Correspondence and Translation for Heterogeneous Data*

Serge Abiteboul[1] and Sophie Cluet[2] and Tova Milo[3]

[1] Stanford University, Standford, CA 94402†
[2] INRIA, Domaine de Voluceau, 78153 Le Chesnay, France
[3] Tel Aviv University,Tel Aviv, Israel

1 Introduction

A primary motivation for new database technology is to provide support for the broad spectrum of multimedia data available notably through the network. These data are stored under different formats: SQL or ODMG (in databases), SGML or LaTex (documents), DX formats (scientific data), Step (CAD/CAM data), etc. Their integration is a very active field of research and development (see for instance, for a very small sample, [10, 6, 7, 9, 8, 12, 19, 20]). In this paper, we provide a formal foundation to facilitate the integration of such heterogeneous data and the maintenance of heterogeneous replicated data.

A sound solution for a data integration task requires a clean abstraction of the different formats in which data are stored, and means for specifying the correspondences/relationships between data in different worlds and for translating data from one world to another. For that we introduce a *middleware* data model that serves as a basis for the integration task, and *declarative rules* for specifying the integration.

The choice of the *middleware* data model is clearly essential. One common trend in data integration over heterogeneous models has always been to use an integrating model that encompasses the source models. We take an opposite approach here, i.e., our model is minimalist. The data structure we use consists of ordered labeled trees. We claim that this simple model is general enough to capture the essence of formats we are interested in. Even though a mapping from a richer data model to this model may loose some of the original semantics, the data itself is preserved and the integration with other data models is facilitated. Our model is similar to the one used in [7] and to the OEM model for unstructured data (see, e.g., [21, 20]). This is not surprising since the data formats that motivated these works are part of the formats that our framework intends to support. A difference with the OEM model is that we view the children of each vertex as ordered. This is crucial to describe lists, an essential component of DX formats. Also, [13] introduces BNF generated trees to unify hierarchical data models. However, due to the fixed number of sons of a rule, collections are represented by left or right deep trees not suitable for the casual users.

A main contribution of the paper is in the declarative specification of correspondences between data in different worlds. For this we use datalog-style rules, enriched with, as a novel feature, *merge* and *cons* term constructors. The semantics of the rules takes into consideration the fact that some internal nodes represent collections with specific properties (e.g., sets are insensitive to order and duplicates). We show that correspondences between data elements can be computed in polynomial time in many cases, and may require exponential time only when insensitivity to order or duplicates are considered.

Deriving correspondences within existing data is only one issue in a heterogeneous context. One would also want to translate data from one representation to another. Interestingly, we show that

† This author's permanent position is INRIA-Rocquencourt, France. His work was supported by the Air Force Wright Laboratory Aeronautical Systems Center under ARPA Contract F33615-93-1-1339, and by the Air Force Rome Laboratories under ARPA Contract F30602-95-C-0119.

* This work was partially supported by EC Projects GoodStep and Opal and by the Israeli Ministry of Science

in most practical cases, translation rules can be automatically be derived from the correspondence rules. Thus, a complete integration task (derivation of correspondences, transformation of data from one world to the other, incremental integration of a new bulk of data, etc.) can be specified using a **single** declarative set of rules. This is an important result. It saves in writing different specifications for each sub-component of the integration task, and also helps in avoiding inconsistent specifications.

It should be noted that the language we use to define correspondence rules is very simple. Similar correspondences could be easily derived using more powerful languages previously proposed (e.g., LDL [5] or IQL [4]). But in these languages it would be much more difficult (sometimes impossible) to derive translation rules from given correspondence rules. Nevertheless, our language is expressive enough to describe many desired correspondences/translations, and in particular can express all the powerful document-OODB mappings supported by the *structuring schemas* mechanism of [2, 3].

As will be seen, correspondence rules have a very simple and intuitive graphical representation. Indeed, the present work serves as the basis for a system, currently being implemented, where a specification of integration of heterogeneous data proceeds in two phases. In a first phase, data is abstracted to yield a tree-like representation that is hiding details unnecessary to the restructuring (e.g., tags or parsing information). In a second phase, available data is displayed in a graphical window and starting from that representation, the user can specify correspondences or derive data.

The paper is organized as follows. Section 2 introduces a core data model and Section 3 a core language for specifying correspondences. In Section 4, we extend the framework to better deal with collections. Section 5 deals with the translation problem. The last section is a conclusion. More examples and figures are given in two appendixes.

2 The Data Model

Our goal is to provide a data model that allows declarative specifications of the correspondence between data stored in different worlds (DX, ODMG, SGML, etc.). We first introduce the model, then the concept of correspondence. To illustrate things we use below an example. A simple instance of an SGML document is given in Figure 1. A tree representation of the document in our middleware model, together with correspondences between this tree and a forest representation of a reference in a bibliographical OODB is given in Figure 2.

2.1 Data Forest

We assume the existence of some infinite sets: (i) **name** of names; (ii) **vertex** of vertexes; (iii) **dom** of data values. A *data forest* is a forest of *ordered labeled trees*. An *ordered labeled tree* is a tree with a labeling of vertexes and for each vertex, an ordering of its children. The internal vertexes of the trees have labels from **name** whereas the leaves have labels from **name** $\cup$ **dom** $\cup$ **vertex**. The only constraint is that if a vertex occurs as a leaf label, it should also occur as a vertex in the forest. Observe that this is a rather conventional tree structure. This is a data model in the spirit of complex value model [17, 1, 11] and many others, it is particularly influenced by models for unstructured data [21, 20] and the tree model of [7]. A particularity is the ordering of vertexes that is important to model data formats essentially described by files obeying a certain grammar (e.g., SGML).

