

# Mapping Generation and Data Translation of Heterogeneous Web Data.\*

L. Popa<sup>1</sup>    Y. Velegakis<sup>2</sup>    R. J. Miller<sup>2</sup>    M. Hernández<sup>1</sup>    R. Fagin<sup>1</sup>

<sup>1</sup>IBM Almaden Research Center  
650 Harry Road  
San Jose, CA 95120

<sup>2</sup>Department of Computer Science  
University of Toronto  
6 King's College Road  
Toronto, ON M5S 3H5

## Abstract

Mapping schemas and translating data stored in different formats continues to be an important problem in modern information systems and e-commerce applications. In this paper, we present a novel framework for translating Web data. We describe a system in which high-level mappings between two schemas are translated to semantically meaningful queries that transform data conforming to the first schema into a representation that conforms to the target schema. The queries have two important properties: 1) they generate data that is guaranteed not to violate any of the constraints in the target schema, and 2) they generate values for required and not nullable parts of the target schema for which the high-level mapping does not specify how to populate. Our approach is unique in that 1) we consider **both** popular kinds of data that can be found on the Web (relational and semi-structured), hence we can translate relational data to XML and vice versa; 2) we generate the **complete** set of mappings that are consistent with the high-level specifications, capturing that way the intended semantics of the user; and 3) we take into consideration schema constraints not only in the source but also in the **target**.

We present our mapping algorithm and explain the architecture of **Clio**, a high-level schema mapping tool that implements our framework. We describe our experience with mapping real-world web data in Clio, and sketch our current research directions.

## 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 (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 elements of a source schema to elements of

---

\*The current paper is an extended abstract of [PVM<sup>+</sup>02]

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 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 elements involved. Therefore, even users 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 elements 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 which is nothing more than a query on the source schema that generates data to populate the target schema.

The efficacy of supporting element-to-element 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 elements of the two schemas [DDH01, MBR01, RB01] (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.

Our mapping generation algorithm works in two phases. In the first phase, the *semantic translation*, the set of 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 elements into nested tables and sets). By exploiting the structure of the schemas we can identify and group the correspondences into a number of (possibly overlapped) sets. Each such set generates a semantically meaningful mapping. Our algorithm is *complete* [PVM<sup>+</sup>02] in the sense that it can identify **all** the semantically meaningful mappings for a given set of correspondences.

The second phase translates the logical mapping into a query that can be executed over the source schemas and populate the target schema. To this end, target element values may need to be invented for parts of the target schema for which there is no correspondence to the source to determine how those parts should be populated. These values ensure that the data respects the constraints (including nested referential constraints) and the (possibly nested) structure of the target schema.

Although we do not specifically address this in this paper, the generated queries can be used as virtual mappings between the two schemas. This means that the target schema is not intended to be materialized but rather used as an interface of the source schema.

An important feature of our approach is that it is based on a nested relational data model, hence, we can translate relational data to XML, XML to relational, relational to relational, and XML to XML. Furthermore, we can represent key and foreign key constraints, even *nested constraints* as found in XML-Schemas [Fal01].

We have implemented our algorithm in the Clio [PHV<sup>+</sup>02] system. Clio graphically presents the users with the source and the target schema and allows them to draw the correspondences. Clio generates all the alternative mappings and let the users browse them and decide those that represent the indented semantics.

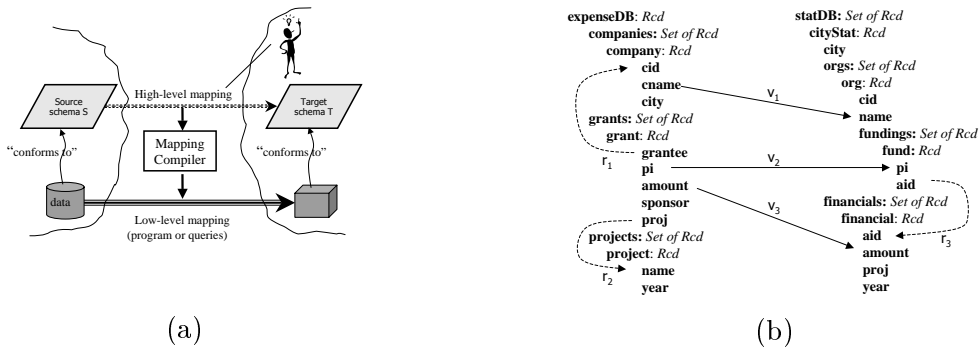


Figure 1: (a) A high-level schema mapping tool. (b) A schema mapping scenario.

