Conjunctive Constraint Mapping for Data Translation

Chen-Chuan K. Chang
Department of Electrical Engineering
Stanford University
E-mail: changcc@cs.stanford.edu

Héctor García-Molina
Department of Computer Science
Stanford University
E-mail: hector@cs.stanford.edu

ABSTRACT

In this paper we present a mechanism for translating information in heterogeneous digital library environments. We model information as a set of conjunctive constraints that are satisfied by real-world objects (*e.g.*, documents, their metadata). Through application of semantic rules and value transformation functions, constraints are mapped into ones understood and supported in another context. Our machinery can also deal with hierarchically structured information.

KEYWORDS: constraint mapping, data translation, semantic interoperability.

1 INTRODUCTION

Digital libraries often need to interact with autonomous systems that structure and represent their information differently. To ensure semantic interoperability [12], information must be appropriately mapped from its source context to its target context where it will be used. For example, the condition [author = "John Smith"] in a query, may need to be translated to [name = "Smith, J."]. Similarly, the component [date = "Jan 1998"] in the resulting answer (either metadata about a document or the document itself) may have to be translated to the pair of components [year = 1998] and [month = "January"].

For translation purposes we view "information" as a collection of *constraints* of the form [attrib = val] or simply [attrib val]. If a constraint refers to a document, it indicates a "fact" about it, *i.e.*, that the attribute or property attrib has the value val. If it refers to a query, then the constraint specifies a desired fact about matching documents. Our attributes can be hierarchical, *e.g.*, publication.date.year (*i.e.*, paths). In general, constraints could have non-equality operators, *e.g.*, [date > "Jan 1998"]. Due to space limitations, we only consider the equality operator, and therefore do not explicitly write it. Our work can be generalized to other operators.

In this paper, we interpret a set of constraints as a *conjunction*, *i.e.*, the desired or described document satisfies all of the constraints. This is adequate for describing documents and vector-space queries (where a query is simply a document similar to the desired ones). However, Boolean queries may include disjunctions, *e.g.*, [author "Smith"] OR [date "Jan 1998"]. In [5] we extend our work to handle arbitrary Boolean queries with disjunctions and conjunctions. It is important to note that translating a set of conjunctive constraints into another is not simply a matter of mapping each constraint separately. In general, the mappings can be many-to-many. For example, the constraints [car-type "ford-taurus"] AND [year 1994] may yield [make "ford"] AND [model "taurus-94"] at the target.

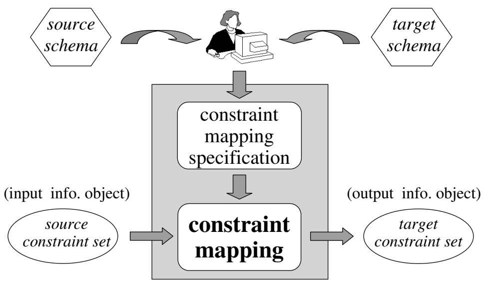


Figure 1: Overview of the data translation process.

This paper presents a framework and algorithms for information translation. The translation is performed by *rules*. Intuitively, each rule specifies how one or more source constraints are to be transformed into one or more target constraints. For translating the values themselves, *e.g.*, "John Smith" into the first initial and last name strings "J.", "Smith," we rely on *functions* that can be written in any programming language. The rules and functions are written by human experts (*e.g.*, librarians), either for pairs of sources or pairs of attribute sets (*e.g.*, Dublin Core to USMARC). Eventually, we expect that libraries of common translation functions will be available, so the librarian will mainly have to select functions (with what parameters) to perform the translation. At run time, queries and documents can then be translated automatically using the rules provided. Fig. 1 gives an overview of the translation process: The librarians analyze the source and target contexts (*i.e.*, their schemas), and define the mapping rules, which then drive the constraint mapping algorithm to perform data translation.

There are many ways one can go about translating digital library information (*e.g.*, [1, 2, 3, 8, 14, 15], see Section 5).

We believe that our approach offers the following advantages over other proposals.

- Because we view information as constraints, our machinery can provide "minimal" translations without explicit user guidance. For example, if several rules are available for translating say first, middle and last names of authors, the one that covers the most constraints will be applied.
- Our framework provides a natural way for dealing with hierarchically structured documents (including HTML, SGML). We encode the location of each value into a hierarchical attribute name, and then use the same simple machinery for translating the value into its appropriate target location.
- We include several features that have turned out to be very useful in a digital library environment. For instance, we provide facilities for dealing with lists of values (*e.g.*, for converting a list of authors to a single string containing all names), and for adding default constraints *e.g.*, if no author is given, generate [author `"anonymous"`].
- Our last "advantage" is subjective, but we believe that our framework provides the correct balance between simplicity and power needed in a digital library environment. At one end of the spectrum, we could say that a single function translates full documents or queries. This is a very simple framework, but leaves all the work for the function. At the other end, we could eliminate functions and do all processing within the translation framework. For example, to translate a PowerPoint image to a GIF, we would not call a function, but we would write rules that specify how each component of the image is translated. This would require the framework to be as powerful as a general purpose programming language (including recursion). As we will see, our framework focuses on the logical mappings between constraints, leaving the pure procedural transformation of *values* to functions. This makes it easy to both implement and use our framework.

We start by defining the subsuming mapping and other fundamental notions for our framework. In Section 3 we develop the rule system framework and the associated algorithm for data translation, initially for "flat" data (*i.e.*, those naturally represented as attribute-value pairs). Section 4 then generalizes the translation framework for hierarchical data (*i.e.*, those with nested structures). Finally, in Section 5 we briefly review and compare the related efforts on data translation to justify why we advocate our particular approach.

2 PRELIMINARIES

Our universe is a collection C of information *objects*, each representing a real world entity of interest (*e.g.*, a book, a document, a car, a person). We define a *query* (or a discriminator) to be a set of conjunctive constraints that selects or identifies one or more objects. For example, the set $Q = \{c_1, c_2\}$, where c_1 = [lname `"Smith"`], and c_2 = [fname `"John"`] is a query that selects the qualified objects (*i.e.*, persons named `"Smith, John"`) from our universe

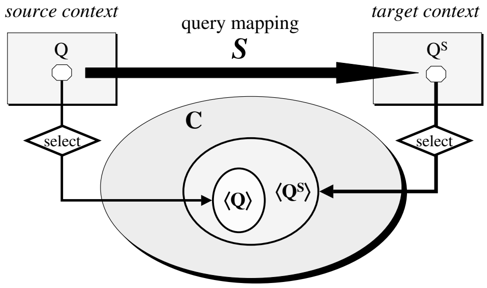


Figure 2: Conceptual illustration of query mapping.

of objects. To stress the fact that Q is a conjunction (selected object must satisfy all constraints), we write it as c_1 AND c_2, or as AND($\{c_1, c_2\}$). An individual object (*e.g.*, a document) can be represented by the query that uniquely identifies it.

Fig. 2 conceptually illustrates the problem we face. We are given a query Q in a *source context*. This context may limit how constraints are expressed, semantically or syntactically. For instance, a context may not support some attributes, or support them with different attribute names, value formats, unit, currency, *etc*. Within the source context, Q can be interpreted and identifies a set of objects $\langle Q \rangle$ that satisfy all the Q constraints.