Definition 1. A *data forest* F is a triple (E, G, L), where (E, G) is a finite *ordered* forest (the ordering is implicit); E is the set of vertexes; G the set of edges; L (the *labeling function*) maps some leaves in E to $E \cup$ **dom**; and all other vertexes to **name**.

For each vertex v in E, the maximal subtree of root v is called the *object* v. The set of vertexes E of a forest F is denoted $vertex(F)$ and the set of data values appearing in F is denoted $dom(F)$.

Remark. Observe that by definition, we allow a leaf to be mapped to a name. For all purposes, we may think of such leaves as internal vertexes without children. This will turn useful to represent for instance the empty set or the empty list. In the following, we refer by the word *leaf* only to vertexes v such that $L(v)$ is a vertex or is in **dom**.

We illustrate this notion as well as syntactic representations we use in an example. Consider the graphical representation of the forest describing the OODB, shown in the lower part of Figure 2. A

tabular representation of part of the same forest is given in Figure 3. Finally, below is the equivalent textual representation:

$$\begin{aligned}
&\&2\ reference\ \{\ \&21\ key\ \{\ \&211\ \text{“}ACM96\text{”}\ \{\ \}\ \},\\
&\qquad\qquad \&22\ title\ \{\ \&221\ \text{“}Correspondence...\text{”}\ \{\ \}\ \},\\
&\qquad\qquad \&23\ authors\ \{\ \&231\ \&3\ \{\ \},\ \&232\ \&4\ \{\ \},\ \&233\ \&5\ \{\ \}\ \},\\
&\qquad\qquad \&24\ abstract\ \{\ \&241\text{“}...\text{”}\ \{\ \}\ \}\ \}\\
&...
\end{aligned}$$

To get a more compact representation, we omit brackets when a vertex has a single or no children, and omit vertex identifiers when they are irrelevant for the discussion. For example the above reference tree may be represented by

$$\begin{aligned}
&\&2\ reference\ \{\ key\ \text{“}ACM96\text{”},\\
&\qquad\qquad \&22\ title\ \text{“}Correspondence...\text{”},\\
&\qquad\qquad authors\{\ \&3, \&4, \&5\ \},\\
&\qquad\qquad \&24\ abstract\ \text{“}...\text{”}\ \}
\end{aligned}$$

Let us now see how various common data sources can be mapped into our middleware model.

We consider here three different types of mappings. The first concerns relational databases, but also all simple table formats. The second is used for object-oriented databases, but a similar one will fit most graph formats. Finally, the last will fit any format having a BNF (or similar) grammar description. Note that the three mappings are invertible and can easily be implemented.

Relations can be represented by a tree whose root label is the relation name and which has as many children as rows in the relation. At depth 2, nodes represent rows and are labeled by the label "tuple". At depth 3, 4 and 5, nodes are labeled respectively by attribute names, types and values.

An object oriented database is usually a cyclic graph. However, using object identifier one may easily represents a cyclic graph as a tree [4].

We pick one possible representation but many other ones can be proposed. A class extent is represented by a tree whose root node is labeled with the class name. This node has as many children as there are objects in the extent, each of which is labeled by the object type. We assume that objects appear in the class extent of their most specific class. We now describe the representation of subtrees according to types.

- A node labeled by an atomic type has a unique child whose label is the appropriate atomic value.
- A node labeled "tuple" has one child for each attribute. The children are labeled with the attribute names and each has one child labeled by the appropriate type and having the relevant structure.
- A node labeled "set" (or "list", "bag", ...) has as many children as elements in the collection, one for each collection member. (For lists the order of elements is preserved). Each child is labeled by the appropriate type, and has the relevant structure.
- A node labeled by an object type has a unique child labeled by the identifier of the node that representing the object in the tree of the class extent to which it belongs.

A document can be described by a simplified representation of its parsing tree. The labels of the internal nodes (resp. leaves) represent the grammar non-terminal symbols (resp. tokens).

SGML and HTML, among other formats, allow references to internal and external data. Parsers do not interpret these references. They usually consider them as strings. In our context, these references should be interpreted when possible. As for object databases, the reference can be replaced by the identifier of the node containing the referred data.

Note that the only identification of data in the middleware model is given by the nodes identifiers. This means that it is the responsability of the data sources to keep relationships between the exported data and the node identifiers. This relationship is not always needed (e.g., for a translation process), and may be of a fine or large grain according to the application needs and the data source capacities.

The identification of data in the data sources can take various forms. It can be the key of a row or some internal address in relational databases. For object databases, it can be the internal oid (for

objects), a query leading to the object/value, or similar ideas as in the relational case. For files it can be an offset in the file, node in the parse tree, etc.

2.2 Correspondence

We are concerned with establishing/maintaining correspondences between objects. Some objects may come from one data source with particular forest F_1, and others from another forest, say F_2. To simplify, we consider here that we have a single forest (that can be viewed as the union of the two forests) and look for correspondences *within* the forest. (If we feel it is essential to distinguish between the sources, we may assume that the nodes of each tree from a particular data source have the name of that source, e.g., F_1, F_2, as part of the label.) We describe correspondences between objects using predicates.

Example 1. Consider the following forest with the SGML and OODB trees of Figure 2.

