Proc. 2002 Very Large Databases Conference (VLDB'02), pp. 598-609

Translating Web Data

Lucian Popa[†] Yannis Velegrakis[‡] Renée J. Miller[‡] Mauricio A. Hernández[†] Ronald Fagin[†]

[†]IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120

[‡]University of Toronto
6 King's College Road
Toronto, ON M5S 3H5

Abstract

We present a novel framework for mapping between any combination of XML and relational schemas, in which a high-level, user-specified 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, the high-level mapping, expressed as a set of inter-schema correspondences, is converted into a set of mappings that capture the design choices made in the source and target schemas (including their hierarchical organization as well as their nested referential constraints). The second phase translates these mappings into queries over the source schemas that produce data satisfying the constraints and structure of the target schema, and preserving the semantic relationships of the source. Non-null target values may need to be invented in this process. The mapping algorithm is complete in that it produces **all** mappings that are consistent with the schema constraints. We have implemented the translation algorithm in Clio, a schema mapping tool, and present our experience using Clio on several real schemas.

1 Introduction

An important issue in modern information systems and e-commerce applications is providing support for inter-operability of independent data sources. A broad variety of data is available on the Web in distinct heterogeneous sources, stored under different formats: database formats (e.g., relational model), semi-structured formats (e.g., DTDs, SGML or XML Schema), scientific formats, 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) to another requires writing and managing complex data transformation programs or queries.

We present a new, comprehensive solution to building, refining and managing transformations between heterogeneous schemas. Given the prevalent use of the Web for data exchange, any data translation tool must handle not only relational data but also data represented in nested data models that are commonly available on the Web. Our solutions are applicable to any structured and semi-structured data that can be described by a schema (a relational schema, a nested XML Schema or DTD). We do not, in this work, consider the exchange of documents or unstructured data (including multimedia and unstructured text). Our approach can be distinguished by its treatment of two fundamental issues. We discuss them below, highlighting our contributions and the main related work.

Heterogeneous Semantics We consider the *schema mapping* problem, where we are given a pair of independently created schemas and asked to translate data from one (the source) to the other (the target). The schemas may have different semantics, and this may be reflected in differences in their logical structures and constraints. In contrast, most work on heterogeneous data focuses on the *schema integration* problem where the target (global) schema is created from one or more source (local) schemas (and designed as a view over the sources) [8].[1] The target is created to reflect the semantics of the source and has no independent semantics of its own. Even our own earlier work on schema mapping considered the problem of mapping a source schema (with a rich logical structure) into a flat (single table) target schema with no constraints, thus ignoring half of the more general problem [12]. In contrast, Section 2 gives a *semantic translation* algorithm that preserves semantic relationships during the translation from source to target, where the source and

**Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002**

[1] Alternatively, in a local-as-view approach each source schema is modeled as a view on the target schema [8].

the target schemas may contain very rich, yet highly heterogeneous constraints.

Heterogeneous Data Content We do not assume that the source and target schema represent the same data. Certainly there may be source data that are not represented in the target (and should be omitted from the translation). More interesting, however, is the case where there is a need for data in the target that are not represented in the source. In some cases, values must be produced for undetermined elements (attributes) in the target schema (i.e., target elements for which there is no corresponding source element). Values may be needed if the target element cannot be null (e.g., elements in a key) and no default is given. More importantly, the creation of new values for such target elements is essential for ensuring the consistency of the target data (e.g., we may need to create values for foreign keys in the target, to ensure that source data is correctly mapped). This problem has been previously addressed by specialized translation languages that include Skolem functions for value creation (most notably ILOG [7]). However, previous research has not considered the problem of automatically determining a correct set of Skolem functions that respects the target constraints and preserves information in a translation. Our data translation algorithm, in Section 3, provides a new solution for automatically generating missing target values, based on Skolem functions, and guarantees that the translated data satisfies the nested structure and constraints of the target.

In developing our solutions, we paid special attention to developing algorithms and techniques that could fuel our schema mapping tool Clio. Hence, the overall design was motivated by practical considerations. Clio [15] has been evaluated on several real world schemas with promising results. Section 4 further details our experience with mapping these schemas.

1.1 Our Solutions

We highlight the main design choices and illustrate our solutions with examples.

Example 1.1 *Consider the two schemas in Figure 1. Both schemas are shown in a common nested relational representation (defined in Section 2.1) which is used in our tool and throughout this paper. 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 grants given for those projects. 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 grouped by city. For each different city, there is an element* `cityStat` *containing the organizations (*`org`*) and the grants (*`financials`*) in that city. Project funding data are then nested within* `org` *and related with the* `financial` *information through a foreign key based on the* `aid` *element.* ∎

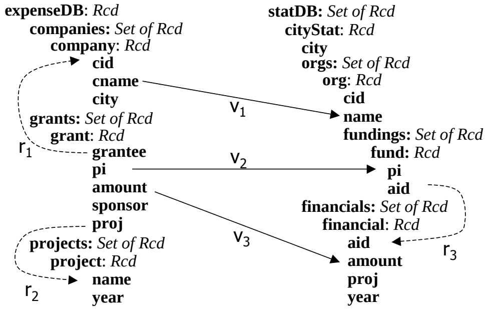

Figure 1: Example source (left) & target (right).

Inter-Schema Correspondences To perform translation, we must understand how two schemas correspond to each other. There are numerous proposals for representing inter-schema correspondences, including proposals for using general logic expressions to state how components of one schema correspond to components of another. We use perhaps the simplest form of correspondence: element (attribute) correspondences. Intuitively, an element correspondence is a pair of a source element and a target element (we give a formal definition in Section 2). Figure 1 shows three example correspondences: v_1, v_2 and v_3.

While semantically impoverished, we use simple element correspondences for several reasons. First, element correspondences are independent of logical design choices such as the grouping of elements into tables (normalization choices) or the nesting of records or tables (for example, the hierarchical structure of an XML schema). Thus, one need not specify the logical access paths (join or navigation) that define the associations between elements involved. Even users unfamiliar with the complex structure of the schema can provide such correspondences. In addition, automated techniques for *schema matching* have proven to be very successful in extracting such correspondences (see the recent work of [6] as well as [18] for a survey of techniques including CUPID, LSD and DIKE). In Clio, we use a modular design that lets us plug in any schema matching component. The current version uses an automated attribute matcher to suggest correspondences [14] and provides a GUI to permit users to augment or correct those correspondences [19].

Understanding Schema Semantics While easy to create and manipulate, element correspondences are inherently ambiguous. There may be many translations that are consistent with a set of correspondences, and not all have the same effect In our approach we then find, among the many possible translations, those that are consistent with the constraints of the schemas.