Our problem is to transform the information encoded by Q into another set of constraints Q^S that can be interpreted in a different *target context* (Fig. 2). In particular, we would like to have a *query mapping function* $\mathcal{S}(\cdot)$ that transforms any query like Q into a "good" query Q^S for the target context. What does "good" mean here? First, Q^S should be interpretable in the target context, *e.g.*, it should use appropriate attributes and values. Second, the objects selected by Q^S, $\langle Q^S \rangle$, should be a superset of $\langle Q \rangle$ (we do not want to miss any objects). For example, consider again the `"Smith, John"` query c_1 AND c_2. Suppose that the target context does not support the fname attribute, but does support last names. Then we may have to translate this query into c_1. This query will identify more objects than the original one, and is acceptable if there is no other way to identify at the target a smaller set of objects that includes "`Smith, John`." The latter minimality requirement represents our third condition for a good mapping.

The next definition formalizes these three properties. Note that the notion of query subsumption is directly related to *query containment* in deductive databases [16] and has been applied extensively in information integration [17].

Definition 1 (Minimal Subsuming Mapping): Let $\mathcal{S}(\cdot)$ be a mapping on queries of a source system T_s. (We use the notation Q^S for $\mathcal{S}(Q)$). $\mathcal{S}(\cdot)$ is the *minimal subsuming mapping w.r.t.* some target system T, if for any query Q in T_s,

1. Q^S is expressible in the context of T,
2. Q^S *subsumes* Q, *i.e.*, $\langle Q \rangle$ is a subset of $\langle Q^S \rangle$; $\langle Q \rangle \subseteq \langle Q^S \rangle$,
3. Q^S is *minimal*, *i.e.*, there is no query Q' such that (i) Q' satisfies 1 and 2, and (ii) Q^S subsumes Q'. ■

A good mapping $\mathcal{S}(\cdot)$ cannot simply translate the individual constraints independently [6], as Example 1 illustrates.

Example 1: Consider again the mapping of $Q = c_1$ AND c_2, where c_1 is [fname "John"], and c_2 is [lname "Smith"]. Suppose that, corresponding to lname and fname, the target system supports the name attribute, but requires that at least the last name component be specified. That is, a name can be stated as "Smith, John", or simply "Smith", if the first name is not known.

If we assume that the constraints are "independent" and handle them separately, we obtain the following mapping, where $\mathcal{S}(c_1)$=*True* as no smaller mapping exists when only the first name is known. Note that *True* actually means no constraint.

$$\begin{aligned} \mathcal{S}(Q) &= \mathcal{S}(c_1) \text{ AND } \mathcal{S}(c_2) \\ &= \mathit{True} \text{ AND [name "Smith"]} \\ &= \text{[name "Smith"]}. \end{aligned}$$

Clearly, the above is not the minimal translation, because the fname constraint is effectively discarded. In fact, constraints c_1 and c_2 are "interrelated" because they collectively decide the target constraint on name. In other words, by considering c_1 and c_2 together, we obtain the minimal translation $\mathcal{S}(Q)$ = [name "Smith, John"]. ■

As Example 1 illustrates, to obtain the optimal mappings, the translation framework cannot blindly assume that constraints are independent. Unfortunately, there is no automatic way for computers to figure out the interdependence of constraints; that knowledge must be supplied by human experts in a computable way. In Section 3 we introduce a rule system framework for data translation that encodes the "dependence" of constraints, and that defines how constraints should be mapped. In Section 4 we extend this rule system to hierarchical information models (such as SGML).

3 CONJUNCTIVE QUERY MAPPING

Given a constraint set I representing a conjunctive query AND(I), our goal is to find its minimal subsuming mapping, $\mathcal{S}$(AND(I)), which is a conjunctive query in the target context. Fig. 3 shows a running example that we use in this section to illustrate the mapping process. A site Car maintains information on used cars; a second site Auto would like to collect data from the first site (and possibly other sites). Thus, information from the source context Car must be translated into the Auto target context. Fig. 3 shows on the left a particular object as represented by Car. That is, if I_{car} represents the constraints on the left of the figure ($I_{car} \equiv \{c_1, c_2, \cdots, c_9\}$), then the query AND($I_{car}$) represents that object. On the right of Fig. 3 we show the same object as represented in the target context. If I_{auto} are the constraints on the right ($I_{auto} \equiv \{a_1, a_2, \cdots, a_8\}$), then we would like our mapping to yield the target query AND(I_{auto}). That is, $\mathcal{S}$(AND(I_{car})) = AND(I_{auto}).

Given a source constraint set, our algorithm first maps individual constraints as directed by a rule system, and then formulates the mapping of the source query as a whole. The rule system, called a *mapping specification*, consists of a set of *mapping rules*, each specifying the mapping of some constraint patterns. For instance, Fig. 4 shows a mapping specification K_{car} for translating instances of Car to Auto. In the remainder of this section, we first discuss the rule system framework, and then the algorithm that uses the rule system to handle conjunctive queries.

Car instance I_{car}		Auto instance I_{auto}	
[lname "Smith"]	(c_1)	[dealer "Ford 101"]	(a_1)
[fname "John"]	(c_2)	[owner "John Smith"]	(a_2)
[license "4HQD973"]	(c_3)	[id "4HQD973"]	(a_3)
[price 9000]	(c_4)	[price 9000]	(a_4)
[car-type "ford-taurus"]	(c_5)	[model "taurus-94"]	(a_5)
[year "1994"]	(c_6)	[make "ford"]	(a_6)
[mileage 80000]	(c_7)	[mileage 50000]	(a_7)
[unit "km"]	(c_8)	[color-code "bl"]	(a_8)
[color "blue"]	(c_9)		

Figure 3: Mapping instance from Car to Auto.

$K_{car} \equiv \{\mathcal{R}_{c.1}, \mathcal{R}_{c.2}, \mathcal{R}_{c.3}, \mathcal{R}_{c.4}, \mathcal{R}_{c.5}, \mathcal{R}_{c.6}, \mathcal{R}_{c.7}\}$

$\mathcal{R}_{c.1}$) [A1 X]; SimpleMapping(A1) $\mapsto$ A2 = AttrNameMapping(A1);
emit: [A2 X]

$\mathcal{R}_{c.2}$) [lname L]; [fname F] $\mapsto$ A = LnFnToName(L,F);
emit: [owner A]

$\mathcal{R}_{c.3}$) [lname L] $\mapsto$ *emit*: [owner L]

$\mathcal{R}_{c.4}$) [car-type T] ; [year Y] $\mapsto$ M = Model(T,Y) ; A = Make(T) ;
emit: [model M] AND [make A]

$\mathcal{R}_{c.5}$) [mileage M1] ; [unit U1] $\mapsto$ M2 = CvtLengUnit(M1,U1,"mile")
emit: [mileage M2]

$\mathcal{R}_{c.6}$) [color C] $\mapsto$ CC = ColorCode(C)
emit: [color-code CC]

$\mathcal{R}_{c.7}$) $\mapsto$ *emit*: [dealer "Ford 101"]

Figure 4: K_{car} for mapping Car to Auto.

3.1 Mapping Rules

A constraint mapping rule specifies how a source constraint, or the conjunction of some related source constraints, can be mapped to the target side. Intuitively, we may think of the operation of a rule for constraint mapping as *pattern matching*, similar to that of the rules used in Lex and Yacc. In particular, a rule specifies some *pre-conditions* in its left hand side, the *head*, and some *actions* in its right hand side, the *tail*, as separated by the $\mapsto$ symbol. The pre-conditions state the source constraints to be matched, and the actions direct how the corresponding target constraints can be generated.