$\&1\ article$ {..., $\&12\ title$ "$Correspondence...$", $\&13\ author$ "$S.Abiteboul$",
$\&14\ author$ "$S.Cluet$", $\&15\ author$ "$T.Milo$", $\&16\ abstract$ "...", ... }
$\&2\ reference$ { key "$ACM96$", $\&22\ title$ "$Correspondence...$", $authors$ { $\&3, \&4, \&5$ },
$\&24\ abstract$ "..." }
$\&3\ author$ { fn "$S.$", ln "$Abiteboul$" }
$\&4\ author$ { fn "$S.$", ln "$Cluet$" }
$\&5\ author$ { fn "$T.$", ln "$Milo$" }

We may want to have the following correspondences:

{ $is(\&1, \&2)$, $is(\&12, \&22)$, $is(\&13, \&3)$, $is(\&14, \&4)$, $is(\&15, \&5)$, $is(\&16, \&24)$
$concat($"$S.Abiteboul$", "$S.$", "$Abiteboul$"), $concat($"$S.Cluet$", "$S.$", "$Cluet$"),
$concat($"$T.Milo$", "$T.$", "$Milo$") }.

Note that there is an essential difference between the two predicates above: *is* relates objects that represent the same real world entity, whereas *concat* is a standard concatenation predicate/function that is defined externally. The *is*-relationship is represented on Figure 2.

Definition 2. Let $\mathbf{R}$ be a relational schema. An $\mathbf{R}$-*correspondence* is a pair (F, I) where F is a data forest and I a relational instance over $\mathbf{R}$ with values in $vertex(F) \cup dom(F)$.

For instance, consider Example 1. Let $\mathbf{R}$ consists of binary relation *is* and a ternary one *concat*. For the forest F and correspondences I as in the example, (F, I) is an $\mathbf{R}$-correspondence. Note that we do not restrict our attention to 1-1 correspondences. The correspondence predicates may have arbitrary arity, and also, because of data duplication, some n-m correspondences may be introduced.

3 The Core Language

In this section, we develop the core language. This is in the style of rule-based languages for objects, e.g., IQL [4], LDL [5], F-logic [15] and more precisely, of MedMaker [19]. The language we present in this section is tailored to correspondence derivation, and thus in some sense more limited. However, we will consider in a next section a powerful new feature.

We assume the existence of two infinite sorts: a sort **data-var** of data variables, and **vertex-var** of vertex variables. Data variables start with capitals (to distinguish them from names); and vertex variables start with the character & followed by a capital letter.

Rules are built from correspondence literals and tree terms. Correspondence literals have the form $R(x_1, x_2, ..., x_n)$ where R is a relation name and $x_1, x_2, .., x_n$ are data/vertex variables/constants. Tree terms are of the form $\&X\ L$, $\&X\ L\ t_1$, and $\&X\ L\ \{t_1, t_2, ..., t_n\}$ where $\&X$ is a vertex variable/constant, L is a label and $t_1, t_2, ..., t_n$ are tree terms. The $\&X$ and Ls can also be omitted. A rule is obtained by distinguishing some correspondence literals and tree terms to be in the body, and some to be in the head. Semantics of rules is given in the sequel. As an example, consider the

following rule that we name r_{so}. Note again the distinction between *concat* which is a predicate on data values and can be thought of as given by extension or computed externally, and the derived *is* correspondence predicate.

$$
\begin{array}{ll}
 & \&X_0\ article\ \{\ \&X_1, \\
 & \qquad \&X_2\ title\ X_3, \\
 & \qquad \&X_4\ author\ X_5,\ \&X_6\ author\ X_7,\ \&X_8\ author\ X_9, \\
 & \qquad \&X_{10}\ abstract\ X_{11}, \\
 & \qquad \&X_{12}\ \} \\
is(\&X_0, \&X_{13}) & \&X_{13}\ reference\ \{\ \&X_{14}, \\
is(\&X_2, \&X_{15}) & \qquad \&X_{15}\ title\ X_3, \\
is(\&X_{10}, \&X_{19}) & \qquad authors\{\ \&Y_{16}, \&X_{17}, \&X_{18}\ \}\ , \\
is(\&X_4, \&X_{16}) \leftarrow & \qquad \&X_{19}\ abstract\ X_{11}\ \} \\
is(\&X_6, \&X_{17}) & \&X_{16}\ author\ \{\ fn\ X_{20},\ ln\ X_{21}\ \} \\
is(\&X_8, \&X_{18}) & \&X_{17}\ author\ \{\ fn\ X_{22},\ ln\ X_{23}\ \} \\
 & \&X_{18}\ author\ \{\ fn\ X_{24},\ ln\ X_{25}\ \} \\
 & \\
 & concat(X_5, X_{20}, X_{21}) \\
 & concat(X_7, X_{22}, X_{23}) \\
 & concat(X_9, X_{24}, X_{25})
\end{array}
$$

A *rule* consists of a body and a head. When a rule has only literals in its head, it is said to be a *correspondence rule*. We assume that all variables in the head of a correspondence rule also occur in the body. We now define the semantics of correspondence rules.

Definition 3. Given an instance (F, I) and some correspondence rule r, a *valuation* ν over (F, I) is a mapping over variables in r such that

1. ν maps data variables to $dom(F)$ and object variables to $vertex(F)$.
2. For each term H in the body of r
   (a) H is a correspondence literal and $\nu(H)$ is true in I; or
   (b) H is a tree term and $\nu(H)$ is an object[5] of F.

We say that a correspondence $C(\&U, \&V)$ is *derived from* (F, I) *using* r if $C(\&U, \&V) = \nu(H)$ for some term H in the head of r, and some valuation ν over (F, I).

Let $\mathcal{P}$ be a set of rules. Let $I' = \{H' \mid H'$ derived from (F, I) using some r in $\mathcal{P}\}$. Then, $(F, I \cup I')$ is denoted $T_{\mathcal{P}}(F, I)$. If $\mathcal{P}$ is recursive, we may be able to apply $T_{\mathcal{P}}$ to $T_{\mathcal{P}}(F, I)$ to derive new correspondences. The limit $T^{\omega}_{\mathcal{P}}(F, I)$, when it exists, of the application of $T_{\mathcal{P}}$ is denoted, $\mathcal{P}(F, I)$.

