94]). Perhaps more importantly, our solutions are complete, for not just well-designed relational schemas (satisfying URA), but for ambiguous queries over schemas with many different semantic associations between the same attributes. Because we are considering schema mapping into a heterogeneous target, rather than query answering, we also provide solutions for translating data in an information preserving way using our compiled queries.

3 A Nested Relational Model

In this section we present the nested data model that is used in our schema mapping framework. Its main features are the ability to represent nested structures (through nested sets and records), and the presence of nested referential integrity constraints (used for either relational or XML schemas). Constraints are described in the next section.

Types and Values. Let $\underline{b}$ represent a basic data type, such as `int`, `string`, or `date`, and let $dom(\underline{b})$ represent the (possibly) infinite domain of values for $\underline{b}$. Each such basic type is extended in our data model by adding a set $O_{\underline{b}}$ of **oid** values to the domain of possible values. Thus an atomic type in our model is of the form $\tau_{\underline{b}}$ and its domain is $dom(\tau_{\underline{b}}) = dom(\underline{b}) \cup O_{\underline{b}}$. Oids are special atomic values that are used to provide values for the parts of the target schema that are not determined (via correspondences) by any of the source attributes. For schema mapping purposes, we can assume that atomic oids will be present only in the target.

Non-atomic types in our nested data model are set types of the form $\texttt{SetOf}\,[\tau]$, and record types of the form $\texttt{Rcd}[\texttt{A}_1 : \tau_1, \ldots, \texttt{A}_k : \tau_k]$, where τ represents either an atomic, set, or record type (i.e., non-atomic types can be nested in any way). The symbols $\texttt{A}_1, \ldots, \texttt{A}_k$ are called *labels* or *attributes*. Record values of type $\texttt{Rcd}[\texttt{A}_1 : \tau_1, \ldots, \texttt{A}_k : \tau_k]$ are unordered tuples of label-value pairs: $[A_1 = a_1, \ldots, \texttt{A}_k = a_k]$, where $a_1, \ldots, a_k$ must be of types $\tau_1, \ldots, \tau_k$, respectively.

A value of type $\texttt{SetOf}\,[\tau]$ is, however, *a set oid*, and not a set in the traditional sense. Each type $\texttt{SetOf}\,[\tau]$ has an infinite domain $O_{\texttt{SetOf}\,[\tau]}$ of such set oids. Then each set oid S in $O_{\texttt{SetOf}\,[\tau]}$ is associated with a set $\{e_1, \ldots, e_n\}$ of "children", each child being of type τ. The association between a child e_i and the set oid S will be denoted by a *fact* $S(e_i)$. Another notation that we may use for a fact $S(e_i)$ is $e_i \in S$. This representation of sets (using set oids) is used to faithfully capture the graph-based data models of XML. In the nested constraints that we define in the next section, whenever we state the equality of two values that are of set type we will mean equality of the two set oids (and not of their sets of children).

Schemas and Instances. A schema $\mathcal{S}$ consists of several names (or roots) with their types, $\texttt{S}_1 : \tau_1, \ldots, \texttt{S}_n : \tau_n$. For schema mapping purposes we usually assume that the source and the target schema consist each of one single root. This is in conformance to XML Schema [Fal01] which allows for one single root. A relational schema with multiple tables can be modeled as one record type the components of which are set types (tables). We provide a straight-forward translation algorithm that maps XML schemas into our nested relational model. For instance, Figure 2 shows a fragment of the XML Schema definition used to create the target schema in Figure 1(b). Due to lack of space, we do not present the full translation algorithm. In the translation, all element tags (as well as attributes) in the XML schema appear as labels in some record types. Elements that can be repeated, according to the XML schema, such as `org`, are grouped within explicit set types (`orgs`). When the XML schema does not provide a tag for the set containing the repeated elements, we provide a *virtual* label (e.g. `orgs`). This label is removed in the final XQuery that realizes the mapping (see Section 6.3). Our nested relational model accounts for optional and nullable elements. In Section 8, we consider support for additional XML constructs such as ordering and union types.

An instance I consists of a set of facts. Figure 3(a) depicts one such instance for the `statDB` schema presented before. In this instance, the set oid `statDB` is the root and is associated with two tuples (the first two facts in

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
 <xsd:element name="statDB">
   <xsd:complexType> <xsd:all>
     <xsd:element name="cityStat" minOccurs="0" maxOccurs="unbounded">
       <xsd:complexType> <xsd:all>
         <xsd:element name="city" nullable="true" type="xsd:string"/>
         <xsd:element name="org" minOccurs="0" maxOccurs="unbounded">
           <xsd:complexType> <xsd:all>
             <xsd:element name="cid" type="xsd:string"/>    <xsd:element name="name" type="xsd:string"/>
             <xsd:element name="fund" minOccurs="0" maxOccurs="unbounded">
               <xsd:complexType> <xsd:all>
                 <xsd:element name="pi" type="xsd:string"/> <xsd:element name="aid" type="xsd:string"/>
               </xsd:all> </xsd:complexType>
           </xsd:all> </xsd:complexType>
         <xsd:element name="financial" minOccurs="0" maxOccurs="unbounded">
           <xsd:complexType> <xsd:all>
             <xsd:element name="aid" type="xsd:string"/>    <xsd:element name="amount" type="xsd:string"/>
             <xsd:element name="proj" minOccurs="0" type="xsd:string"/>
             <xsd:element name="year" nullable="true" type="xsd:string"/>   ...
```

Figure 2: XML Schema Example

$$\texttt{statDB}\ ([\texttt{cityStat} = [\texttt{orgs} = O_1,\ \texttt{financials} = F_4,\ \texttt{city} = NY]])$$

$$\texttt{statDB}\ ([\texttt{cityStat} = [\texttt{orgs} = O_2,\ \texttt{financials} = F_1,\ \texttt{city} = SJ]])$$

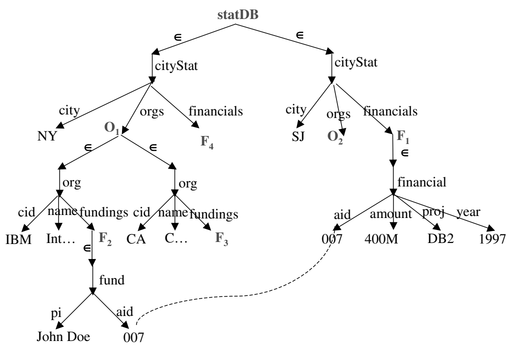
$$O_1([\texttt{org} = [\texttt{cid} = IBM,\ \texttt{name} = International...,\ \texttt{fundings} = F_2]])$$

$$O_1([\texttt{org} = [\texttt{cid} = CA,\ \texttt{name} = Computer...,\ \texttt{fundings} = F_3]])$$

$$F_2([\texttt{fund} = [\texttt{pi} = JohnDoe,\ \texttt{aid} = 007]])$$

$$F_1([\texttt{financial} = [\texttt{aid} = 007, \texttt{amount} = 400M, \texttt{proj} = DB2, \texttt{year} = 1997]])$$

(a)



(b)

Figure 3: (a): A nested relational instance. (b): Graph representation of the instance on the right.

Figure 3(a)). Each tuple has a `cityStat` component, which is another (nested) tuple with components `orgs` (a set oid), `financials` (also, a set oid), and `city` (a string). For the first `statDB` the `orgs` component is a set oid O_1. There may be multiple elements belonging to O_1. They would form a nested set (as opposed to the `statDB` tuples which form a top-level set). In the instance, there is only one tuple that belongs to O_1.

Alternatively, a useful way of representing a nested instance is as a labeled directed graph; see Figure 3(b). There is exactly one node in the graph for each distinct value in the instance. The root of the graph is `statDB`. The relationship between a record node and its components are denoted by edges labeled with the attribute. The relationship between a set oid and any of its elements is denoted by an $\in$-edge. There is an $\in$-edge between node x and node S whenever $S(x)$ is a fact in the instance. Because of the types that are imposed on values, the graph is acyclic. However, sharing of nodes is possible. For example, 007 is shared (and the dotted line in the figure is intended to show that the two nodes for the value 007 are really the same).

4 Semantic Translation

To perform schema mapping, we must interpret correspondences in a way that is consistent with the logical design of both the source and target schemas. We call this interpretation process *semantic translation*. Since correspondences may be ambiguous, we also seek to enumerate all interpretations that are consistent and let a user choose the appropriate mapping (perhaps by augmenting the semantics embedded in the schema constraints). In this section, we present our techniques for first representing, then creating, these logical interpretations. To present our ideas, we first present a data structure (*tableau*) that we use in the representation of relationships between combinations of atomic types in a schema, as well as in the representation of logical design choices (referential integrity constraints and nesting) in a schema. We then give our algorithm for enumerating all relationships that are consistent with the semantics of a schema.

4.1 Preliminaries

In relational schemas, connections between facts (tuples) are represented in two ways. First, the organization of attributes into tables indicates logical groupings. In addition, attributes within different tables may be associated using foreign key dependencies. For example, information about companies, the grants they hold and the projects for which they receive grants may, in a logical design, be broken up into separate tables using foreign key dependencies. To represent how tuples may be assembled into larger tuples, we use the traditional relational notion of a *tableau* [ASU79] (however, with a different notation style that is usable in the case of nesting as well).

Relational Expressions. If x is a variable over relation R then $x.A$ is an expression, for each attribute A in R.

Relational Tableaux. Let $\vec{x} = \langle x_1, \ldots, x_n \rangle$ be a non-empty sequence[4] of variables over relations $R_1, \ldots, R_n$. A tableau has the following form:

$$T \ ::= \ \langle \ x_1 \in R_1, x_2 \in R_2, \ldots, x_n \in R_n; \ B \ \rangle$$

Here B is a (possibly empty) conjunction of equalities between relational expressions, i.e. B is of the form $\bigwedge_j (e'_j = e''_j)$, where these expressions use only the variables in $\vec{x}$. For the cases when B is empty (meaning *true*) we write nothing after the semi-colon sign in the tableau.

Example 4.1 *To illustrate,* $\langle g \in \texttt{grant}, c \in \texttt{company};\ g.\texttt{cid} = c.\texttt{cid} \rangle$ *is a tableau over a relational schema with relations* `grant` *and* `company`*. This tableau represents a specific relationship between* `grant` *tuples and* `company` *tuples where tuples are related iff they share a common* `cid` *value.* ∎

Tableaux are simple query-like expressions[5] representing equality joins (despite the fancy name which we use for consistency with the DB literature). In a nested relational model, we need to be able to represent relationships between attributes at any place within a nested structure. To do this, we generalize the notion of expression to a simple form of path expression and we adopt a generalized notion of a tableau (as in [PT99]) that makes use of these expressions.