Rule $\mathcal{R}_{c.6}$ (Fig. 4) states that a source constraint on the color attribute is mapped to a target constraint on the color-code attribute, with the color value replaced with some standard code. The rule calls upon the function ColorCode() to look up the color code. Similarly, rule $\mathcal{R}_{c.2}$ specifies that the source constraints on fname and lname are translated jointly by first merging them into a full name, and then formulating the target constraint on owner.

The head of a rule consists of constraint patterns and conditions to be matched against the source constraints. A *con-*

straint pattern is a 2-tuple [attrib val], where each component can be either a constant (of the corresponding domain) or *variable*. A source constraint *matches* a pattern if both agree on the non-variable components. As the result of a matching, the variables of the pattern is then *bound* to the corresponding constants in the matching constraint. For instance, matching the pattern [color C] with the constraint [color "blue"] results in the binding of C to the constant "blue."

A *condition* is a predicate function that takes bound variables as arguments, and return either *True* or *False*. Conditions can be used in the head of a rule, in addition to constraint patterns, to restrict matchings to only those variable bindings that hold the conditions. For instance, in $\mathcal{R}_{c.1}$, the condition SimpleMapping($\cdot$) tests if the attribute A1 is one that requires only straightforward name mapping. If so, the matching constraint will be "copied" with only the attribute name mapped appropriately. For example, referring to Fig. 3, license and price are such "simple" attributes.

The tail (right hand side) of a rule directs, for each set of matching source constraints, how the corresponding target constraints can be generated. It consists of two parts: a list of *function statements* and an "*emit*:" clause, with the former responsible for converting value formats, attribute mapping, and so on, and the latter specifying the corresponding target constraints to be generated.

A function statement is of the form:

$\mathsf{Y}_1, \mathsf{Y}_2, \ldots, \mathsf{Y}_m$ = FunctionName($\mathsf{X}_1, \mathsf{X}_2, \ldots, \mathsf{X}_n$),

where X_i's are input constants or variables that are already bound in the rule head or the preceding function statements, and Y_j's are variables to be bound by the function output. For instance, function LnFnToName($\cdot$) (in $\mathcal{R}_{c.2}$) takes the bound variable L and F as its argument, and binds A as the full name. Similarly, function CvtLengUnit(M1,U1,"mile") converts length M1 from unit U1 to "mile" (*i.e.*, the target mileage is always specified in miles).

Note that the functions in the tail as well as the conditions in the head are supplied externally, and in principle can be written in any programming languages.

As the last component of a rule, the *emit*: clause specifies the corresponding target constraints. Note that it can generate more than one constraint connected with AND. (Recall that constraint mapping is in general many-to-many.) For instance, rule $\mathcal{R}_{c.4}$ produces the conjunction of the constraints on model and make.

Having described the components of mapping rules, we next discuss their evaluation and semantics. The evaluation of a rule, given a set of source constraints as inputs, involves finding matching constraints, from which the target constraints will be emitted as directed by the rule actions. Moreover, the semantics of the rule requires that the emitted constraints be the minimal subsuming mapping of the matching constraints (except for a special case, see later). In Definition 2 we formalize the notion of matchings and emissions. Example 2 then illustrates these notions and uses them to show the rule semantics, which we will formalize in Definition 3.

Definition 2 (Matchings and Emissions): Let r be a mapping rule and I be a set of source constraints.

1. A *matching set* (or *matching* for short) of r *w.r.t.* I is a subset of I, of which the constraints together satisfy the head of r. We denote the set of *all the matching sets* of r *w.r.t.* I by $\mathcal{M}(r, I)$.
2. The *emission* of r *w.r.t.* a matching m, *i.e.*, $m \in \mathcal{M}(r, I)$, is the query generated by the *emit*: clause of r *w.r.t.* m, and is denoted $\mathcal{E}(r, m)$. ■

Example 2: Let's consider the evaluation of the rules in K_{car} (Fig. 4) with respect to the source constraint set I_{car} (Fig. 3).

For $\mathcal{R}_{c.1}$, the pattern [A1 X] can match any constraint in I. Assuming that the function SimpleMapping($\cdot$) returns *True* for only license and price, we obtain two matchings: $\{c_3\}$ and $\{c_4\}$, *i.e.*, $\mathcal{M}(\mathcal{R}_{c.1}, I) = \{\{c_3\}, \{c_4\}\}$.

With the matching $\{c_3\}$, assume $\mathcal{R}_{c.1}$ emits the target constraint a_3, *i.e.*, $\mathcal{E}(\mathcal{R}_{c.1}, \{c_3\}) = a_3$. By definition, this means $\mathcal{S}(c_3) = a_3$.

For $\mathcal{R}_{c.2}$, $\mathcal{M}(\mathcal{R}_{c.2}, I) = \{\{c_1, c_2\}\}$, *i.e.*, constraints c_1 and c_2 together form a matching. Since $\mathcal{E}(\mathcal{R}_{c.2}, \{c_1, c_2\}) = a_2$, we have

$$\mathcal{S}(\mathsf{AND}(\{c_1, c_2\})) = \mathcal{S}(c_1 \ \mathsf{AND}\ c_2) = a_2.$$

Finally, observe that rule $\mathcal{R}_{c.7}$ does not require any constraints to form a matching, because the rule head is empty. In this case, $\mathcal{M}(\mathcal{R}_{c.7}, I) = \{\varnothing\}$ (the empty set is a matching). Note that the rule will be fired no matter what the source constraints are. The special case of "empty" matching is useful in making implicit constraints explicit at the target side. For $\mathcal{R}_{c.7}$, even based on no source constraints, the rule still emits [dealer "Ford 101"], an implicit constraint that all Car objects assume. ■

The next definition tells us that each rule must emit "good" target constraints, in the sense discussed in Section 2. In other words, Definition 3 states the requirement for a rule to be correct. Clearly, as in any other rule systems (or programming in general), it is always possible to give incorrect or inconsistent rules, in which case best translations cannot be guaranteed. Furthermore, this notion of goodness does not apply for empty matchings. For instance, for rule $\mathcal{R}_{c.7}$ in Example 2, $\mathcal{S}(\mathsf{AND}(\varnothing)) \neq$ [dealer "Ford 101"], because $\mathsf{AND}(\varnothing) \equiv$ *True* (*i.e.*, no constraints).

Definition 3 (Rule Semantics): Let r be a mapping rule and I be a set of source constraints. For any matching m, *s.t.* $m \in \mathcal{M}(r, I)$

1. if $m \neq \varnothing$, then $\mathcal{S}(\ \mathsf{AND}(m)\ ) \equiv \mathcal{E}(r, m)$. That is, the emission of the rule *defines* the minimal subsuming mapping of the matching constraints.
2. otherwise, if $m = \varnothing$, then the emission of the rule represents the mapping of some *implicit constraints* assumed at the source system. ■

3.2 Algorithm for Conjunctive Query Mapping

The conjunctive query mapping problem can be stated as follows. Given a set of source constraints I representing a conjunctive query $\mathsf{AND}(I)$, and a set of rules K as the constraint mapping specification *w.r.t.* some target system T, the problem is to find the minimal subsuming mapping of the source query $\mathsf{AND}(I)$ *w.r.t.* T, *i.e.*, $\mathcal{S}(\mathsf{AND}(I))$.

