Adapting Mappings in Frequently Changing Environments

Yannis Velegrakis[†] Renée J. Miller[†] Lucian Popa[‡]

CSRI - Technical Report TR468
Department of Computer Science
University of Toronto

Abstract

To achieve interoperability, modern information systems and e-commerce applications use mappings to translate data from one representation to another. In dynamic environments like the Web, data sources may change not only their data but also their schemas, their semantics, and their query capabilities. Such changes must be reflected in the mappings. Mappings left inconsistent by a schema change have to be detected and updated. As large, complicated schemas become more prevalent, and as data is reused in more and more applications, manually maintaining mappings (even simple mappings like view definitions) is becoming impractical. We present a novel framework and tool for automatically adapting mappings as schemas evolve. Our approach considers not only local changes to a schema, but also changes that may affect and transform many components of a schema. We consider a comprehensive class of mappings for relational and XML schemas with choice types and (nested) constraints. Our algorithm detects mappings affected by a structural or constraint change and generates all the rewritings that are consistent with the semantics of the mapped schemas. Our approach explicitly models mapping choices made by a user and maintains these choices, whenever possible, as the schemas and mappings evolve. We describe an implementation of a mapping management and adaptation tool based on these ideas and present a case study using our tool to manage mappings used in a physical DB design tool.

1 Introduction

A broad variety of data is available in distinct heterogeneous sources, stored under different formats: database formats (in relational and object-oriented models), document formats (SGML/XML), browser formats (HTML), message formats (EDI), etc. The integration, transformation, and translation of such data is increasingly important for modern information systems and e-commerce applications. Views, and more generally, transformation specifications or *mappings*, provide the foundation for many data transformation applications.

A mapping specifies how data instances of one schema correspond to data instances of another. Mappings are often specified in a declarative, data-independent way (for example, as queries or view definitions). However, they necessarily depend on the schemas they relate. When these schemas change, the mappings must be updated or adapted to the new schemas. In this work, we consider the *adaptation* and management of mappings as schemas evolve.

To motivate our work, we first consider a number of applications and environments in which mappings are used extensively. Our discussion here highlights not only the ubiquity of mappings in modern data management tasks, but also the considerable effort that must be put into defining and verifying mappings and their semantics. We will argue that we can ill effort to recreate mappings as schemas change, but should instead reuse previous mappings. Furthermore, mapping creation, although aided tremendously by modern tools that suggest (syntactic) schema matches [RB01] and full (semantic) mappings [PVM$^+$02b], still requires input from human experts. It is the semantic decisions input by these experts that we will especially try to manage and preserve in order to save the most precious administrative resource, human time.

Data Integration. In data integration, a unified, virtual, view is used to query a set of heterogeneous data sources [Len02]. The process of creating this view is called schema (or view) integration [BLN86]. Numerous algorithms and tools have been proposed to automate or semi-automate schema integration [BLN86, SP94][and others]. However, at its core, schema integration is a schema design problem. Some integration choices will necessarily be subjective and different users or designers may wish to make different choices or alter a heuristic choice made by a tool. Some tools anticipate this and for a limited set of alternative designs, will still produce a correct mapping between the source schemas and the selected integrated schema [SP94]. Others will permit users to use a set of composable schema transformation operators to produce an integrated (transformed) schema (with a composed mapping) [GLS95]. However, these approaches in general do not permit arbitrary changes to the integrated schema. Even a simple horizontal decomposition of an integrated table based on a user-defined predicate will typically require the designer to manually edit the mapping. Furthermore, changes in the source schema (even modest ones) are not supported. Such changes require the schema integration algorithm to be rerun [RR99].

Modeling Source Descriptions. An important and influential proposal in data integration was to define a collection of heterogeneous data sources as views over an integrated schema [LRO96]. This technique (now known as local-as-view or LAV) proved to provide considerable flexibility and data independence for applications on the integrated data (for example, data warehousing applications). LAV can be contrasted with classical data integration where the integrated (global) schema was modeled as a view over the sources (global-as-view or GAV). More recently, a combined GLAV (global and local-as-view) approach has been studied [FLM99]. Consider a LAV system in which a source schema has been de-normalized (perhaps tables R and S are joined to form a combined table T). The LAV mappings that define R and S must be combined. It seems clear that there is a systematic, natural way to do this, but to the best of our knowledge, no tools exist for automatically updating such mappings as schemas evolve.

Schema Mapping. Schema mapping tools such as Clio [**?**] help users to create mappings between pairs of schemas. Clio uses the semantics embedded in the schemas (and in the data) to suggest possible mappings to a user. However, a user may override these choices. This happens when a desired semantic relationship is not represented explicitly in a schema. Consider a target schema representing information about employees and companies. The source schema may have information about employees, their spouses, companies they work in, and stocks they own. To map these schemas, Clio may suggest that employees appear in the target with companies in which they work. However, a user may have a very different semantic relationship in mind (perhaps employees together with the companies in which either they or their spouse have invested more money than they earn in a year). Clio will not find such a relationship but will let a user manually modify a mapping. As the schemas evolve, we would want to be able to reuse this semantic input from a user. So if an address field is added to the target, rather than using Clio to suggest that a target employee be related to the address of the company she works in, we would like to take this decision based on the previous existing mappings.

Data Exchange. In data exchange, mappings are used to transform a source instance into a target instance that represents, to its best, the same information under a different schema [FKMP03, FKP03]. The source and target schemas may however be inconsistent, so for a given source instance, there may be no target instance that represents the same information. While we have algorithms for detecting large classes of such inconsistency, designers may wish to modify either the source or target schema to prevent the inconsistency. This may be done by cleaning any inconsistent data in the source and adding a constraint to the source schema (or modifying its structure), modifying the target, or by modifying the mapping to only map consistent data (guaranteeing that a target instance exists). While the last option has been explored [CCGL02a], efficiently and effectively adapting a mapping to such constraint or structure modifications (in either the source or target) has not yet been considered.

Physical data design. Physical storage wizards, which permit the customization of physical schemas and storage structures [TSI96], must maintain a mapping between the physical and logical schemas. A common example includes tools for customizing the relational storage of XML data [BFH$^+$02]. Tuning wizards evaluate (or help a designer to evaluate) the relative cost of different physical relational designs. These wizards consider only a fixed (though large) set of physical schemas, each with a built-in mapping to the given logical (XML) schema. To permit a designer to suggest schema designs outside of this limited set, the tool would have to be able to adapt the XML to relational mapping to the *ad hoc* user-proposed schema change.

Other applications that rely on mappings including modeling of query capabilities [VP97], and view management [BHL83, KR01]. In all of these applications, mappings provide the main vehicle for data sharing and data transformation. Yet, current solutions in these application areas typically assume that the schemas are relatively static. Schema evolution is an important problem that has not received considerable attention. The Web is a dynamic environment with no centralized authority and as often happens in heterogeneous dynamic environments, sources may evolve without prior notice not only their context but also their schemas or their query capabilities. Mappings that have been defined between sources may become invalid or inconsistent. Those mappings have to be detected and adapted appropriately in order to reflect the changes of the schemas. For reasonable small schemas, browsing a short list of simple mappings is a feasible option. However, as the structure of the data becomes larger and more complicated schemas become prevalent, one soon feels the need for an automatic way of maintaining the consistency of the mappings under schema changes. Even when the mappings that need to be modified after a schema change have been detected, there are usually numerous candidate rewritings. Especially, for users with no strong technical background the effort evolved in such a task is considerable since it requires writing and managing complex transformation queries and programs. Finding the right rewritting of a mapping affected by a schema change is a real challenge and the main issue on which we elaborate in this paper.

We advocate a novel framework that maintains the consistency of mappings under schema changes by finding rewritings that try to preserve as much as possible the semantics of the mappings. We call this problem *mapping adaptation* to differentiate it from view adaptation [MD96, GMR95b], view synchronization [LNR02], and view maintenance [Wid95].

One way to approach the problem is to have a predefined finite set of interesting changes. Indeed, this is the approach used in several of the application areas that we have mentioned, including in physical design tools. For each such change, a modified mapping is stored ("hard-coded" if you will). The advantage of this approach is that we will known exactly how to handle each change. The disadvantage is that the way in which the schema can evolve is restricted to a set of predefined schemas. Though, if the set is rich enough, it may embrace all the possible schemas that are important for a specific application. A second alternative is to allow schemas to evolve and then find the changes that took place by comparing the modified schema (S') to the original version S). For example, we could using a matching tool to find corresponding portions of the two schema versions [RB01] and then use a mapping creation tool to add semantics to these correspondences [PVM^{+}02b]. This will produce a mapping from S' to S which can be composed with the original mapping. Such an approach is complementary to the approach we consider here (indeed, we will compare our solution, with this approach in Section 8).

Our approach is to use a mapping adaptation tool in which a designer can change and evolve schemas. The tool detects mappings that are made inconsistent by a schema change and incrementally modifies the mappings in response. This approach has the advantage that we can tract semantic decisions made by a designer either in creating the mapping or in earlier modification decisions. These semantic decisions are needed because schemas are often ambiguous (or semantically impoverished) and may not contain sufficient information to make all mapping choices. We can then reuse these decisions when appropriate.

Our main contributions are the following. (i) We motivate the problem of adapting mappings to schema changes and we present a simple and powerful model for representing schema changes. (ii) We consider changes not only to the structure of schemas (which may make the mapping syntactically incorrect [BHL83]) but also to the schema semantics. The latter changes may make mappings semantically incorrect. (iii) We develop an algorithm for enumerating possible rewritings for mappings that have become invalid or inconsistent. The generated rewritings are consistent not only with the the structure but also with the semantics of the schema (iv) We consider changes not only in the source schemas but also in the target. This is equivalent to adapting mappings to reflect changes in both their interface and in the base schema. (v) We support changes not only on atomic elements but also on more complex structures like relational tables or complex (nested) XML structures. (vi) We present a mapping adaptation algorithm that efficiently computes rewritings by exploiting knowledge about user decisions that is embodied in the already existing mappings.

In Section 2, we present related work We define mappings in Section 3 and 4 and we present our algorithm in Section 5 and our experience with our tool in Section 8 before we conclude in Section 9.

2 Related Work

Schema evolution is a broad research area that has been identified long ago and has been studied in different contexts and under different assumptions. It includes problems that are related to changes in the schema.

In *object-oriented database management systems (OODBMS)* researchers deal with the problem of how to modify efficiently the instance data under schena chanege. Banerjee et al. [BKKK87] gave a taxonomy of the changes that may occur in OODBMS and provided an implementation for them. Those changes were local to a type, e.g., renaming an attribute or changing the position of a class in the class hierarchy. Lerner [Ler00] extended the above work to include complex changes by creating templates for common changes. Complex changes involve more than one type, for example, moving an attribute from one type to another. Lerner exploited the power of object-oriented languages to describe a template for each kind of change. When a schema change occurs, the corresponding template is executed. Despite the expressive power of Lerner's model, the approach is not generic in the sense that it can support only those changes for which a template has been defined. Furthermore, the scope of the work in OODBS in general is restricted only to updates of the data and does not consider methods to update existing queries or views.

A problem closely related to mapping adaptation that has received considerable attention is *incremental view maintenance* [MQM97],[CW91]. View maintenance considers materialized views and a set of changes on the data of the base relations. The problem is to efficient update the materialized view to reflect the data changes. View maintenance has been extensively studied in the literature and a number of interesting solutions have been proposed ranging from the use of auxiliary data structures [BLT86] and production rules [CW91], to selective recomputation [MQM97]. In view maintenance, schemas are assumed to be static so updates of the view definitions have not been considered.

View adaptation is a variant of the *view maintenance* problem. It deals with the problem of keeping a materialized view up-to-date in response to changes of the view definition itself. The goal is to find better ways than recomputing the view from the base relations. One way to attack this problem is to keep a small amount of additional information beyond the data in the view, such as keys of participating relations [GMR95a] or join counts [MD96]. View adaptation is a problem complementary to mapping adaptation. Once a schema has changed and the mappings (or views) have been adapted to reflect the change, view adaptation techniques can be used to update the view data to be consistent to the new definition. View adaptation is a part of a broader problem called *view management* that includes any issue related to the creation and manipulation of views, e.g., reusing view to optimize query answering or data storage in cases of materialization [KR99]. The problem of view management has been identified long ago in the commercial database system community where there was a need for detecting views (materialized or not) that become invalid due to schema changes on the base tables [BHL83].

A different approach in schema evolution has been followed by McBrien and Poulovassilis [MP02] who combine schema evolution and schema integration in one unified framework. By using a series of primitive schema transformations one can map a local schema to a global schema. Each transformation must be accompanied by a query that describes its semantics. This query has to be manually specified by the user. Their approach enables easy composition of the transformations and permits optimization. In our approach, we are trying to relieve the user from the task of manually specifying those queries.

The *view synchronization* problem (investigated in the EVE [LNR02] system) is the problem of updating the view definition when the base relational schema is modified. EVE requires the user who defines a view to specify how the system should behave when there is a change in the base schema. Inclusion dependences are utilised to find good substitutions for elements that have been deleted. The set of changes supported by EVE is restricted to only deletions and renamings. Changes such as moving or coping attributes are not considered. Schema constraints are used only in conjunction with another schema change (e.g., a deletion). A deletion of an inclution dependency for example has no effect on the view definitions. Furthermore, since the mappings that are considered follow the *global-as-view* model [Len02], the target can be used in the computation.