Example 1.2 *Consider correspondences v_1 and v_2. To translate data, 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 from the source schema to determine which combinations of values are meaningful. The foreign key constraint between* `grant.grantee` *and* `company.cid` *indicates that each* `grant` *(and its* `pi`*) is naturally associated with one* `company` *(through a foreign key). To populate the target, that is to place values of* `cname` *and* `pi` *correctly in the target, 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 the nested* `fundings` *set.*

There are many semantic associations in a schema, and even the same set of elements 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.* ∎

Reasoning about data semantics can be a time consuming process for large schemas. Clio supports incremental creation and modification of mappings. Thus, it is important that such modifications can be made efficiently. To support this, we compile the semantics of the schemas into a convenient data structure that represents the semantic relationships embedded in each schema. Using this compiled form, our semantic translation algorithm efficiently interprets correspondences.

Semantic Translation Our semantic translation provides an interpretation of correspondences that is faithful to the semantics of the schemas. In addition, we enumerate all such faithful interpretations, which we call *logical mappings.* Enumeration of all such mappings is an essential ingredient of our approach. 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 schema specifications, 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 (see Section 4) and that there are real situations in which the intended semantics includes all these mappings. Thus, a heuristic approach that prunes some consistent mappings will not work in general. To reduce the burden on the user, we order mappings so that users can focus on the most likely mappings, and we provide a data viewer (described elsewhere [19]) that uses carefully choosen data examples to help explain each mapping.

Data Translation The second translation phase is *data translation*, in which we generate an implementation of the logical mappings. The result of this phase is a set of internal *rules*, one for each logical mapping. These rules have a direct translation as external queries, and we provide query wrappers for XQuery and XSLT (in the XML case) and SQL (in the relational case). To correctly translate data, values may need to be produced for undetermined target elements and the data may need to be nested according to the target structure.

Example 1.3 *In Figure 1,* `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 generates a logical mapping that includes both v_2 and v_3 (v_1 as well, but let us focus on v_2 and v_3). However, to populate the target, we must have values for the two* `aid` *elements. To maintain the proper association in the target, these values may neither be arbitrary nor null. 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 source data associations.* ∎

In Section 2, we present our semantic translation algorithm, along with a completeness guarantee that all semantic relationships that exist between elements in a schema are discovered by the algorithm. Section 3 contains the data translation algorithm that converts logical mappings into queries using rich restructuring constructs (resembling ILOG [7], WOL [4] and XML-QL [5]). Section 4 describes our experience mapping real schemas using our prototype.

2 Semantic Translation

To perform schema mapping, we seek to interpret the correspondences in a way that is consistent with the semantics of both the source and target schemas. We call this interpretation process *semantic translation.* Since we use semantics that is encoded in logical structures, we call the resulting interpretation a *logical mapping.*

2.1 Data Model

To present our results, we use a simple nested relational data model. In our tool, relational and semi-structured schemas (including DTDs and XML Schemas) are represented internally in this model.

The model includes a set of atomic types τ, set types of the form SetOf[τ], and record types of the form Rcd[$A_1 : \tau_1, \ldots, A_k : \tau_k$], where each τ_i represents either an atomic, set, or record type. The symbols $A_1, \ldots, A_k$ are called *labels* or *elements*. If τ_i is an atomic type, then A_i is an atomic element. Records of type Rcd[$A_1 : \tau_1, \ldots, A_k : \tau_k$] are unordered tuples of label-value pairs: $\langle A_1 = a_1, \ldots, A_k = a_k \rangle$, where $a_1, \ldots, a_k$ are of types $\tau_1, \ldots, \tau_k$, respectively. Within a record, elements are non-repeatable. Elements that may be repeated are modeled by set types. A value of type SetOf[τ] is represented by *a set ID* and an associated set $\{e_1, \ldots, e_n\}$ of "children", with each e_i of type τ. This representation of sets (using set IDs) is used to faithfully capture the graph-based data models of XML. For ease of exposition, we assume in this paper that a schema consists of a single named (root) type (as in XML Schema[2]). A relational schema with multiple tables is modeled by a record type the components of which are set types (one for each table). In queries and constraints, we refer to an element of a set by using clauses such as c in expenseDB.companies. To refer to the value (contents) of such elements, we use record projection notation (e.g. c.company).

Our model also supports optional and nullable elements, along with key constraints. Referential constraints (foreign keys) can be expressed as explained in the next subsection.

2.2 Primary Paths and Constraints

Consider the correspondences v_1 and v_2 of Figure 1. To understand these arrows, we must first understand if company names are semantically related to the PIs of grants in the source schema and also if the target elements org.name and fund.pi are associated. If not, then we can treat these correspondences separately. That is, we can map company data independently of grant data. However, if any of these elements are associated, we must interpret the correspondences in a way that is consistent with this association. Hence, our first step is to learn how elements may be semantically related within each schema.

In relational schemas, semantic associations between atomic elements are represented in two ways. First, the organization of attributes into tables (the schema structure) indicates semantic groupings. The fact that cname and city are attributes of the same table indicates that for each tuple these attribute values are semantically related. 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.

S_1 : select * from c in expenseDB.companies
(a) S_2 : select * from g in expenseDB.grants
S_3 : select * from p in expenseDB.projects

T_1 : select * from s in statDB
T_2 : select * from s in statDB, o in s.cityStat.orgs
(b) T_3 : select * from s in statDB, o in s.cityStat.orgs,
f in o.org.fundings
T_4 : select * from s in statDB, f' in s.cityStat.financials

Figure 2: Primary paths of (a) expenseDB (b) statDB.

Similarly, in semi-structured data models, both the schema structure and constraints can represent semantic associations. The nesting structure of the schema represents semantic groupings of elements. This structure may then be augmented with (nested) referential constraints (often in the form of simple pointers or key references as in XML Schema) to provide a richer semantics for connecting related elements.

In this section, we consider how to compute sets of semantically related atomic elements either within the source schema or within the target schema. We first represent the elements related through the schema structure without constraints. We refer to these sets as *primary paths* and show next how they are computed.

Example 2.1 *Figure 2 shows the primary paths for the schemas of Figure 1, represented as queries that return a set of semantically related atomic elements. We use the notation "*select* *" as a shorthand for all atomic elements immediately reachable from a variable of the query (this excludes atomic elements contained in nested set elements). For example, in T_1, "*select* *" is equivalent to "*select* s.cityStat.city" since all other atomic elements underneath* cityStat *are reachable only through a nested set type (either* orgs *or* financials*).* ■

For a relational schema, there is a primary path for each individual relation. For a nested schema, the primary paths are obtained by constructing a tree with a node at each set type in the schema, and with an edge between two nodes whenever the first node is a set type that contains the second. A primary path is then the set of all elements found on a path from the root to any intermediate node or leaf in this tree.