The algorithm is relatively straightforward; essentially, the mapping of the whole query (as a constraint set) is the conjunction of the mappings of some subsets of the constraints. We first evaluate the mapping rules with respect to the source constraints to find the matchings. Note that each matching is a subset of constraints that are deemed "interrelated" and thus must be processed together (*e.g.*, constraints c_1 and c_2 in Example 2). That is, the mapping rules effectively partition the source constraint set into subsets of interrelated constraints, and define their mappings (as the emissions). The mappings of those matching subsets are then assembled in a conjunctive form as the mapping of the whole source query.

However, some matching subsets may be redundant and thus should be removed. For instance, in K_{car} (Fig. 4) $\mathcal{R}_{c.2}$ defines the mapping to owner from both the source lname and fname, while $\mathcal{R}_{c.3}$ from only the lname constraint. Note that $\mathcal{R}_{c.3}$ can be useful to generate a partial name of owner if fname is optional and can be omitted in some Car instances. However, for a source instance with both lname and fname, such as I_{car}, $\mathcal{R}_{c.3}$ gives a redundant matching $\{c_1\}$, because in this case $\mathcal{R}_{c.2}$ can generate better mappings (*i.e.*, full names) with the "larger" matching $\{c_1, c_2\}$.

In general, if a matching is a subset of some other matching, we can eliminate the former because, with a larger set of source constraints, the latter will generate a "smaller" mapping. However, when the matching is $\varnothing$, it cannot be eliminated because it actually represents some implicit constraints, as discussed in Section 3.1. Due to space limitation, we are not able to discuss the formalism supporting this *sub-matching suppression*; please refer to [5] for more details.

Fig. 5 presents the mapping algorithm, which we next illustrate with Example 3. Please refer to [5] for the proof that Algorithm 1 does generate minimal subsuming mappings.

Input: Let I be a set of constraints, and K be the constraint mapping specification *w.r.t.* a target system T.
Output: $\mathcal{S}(\mathsf{AND}(I))$, the minimal subsuming mapping of $\mathsf{AND}(I)$ *w.r.t.* T.
Procedure:
(*1*) Find all the matchings for all the rules in K:
$A := \cup(\mathcal{M}(r, I))$, *for all* $r \in K$.
(*2*) Remove any non-empty matching that is a subset of other matchings (*i.e.*, sub-matching suppression):
for all $m_i \in A$:
for all $m_j \in A$ $(j \neq i)$:
if $m_j \subseteq m_i$ *and* $m_j \neq \varnothing$, *remove* m_j *from* A.
(*3*) Output the conjunction of all the emissions:
$\mathcal{S}(\mathsf{AND}(I)) = \mathsf{AND}(\mathcal{S}(\mathsf{AND}(m_i))) = \mathsf{AND}(\mathcal{E}(r, m_i))$,
for all $m_i \in A$, *s.t.* m_i *matches* r, *i.e.*, $m_i \in \mathcal{M}(r, I)$.

Figure 5: Algorithm 1 for conjunctive query mapping.

Example 3: To illustrate Algorithm 1 for data mapping, we use it to translate the Car instance I_{car} (Fig. 3). That is, we run the algorithm with the constraint set I_{car} and mapping specification K_{car} (Fig. 4) as inputs, and show that it outputs I_{auto} as the mapping, *i.e.*, $\mathcal{S}(\mathsf{AND}(I_{car})) = \mathsf{AND}(I_{auto})$. We illustrate the process step by step.

1. $A = \cup(\mathcal{M}(\mathcal{R}_{c.1}, I), \mathcal{M}(\mathcal{R}_{c.2}, I), \cdots, \mathcal{M}(\mathcal{R}_{c.7}, I))$
$= \cup\,(\,\{\{c_3\},\{c_4\}\},\{\{c_1,c_2\}\},\{\{c_1\}\},\{\{c_5,c_6\}\},$
$\{\{c_7,c_8\}\},\{\{c_9\}\},\{\varnothing\}\,)$
$= \{\,\{c_3\},\{c_4\},\{c_1,c_2\},\{c_1\},$
$\{c_5,c_6\},\{c_7,c_8\},\{c_9\},\varnothing\}$
2. $\{c_1\}$ is removed, because it is a subset of $\{c_1, c_2\}$.
3. $\mathcal{S}(\,\mathsf{AND}(I_{car})\,)$
$= \mathsf{AND}(\,\mathcal{E}(\mathcal{R}_{c.1},\{c_3\}),\mathcal{E}(\mathcal{R}_{c.1},\{c_4\}),$
$\mathcal{E}(\mathcal{R}_{c.2},\{c_1,c_2\}),\mathcal{E}(\mathcal{R}_{c.4},\{c_5,c_6\}),\cdots\cdots,$
$\mathcal{E}(\mathcal{R}_{c.7},\{\varnothing\})$
$= \mathsf{AND}(a_3, a_4, a_2, a_5 \;\mathsf{AND}\; a_6, a_7, a_8, a_1)$
$= \mathsf{AND}(I_{auto})$ ■

4 HIERARCHICAL DATA TRANSLATION

We have studied the translation machinery for "flat" data (as sets of attribute-value pairs); in this section we generalize the framework for hierarchical (*i.e.*, nested) data models. While the flat representation of data has been widely used (*e.g.*, BibTex, Dublin Core [18], and relational databases), hierarchical structures have also been developed to provide richer information abstraction. In particular, they have been used for structured documents (*e.g.*, SGML [9], XML, see `http://www.w3.org/XML`), metadata (*e.g.*, USMARC, Warwick Framework [13]), semi-structured data (*e.g.*, the OEM model in [14]), and scientific data [3].

This section is organized as follows. In Section 4.1, we discuss what we view as hierarchical data by introducing the underlying "conceptual" model as well as our restrictions (*e.g.*, we do not handle recursive schemas, see later). We then generalize the framework for translating data in this model. First, to base translation on the constraint mapping framework, we need to extend the notion of constraints for tree-like hierarchical data, so that they can be represented as conjunctive constraints. Second, the evaluation of mapping rules (in particular, the pattern matching) is slightly complicated because patterns can match "macro" constraints that represent, say, subtrees. We discuss them in Sec. 4.2 and 4.3 respectively.

4.1 Conceptual Grammatical Model

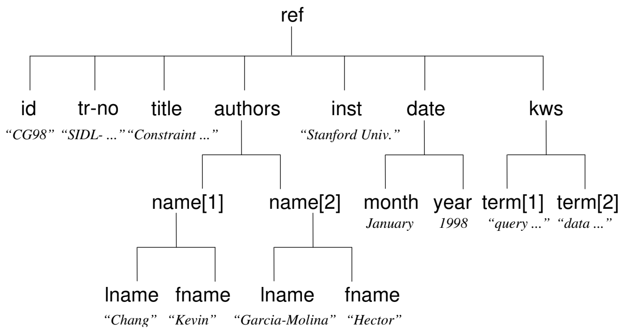
As a basis for hierarchical data translation, we conceptually abstract information objects (*e.g.*, documents, metadata, *etc.*) as EBNF (Extended Backus-Naur Form [19]) grammar-generated trees. Recognizing the fundamental characteristics of data to be its hierarchical and grammatical structure, many have used the well-understood concept of formal language grammars to model data [9, 10]. For instance, in SGML [9]

```
<ref>
<id> CG98 </id>
<tr-no> SIDL-WP-98-01 </tr-no>
<title> Connstraint Mapping </title>
<authors>
   <name> <lname> Chang </lname>
          <fname> Kevin </fname> </name>
   <name> <lname> Garcia-Molina </lname>
          <fname> Hector </fname> </name>
</authors>
<inst> Stanford Univ. </inst>
<date> <month> January </month>
       <year> 1998 </year> </date>
<kws> <term> query translation </term>
      <term> data mapping </term> </kws>
</ref>
```

(a) *Example SGML* Ref *document.*

V = { ref, authors, name, date, kws }
T = { id, tr-no, title, inst, lname, fname, month, year, term }
S = ref
P = { ref $\rightarrow$ id tr-no$^?$ title authors inst date kws
authors $\rightarrow$ name$^+$
name $\rightarrow$ lname fname
date $\rightarrow$ month year
kws $\rightarrow$ term* }
D = *all map to* `string`

(b) *Conceptual schema* G_{ref} *for SGML* Ref *documents.*



(c) *Example instance tree* I_{ref} *of schema* G_{ref}.