## 2 Semantic Translation

**Example 2.1** Consider the two schemas in Figure 1(b). For convenience, both schemas are shown in a common nested relational representation<sup>1</sup>. 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 elements 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<sup>+</sup>02]. ■

The first step of our approach is the *semantic translation*, in which we generate an intermediate *logical mapping* representation that is a precise and faithful understanding of the high-level mapping. Since correspondences specify only element-to-element 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 2.2** 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

<sup>1</sup>Which is in fact adopted by our mapping compiler and used throughout this paper.

$$\begin{array}{ll}
S_1 = \{ c \in \text{expenseDB.companies}; \} & T_1 = \{ c \in \text{statDB} \} \\
S_2 = \{ g \in \text{expenseDB.grants}; \} & T_2 = \{ c \in \text{statDB}, o \in c.\text{cityStat.orgs} \} \\
S_3 = \{ p \in \text{expenseDB.projects}; \} & T_3 = \{ c \in \text{statDB}, o \in c.\text{cityStat.orgs}, \\
& \quad f \in o.\text{org.fundings} \} \\
& T_4 = \{ c \in \text{statDB}, f' \in c.\text{cityStat.financials} \}
\end{array}
\begin{array}{l}
\text{(a)} \\
\text{(b)}
\end{array}$$

Figure 2: Primary paths for: (a): relational schema `expenseDB`, (b): nested schema `statDB`.

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 its `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. In the first step, we have to enumerate the sets of logically associated elements in each schema. Associated elements are identified by using two basic forms of relationships that exist in relational and nested schemas: parent/child and foreign key/key. We first identify all the sets of elements related through the schema structure without any schema constraints. We refer to these sets as *primary paths*. There is one primary path for each *set* type element in our nested relational representation of the schemas. For a relational schema, each primary path corresponds to a relation. This was expected since without any schema constraints, the only way attributes are associated in a relational schema is through the relations.

**Example 2.3** Figure 2 indicates the primary paths for the two schemas of our running example. Primary paths are represented as nested queries (or nested tableaux) on the nested relational representation that return record elements<sup>2</sup>. Intuitively, for the `statDB` schema,  $T_1$  denotes the set of cities (`cityStat` records) that may or may not have organizations (`orgs`) or financial data (`financial`).  $T_2$  denotes `cityStat` records that may have `financials` but do have organizations (`orgs`) that may or may not have `fundings` (`fundings`). ■

In order to identify the complete sets of semantically associated elements we use an extension of the classical relational chase [BV84]. We chase each primary paths with the foreign key constraints of the schema. Relational and XML schema constraints are uniformly represented using an extension of the relational data dependencies [FV86]. The result of the chasing is a *logical relation*. For example, if a logical path includes a relation that has a (non-nullable) foreign key into another relation then both relations are included in the same *logical relation*. A *logical relation* is a union of primary paths with a number of equality conditions on their atomic elements.

<sup>2</sup>For more details of our representation please refer to the full version of this paper [PVM<sup>+</sup>02]

**Example 2.4** 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 logical relation between them is discovered if the additional foreign key constraint on `sponsor` is given. Figure 3 indicates the logical relations that are generated by chasing the primary paths of Figure 2 with the 3 foreign key constraints of Figure 1. ■

The second step of semantic translation generates all possible ways of mapping source relations to target relations based on an given set of source-to-target correspondences. The result is a *set* of logical mappings. Each logical mapping is a meaningful and *different* mapping. Enumeration of all such mappings is an essential ingredient of our approach. As we have seen in Example 2.2, any *one*, *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 the subset of them that corresponds to his/her intended semantics.