Theorem 4. *For each (possibly recursive) finite set $\mathcal{P}$ of correspondence-rules and each data forest (F, I), $\mathcal{P}(F, I)$ is well-defined (in particular, the sequence of applications of $T_{\mathcal{P}}$ converges in a finite number of stages). Furthermore, $\mathcal{P}(F, I)$ can be computed in* PTIME.

Sketch: We represent data forests using a relational database. A relation *succ* gives a portion of the successor function over the integers. The number of facts that can be derived is polynomial. Each step can be computed with a first-order formula, so it is in PTIME. □

The above rule r_{so} is an example of a non-recursive correspondence rule. (We assume that the extension of *concat* is given in I.) To see an example of a recursive rule, we consider the correspondence between "left-deep" and "right-deep" trees. For instance, we would like to derive a correspondence between the right and left deep trees shown in Figure 4. This is achieved using the program *r2l* which consists of the following rules:

[5] Recall that an object of a forest F is a maximal subtree of F rooted in some vertex of F.

$$r2l: \quad \frac{R2L(\&U, \&V) \leftarrow \begin{array}{l} \&U\ right\{\} \\ \&V\ left\{\} \end{array}}{R2L(\&U, \&V) \leftarrow \begin{array}{l} \&U\ right\{X,\ \&Y\} \\ \&V\ left\{\&Z,\ X\} \\ R2L(\&Y, \&Z) \end{array}}$$

Suppose that we start with $I = \emptyset$, and the forest F shown on Figure 4. Then we derive the correspondences $R2L(\&3, \&3')$, $R2L(\&2, \&2')$, $R2L(\&1, \&1')$. The computation is:

$$T^1_{r2l}(F, \emptyset) = F \cup R2L(\&3, \&3') \qquad T^2_{r2l}(F, \emptyset) = T^1_{r2l}(F, \emptyset) \cup R2L(\&2, \&2')$$
$$T^3_{r2l}(F, \emptyset) = T^2_{r2l}(F, \emptyset) \cup R2L(\&1, \&1') \qquad r2l(F, \emptyset) = T^3_{r2l}(F, \emptyset)$$

This kind of deep trees is frequent in data exchange formats and it is important to be able to handle them. However, what we have seen above is not quite powerful enough. It will have to be extended with particular operations on trees and to handle data collections. This is described next.

4 Dealing with Collections

When data sources are mapped into the middleware model, some forest vertexes may represent data collections. Observe that, in the above rules, the tree describes vertexes with a bounded number of children, (where the number depends on the term structure). Data collections may have an arbitrary number of members, and thus we need to extend our language to deal with vertexes having arbitrary number of children. Also observe that ordered trees are perfect to represent ordered data collections such as lists or arrays. However, if we want to model database constructs such as sets or bags, we have to consider properties such as insensitivity to order or duplicates. The rules that we developed so far do not support this. In this section, we address these two issues by extending our framework to incorporate (i) operators on trees and (ii) special collection properties.

4.1 Tree Constructors

We consider two binary operations on trees. The first, $cons(T_1, T_2)$, takes two objects as input. The first one is interpreted as an element and the second as the collection of its children vertexes. The operation adds the element to the collection. The second operator, *merge*, allows to merge two data collections into one. (The *cons* operator can be defined using a merge with a singleton collection.) For example

$$mylist\{1, 2, 3, 4\} \equiv cons(1, mylist\{2, 3, 4\}) \equiv$$
$$merge(mylist\{1\}, mylist\{2, 3, 4\}) \equiv merge(mylist\{1, 2\}, mylist\{3, 4\})$$

More formally, let T, T', T'' be some trees where the roots of T' and T'' have the same label l and children $S'_1, ..., S'_n$ and $S''_1, ..., S''_m$ respectively. Then

- $cons(T, T')$ is a tree with root labeled by l and children $T, S'_1, ..., S'_n$, in that order.
- $merge(T', T'')$ is a tree with root labeled by l and children $S'_1, ..., S'_n, S''_1, ..., S''_m$, in that order.

The *cons* and *merge* operators provide alternative representations for collections that are essential to describe restructuring. The data trees in the forests we consider are all reduced in the sense that they will not include *cons* or *merge* vertexes. But, when using the rules, we are allowed to consider alternative representations of the forest trees. The vertexes/objects of the trees with *cons* and *merge* are regarded as implicit. So, for instance if we have the data tree $\&1\ mylist\{\&2, \&3\}$, we can view it as $\&1\ cons(\&2, \&v)$ where the object $\&v$ is implicit and has the structure $mylist\{\&3\}$. Indeed, we will denote this object $\&v$ by $mylist(\&1, \&3)$ to specify that it is an object with label *mylist*, that it is a subcollection of $\&1$, and that it has a single child $\&3$. This motivates the following definition:

Definition 5. Given a forest F, a vertex $\&v$ in F with children $\&v_1, ..., \&v_n$ (for $0 \leq n$) and label l, the expression $l(\&v, \&v_i, \&v_{i+1}, ..., \&v_j)$ is called an *implicit object* of F for each subsequence[6] $\&v_i, \&v_{i+1}..., \&v_j$ of $\&v_1, ..., \&v_n$. The set of all implicit objects of F is denoted $impl(F)$.

[6] A *subsequence* $\&v_i, \&v_{i+1}..., \&v_j$ of $\&v_1, ..., \&v_n$ is obtained by removing 0 or more elements from the head and the tail of $\&v_1, ..., \&v_n$.