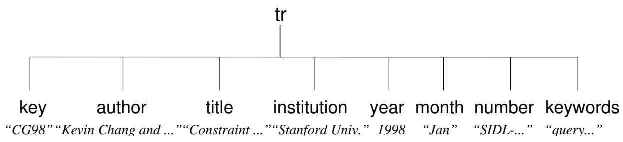
Figure 6: Schema and instance for SGML Ref**.**

```
@TechReport{CG98,
          author      = "Kevin Chang and
                         Hector Garcia-Molina",
          title       = "Constraint Mapping",
          institution = "Stanford University",
          year        = 1998,
          month       = "Jan",
          number      = "SIDL-WP-98-01",
          keywords    = "query translation,
                         data mapping" }
```

(a) *Example BibTex* TechReport *entry.*

V = { tr }
T = { key, author, title, institution, year, month, number, keywords }
S = tr
P = { tr $\rightarrow$ key author title institution
year month$^?$ number$^?$ keywords$^?$}
D = key: `string`, author: `string`, title: `string`, institution: `string`, year: `int`, month: `string`, number: `string`, keywords: `string`

(b) *Conceptual schema* G_{tr} *for BibTex* TechReport *entries.*



(c) *Example instance tree* I_{tr} *of schema* G_{tr}.

Figure 7: Schema and instance for Bibtex TechReport**.**

(or the evolving XML), documents are defined with a DTD (document type definition), which is essentially a grammar. In particular, we adopt and extend the formalism of *grammatical model* [10] to model hierarchical data.

Definition 4 (Conceptual Grammatical Model):
A *conceptual grammatical schema* G is a five-tuple $G = (V, T, P, S, D)$:

- (V,T,P, S) is a context free grammar in EBNF , of which V and T are finite sets of *structural attributes* and *terminal attributes* respectively, S a *start symbol*, $S \in V$, and P a finite set of EBNF production of the form $A \rightarrow \alpha$ where $A \in V$, α is a non-empty regular expression over the alphabet $V \cup T$.
- D is a mapping on T, *s.t.* $D(A)$ is a set called the domain of A, for $A \in T$.

An *instance* (or *instance tree*) of G is a *derivation tree* of G, where the leaves (with terminal attributes) are assigned *values* from the corresponding domains that D defines. ■

We illustrate the notions with the running examples in SGML (Fig. 6) and BibTex (Fig. 7). Specifically, Fig. 6(a) shows an (assumed) SGML document of type Ref for recording bibliographic reference information. In (b) we illustrate the conceptual schema that defines the logical structure of such documents. The example document is then represented as an instance tree in (c). Furthermore, Fig. 7 shows the same information in the BibTex TechReport format; in particular, it also shows the corresponding BibTex entry in (a), the conceptual schema in (b), and the instance tree in (c). We will use these examples to illustrate the translation process, assuming we want to translate information from Ref to TechReport. Note that, in the regular expressions of grammar productions, we use * for Kleene closure (0 or more times), $^+$ for positive closure (1 or more times), $^?$ for optionality (0 or 1 time), and | for separating alternatives. For instance, tr-no$^?$ in G_{ref} indicates that the attribute is optional.

The iteration constructs in EBNF ("$^+$" and "*") provide a way for specifying *lists* of repeating attributes, which we use to model *collections*, for instance, a collection of names as the value of authors (authors $\rightarrow$ name$^+$). Note that, to unambiguously reference list elements, the repeated occurrences of an attribute (as generated by the iteration constructs) in instance trees (*e.g.*, Fig. 6(c)) are ordered and labeled with the usual list operators, *e.g.*, name[1] for the first element in the name list. In addition, the notation A[$k1$:$k2$] refers to the sub-list from k_1-th through the k_2-th elements. Either k_1 or k_2 can be omitted and defaulted to 1 and the length of the list, *e.g.*, A[:] refers to the whole list.

We believe the conceptual model is sufficient, for translation purposes, in describing information objects observing some implicit or explicit grammatical structures. In partic-

instance $I_{ref} = \{c_1, c_2, \cdots, c_{12}\}$	
[id "CG98"]	(c_1)
[tr-no "SIDL-WP-98-01"]	(c_2)
[title "Constraint ..."]	(c_3)
[authors.name[1].lname "Chang"]	(c_4)
[authors.name[1].fname "Kevin"]	(c_5)
[authors.name[2].lname "Garcia-Molina"]	(c_6)
[authors.name[2].fname "Hector"]	(c_7)
[inst "Stanford Univ."]	(c_8)
[date.month "January"]	(c_9)
[date.year "1998"]	(c_{10})
[kws.term[1] "query ..."]	(c_{11})
[kws.term[2] "data ..."]	(c_{12})

Figure 8: I_{ref} as a set of terminal constraints.

ular, because a list is by definition an order and duplicate-sensitive collection, one may argue that we need other constructs such as *bags* (order-insensitive) and *sets* (order and duplicate-insensitive). However, it suffices to conceptually model them as lists, *i.e.*, by assuming some ordering in a set or bag, because the additional orderings should have no implication on the translation correctness. For instance, although the schema G_{ref} conceptually represents keywords as a list of terms (by keywords $\rightarrow$ term*), the underlying data source may actually model it as a set.

Without loss of generality, we assume *normalized* schemas, which ensure that the generated instance trees have distinct paths (*i.e.*, no two different paths have the same sequence of labels). This path uniqueness is critical because (for hierarchical data) constraints will be specified with paths (Section 4.2). Formally, a schema is normalized, if every grammar production has the following characteristics:

1. The iteration operators (* and $^+$) associate only with individual attributes. For example, $\mathsf{A} \rightarrow (\mathsf{X}\,|\mathsf{Y})^+$ is not allowed; it can be normalized to $\mathsf{A} \rightarrow \mathsf{B}^+$ and $\mathsf{B} \rightarrow \mathsf{X}\,|\mathsf{Y}$.
2. Attributes are unique within a concatenation. For example, $\mathsf{A} \rightarrow \mathsf{XYX}$ is not allowed; instead, it can be normalized (by annotation) to, say, $\mathsf{A} \rightarrow \mathsf{X_1YX_2}$.

We believe that the restrictions are reasonable for grammars whose purpose is to describe the logical structures of information; most "natural" schemas are already normalized, *e.g.*, the schemas of relational databases, BibTex, Dublin Core, USMARC, *etc*. In other rare cases of unnormalized schemas, it is straightforward to normalize them (as informally suggested in the above requirements). The full machinery for normalizing schemas and their instance trees is given in [4].

Our framework currently does not deal with schemas with recursive grammars, *e.g.*, section $\rightarrow$ title section, where section derives itself. Although in principle it should be feasible, supporting grammar recursion would require more "powerful" path patterns to represent recursively generated paths, which will greatly complicate the mechanism. In fact, we believe that non-recursive schemas cover most practical cases, for instance, document metadata and structured information.