Expressions are defined by the following grammar: $\quad e \ ::= \ \texttt{S} \ | \ x \ | \ e.\texttt{A}$
where `S` is a schema root, x is a variable, and $e.\texttt{A}$ is record projection. We use the notation $e(\vec{x})$ to denote an expression that uses only the variables in $\vec{x}$. Each expression has a type (see Section 3) and we assume that expressions are well-formed (so, for example $e.\texttt{A}$ is an expression only if e is of record type with attribute `A`).

Example 4.2 *Example expressions from Figure 1(b) include* `expenseDB.grants` *and* x.`cityStat.city`*. The latter might be used if the variable x has been bound to* `statDB`*.* ∎

Tableaux over nested structures use variables bound not just to relations, but to any expression of set type within the nested structure. Furthermore, variables may be defined relative to other variables within the structure. Thus we follow the general definition of nested tableaux from [PT99].

Tableaux. Let $\vec{x} = \langle x_1, \ldots, x_n \rangle$ be a non-empty sequence of variables over expressions $e_1, \ldots, e_n$, of set types. An expression e_i may only use previously bound variables (x_j where $j < i$). A tableau has the following form:

[4] The order is not important for relational tableaux. It will become important for nested tableaux, defined later in this section.
[5] No output is specified. However, output expressions could be added and a tableau would become a query.

$$S_1 = \langle\, c \in \texttt{expenseDB.companies};\, \rangle$$
$$S_2 = \langle\, g \in \texttt{expenseDB.grants};\, \rangle$$
$$S_3 = \langle\, p \in \texttt{expenseDB.projects};\, \rangle$$

(a)

$$T_1 = \langle\, c \in \texttt{statDB};\, \rangle$$
$$T_2 = \langle\, c \in \texttt{statDB}, o \in c.\texttt{cityStat.orgs};\, \rangle$$
$$T_3 = \langle\, c \in \texttt{statDB}, o \in c.\texttt{cityStat.orgs}, f \in o.\texttt{org.fundings};\, \rangle$$
$$T_4 = \langle\, c \in \texttt{statDB}, f' \in c.\texttt{cityStat.financials};\, \rangle$$

(b)

Figure 4: Primary paths for: (a): relational schema `expenseDB`, (b): nested schema `statDB`.

$$T \ ::= \ \langle\, x_1 \in e_1, x_2 \in e_2, \ldots, x_n \in e_n;\ B \,\rangle$$

where B is a (possibly empty) conjunction of equalities, i.e. B is of the form $\bigwedge_j (e'_j = e''_j)$ over expressions that may use any of the variables in $\vec{x}$.

Example 4.3 *To illustrate, the tableau* $\langle g \in \texttt{expenseDB.grants}, c \in \texttt{expenseDB.companies};\ g.\texttt{grant.cid} = c.\texttt{company.cid}\rangle$ *is a tableau over the schema* `expenseDB` *of Figure 1(b). On schema* `statDB` *of the same figure, the tableau* $\langle c \in \texttt{statDB}, o \in c.\texttt{cityStat.orgs}, f \in o.\texttt{org.fundings};\rangle$ *represents a relationship between* `cityStat`*,* `orgs`*, and* `fundings`*.* ∎

We use tableaux to represent relationships, but obviously not all tableaux will form semantically meaningful relationships. Within a schema, there are several basic relationships. Certainly all attributes within a record can be viewed as being logically related. In addition, if records are nested within other records, there is a natural logical relationship between them. To represent these **basic** relationships, we use a simple form of tableaux called **primary paths**.

Primary paths are (linear) tableaux of the form: $P \ ::= \ \langle\, x_1 \in e_1(), x_2 \in e_2(x_1), \ldots, x_n \in e_n(x_{n-1});\ \rangle$
Hence, e_1 must be either a schema root or a sequence of projections of the schema root since it does not reference any variables.

The basic relationships in a schema are the primary paths of that schema. Figure 4 shows the primary paths for the two schemas of Figure 1(b). For a relational schema, the primary paths are simply the tables of the schema. For a nested schema, the primary paths are obtained by: 1) constructing a tree with a node at each set type in the schema, and with an edge between two nodes τ_1 and τ_2 whenever the first node, τ_1, is a set type that directly contains[6] τ_2, and 2) enumerating all paths from the root to any intermediate node or leaf.

For our nested schema example, `statDB`, the first primary path T_1 intuitively denotes the set of `cityStat` records (within `statDB`) that may or may not have organizations or financials. T_2 denotes `cityStat` records that do have organizations (although these organizations may or may not have `funds`). Thus primary paths for a schema denote all *vertical* data relationships that can exist in any instance conforming to that schema. To consider data relationships that span horizontally we need to consider the various ways in which primary paths can be joined. There are many such ways of joining, including taking the Cartesian product of the records. However, we will consider only joins that are faithful to the semantics conveyed by (nested) referential integrity constraints. In the absence of such semantics, we ask the user how to relate attributes, rather than guessing.

4.2 Constraints

Pairs of tableaux can be used to represent referential constraints between attributes.

Example 4.4 *For the schemas of Figure 1(b), the two source foreign keys can be represented as follows.*

(r_1) $\forall (g \in \texttt{expenseDB.grants})\ \exists (c \in \texttt{expenseDB.companies})\ c.\texttt{company.cid} = g.\texttt{grant.grantee}$

(r_2) $\forall (g \in \texttt{expenseDB.grants})\ \exists (p \in \texttt{expenseDB.projects})\ p.\texttt{project.name} = g.\texttt{grant.proj}$

[6] Maybe through some intermediate record types.

Notice that each constraint is of the form $\forall P_1 \exists P_2 B$ *where* P_1 *and* P_2 *are primary paths and* B *is an equality condition relating these paths.*[7] *The tableau in Example 4.1 represents a valid semantic relationship* because *of the constraint* r_1*. For the target schema, we can use a similar notation to represent nested referential integrity constraints.*

$$(r_3)\quad \forall(c \in \texttt{statDB})(o \in c.\texttt{cityStat.orgs})(f \in o.\texttt{org.fundings})$$
$$\exists(f' \in c.\texttt{cityStat.financials})\ f'.\texttt{financial.aid} = f.\texttt{fund.aid}$$

∎

For the nested constraint, we used a primary path, then a second path that is relative to the first path (through the variable c in the example). The second path does not start at the root, but rather from the `cityStat` record of the first path. This allows us to express the fact that each `fund` within an organization and within a `cityStat` refers (through `aid`) to some `financial` tuple within the set `financials` of the *same* `cityStat` record. The ability to express such nested dependencies is central to being able to generate nested logical relationships. Given two primary paths P_1 and P_2, where P_2 may be specified relative to a variable in P_1, along with an equality condition B relating the two paths, a *nested referential integrity constraint* (NRI) is then an expression of the form $\forall P_1 \exists P_2 B$. To be meaningful, the last variable of each path must be used in the equality B. Intuitively, we do not need to write a very long path if we refer only to a component at the beginning of it.

4.3 Logical Relations

We now consider how to generate tableaux that are consistent with the logical design of a schema. The chase is a classical relational method [BV84] that can be used to compute all the joins that can be applied to a given relation and that are based on a set of constraints.[8] In the following, we use an extension of the relational chase in order to enumerate joins in nested schemas, and consequently to enumerate semantic relationships in such schemas.

Definition 4.5 (Chase Step) *Given an NRI:*

$$(d)\quad \forall(x_1 \in e_1)\ldots(x_n \in e_n)\ \exists(z_1 \in e_{n+1})\ldots(z_m \in e_{n+m})\ B$$

the following rewrite[9] *is a chase step on tableaux:*

$$T \ = \ \langle\ \ldots, x_1 \in e_1, \ldots, x_n \in e_n, \ldots;\ B_1\ \rangle$$
$$\xrightarrow{(d)}\quad T' \ = \ \langle\ \ldots, x_1 \in e_1, \ldots, x_n \in e_n, \ldots, z_1 \in e_{n+1}, \ldots, z_m \in e_{n+m};\ B_1\ \wedge\ B\ \rangle$$

However the chase step is not applied *if the following conditions are both satisfied: (i)* T *contains* $z_1 \in e_1, \ldots, z_m \in e_{n+m}$*, and (ii)* B_1 *logically implies* B*.* ∎

Thus, whenever the left-hand path of an NRI matches a sub-part of a tableau T, the chase adds to T (if it is not there already) the right-hand path together with the condition of the NRI, yielding T'. In general, the chase could be defined to work with (nested) dependencies that are more general than (nested) referential integrity constraints. In that case it is well known that the complexity of checking whether a chase step applies[BV84, PT99] is exponential in the size of the dependency. In contrast, the complexity of a chase step with an NRI d (as defined above) is *polynomial* in the size of T and d. The reason is that NRIs are a *special* case of nested dependencies in which matching involves paths and is therefore polynomial.

In general, more than one chase step can be applied to a tableau. Given a tableau T and a set Σ of NRIs, we denote by $\texttt{Chase}_\Sigma(T)$ the final tableau T' produced by a sequence of chase steps starting at T and using NRIs from

[7] We have omitted the brackets around the two tableaux in these constraints to improve readability.
[8] Even though, originally, the chase was not introduced for this reason.
[9] Here, variables $x_1, \ldots, x_n$ are the same in d and T. In general we may need to search for renamings of variables from d to T.

Σ. By final tableau, we mean that no chase steps are applicable to T'. Of particular interest to us is chasing of primary paths.