Observe that $vertex(F)$ can be viewed as a subset of $impl(F)$ if we identify the object $l(\&v, \&v_1, ..., \&v_n)$ of the definition, with $\&v$. Observe also that the cardinality of $impl(F)$ is polynomial in the size of F.

We can now use *cons* and *merge* in rules. The following example uses *cons* to define a correspondence between a list structured as a right-deep tree and a list structured as a tree of depth one (Observe that in the example *mylist* is not a keyword but only a name with no particular semantics; *cons* is a keyword with semantics, the cons operation on trees):

$$tl: \frac{\begin{array}{ll} TreeList(\&U, \&V) \leftarrow & \&U\ R\{\} \\ & \&V\ mylist\{\} \end{array}}{\begin{array}{ll} TreeList(\&U, \&V) \leftarrow & \&U\ R\{Z,\ \&X\} \\ & \&V\ cons(Z,\ \&Y) \\ & TreeList(\&X, \&Y) \end{array}}$$

Of course, to use such rules, we have to extend the notion of valuation to allow terms containing *cons*. The new valuation may now assign implicit objects to object variables.

The fixpoint $T^{\omega}_{\mathcal{P}}(F, I)$ is computed as before using the new definition of valuation. Observe that $T^{i}_{\mathcal{P}}(F, I)$ may now contain correspondences involving vertexes in $impl(F)$ and not only F. Since we are interested only in correspondences between vertexes in F, we ultimately ignore all other correspondences. So, $\mathcal{P}(F, I)$ is the restriction of $T^{\omega}_{\mathcal{P}}(F, I)$ to objects in F. For instance, consider rule tl and $F = \{\ \&1\ R\{a, \&2\ R\{b, \&3\ R\{\}\}\}\ ,\ \ \&1'\ mylist\{a, b\}\ \}$.
Then:

$impl(F) = F \cup \{mylist(\&1',\ ), mylist(\&1', a), mylist(\&1', b), R(\&1), R(\&1, a), etc.\}$
$T^{1}_{tl}(F, \emptyset) = TreeList(\&3, mylist(\&1',\ ))$
$T^{2}_{tl}(F, \emptyset) = T^{1}_{tl}(F, \emptyset) \cup TreeList(\&2, mylist(\&1', b))$
$T^{\omega}_{tl}(F, \emptyset) = T^{2}_{tl}(F, \emptyset) \cup TreeList(\&1, mylist(\&1', a, b)) = T^{2}_{tl}(F, \emptyset) \cup TreeList(\&1, \&1')$
$tl(F, \emptyset) = (F, TreeList(\&1, \&1'))$

In the sequel, we call the problem of computing $\mathcal{P}(F, I)$, the *matching problem*.

Theorem 6. *The matching problem is in* PTIME *even in the presence of* cons *and* merge.

Sketch: The number of facts that can be derived is polynomial and each step can be computed with a first-order formula, so is polynomial. □

4.2 Special Properties

Data models of interest include collections with specific properties: e.g., sets that are insensitive to order or duplicates, bags that are insensitive to order. In our context this translates to properties of vertexes with particular labels. We consider here two cases, namely insensitivity to order (called bag property), and insensitivity to both order and duplicates (called set property). For instance, we may decide that a particular label, say *mybag* (resp. *myset*) denotes a bag (resp. a set). Then, the system should not distinguish between:

$$cons(a, cons(a, mybag\ \{b\})) \equiv cons(a, cons(b, mybag\ \{a\}))$$
$$cons(a, cons(a, myset\ \{b\})) \equiv cons(a, myset\ \{b\})$$

The fact that these should be the same implicit objects is fundamental. (Otherwise the same set would potentially have an infinite number of representations and computing correspondences would become undecidable.) In the context of set/bag properties, the definition of implicit objects becomes a little bit more intricate.

Definition 7. Given a forest F, a vertex $\&v$ in F with children $\&v_1, ..., \&v_n$ (for $0 \leq n$) and label l, implicit objects of vertexes with bag/set properties are obtained as follows:
l has *set* property: $l(\&v, \&v_{i_1}, ..., \&v_{i_j})$ for each subset $\{\&v_{i_1}, ..., \&v_{i_j}\}$ of $\{\&v_1, ..., \&v_n\}$.
l has *bag* property: $l(\&v, \&v_{i_1}, ..., \&v_{i_j})$ for each subbag $\{\{\&v_{i_1}, ..., \&v_{i_j}\}\}$ of $\{\{\&v_1, ..., \&v_n\}\}$.

The notion of valuation is extended in a straightforward manner to use the above implicit objects and take into consideration tree equivalence due to insensitivity to order and duplicates, (details omitted for lack of space). It is important to observe at this point that the number of implicit objects is now exponential in the size of F.

The next example shows how *cons*, and the set property can be used to define a correspondence between a list and a set containing one copy for each distinct list member:

$$ls: \quad \begin{array}{lll} \text{label } myset & : & \text{set} \\ \hline ListSet(\&U, \&V) & \leftarrow & \&U\ mylist\ \{\} \\ & & \&V\ myset\ \{\} \\ \hline ListSet(\&U, \&V) & \leftarrow & \&U\ cons(Z,\ \&X) \\ & & \&V\ cons(Z,\ \&Y) \\ & & ListSet(\&X, \&Y) \end{array}$$

Observe the symmetry of the rules between set and list. The only distinction is in the specification of label *myset*. Using essentially the same proof as in Theorem 6 and a reduction to 3-sat, one can prove:

Theorem 8. *In the presence of* cons, merge, *and collections that are insensitive to order/duplicates, the matching problem can be solved in* EXPTIME. *Even with insensitivity to order and cons only, the matching problem becomes* NP*-hard.*

Remark. The complexity is data complexity. This may seem a negative result (that should have been expected because of the matching of commutative collections). But in practice, merging is rarely achieved based on collections. It is most often key-based and, in some rare cases, based on the matching of "small collections", e.g., sets of authors.