4.2 Constraints as Path-Value Pairs

While constraints correspond naturally to attribute and value pairs for flat data, we need to generalize the notion of constraints for hierarchical data, which is in turn based on extending attributes to paths.

Intuitively, a natural generalization is to "flatten" a hierarchical structure (*e.g.*, Fig. 6(c)) into *path-value* pairs, each representing a constraint (*e.g.*, Fig. 8). More formally, we generalize a constraint to be of the form [P $\mathcal{V}$(P)], where P is a *path* expression, and $\mathcal{V}$(P) the *value* associated with P. We next discuss what paths are, and their values.

In the simplest case, if a path P (starting at the root) leads to a leaf node, then $\mathcal{V}$(P) is the value assigned to the leaf node that P ends with. We call such P a *terminal path*, and [P $\mathcal{V}$(P)] a *terminal constraint*. For instance, in I_{ref} (Fig. 6) the path date.year leads to the value "1998" (*i.e.*, $\mathcal{V}$(date.year) = "1998"), thus representing the constraint [date.year "1998"]. Note that, as (for translation) we are only interested in paths starting from the root, we omit the root label (*e.g.*, ref) in the paths.

Note that, because each terminal path is unique (as guaranteed by normalized schemas [4]), an instance tree can be flattened into (as well as reconstructed from) a set of terminal constraints, which is a generalization of the constraint set representation for flat data. Thus, as we have intuitively observed., the constraint set in Fig. 8 is indeed a valid representation for the instance tree I_{ref} in Fig. 6(c).

Furthermore, as the translation is on tree structures, it is desirable to have mapping functions that process some whole "sub-structures," *e.g.*, subtrees, or lists of terminal paths. In other words, we need *macro constraints* to conveniently represent sets of terminal constraints in the sub-structures. In the following, we discuss such constraints, which in turn are based on the notions of structural paths (for subtrees), and iteration paths (for list construction).

First, to represent a subtree, a *structural path* is a path that ends with some interior node, *e.g.*, in I_{ref}: authors.name[1], date, keywords, *etc*. The value of a structural path is the set of terminal constraints contained in the (sub-)tree rooted at the path. For example, in I_{ref} $\mathcal{V}$(authors.name[1]) is

$\{$[lname "Chang"], [fname "Kevin"]$\}$,

and $\mathcal{V}$(date) is $\{$[month "January"], [year "1998"]$\}$. Formally, the value of a structural path p_s *w.r.t.* an instance I is defined as $\mathcal{V}(\mathsf{p}_s) =$

$\{[\mathsf{p}'\ \mathsf{v}] \mid \mathsf{p}_s.\mathsf{p}'$ is a terminal path in I, $\mathsf{v} = \mathcal{V}(\mathsf{p}_s.\mathsf{p}')\}$.

Second, we use iteration paths to construct lists of (terminal or structural) paths. An *iteration path* p_x is a sequence of labels, $\mathsf{l}_1.\mathsf{l}_2.\cdots.\mathsf{l}_n$, where some l_i is of the form $\mathsf{a}_i[k_1\!:k_2]$. For instance, kws.term[:] denotes the (whole) list of keywords, and its value $\mathcal{V}$(kws.term[:])

$= \langle\ \mathcal{V}$(kws.term[1]), $\mathcal{V}$(kws.term[2]) $\rangle$

$= \langle$ "query $\cdots$", "data $\cdots$" $\rangle$,

i.e., a list of terms (we use $\langle\cdots\rangle$ to denote a list).

As another example, authors.name[1: 2] represents the list of subtrees rooted at authors.name[k], $k \in 1\!:2$. Therefore, we have $\mathcal{V}$(authors.name[1: 2])

= ⟨ $\mathcal{V}$(authors.name[1]), $\mathcal{V}$(authors.name[2]) ⟩
= ⟨ {[lname "Chang"], [fname "Kevin"]},
{[lname "Garcia−···"], [fname "Hector"]} ⟩.

Formally, the value of an iteration path p_x is defined recursively as follows. Assuming p_x = p_1.a[k_1:k_2].p_2, *s.t.* $a[k_1\!:\!k_2]$ is the first label with an iteration range, $\mathcal{V}(\mathsf{p}_x)$:=

$$\langle \mathcal{V}(\mathsf{p}_1.\mathsf{a}[k_1].\mathsf{p}_2), \mathcal{V}(\mathsf{p}_1.\mathsf{a}[k_1+1].\mathsf{p}_2), \cdots, \mathcal{V}(\mathsf{p}_1.\mathsf{a}[k_2].\mathsf{p}_2)\rangle.$$

Finally, note that a macro constraint actually represents a set of terminal constraints. For instance, a constraint specified with a structural path p_s, [p_s $\mathcal{V}(\mathsf{p}_s)$], represents the set of terminal constraints contained in the subtree rooted at p_s. To see what terminal constraints are actually covered, as the translation algorithm requires, we can "flatten" a constraint with the following operational definition.

Definition 5 (Flattening Function): Let c = [p $\mathcal{V}$(p)] be a constraint. The flattening function $\mathcal{F}(c)$ returns the set of terminal constraints contained in c:

1. if p is a terminal path (*i.e.*, c is a terminal constraint), then
$$\mathcal{F}([\mathsf{p}\ \mathcal{V}(\mathsf{p})]) := \{[\mathsf{p}\ \mathcal{V}(\mathsf{p})]\};$$
2. otherwise, if p is a structural path, then
$$\mathcal{F}([\mathsf{p}\ \mathcal{V}(\mathsf{p})]) := \{[\mathsf{p}.\mathsf{p}'\ \mathsf{v}] \mid [\mathsf{p}'\ \mathsf{v}] \in \mathcal{V}(\mathsf{p})\};$$
3. otherwise, if p is an iteration path p_1.a[k_1:k_2].p_2, where a[k_1:k_2] is the first label with an iteration range, then
$$\mathcal{F}([\mathsf{p}\ \mathcal{V}(\mathsf{p})]) := \bigcup_{j=k_1}^{k_2} \mathcal{F}(\ [\mathsf{p}_1.\mathsf{a}[j].\mathsf{p}_2\ \mathcal{V}(\mathsf{p}_1.\mathsf{a}[j].\mathsf{p}_2)]\ )$$
■

For example, to find the terminal constraints contained in the macro constraint with the iteration path author.name[1: 2], we evaluate the flattening function as follows:

$\mathcal{F}$([author.name[1: 2] $\mathcal{V}$(author.name[1: 2])])
= $\cup_{j=1}^{2}$ $\mathcal{F}$([author.name[j] $\mathcal{V}$(author.name[j])])
= $\mathcal{F}$([author.name[1]
{[lname "Chang"], [fname "Kevin"]}]) ∪
$\mathcal{F}$([author.name[2]
{[lname "Garcia−···"], [fname "Hector"]}])
= { [author.name[1].lname "Chang"],
[author.name[1].fname "Kevin"] } ∪ { ··· }
= $\{\ c_4,\ c_5\ \} \cup \{\ c_6,\ c_7\ \} = \{\ c_4,\ c_5,\ c_6,\ c_7\ \}$

4.3 Constraint Mapping for Hierarchical Data

With the notion of constraints, we can translate hierarchical data using the constraint mapping machinery discussed in Section 3. Specifically, in our example of translating Ref (Fig. 6) to TechReport (Fig. 7), to apply Algorithm 1, the input will be a source instance tree (*e.g.*, I_{ref} in Fig. 6(c)), and a mapping specification consisting of rules (*e.g.*, K_{ref} in Fig. 9) that directs translation to the target context. Note that the source instance tree conceptually represents a set of terminal constraints, as the algorithm requires (*e.g.*, I_{ref} = $\{c_1, c_2, \cdots, c_{12}\}$ in Fig. 8). However, it is not necessary to actually "materialize" the constraint set representation; the matching and evaluation of rules can be done directly on the tree representation. As its output, the algorithm generates conjunctive constraints that represent the target instance tree (*e.g.*, I_{tr} in Fig. 7(c)).