Example 4.6 *Recall the schemas of Figure 1(b) and the NRIs of Example 4.4. The primary path*

$$S_2 = \langle\, g \in \texttt{expenseDB.grants};\, \rangle$$

chases, in a first step with (r_1), to the tableau

$$S_2' = \langle\, g \in \texttt{expenseDB.grants}, c \in \texttt{expenseDB.companies};\ c.\texttt{company.cid} = g.\texttt{grant.grantee}\,\rangle$$

In a second chase step with (r_2), S_2' is rewritten to the following tableau, which cannot be chased anymore:

$$\begin{aligned}\texttt{Chase}_{\{r_1,r_2\}}(S_2) \ = \ \langle\, & g \in \texttt{expenseDB.grants}, c \in \texttt{expenseDB.companies}, p \in \texttt{expenseDB.projects};\\ & c.\texttt{company.cid} = g.\texttt{grant.grantee} \ \wedge\ g.\texttt{grant.proj} = p.\texttt{project.name}\,\rangle\end{aligned}$$

Chasing S_2 with the NRIs of the schema brings together all the components of the schema containing tuples that are logically related with tuples in expenseDB.grants *(i.e.,* company *and* project *tuples). Moreover, the chase computes the join conditions that exist between such tuples. On the other hand, the primary path:*

$$S_1 = \langle\, c \in \texttt{expenseDB.companies};\, \rangle$$

cannot be chased with either of the two NRIs in the schema. Thus, S_1 tuples can exist without any other tuples; hence they make up a logical relationship by themselves. Finally, here is an example of a nested chase:

$$\begin{aligned} & T_3 = \langle\, c \in \texttt{statDB}, o \in c.\texttt{cityStat.orgs}, f \in o.\texttt{org.fundings};\, \rangle\\ \xrightarrow{r_3}\ & \texttt{Chase}_{\{(r_3)\}}(T_3) = \langle\, c \in \texttt{statDB}, o \in c.\texttt{cityStat.orgs}, f \in o.\texttt{org.fundings},\\ & \qquad f' \in c.\texttt{cityStat.financials};\ f'.\texttt{financial.aid} = f.\texttt{fund.aid}\,\rangle\end{aligned}$$ ∎

Thus, chasing primary paths can be used to compute logical relationships that exist in the schema. A *logical path* can therefore be formally defined as the result of replacing a primary path with the result of its chase. However, since in the schema mapping context we are interested in atomic type attributes (where the data is), we need to define one more operator on tableaux, that of unnesting. Given a tableaux T, we define $\texttt{Unnest}_T$ to be a *query* that, when given an instance I, evaluates T on I and returns only (but all) the *atomic type* attributes that occur in T (at any depth). *Logical relations* are then defined by applying $\texttt{Unnest}_T$ of logical paths.

Definition 4.7 ($\texttt{Unnest}_T$) *Let $T = \langle x_1 \in e_1, \ldots, x_n \in e_n;\ B\rangle$ be a tableau, and let $a_1, \ldots, a_k$ be all the atomic type expressions that use variables in $\langle x_1, \ldots, x_n\rangle$. If some of these expressions are equal, according to B, then take only one representative α among all that are equal. Let $\alpha_1, \ldots, \alpha_l$ be the set of all such representatives. The total unnesting of T is defined as the following query:*

$$\texttt{Unnest}_T \stackrel{\text{def}}{=} \{\, [\alpha_1, \ldots, \alpha_l] \ \mid\ x_1 \in e_1, \ldots, x_n \in e_n;\ B\,\}$$

where, for notational simplicity, we ignore the attribute labels that must be associated with each of the α_j. The query is written in a set-comprehension style of syntax. Its evaluation on an instance I produces, for each valuation[10] *$u : T \to I$, a tuple $[u(\alpha_1), \ldots, u(\alpha_l)]$.* ∎

Definition 4.8 (Logical Relations) *Given a schema $\mathcal{S}$ with NRIs Σ, its logical relations are defined as a set of queries:*

$$\{\, \texttt{Unnest}_{\texttt{Chase}_\Sigma(P)} \ \mid\ P \textit{ primary path of } \mathcal{S}\,\}$$ ∎

Thus, each logical relation is a query (a relational view) and therefore is a schema notion. Figure 5 shows all the logical relations for the two schemas of Figure 1(b). These are also example of unnesting.

[10]This simply means an instantiation of the variables in the tableau to actual elements in the instance. The structure and the join conditions in the tableau must be satisfied by this instantion.

$$
\begin{aligned}
A_1 &= \{\ [c.\texttt{company.cid}, c.\texttt{company.cname}, c.\texttt{company.city}] \mid c \in \texttt{expenseDB.companies};\ \} \\
A_2 &= \{\ [c.\texttt{company.cid}, c.\texttt{company.cname}, c.\texttt{company.city}, g.\texttt{grant.pi}, g.\texttt{grant.amount}, g.\texttt{grant.sponsor}, \\
&\qquad g.\texttt{grant.proj}, p.\texttt{project.year}] \\
&\quad \mid g \in \texttt{expenseDB.grants}, c \in \texttt{expenseDB.companies}, p \in \texttt{expenseDB.projects}; \\
&\qquad c.\texttt{company.cid} = g.\texttt{grant.grantee} \ \wedge\ p.\texttt{project.name} = g.\texttt{grant.proj}\ \} \\
A_3 &= \{\ [p.\texttt{project.name}, p.\texttt{project.year}] \mid p \in \texttt{expenseDB.projects};\ \} \\
\\
B_1 &= \{\ [c.\texttt{cityStat.city}] \mid c \in \texttt{statDB};\ \} \\
B_2 &= \{\ [c.\texttt{cityStat.city}, o.\texttt{org.cid}, o.\texttt{org.name}] \mid c \in \texttt{statDB}, o \in c.\texttt{cityStat.orgs};\ \} \\
B_3 &= \{\ [c.\texttt{cityStat.city}, o.\texttt{org.cid}, o.\texttt{org.name}, f.\texttt{fund.pi}, f.\texttt{fund.aid}, f'.\texttt{financial.amount}, f'.\texttt{financial.proj}, f'.\texttt{financial.year}] \\
&\quad \mid c \in \texttt{statDB}, o \in c.\texttt{cityStat.orgs}, f \in o.\texttt{org.fundings}, f' \in c.\texttt{cityStat.financials}; \\
&\qquad f'.\texttt{financial.aid} = f.\texttt{fund.aid}\ \} \\
B_4 &= \{\ [c.\texttt{cityStat.city}, f'.\texttt{financial.aid}, f'.\texttt{financial.amount}, f'.\texttt{financial.proj}, f'.\texttt{financial.year}] \\
&\quad \mid c \in \texttt{statDB}, f' \in c.\texttt{cityStat.financials};\ \}
\end{aligned}
$$

Figure 5: All the logical relations for: relational schema `expenseDB` (top), nested schema `statDB` (bottom).

4.4 Mapping Algorithm

As seen in the previous section, logical relations are not necessarily disjoint. They overlap: for example, A_1 and A_2 of Figure 5 can both produce `company` information; however, A_2 can produce in addition `grant` and `project` information. Thus, a correspondence can be relevant to several logical relations (in both source and target). Rather than looking at each individual correspondence, the mapping algorithm of this section (Algorithm 4.15) looks at each pair of source logical relation and target logical relation. For each such pair, it then computes a source-to-target dependency that will express how the source logical relation is to be mapped into the target logical relation. The computation of the dependency is driven by *all* the correspondences that are relevant for the given source logical relation and target logical relation. We illustrate the algorithm with an example.

Example 4.9 *Our algorithm looks at all pairs of of source and target logical relations. For each pair it performs the steps we describe here. Consider the pair $\langle A_1, B_2\rangle$ of our running example (with A_1 and B_2 defined in Figure 5) and the correspondences v_1 and v_2 of Figure 1(b) (let us ignore v_3 in this example). The correspondence v_1 maps the attribute* `cname` *of* `company` *in the source schema to attribute* `name` *of* `org` *in the target. The two attributes are included in A_1 and, respectively, B_2. We say that the pair $\langle A_1, B_2\rangle$ covers the correspondence v_1. (Coverage is defined formally later in the section.) We can then combine $\langle A_1, B_2\rangle$ with v_1 to generate the following dependency:*

$$
d^{\langle A_1,B_2\rangle} \quad \forall (c \in \texttt{expenseDB.companies})\ \exists (c' \in \texttt{statDB})(o \in c'.\texttt{cityStat.orgs})\ c.\texttt{company.cname} = o.\texttt{org.name}
$$

The universally quantified part of the dependency is the tableau of A_1, while the existential part is the tableau of B_2. The equality is the one asserting that `cname` *in the source should be equal to* `name` *in the target.*

A slightly more complex case is for the pair A_2, B_3 of logical relations. In this case, both *v_1 and v_2 are covered. This is because A_2 includes both* `cname` *and* `pi` *of the source. Similarly, B_3 includes both* `name` *and* `pi` *of the target. The dependency that will be computed by our algorithm in this case is:*

$$
\begin{aligned}
d^{\langle A_2,B_3\rangle} \quad & \forall (g \in \texttt{expenseDB.grants})(c \in \texttt{expenseDB.companies})(p \in \texttt{expenseDB.projects}) \\
& \quad [\ g.\texttt{grant.cid} = c.\texttt{company.cid}\ \wedge g.\texttt{grant.proj} = p.\texttt{project.name}\ \Rightarrow \\
& \qquad \exists (c' \in \texttt{statDB})(o \in c'.\texttt{cityStat.orgs})(f \in o.\texttt{org.fundings})(f' \in c'.\texttt{cityStat.financials}) \\
& \qquad\quad f.\texttt{fund.pi} = g.\texttt{grant.pi}\ \wedge\ o.\texttt{org.name} = c.\texttt{company.cname}\ \wedge \\
& \qquad\quad f'.\texttt{financial.aid} = f.\texttt{fund.aid}\ ]
\end{aligned}
$$

The mapping algorithm ends with a set consisting of the two dependencies above. Each of them is one logical mapping. Notice that v_2 is not covered by the first pair of logical relations. ∎

Before continuing any further, some important explanations regarding the algorithm are in order:

- The generated dependencies are more general assertions than NRIs. They have the same form $\forall T_1 \exists T_2 B$ with the main difference that T_1 and T_2 are general tableaux rather than paths. Therefore source-to-target dependencies belong to a class[11] that we call *nested dependencies*.
- The two dependencies above describe transformations that are relevant for *different* logical relations. As a result, both could be needed, depending on what an actual user intends. Our prototype system is able to help a user to navigate through all transformations that are generated by our algorithm and then to select a subset.
- In the second dependency, $p \in$ `expenseDB.projects` is not needed (is redundant, together with the join condition g.`grant.proj` $= p$.`project.name`). However, if more correspondences are inserted, it *may* become needed. In general, determining whether conditions in a source-to-target dependency are unnecessary depends on what correspondences are present. Removing unnecessary conditions is very similar to query optimization. We intend to address this issue in future work.
- Related to the previous point, $f' \in c'$.`cityStat.financials` is in the target part of the second dependency, even though no source attribute is mapped into the `financial` attribute. Nevertheless, it is not redundant. It describes the fact that for any `aid` value that is generated in the `fund` in the target database, there must be a `financial` tuple that has the same `aid` value, since `aid` in `fund` is a foreign key referencing the `financial`. Hence, the reason that such expressions may appear in the generated dependencies is to guarantee that constraints of the target schema are not violated.

Before giving the details of the algorithm, we give a formal definition of *correspondence* and define what it means for a logical relation (or a pair of logical relations) to *cover* a correspondence. A correspondence can be described as a simple case of dependency that specifies how a value in the target is generated by a value in the source.