To conclude the discussion of correspondence rules, and demonstrate the usage of *cons* and *merge*, let us consider the following example where a correspondence between articles and OO references is defined. Observe that while the correspondence rule r_{so} presented at the beginning of the paper handles articles with exactly three authors, articles/references here deal with arbitrary number of authors. They are required to have the same title and abstract and the same author list (i.e., the authors appear in the same order). The definition uses an auxiliary predicate *same_list*.

The first rule defines correspondence between authors. The second and third rules define an auxiliary correspondence between sequences from both world. It is used in rule R_4 that defines correspondence between articles and references. It also defines correspondence between titles and abstracts from both worlds.

$$r_1 : is(\&X_0, \&X_2) \leftarrow \begin{array}{l} \&X_0\ author\ X_1 \\ \&X_2\ author\ \{\ fn\ X_3,\ ln\ X_4\ \} \\ concat(X_1, X_3, X_4) \end{array}$$

$$r_2 : same_list(\&X_0, \&X_2) \leftarrow \begin{array}{l} \&X_0\ cons(\&X_1,\ \&X_2) \\ \&X_3\ cons(\&X_4,\ \&X_5) \\ is(\&X_1, \&X_4) \\ same_list(\&X_2, \&X_5) \end{array}$$

$$r_3 : same_list(\&X_0, \&X_3) \leftarrow \begin{array}{l} \&X_0\ \{\} \\ \&X_3\ \{\} \end{array}$$

$$r_4 : \begin{array}{l} is(\&X_0, \&X_8) \\ is(\&X_2, \&X_{10}) \\ is(\&X_5, \&X_{12}) \end{array} \leftarrow \begin{array}{l} \&X_0\ merge(\ \{\ \&X_1, \&X_2\ title\ X_3\}, \\ \qquad merge(\&X_4, \\ \qquad\quad article\{\&X_5\ abstract\ X_6, \&X_7\ \}\ )\ ) \\ \&X_8\ reference\ \{\ \&X_9, \\ \qquad \&X_{10}\ title\ X_3, \\ \qquad \&X_{11}, \\ \qquad \&X_{12}\ abstract\ X_6\ \ \} \\ same_list(\&X_4, \&X_{11}) \end{array}$$

We illustrated the language using rather simple examples. Nevertheless, it is quite powerful and can describe many desired correspondences, and in particular all the document-OODB mappings supported by the *structuring schemas* mechanism of [2, 3] (Omitted).

5 Data Translation

Correspondence rules are used to derive relationships between vertexes. We next consider the problem of translating data. We first state the general translation problem (that is undecidable). We then introduce a decidable subcase that captures the practical applications we are interested in. This is based on *translation rules* obtained by moving tree terms from the body of correspondence rules to the head.

We start with a data forest and a set of correspondence rules. For a particular forest object $\&v$ and a correspondence predicate C, we want to know if the forest can be extended in such a way that $\&v$ is in correspondence to some vertex $\&v'$. In some sense, $\&v'$ could be seen as the "translation" of $\&v$. This is what we call the *data translation problem*.

input: an **R**-correspondence (F, I), a set $\mathcal{P}$ of correspondence rules, a vertex $\&v$ of F, and a binary predicate C.

output: an extension F' of F such that $C(\&v, \&v')$ holds in $\mathcal{P}(F', I)$ for some $\&v'$; or *no* if no such extension exists.

For example consider a forest F with the right deep tree $\&1\ \{1, \{2, \{3, \{\}\}\}\}$. Assume we want to translate it into a left deep tree format. Recall that the $r2l$ correspondence rules define correspondences between right deep trees and left deep trees. So, we can give the translation problem the **R**-correspondence (F, I), the root vertex $\&1$, and the correspondence predicate $R2L$. The output will be a forest F' with some vertex $\&v'$ s.t. $R2L(\&1, \&v')$ holds. The tree rooted at $\&v'$ is exactly the left deep tree we are looking for.

Remark. In the general case: (i) we would like to translate an entire collection of objects; and (ii) the correspondence may be a predicate of arbitrary arity. To simplify the presentation, we consider the more restricted problem defined above. The same techniques work for the general case with minor modifications.

It turns out that data translation is in general very difficult. (The proof is by reduction of the acceptance problem of Turing machines.)

Proposition 5.1 *The translation problem is undecidable, even in absence of cons, merge, and labels with set/bag properties.*

Although the problem is undecidable in general, we show next that translation is still possible in many practical cases and can often be performed efficiently. To do this, we impose two restrictions:

1. The first restriction we impose is that we separate data into two categories, input vertexes and output vertexes. Vertex variables and labels are similarly separated[7]. We assume that the presence of an output object depends solely on the presence of some input object(s) and possibly some correspondence conditions. It allows us to focus on essentially one kind of recursion: that found in the source data structure.
2. The second restriction is more technical and based on a property called *body restriction* that is defined in the sequel. It prevents pathological behavior and mostly prevent correspondences that relate "inside" of tree terms.

These restrictions typically apply when considering data translation or integration, and in particular we will see that all the examples above have the appropriate properties.