To generate that *complete* set of logical mappings we enumerate all the possible pairs  $\langle S, T \rangle$  where  $S$  and  $T$  are source and target logical relations correspondingly. For each such pair, we find the set of correspondences that use only elements of  $S$  and  $T$ . If that set is found to be not empty, then a logical mapping is generated by combining  $S$ ,  $T$  and the set of correspondences that were found. 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 logical relation correspond to data instances of a target logical relation. As an internal representation of the mapping, dependencies are simpler than queries and therefore easier to manipulate and optimize.

**Example 2.5** Combining  $S_2$ ,  $T_3$  and the 3 value correspondences of our running example we get the following logical mapping:

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

Note how the last equality guarantees the satisfaction of the foreign key constraint on the `aid` in the target. ■

It has been proved [PVM<sup>+</sup>02] that our algorithm is *correct*, i.e., the generated mappings respect the constraints of the schemas, and *complete*, i.e., generates *all* the mappings. It has also been designed to work for a large class of nested schemas and their constraints. However, we do not allow for cyclic schemas, i.e. schemas that have: 1) recursive type definitions (as they may occur in XML schemas or DTDs), or 2) cyclic integrity constraints. Cyclic schemas give rise in general to infinitely many semantic associations and, consequently, infinitely many relationship-to-relationship mappings. Furthermore, the algorithm generates *only* mappings between (any) one logical relation in the source and (any) one logical relation in the target. We believe that logical relations (as implied by the constraints of a schema) are the fundamental relationships in a schema, hence, we 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 (which 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.

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

Figure 3: All the logical relations for the relational `expenseDB` (top) and the nested `statDB` (bottom) schema.

### 3 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) by using reach restructuring constructs (resembling ILOG [HY90], WOL [DK97], and XML-QL [DFF<sup>+</sup>99]). For each logical mapping, the generated query has two roles: *retrieve* and *insert*. First it retrieves the data instances from the source by performing the required join and unnest operations (i.e. following the source logical paths). 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 and is similar to ILOG [HY90].

**Example 3.1** *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: [pi = Gerstner, cid = IBM, proj = DB2, sponsor = NSF, amount = \$100K] and [pi = Gates, cid = MSFT, proj = SQLServer, sponsor = NSF, amount = \$200K]. When the first tuple is mapped to the target, it will generate the fund tuple [pi = Gerstner, aid = G<sub>1</sub>] and the financial tuple [aid = G<sub>1</sub>, amount = \$100K, proj = null, year = null]. Similarly for the second grant tuple, the mapping will generate the fund tuple [pi = Gates, aid = G<sub>2</sub>] and the financial tuple [aid = G<sub>2</sub>, amount = \$200K, proj = null, year = null]. If the generated value G<sub>2</sub> was not different than G<sub>1</sub>, 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. ■

In order to generate values for those parts of the target schema we use Skolem functions. We have developed an algorithm that determines the arguments of the Skolem functions. The algorithm is design such that the generated data satisfy a well-known design principle: Partition Normal Form (PNF) [AB86]: *In any target nested relation, there can not exist two distinct tuples that coincide on all the atomic elements (whether from source or created).* For our running example, this means that an organization in the target will be identified by the name of the company. As a consequence, if two companies have the same name, they will both be mapped on the same organization in the target. This was expected since the only information that was provided by the user was how to generate the name of the organization. If there was one extra correspondence from the cid of the company to the cid of the org then two different companies with the same name would have been mapped to different organizations in the target.

The value determination algorithm is as follows: First we construct the *query graph* associated with the mapping. The query graph is a canonical instance, over the target schema, that describes the right-hand side of the mapping. Figure 4 denotes the query graph for the mapping in Example 2.5. We annotate each node of the graph for which there is a correspondence to the source to determine its value, with the source element name. We call those nodes *fixed* nodes. Fixed nodes are indicates in the figure with a double circle. For each fixed node, we propagat its value to the remaining nodes, by applying the following rule: If a node  $n$  is annotated with a value  $x$  then:

1. its parent is also annotated with value  $x$  unless its parent represent a set type.
2. each of its children that are not *fixed* nodes, is also annotated with value  $x$ .
3. each node that is associated with  $n$  through an equality in the mapping, is also annotated with value  $x$ .

The result of the above process is an annotated query graph as depicted in Figure 4. The values our algorithm generates for the undetermined parts is a Skolem function that has as arguments the annotations of its corresponding node in the query graph.