Our work can be seen within a general framework of model management in which schemas and views or mappings between them are considered and manipulated as first-class citizens. Schema matching [RB01] is a common first step that generates a set of syntactic correspondences between portions of two schemas. A schema mapping tool like Clio [PVM$^+$02b] can take those correspondences and (by using the semantics embedded in the schemas) generate semantic mappings. Our approach complements the above scenario. We take the mappings generated by a mapping tool or defined by a user and adapt them when schemas are changed in order to preserve the mapping consistency. This incremental approach allows us to reuse semantic choices made by a user in creating the mappings.

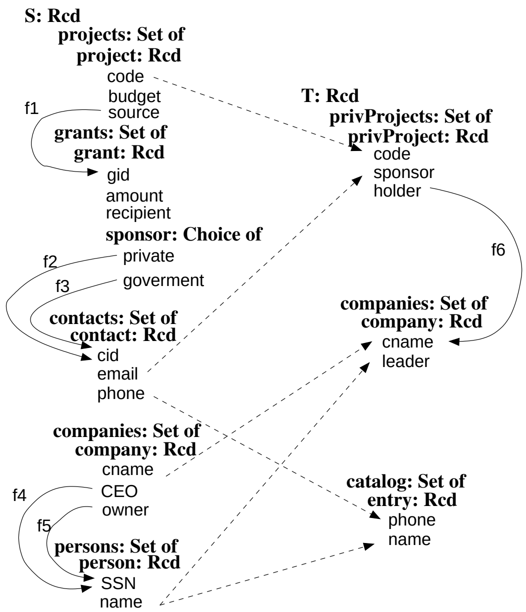
3 Mapping System

We consider a very general form of mapping that subsumes a large class of mappings used in a variety of applications. A *mapping* m from a schema $\mathcal{S}$ to schema $\mathcal{T}$ is an assertion of the form: $Q^S \rightsquigarrow Q^T$, where Q^S is a query over $\mathcal{S}$ and Q^T is a query over $\mathcal{T}$ [Len02]. Most commonly the queries are restricted to (type compatible) queries that return sets of tuples and the relation $\rightsquigarrow$ is the subset-or-equals relation $\subseteq$; such mappings are called *sound* mappings [Len02]. Other types of mappings include *complete* ($Q^S \supseteq Q^T$) and *exact* ($Q^S = Q^T$) [GM99]. Note that although the queries are restricted to return sets of tuples, the schemas may be nested schemas and may contain complex or abstract types. This form of mapping is very general and includes the following special cases.

User views. A user view on $\mathcal{S}$ is a query Q^S defining a schema element $\mathcal{T}$. For relational schemas, $\mathcal{T}$ is a single table (with no constraints). For XML schemas, $\mathcal{T}$ may be a complex structure defined in the *return* clause of, for example, an XQuery Q^S. Abusing notation a bit, the mapping is $Q^S \rightsquigarrow \mathcal{T}$.

Integrated views. In classical data integration, an integrated (or global) view $\mathcal{T}$ is defined over a set of source schemas $\mathcal{S}_1$, ..., $\mathcal{S}_n$. A set of mappings $Q^{\mathcal{S}_i} \rightsquigarrow T$ define how instances of the integrated view may be created from the sources. Such mappings are often referred to under the acronym GAV (global-as-view). Most work has considered sound mappings [Len02]; complete and exact integrated view mappings of this form have been studied in [GM99].

Modeling source descriptions. Given a fixed integrated schema $\mathcal{T}$, data sources ($\mathcal{S}_i$) related to this schema may be




```
m1: foreach p in S.projects, g in S.grants,
            r in g.grant.sponsor→private, c in S.contacts
      where p.project.source=g.grant.gid and
            r=c.contact.cid
    exists  i in T.privProjects, o in T.companies
      where o.company.cname=i.privProject.holder
    with    i.privProject.code=p.project.code and
            i.privProject.sponsor=c.contact.email

m2: foreach c in S.companies, p in S.persons,
      where c.company.owner=p.person.SSN
    exists  o in T.companies
    with    o.company.cname=c.company.cname and
            o.company.leader=p.person.name

m3: foreach c in S.contacts, p in S.persons,
      where p.person.SSN=c.contact.cid
    exists  e in T.catalog
    with    e.entry.name=p.person.name and
            e.entry.phone=c.contact.phone
```

Figure 1: A Mapping System.

modeled as views on $\mathcal{T}$ [LRO96]. Such mappings have the form $\mathcal{S}_i \leadsto Q^T$ and are referred to as LAV (local-as-view). **GLAV.** When mappings are not restricted at all and have their most general form $Q^S \leadsto Q^T$, they may be called GLAV (global or local-as-view) mappings [FLM99]. Such mappings have been used in transforming data between arbitrary schemas [PVM+02b] and in data exchange [FKMP03]. Views are traditionally restricted to contain no constraints. For example, user views and integrated views typically have no contraints. However, some research has considered mappings between two schemas both containing constraints [CCGL02b].

In this work, we will manage sets of mappings. These mappings may be created by a schema mapping tool [PVM+02b] or created manually. Alternatively, the mappings may be a set of views (user views or integrated views) defined, again manually or automatically, over a schema $\mathcal{S}$. In this case, the schema $\mathcal{T}$ is the set of structures defined by the views.

Definition 3.1 *A* **mapping system** *is a triple* $<\mathcal{S}, \mathcal{T}, \mathcal{M}>$ *where* $\mathcal{S}$ *and* $\mathcal{T}$ *are a source and target schema and* $\mathcal{M}$ *is a set of mappings between* $\mathcal{S}$ *and* $\mathcal{T}$.

Before defining mappings and schemas formally, we give an example to show how mappings may determine or constrain the placement of source data in the target.

Example 3.2 *Consider the mapping system of Figure 1. The schemas are shown in a nested relational representation that is used as a common data model. The left-hand schema* **S** *represents a source XML-Schema with information about projects, grants, companies and persons. Each project has a specific grant. A grant has a sponsor that is either a private individual or a government. Companies have an owner and a CEO. Relationships between different schema elements are specified via foreign keys (shown with solid lines in the figure). The right-hand schema* **T** *is a relational schema that also contains information about projects and companies. However, it contains only privately funded projects and associates each project with the company in charge of the project. Three mappings (m_1, m_2 and m_3) have been defined from* **S** *to* **T**. *The mappings are also expressed in a nested relational representation (defined formally below) that can easily be transformed to other representations [PVM+02a], e.g., XQuery. Each mapping has the form $Q^S \leadsto Q^T$. In our notation, the* foreach *clause (with the associated* where*) defines Q^S while the* exists *clause (with the associated* where*) defines Q^T. These mappings specify a containment assertion ($\subseteq$): for each tuple returned by Q^S, there must* exist *a corresponding tuple in Q^T. We use the* with *clause to make explicit how the source and target elements relate to each other.*

Mapping m_2 specifies how to populate companies in the target schema with a company name and the name of the owner who is considered to be the leader of the company. Mapping m_2 is a GAV mapping. Mapping m_1 populates the target with privately funded projects and is a GLAV mapping. Note that m_1 respects the foreign key on the target and requires that for each project there be an associated company with `holder=cname`. *However, a specific value for this company cname is not specified (it is only required to exist). So m_1 constrains the target but does not completely specify a target instance* *(a property shared by many LAV and GLAV mappings). The third mapping m_3 generates phone entries in the target by joining* `persons` *and* `contacts` *in the source.* ∎

Schemas. The schemas we consider are nested schemas. A *schema* is a set of labels (called roots), each with an associated type. A type τ is defined by the grammar: τ::= String | Int | SetOf τ | Rcd$[l_1{:}\tau_1, \ldots, l_n{:}\tau_n]$ | Choice$[l_1 : \tau_1, \ldots, l_n{:}\tau_n]$. Types Int and $String$ are called atomic types, Set is a collection type and the types Rcd and $Choice$ are complex types. With respect to XML Schema, we use Set to model repeatable elements (or groups of elements), while Rcd and $Choice$ are used to represent the "all" and "choice" *model groups*. For each set type SetOf τ, τ must be an atomic (String or Int) type or a record type. We do not consider order, Set represents unordered sets. "Sequence" model groups of XML Schema are represented as (unordered) Rcd types (as with "all").

For queries we adopt the OQL *select-from-where* syntax [BCD92] enhanced with choice type selections. An expression e is defined by the grammar $e ::= S|x|e.l$ where x is a variable, S a schema root, l a record label and $e.l$ a record projection. Queries have the following form where e_i, c_i and c'_i are expressions containing only variables x_i that are bound in the from clause.

select $e_0, e_1, ..., e_m$
from x_0 in P_0, x_1 in $P_1, ...\ x_n$ in P_n
where $c_0{=}c'_0$ and $c_1{=}c'_1$ and ... and $c_k{=}c'_k$

Each P_i in the from clause is either: (1) an expression e with type SetOf τ; in this case, the variable x_i will bind to individual elements of the set e, or (2) $e{\rightarrow}l$ (where e is an expression with a type Choice $[\ldots, l : \tau, \ldots]$) representing the selection of attribute l of the expresssion e; in this case, the variable x_i will bind to the element of type τ under the choice l of e. The query is well-formed if the variable (if any) used in P_i is defined by a previous in clause. The conditions in the where clause are optional. Following XQuery and OQL convention we will use queries to identify elements within schemas. A schema element is identified with the path query that can be used (intuitively) to retrieve all the instances of that element.

Definition 3.3 *A* **schema element** *in schema S is a path query, that is a query of the form:*
select e_{n+1} from x_0 in P_0, x_1 in P_1, ... x_n in P_n
where each P_k with $k{>}1$ uses variable x_{k-1}, P_0 is an expression starting at a schema root in S and expression e_{n+1} uses variable x_n.

If the details of the from clause are unimportant, we refer to such schema element using the notation select e from P.

Example 3.4 *For Schema* **S** *of Figure 1, the schema elements* `amount` *and* `private` *under* `grant` *are formally defined, respectively, by the following two path queries:*

a_1 select g.grant.amount from g in S.grants
a_2 select s from g in S.grants,
s in g.grant.sponsor→private

∎

Constraints. For *schema constraints* we consider a very general form of referential constraints called *nested referential integrity constraints (NRIs)* [PVM+02b]. NRIs capture naturally relational foreign key constraints as well as the more general XML Schema keyref constraints.

Example 3.5 *The foreign key f_2 on the source schema of Figure 1 is expressed as the following NRI.*

f_2: foreach g in S.grants, r in g.grant.sponsor→private
exists c in S.contacts
with c.contact.cid= r

∎

The simplest form of NRI relates two schema elements select e_1 from P_1 and select e_2 from P_2. Such a constraint has the form foreach P_1 exists P_2 with $e_1 = e_2$. This is a simple unary inclusion constraint. More generally, and as in XML Schema, an NRI is relative (i.e., local) to a given schema element (the "context" element) select e_0 from P_0. So an NRI has the general form: foreach P_0 [foreach P_1 exists P_2 with C], where P_1 and P_2 are now relative to (i.e, start from) the last variable of P_0. In this expression C is a conjunction of one or more equalities $e_1 = e_2$ where e_1 and e_2 are expressions that use the last variable of P_1 and P_2, respectively. Note that such constraints can also be written as foreach X exists Y with C, where Y may be relative to some variable of X.

Union types occur frequently in real world schemas and play an important role in query answering and data transfer. The NRIs we are using have been naturally extended to support the structure and the semantics of those types. In particular, the NRIs we consider are a special form of the *disjunctive embedded dependences* as introduced by Deutch and Tanner [DT01]. A disjunctive embedded dependency is a dependency that may contain more than one exists clauses separated by disjunctions. Disjunctions are useful in order to explicitly model the various selections that can be made to union type elements.

Example 3.6 *The foreign key constraint f_1 of Figure 1 is represented as:*

f_1: foreach p in S.project,
  exists g in S.grants
    where g.grant.gid= p.project.source

The meaning of the above constraint is that for each project element, there is a grant that is used a the funding source of the project. Based on the schema information it can be infered that variable g that is bound to grants, may represent either a private grant or one that comes from the goverment. Constraint f_1 can be rewritten to the followin equivalent constraint that explicitly state that fact:

f_1: foreach p in S.project,
  exists g in S.grants, s in g.grant.sponsor→private
    where s.grant.gid= p.project.source
  OR
  exists g in S.grants, s in g.grant.sponsor→goverment
    where s.grant.gid= p.project.source

■

We will refer to this representation in which all the choices of the union types involved in the constraint are explicitly shown, as the *analytic* representation of the constraint. We will see later when the chase method is described, the important role of this representation.

The number of disjunctions that exist in the analytic representation of a constraint depends exponentially to the number of union types of the schema and the number of choices that each one has. The time needed to generate the analytic representation of a constraint, on the other hand, is linear to the size of the schema. More specifically, the algorithm works as follows. For each schema root or bound variable in the exist part of the constraint, we start visiting its type in a depth first order. Traversion is not continued past set type elements or leaves. When a a union type is met, if none of its choices is already recorded in the constraint, a set of new conjunctions of exists part is generated. Each new conjunction is one of the previous one, enhanced with a selection of the choice type that was met.

The analytic representation of a constraint can always be transformed to one with one singe exists part, in which the union type choices are implicit. We will refer to this representation as the *compact* representation of the constrait. A *compact* representation can always be created in time linear to the size of the constraint.