The basic idea is to use correspondence rules and transform them into translation rules by moving data tree terms containing output variables from the body of rules to their head. For example, consider the $r2l$ correspondence rules. To translate a right deep tree into a left deep tree, we move the terms of the left deep trees to the head of rules, and obtain the following translation rules. (Variables with prime are used to stress the separation between the two worlds.)

$$r2l' : \frac{\begin{array}{l} r : \&U'\ L\{\} \\ \quad R2L(\&U, \&U') \end{array} \leftarrow \&U\ R\{\}}{\begin{array}{l} r' : \&U'\ L\{\&Y',\ X\} \\ \quad R2L(\&U, \&U') \end{array} \leftarrow \begin{array}{l} \&U\ R\{X,\ \&Y\} \\ R2L(\&Y, \&Y') \end{array}}$$

Of course we need to extend the semantics of rules. The tree terms in the head, and in particular those containing variables that do not appear in the body, are used to create new objects. (This essentially will do the data translation). We use Skolem functions in the style of [9, 14, 16, 18] to denote new object ids. There are some difficulties in doing so. Consider a valuation ν of the second rule above. One may be tempted to use the Skolem term $r'(\nu(\&U), \nu(X), \nu(\&Y), \nu(\&Y'))$ to denote the new object. But since $\&Y'$ is itself a created object, this may lead to a potentially non-terminating loop of object creation. To avoid this we choose to create objects only as a function of **input** objects and not of new output created objects. (Thus in the above case the created object is denoted $r'\&U'(\nu(\&U), \nu(X), \nu(\&Y))$.)

Now, the price for this is that (i) we may be excluding some object creation that could be of interest; and (ii) this may result in inconsistencies (e.g., the same object with two distinct values). We accept (i), although we will give a class of programs such that (i) never occurs. For (ii), we rely on non determinism to choose one value to be assigned to one object. Note that we need some form of nondeterminism for instance to construct a list representation from a set.

For lack of space we do not give here the refined semantics for rules, (it is given in the full paper,) but only state that:

Proposition9. *For each finite set $\mathcal{P}$ of translation-rules and each* **R***-correspondence (F, I), each of the possible sequences of application of $T_{\mathcal{P}}$ converges in a finite number of stages. Furthermore, for rules with no set/bag labels each sequence converges in* PTIME*, and otherwise in* EXPTIME.

So far, a program computation can be viewed as purely syntactic. We are guaranteed to terminate, but we don't know the semantic properties of the constructed new objects and derived correspondences.

It turns out this evaluation of translation rules allows to solve the translation problem for a large class of correspondence rules. (Clearly not all since the problem is unsolvable in general.) In

[7] Note that vertexes can easily be distinguished using their label.

particular it covers all rules we presented in the previous section, and the translations specified by the structuring schemas mechanisms of [3] (proof omitted). We next present conditions under which the technique can be used.

Definition 10. A correspondence rule r is said to be *body restricted* if in its body (1) all the variables in correspondence literals are leaves of tree terms, and each such variable has at most one occurrence in a correspondence literal, and (2) non-leaf variables have at most one occurrence (as non leafs) in tree terms, and (3) the only variables that input and output tree terms share are leaf variables.

We are considering correspondences specified with input/output data forests.

Proposition 5.2 *Consider an input/output context. Let $\mathcal{P}$ be a set of body restricted correspondence rules where correspondence literals always relate input to output objects. Let (F, I) be an R-correspondence where F is an input data forest, $\&v$ a vertex in F, and C a binary correspondence predicate. Let $\mathcal{P}'$ be the translation rules obtained from $\mathcal{P}$ by moving all output tree terms to the head of rules. Then,*

- *If the translation problem has a solution on input (F, I) $\mathcal{P}$, $\&v$, C that leaves the input forest unchanged, then each possible computations of $\mathcal{P}'(F, I)$ derives $C(\&v, \&v')$ for some output object $\&v'$.*
- *if some computation of $\mathcal{P}'$ derives $C(\&v, \&v')$ for some object $\&v'$, then the forest F' computed by this computation is a correct solution to the translation problem.*

By Proposition 5.2, to solve the translation problem (with unmodified input) for body restricted rules, we only need to compute nondeterministically one of the possible outputs of $\mathcal{P}$, and test if $C(\&v, \&v')$ holds for some $\&v'$.

6 Conclusion

We presented a specification of the integration of heterogeneous data based on correspondence rules. We showed how a unique specification can served many purposes (including two-way translation) assuming some reasonable restrictions. We claim that the framework and restrictions are acceptable in practice, and in particular one can show that all the document-OODB correspondences/translations of [2, 3] are covered. We are currently working on further substantiating this by more experimentation.

When applying the work presented here a number of issues arise such as the specification of default values when some information is missing in the translation. A more complex one is the introduction of some simple constraints in the model, e.g., keys.

Another important implementation issue is to choose between keeping one of the representations virtual vs. materializing both. In particular, it is conceivable to apply in this larger setting the optimization techniques developed in a OODB/SGML context for queries [2] and updates [3].

Acknowledgment: We thank Catriel Beeri for his comments on a first draft of the paper.

References

1. S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. Technical report, INRIA and the department of computer science of the Hebrew University of Israel, 1987.
2. S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In *Proc. VLDB*, 1993.
3. S. Abiteboul, S. Cluet, and T. Milo. A database interface for files update. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, San Jose, California, 1995.
4. S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 159–173, 1989. to appear in *J. ACM*.
5. C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli, and S. Tsur. Sets and negation in a logic database language (LDL1). In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 21–37, 1987.