$K_{ref} \equiv \{\mathcal{R}_{r.1}, \mathcal{R}_{r.2}, \mathcal{R}_{r.3}, \mathcal{R}_{r.4}, \mathcal{R}_{r.5}\}$
$\mathcal{R}_{r.1}$) [A1 X] ; SimpleMapping(A1) ↦ A2 = AttrNameMapping(A1) ; *emit*: [A2 X]
$\mathcal{R}_{r.2}$) [authors.name[:] L] ↦ S = ConsolidateNames(L) ; *emit*: [author S]
$\mathcal{R}_{r.3}$) [date D] ↦ M, Y = MonthYear(D) ; *emit*: [month M] AND [year Y]
$\mathcal{R}_{r.4}$) [kws.term[1: K2] L] ; K2 <= 3 ↦ S = MergeString(L) ; *emit*: [keywords S]
$\mathcal{R}_{r.5}$) NoConstraint([tr-no N]) ↦ *emit*: [number "to be assigned"]

Figure 9: K_{ref} for mapping Ref **to** TechReport**.**

The evaluation and semantics of mapping rules are essentially the same as discussed in Section 3, although the matching of constraint patterns is slightly more complex. In particular, a pattern can represent a macro constraint and thus match a set of terminal constraints, while in Section 3 a pattern can match only a single attribute-value pair. To see how data translation works, we next explain the evaluation of the rules in K_{ref} with respect to the source instance I_{ref}, which will generate I_{tr} as the translation.

For instance, the rule $\mathcal{R}_{r.3}$ in K_{ref} converts the source date (subtree) into the target month and year. Given I_{ref}, the evaluation will consider all the paths (that start from the root), and find that date is a matching path for the pattern [date D]. That is, the pattern matches the macro constraint [date $\mathcal{V}$(date)] (denoted C_d), resulting in binding D to $\mathcal{V}$(date), which is {[month "January"], [year "1998"]}. Note that C_d is a macro constraint representing the set of terminal constraints $\mathcal{F}(C_d) = \{c_9,\ c_{10}\}$, thus $\mathcal{M}(\mathcal{R}_{r.3}, I_{ref}) = \{\{c_9, c_{10}\}\}$. The function MonthYear(·) then takes D as input, in whatever data structure (appropriate for the function's programming language) encoding the value of D, and returns the month and year values. Finally, the rule emits $\mathcal{E}(\mathcal{R}_{r.3}, \{c_9, c_{10}\})$ = [month "January"] AND [year 1998] as the target constraints.

In general, a constraint pattern in mapping rules is of the form [P X], where P is a *parameterized path* and X is a variable or constant representing $\mathcal{V}$(P). A parameterized path P is a path expression that may contain variables to be assigned values such that P matches a path in the source instance. For example, let's consider $\mathcal{R}_{r.1}$, which handles those "simple" translations that require only attribute name mapping. In particular, A1 is a parameterized path that can match any path in I_{ref} consisting of a single attribute (such as id, tr-no, title, authors, *etc.*), and X will be bound to whatever $\mathcal{V}$(A1) is. Further restricted by condition SimpleMapping(A1), which we assume returns *True* for the "simple" attributes id, tr-no, title, and inst, the matchings are thus $\mathcal{M}(\mathcal{R}_{r.1}, I_{ref}) = \{\{c_1\}, \{c_2\}, \{c_3\}, \{c_8\}\}$. Then $\mathcal{R}_{r.1}$ emits the mapping for each matching, *e.g.*, $\mathcal{E}(\mathcal{R}_{r.1}, \{c_1\})$ = [key "CG98"].

A parameterized path can also contain variables that represent integers in the index ranges of an iteration path, *e.g.*, kws.term[1: K2] in rule $\mathcal{R}_{r.4}$. Note that the rule emits the target keywords from at most the first 3 source terms (assuming the target context imposes this restriction). In evaluation, because the list kws.term[:] is of length 2 in I_{ref}, only the bindings of K2 to either 1 or 2 will result in valid (iteration) paths representing sub-lists. Furthermore, the binding of K2 $=$ 1 represents the matching

$\mathcal{F}$([kws.term[1: 1] $\mathcal{V}$(kws.term[1: 1])]) $= \{c_{11}\}$,

while that of K2 $=$ 2

$\mathcal{F}$([kws.term[1: 2] $\mathcal{V}$(kws.term[1: 2])]) $= \{c_{11}, c_{12}\}$.

(This example does not have the matchings of K2 $=$ 3 and above.) Therefore, the former "sub-matching" will be suppressed by the latter (step 2, Algorithm 1). This suppression mechanism and the condition K2 $<=$ 3 will select the "largest" matching within the restriction. Thus, K2 $=$ 2, L is bound to the list ⟨`"query ···"`, `"data ···"`⟩, and the target constraint is [keywords `"query ···, data ···"`].

As another example of iteration paths, rule $\mathcal{R}_{r.2}$ emits the author constraint. The expression authors.name[:] matches the whole list (*i.e.*, authors.name[1: 2]), resulting in the binding of L to $\mathcal{V}$(authors.name[1: 2]) (given in Section 4.2). The function `ConsolidateNames`(L) then merges the names in the required target format.

Finally, rule $\mathcal{R}_{r.5}$ shows the special case of empty matchings to emit "default" constraints (see Definition 3), which is similar to $\mathcal{R}_{c.7}$ (Fig. 4) and is not specific to hierarchical data. Note that a source Ref instance may not have the tr-no constraint (which is optional in G_{ref}, Fig. 6). The condition `NoConstraint`([tr-no N]) checks if the pattern (in which N is simply a dummy variable) does not find any matching in the source tree. If so, the rule emits the default constraint [tr-no `"to be assigned"`]. Otherwise, the transaltion of tr-no will be handled by $\mathcal{R}_{r.1}$ instead, as we just discussed.

5 Related Work

Information integration has long been recognized as a central problem of modern information systems [12, 17]. In this paper we have presented our data translation framework with the goal of coping with semantic or schematic inconsistency in data exchange.

Fully automatic semantic integration is extremely difficult, if not impossible [12]. Thus, most related efforts advocate, like we do, using human-specified translation rules. Most prominently, some systems [1, 14] use variations of datalog rules [16], and some others use the semantic rules of attribute grammars [8]. In addition, there are also systems using special-purpose query languages [2, 3] for transformation. The common idea behind all these approaches is that a rule essentially specifies how target "patterns" are to be generated from matching source "patterns."