4 Semantically Valid Mappings

For a given pair of a source and a target schema there are many mappings of the form foreach A^S exists A^B with C. When a schema changes, we need to rewrite the affected mappings. Our goal is to find rewritings that are consistent with the semantics of the new schema and with the current semantics of the mapping. To achieve the former (consistency with the new schema), we use an extension of the Clio mapping creation framework [PVM$^+$02b] in which mappings are created based on the semantics of the schemas. We extend this foundation by providing algorithms to efficiently compute this schema semantics incrementally when a change to the schema structure or constraints occurs. For the latter (consistency with the mapping), we present new techniques for modeling and reusing the semantics embedded within a mapping. When the semantics of the mapping must change, we make the minimum change necessary to achieve a mapping that is consistent with the new schema. While Section 5 will give the algorithms necessary for adapting such mappings when schemas change, in this section we describe in detail the mappings that we consider.

We define the notion of *association* to describe a set of associated atomic schema elements. Intuitively, an association is a query that returns all the atomic elements mentioned in a query.

Definition 4.1 *An* **association** *is a query on schema S:*

select $*$ from x_1 in P_1, x_2 in $P_2, \ldots x_n$ in P_n
where $e_1{=}e'_1$ and $e_2{=}e'_2$ and ... and $e_n{=}e'_n$

The '' symbol denotes all the valid expressions with an atomic type that can be in the* select *clause of the query.*

Example 4.2 *To identify the element* `private` *in Schema* **S**, *we use the query a_2 from Example 3.4. The following (similar) query defines an association containing not only* `private`, *but also the atomic elements* `gid`, `amount`, *and* `recipient`.

A_2 select $*$ from g in `S.grants`,
s in g`.grant.sponsor`$\rightarrow$`private` ∎

Associations are simply collections of atomic elements and can be used in mappings. So for our schemas, mappings are simply very general referential constraints between a source and target association. We note, however, that these constraints are more general than the NRIs that we use to express schema constraints.

Definition 4.3 *A* **mapping** *is a constraint* foreach A^S exists A^T with C, *where A^S is an association on a source schema S, A^T is an association on a target schema T and C is a non-empty conjunction of equality conditions relating expressions over S with expressions over T.*

While our techniques are designed to manage very general mappings of this form, we will make use of a few important special classes of mappings. The first are correspondences which relate two schema elements, and the second are meaningful mappings which are considered in more detail in the next section.

Correspondences. A *correspondence* is a specification that describes how the value of an atomic target schema element is generated from the source. A correspondence can be represented as simple inter-schema referential constraints. A correspondence from a source element select e^S from P^S to a target element select e^T from P^T is an inter-schema NRI foreach P^S exists P^T with $e^S{=}e^T$. Correspondences are implicit in the mappings (and view definitions) and can be easily extracted from them.

Example 4.4 *The correspondence between the* `leader` *in the source schema and the* `name` *in the target (depicted in Figure 1 with a dotted line) is:*

v: foreach e in `S.persons`
exists c in `T.companies`
with c`.company.leader=` e`.person.name` ∎

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

Definition 4.5 *An association A is* **dominated** *by association B (noted as $A\dot{\preceq}B$) if there is a renaming function h from the variables of A to the variables of B such that the* from *and* where *clauses of $h(A)$ are subsets, respectively, of the* from *and* where *clauses of B.*

Example 4.6 *The association A_2 shown in Example 4.2 is dominates by association*

A_t select $*$ from x_1 in `S.projects`, x_2 in `S.grants`, x_3 in x_2`.grant.sponsor`$\rightarrow$`private` x_4 in `S.contacts`
where x_1`.source=`x_2`.gid` and x_3`=`x_4`.cid`

since there is a renaming function $h:\{g{\rightarrow}x_2, s{\rightarrow}x_3\}$ that makes the from *clause of the A_2 a subset of A_t. (The* where *clause of A_2 is empty so it is by default a subset of A_t).* ∎

Domination can naturally extend to mappings as follows. Mapping m_1:=foreach A_1^S exists A_1^T with C_1, is dominated by mapping m_2:=foreach A_2^S exists A_2^T with C_2 (noted as $m_1\dot{\preceq}m_2$) if and only if $A_1^S\dot{\preceq}A_2^S$, $A_1^T\dot{\preceq}A_2^T$ and for every equality conditions $e{=}e'$ in C_1, $h_1(e)=h_2(e')$ is in C_2, where h_1 and h_2 are the renaming functions from A_1^S and, respectively, A_1^T to A_2^S and, respectively, A_2^T.

There are three ways in which semantic relationships between schema elements can be encoded. The first is through the structure of the schema. Elements may be related by their placement in the same record type or more generally by their placement under the same parent schema. An association containing elements that are related only through the schema structure is referred to as a *structural association*. Structural associations correspond to the primary paths used in [PVM$^+$02b] where it is shown that they can be computed by an one time traversal over the schema.

P_1^S : select * from p in S.projects
P_2^S : select * from g in S.grants,
r in g.grant.sponsor→private
P_3^S : select * from g in S.grants,
m in g.grant.sponsor→goverment
P_4^S : select * from a in S.contacts
P_5^S : select * from c in S.companies
P_6^S : select * from i in S.persons

P_1^T : select * from p in T.privProjects
P_2^T : select * from p in T.companies
P_3^T : select * from p in T.persons

Figure 2: Source and target structural associations

Definition 4.7 *A* **structural association** *is an association*
select * from x_1 in P_1, x_2 in P_2, ... x_n in P_n
with no where *clause and where the expression* P_1 *must start at a schema root and every expression* P_k, k>0 *starts with variable* x_{k-1}.

The schema structure encodes a set of semantic relationships that a designer chose to model explicitly. A second way of encoding semantic associations is in a mapping. A mapping, is an encoding of a pair of source and target associations (which may or may not be explicitly present in the schema structure). A mapping may use associations provided by a user or mapping tool. Such mappings may expose hidden semantic relations between schema elements.

Definition 4.8 *A* **user association** *is an association that has been provided to the system via an existing mapping.*

Example 4.9 *Mapping* m_3 *joins* contacts *and* persons *based on the* SSN *and* cid *which indicates that a person can be associated with its contact information through the SSN. This generates the user association:*

select * from c in S.contacts, p in S.persons,
where c.contact.cid=p.person.SSN

■

A third way to semantically relate elements is through schema constraints. Chasing is a classical relational method [MMS79] that can be used to assemble elements that are semantically related through constraints. A chasing is a series of chase steps. A chase step of association R with an NRI F: foreach X exists Y with C, can be applied if, by definition, the association R contains (a renaming of) X but does not satisfy the constraint, in which case the Y clause and the C conditions (under the respective renaming) are added to the association. The chase can be used to enumerate logical join paths, based on the set of dependencies in a schema. We use the nested chase [PT99], a variation of the classical relational chase that can handle nested structures. However, the specific chase does not work well in the presence of union types. There are two reasons for that. The first is that the variable that represents a choice in a union type may not be present. The second is depending on the choices taken in a union type elements, the chase may result to different results. We extended the nested chase to cope with the above two problems, in two ways. First, we consider only the extended version of constraints, that is the version in which the choices are explicitly stated. Those constraints as mentioned earlier, usually consist of conjuctions. A natural consequence of the conjunstions is that chasing with a specific constraint results to different results. Hence, we have also extended the nested chase to consider as a result of the chase, not only one associoantion, but multiple. We illustrate the above with the following example.

Example 4.10 *Association* P_1^S *of Figure 2 can be chased with the constraint* f_1 *since there is a renaming function (in this case is the identity function) that makes the* foreach *part of the constraint to be a subset of the association* P_1^S*, but not both the* foreach *and the* exists *part. If we use the compact version of the constraint shown in Example 3.6, the* exists *part of the constraint will be appended in the association and the result of the chase will be the new association:*

select * from p in S.projects, g in S.grants,
where p.project.source=g.grant.gid

The above constraint cannot be chased further with any constraint, e.g., constraint f_2 *shown in Example 3.5 since there is no renaming function to make* foreach *part of* f_2 *a subset of the association (there is no way for variiable* r *to be mapped in the above association). However, if during the first chase step we had used the analytic form of the constraint* f_1 *as shown in Example 3.6, the result would have been two different associations. One that considers the private grants and one that considers grants coming from the goverment. More spefically, the result would have been:*

```
A1: select *
    from   p in S.projects, g in S.grants,
           r in g.grant.sponsor→private, c in S.contacts
    where  c.cid=r and g.gid=p.source
A2: select *
    from   p in S.projects, g in S.grants,
           r in g.grant.sponsor→goverment, c in S.contacts
    where  c.cid=r and g.gid=p.source
A3: select *
    from   g in S.grants, c in S.contacts,
           r in g.grant.sponsor→private,
    where  c.cid=r
A4: select *
    from   g in S.grants, c in S.contacts,
           r in g.grant.sponsor→goverment
    where  c.cid=r
A5: select * from c in S.contacts
A6: select *
    from   c in S.companies, p in S.persons, w in S.persons
    where  c.company.CEO=p.person.SSN and
           c.company.owner=w.person.SSN
A7: select * from c in S.persons
A8: select *
    from   c in S.contacts, p in S.persons,
    where  p.contact.cid=p.person.SSN

B1: select *
    from   p in T.privProjects, c in T.companies
    where  p.privProject.holder=c.company.cname
B2: select * from c in T.companies
B3: select * from c in T.persons
```

Figure 3: Logical associations for schemas S and T.

```
select * from p in S.projects, g in S.grants, s in g.grant.sponsor→private
where p.project.source=g.grant.gid
```

and also

```
select * from p in S.projects, g in S.grants, s in g.grant.sponsor→goverment
where p.project.source=g.grant.gid
```

From the above two associations, the first one can now be further chased with f_2, and the second with f_3. ∎

Definition 4.11 *Let $chase_{\mathcal{X}}(P)$ denote the result set of the chase of a structural or a user association P with the set $\mathcal{X}$ of all the NRIs of the schema. A* **logical association** *R is an association in $chase_{\mathcal{X}}(P)$.*

Example 4.12 *The fact that* name, CEO, *and* owner *are all under the* company *element indicates that they are semantically associated since they all refer to the same company. These three elements form the structural association P_5^S of Figure 2. This figure gives all the structural associations of the schemas in Figure 1. The foreign key f_5 on the source schema indicates that a person and a company can be associated through an owner relationship. Similarly, the foreign key f_6 indicates that they can also be related through the CEO. Chasing structural relation P_5^S with the constraints f_5 and f_6 results in the logical association A_6 of Figure 3. Logical associations A_1 to A_7 and B_1 to B_3 are the logical associations that resulted from the chase of structural associations of Figure 2 with all the constraints defined on the two schemas. Finally, the structural association we mentioned in Example 4.9 cannot be chased with any constraints, thus, it is a logical association (indicated in Figure 3 as A_8).* ∎

We can now give a formal definition of what we consider a semantically valid mapping:

Definition 4.13 *Given a pair of schemas $\mathcal{S}$ and $\mathcal{T}$, and a set of correspondences between them, a* **semantically valid mapping** *is an expression of the form* foreach A^S exists A^T with D, *where A^S and A^T are logical associations in the source and the target schema correspondingly, and D is the conjunction of the conditions of the correspondences that are* **covered** *by the pair $<A^S, A^T>$ (provided that at least one such correspondence exists). A correspondence v:*foreach P^S exists P^T with D **is covered** *by the associations $<A^S, A^T>$ if $P^S \dot{\preceq} A^S$ and $P^T \dot{\preceq} A^T$.*

Example 4.14 *The two correspondences of the mapping m_2 are covered by the pair $<A_6, B_2>$ in two ways. One way of coverage produces the mapping*

```
mu: foreach c in S.companies, p in S.persons, p' in S.persons
      where c.company.CEO=p.person.SSN and
            c.company.owner=p'.person.SSN
    exists  o in T.companies
    with    o.company.cname=c.company.cname and
            o.company.leader=p.person.name
```

which is a semantically valid mapping that is equivalent to

```
mo: foreach c in S.companies, p in S.persons
      where c.company.CEO=p.person.SSN
    exists  o in T.companies
    with    o.company.cname=c.company.cname and
            o.company.leader=p.person.name
```

The second way of covering gives mapping m_2 of Figure 1. The difference between m_o and m_2 is that the M_o considers the CEO as the leader of a company the CEO while m_2 considers the owner. This example shows how our approach can capture the different join paths in the schema and produce semantically different mappings. ∎

We have to note here that the semantically valid mappings set includes the mappings produced by the mapping generation tool Clio [PVM+02b] with the addition of those based on user choices and those including choice types. This set will be our search space when looking for possible rewritings when the schema changes.

Definition 4.15 *Given a source and a target schema $\mathcal{S}$ and $\mathcal{T}$, along with a set of correspondences $\mathcal{C}$ from $\mathcal{S}$ to $\mathcal{T}$, a* mapping universe $\mathcal{U}^{\mathcal{C}}_{\mathcal{S},\mathcal{T}}$ *is the set of all the semantically valid mappings.*

5 Handling Schema Evolution

Schemas usually evolve to adapt to new data requirements and semantics. When a schema changes, we need to rewrite the affected mappings in a way that is consistent with the semantics of the new schema and with the semantics of the existing mappings. To achieve the former we exploit information provided by the schema structure and semantics (constraints) by extending the algorithm presented in [PVM+02b]. We provide algorithms to efficiently compute the schema semantics incrementally when a change to the schema structure or constraints occurs. For the latter, we present new techniques for modeling and reusing the semantics embedded within a mapping. When the semantics of the mapping must change, we make the minimum change necessary to achieve a mapping that is consistent with the new schema.