For our nested schema example, statDB, the first primary path T_1 denotes the set of cityStat records (within statDB) that may or may not have orgs 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 (that is, relationships between two records where one is not nested within the other), we need to consider the various ways in which primary paths can be associated. We consider all associations that are faithful to the semantics conveyed by the (nested) referential integrity constraints in the schemas. Referential constraints assert the equality of an element (or set of

[2] http://www.w3.org/TR/xmlschema-0

elements) in two primary paths. They can be used to combine primary paths into larger sets of logically related elements.

Example 2.2 *For the schemas of Figure 1, the two source foreign keys can be represented as follows.*

```
r1:  for g  in expenseDB.grants
     exists c  in expenseDB.companies
     where c.company.cid= g.grant.grantee
r2:  for g  in expenseDB.grants
     exists p  in expenseDB.projects
     where p.project.name= g.grant.proj
```

Each constraint is of the form $\forall P_1 \exists P_2 B$ where P_1 and P_2 are bodies of primary paths and B is an equality condition relating the two paths. Path P_1 in r_1 is the primary path S_2 of Figure 2, and P_2 is S_1. In constraints, we write the primary paths in a simplified form ignoring the atomic elements they return in the select clause. For the target schema, we use a similar notation to represent nested referential integrity constraints:

```
r3 :  for s  in statDB, o  in s.cityStat.orgs, f  in o.org.fundings
      exists f'  in s.cityStat.financials
      where f'.financial.aid= f.fund.aid    ■
```

For the nested constraint in our example, we used a primary path, then a second path that is relative to the first (through the variable s 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.

Definition 2.3 *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* for P_1 exists P_2 where B.

Nested referential integrity constraints include a large class of referential constraints. Relational foreign keys fall into this class along with XML Schema's Key Reference constraints. Note that our solutions do not require that there be any declared constraints. We simply use these constraints to our advantage if they are declared or if they are suggested by a constraint-discovery tool.

Constraints can be used to combine primary paths.

Example 2.4 *Constraint r_3 of Example 2.2 represents a horizontal relationship between the primary paths T_3 and T_4 of Figure 2. Using r_3, we can combine these paths to form the logical relation B_3 of Figure 3. Note that the logical relation contains only one copy of equated elements (*`aid` *in this case). Intuitively, B_3 represents* `fund` *information with its* `org` *and* `city` *(associated through the nesting structure) and its* `financial` *data (associated through the key reference on* `aid`*).* ■

2.3 Logical Relations

We now consider how to generate maximal sets of logically related elements. The *chase* is a classical relational method [10] that can be used to assemble logically connected elements. Intuitively, the chase works by enumerating logical joins, based on a set of dependencies, in a relational schema. We use an extension [16] of the relational chase to enumerate logical joins in nested schemas.

Definition 2.5 *A* logical relation *is the result of chasing a primary path of a schema using its NRIs.*

We assume the reader is familiar with the basics of the chase. Here, we illustrate with an example.

Example 2.6 *The result of chasing the source primary path S_2 of Figure 2 using the constraint r_1 of Example 2.2 can be represented by the following query.*

```
S2' :  select *
       from g  in expenseDB.grants, c  in expenseDB.companies
       where c.company.cid= g.grant.grantee
```

Note that S_2 is not equivalent to S'_2 since the latter returns a larger set of atomic elements. (However, S'_2 projected on the elements of S_2 is equivalent to S_2.) Applying the chase again to S'_2 using r_2 we obtain the following query.

```
S2'' :  select *
        from g  in expenseDB.grants, c  in expenseDB.companies,
             p  in expenseDB.projects
        where c.company.cid= g.grant.grantee and
             g.grant.proj= p.project.name
```

Since no further chase steps can be applied (i.e., S''_2 cannot be expanded further) S''_2 is a logical relation. 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.*

The primary path S_1 cannot be chased with either of the two NRIs in the schema. Intuitively, this means that S_1 tuples can exist without any other tuples; hence they make up a logical relation by themselves. Figure 3 lists the logical relations for our schemas (note that S''_2 is A_2 in the figure). In our nested target, paths T_1, T_2 and T_4 cannot be chased, so they form logical relations by themselves (B_1, B_2 and B_4, respectively). The primary path T_3 can be chased using r_3 to produce the logical relation B_3. ■

2.4 Mapping Algorithm

Logical relations are used to understand correspondences. In our example, the existence of a logical (source) relation containing both the elements `company.cname` and `grant.pi` tells us that the correspondences v_1 and v_2 should be interpreted together. In this section, we formalize this intuition and give an algorithm for interpreting correspondences.

```
A1: select c.company.cid, c.company.cname, c.company.city
    from c  in expenseDB.companies
A2: select c.company.cid, c.company.cname, c.company.city,
           g.grant.pi, g.grant.amount, g.grant.sponsor,
           g.grant.proj, p.project.year
    from g  in expenseDB.grants, c  in expenseDB.companies,
           p  in expenseDB.projects
    where c.company.cid= g.grant.grantee and
           p.project.name= g.grant.proj
A3: select p.project.name, p.project.year
    from p  in expenseDB.projects

B1: select s.cityStat.city from s  in statDB
B2: select s.cityStat.city, o.org.cid, o.org.name
    from s  in statDB, o  in s.cityStat.orgs
B3: select s.cityStat.city, o.org.cid, o.org.name,
           f.fund.pi, f.fund.aid, f'.financial.amount,
           f'.financial.proj, f'.financial.year
    from s  in statDB, o  in s.cityStat.orgs,
           f  in o.org.fundings, f'  in s.cityStat.financials
    where f'.financial.aid= f.fund.aid
B4: select s.cityStat.city, f'.financial.aid, f'.financial.amount,
           f'.financial.proj, f'.financial.year
    from s  in statDB, f'  in s.cityStat.financials
```

Figure 3: All logical relations for expenseDB, statDB.