**Example 3.2** *The aid values that will be generated in the target schema will be a Skolem function that depends on the company cname, the pi, and the amount of the source schema. This means that for each different triplet (cname, pi, amount), a different aid will be generated in the target.* ■

The value-generation algorithm has the important property of preserving the information of the source data by not generating any extra facts in the target.

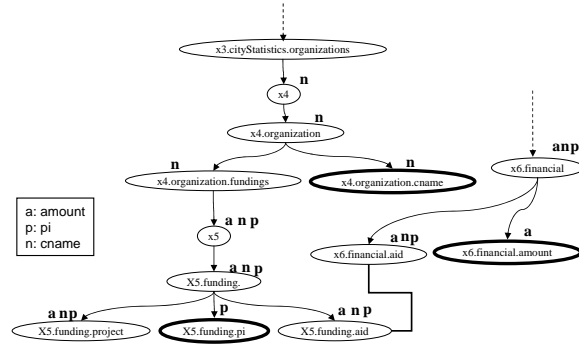
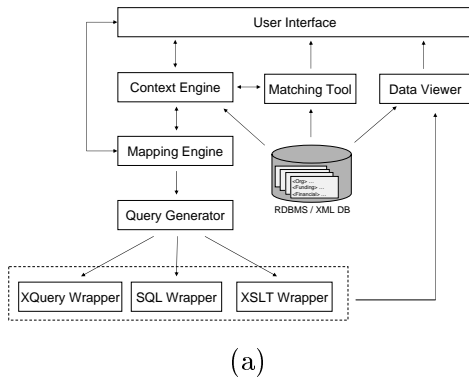
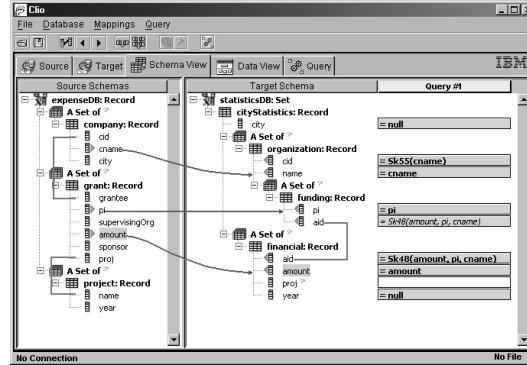


Figure 4: An annotated query graph



(a)



(b)

Figure 5: The general architecture and the GUI of Clío

The query graph can be seen as an abstract query language that describes the mapping. We can translate the annotated query graph to an actual query that can be executed on the source schema database and generate data for the target. Based on the kind of the source data, we can generate SQL queries (if relational), or XQueries and XSLT (if XML). For elements of the target schema that are optional, if there is no correspondence for them we generate no value. If they are not optional but can be null, then we generate a null value. If, however, they are not optional and not nullable, then we generate a value based on the Skolem function of the annotated graph.

## 4 The CLIO System

We have implemented our solutions in a prototype system called *Clío*. Its architecture is depicted in Figure 5. Due to the dynamicity of standards and technologies and in order to facilitate product evolution, Clío avoids the monolithic application design and adopts a modular architecture. In principle, modules could be substituted or upgraded without affecting the overall operation of the system. One of our major requirements was to make the whole process incremental. Often, users will provide value correspondences incrementally and want to see partial results before adding additional correspondences. Thus, modules should avoid redundant re-computation and redundant exchange of data. Furthermore, Clío adheres to well-established open and commercial standards like Java, DOM, XML, XSLT and SQL, in order to capitalize on existing software, increase interoperability and reduce development and maintenance time.

**Graphical User Interface:** The user interacts with Clio through a graphical user interface. Visual tools present the complete structure of the schemas and the schema constraints and let the user draw the correspondences between them. The mappings are computed and are presented either as (SQL, XSLT, or XQuery) queries, or as graphical annotations on the target schema. The user has the ability to browse the set of the generated mappings and select only those that describe the intended semantics of the drawn correspondences.

**Context Engine:** The heart of Clio is the *context engine* that manipulates and interprets the *context*, an internal representation of the input schemas and value correspondences. It is in charge of loading the schemas and translating them into the nested relational representation. In the sequel, it detects all the primary paths, chase them and generates the logical relations. This computation takes place only once at schema load time and if the user makes any changes to the source or target schema and constraints, the structures are incrementally updated.