When schemas evolve, most of the time, they do not change radically and the evolution can be described as a sequence of primitive changes. We have identified a set of such changes that describe how a schema can be modified. The changes can be categorized in three main groups. The first group contains operations that change the schema semantics by adding or removing constraints. The second includes modifications to the schema structure by adding or removing elements, while the third category is about changes that reshape the schema structure by moving, copying and renaming elements. In the following subsections, we will explain how the mappings adapt whenever each such operation occurs.

To make the presentation less verbose we will often assume that the schema changes occur in the source schema. However, the algorithms we describe equally apply in the case in which the changes occur in the target schema.

Given a mapping framework $< \mathcal{S}, \mathcal{T}, \mathcal{M} >$, a schema changes generated a new mapping framework $< \mathcal{S}', \mathcal{T}', \mathcal{M}' >$ such that $\mathcal{S}'$ and $\mathcal{T}'$ will be the schemas $\mathcal{S}$ and $\mathcal{T}$ correspondingly after the schema change is applied to them, and $\mathcal{M}'$ the new updated set of mappings. In the following sections we explain how to detect such mappings and how a good replacement is selected.

5.1 Constraint modifications

5.1.1 Adding Constraints

Adding a new constraint on a schema does not make any of the existing mappings invalid, i.e., syntactically incorrect. However, it may make some of the mappings inconsistent, in the sense that they will no longer reflect the semantics of the schema. More precisely, a mapping may fall out of the mapping universe (recall Definition 4.15) as a result of adding a constraint. Let $<\mathcal{S},\mathcal{T},\mathcal{M}>$ be a mapping system, and $\mathcal{C}$ be the set of correspondences extracted from the mappings $\mathcal{M}$. Assume that a new constraint F: foreach X exists Y with C is added in the source schema.

We first detect what are the mappings that are affected by the change, that is mappings that are not semantically valid any more according to the new requirements of the schemas. A mapping m: foreach A^S exists A^T with D, with $m\in\mathcal{M}$ of a mapping system $<\mathcal{S},\mathcal{T},\mathcal{M}>$ needs to adapt after the addition of a source constraint foreach X exists Y with C if X is dominated by A^S ($X\dot{\preceq}A^S$), with a renaming h, but there is no extension of h to a renaming from $X\cup Y\cup C$ to A^S. In other words, the addition of the new constraint caused A^S not be closed under the chase and therefore not be a logical association. The mapping must therefore be adapted.

If mapping m: foreach A^S exists A^T with D, with $m\in\mathcal{M}$ of the mapping system $<\mathcal{S},\mathcal{T},\mathcal{M}>$ needs to adapt, the association A^S is chased with the set of the old schema constraints enhanced with the new constraint F. Note that it is not enough to chase only with F since the result of that chase step may allow further chasing with some old constraints that was not possible before. The result is a set of new logical associations. For each such association A, a new mapping is generated in the form m_c: foreach A exists A^T with D'. The set D' consists of the conditions derived from the correspondences in $\mathcal{C}$ that are covered by the pair $<A, A^T>$. Since A is generally a larger logical association than A^S, naturally, $D\subseteq D'$. Each mapping m_c generated by the above procedure is appended in $\mathcal{M}$ and mapping m is removed. Algorithm 5.1 gives a brief description of the steps taken when a new constraint is added.

Example 5.1 *Assume that the following new constraint is added in the source schema of Figure 1:*

```
f7: foreach g in S.grants
    exists  c in S.companies
    with    c.company.cname= g.grant.recipient
```

allowing each grant to specify the company that receives the grant. Mappings m_2 and m_3 will not be affected since the foreach *part of the constraint is not dominated by the* foreach *part of those mappings. Indeed, the fact that we can now determine the company that receives each grant has nothing to do with those two mappings that deal with companies and persons only. On the flip side, this change greatly affects mapping m_1. Remember that the specific mapping was populating the target schema with private projects and the associated companies, but the information of what company is related to each project was not available in the source schema. After the addition of the new constraint, this information becomes available, so mapping m_1 needs to adapt to the new schema semantics. We detect this by verifying that the* foreach *clause of the constraint is dominated by (contained in) association A_1 used in mapping m_1 but the union of the* exists *and* with *clause is not. When chased with the new set of constraints that includes F, association A_1 gives a new logical association:*

```
A1a: select *
     from   p in S.projects, g in S.grants,
            r in g.grant.sponsor→private, c in S.contacts
            o in S.companies, e in S.persons, e' in S.persons
     where   p.project.source=g.grant.gid and
             r=c.contact.cid and
             g.grant.recipient=o.company.cname and
             o.company.CEO=e.person.SSN and
             o.company.owner=e'.person.SSN
```

This association generates, in turn, two rewritings for m_1, depending on the way the value of `leader` *in the target can be obtained: as the name of the* `CEO` *of a company or as the name of the* `owner` *of the company. The first is:*

```
m'1a: foreach p in S.projects, g in S.grants,
              r in g.grant.sponsor→private, c in S.contacts
              o in S.companies, e in S.persons
        where p.project.source=g.grant.gid and
```

r=c.contact.cid and
g.grant.recipient=o.company.cname and
o.company.CEO=e.person.SSN
exists j in T.privProjects, m in T.companies
where j.privProject.holder=m.company.cname
with m.company.cname=o.company.cname and
m.company.leader=e.person.name and
j.privProject.code=p.project.code and
j.privProject.sponsor=c.contact.email

and the second is

m'_{1b}: foreach p in S.projects, g in S.grants,
r in g.grant.sponsor→private, c in S.contacts
o in S.companies, e in S.persons
where p.project.source=g.grant.gid and
r=c.contact.cid and
g.grant.recipient=o.company.cname and
o.company.CEO=e.person.SSN
exists j in T.privProjects, m in T.companies
where j.privProject.holder=m.company.cname
with m.company.cname=o.company.cname and
m.company.leader=e.person.name and
j.privProject.code=p.project.code and
j.privProject.sponsor=c.contact.email

The second mapping m'_{1b} is the same as m'_{1a} apart from the last condition in the first foreach where *clause. This means that the first one populates the target schema with private projects and companies, using the CEO as the leader of the company while the second uses the owner.* ■

Choosing one mapping rewriting in favor of another cannot always be done using the available information. All the rewritings are consistent with the new schema and the previously defined mappings (i.e., they are valid members of the new mapping universe). If some of them have to be rejected, this would have required human intervention. In Section 7 we describe a methodology for ranking the generated rewritings based on how semantically close they are to the previous mappings that can be used to assist the user in such a decision.

A special, yet interesting, case is when the chase will not introduce any new schema elements in the association but only some extra conditions. Those conditions will introduce new ways (join paths actually) that the elements in the association can be related. Despite the fact that this adds new semantically valid mappings in the mapping universe, none of the existing mappings is adapted and no new mapping is generated. The intuition behind this is that there is no indication of whether those new mappings are included among the intentions of the initial mappings but could not have been defined before. Neither the existing mappings nor the schemas and constraints can specify that. Hence, since our goal is to maintain the semantics of existing mappings as much as possible, we perform no adaptations unless necessary.

Algorithm 5.1 - Constraint addition

Input: Mapping System $<\mathcal{S}, \mathcal{T}, \mathcal{M}>$
New constr. F : foreach X exists Y with C in $\mathcal{S}$
Body: $\mathcal{X} \leftarrow$ constraints in $\mathcal{S}$, $\mathcal{M}' \leftarrow \emptyset$
$\mathcal{C} \leftarrow$ compute correspondences from $\mathcal{M}$
For every m←(foreach A^S exists A^T with D) $\in \mathcal{M}$
if ($X \dot{\preceq} A^S$ with renaming h and $h(X \cup Y \cup C) \dot{\npreceq} A^S$)
For every $A \in chase_{\mathcal{X} \cup \{F\}}(A^S)$
$D' \leftarrow \{c \mid c \in \mathcal{C} \wedge$ c is covered by $<A, A^T>\}$
$m_r \leftarrow$ foreach A exists A^T with D'
include m_r in $\mathcal{M}'$
else include m in $\mathcal{M}'$
Output: New set of mappings $\mathcal{M}'$

5.1.2 Removing Constraints

Similarly to adding a constraint, removing one has no effect on the validity of the existing mappings but may affect the consistency of their semantics. The reason is that mappings may have used assumptions that were based on the constraint that is about to be removed. As before, we assume that a source constraint is removed. (The same reasoning applies for the target case.) We consider a mapping to be affected if its source association uses some join condition(s) based on the constraint being removed. More precisely, a mapping m: foreach A^S exists A^T with D, with $m \in \mathcal{M}$ of a mapping system $<\mathcal{S}, \mathcal{T}, \mathcal{M}>$, needs to adapt after the removal of a source constraint F = foreach X exists Y with C if $X \cup Y \cup C \dot{\preceq} A^S$.

Once we detect that a mapping needs to be adapted, we apply the following steps. (Algorithm 5.2 provides a succint description of these steps.) We start by breaking apart the source association A^S into its set $\mathcal{P}$ of structural associations, that is, we enumerate all the structural associations of the source schema that are dominated by A^S. To this, we add any possible user association that may exist (Recall that use associations are associations that are not determined by the schema design, but from the user through the defined mappings). We then chase these associations by considering the set of schema constraints *without* F. The result is a set of new logical associations. Some of these logical associations may include choices (due to the existence of choice types) that were not part of the original association A^S. We eliminate such associations. The criterion is based on dominance, again: we only keep those new logical associations that are dominated by A^S. Let us call this set of resulting associations $\mathcal{A}'$. By construction, the logical associations in $\mathcal{A}'$ will contain only elements and conditions that were also in A^S, hence, they will not represent any additional semantics. Among them, we need the one(s) that are as close as possible to A^S. Hence, the next step of the algorithm is to eliminate an association B from $\mathcal{A}'$ if B is dominated by some other A in $\mathcal{A}'$. In other words we only keep the maximal associations of $\mathcal{A}'$. Call this set $\mathcal{A}''$.

Finally, we generate a new mapping m_a: foreach A_a exists A^T with D', for every A_a in $\mathcal{A}''$, where D' represents the correspondences covered by m_a. Since, in general, a set of correspondences can D' can be covered in more than one way by an association, we would like also to keep the same coverage choices that have been made by the original mapping. This can be achieved by simply requiring that $m_a \dot{\preceq} m$. Due to the way it is constructed, a mapping like this always exists. The following example illustrates the algorithm.

Example 5.2 *Consider the mappings m'_{1a}, m'_{1b}, m_2 and m_3 we ended up in Example 5.1 and let us remove the constraint f_7 we added there. It is easy to see that mappings m_2 and m_3 are not affected because they do not include a join between* `S.grants` *and* `S.companies`. *However, both m'_{1a} and m'_{1b} are affected. Consider the mapping m'_{1a} (the other is handled in a similar way). Its source association, A_{1a} of Example 5.1, is broken apart into structural associations. These structural associations are (recalling Figure 2): P^S_1, P^S_2, P^S_4, P^S_5, and P^S_6. To this, we add the user associations that are also dominated by A_{1a}. The only user association that exists in our example is the one defined by mapping m_3:*

select *
from c in S.contacts, p in S.persons,
where c.contact.cid=p.person.SSN

but is not dominated by A_{1a}, so it is not included in the list of associations we consider. The associations we selected are then chased. Among the resulting logical associations, we have A_1 (recall Figure 3) as the result of chasing P^S_1. We also have an association A'_1, also as the result of chasing P^S_1 but with the choice of selecting `goverment` *rather than* `private` *for a* `grant` *element. This association is eliminated, because it is not dominated by the original association A_{1a}, which selects* `private` *and not* `goverment`. *The other associations that will be included in $\mathcal{A}'$ are A_3, A_5, A_6, A_7. However, in the next step, A_3, A_5 and A_7 are eliminated because they are not maximal: A_3 and A_5 are dominated by A_1, while A_7 is dominated by A_6. So, we are left with $\mathcal{A}'' = \{A_1, A_6\}$. Using A_1, the mapping m_1 is generated (which is the one we started in Example 5.1). Using A_6, two mappings can be generated, due to two different coverages for the* `leader` *of a company: one joins companies with persons via a join path based on* `CEO` *while the other joins companies with persons via a join path based on* `owner`. *Only the first mapping (call it m'_2) is dominated by the original mapping m'_{1a}: recall that m'_{1a} was also using the same join path based on* `CEO`. *The second mapping is thus eliminated. Hence, the result of the algorithm, for the mapping m'_{1a}, consists of the mappings m_1 and m'_2.* ∎

5.2 Schema pruning or expansion

Of the most common changes that are met in schema evolution systems are those that add or remove parts of the schema structure, like for example adding a new attribute on a relational table or removing an XML-Schema element.