Example 2.7 *The correspondences v_1 and v_2 (Figure 1) can be interpreted as simple (inter-schema) referential constraints.*

```
v1  for c  in expenseDB.companies
    exists s  in statDB, o  in s.cityStat.orgs
    where c.company.cname= o.org.name
v2  for g  in expenseDB.grants
    exists s  in statDB, o  in s.cityStat.orgs, f  in o.org.fundings
    where g.grant.pi= f.fund.pi
```

The correspondence v_1 uses the primary paths S_1 from the source and T_2 from the target. (The primary paths are used simply as a way of unambiguously referring to atomic elements in the nested schemas.) Similarly, v_2 uses S_2 and T_3. This interpretation ignores the semantics of the source and target schemas. It indicates that fundings *are created from* grants *but does not say that* fundings *must be nested within an* org *generated from the appropriate company (that is, the company receiving the grant). To create an interpretation that is faithful to the semantics of the schemas, we can use logical relations rather than primary paths to state the (inter-schema) constraint. The source elements* cname *and* pi *both appear in the logical relation A_2 (Figure 3) which represents how* companies *and* grants *are logically associated. The target elements* name *and* pi *both appear in the logical relation B_3 (which represents how* orgs *and* fundings *are logically associated). Hence, we can combine v_1 and v_2 using these two logical relations into an interpretation v_{12}.*

```
v12  for g  in expenseDB.grants, c  in expenseDB.companies,
         p  in expenseDB.projects
          where c.company.cid= g.grant.grantee
                and p.project.name= g.grant.proj
     let w1 = c.company.cname, w2 = g.grant.pi
     exists s  in statDB, o  in s.cityStat.orgs,
         f  in o.org.fundings, f'  in s.cityStat.financials
          where f'.financial.aid= f.fund.aid
                and o.org.name= w1  and f.fund.pi= w2
```

The for *clause comes from the source logical relation A_2. It indicates which source data should be used to populate the target. We have added a* let *clause to make it clear which source data will be translated. The* exists *clause comes from the target relation B_3 and indicates how the source data should appear in the target. The* where *clause includes the target join condition (from B_3) along with conditions indicating the placement of the source data w_1 and w_2.*

So far, the example ignored v_3. If we consider v_3 then amount *of* financial *also appears in the target logical relation B_3. We would then replace v_{12} with a more meaningful v_{123} that covers v_1, v_2* and *v_3. The interpretation v_{123} is similar to v_{12} above except that it binds one more variable in the* let *clause and it has one more equality in the target* exists *clause (corresponding to v_3).* ∎

We describe next the algorithm for creating interpretations like v_{12}. The algorithm begins with a set of correspondences expressed using the primary paths of the schema (such as v_1 and v_2 in Example 2.7). A *correspondence* is a simple case of an (inter-schema) referential constraint that specifies the placement of source values in the target.

Definition 2.8 *Given a primary path P_1 from the source and P_2 from the target, along with an equality condition B equating a single atomic element of P_1 with a single atomic element of P_2, a* correspondence *is an expression of the form* for *P_1* exists *P_2* where *B.*

To determine sets of correspondences that can be interpreted together, we find sets of correspondences that all use source elements in one logical relation and target elements in one target logical relation. As seen in the previous section, logical relations are not necessarily disjoint. For example, A_1 and A_2 of Figure 3 both include company information; however, A_2 also includes 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 (Algorithm 2.11) looks at each pair of a source logical relation and a target logical relation. For each such pair, it then computes an interpretation of the correspondences that expresses how the source logical relation is mapped into the target logical relation. Like correspondences, these interpretations are modeled as *source-to-target (s-t) referential constraints or dependencies*. However, instead of primary paths, these interpretations use logical relations. The constraint v_{12} of Example 2.7 is an example of an s-t dependency. The computation of the dependency is driven by *all* the correspondences that are *covered* by a pair of logical relations.

Coverage of a correspondence v by a logical relation is slightly more complicated than is suggested by Example 2.7. It is not enough to check whether the logical relation includes the element name involved in

v. This is ambiguous, in general. We do not assume that element names are different across the schema. Moreover, the *same* element of a schema may be included more than once in a logical relation (see the next example). To precisely identify which element in the logical relation corresponds to the element involved in the correspondence, we need to match the path defining v with the variables defining the logical relation. Specifically, we must find a *renaming function* from the variables of a correspondence into the variables of a logical relation. Since we are using nested schemas, we also have the additional problem of matching paths, rather than flat element references. In general, there may be multiple ways to cover a correspondence with respect to a pair of source and target logical relations. Our algorithm takes into account all such choices.

Example 2.9 *Assume that the source schema of Figure 1 has an additional restriction that the sponsor is always a company. This means that* `sponsor` *is also a foreign key referencing* `company`. *We can describe it by the following NRI:*

```
r4  for g  in expenseDB.grants
    exists c  in expenseDB.companies
    where c.company.cid= g.grant.sponsor
```

The logical relation A_2 in Figure 3 is the result of chasing the primary path S_2 of Figure 2 with the constraints r_1 and r_2. Given the additional constraint r_4, we can extend A_2 to the logical relation:

```
A'2  select c.company.cid, c.company.cname, c.company.city,
            g.grant.pi, g.grant.amount, g.grant.proj,
            c'.company.cid, c'.company.cname, c'.company.city,
            p.project.year
     from g  in expenseDB.grants, c  in expenseDB.companies,
            c'  in expenseDB.companies, p  in expenseDB.projects
     where c.company.cid= g.grant.grantee and
            c'.company.cid= g.grant.sponsor and
            p.project.name= g.grant.proj
```

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. More specifically, there are two possible renaming functions. 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 element `cname` *in v_1 can be renamed in two ways with the* `cname` *element of the source schema. This reflects 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.* ∎

Definition 2.10 *A* coverage *of a pair $\langle A, B \rangle$ of logical relations is a set V' of correspondences together with renaming functions from the variables of correspondences in V' into those of A and B.*

```
Algorithm 2.11 - Semantic Translation

Input: source schema $S$ with NRIs $\Sigma_s$,
       target schema $T$ with NRIs $\Sigma_t$,
       set $V$ of correspondences.
Phase 1. Compile schemas
       Chase $\Sigma_s$ on $S$ to compute source logical relations
                    $\{A_1, \ldots, A_n\}$.
       Chase $\Sigma_t$ on $T$ to compute target logical relations
                    $\{B_1, \ldots, B_m\}$.
Phase 2. Interpret correspondences
       For each pair $\langle A_i, B_j \rangle$
           For each coverage $\delta$ of $\langle A_i, B_j \rangle$ by $V' \subseteq V$
               create s-t dependency: for $A_i$ exists $B_j$ where $V'$
Output: the set $\Sigma_{st}$ of all generated s-t dependencies
```

For each coverage, we generate a source-to-target dependency of the form for A exists B where C with C containing the equality conditions of the correspondences in V' (with the variables renamed appropriately). Note that since A and B are logical relations, they may each have their own where clause as in our example v_{12} of Example 2.7 where we wrote C inside the where clause of B. The result of the semantic translation algorithm (Algorithm 2.11) is a systematic *enumeration of all s-t dependencies.* The full paper [17] gives further details, including a component that eliminates redundant s-t dependencies, thus drastically reducing the number of interpretations.