**Mapping Engine:** The *mapping engine* takes the *context* from the context engine and generates all the possible source to target *logical mappings*. Mappings are kept in internal structures that are incrementally updated when a new value correspondence is added/deleted.

**Query Engine:** The *query engine* is in charge of the *data generation* phase described in Section 3. The generated queries are represented in an internal structure that is independent of the final query language that will be used. This allows us to modify, extend or optimize the data generation algorithm without affecting any other part of the schema.

**Pluggable query wrappers:** Clio is equipped with a number of pluggable *query wrappers* that translate the internal representation of the queries in the query engine to actual queries that will be executed on the source data. Currently, the system supports XQuery, XSLT, and SQL wrappers. Support of other query/transformation languages or even a programming language like Java can be easily added by simply building and plugging the appropriate wrappers.

**Data Viewer:** An alternative way of presenting the user the results of a mapping is through the data. The *data viewer* [YMHF01] is a visual tool that carefully select data examples for a mapping and presents them for inspection to the user. Those sample data examples are used to help the user understand the effects of each mapping (and the differences between alternative mappings).

**Attribute Matcher:** To help the user in specifying the correspondences, an attribute matching tool [NHT<sup>+</sup>02] can automatically suggest likely correspondences based on name and structural similarity, or by mining the data if the source and target databases are both populated. Other techniques like those surveyed in [RB01] can also be implemented in this module.

A typical session in Clio starts with the user loading a source and a target schema into the system. These schemas are read from either an underlying relational database (using JDBC) or from an XML store (with schemas represented as XML Schemas or DTDs). Correspondences are then drawn on the schemas and the complete set of mappings that respects the drawn correspondences is computed. However, since this number may be big, we use some heuristic to detect the most likely mapping and present it to the user. The user can browse through the other mappings and decides which of those describe his/her intended semantics. He/She can also see the complete SQL, XSLT, or XQuery query that Clio generates and make modifications if needed. Alternatively, he/she can use the data viewer to get an idea of the generated mappings. When he/she is satisfied, the generated queries can be sent for execution.

Schemas	Nesting Depth	Total Nodes	#f Constr.	Load Time	Compile Time
expenseDB(RDB)	1	17	2	0.03	0.03
statsDB (XML)	3	17	1	0.03	0.04
DBLP <sub>1</sub> (XML)	2	88	0	0.52	0.19
DBLP <sub>2</sub> (XML)	4	27	1	0.15	0.15
TPC-H (RDB)	1	51	9	0.21	0.44
TPC-H (XML)	3	19	1	0.03	0.02
GeneX (RDB)	1	84	9	0.11	0.72
GeneX (XML)	3	88	3	0.13	0.50
Mondial (RDB)	1	159	15	0.58	5.41
Mondial (XML)	4	144	21	0.57	3.68
Amalgam <sub>2</sub> (RDB)	1	108	26	0.59	6.37
Amalgam <sub>1</sub> (RDB)	1	132	14	0.47	1.85

Table 1: Test Schemas Characteristics

## 5 Experiments

We have tested our system with a number of real-world schemas of different size and complexity. Our experience has shown that with Clio the user effort and time in generating a mapping between two schemas is significantly less than manually writing the queries. The ability to interact with the system, get immediate feedback of the generated mappings and browse them helped the users in better understanding of the translation and led them to create queries that better describe the semantics of the translation. Users were also impressed how easily they could get a nine-way join query right by simply drawing a few lines (the correspondences) between the schemas and by the fact that they didn't have to learn SQL or the complicated XQuery. Furthermore, the fact that the target schema constraints were playing an important role in our computation process eliminated the risk of generating queries that could not be used to populate the target because the generated data was violating the constraints of the target database.

We tested our schema in a Intel Pentium III, running at 1.1GHz with 512MB of RAM and we measured its performance. Table 1 indicates the characteristics of the our various test schemas with pairs of source and target listed consecutively, as well as their loading performance. The schemas we used varied in terms of complexity, nesting depth, size, and number of schema constraints. We tested both relational and nested schemas (DTDs, or XML Schemas).