Our work differs from other rule translation efforts in its data representations and its supporting theoretic framework, which in turn impact how mapping rules are specified and evaluated. In particular, while we abstract information objects as "flattened" sets of constraints, other systems perform translation directly on object structures (trees or graphs). Furthermore, viewing information as constraints, we address the data translation problem with the theoretic framework of conjunctive query mapping, and specifically using the notion of query subsumption. To the best of our knowledge, no other existing work uses these notions for data translation. We believe that our approach leads to the following advantages:

- *Simple Rules.* Because of our unique data representation, our rules use patterns for constraints, instead of *object patterns* that match the object layouts (*e.g.*, in [1, 2, 3, 8, 14]). Consequently, our rules are more modular and declarative; independent constraints are separated into different rules, each declaring a correspondence between some source and target constraints. (We could call this a "divide and conquer" approach.) In contrast, with object patterns, rules must be composed in a way that mirror the structure of objects. In fact, object rules are typically not independent of each other, because they together direct the bottom-up (or top-down) construction of target structures, in a more procedural fashion.
- *Minimal Translations.* Since we model information as queries, we can generate minimal translations that use the most source constraints, when there are multiple options (as supported by sub-matching suppression, Section 3). In contrast, in other systems, first, it is cumbersome to enumerate all such options in object patterns (*e.g.*, the combination of the absence or presence of some attributes with respect to the whole layout). Second, it would then require an additional mechanism, *e.g.*, rule ordering, to resolve the multiple choices, which introduces more complications.
- *Duality of Data and Queries.* Although in this paper we mainly focused on data translation, our same mechanism can be used for the translation of arbitrary user queries as well [5]. This unification can be very beneficial in the integration of heterogeneous digital libraries that support Boolean query languages.

Some data translation proposals do not rely on rules, as we do, but instead rely on explicit metadata that encodes the context for automated mediation. For instance, the COIN framework [15] uses the *semantic-value* model, where each attribute is annotated with a *property* list that defines its context (*e.g.*, unit = `"usd"` for attribute revenue). This approach assumes one-to-one mappings and thus cannot handle the general case of many-to-many mappings for interrelated attributes. In addition, it only deals with attribute "value" conversion, and is not applicable to hierarchical data. Lastly, it relies on implicit agreement on what the properties for different attributes are, as well as what conversion functions are applicable for translation.

We did not discuss how to extract constraints (attribute-value pairs) from information objects. For structured objects this can be done simply by parsing. In other cases, pattern-based extraction may be used, as illustrated in [11].

Finally, we note that the framework described in this paper is closely related to and complements our earlier work on query translation for Boolean IR systems [6, 7]. In particular, in [6] we discussed the theoretic framework for optimal query mapping; however, it assumes that constraints are independent in translation (see Section 2). The framework in this paper does consider the dependence. Furthermore, our earlier work does not incorporate the ability to call arbitrary functions to translate values, nor does it deal with hierarchical attributes. Actually, we can view the algorithms of [7] as providing translation functions for the particular case of "text search" constraints (*e.g.*, ones that search for words near others, or words matching wildcards).

6 CONCLUSION

Autonomous information systems often structure and represent their information inconsistently in a heterogeneous environment. To ensure semantic interoperability, information must be mapped appropriately when exchanged across interacting contexts. In this paper, we have presented a rule-system framework for information translation. We view information as a set of conjunctive constraints that identify the represented object(s). Consequently, our machinery provides "minimal" translation, as guaranteed by the supporting query mapping formalism [5].

We have also shown how to generalize the constraint mapping framework for hierarchical data, such as SGML or XML documents. The generalization is based on extending attributes to paths in trees, thus flattening the hierarchy into a set of path-value pairs. In addition, we have also presented macro constraints that can be used to process a whole substructure in the trees, *e.g.*, a subtree, or a list of children.

Although our translation rules are modular and declarative, it may still be a tedious task to develop mapping specifications like the ones in Figures 4 and 9. Therefore, it may be useful to develop an interactive graphical interface, where a librarian can browse through metadata specifications for the source and target contexts, and through libraries of available value transformation functions. Through this front-end system the librarian can generate the specifications by selecting options on the screen, as opposed to writing text as shown in our figures. We believe that such a front-end design system can significantly enhance the effective data translation machinery we have developed.

REFERENCES

1. Serge Abiteboul, Sophie Cluet, and Tova Milo. Correspondence and translation for heterogeneous data. In *Proc. of ICDT '97*, Delphi, Greece, 1997. Springer.
2. Dennis S. Arnon. Scrimshaw: a language for document queries and transformations. *Electronic Publishing: Origination, Dissemination and Design*, 6(4):361–372, Dec. 1993.
3. P. Buneman, S.B. Davidson, K. Hart, C. Overton, and L. Wong. A data transformation system for biological data sources. In *Proc. of the 21st VLDB Conference*, Zurich, Switzerland, 1995. VLDB Endowment, Saratoga, Calif.
4. Chen-Chuan K. Chang and Héctor García-Molina. Conjunctive constraint mapping for data translation (extended version). Technical Report SIDL-WP-1998-0093, Stanford Univ., 1998. Accessible at `http://www-diglib.stanford.edu`.
5. Chen-Chuan K. Chang and Héctor García-Molina. Constraint mapping for translating boolean selection queries. Technical Report SIDL-WP-1998-0095, Stanford Univ., 1998. Accessible at `http://www-diglib.stanford.edu`.
6. Chen-Chuan K. Chang, Héctor García-Molina, and Andreas Paepcke. Boolean query mapping across heterogeneous information sources. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):515–521, August 1996.
7. Chen-Chuan K. Chang, Héctor García-Molina, and Andreas Paepcke. Predicate rewriting for translating boolean queries in a heterogeneous information system. Technical Report SIDL-WP-1996-0028, Stanford Univ., January 1996. Accessible at `http://www-diglib.stanford.edu`.
8. An Feng and Toshiro Wakayama. SIMON: a grammar-based transformation system for structured documents. *Electronic Publishing: Origination, Dissemination and Design*, 6(4):361–372, December 1993.
9. Charles F. Goldfarb. *The SGML Handbook*. Oxford University Press, New York, 1990.
10. M. Gyssens, J. Paredaens, and D. V. Gucht. A grammar-based approach towards unifying hierarchical data models. In *Proc. of the 1989 ACM SIGMOD Conference*, Portland, Oreg.
11. Joachim Hammer, Héctor García-Molina, Junghoo Cho, Arturo Crespo, and Rohan Aranha. Extracting semistructured information from the Web. In *Proc. of the Workshop on Management of Semistructured Data*, Tucson, Ariz., 1997.
12. Sandra Heiler. Semantic interoperability. *Computing Surveys*, 27(2):271–273, June 1995.
13. Carl Lagoze, Clifford A. Lynch, and Ron Daniel, Jr. The Warwick Framework: A container architecture for aggregating sets of metadata. Technical Report TR96-1593, Cornell Univ., Computer Science Dept., June 1996.
14. Yannis Papakonstantinou, Héctor García-Molina, and Jeffrey Ullman. Medmaker: A mediation system based on declarative specifications. In *Proc. of ICDE '96*, New Orleans, La., 1996.
15. Edward Sciore, Michael Siegel, and Arnon Rosenthal. Using semantic values to facilitate interoperability among heterogeneous information systems. *Transactions on Database Systems*, 19(2):254–290, June 1994.
16. Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Rockville, Md., 1988.
17. Jeffrey D. Ullman. Information integration using logical views. In *Proc. of the 6th International Conference on Database Theory*, Delphi, Greece, Jan. 1997. Springer.
18. Stuart Weibel, Jean Godby, Eric Miller, and Ron Daniel, Jr. OCLC/NCSA metadata workshop report. URL: `http://purl.org/metadata/dublin_core`, March 1995.
19. Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions. *Communications of the ACM*, 20(11):822–823, November 1977.