Algorithm 5.2 - Constraint Removal

Input: Mapping System $<\mathcal{S}, \mathcal{T}, \mathcal{M}>$
Constraint F: foreach X exists Y with C of $\mathcal{S}$

Body: $\mathcal{X} \leftarrow$ constraints in $\mathcal{S}$, $\mathcal{M}' \leftarrow \emptyset$
For every m$\leftarrow$(foreach A^S exists A^T with D) $\in\mathcal{M}$
  If $(X \cup Y \cup C \dot{\preceq} A^S)$ {
    $\mathcal{T}\leftarrow\{T \mid T$ structural or user association$\}$
    $\mathcal{P}\leftarrow\{P \mid P\in\mathcal{T} \ \wedge \ P \dot{\preceq} A^S\}$
    $\mathcal{A}'\leftarrow\{A \mid A\in chase_{\mathcal{X}-F}(P) \ \wedge \ P\in\mathcal{P} \ \wedge \ A\dot{\preceq}A^S\}$
    $\mathcal{A}''\leftarrow\{A \mid A\in\mathcal{A}' \wedge$ there is no $A_0\in\mathcal{A}'$ with $A \dot{\preceq} A_0$ $\}$
    For every $A_a\in\mathcal{A}''$
      $m_a \leftarrow$ (foreach A_a exists A^T with D')
      If $(m_a \dot{\preceq} m)$ include m in $\mathcal{M}'$
  }
  else include m in $\mathcal{M}'$

Output: New set of mappings M'

5.2.1 Adding new structures

When a new structure is added to a schema, it may introduce some new structural associations. Those structural associations can be chased and generate new logical associations. Using those associations new semantically semantically valid mappings can be generated, hence the mapping universe is expanded. However, they are not added in the set of existing mappings. The reason is that there is no indication of whether they describe any of the intended semantics of the mapping system. This can be explained by the fact there is no correspondence covered by any of the new mappings that is not covered by any of those that already exist. On the other hand, since the structure and constraints used by the existing mappings in not affected, there is no reason for adapting any of them.

Example 5.3 *Consider the case in which the source schema of Figure 1 is modified so that each company has nested within its structure the set of laboratories that the company operates. This introduces some new mappings in the mapping universe, like, for example, the one that populates the target schema only with companies that have laboratories. Whether this mapping is among the intentions of the user who defined the mappings is something that cannot be determined neither from the schemas, nor from the existing mappings. On the other hand, mapping m_2 that populates the target with companies, independently of whether they have labs, continues to be valid and consistent.* ∎

5.2.2 Removing schema parts

In many practical cases, a part of the schema is removed either because the owner of the data source does not want to store that information any more, or because she may want to stop publishing it. The removal of an element forces all the mappings that are using that element to be adapted. An element is used in a mapping because it participates either in a correspondence or in a constraint (or both), or it is a join condition used in a user association, . In the relational world this is equivalent to attributes and relations that are used in the select clause of a view definition query or in the where clause of it as parts of a join path. We consider first the removal of atomic type elements.

First, we check whether the element to be removed is used in any of the user associations. An atomic element e := select e_{n+1} from x_0 in P_0, x_1 in P_1, ... x_n in P_n is used in a user association A if and only if there is a renaming function h from the variables of e to the variables of A and expression $f(e_{n+1})$ is used in a condition C of A. If the atomic element to be removed is used in an association A then every mapping that is using A is removed. Mapping m: foreach A^S exists A^T with D is using user association A if A is dominated by A^S.

Example 5.4 *If the* `SSN` *element is removed from the schema, then there is no way for a contact to be connected to person, so mapping m_3 that uses the user association between contacts and persons will be removed.* ∎

Once the mappings that depend on the used associations have been removed, the remaining mappings have to be adapted if they are affected by the removal. An atomic element e := select e_{n+1} from x_0 in P_0, x_1 in P_1, ... x_n in P_n is used in constraint F:=foreach X exists Y with C if there is a renaming function f from the variables of e to the variables of F and expression $f(e_{n+1})$ is used in the condition C of F. When atomic element e is to be removed each constraint F in which e is used is removed by following the procedure described in Section 5.1. Similarly, an atomic element e participates in a correspondence V := foreach P^S exists P^T with C if there is a renaming function g from the variables of e to the variables of V and $g(e_{n+1})$ is used in the condition C. If the atomic element e to be removed is used in a correspondence V then every mapping m that is covering V has to be adapted. More specifically, the equality condition in the with clause of the mapping that corresponds to V is removed from the mapping. If mapping m was covering only V, then the with clause of m becomes empty, thus m can be removed. If the atomic element e is used neither in a correspondence, nor in a constraint, it can be removed from the schema without affecting any of the existing mappings. Algorithm 5.3 describes the steps followed to remove an atomic element. To remove an element that is not atomic, its whole structure is visited in a bottom up fashion starting from the leaves and removing one element at a time following the procedure described above. A complex type element can be removed if all its attributes (children) have been removed.

Example 5.5 *In the mapping system of Figure 1 removing element* `topic` *from* `project` *will not affect any mappings since it is neither used in a constraint, nor in a correspondence. On the other hand, removing* `code` *will invalidate mapping* m_1 *that populates the target with project codes and the sponsor of privately funded projects. After the removal of* `code` *the element* `projects` *does not contribute with any data values to the population of the target schema. However, according to our algorithm the only modification that will take place in mapping* m_1 *will be the removal of the condition* i.`privProject.code`=p.`project.code` *from the* with *clause. This is due to the feature of our approach that tries to preserve the semantics of the initial mapping by performing the minimum required changes when adapting a mapping.* ∎

Algorithm 5.3 - Atomic Element Deletion

Input: Mapping System $<\mathcal{S}, \mathcal{T}, \mathcal{M}>$
Atomic element e
Body: Foreach user association A using e)
  M'={(foreach A^S exists A^T with D) | $A \dot{\preceq} A^S$}
  $\mathcal{M} \leftarrow \mathcal{M} - M'$
While exists constraint F that uses e
  remove F
$\forall$ $m \leftarrow$(foreach A^S exists A^T with D) $\in M$
  $D \leftarrow \{c \mid c$ is correspondence covered by m,
    c is not using $e\}$
  if D=$\emptyset$ remove m from M
Output: The updated set M

Another common operation in schema evolution is updating the type of an element e to a new type t. This case will not be considered seperately since it can be shown that this is equivalent to removing element e and then adding one of type t and with the same name with e.

5.3 Schema restructuring

One way a schema may evolve is by changing its structure without removing or adding elements. There are three common operations of this kind of evolution that we consider: rename, copy, and move. The first renames a schema element. The second operation moves a schema element to a different location while the third does the same but moves a replica of the element instead of the element itself. When an element is copied or moved, it is carrying with it design choices and semantics it had in its original location, i.e., schema constraints. Mapping selections and decisions that were applying in the original location, should also apply in the new one.

5.3.1 Renaming an element

Renaming an element is mainly a syntactic change. First the schema element name is updated. Then, each mapping is visited and every reference to the specific element name is also updated to the new name.

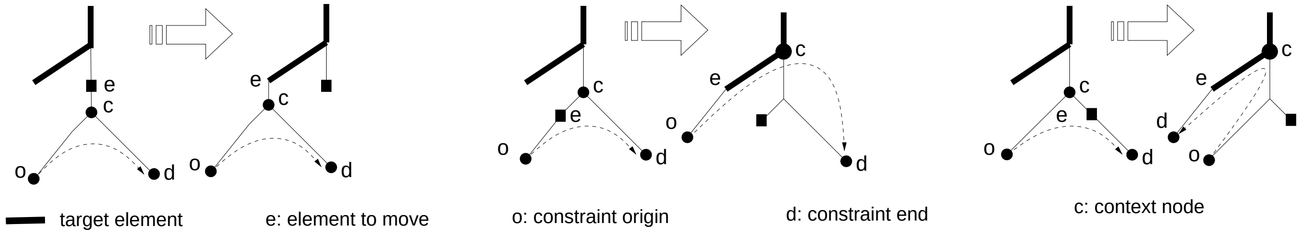


Figure 4: Updating constraint when element moves

5.3.2 Moving elements

Copying a part of a schema to another location is a 3 step process. During the first step, the schema is enhanced with the new structure which is a copy of the copied element. Then the schema constraints are adapted and finally the mappings.

Updating the schema structures Assume that element e:select e_{n+1} from x_0 in P_0, x_1 in P_1, ... x_n in P_n is to be moved to a new location, say under element p. If the type of p is a record (union) type, then a new entry is appended in that record (union) with the same name and type as e. If p is a set type, then the entry is appended to the record or union type that the set consists of (Recall that our model supports sets of Records and sets of Union types. No sets of sets are allowed).

Adapting schema constraints. Due to the move of element e, constraints that are using element e become invalid and must be adapted. A constraint F uses element e if there is a renaming from the variables of the path query P_e that identifies e to the variables of F. To adapt a constraint we have to bring it to the compact form (the one that has only one exists clause. To realize how F is affected by the change, we have to consider the relative position of e with respect to the context element of F. (In more technical terms, we have to consider where the image of the last variable of the path P_e, under the above mentioned renaming, is within F.) Recall that F has the form foreach P_0 [foreach P_1 exists P_2 with C] where the path queries P_1 and P_2 start from the last variable of the path query P_0 which represents the context node. Figure 4 provides a graphical explanation of how F has to be adapted to the move of the element e. In the figure, for constraint F, we use c, o, and d to denote (both before and after the move) the context element, the element identified by the last variable of path P_1, and the element identified bt the last variable of path P_2, respectively. If element e is an ancestor of the context node c, then the nodes c, o, and d move rigidly with e. The modified constraint will have the form foreach P_0' [foreach P_1 exists P_2 with C] where P_0' is the path to the new location of the context node c. If the context node is an ancestor of e then e is either used in P_1 or in P_2. Assume that it is used in P_1 (the other case is symmetric). This case is shown in the second part of the figure. Then the node o moves rigidly with e to a new location, while d remains in the same position. We then compute a new context node as the lowest common ancestor between the new location of o and d. The resulting constraint is then foreach P_0' [foreach P_1' exists P_2' with C'] where P_0' is the path to the new context node and P_1' and P_2' are the relative paths from P_0' to (the new location of) o and d. The condition C' must also be changed to use the last variables of P_1' and P_2'.

Example 5.6 *Assume that the schema owner of schema S in the mapping system of Figure 1 has decided to store the grants nested within each company so that each company keeps its own grants. This translates to a move of the element* `grants` *under the element* `company`. *Consider the constraint f_2 of example 3.5 specifying that each grant having a private sponsor refers to its contact information. Once the grants are moved, this constraint becomes inconsistent since there are no grant elements under the schema root* `S`. *To adapt the constraint, we use the previously described algorithm: we are in the second case shown in Figure 4, in which the element that moves is between the context element c (the root, in this case) and o. The element* `grants` *in the new location is expressed as:*

select a.company.grants from a in S.companies

The variable binding of a does not exist in f_2 so it is appended to it and every reference to expression `S.grants` *is replaced by the expression a.*`company.grants`. *The final form of the adapted constraint f_2 is shown below. (The path in the* exists *clause need not be changed, since the new context element continues to be the root.)*

foreach a in S.companies, g in a.company.grants,
p in g.grant.sponsor→private
exists c in S.contacts

with c.contact.cid=p

■

Adapting mappings. When an element is moved to a new location, some of the old logical associations that were using it become invalid and new ones have to be generated. To avoid redundant recomputations by regenerating every association, we exploit information given by existing mappings and computations that have already been performed. In particular, we first identify the mappings that need to adapt by checking whether the element that is moved is used in any of the two associations on which the mapping is based. Let A be an association that is using the element e that is about to move, and let t be the element in its new location. More precisely, assume that e and t have the following forms:

e = select e_{n+1} from x_0 in e_0, x_1 in $e_1, ..., x_n$ in e_n

t = select t_{m+1} from y_0 in t_0, y_1 in $t_1, ..., y_m$ in t_m

We first identify and isolate the element e from association A, by finding the appropriate renaming from the from clause of e to A. For simplicity, assume that this renaming is the identity function, that is, A contains literally the from clause of e. In the next step, the from clause of t is inserted in the front of the from clause of A. We then find all *usages* of e_{n+1} within A, and replace them with t_{m+1}. After these replacements, it may be the case that some (or all) of the variables $x_0, \dots, x_n$ have become redundant (i.e. not used) in the association. We eliminate all such redundant variables. Let us denote by A' the resulting association.

Since the element t in the new location may participate in its own relationships (based on constraints) with other elements, those elements have to be included as well in the new adapted version of association A'. We do this by chasing A' with the schema constraints. The chase may produce multiple associations $A'_1, \dots, A'_k$ (due to the choice types). Finally, any mapping using the old association A, say foreach A exists B with D, is removed from the list of mappings and is replaced with a number of mappings m_i : foreach A'_i exists B with D' one for each association A'_i. The conditions D' correspond to the correspondences in D *plus* any additional correspondences that may be covered by the pair $<A'_i, B>$ (but not by the original pair $< A, B >$).

As an important consequence of our algorithm, all the joins that were in use by the original mapping and that are still well-formed are still used, unchanged, by the new, adapted, mapping. Hence, we preserve to the best any design choices that might have been made by a human user based on the original schemas. We illustrate the adaption algorithm with the following example.