2.5 Complexity and Completeness

Logical Compilation NRIs generalize the class of relational inclusion dependencies [2]. They are also a special case of the embedded path-conjunctive dependencies (EPCDs) used in [16] to express constraints in nested and object-oriented schemas. The chase that we use to compute logical relations is a special case of the chase with EPCDs. Even for this special case, the chase may not terminate. One possible solution to this problem is to restrict the class of legal constraints so that the chase is guaranteed to terminate. Acyclic sets of inclusion dependencies [3] are a particularly useful class since they capture naturally many of the integrity constraints of relational schemas, and most of the constraints met in real world semi-structured schemas. For acyclic sets of NRIs (fully defined in [17]), our compilation terminates in polynomial time in the size of the schema and the given NRIs. Furthermore, it is complete as we mention next.

We have given a procedural definition of logical relations (that is, the result of chasing along any primary path of the schema). The chase clearly finds "natural" associations between elements. What is not so obvious is whether *all* the natural associations are discovered by our algorithm. In fact, we can dust off some old relational theory to find a declarative definition of the set of all logical associations or *connections* that exist in a schema [11]. Using their definition, we can show that our algorithm is complete in that it finds all such connections (and only those connections).

Theorem 2.12 (Completeness) *Let S be a schema with an acyclic set of NRIs. Let* $\mathtt{X}$ *be any set of atomic elements in S. Then there is a* connection *between* $\mathtt{X}$ *[11] iff there exists a logical relation that contains* $\mathtt{X}$*.*

Algorithm 2.11 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 logical mapping for a given set of correspondences, as long as the elements have the same meaning.

Cyclic schemas are schemas with cyclic sets of constraints (consider the standard example of the employees whose managers are also employees), or with recursively defined strutures (e.g., a part of a machine is composed of other parts). Mapping between such schemas may result to infinitely many possible translations. Clio permits users to work with such schemas but only in a restricted way that gives up completeness. In particular it stops the chase on every branch that closes a cycle. As a result, when mapping employee name and address, Clio will associate an employee name with the employee's address (but not with the manager's address neither with the manager's manager's address, etc.)

Interpretation of Correspondences To interpret correspondences, we must compute coverages – that is, the set of elements covered by a logical relation. This process is similar to that of determining when a view can be used to answer a query [9]. However, it is important to note that the correspondences are represented by simple paths. Hence, the process of matching them to logical relations is very fast (linear for each correspondence and relation).

We interpret correspondences as source-to-target dependencies. An important consequence of our mapping algorithm is the fact that any implementation of the generated 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 3 gives a canonical implementation of s-t dependencies as queries. Here we state the following result (for any implementation).

Theorem 2.13 *Let Σ_t and Σ_{st} be as in Algorithm 2.11. Moreover, we require that Σ_t is an acyclic set of NRIs. Let $\mathtt{I}_s$ and I_t be instances over the source and, respectively, target schemas. Then I_t satisfies Σ_t whenever I_s and I_t together satisfy Σ_{st}.*

This result allows us to focus on the implementation of s-t dependencies alone, i.e., we do not have to worry about satisfaction of target constraints. They have already been taken into account when producing the target logical relations (via the chase with Σ_t). This theorem does not hold for Algorithm 2.11 if we extend the class of legal target constraints from NRIs to arbitrary nested dependencies (e.g. join dependencies) [17]. Thus, it is essential that NRIs are based on linear paths for the above theorem to hold. Still, NRIs are a very useful fragment for representing integrity constraints in relational and nested schemas, as they include the referential constraints expressible in most standards (notably SQL and XML Schema).

3 Data Translation

The source-to-target dependencies generated by Algorithm 2.11 are inclusion dependencies (in the relational sense [2]) between source logical relations and target logical relations. For example, v_{12} of Section 2.4 is an inclusion dependency $A_2[\mathtt{cname}, \mathtt{pi}] \subseteq B_3[\mathtt{name}, \mathtt{pi}]$ meaning that the projection of logical relation A_2 on `cname` and `pi` must be contained in the projection of B_3 on `name` and `pi`. If we also take the correspondence v_3 into account, then v_{12} is replaced by a similar dependency, $A_2[\mathtt{cname}, \mathtt{pi}, \mathtt{amount}] \subseteq B_3[\mathtt{name}, \mathtt{pi}, \mathtt{amount}]$, that "specifies" one more atomic emelement of the target. Still, not all atomic elements of the target logical relation may be specified by a dependency. In order to materialize the target we will have to fill in the values for the undetermined atomic elements. Null values may not always be sufficient. Moreover, we do not want to materialize a target logical relation but rather the nested schema over which the logical relation is only a flat view.

These issues are addressed by the second phase of our translation process: **data translation**. Specifically, two main problems are considered: 1) *creation of new values in the target*, whenever such values are needed and whenever they are not specified by the dependencies, and 2) *grouping of nested elements*, a form of data merging that is also not specified by the dependencies but is often desirable. We illustrate these issues with some simple examples on which we also highlight our design principles. These design principles are implemented by a component of our system that compiles the s-t dependencies into a set of low-level, but language-independent, **rules**. These rules in turn are compiled into the transformation language at hand (XSLT or XQuery for XML, SQL for relational).

3.1 Creation of New Values in the Target.

Consider the simple mapping scenario of Figure 4(a). The source ($\mathtt{Emps}_1$) and the target (`Emps`) are sets of employee (`Emp`) elements. An `Emp` element in the source has atomic subelements $\mathtt{A}, \mathtt{B}$ and `C`, while an `Emp` element in the target has an extra atomic subelement `E`. For the purpose of the discussion, we chose to use the abstract names $\mathtt{A}, \mathtt{B}, \mathtt{C}$ and `E` because we will associate several scenarios with these elements. In the mapping, the two source elements `A` and `B` are mapped

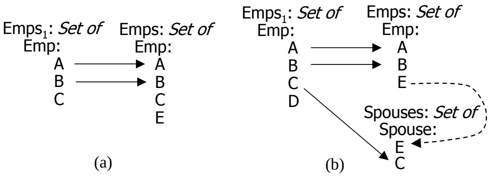

Figure 4: Creation of new values in the target.

into the target elements A and B, while C and E in the target are left unmapped. The s-t dependency:

```
for x  in Emps_1
exists y  in Emps
where y.Emp.A = x.Emp.A and y.Emp.B = x.Emp.B
```

does not specify any value for C and E either. However, to populate the target we need to decide on what values (if any) to use for these elements. A frequent case in practice is the one in which an unmapped element does not play a crucial role for the integrity of the target. For example, A and B could be employee name and salary, while C and E could be address and, respectively, date of birth. Creating a null value for either C or E is then sufficient. Or, if the unmapped element is optional in the target schema, then we can leave it out entirely. For example, if C is optional while E is not optional but nullable, we translate the s-t dependency as the following rule (with the obvious semantics):

```
for x  in Emps_1
let a = x.Emp.A,  b = x.Emp.B
return ⟨ Emp  =  ⟨ A = a,  B = b,  E =  null  ⟩ ⟩ in Emps
```

Creation of needed values However, E could be a key in the target relation, e.g. E could be employee id. The intention of the mapping would be in this case to copy employee data from the source and assign a new id for each employee, in the target. Thus a non-null value for E is needed for the integrity of the target.