Definition 4.10 (Correspondence) *A correspondence is a triplet $v = \langle P, P', e = e' \rangle$, where P is a primary path of the source schema, P' is a primary path of the target schema, and e and e' are atomic type expressions that use the last variable of path P and P' respectively.*

Example 4.11 *The correspondences v_1 and v_2 shown in Figure 1(b) can be described by:*

$\langle\ \langle c \in \texttt{expenseDB.companies};\rangle,\ \langle c' \in \texttt{statDB}, o \in c'.\texttt{cityStat.orgs};\rangle,\ c.\texttt{company.cname} = c'.\texttt{org.name}\rangle$

$\langle\ \langle g \in \texttt{expenseDB.grants};\rangle,\ \langle c' \in \texttt{statDB}, o \in c'.\texttt{cityStat.orgs}, f \in o.\texttt{org.fundings};\rangle,\ g.\texttt{grant.pi} = f.\texttt{fund.pi}\ \rangle$ ∎

The next definition gives a name to a pair of a source logical relation and a target logical relation.

Definition 4.12 (Skeleton) *A* skeleton *S is a pair $\langle A, B\rangle$ where A is an logical relation of the source schema and B is a logical relation of the target schema.* ∎

Coverage of a correspondence v by a logical relation is slightly more complicated than is suggested by Example 4.9. It is not enough to check whether the logical relation includes the attribute name involved in v. This is ambiguous, in general. We do not assume that attribute names are different across the schema. Moreover, the *same* attribute of a schema may be be included twice or more[12] in a logical relation! In order to precisely identify which attribute in the logical relation corresponds to the attribute involved in the correspondence we need to match the path defining v with the tableau defining the logical relation.

Definition 4.13 (Coverage) *Correspondence $v = \langle P, P', e = e'\rangle$ is covered by skeleton $\langle A_i, B_j\rangle$ with $\langle h, h'\rangle$ and result $h(e) = h'(e')$ if there exist two renaming functions: h from the variables of P into variables of A_i, and h' from the variables of P' into variables of B_j such that $h(P)$ is a subset of A_i and $h'(P')$ is a subset of B_j.* ∎

[11] The full definition is omitted due to lack of space.
[12] See the last example of this section.

Algorithm 4.15 (Semantic Translation)
Input: *source schema with NRIs* Σ_s,
target schema with NRIs Σ_t,
set of correspondences
Phase 1. *compute (by chasing with* Σ_s*, respectively,* Σ_t*) all the source logical relations* $\{A_1, \ldots, A_n\}$ *and all the target logical relations* $\{B_1, \ldots, B_m\}$;
Phase 2. for each *skeleton* $\langle A_i, B_j \rangle$:
find all correspondences $v_1, \ldots, v_k$ *that are covered by* $\langle A_i, B_j \rangle$ *(each with one or more way of coverage);*
if $(k = 0)$ or *(there exists a sub-skeleton* $\langle A_k, B_l \rangle$ *of* $\langle A_i, B_j \rangle$ *that covers the same correspondences)*
(pruning) then continue *with the next skeleton;*
for each *combination* δ *of coverages of* $v_1, \ldots, v_k$ *by* $\langle A_i, B_j \rangle$
(with $\langle h_1, h'_1 \rangle, \ldots, \langle h_k, h'_k \rangle$ *and results* $h_1(e_1) = h'_1(e'_1), \ldots, h_k(e_k) = h'_k(e'_k)$*, respectively) :*
let B'_j *be an extension of tableau* B_j *with the conjunction* $h_1(e_1) = h'_1(e'_1) \wedge \ldots \wedge h_k(e_k) = h'_k(e'_k)$
create a source-to-target dependency $d_\delta^{\langle A_i, B_j \rangle} = dep(A_i, B'_j)$
Output: *the set of all* $d_\delta^{\langle A_i, B_j \rangle}$

Figure 6: Semantic translation algorithm.

In general, there may be multiple coverages of v with $\langle A_i, B_j \rangle$. Our algorithm takes into account all such choices.

Example 4.14 *Consider again the mapping scenario of Figure 1(b), but where the source schema has an additional restriction that the sponsor is always a company. This means that* `sponsor` *is also a foreign key referencing* `company`*. This new foreign key constraint is described by the following NRI:*

$$(r_4) \quad \forall (g \in \texttt{expenseDB.grants})\, \exists (c \in \texttt{expenseDB.companies})\; c.\texttt{company.cid} = g.\texttt{grant.sponsor}$$

The logical relation A_2 *in Figure 5 is the result of chasing the primary path* S_2 *of Figure 4 with the constraints* r_1 *and* r_2*. Given the additional constraint* r_4*, logical relation* A_2 *becomes* $A'_2 = Chase_{\{r_4\}}(A_2)$*:*

$$\begin{aligned} A'_2 = \{ & [c.\texttt{company.cid}, c.\texttt{company.cname}, c.\texttt{company.city}, g.\texttt{grant.pi}, g.\texttt{grant.amount}, g.\texttt{grant.proj}, \\ & \quad c'.\texttt{company.cid}, c'.\texttt{company.cname}, c'.\texttt{company.city}, p.\texttt{project.year}] \\ & \mid\; g \in \texttt{expenseDB.grants}, c \in \texttt{expenseDB.companies}, p \in \texttt{expenseDB.projects}; \\ & \; c.\texttt{company.cid} = g.\texttt{grant.grantee} \;\wedge c'.\texttt{company.cid} = g.\texttt{grant.sponsor} \;\wedge\; p.\texttt{project.name} = g.\texttt{grant.proj} \; \} \end{aligned}$$

It is not hard to see that v_1 *and* v_2 *are both covered by* $\langle A'_2, B_2 \rangle$*. However,* v_1 *can be covered in multiple ways by* $\langle A'_2, B_2 \rangle$*. More specifically, there are two mapping functions* h*. One maps the variable* c *of* v_1 *to the variable* c *of* A'_2 *and the other maps* c *to* c' *of* A'_2*. Thus the attribute* `cname` *in* v_1 *can be matched in two ways with the* `cname` *attribute of the source schema. The two ways of matching reflect the fact that companies and grants can be joined in two ways, one through the* `grantee` *foreign key and the other through the* `sponsor`*. The two different renaming functions are two different interpretations of the correspondences. The first one generates a query that maps the companies that have some grants for some projects while the second maps the companies that are funding other companies. Both interpretations are meaningful.* ∎

Not all skeletons generate a dependency. For example, skeletons that do not cover any correspondence, such as $\langle A_1, B_1 \rangle$, do not generate a dependency Also, a skeleton does not generate a dependency (it is not considered a meaningful part of the mapping) if it does not cover at least one correspondence that no sub-skeleton of it covers. A sub-skeleton of $\langle A_i, B_j \rangle$ is defined as a skeleton $\langle A_k, B_l \rangle$ such that A_k is a sub-tableau of A_i and B_l is a sub-tableau of B_j. For example, suppose that v_1 is the only correspondence. Then v_1 is covered by the skeleton $\langle A_2, B_3 \rangle$. However, there exists a sub-skeleton, $\langle A_1, B_2 \rangle$, that already covers v_1. Algorithm 4.15 does not generate any dependency for A_2 and B_3 in this case.

From inclusion dependencies to nested queries. The source-to-target dependencies generated by Algorithm 4.15 are inclusion dependencies (in the relational sense [CFP84]) between source logical relations and target logical relations (which are relational views on the source schema and, respectively, target schema). To illustrate, source logical relation A_2 and target logical relation B_3 of Figure 5 are both relational views with schemas[13]:

$$A_2(\texttt{cid}, \texttt{cname}, \texttt{city}, \texttt{pi}, \texttt{proj}, \texttt{amount}, \texttt{year}) \qquad B_3(\texttt{city}, \texttt{cid}, \texttt{name}, \texttt{pi}, \texttt{aid}, \texttt{amount}, \texttt{proj}, \texttt{year})$$

Then $d^{\langle A_2, B_3 \rangle}$ is the same as the inclusion dependency: $A_2[\texttt{cname}, \texttt{amount}] \subseteq B_3[\texttt{name}, \texttt{amount}]$ meaning that the projection of A_2 on `cname` and `amount` must be contained in the projection of B_3 on `name` and `amount`. This inclusion dependency could then be used to produce a query that materializes B_3 with all `cname` and `amount` values of A_2, and fills the rest of the attributes with nulls. However, we do not want to materialize B_3 but rather the nested schema over which B_3 is only a flat view. Thus we need to nest data, appropriately. Moreover, null values are not what we want for the attribute `aid`: even though B_3 gathers together all the attributes of the logical relation, in the nested schema the attributes are partitioned in two different parts of the schema. In order to implement this partitioning without loss of information, `aid` will have to play the role of a placeholder to be created in both parts of the schema. These issues are addressed by the query generation algorithm of Section 6.

5 Semantic Translation: Correctness and Completeness

Termination of chase: acyclic schemas NRIs generalize the class of relational inclusion dependencies, and, similarly, our chase is a generalization of the relational chase. It is well known that, in general, the chase with inclusion dependencies may not terminate. A solution to this problem is to restrict the class of legal constraints so that the chase is guaranteed to terminate. Among several such classes, acyclic inclusion dependencies [CK86] are a particularly useful one. They capture naturally many of the integrity constraints of relational schemas.

Consequently, we generalized the notion of acyclicity to the case of nested schemas and constraints. The lack of space does not allow us to elaborate on this subject. However, we mention that the class of acyclic NRIs guarantees that any chase sequence of a tableau terminates and, moreover, it does so in polynomial time in the size of the tableau and the given NRIs. Termination of chase ensures that there are only finitely many logical paths (and logical relations) for a schema, and therefore the search space of all possible interpretations of the mapping is finite. Our working restriction in this paper is that schemas are acyclic. In Section 8 we discuss another form of cyclicity that may arise in XML schemas: recursive types (currently not allowed in our nested relational model).

Enforcement of target constraints An important consequence of our mapping algorithm is the fact that any implementation of the generated source-to-target (s-t) dependencies automatically satisfies all the target constraints. Here implementation means: given a source instance, find a target instance such that the s-t dependencies are satisfied. There are many such implementations and section 6 gives a canonical implementation (with good properties!) of s-t dependencies as queries. But first we state the following theorem (for any implementation).

Theorem 5.1 *Let Σ_{st} be a set of source-to-target dependencies generated by Algorithm 4.15. Moreover, let Σ_t be the set of NRIs on the target schema, and assume $\mathtt{I}_s$ is an instance over the source schema and I_t is an instance over the target schema. Then I_t satisfies Σ_t whenever I_s and I_t together satisfy Σ_{st}.*

The above result allows us to focus on the implementation of the source-to-target dependencies alone, i.e. we do not have to worry about satisfaction of target constraints. They have already been taken into consideration when producing the target logical relations (via chase with Σ_t). The above theorem does not hold for Algorithm 4.15

[13] We give here convenient attribute names.