Example 5.7 *Assume that in the mapping system of Figure 1* grants *are moved under* company *as in Example 5.6. This change affect neither mapping m_3, nor mapping m_2. (Recall from Section 5.2 that just the addition of new structure* (grants*, in this case) for m_2 does not require m_2 to adapt.) However, mapping m_1, based on the logical association A_1 (see Figure 3), is affected. First, schema constraints adapt as described in Example 5.6. Then we run the mapping adaption algorithm described above, for $e =$* select S.grants from _ *and $t =$* select o.company.$grants$ from o in S.companies *(we denote here by _ the empty* from *clause). The clause o* in S.companies *is added in the* from *clause of A_1. Next, all occurrences of* S.grants *are replaced by o*.company.grants*. After this, the resulting association is chased with the source schema constraints. The (adapted) constraints f_1, f_2, and f_3 are already satisfied, and hence not be applicable. However, f_4 and f_5 will be applied. The chase makes then the two correspondences on* cname *and* name *to be also covered, the last one in two different ways. Hence, two new mappings are generated. The first is:*

```
m'_1a: foreach o in S.companies, p in S.projects,
               g in o.company.grants,
               r in g.grant.sponsor→private,
               c in S.contacts, e  in S.persons,
         where p.project.source=g.grant.gid and
               r=c.contact.cid and
               o.company.CEO=e.person.SSN
       exists  j in T.privProjects, m in T.companies
         where j.privProject.holder=m.company.cname
       with    m.company.cname=o.company.cname and
               m.company.leader=e.person.name and
               j.privProject.code=p.project.code and
               j.privProject.sponsor=c.contact.email
```

while the second is the one that considers the owner of the company as a leader instead of the CEO. Note in the above query how our algorithm preserved the choices of mapping m_1 on considering only private projects, and how the relationships

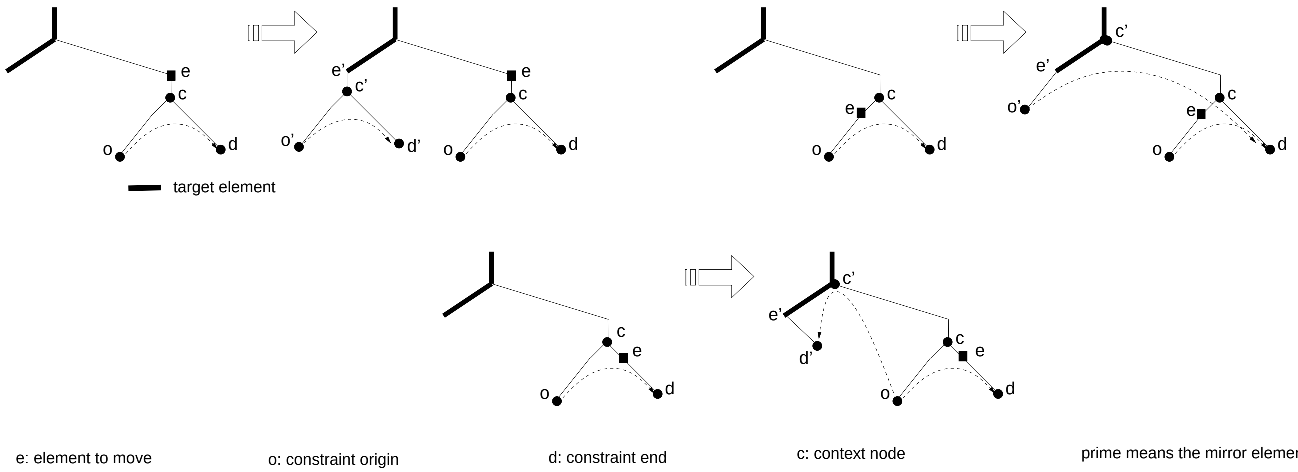


Figure 5: Updating constraint when an element is copied

between the `projects` *and* `grants`*, as well as the* `grants` *and* `contacts` *that were in the initial m_1 were also preserved in the generated mapping.* ∎

5.3.3 Copying elements

In the above analysis we considered the case of moving an element from one place of the schema to another. In the case that the element is copied instead of moving, the same reasoning takes place and the same steps are executed. The only difference is that the affected mappings and constraints are not removed from the mapping system as in the case of a move. Schema constraints and mapping choices that have been made, continue to hold unaffected after a structure of the schema is copied. Figure 5 shows how the schema constraints are updated when an element is copied, depending on the relative position of the element and the elements involved in the constraint. Copying an element is a frequent operator, when for examle a relation is to be split horizontally in two. It is not hars to see that constraints that hold on the tuples of the initial relations will keep holding on the tuples of both halfs of the relations that it has been split.

6 Mapping and Constraint Optimization

During the process of updating the mappings, some redundency may be introduced. Redundency is realized as a set of variable bindings that their presence does not change the semantics of the mapping. Mappings can be optimized by detecting and removing those binding.

A variable bound to the elements of a set type is redundant if it is not used neither in a condition of an association of the mapping, nor in a correspondence. A variable bound to a union type selection is required, if and only if it is based on a schema root, or, on a variable that is also required. By the term "based", we mean the variable with which the bounded expression starts. According to the above observations, we have the following algorithm to optimise a mapping. The algorithm requires two passes over the mapping. During the first, it characterizes variables that are required because an expression is based on them, while during the second, it characterizes variables bounded to union choices that are required. The process is described analytically in Algorithm 6.1.

Example 6.1 *In mapping m_u of Example 4.14 the variable p' is used neither in the* where *conditions nor in the* with*, and no other variable is based on it, hence it is redundant and can be eliminated. The resulted equivalent mapping is the m_o shown in the same example.* ∎

The same optimization algorithm can be used to turn a schema constraint from its alalytic to the compact representation. This is based on the fact that constraints and mappings are specialized forms of a dependency. A mapping can be seen as an interschema constraint while an intraschema constraint as a special form of a mapping. Thus, given a constraint in its analytic representation, we can take one of the where clauses (any one would qualify) and apply Algorithm 6.1. The result will be compact representation of the constraint.

Algorithm 6.1 - Mapping Optimization

Input: Mapping M: foreach X exists Y with C
Optimized Mapping M
Body: boolean[ ] isNeeded;
Initialize every element of isNeeded to false;
// First pass
For eachvariable x of X used in C
    isNeeded[x] = true
For eachvariable x of X used in the conditions of X
    isNeeded[x] = true
For i=NumOfVars in X..0
    $e \leftarrow$ expression on which variable x_i is bound
    if (isNeeded[i] == false) continue
    $v_k \leftarrow$ the variable with which expression e starts (if any)
    isNeeded[k] = true
// Second Pass
For i=0..NumOfVars in X
    if (isNeeded[i] == true) continue
    $e \leftarrow$ expression on which variable x_i is bound
    if (e does not represents a union type choice) continue
    $v_k \leftarrow$ the variable with which expression e starts (if any)
    if (isNeeded[k] == true) isNeeded[i] = true
// Cleaning redundant variables
For i=0..NumOfVars in X
    if (isNeeded[i] == false) remove x_i binding from X.
Output: The updated mapping M

7 A Ranking Mechanism

In the previous sections we presented a methodology for adapting mappings that are affected by a change on the schema structure or constraints. The presented methodology detects the mappings that are affected by the change and generates deterministically a number of semantically valid rewritings. All those rewritings are consistent with the semantics of the schemas, the structure, and also the choices the have been done in the past and are encoded in the previous mappings. An affected mapping can be replaced with one, two, or even all of its generated rewritings. The existence of more than one candidates is natural since the new modified schema has new semantics. It wouldn't have been wrong of all the generated mappings would have been used. However, there are cases in which not all of them describe the semantics of the intended data translation between the two schemas. If only a part of the generated mappings is to be used, this would have require human intervention. This case raises a need for a model that will help the user in taking the right decisions. The basic principle of our approach is to find candidate mappings that preserve as much as possible the semantics of the existing mappings. Based on this we introduce a model for systematically ranking the various mappings in decreasing order.

A number of approaches in the literature [CJR98, MD96] have been using the cost of updating a materialized target as a measurement of the importance of the mapping. Although the cost of updating is an important factor, we think that maintaining the semantics of the mapping is the most important factor that should be taken into consideration. Once the replacement mapping has been decided, then various methods can be investigated to minimize the cost of the update of the materialized target.

Each semantically valid mapping that is produced by our algorithm preserves a different number of elements of the original mapping and may introduce new ones. We introduce a factor that measure how semantically close is a mapping to another.

Definition 7.1 *Let $|m|$ represent the number of schema elements and conditions between those elements in a mapping m. The* relative similarity *of mapping m_1 to mapping m_2 (noted as $\mathcal{S}(m_1, m_2)$) is defined as the weighted sum:*

$$\mathcal{S}(m_1, m_2) = \rho_1 \frac{|m_1 \cap m_2|}{|m_2|} + \rho_2 \frac{|m_1 - m_2|}{|m_1|} \tag{1}$$

where $|m_1 \cap m_2|$ are the number of common attributes and conditions of mappings m_1 and m_2, $|m_1 - m_2|$ is the number of attributes and conditions that are present in m_1 but not in m_2, and $\rho_1 + \rho_2 = 1$

Intuitively, the first fraction of the *relative similarity* $\mathcal{S}(m_1, m_2)$ measures how many elements mapping m_1 has in common with mapping m_2 while the second how many surplus elements mapping m_1 has compared to mapping m_2. It is clear from this that the relative similarity is not a symmetric relationship.

Most of the time, the relative similarity is not enough to determine if a mapping is more preferable than another. Consider for example the case that a new constraint is added in one of the schemas. An association that is affected by the change is chased with the new set of constraints, and may result to more than one associations. Which one of those is more preferable? All of them have the same number of common elements with the initial association and may have the same number of surplus elements. In this case the relative similarity will be the same. However, each of the resulted associations has different conditions. We can use information from other mappings to decide which one is preferable by measuring how often those extra conditions are met in the remaining unaffected mappings.

Definition 7.2 *Let M be a set of mappings. The* level of support *of mapping m from the set M is defined as:*

$$\mathcal{L}_M(m) = \frac{\Sigma_{m' \in M} \mathcal{S}(m, m')}{|M|} \tag{2}$$

Example 7.3 *Consider the two mappings m'_{1a} and m'_{1b} in Example 4.14, that were generated as a result of the addition of a new constraint. Recall that the difference between those two mappings is that the first one considers the CEO as a leader of the company while the second considers the owner. The relative similarity of those mappings to mapping m_1 is the same. However, it can be seen that in mapping m_2 the leader of the company is considered to be the owner. This makes mapping m'_{1b} a better candidate compared to m'_{1a}. Indeed, the level of support of mapping m'_{1b} is higher than the one of level m'_{1a}.* ∎

8 Mapping adaptation experience

To evaluate the effectiveness and usefulness of our approach, we have implemented a prototype tool called ToMAS[1] and we have successfully applied it to a variety real application scenarios. In this section we describe the architecture of ToMAS and the experiments that we have conducted to evaluate the tool. Through the experiments we investigated the usability and the effectiveness of our tool. We report our experience using ToMAS with a set of real schemas (both relational and XML) that are available on the Web. The results of this study is that indeed it is worth using a mapping adaptation system like ToMAS to automatically maintain the mappings between schemas rather than running the mapping process from scratch as schema evolve. Specifically, we show that (i) the time needed for incrementally updating the mappings under schema changes is negligible, and (ii) this incremental adaptation requires much less effort than the from scratch rebuilt of the mappings. Finally, we highlight the benefits of using a mapping adaptation tool in concert with a physical design tool that manipulates schemas. In particular, we show how our tool can facilitate the selection of a good relational schema for storing XML data.

8.1 Architecture

ToMAS follows a modular architecture as can be seen in Figure 6. At the heart of ToMAS is a schema evolution engine that contains implementations for each of the evolution operators we described earlier. ToMAS manages a mapping system (a pair of schemas and a set of mappings between them). As the schemas evolve, ToMAS detects mappings that may need to be updated and based on the schema constraints and the user associations, it creates a set of potential rewritings that are consistent with the modified schemas.

To update the mappings, apart from the schema and constraint information, the evolution engine requires to know the user associations that are contained in the mappings. This information is provided by the *Mapping Analyzer*. The Mapping Analyzed examines the existing mappings and for each equality condition between elements of the same schema, it investigates whether this condition is based on a schema constraint or not. If it is not based on a schema constraint, then it means that it represents a user's choice so a user association is generated. This step is taking place at the beginning of

[1] ToMAS stands for **<u>To</u>**ronto **<u>M</u>**apping **<u>A</u>**daptation **<u>S</u>**ystem

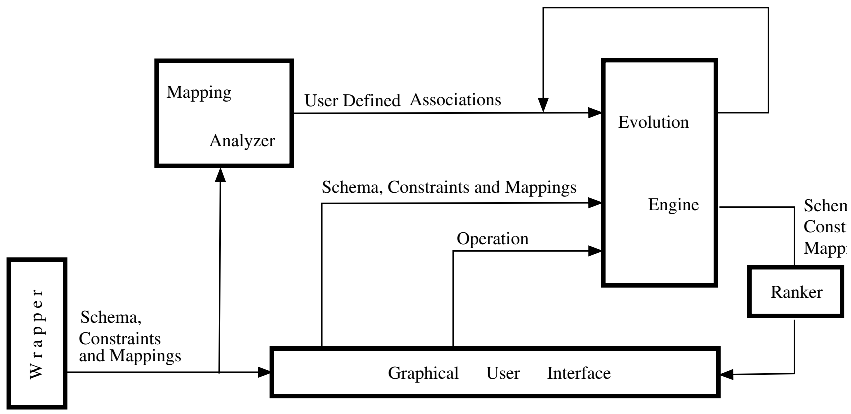


Figure 6: ToMAS Architecture

Schema	Size	Constraints	Correspondences	Possible Mappings
ProjectGrants	16	6	6	7
DBLP	88	0	6	12
TPC-H	51	10	10	9
Mondial	159	15	15	60
GeneX	88	9	33	2

Table 1: Test schemas characteristics

the evolution process, right after the schemas are loaded. After that, the list of user associations is then updated by the evolution engine after each schema change.