6. P. Buneman, S. Davidson, K. Hart, C. Overton, and L. Wong. A data transformation system for biological data sources. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 158–169, Zurich, Switzerland, 1995.
7. P. Buneman, S. Davidson, and D. Suciu. Programming constructs for unstructured data, May 1996.
8. M.J. Carey et al. Towards heterogeneous multimedia information systems: The Garlic approach. Technical Report RJ 9911, IBM Almaden Research Center, 1994.
9. T.-P. Chang and R. Hull. Using witness generators to support bi-directional update between object-based databases. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, San Jose, California, May 1995.
10. V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proc. ACM Sigmod, Minneapolis*, 1994.
11. O. Deux. The story of O_2. *IEEE Trans. on Data and Knowledge Eng.*, 2(1):91–108, March 1990.
12. J.C. Franchitti and R. King. Amalgame: a tool for creating interoperating persistent, heterogeneous components. *Advanced Database Systems*, pages 313–36, 1993.
13. M. Gyssens, J.Paredaens, and D. V. Gucht. A grammar based approach towards unifying hierarchical data models. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 1989.
14. R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object-identifiers. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, Brisbane, Australia, Aug 1990.
15. M. Kifer and G. Lausen. F-logic: A higher-order language for reasoning about objects. In *sigmod*, 1989.
16. M. Kifer, G. Lausen, and Wu James. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, MAY 1995.
17. G. M. Kuper and M. Y. Vardi. The logical data model. *ACM Trans. on Database Systems*, 18(3):379–413, Sept 1993.
18. D. Maier. A logic for objects. Technical Report TR CS/E-86-012, Oregon Graduate Center, Nov 1986.
19. Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specifications. Available by anonymous ftp at `db.stanford.edu` as the file `~/pub/papakonstantinou/1995/medmaker.ps`.
20. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *International Conference on Data Engineering*, 1995.
21. D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. Technical report, Stanford University, 1995. Available by anonymous ftp from `db.stanford.edu`.

```
<article status=final>
<title> Correspondence and Translation for Heterogeneous Data < \title>
<author> Serge Abiteboul
<author> Sophie Cluet
<author> Tova Milo
<abstract>
A primary motivation for new database technology is to provide support
for the broad spectrum of multimedia data available notably through
...
< \abstract>
<body>
...
< \body>
< \article>
```

Fig. 1. An instance of an SGML document

This article was processed using the LaTeX macro package with LLNCS style

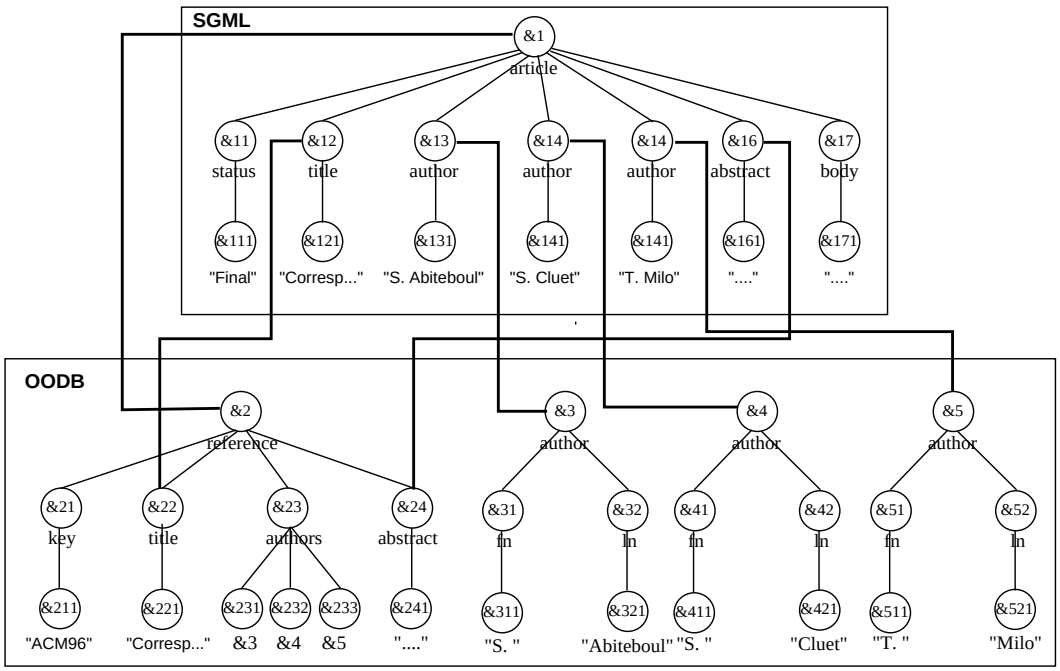

Fig. 2. Correspondence between SGML file and an OODB

.

vertex label		*children*
&2	*reference*	{&21, &22, &23}
&21	*key*	{&211}
&211	"*ACM96*"	{ }
&22	*title*	{&221}
&221	"*Correspondence...*"	{ }
&23	*authors*	{&231, &232, &233}
&231	&3	{ }
&232	&4	{ }
&233	&5	{ }
&24	*abstract*	{&241}
&241	"..."	{ }
&3	*author*	{&31, &32}
&31	*fn*	{&311}
&311	"*S.*"	{ }
&32	*ln*	{&321}
&321	"*Abiteboul*"	{ }
...		

Fig. 3. A Data Forest

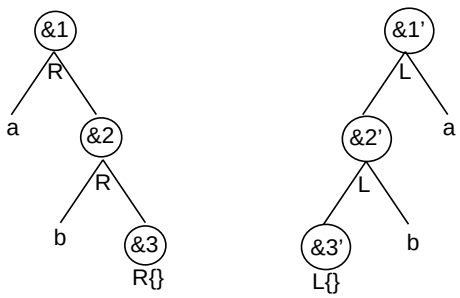

Fig. 4. Right- and Left-Deep Trees