if we try to extend the class of legal target constraints from NRIs to arbitrary nested dependenciess (e.g. join dependencies). In other words it is essential that NRIs are based on linear paths for the above theorem to hold. Still, NRIs are a very useful fragment of nested dependencies for representing integrity constraints in relational and nested schemas. The next section shows how we implement source-to-target dependencies as *queries*.

Chase computes all associations that exist with respect to the universal relation. It is obvious that the chase used to compute logical relations finds "natural" associations between attributes. What is not so obvious is whether *all* the natural associations are discovered by our algorithm. To answer this question, we need first to find a declarative definition of associations that is independent of the particular way (chase) that is used to compute them. Then the answer is affirmative (i.e. we find indeed all associations) if we can show that all the declaratively defined associations are computed by our algorithm. To find the declarative definition for association we need to look at a rather forgotten piece of database theory: the universal relation model [MUV84]. This model aimed at achieving logical access path independence in relational databases. That is, a user can view the underlying database as one relation comprising *all* the attributes in the universe of that database. The user can query then any subset `X` of attributes without having to specify the join paths that must be used in the underlying database to associate the attributes of `X`. The system is supposed to find automatically the "natural" associations among the attributes of `X`. This is very similar to the way in which our mapping compiler is used.

In our system we do not use the universal model approach. However, we find it a useful modeling tool that allows us to define, in a declarative way, independent of any computation procedure, the associations that exist amongst attributes of a schema. We can then prove that the chase used in the semantic translation is able to compute all the connections that are predicted by the universal relation model. We now focus on the case when the schema is relational and moreover restricted so that the usual universal relation assumptions hold. We believe that the universal relation analogy extends to the nested case as well.

Associations. Assume a schema consisting of several relations. All the attributes in the schema are assumed to have unique names, with one exception: attributes involved in foreign key constraints, which must have the same name. Thus, all the information in the database is in the attributes: each reflects a unique feature of the data, and hence has a unique name. Let `U` be the universe of all such unique attributes in the schema, and let I be an instance over the set of database relations. We assume that I satisfies all the foreign key constraints (NRIs) *and* the corresponding key constraints. Moreover, let Δ be the set of all functional dependencies (fd's) that are obtained by "lifting" each key constraint from an fd on the universe of its respective relation, to an fd on the universe `U`.

Following [MUV84], a *weak universal relation (or weak UR)* associated to I and Δ is defined as any relation u over `U` with the following properties: 1) $\Pi_{\mathtt{R}}(u)$ is a superset of the current relation for `R` in I, and 2) u satisfies Δ. Thus the weak universal relation can be viewed as some "idealized" completion of the database relations in I, satisfying Δ. It provides a view of the underlying database that is independent of any particular normalization scheme. The *association*[14] *on a subset of attributes* `X` *of* `U`, with respect to Δ, denoted by $[\mathtt{X}]^{\Delta}$, is defined as the set of tuples t such that for *every* weak UR u, there is tuple t' in u that agrees with t on `X`.

[MUV84] show that $[\mathtt{X}]^{\Delta}$, for a given I, can be computed by materializing first a *representative* weak UR, called RI_{Δ}, as follows: each tuple of each relation in I is inserted in RI_{Δ}, after it has been padded with new "null" symbols. Then the resulting relation is chased to enforce the satisfaction of the dependencies in Δ. The result of the chase is RI_{Δ}. The association $[\mathtt{X}]^{\Delta}$ is then equal to the projection of RI_{Δ} on `X` followed by the elimination of any tuples that contain "null" symbols. In practice, we want $[\mathtt{X}]^{\Delta}$ to be computed by efficient queries without having to chase instances. However, the general decision problem of whether there exist such queries is not known

[14] This is called a *connection* in [MUV84].

to be solvable. Nonetheless, our schemas are restricted: they are normalized according to the given foreign key/key constraints. Moreover, Δ contains only the fd's that arise from the key constraints. Thus, not surprisingly, we can show that $[\mathtt{X}]^{\Delta}$ is computable from our logical relations:

Theorem 5.2 (Completeness) *Assume* $\mathtt{X} \subseteq \mathtt{U}$ *and let* $\mathtt{A}_1, \ldots, A_n$ *be the logical relations for which the set of attributes includes* X*. Then* $[\mathtt{X}]^{\Delta} = \Pi_{\mathtt{X}}(A_1) \cup \ldots \cup \Pi_{\mathtt{X}}(A_n)$*.*

The theorem shows that each association is fully computable by a union of projections of logical relations. These relations are in turn computed by the chase. Hence, our semantic translation algorithm is *complete* as a method for computing associations. In the actual algorithm, we precompute all the logical relations, independently of any correspondences. Then, as suggested by the theorem, we discover maximal sets of associated attributes by checking coverage by some logical relation. (We say that logical relation A *covers* X if the attributes of A include X.)

Our semantic translation algorithm is able to abstract away the specific schema details such as normalization and nesting. The above theorem is a precise statement of this. We could change the logical design (in the source and/or target), but we would get the same mapping, as long as the attributes have the same semantics associated to it, and the attribute-to-attribute correspondences remain the same.

6 Data Translation: Query Generation

The algorithm that we illustrate in this section translates into a query each individual nested dependency generated in the semantic translation phase. To obtain the final mapping, we take the union of resulting queries.

The main idea of the query generation algorithm is the following. Let d be an inclusion dependency from a source logical relation $A = \mathtt{Unnest}_{T_1}$ to a target logical relation $B = \mathtt{Unnest}_{T_2}$. The individual query that we generate for d will work, conceptually, in two steps. First, it materializes the flat relational view $\mathtt{Unnest}_{T_1}$ and projects on the attributes that are used by correspondences. After some additional renaming, which may be necessary, from attributes of A to attributes of B, the result is essentially a projection[15] of $\mathtt{Unnest}_{T_2}$. In a second step, it nests the result according to the structure of the target (i.e. the types involved in T_2) and creates new values for the undetermined attributes. The first part is rather standard: it involves unnesting, join, projection and renaming operations. The second part is non-trivial and is essentially an inverse of $\mathtt{Unnest}_{T_2}$ (as we prove in Section 6.2).

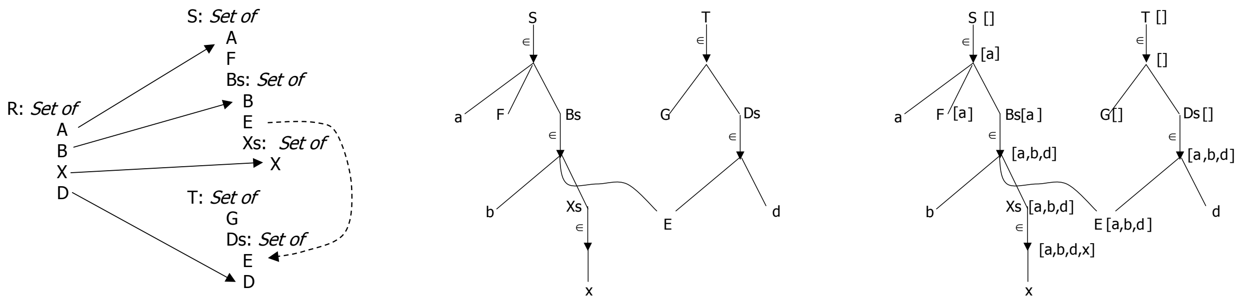
In Section 6.1, we present the algorithm as if the view $A = \mathtt{Unnest}_{T_1}$ has already been materialized. The focus is on the second step of the algorithm. In the full-fledged query generation algorithm, there will be one more step in which A is expanded with its definition $\mathtt{Unnest}_{T_1}$. Section 6.2 shows that the query generation algorithm is not arbitrary, but satisfies some fundamental properties.

6.1 Query Generation Algorithm

We illustrate the algorithm in a rather informal way, based on an abstract and simplified example. Consider the mapping scenario of Figure 7(a) and the following dependency d describing the mapping:

$$(d) \quad \forall (r \in \mathtt{R})\ \exists (s \in \mathtt{S})(p \in s.\mathtt{Bs})(x \in \mathtt{Xs})(t \in \mathtt{T})(u \in t.\mathtt{Ds})$$
$$s.\mathtt{A} = r.\mathtt{A}\ \wedge\ p.\mathtt{B} = r.\mathtt{B}\ \wedge\ x.\mathtt{X} = r.\mathtt{X} \qquad \text{(attributes mapped by value correspondences)}$$
$$\wedge\ p.\mathtt{E} = u.\mathtt{E} \qquad \text{(join condition in the target association)}$$

[15] Since not all attributes in the target logical relation may be determined by correspondences.



(a) (b) (c)

Figure 7: (a): mapping used to illustrate query generation. (b): query graph. (c): annotated query graph.

To simplify the presentation even further, we will make a change of notation and present d, below, in an equivalent *rule notation*. This notation is less verbose than the one above, but it works (nicely) only when set and record types alternate (which is not the case for many translations of real XML schemas, including our running example).

(r_d) $\quad \texttt{R}(a,b,x,d) \rightarrow \texttt{S}(a,\texttt{F},\texttt{Bs}),\ \texttt{Bs}(b,\texttt{E},\texttt{Xs}),\ \texttt{Xs}(x),\ \texttt{T}(\texttt{G},\texttt{Ds}),\ \texttt{Ds}(\texttt{E},d)$

The rule asserts the existence of several facts in the target (right-hand side or *head* of the rule in datalog terminology) for any fact of the form $\texttt{R}(a,b,x,d)$ that exists in the source (left-hand side or *body* of the rule in datalog terminology). In the new notation, variables range over individuals rather than over tuples. As opposed to datalog, there are multiple facts in the head and these facts are correlated to each other. For example, in the second fact in the head, the set variable Bs appears earlier as a variable in the fact $\texttt{S}(a,\texttt{F},\texttt{Bs})$. This means a nesting of the second fact inside the first fact. Also, the second and the fifth facts are related: they must have the same E component. Moreover, not all the variables occuring in the head occur in the body. Such variables denote values that must be created. We call them existentially quantified variables.