The system is equipped with a set of pluggable schema wrappers that are used to handle the discrepancies and import schemas and mappings from different data models into the internal nested relational representation. Currently, relational and XML wrappers have been implemented.

The schemas and the mappings are presented to the user through a graphical user interface. Through this interface, the user may also modify the schemas. For each modification, the schemas, the requested modification and the existing mappings are provided to the evolution engine that updated both the mappings and the schemas. The results are returned back to the interface for presentation to the user and further modifications. Figure 7 shows a snapshot of the ToMAS graphical user interface.

An important module of the system is the mapping ranker that implements the ranking mechanism we presented in Section 7. The ranker accepts as input the new replacement mappings that were generated by the evolution engine and ranks them based on their semantic similarity and the level of support. The result are then presented to the user through the interface in descending order.

8.2 Performance

We now set out to investigate the efficiency of our proposed incremental mapping maintenance algorithms. We conducted a series of experiments on some well-known schemas that are available on the web and vary in terms of size and complexity. We have made them available on our web page[2] which also contains links to their original sources. We took two versions of each schema and we used the Clio tool to generate an initial set of mappings from the first to the second using a number of correspondences between atomic schema elements. The characteristics of each schema are summarized in Table 1. The second column of the table shows the size of each schema in terms of number of elements and within the brackets is the number of constraints contained in the schemas. The third columns indicates the number of correspondences that were used. From the set of mappings that Clio generated, we selected a subset as the set describing the semantics of the correspondences. The fourth column of the table indicates the number of mappings we selected and the number in the brackets the sum of the number of joins and nesting depth of the queries in the mappings as an indication of their size.

[2] http://www.cs.toronto.edu/db/clio/schemas.html

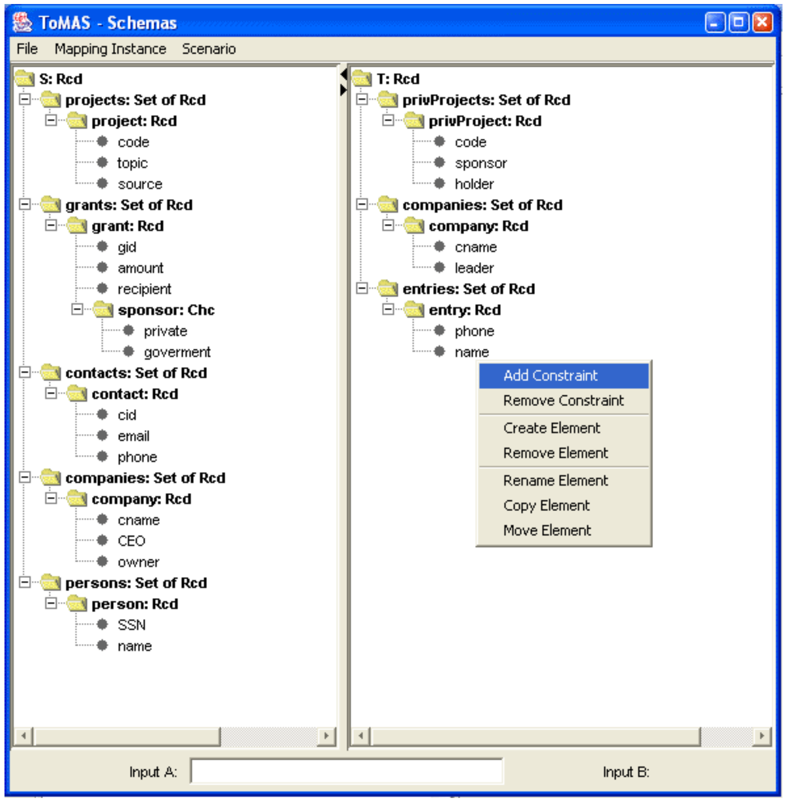


Figure 7: ToMAS User Interface

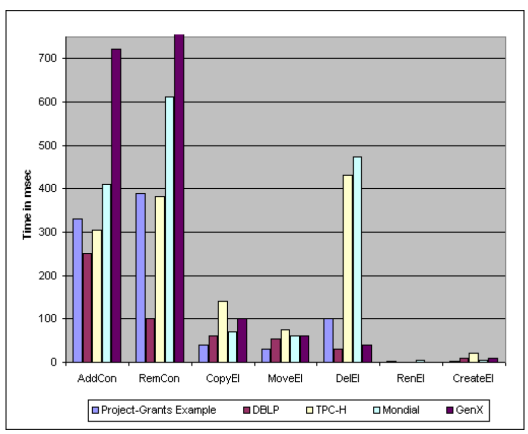


Figure 8: Average time of mapping adaptation for various schema changes

A number of random sequences of 15 to 20 schema changes ware generated and applied to the schemas of the mapping system. For each change we measure the time that is needed for the schema to be updated and the mappings to be adapted. Figure 8 summaries the results of this experiment. The time indicated for each operator is the average time of this kind of operator in the mapping system. What was noticed by observing the analytic results is that the time of each operator varies depending not only on the nature of the schemas and the mappings but also on the sequence of changes that happened before. However, the average time that has been calculated and is shown in Figure 8 gives a very good idea of the expected time of completion of the adaptation procedure after each schema change.

As a general observation, the required time in most of the cases is less than a second. The time required for the rename operator is almost 0. This is due to the fact that no mappings need to be involved in the adaptation process. When mappings are loaded into memory, they are first type-checked. The type-checking replaces each reference to an element name with a pointer to the schema structure of the corresponding element. This means that renaming needs only to change the name of the element in the schema and nothing else. Hence, the whole time needed for the completion of the renaming operation is the time to locate the affected element in the schema which is really small.

Creation of new elements in the schema neither requires any update on the mappings. However, the inserted structure has to be type-checked for consistency before inserted in the schema. This checking justifies the slightly extra time of the create element operation compared to renaming.

Copying and moving of an element are mostly syntactic modifications. They require deletion of the elements to be moved (copied) in a mapping or in a constraint and update accordingly. At the end of the move (copy) there is some chasing taking place at the new location. This chasing is basically responsible for the time difference between those two restructuring operators and the element renaming or creation. The fact that moving simply modified some of the existing structured while copying is appending new, justified the small performance difference between copy and move. TPC-H is the schema with the greater number of constraints (compared to the size of the schema) which makes chasing taking longer than any other schema. On the other hand, if we exclude the ProjectGrant schema that is really small, the shortest adaptation time is given by DBLP which is the only schema that has no constraints.

Addition or removal of constraints, as expected, are the most expensive operations since they involve chasing. Adding a new constraint requires to chase an affected association, while removal involves the reconstruction of a number of structural associations, their chasing, and the checking of whether the results of the chasing are subsumed by the affected association. All those operations make mapping adaptation under constraint removal to be more expensive than the adaptation under constraint addition. The surprising result is the DBLP schema in which every test we have conducted the constraint removal outperformed the constraint addition. This may be happening because DBLP is the only schema with no constraints at the beginning of the experiment. Constraints that are removed, are those having been added that are not creating long chains of consequentive constraints. As a consequence, chasing was never having more than 1 chase steps. Furthermore, DBLP is very swallow, so the time required for the construction of the structural associations was almost negligible. An interesting observation is the performance of the GeneX schema under constraint removal, which is the only one that took more than one second even though there was only one mapping. The reason is that the mapping was really large, involving 29 joins, hence, the corresponding association had 29 bound variables. Homomorphism becomes a really expensive operation, especially under such big associations.

Deletion of an element requires first the removal of the correspondences and the constraints that are using the specific element. This means that the cost of an element deletion is at least as much as the constraint removal. On the other hand, if the element is not used by any constraint, it can be removed in time that is almost equal to the time of renaming. In average, the first kind of removal operations is balanced by the low cost of the second kind, hence the average performance is the one shown in Figure 8.

The above analysis shows that despite the size and the complexity of the schemas, the time for mapping adaptation is so short that permits its use both in interactive applications or automatic mapping adaptation.

8.3 Benefits

There are two ways mappings can be adapted after schemas have evolved. One is to use a mapping tool, like Clio, and generate the mappings from scratch on the updated new schema versions. On the other hand, one could use a tool like ToMAS to incrementally maintain the mappings every time a schema change occurs. In this section we tried to investigate which approach and under what conditions is preferable.

To be able to compare the two approaches, we had to create a cost model. The main goal of schema and mapping management tools are to relief the user from the effort of manually creating and maintaining the mappings. For that reason, as a cost model, we considered the number of actions a user has to do in each time. In the case of ToMAS, when the mappings have been adapted, the user has only to verify that the adapted mappings resulted by the system are describing indeed the intended semantics of the user. In the case where a mapping tool had to be used to define the mappings from scratch, the user had first of all to draw the correspondences between the two schema elements, then let the mapping tool to generate the mappings and finally browse those generated mappings and select those that describe the intended semantics of the correspondences. More specifically we considered the following factor as a way to measure the advantage of ToMAS approach over the manual and from scratch creation of the mappings:

$$\textit{ToMAS Advantage } \mathbf{TA} = 1 - \frac{\#\ of\ mappings\ generated\ by\ ToMAS}{\#\ of\ mappings\ generated\ by\ Clio + \#\ of\ correspondences}$$

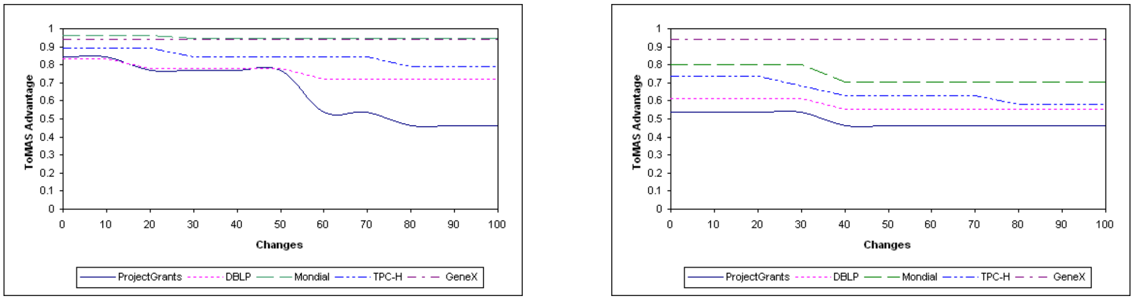


Figure 9: Benefit of ToMAS use compared to the from-scratch mapping regeneration

To investigate the usability of our tool we conducted a series of experiments on the schemas of Table **??**. We started by a number of mappings between two versions of the same schema and we performed a random number of schema changes. After each change we were measuring the *ToMAS Advantage* factor. Since the number of changed could not be the same for all the schemas, we normalized them in a scale from 0 to 100 to be able to represent them all in the same graph. Given the number of correspondences given in the fourth column of Table **??** there are numerous possible valid mappings as the numbers in the fifth column of the same table indicates. We conducted two kinds of experiments. In the first we considered only two mappings and we investigated how ToMAS maintains them under changes. In the second we repeated the same procedure but considering half of the semantically valid mappings. The results of those experiments are summarized in Figure 9.

By looking at the charts of Figure 9 one can make various observations regarding ToMAS. First of all, it can be seen that as the number of changes becomes bigger, the benefit of using ToMAS is decreasing. The reason is that the more the changes are, the less the relationship to the initial mappings becomes. ToMAS advantage is that it can maintain the semantics of the existing schema and mappings. While the changes become more, the semantic distance between the initial schema and the modified one becomes larger and it is harder for ToMAS to maintain the mappings. This is translated to the introduction of new mappings that were not there before. When the number of changed has increased enough the line will eventually come very close to 0, without ever reaching it. This can be explained by the definition formula of *ToMAS Advantage* above. Even when the mappings generated by ToMAS are all the possible valid mappings, the from-scratch approach will have the small overhead of re-specifying the correspondences, while in ToMAS those can be extracted from the initial mappings. Regarding the decreasing rate, no specific conclusions can be reached. The experienced showed that it is heavily dependent not only on the kind of schemas and mappings but on the sequence of changes that is applied as well.

Comparing the two graphs of Figure 9, it can e seen that when the percentage of the set of all the possible valid mappings that is to be maintained is large, then the advantage of ToMAS is smaller. The important benefit from ToMAS use, is that it can maintain mapping decision that had been taken during the selection of the initial mappings. Those decisions are detected and are used by the tool to identify among all those possible valid mappings, those that satisfy those decisions. Otherwise, the user will have to go through all the mappings and manually detect all of them. This advantage is getting lost while the percentage of the mappings that are considered decreases. If, for example, all the possible valid mappings are to be maintained, then the advantage from the use of ToMAS is really small.

However, in real practical situations, ToMAS is proved to be a very useful tool for two reasons: First because the mappings that are to be maintained are a small portion of the complete set of valid mappings, and second, because the changes that are to be applied to the schemas are not many. When schemas evolve, they do not change radically. The reason is that the goal of the change is to make the schema to better reflect the semantics of the data, but the initial design decisions keep being applied. "When schemas are changing it is usually a schema evolution, not a revolution." [Ler00].