A target element E *is* **needed** *if* E *is (part of) a key or foreign key or is both not nullable and not optional.*

For our example, we create a *different* but *unique* value for E, for *each* combination of the source values for A and B. Technically speaking, values for E are created by using a one-to-one (Skolem) function $f_{\mathtt{E}}[\mathtt{A},\mathtt{B}]$. The rule that translates the s-t dependency in this case is the same as the previous rule except that we use $f_{\mathtt{E}}[\mathtt{A},\mathtt{B}]$ instead of null for E.

Similarly, if C is a needed element in the target, it will be created by a different function $f_{\mathtt{C}}[\mathtt{A},\mathtt{B}]$. This function does not depend on the value of the *source* element C. Thus even if in the source there may exist two tuples with the same combination for A and B but with two different values for C (e.g. C is spouse, and an employee could be listed with two spouses), in the target there will be only one tuple for the given combination of A and B (with one, unknown, spouse). Thus, the semantics of the target is given solely by the values of the source elements that are *mapped*. Of course, a new correspondence from C to C will change the mapping: the employee with two spouses will appear twice in the target and E will be $f_{\mathtt{E}}[\mathtt{A},\mathtt{B},\mathtt{C}]$.

A similar mechanism for creation of new values in one target relation is adopted by the semantics of ILOG [7]. Next, we generalize this mechanism.

Generalization: multiple, correlated, sets of elements in the target The second frequent case that requires creation of non-null values is that of a foreign key. In Figure 4(b) the target element C is stored in a different location (the set Spouses) than that of elements A and B. However, the association between A, B values and C values is meant to be preserved by a foreign key constraint (E plays the role of a pointer in this case). Our semantic translation recognizes such situations by computing the logical relation L(A, B, E, C) that joins Emps and Spouses, and generating an s-t dependency $\mathtt{Emps}_1[\mathtt{ABC}] \subseteq \mathtt{L}[\mathtt{ABC}]$. However, this does not give a value for the needed element E. As in the case of a single target set (but this time with a logical relation instead), we assign a function $f_{\mathtt{E}}[\mathtt{A},\mathtt{B},\mathtt{C}]$ to create values for E. The generated rule is:

```
for x  in Emps_1
let a = x.Emp.A,  b = x.Emp.B,  c = x.Emp.C
return ⟨ Emp  =  ⟨ A = a,  B = b,  E = f_E[a, b, c] ⟩ ⟩ in Emps,
       ⟨ Spouse  =  ⟨ E = f_E[a, b, c],  C = c ⟩ ⟩ in Spouses
```

Briefly, the meaning of the rule is the following: for each (a, b, c) triple of values from the source create an element Emp in Emps and create an element Spouse in Spouses, with the given source values (a, b and c) in the corresponding places and with the *same* invented value $f_{\mathtt{E}}[a, b, c]$ in the two places where the element E occurs. If duplicate (a, b, c) triples occur in the source (maybe with different D values) only one element is generated in each of Emps and, respectively, Spouses. Thus, we eliminate duplicates in the target.

In general, when the level of nesting is one (i.e. flat case), the rule used to associate Skolem functions with needed elements is as follows (for the fully nested case we will make an adjustment in Section 3.3).

Skolemization for atomic elements: *Given an s-t dependency, each needed element in the target is computed by using a (different) one-to-one function that depends on all the mapped atomic elements of the target logical relation.*

The presence of functional dependencies in the target schema may change the functions used for value creation. To illustrate, if C is a key in Spouses (i.e., the functional dependency C → E holds) then in the above rule we replace $f_{\mathtt{E}}[a, b, c]$ with $f_{\mathtt{E}}[c]$.

3.2 Grouping of Nested Elements.

Consider Figure 5(a), in which the target schema is nested on two levels: elements A and C are at the top level, while a Bs element can have multiple B subelements (Bs is of set type). Elements A, B, and C of the source $\mathtt{Emps}_1$ are mapped, via correspondences, into the respective elements of the target Emps. The mapping, and the corresponding s-t dependency, requires all (A, B, C) values that can be found in the source to

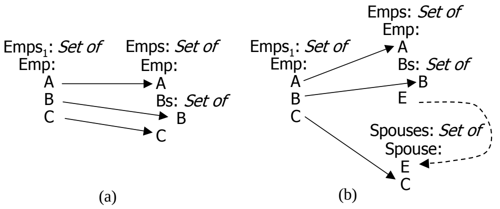

Figure 5: Grouping of elements in the target.

be moved to the target. However, *in addition to this*, the intended semantics requires all the different b values to be *grouped* together, for fixed values of A and C. For illustration, if A, B, and C are employee, child, and spouse names, respectively, the mapping requires the grouping of all children under the same set, if the employee and the spouse are the same. This behavior is not specified by an s-t dependency (which is stated at the level of flat logical relations). Thus, one of the tasks of data translation is providing the desired grouping in the target. Grouping is performed at every nesting level, as dictated by the target schema. The property that the resulting target instance will satisfy is a well-known one: Partitioned Normal Form (PNF) [1].