The result showed that Clio can load even large and complicated schemas in less than a second. Loading times is the time needed to import a relational or XML schema (this includes communication with the database or accessing the file on the disk), and translating it into the nested relational representation. The *compile* time is heavily affected by the nesting depth of the schemas and the number of constraints. The compile time is the time needed to find the primary paths of the schemas and chase them. For really large schemas with many constraints this time may reach the 5 or 6 seconds, however, this is happening only once at the beginning of the process, thus, this time is not a critical issue. When the user draws correspondences the mappings are computed incrementally and presented to the user. We have never witnessed a case in which the mapping computation takes more than a fraction of a second.

The experiments have also shown that the use of schema constraints in the mapping computation increases the number of generated queries. However, this number remains manageable and the users do not get overwhelmed with too many choices. Furthermore, in most of the cases, all the queries generated by Clio were needed in order to have a complete and information preserving mapping which is an indication of how successful our approach is.

## 6 Conclusion and Current Work

We have presented Clio, a research prototype developed in IBM Almaden Research center in collaboration with the University of Toronto. Clio uses a new technique that tries to infer the complete set of mappings (queries) that populate a target schema with data from an instance of a source schema. The approach used by Clio is unique in that it takes into consideration constraints on the target schema and generates values whenever needed to guarantee the correctness and the quality of the translation.

We have envisioned a number of extension for our schema mapping management tool and we are currently working on them. Ultimately, we view Clio as an extensible mapping management platform on which mappings and schemas can be stored, queried, and maintained the same way regular data do. We are designing a query facility that allows users to ask queries about mappings, schema elements, and their properties. A user, for example, may want to update the source schema of our example by deleting the `cid` element in the `grant`, and he/she would be interested to know what are the mappings that will be affected by that action. Normally, he could browse the list of mappings and reason about them, however, as schemas get larger and mappings become more complicated, such a solution is not feasible anymore. A query facility would greatly facilitate that process. Furthermore, instead of manually updating the mappings after such changes in the schema, an automatic mechanism could have been responsible for keeping those mappings consistent. We are interested in such a mechanism and we are investigating various methods for implementing it.

Another direction we are currently working on is the *schema evolution*. Clio can be used as a schema evolution management tool in which multiple (evolved) versions of the same schema along with their data instances are stored. Mappings between versions specify how an instance in one version can be expressed as an instance of another version. Clio will be able to provide the capability of querying all the data, independently of the version of the schema they conform to.

## References

- [AB86] S. Abiteboul and N. Bidoit. Non-first Normal Form Relations: An Algebra Allowing Data Restructuring. *JCSS*, 33:361–393, December 1986.
- [BV84] C. Beeri and M. Y. Vardi. A Proof Procedure for Data Dependencies. *Journal of the ACM*, 31(4):718–741, 1984.
- [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<sup>+</sup>99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. In *WWW8*, pages 77–91, 1999.
- [DK97] S. Davidson and A. Kosky. WOL: A Language for Database Transformations and Constraints. In *ICDE*, pages 55–66, 1997.
- [Fal01] D. C. Fallside. XML Schema Part 0: Primer. W3C Recommendation, May 2001. <http://www.w3.org/TR/xmlschema-0>.
- [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.
- [HY90] R. Hull and M. Yoshikawa. ILOG: Declarative Creation and Manipulation of Object Identifiers. In *VLDB*, pages 455–468, 1990.
- [MBR01] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. In *VLDB*, 2001.

- [NHT<sup>+</sup>02] F. Naumann, C. T. Ho, X. Tian, L. M. Haas, and N. Megiddo. Attribute Classification Using Feature Analysis. In *ICDE*, 2002. (Poster).
- [PHV<sup>+</sup>02] L. Popa, M. A. Hernández, Y. Velegrakis, R. J. Miller, F. Naumann, and H. Ho. Mapping XML and Relational Schemas with CLIO, *System Demonstration*. In *ICDE*, 2002.
- [PVM<sup>+</sup>02] 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, February 2002. <ftp://ftp.cs.toronto.edu/cs/ftp/pub/reports/csri/441>.
- [RB01] E. Rahm and P. A. Bernstein. On Matching Schemas Automatically. *VLDB Journal*, 2001.
- [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.