8.4 Case Study: The use of ToMAS in the field of relational XML storage

We present our experience using ToMAS within one important application: physical data design. In the last few years we have noticed a growing interest for storing XML data in relational database systems in order to be able to re-use their well-developed features (e.g. concurrency control, query processing, etc.). A number of approaches have been proposed to tackle with the mismatch between the nested semi-structured nature of the XML data and the relational model [FK99]. Unfortunately, no approach is universally accepted since none has been found to perform well in all the cases. LegoDB [BFH$^+$02]

TABLE Show	**TABLE Show1**	**TABLE NYTRev**
ShowId,	ShowPart1Id,	ReviewsId,
type,	type,	review,
title,	title,	parentShow
year,	year,	**TABLE Reviews**
boxOffice,	boxOffice,	ReviewsId,
videoSales,	videoSales	review,
seasons,	**TABLE Show2**	parentShow
description	ShowPart2Id,	**TABLE Episode**
TABLE Review	type,	EpisodeId,
ReviewsId,	title,	name,
tilde,	year,	parentShow
reviews,	seasons,	
parentShow	description	
TABLE Episode		
EpisodeId,		
name,		
parentShow		
(a)	**(b)**	

Figure 10: Two relational designs for the IMDB DTD

is a physical database design tool for designing (optimized) relational storage structures for XML data. LegoDB helps a user to evaluate some of the many XML-to-relational "shredding" options [FK99]. Since different XML-to-relational translations are best for different work loads and data characteristics, LegoDB provides an automated wizard for finding good relational designs. For this case study, we use the Internet Movie Database DTD example described in [BFH$^+$02].

<!ELEMENT imdb (show*, director*, actor*)>
<!ELEMENT show (title, year, reviews*,
((boxOffice, videoSales)|(seasons, description, episode*)))>

The two relational schemas of Figure 10 represent two different shredding methods for the above DTD. Mappings between each of these two schemas and the DTD might be output by a design tool or might be created with a mapping tool like Clio. Although this is a very simple example, it is important to note that the mappings from Figure 10(a) to the DTD is complicated enough. Figure 11 indicates the specific mapping in XQuery form. Normally the mapping should have been in a relational query language (e.g. SQL) since the source schema is a relational schema, but we present it as an XQuery since the nesting structures are not easily presented in SQL. The query is in such a form that is easily and straight forward translated to a series of consequentive SQL queries. The figure indicates the complexity of the mapping that has eight joins between the three participating tables of the source schema and a nesting depth three. So even for this simple example, generating the correct mapping is clearly hard if it is to be done manually.

In tools like LegoDB, each shredding method is accompanied by the corresponding mapping. For the second shredding method of Figure 10(b), for example, the mapping to the DTD has to be generated in advanced and hard-wired in the LegoDB cost engine. Manual generation of such mapping, or generation through a mapping tool can be avoided with the use of ToMAS. Using ToMAS, one can take the DTD, the first shredding method, and the generated mapping of Figure 11 and start modifying the schema to bring it in the form of Figure 10(b). Throughout this process, ToMAS will be maintaining the mappings, and at the end, it will be able to provide automatically the mapping from the relational tables of the new shredding method to the DTD.

We have performed the above test. After most operations, the rewriting could be automatically updated without user intervention, but for some operations there was a choice of what semantics to use. During the entire evolution, the ToMAS user had to make very few choices and we were able to verify that the resulted mapping is the one we would have created if we were about to be done automatically, or using a mapping tool like Clio.

Notice that our approach permits design tools like LegoDB to explore new storage schemes that might not be part of their (predefined) search space of designs for efficiency reasons. For example, using ToMAS we can permit a designer to suggest a different, *ad hoc*, vertical or horizontal decomposition of the relations (one not suggested by the workload). If the cost-based engine of LegoDB selects such a user provided design, ToMAS can generate the mapping needed to use the original XML Schema as a view over this design.

```
<imdb>
  FOR $x0 IN $doc/DB/Review, $x1 IN $doc/DB/Show
  WHERE $x0/ReviewsId/text() = $x1/ShowId/text()
  RETURN
    <Show>
      <type> $x1/type/text() </type>
      <title> $x1/title/text() </title>
      <year> $x1/year/text() </year>
      FOR $x0L1 IN $doc/DB/Review, $x1L1 IN $doc/DB/Show
      WHERE
        $x0L1/ReviewsId/text() = $x1L1/ShowId/text() AND
        $x1/videoSales/text() = $x1L1/videoSales/text() AND $x1/boxOffice/text() = $x1L1/boxOffice/text() AND
        $x1/type/text() = $x1L1/type/text() AND $x1/title/text() = $x1L1/title/text() AND
        $x1/year/text() = $x1L1/year/text()
      RETURN
        <review> $x0L1/reviews/text() </review>
      <boxOffice> $x1/boxOffice/text() </boxOffice>
      <videoSales> $x1/videoSales/text() </videoSales>
    </Show>
  FOR $x0 IN $doc/DB/Episode, $x1 IN $doc/DB/Show, $x2 IN $doc/DB/Review
  WHERE $x0/EpisodeId/text() = $x1/ShowId/text() AND $x2/ReviewsId/text() = $x1/ShowId/text()
  RETURN
    <Show>
      <type> $x1/type/text() </type>
      <title> $x1/title/text() </title>
      <year> $x1/year/text() </year>
      FOR $x0L1 IN $doc/DB/Episode, $x1L1 IN $doc/DB/Show, $x2L1 IN $doc/DB/Review
      WHERE
        $x0L1/EpisodeId/text() = $x1L1/ShowId/text() AND $x2L1/ReviewsId/text() = $x1L1/ShowId/text() AND
        $x1/seasons/text() = $x1L1/seasons/text() AND $x1/description/text() = $x1L1/description/text() AND
        $x1/type/text() = $x1L1/type/text() AND $x1/title/text() = $x1L1/title/text() AND
        $x1/year/text() = $x1L1/year/text()
      RETURN
        <review> $x2L1/reviews/text() </review>
      <seasons> $x1/seasons/text() </seasons>
      <description> $x1/description/text() </description>
      FOR $x0L1 IN $doc/DB/Episode, $x1L1 IN $doc/DB/Show, $x2L1 IN $doc/DB/Review
      WHERE
        $x0L1/EpisodeId/text() = $x1L1/ShowId/text() AND $x2L1/ReviewsId/text() = $x1L1/ShowId/text() AND
        $x1/seasons/text() = $x1L1/seasons/text() AND $x1/description/text() = $x1L1/description/text() AND
        $x1/type/text() = $x1L1/type/text() AND $x1/title/text() = $x1L1/title/text() AND
        $x1/year/text() = $x1L1/year/text()
      RETURN
        <episode> $x0L1/name/text() </episode>
    </Show>
  </imdb>
```

Figure 11: An initial mapping for the IMDB

Additionally, our ability to transform the relational schema of Figure 10(a) to the one of Figure 10(b) indicates the kind of transformations we are supporting. We do not consider only simple structural changes that take place locally on a table or on a class. We can have complex schema modifications involving more than one schema structures (in the specific example relations) like copying an attribute from one table to another. In short, we have found that with our small set of primitive operators we can support the majority of compound schema changes that exists in the literature [Ler00].

9 Conclusion

In this paper, we identified the problem of mapping adaptation in dynamic environments with evolving schemas. We motivated the need for an automated system to adapt mappings and we described several areas in which our solutions can be applied. We presented a novel framework and tool that automatically maintains the consistency of the mappings as schemas evolve. Our approach is unique in many ways. First, we consider and manage a very general class of mappings including Global-or-Local-As-View [Len02] mappings. This class includes the mappings used in a large variety of applications from data integration [Len02] to physical data design [TSI96] to web source descriptions [LRO96]. Second, our approach is the first to consider both source and target schema changes in concert. Third, our algorithms adapt mappings in response not only to changes on the schema structure but also to changes in the schema semantics (i.e., constraint modifications on either the source or target). We explicitly model mapping choices made by a user and maintain these choices. Finally, we support complex schema changes involving many schema elements (e.g., moving an attribute or subtree from one type to another).

References

[BCD92] F. Bancilhon, S. Cluet, and C. Delobel. *A Query language for O_2*, chapter 11. Morgan Kaufman, 1992.

[BFH+02] P. Bohannon, J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. LegoDB: Customizing Relational Storage for XML Documents. In *VLDB*, pages 1091–1094, 2002.

[BHL83] E. Bertino, L. M. Haas, and B. G. Lindsay. View management in distributed data base systems. In *VLDB*, pages 376–378, 1983.

[BKKK87] J. Banerjee, W. Kim, H. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *SIGMOD*, pages 311–322, May 1987.

[BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.

[BLT86] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *SIGMOD*, pages 61–71, May 1986.

[CCGL02a] A. Cali, D. Calvanese, G. De Giacomo, and M. Lenzerini. Data Integration Under Integrity Constraints. In *CAiSE*, pages 262–279, 2002.

[CCGL02b] A. Cali, D. Calvanese, G. De Giacomo, and M. Lenzerini. On the Role of Integrity Constraints in Data Integration. *DEBul*, 25(3):41–44, September 2002.

[CJR98] K. T. Claypool, J. Jin, and E. A. Rundensteiner. SERF: Schema Evolution through an Extensible Re-usable and Flexible Framework. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM-98)*, pages 314–321, November 3–7 1998.

[CW91] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *VLDB*, pages 277–289, September 1991.

[DT01] Alin Deutsch and Val Tannen. Optimization properties for classes of conjunctive regular path queries. In *DBPL'01*, 2001.

[FK99] D. Florescu and D. Kossmann. Storing and querying xml data using an rdmbs. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[FKMP03] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. In *Proceedings of International Conference of Database Theory (ICDT)*, pages 207–224, 2003.

[FKP03] Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. Data exchange: Getting to the core. In *PODS*, 2003. To appear.

[FLM99] M. Friedman, A. Levy, and T. Millstein. Navigational Plans for Data Integration. In *AAAI*, pages 67–73, 1999.

[GLS95] M. Gyssens, L. Lakshmanam, and I. N. Subramanian. Tables as a Paradigm for Querying and Restructuring. In *PODS*, pages 93–103, 1995.

[GM99] Gøsta Grahne and Alberto O. Mendelzon. Tableau Techniques for Querying Information Sources through Global Schemas. In *Proceedings of International Conference of Database Theory (ICDT)*, pages 332–347, 1999.

[GMR95a] A. Gupta, I. Mumick, and K. Ross. Adapting Materialized Views After Redefinition. In *SIGMOD*, pages 211–222, 1995.

[GMR95b] A. Gupta, I. Singh Mumick, and K. A. Ross. Adapting Materialized Views after Redefinitions. In *SIGMOD*, pages 211–222, 1995.

[KR99] Y. Kotidis and N. Roussopoulos. DynaMat: A Dynamic View Management System for Data Warehouses. In *SIGMOD*, pages 371–382, 1999.

[KR01] Y. Kotidis and N. Roussopoulos. A case for dynamic view management. *ACM TODS*, (4):388–423, 2001.

[Len02] M. Lenzerini. Data Integration: A Theoretical Perspective. In *PODS*, pages 233–246, 2002.

[Ler00] B. S. Lerner. A Model for Compound Type Changes Encountered in Schema Evolution. *ACM TODS*, 25(1):83–127, March 2000.

[LNR02] A. J. Lee, A. Nica, and E. A. Rundensteiner. The EVE Approach: View Synchronization in Dynamic Distributed Environments. *TKDE*, 14(5):931–954, 2002.

[LRO96] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *VLDB*, pages 251–262, 1996.

[MD96] M. K. Mohania and G. Dong. Algorithms for Adapting Materialised Views in Data Warehouses. In *Proceedings of the International Symposium on Cooperative Database Systems for Advanced Applications*, pages 309–316, December 1996.

[MMS79] D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing implications of data dependencies. *ACM TODS*, 4(4):455–469, 1979.

[MP02] P. McBrien and A. Poulovassilis. Schema Evolution in Heterogeneous Database Architectures, A Schema Transformation Approach. In *CAiSE*, pages 484–499, 2002.

[MQM97] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. In *SIGMOD*, pages 100–111, May 13–15 1997.

[PT99] L. Popa and V. Tannen. An Equational Chase for Path-Conjunctive Queries, Constraints, and Views. In *Proceedings of International Conference of Database Theory (ICDT)*, pages 39–57, 1999.

[PVM+02a] L. Popa, Y. Velegrakis, R. J. Miller, M. Hernández, and R. Fagin. Translating Web Data. Technical Report CSRG-441, Un. of Toronto, Dep of Comp. Sc., February 2002. ftp://ftp.cs.toronto.edu/cs/ftp/pub/reports/csri/441.

[PVM+02b] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernandez, and R. Fagin. Translating Web Data. In *VLDB*, pages 598–609, August 2002.

[RB01] E. Rahm and P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal*, 10(4):334–350, 2001.

[RR99] S. Ram and V. Ramesh. Schema Integration: Past, Current and Future. In *Management of Heterogeneous and Autonomous Database Systems*, pages 119–155. 1999.

[SP94] S. Spaccapietra and C. Parent. View Integration : A Step Forward in Solving Structural Conflicts. *TKDE*, 6(2):258–274, 1994.

[TSI96] O. Tsatalos, M. Solomon, and Y. Ioannidis. The GMAP: A Versatile Tool for Physical Data Independence. *VLDB Journal*, 5(2), April 1996.

[VP97] Vasilis Vassalos and Yannis Papakonstantinou. Describing and using query capabilities of heterogeneous sources. In *VLDB*, pages 256–265, 1997.

[Wid95] J. Widom. Research Problems in Data Warehousing. In *4th International Conference on Information and Knowledge Management*, pages 25–30, Baltimore, Maryland, 1995.