PNF: *In any target nested relation, there can not exist two distinct tuples that coincide on all the atomic elements (whether from source or created).*

To achieve this behavior, we use Skolemization as well. If a target element has set type, then its identifier (recall that each set is identified in the data model by a set id) is created via a Skolem function. This function *does not* depend on any of the atomic elements that are mapped *under* the respective set type, in the schema hierarchy. Instead it depends on all the atomic elements at the same level or above (up to the root of the schema). The same Skolem function (for a given set type of the target schema) is used across all s-t dependencies. Intuitively, we perform a deep union of all data in the target independently of their source. For the example of Figure 5(a), we create the rule:

```
for $x$ in $\texttt{Emps}_1$
let $a = x.\texttt{Emp.A}, b = x.\texttt{Emp.B}, c = x.\texttt{Emp.C}$
return $\langle$ Emp = $\langle$ A = $a$, Bs = $f_{\texttt{Bs}}[a, c]$, C = $c$ $\rangle$ $\rangle$ in Emps,
       $\langle$ B = $b$ $\rangle$ in $f_{\texttt{Bs}}[a, c]$
```

The meaning of the above rule is the following: for each (a, b, c) triple of values from the source, create first (if not already there) a sub-element Emp in Emps, with the appropriate A and C sub-elements, and with a Bs sub-element the value of which is the set id $f_{\texttt{Bs}}[a, c]$. Thus, the Skolem function $f_{\texttt{Bs}}$ is used here to create a set node. Also, we create (if not already there) a sub-element B, with value b, under $f_{\texttt{Bs}}[a, c]$. Later on, if a triple with the same A and C values (i.e. a, c) but different B value (b') is retrieved from the source, then we skip the first creation step (the required Emp sub-element already exists). However, the second part of the return clause applies, and we *append* a new B sub-element with value b' under the previously constructed set node $f_{\texttt{Bs}}[a, c]$. This mechanism achieves the desired grouping of B elements for fixed A and C values. A similar grouping mechanism can be expressed in XML-QL [5], using Skolem functions. It can also be implemented for languages that do not support Skolem functions like XQuery or XSLT as described in [17].

3.3 Value Creation Interacts with Grouping

To create a nested target instance in PNF, we need to refine the process of creation of new values, which was described in Section 3.2 subsection only for the non-nested case. We again explain our technique with an example. Consider Figure 5(b), where the elements A and C are stored in separate target sets. The association between A (e.g., employee name) and C (e.g., spouse name) is preserved via the foreign key E (spouse id). Thus, E is a needed element and must be created. However, in this case, it is rather intuitive that the value of E should not depend on the value of B but only on the A and C value. This, combined with the PNF requirement, means that all the B (child) values are grouped together if the employee and spouse names are the same. We achieve therefore the same effect that the mapping of Figure 5(a) achieves. In contrast, if E is created differently based on the different B values, then each child will end up in its own singleton set. In our implementation of the logical mapping, we choose the first alternative, because we believe that this is the more natural interpretation. Thus, we adjust the Skolemization scheme of Section 3.1 as follows.

The function used for creation of an atomic element E *does not depend on any of the atomic elements that occur at a lower level than* E*, in the target schema.*

For the example of Figure 5(b) we create the rule:

```
for $x$ in $\texttt{Emps}_1$
let $a = x.\texttt{Emp.A}, b = x.\texttt{Emp.B}, c = x.\texttt{Emp.C}$
return
  $\langle$ Emp = $\langle$ A = $a$, Bs = $f_{\texttt{Bs}}[a, c, f_{\texttt{E}}[a, c]]$, E = $f_{\texttt{E}}[a, c]$ $\rangle$ $\rangle$ in Emps,
  $\langle$ B = $b$ $\rangle$ in $f_{\texttt{Bs}}[a, c, f_{\texttt{E}}[a, c]]$,
  $\langle$ Spouse = $\langle$E = $f_{\texttt{E}}[a, c]$, C = $c$ $\rangle$ $\rangle$ in Spouses
```

As an extreme but instructive case, suppose that in Figure 5(b) we remove the correspondences into A and C, but keep the one into B. If A and C are needed, then they will be created by Skolem functions with no arguments. This is the same as saying that they will be assigned some constant values. Consequently, the two target sets Emps and Spouses will each contain a single element: some unknown employee, and some unknown spouse, respectively. In contrast, the nested set Bs will contain all the B values (all the children listed in Emps$_1$). Thus, the top-level part of the schema plays only a structural role: it is *minimally* created in order to satisfy the schema requirements but the respective values are irrelevant. But this is fine, since there is nothing mapped into it. Later on, as correspondences may be added to A and C, the children will be separated into different sets, depending on the mapped values.

The Skolemization algorithm sketched in this section is a polynomial-time, graph-walking algorithm. It is described in full detail in [17], which also describes the translation from rules to XQuery and XSLT.

4 Experience

We have implemented our solutions in our prototype system *Clio*. Our experience has shown that the intended semantics of the correspondences are captured by the system, and the user effort in creating (and debugging) the translation queries is significantly less than with other approaches (including manually writing the query or using other query building tools). Although there is no standard benchmark or evaluation methodology for the subjective task of integration, we attempt to provide some evidence for the performance and effectiveness of our tool by discussing our usage of Clio on several schemas of different sizes and complexity. Our test schemas are listed in Table 1 with pairs of source and target schemas listed consecutively. They include: two XML Schemas for the DBLP bibliography (the first obtained from the DBLP Web Site); the relational `TPC-H` schema and a nested XML view of this benchmark; two relational schemas from the `Amalgam` integration suite for bibliographic data; the relational and DTD version of the `Mondial` database; and two schemas representing a variety of gene expression (*microarray*) experimental results. We have made all these schemas available on our web page (`www.cs.toronto.edu/db/clio`), which also contains links to their original sources.

We included the DBLP schemas as examples of schemas with few constraints. These schemas still differ semantically, only the semantics is encoded primarily in the (different) nesting structure of the schemas, rather than through constraints. The `Amalgam` relational schemas, on the other hand, are examples of schemas without any nesting structure where all the semantics are captured by a rich set of referential constraints. We have also included real XML schemas with relatively few constraints (GeneX) and with many constraints (Mondial). Table 1 shows some characteristics of all these schemas in our internal nested relational representation. The nesting depth indicates the nesting of repeated elements (set types). There is often more nesting through record types, but this does not affect the efficiency of our algorithms.

The load and compile time indicate, respectively, the time to read the schemas and the time to compile the schemas into our internal logical representation. The compile time includes the time to understand the nested structure (by computing primary paths) and to combine structures linked by constraints (using the chase to compute logical relations). While the load time, as expected, closely reflects the schema size (including both the schema structure and constraints), the compile time is mostly affected by the number of NRIs. On schemas with few or no NRIs, the compile time is almost negligible while schemas with a large number of NRIs take more time to compile. Although compile time can be as large as several seconds (in our unoptimized prototype) for schemas with many constraints, we found this to be an acceptable delay for Clio users. Recall that compilation occurs only *once* when a schema is loaded.