Rule evaluation and creation of oids: There are many possible ways of evaluating (r_d). A naive evaluation would be to create the five facts asserted in the head for each possible instantiation of the body, while at the same time assigning new values (oids) for the existentially quantified variables. However, this approach would not produce the desired result. In particular, too many oids would be created. For example, for all tuples of R that happen to have the same value for a, a different set oid for Bs is created. This is not what we expect from the mapping of Figure 7(a). What is needed is rather a *grouping* of all facts $\texttt{Bs}(b,\texttt{E},\texttt{Xs})$ within the same fact $\texttt{S}(a,\texttt{F},\texttt{Bs})$, for a given value of a. Similarly, we do not want to create too many oids for F (or for E or G for that matter). In fact, one oid for F, for each value of a, would suffice. On the other hand, there is also a danger of creating two few oids. For example, we could create an oid E (in both S and T) for each distinct combination of a and b (and ignoring d). However this will result in loss of information: if we take the resulting instance and unnest back (according to the definition of the target logical relation), we could obtain more tuples than originally mapped from R. (See Section 6.2 for a more detailed discussion on lossless mappings.) To avoid loss of information during evaluation, E must in fact be generated as a distinct oid for each combination of a, b, d.

Skolemization and Query Generation Algorithm. Our algorithm is based on a scheme that associates with each existentially quantified variable a *subset* of the variables in the body (a subset of $\{a,b,x,d\}$ in the example). This subset will control the creation of fresh values: each existentially quantified variable is a function of that subset (rather than the entire set of source variables). Based on such functions (Skolem functions) a rule is then immediately translatable as a query. We divide the algorithm into three main steps.

Step 1: create annotated query graph. First, we construct a *query graph* associated with the rule, as follows. The query graph is a canonical instance, over the target schema, containing all the facts that are asserted in the right-hand side of the rule (target facts). In this instance, each source variable is considered to be an atomic value, while the capital letter variables are oids. Figure 7(b) shows the query graph for our rule, (r_d).

Next, we compute a list of source variables for each node in the query graph that is not a source variable. This is done by *propagating* each source variable to the rest of the nodes, in any of the following three ways:

1) propagate up the record projection edges: for our example, a is propagated up to its parent record. This will ensure that the record depends on a, i.e. there will be (at least) one distinct record in the target instance for each distinct value of a in the source.

2) propagate down the record projection edges: a is propagated from its parent down to the children of the parent (excluding children that are source variables). Thus, a is added to F as well as to Bs. This will ensure that there will be (at least) one oid created for F for each distinct a (similarly for the set oid Bs).

3) propagate down the $\in$-edges: a is propagated from Bs to its $\in$-child (again excluding children that are source variables). Thus, for each such child there will be (at least) one oid created for each distinct value of a.

The propagation rules are applied recursively until no rules can be applied. In our example, the next propagation (using Rule 2) propagates a, along record projection edges, to nodes Xs and E. Then, a is propagated from E to (both) of E's record parents. At this point, no additional rules apply and propagation stops. Notice that propagation is not performed up along an $\in$-edge from child to parent. For example, d is not propogated to the node labeled Ds. The propagation process ends in a unique configuration in which each node in the the query graph has a list of the source variables. (See Figure 7(c), for our example.)

Finally, we associate to each (atomic or set) existentially quantified variable in the query graph a unique Skolem function name. For simplicity, in our example, we use the same name for both the variable and the Skolem function (e.g. we associate to E the Skolem function name E). In general, when we have multiple source-to-target dependencies (rules), we make sure that *different* Skolem function names are generated for different dependencies. Then, the annotation of an existentially quantified variable is a *Skolem term* obtained by concatenating the Skolem function name with the list of source variables. For instance, the annotation of E is $\texttt{E}[a, b, d]$.

Step 2: translate (r_d) into a set $\texttt{Nest}_d$ of rules, with no existentially quantified variables. The annotated query graph already encodes the evaluation semantics that we want for (r_d). This evaluation proceeds in two phases (similar to ILOG [HY90]). First, we instantiate the query graph for each tuple of R and generate a collection of facts, as follows. Each source variable occuring in the query graph (e.g. a) is instantiated with the actual value in the source tuple. Each existentially quantified variable in the query graph (e.g. E) is instantiated to its Skolem term in which source variables are replaced by actual values (call such Skolem term a *ground* Skolem term). Ground Skolem terms are not legal values in the data model. Thus, in the second phase, we replace each ground Skolem term, consistently in the entire collection of generated facts, by exactly one oid from the appropriate domain. Syntactically, we describe the two-phase evaluation[16] by the following Skolemized rule:

$$\texttt{R}(a,b,x,d) \;\rightarrow\; \texttt{S}[\,](a,\texttt{F}[a],\texttt{Bs}[a]),\; \texttt{Bs}[a](b,\texttt{E}[a,b,d],\texttt{Xs}[a,b,d]),\; \texttt{Xs}[a,b,d](x),\; \texttt{T}[\,](\texttt{G}[\,],\texttt{Ds}[\,]),\; \texttt{Ds}[\,](\texttt{E}[a,b,d],d)$$

The target facts in the head of the Skolemized rule are no longer correlated, as they were in the original (r_d). Therefore, the rule above is equivalent to a set of five rules. We show below the equivalent set of rules, $\texttt{Nest}_d$, after which we give several remarks regarding $\texttt{Nest}_d$.

[16]In the actual run-time, evaluation is done in one phase by using lookup tables for mapping of Skolem terms to oids.

$$
\begin{array}{l}
\texttt{S} \;=\; \underline{\text{for}}\; r_0 \;\underline{\text{in}}\; \texttt{R} \\
\quad \underline{\text{return}}\; [\, \texttt{A} \;=\; r_0.\texttt{A},\; \texttt{F} \;=\; \texttt{F}[r_0.\texttt{A}], \\
\qquad\quad \texttt{Bs} \;=\; \underline{\text{for}}\; r_1 \;\underline{\text{in}}\; \texttt{R} \\
\qquad\qquad\quad \underline{\text{where}}\; r_0.\texttt{A} = r_1.\texttt{A} \\
\qquad\qquad\quad \underline{\text{return}}\; [\, \texttt{B} \;=\; r_1.\texttt{B},\; \texttt{E} \;=\; \texttt{E}[r_1.\texttt{A}, r_1.\texttt{B}, r_1.\texttt{D}], \\
\qquad\qquad\qquad\quad \texttt{Xs} \;=\; \underline{\text{for}}\; r_2 \;\underline{\text{in}}\; \texttt{R} \\
\qquad\qquad\qquad\qquad\quad \underline{\text{where}}\; r_2.\texttt{A} = r_1.\texttt{A} \;\underline{\text{and}}\; r_2.\texttt{B} = r_1.\texttt{B} \;\underline{\text{and}}\; r_2.\texttt{D} = r_1.\texttt{D} \\
\qquad\qquad\qquad\qquad\quad \underline{\text{return}}\; [\, \texttt{X} = r_2.\texttt{X} \,]\,]\,] \\
\texttt{T} \;=\; \{\, [\, \texttt{G} \;=\; \texttt{G}[\,],\; \texttt{Ds} \;=\; \underline{\text{for}}\; r_0 \;\underline{\text{in}}\; \texttt{R}\; \underline{\text{return}}\; [\, \texttt{E} \;=\; \texttt{E}[r_0.\texttt{A}, r_0.\texttt{B}, r_0.\texttt{D}],\; \texttt{D} \;=\; r_0.\texttt{D} \,]\,]\,\}
\end{array}
$$

Figure 8: $\texttt{Nest}_d$, in an abstract query language. (Our XQuery translation follows closely this representation.)

$$
\begin{array}{lll}
(\texttt{Nest}_d) & \texttt{R}(a,b,x,d) \;\rightarrow\; \texttt{S}[\,](a, \texttt{F}[a], \texttt{Bs}[a]) & \texttt{R}(a,b,x,d) \;\rightarrow\; \texttt{T}[\,](\texttt{G}[\,], \texttt{Ds}[\,]) \\
 & \texttt{R}(a,b,x,d) \;\rightarrow\; \texttt{Bs}[a](b, \texttt{E}[a,b,d], \texttt{Xs}[a,b,d]) & \texttt{R}(a,b,x,d) \;\rightarrow\; \texttt{Ds}[\,](\texttt{E}[a,b,d], d) \\
 & \texttt{R}(a,b,x,d) \;\rightarrow\; \texttt{Xs}[a,b,d](x) &
\end{array}
$$

• The first rule computes tuples in the top-level set S by iterating over all tuples of R. For every value of a a set oid $\texttt{Bs}[a]$ is created (as well as an atomic oid $\texttt{F}[a]$). The second rule iterates (again!) over R and for each fixed a, retrieves the corresponding b and d values to compute tuples in the nested set $\texttt{Bs}[a]$. By using the same Skolem term $\texttt{Bs}[a]$ in both rules we achieve a *grouping* condition (similar role for $\texttt{Xs}[a,b,d]$).

• The Skolem terms for G and Ds have no arguments, and therefore they are constants (they do not depend on any of the source variables). Thus, there is one single tuple that is generated inside the set T. This is a consequence of the fact that no correspondence maps into the G attribute in the scenario of Figure 7.

• The Skolem terms for the target roots S and T have no arguments. Thus, only one oid is created for each of them. This should be the case since a root must satisfy a cardinality constraint of 1. Still, it is possible that such uniqueness constraint is violated if there is a correspondence from a repeated source element (cardinality more than 1) into a non-repeated target element. Such a mapping is rare, since the only non-repeated target elements are the root or attribute projections from the root. We currently do not handle this case in our tool. To produce a correct mapping for such cases, the user could simply be required to provide an aggregate function.

• Not only set type Skolem terms are shared among rules of $\texttt{Nest}_d$. For example, $\texttt{E}[a,b,d]$ is an atomic type Skolem term that is shared. This means that the same oid will be generated for the same combination of a, b, d for the two E attributes of the schema. Thus, the referential integrity constraint is satisfied and moreover the association between information mapped into T and information mapped into S is not lost.

Step 3: reformulate $\texttt{Nest}_d$ as a query, by inlining set type Skolem terms with subqueries. It is straightforward to translate $\texttt{Nest}_d$ into a more explicit query-like language that uses for_where_return_ query blocks. This translation can be performed by a technique of *inlining*. Rather than giving the tedious but straightforward details of inlining, we show in Figure 8 the result of such inlining for our example. The result consists of several top-level query blocks (one for each root). The return clause of each query block may contain other query blocks (subqueries) for the corresponding nested sets. Subqueries are correlated with the parent queries via grouping conditions. In the example, the top-level queries are not independent: they share the Skolem function names E. This is reminiscent of XML-QL [DFF+99]. For languages such as XQuery [CCDF+01] we must *simulate* the behavior of Skolem terms by using lookup tables.

The resulting query is an abstract query, independent of the external query language that is available for performing the data transformation from source to target. Our system provides then for specific query wrappers that translate abstract queries into the desired query language (such as SQL for relational schemas, XQuery/XSLT for XML schemas). Section 6.3 addresses some of the specific details required for translation into XQuery.