Schemas	Nest. Depth	Total Nodes	Leaf Nodes	NRIs	Load Time	Compile Time
DBLP$_1$ (XML)	2	88	52	0	0.52	0.19
DBLP$_2$ (XML)	4	27	12	1	0.15	0.15
TPC-H (RDB)	1	51	34	9	0.21	0.44
TPC-H (XML)	3	19	10	1	0.03	0.02
GeneX (RDB)	1	84	65	9	0.11	0.72
GeneX (XML)	3	88	63	3	0.13	0.50
Mondial (RDB)	1	159	102	15	0.58	5.41
Mondial (XML)	4	144	90	21	0.57	3.68
Amalgam$_2$(RDB)	1	108	53	26	0.59	6.37
Amalgam$_1$(RDB)	1	132	101	14	0.47	1.85

Table 1: Test Schemas Characteristics (time is in sec)

Schema	With NRIs	No NRIs	Subsumed
DBLP	13	11	2
TPC-H	14	7	3
GeneX	4	4	1
Mondial	5	4	2
Amalgam	10	9	8

Table 2: Source-to-target dependencies generated with and without NRIs in the schemas

To evaluate the results of our semantic translation, we sought to understand whether our algorithms were producing the right results and whether they were doing so in an effective way. Table 2 shows the total number of source-to-target dependencies that Clio generates for our test schemas (column labeled "With NRIs"). For comparison, we have also included the number of s-t dependencies that would be produced if we ignored all (intra-schema) NRIs (column labeled "No NRIs"). These dependencies preserve the semantics of nesting but ignore any referential semantics embedded in schema constraints. Although schema constraints may increase the number of possible s-t dependencies, we find that in practice the number of s-t dependencies remains manageable, and users do not get overwhelmed with too many choices. Furthermore, the last column (labeled "Subsumed") shows how many of the dependencies generated without NRIs were subsumed by better, association-preserving dependencies when NRIs were used. For instance, in the case of the `expenseDB/statDB` mapping, the number of dependencies created with and without dependencies is the same: five. Three of the dependencies created without NRIs appeared again, unchanged, when NRIs were used. The other two dependencies were subsumed by dependencies that embraced the available NRIs. We, thus, end up with a better quality mapping that maintains the implied relationships in the schemas.

For all the schemas, the user was able to select a subset of the generated s-t dependencies to form the correct (desired) translation. In some cases (e.g., `DBLP`, `Amalgam`, `expenseDB`), the intended transformation required all the created dependencies, in other cases it required only a strict subset. Clio's DataViewer [19] has proven to be very effective in helping users understand and select the desired s-t dependencies.

5 Discussion and Future Work

We have presented new algorithms for schema mapping and data translation between nested schemas with nested referential constraints. Our solutions are distinguished in that we take advantage of schema semantics to generate all consistent translations. We guarantee that the target instance we produce satisfies the target structure and constraints, even when such an instance must contain data that are not derived from the source. The logical mappings we produce are source-to-target dependencies relating a source query (specifying how source data may be collected together) and a target query (specifying how this data is restructured in the target). Such dependencies may be used in a data integration scenario where the target is virtual and queries on the target are answered using the source instance [8]. They may also be used to materialize a target instance. Our data translation algorithm gives one such implementation of the dependencies which creates a target instance in PNF.

Our queries make use of the rich restructuring capabilities and id-invention of the translation languages that precede this work [7, 4]. Yet, we differ in that we automatically generate translation queries in a way that respects the semantics encoded in the schemas. In this respect, our work is similar to that of TranScm [13], one of the first techniques to use schema information to help automate data translation. TranScm uses predefined matching rules that describe commonly used transformations to derive a translation. Hence, their technique is not as flexible and general as ours. The relationship of our work to other integration approaches and to work on logical data independence [11] is explored in greater detail in the full version of this paper [17].

Our motivation was to provide a tool that quickly generates correct translations between rich Web data sources. Hence, we have focused on building a robust tool that captures the semantics embedded in nested structures. We have currently extended our algorithms to consider additional types from XML Schema, including union types, and we are planning to include order in our framework. To support order, we would need to include list types and enhance the generated mappings with ordering predicates.

Many thanks to Peter Buneman, Howard Ho, Phokion G. Kolaitis, Felix Naumann, Val Tannen and the anonymous reviewers for helpful suggestions.

References

[1] S. Abiteboul and N. Bidoit. Non-first Normal Form Relations: An Algebra Allowing Data Restructuring. *JCSS*, 33:361–393, Dec. 1986.

[2] M. A. Casanova, R. Fagin, and C. H. Papadimitriou. Inclusion Dependencies and their Interaction with Functional Dependencies. *JCSS*, 28(1):29–59, Feb. 1984.

[3] 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.

[4] S. Davidson and A. Kosky. WOL: A Language for Database Transformations and Constraints. In *ICDE*, pages 55–66, 1997.

[5] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. In *WWW8*, pages 77–91, 1999.

[6] H.-H. Do and E. Rahm. COMA - A System for Flexible Combination of Schema Matching Approaches. In *VLDB*, 2002.

[7] R. Hull and M. Yoshikawa. ILOG: Declarative Creation and Manipulation of Object Identifiers. In *VLDB*, pages 455–468, 1990.

[8] M. Lenzerini. Data Integration: A Theoretical Perspective. In *PODS*, pages 233–246, 2002.

[9] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering Queries Using Views. In *PODS*, pages 95–104, 1995.

[10] D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing implications of data dependencies. *ACM TODS*, 4(4):455–469, 1979.

[11] 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.

[12] R. J. Miller, L. M. Haas, and M. Hernández. Schema Mapping as Query Discovery. In *VLDB*, pages 77–88, 2000.

[13] T. Milo and S. Zohar. Using Schema Matching to Simplify Heterogeneous Data Translation. In *VLDB*, pages 122–133, 1998.

[14] F. Naumann, C. T. Ho, X. Tian, L. M. Haas, and N. Megiddo. Attribute Classification Using Feature Analysis. In *ICDE*, 2002. (Poster).

[15] L. Popa, M. A. Hernández, Y. Velegrakis, R. J. Miller, F. Naumann, and H. Ho. Mapping XML and Relational Schemas with CLIO, *Demo*. In *ICDE*, 2002.

[16] L. Popa and V. Tannen. An Equational Chase for Path-Conjunctive Queries, Constraints, and Views. In *ICDT*, pages 39–57, 1999.

[17] L. Popa, Y. Velegrakis, R. J. Miller, M. Hernández, and R. Fagin. Translating Web Data. Technical Report CSRG-441, U. of Toronto, Dept. of CS, Feb. 2002.

[18] E. Rahm and P. A. Bernstein. On Matching Schemas Automatically. *VLDB Journal*, 10(4), Dec. 2001.

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