6.2 Properties of the Query Generation Algorithm

Our query generation algorithm has several important properties. First, the resulting target instance is in Partition Normal Form (or PNF) [AB86, ÖY87, RK87]. This means that in any set, at any nesting level of the target instance, the atomic type attributes functionally determine the set type attributes (if any). For example, in the set S of the mapping example of Figure 7(a) there cannot be two tuples $t_1 = [\mathtt{A} = a, \mathtt{F} = f, \mathtt{Bs} = O_1]$ and $t_2 = [\mathtt{A} = a, \mathtt{F} = f, \mathtt{Bs} = O_2]$ where O_1 and O_2 are *distinct* set oids. In effect, this property guarantees that any nested sets are *merged* whenever the atomic type attributes coincide. This property is a consequence of the Skolemization approach described in the previous section. Partition Normal Form is recognized as an important property of nested instances since it ensures that the redundancy that can occur due to nesting has been minimized.

A second property of the algorithm is that all the generated atomic oids are keys. In other words in any (nested) set, there can be no two tuples with the same atomic oid. For example, the attribute E in the set Ds of the mapping example of Figure 7(a) is a key. In fact, this should better be the case since there exists a foreign key/key constraint from E of Bs to E of Ds.

The rest of this section shows that the query generated by our algorithm populates the target with exactly the source data that was "intended" to be mapped by the logical mapping. We formalize this notion as *information preservation* (with respect to a given source-to-target dependency).

Information preservation. As in the previous section we assume that the source is a flat relation R (it represents a source logical relation, i.e. $\mathtt{Unnest}_{T_1}$ for some T_1). Let $\mathtt{Unnest}_{T_2}$ be a target logical relation, and d be an inclusion dependency from a subset of attributes X of R to a subset of attributes Y of $\mathtt{Unnest}_{T_2}$: $\quad \mathtt{R}[\mathtt{X}] \subseteq \mathtt{Unnest}_{T_2}[\mathtt{Y}]$

Based on d, the query generation algorithm computes $\mathtt{Nest}_d$, which transforms the projection of R on X to a nested target instance[17]. Then $\mathtt{Nest}_d$ is *information preserving* if the following property is satisfied:

$$(*) \qquad \rho_{\mathtt{Y}\mapsto\mathtt{X}}\, \Pi_{\mathtt{Y}}\, \mathtt{Unnest}_{T_2}\, (\mathtt{Nest}_d(\mathtt{R})) \;=\; \Pi_{\mathtt{X}}(\mathtt{R})$$

In the above, Π is the relational projection operator, while $\rho_{\mathtt{Y}\mapsto\mathtt{X}}$ renames all attributes Y to attributes X. Thus, information preservation says that if we apply $\mathtt{Nest}_d$ on an instance of R, then unnest back, by using $\mathtt{Unnest}_{T_2}$, and then only project on those target attributes that were mapped into, i.e. Y, we obtain the same tuples that were in $\Pi_{\mathtt{X}}(\mathtt{R})$.

It is easy to see that one inclusion is satisfied: all tuples of $\Pi_{\mathtt{X}}(\mathtt{R})$ will be part of the left-hand side of equation (*). Thus, all the source tuples (with only the needed attributes) that were intended to be mapped made their way into the target. Hence, the target instance satisfies the dependency d (with respect to the source instance). The other inclusion means that we did not introduce *extra* tuples in the target. As we will see shortly, $\mathtt{Nest}_d$ satisfies this inclusion too! This is a consequence of the careful way in which Skolem terms are generated.

Example 6.1 *Recall from Section 6.1 the set* $\mathtt{Nest}_d$ *of rules computed in Step 2 of the query generation algorithm. Consider an instance for* R *that contains just two facts:* $\mathtt{R}(a, b, x, d)$, $\mathtt{R}(a, b, x', d')$. *Then, by running* $\mathtt{Nest}_d$ *we produce the following target facts:*

$$\begin{array}{lllll} \mathtt{S}(a,\mathtt{F},\mathtt{Bs}), & \mathtt{Bs}(b,\mathtt{E},\mathtt{Xs}), & \mathtt{Xs}(x), & \mathtt{T}(\mathtt{G},\mathtt{Ds}), & \mathtt{Ds}(\mathtt{E},d) \\ & \mathtt{Bs}(b,\mathtt{E}',\mathtt{Xs}'), & \mathtt{Xs}'(x'), & & \mathtt{Ds}(\mathtt{E}',d') \end{array}$$

where all capital letter symbols are the created oids. To perform the inverse operation required in the information preservation condition, we unnest (using $\mathtt{Unnest}_{T_2}$*) and project on the attributes used by correspondences. In rule notation, this can be written as follows (for the output we use the same relation name* R*):*

$$\mathtt{S}(a,\mathtt{F},\mathtt{Bs}),\ \mathtt{Bs}(b,\mathtt{E},\mathtt{Xs}),\ \mathtt{Xs}(x),\ \mathtt{T}(\mathtt{G},\mathtt{Ds}),\ \mathtt{Ds}(\mathtt{E},d) \;\rightarrow\; \mathtt{R}(a,b,x,d)$$

[17] For this section, the formulation of $\mathtt{Nest}_d$ as a set of rules is more convenient than the one as a query.

By running this rule on the target instance that was computed by $\texttt{Nest}_d$*, we obtain* $\texttt{R}(a, b, x, d)$ *and* $\texttt{R}(a, b, x', d')$*. Thus we get back the same two tuples that were originally in* R*. In general, one can prove that this is always the case (i.e. for any instance of* R*) for this particular example of* $\texttt{Nest}_d$*. This is a consequence of the Skolemization scheme of Section 6.1. If for example, we replace the Skolem terms* $\texttt{E}[a, b, d]$ *and* $\texttt{Xs}[a, b, d]$*, in* $\texttt{Nest}_d$*, by* $\texttt{E}[a, b]$ *and, respectively,* $\texttt{Xs}[a, b]$*, the information preservation property is not satisfied anymore. For the two tuples in our example, by applying the modified* $\texttt{Nest}_d$ *and then the inverse rule, we would obtain four tuples: the original two, plus* $\texttt{R}(a, b, x, d')$ *and* $\texttt{R}(a, b, x', d)$*. The last two tuples were not in* R*.* ∎

The following theorem states that $\texttt{Nest}_d$ is information preserving, for *any* s-t dependency d that is generated by Algorithm 4.15. An important consequence of this result is that one can use Unnest and $\texttt{Nest}_d$ to transmit data in both directions. In other words, for each dependency the mapping is invertible.

Theorem 6.2 (Information Preservation) *For any source-to-target inclusion dependency* d *generated in the semantic translation phase,* $\texttt{Nest}_d$ *is information preserving.*

6.3 XQuery Generation

It is not a difficult task to translate into XQuery the internal query representation that is produced by the query generation algorithm of Section 6.1. Figure 9 illustrates an XQuery generated by our tool for the mapping example of Figure 1(b) when correspondences v_1, v_2, and v_3 are given. To illustrate our techniques, we have assumed that the source schema was an XML schema. The XQuery query follows closely the for_where_return_ form of the internal abstract query. A few details specific to generating XQuery are described here by example.

When an output element/attribute is of atomic type and there are no value correspondences defining it, the internal query representation provides a Skolem term for it. In the XQuery form, we do not always use the Skolem term. More precisely: 1) if the element/attribute is optional, then nothing is generated (like element `proj` in the example); 2) if it is not optional, but is nullable, then an element with a null value is generated (like element `year` in the example); otherwise, the Skolem term is used (like element `aid` in the example). In addition, the internal query representation may use virtual elements that we introduced when we wrapped the XML Schemas in our nested relational representation (see Section 3). Thus, one extra pass through the query is performed to eliminate those names. The final result is the produced XQuery as shown in Figure 9.

7 Architecture and Experimentation

We have implemented our solutions in our prototype Clio. At the heart of Clio is its *mapping engine*. The mapping engine manipulates and interprets the *mapping state*, an internal representation of the input schemas and the value correspondences. The mapping engine is in charge of chasing the constraints at the source and target schemas producing the logical associations. These associations are built once, at schema load time and updated if the user makes any changes to the source or target constraints. Clio provides several tools that help the user with the mapping process. An *attribute matching* tool automatically suggests likely correspondences by matching the data characteristics of source and target attributes. This matching tool is integrated with a schema browser that lets users browse the schemas, modify suggested correspondences, and enter their own correspondences. As value correspondences are entered by the user using our GUI, the semantic translation algorithm is applied to create a set of possible logical mappings. Clio provides visual tools for displaying these alternatives using either schema constructs (in a representation that is similar to our internal tableau data structure) or using the data itself. In the latter case, sample data examples are used to help the user understand the effects of each mapping (and the differences between alternative mappings). The user may interactively select a subset of the mappings.

```
<statDB>
  <cityStat>
    <city>  Null  </city>
    { FOR $x0 IN /expenseDB/grants/grant, $x1 IN /expenseDB/projects/project, $x2 IN /expenseDB/companies/company
      WHERE $x2/cid/text() = $x0/cid/text() AND $x0/proj/text() = $x1/name/text()
      RETURN <org>
               <cid>  { CID($x2/cname/text()) }  </cid>
               <name>  { $x2/cname/text() }  </name>
               { FOR $x0L1 IN /expenseDB/grants/grant, $x1L1 IN /expenseDB/projects/project,
                     $x2L1 IN /expenseDB/companies/company
                 WHERE $x2L1/cid/text() = $x0L1/cid/text() AND $x0L1/proj/text() = $x1L1/name/text()
                 AND $x2/cname/text() = $x2L1/cname/text()
                 RETURN <fund>
                          <pi>  { $x0L1/pi/text() }  </pi>
                          <aid>  { AID($x0L1/amount/text(), $x0L1/pi/text(), $x2L1/cname/text()) }  </aid>
                        </fund>  }
             </org>  }
    { FOR $x0 IN /expenseDB/grants/grant, $x1 IN /expenseDB/projects/project, $x2 IN /expenseDB/companies/company
      WHERE $x2/cid/text() = $x0/cid/text() AND $x0/proj/text() = $x1/name/text()
      RETURN <financial>
               <aid>  { AID($x0/amount/text(), $x0/pi/text(), $x2/cname/text()) }  </aid>
               <amount>  { $x0/amount/text() }  </amount>
               <year>  Null  </year>
             </financial>  }
  </cityStat>
</statDB>
```

Figure 9: An XQuery example generated by our mapping tool

The query generation module uses pluggable query wrappers to produce a final query into one of several query language. Currently, Clio has XQuery and SQL wrappers. New wrappers can be added to support, if needed, other transformation languages like XSLT or even a programming language like Java.

We have tested Clio on a publicly available schema integration test suite that includes four heterogeneous relational schemas for a bibliographic domain [MFH+01]. Although designed as a test benchmark for evaluating and comparing schema integration approaches, pairs of these schemas can also be used to test schema mapping tools. Our developers were able to quickly create mappings for these schemas with our tool. Our algorithms, model and query representations were powerful enough to create the mappings required by these example schemas. As a more robust test of the effectiveness of our tool, we also tested Clio on some real-world schemas representing gene expressions (*microarray*) experiment results. This application included a relational representation of the data (divided into 63 relational tables) and a nested representation (as a DTD). The scientists who provided this data wished to publish their relational data as an XML document using the given DTD. The DTD was deeply nested with up to seven levels of nesting from the root.

Loading both schemas into Clio took approximately 20 seconds while Clio does the chase and computes the existing associations. Note that Clio is an unoptimized prototype, and while not huge, the schemas were of significant size and complexity. After this pre-processing was done, Clio's GUI displayed both schemas and allowed the user to enter correspondences. The response time of each correspondence was immediate and no performance degradation was noticeable as the mapping got more complex. This is significant, since we are invoking our semantic translation algorithm repeatedly as new correspondences are generated to produce new logical mappings.

As an example mapping that nicely shows the capabilities of Clio, the target schema has a set representing `species` which contains a nested set for the `chromosome` information of each species. On the source side, species and chromosome information is divided among two tables (connected with a key/foreign key path). Mapping attributes from the two source tables into the target nested structure immediately displays a single mapping that joins both source tables and produces the correct grouping at the target. However, the user may request to see any other interpretation (or mapping) that is implied by the given correspondences. In this case, Clio reveals a mapping in which the source species are mapped only to the target species without chromosomes. This additional query, thus, extracts species for which no chromosome information may be known in the source database. The user can easily decide if this second query should be part of the final mapping (along with the first). In a more complex example, nine relational tables are mapped into a target nested structure representing the actual experiment results. Here, both source and target constraints are used when producing the resulting mappings. Even for such a complex scenario, there are only six possible mappings. Users can browse each interpretations and quickly select a subset that define data they want to appear in the target XML document. The application scientists were very impressed with the effectiveness of the tool[18]. They had previously had to create such mappings by hand. Not only did this require them to learn an XML query language, but they found creating and debugging their mapping programs very tedious. They were amazed at how easily they could get a nine-way join right using Clio. This experience suggests that in real schemas there is sufficient semantic and structural information so that a tool can effectively manage large parts of the integration process, involving the user only when the semantic information is missing, incorrect or ambiguous.

8 Discussion and Future Work

We have presented new techniques and algorithms for schema mapping and data translation between any given nested source and target schemas with nested referential constraints. In schema mapping, unlike schema integration, one must consider the data content of the target schema and the target constraints, both of which may differ from that of the source schema. Our solutions are unique in that we guarantee that the target constraints are satisfied. We also guarantee that the query generation algorithm is information preserving, even if the target represents data that is not present in the source.

Our solutions are complete for an expressive, yet common, class of nested schemas and nested referential constraints, including some well-behaved cyclic constraints for which our chase procedure terminates.[19] Indeed our motivation was to capture constraints that arise most frequently in XML Schemas. Recursive types and general cyclic referential constraints can lead to infinitely many associations in a schema. Our chase procedure must therefore be modified to limit the depth of the recursion. However, we are still able to enumerate an incomplete, yet likely, set of mappings for such schemas.

One commonly used feature of XML that we have not discussed is union types. To support unions, we would need to modify our coverage algorithm to consider the different attributes of a union type separately. In addition, XML query languages often support the use of schema labels (such as element names) as data. In our terminology this translates to mapping the attribute name of a record in the source to a value in the target. Our approach is based solely on *value* correspondences, meaning that correspondences are defined only between atomic type attributes and refer to their atomic values. Therefore, mappings that require the use of XQuery [CCDF+01] or XML-QL [DFF+99] queries that use element names as data, or Schema-SQL [LSS96] queries, cannot be automatically generated by our system. A final limitation of our data model is that it supports only sets, hence, ordering information is not taken into consideration. To support order, we would need to extend our data model to include list types and enhance the generated mappings with ordering predicates.

Each mapping query we produce is information preserving. However, to populate the target, we must combine the results of a set of mapping queries. As with the individual queries, we want this combination process to both minimize redundancy and preserve information. We are currently investigating nested extensions to the relational *minimal union* operator which eliminates redundancies when integrating relational data [GL94, RU96]. Finally, we are considering general techniques for optimizing data translation queries.

[18] So impressed that they are now funding this research!

[19] The definition of this class and proof of completeness will appear in the full version of this paper.

References

[AB86] S. Abiteboul and N. Bidoit. Non-first Normal Form Relations: An Algebra Allowing Data Restructuring. *Journal of Computer and System Sciences*, 33:361–393, December 1986.

[ACM97] S. Abiteboul, S. Cluet, and T.. Milo. Correspondence and Translation for Heterogeneous Data. In *Proc. of the Int'l Conf. on Database Theory (ICDT)*, pages 351–363, 1997.

[ASU79] A. V. Aho, Y. Sagiv, and J. D. Ullman. Efficient optimization of a class of relational expressions. *ACM TODS*, 4(4):435–454, December 1979.

[AT97] P. Atzeni and R. Torlone. MDM: A Multiple-Data Model Tool for the Management of Heterogeneous Database Schemes. In *ACM SIGMOD Conference*, pages 528–531, New York, May 1997.

[BV84] C. Beeri and M. Y. Vardi. A Proof Procedure for Data Dependencies. *Journal of the ACM*, 31(4):718–741, 1984.

[CCDF+01] D. Chamberlin, J. Clark, J. Robie D. Florescu, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML Query Language. W3C Working Draft, June 2001. http://www.w3.org/TR/xquery/.

[CDSS98] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your Mediators Need Data Conversion. In *ACM SIGMOD Conference*, pages 177–188, 1998.

[CFP84] M. A. Casanova, R. Fagin, and C. H. Papadimitriou. Inclusion Dependencies and their Interaction with Functional Dependencies. *Journal of Computer and System Sciences*, 28(1):29–59, 1984.

[CK86] S. S. Cosmadakis and P. C. Kanellakis. Functional and Inclusion Dependencies: A Graph Theoretic Approach. In *Advances in Computing Research*, volume 3, pages 163–184. JAI Press, 1986.

[DDH01] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach. In *ACM SIGMOD Conference*, pages 509–520, May 2001.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. In *Proceedings of the 8th International World Wide Web Conference*, pages 77–91, 1999.

[DK97] S. Davidson and A. Kosky. WOL: A Language for Database Transformations and Constraints. In *Proc. of the Int'l Conf. on Data Eng.*, pages 55–66, April 1997.

[Fal01] D. C. Fallside. XML Schema Part 0: Primer. W3C Recommendation, May 2001. http://www.w3.org/TR/xmlschema-0/.

[FMU82] R. Fagin, A. O. Mendelzon, and J. D. Ullman. A Simplified Universal Relation Assumption and Its Properties. *ACM TODS*, 7(3):343–360, September 1982.

[FV86] R. Fagin and M. Y. Vardi. The Theory of Data Dependencies - A Survey. In M. Anshel and W. Gewirtz, editors, *Proc. of Symposia in Applied Mathematics*, volume 34 - Mathematics of Information Processing, pages 19–71. American Mathematical Society, Providence, Rhode Island, 1986.

[GL94] C. A. Galindo-Legaria. Outerjoins as Disjunctions. In *ACM SIGMOD Conference*, pages 348–358, 1994.

[HY90] R. Hull and M. Yoshikawa. ILOG: Declarative Creation and Manipulation of Object Identifiers. In *Proc. of the Int'l Conf. on VLDB*, pages 455–468, August 1990.

[LL94] M. Leven and G. Loizou. The Nested Universal Relation Data Model. *Journal of Computer and System Sciences*, 49(3):683–717, 1994.

[LSS96] L. Lakshmanam, F. Sadri, and I. N. Subramanian. SchemaSQL - A Language for Interoperability in Relational Multi-database Systems. In *Proc. of the Int'l Conf. on VLDB*, Bombay, India, 1996.

[MBR01] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. In *Proc. of the Int'l Conf. on VLDB*, 2001.

[MFH+01] R. J. Miller, D. Fisla, M. Huang, D. Kymlicka, F. Ku, and V. Lee. The Amalgam Schema and Data Integration Test Suite. www.cs.toronto.edu/ miller/amalgam, 2001.

[MHH00] R. J. Miller, L. M. Haas, and M. Hernández. Schema Mapping as Query Discovery. In *Proc. of the Int'l Conf. on VLDB*, Cairo, Egypt, 2000.

[MUV84] D. Maier, J. D. Ullman, and M. Y. Vardi. On the Foundations of the Universal Relation Model. *ACM TODS*, 9(2):283–308, June 1984.

[MZ98] T. Milo and S. Zohar. Using Schema Matching to Simplify Heterogeneous Data Translation. In *Proc. of the Int'l Conf. on VLDB*, pages 122–133, NY, NY, 1998.

[NHT+02] F. Naumann, C. T. Ho, X. Tian, L. M. Haas, and N. Megiddo. Attribute Classification Using Feature Analysis. In *Proc. of the Int'l Conf. on Data Eng.*, 2002. (Poster Presentation).

[ÖY87] Z. Meral Özsoyoglu and L. Yuan. A New Normal Form for Nested Relations. *ACM TODS*, 12(1):111–136, 1987.

[PT99] L. Popa and V. Tannen. An Equational Chase for Path-Conjunctive Queries, Constraints, and Views. In *Proc. of the Int'l Conf. on Database Theory (ICDT)*, pages 39–57, 1999.

[RB01] E. Rahm and P. A. Bernstein. On matching schemas automatically. *The Int'l Journal on Very Large Data Bases*, 2001.

[RK87] M. A. Roth and H. F. Korth. The Design of non-1NF Relational Databases into Nested Normal Form. In *ACM SIGMOD Conference*, pages 143–159, 1987.

[RR99] S. Ram and V. Ramesh. Schema Integration: Past, Current and Future. In A. Elmagarmid, M. Rusinkiewicz, and A. Sheth, editors, *Management of Heterogeneous and Autonomous Database Systems*, pages 119–155. Morgan Kaufmann Publishers, 1999.

[RU96] A. Rajaraman and J. D. Ullman. Integrating Information by Outerjoins and Full Disjunctions. In *Proc. of the ACM Symp. on Principles of Database Systems (PODS)*, pages 238–248, Montréal, Canada, 1996.

[YMHF01] L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data-Driven Understanding and Refinement of Schema Mappings. In *ACM SIGMOD Conference*, pages 485–496, 2001.