Using Schema Matching to Simplify Heterogeneous Data Translation *

Tova Milo
Tel Aviv University
milo@math.tau.ac.il

Sagit Zohar
Tel Aviv University
sagit@math.tau.ac.il

Abstract

A broad spectrum of data is available on the Web in distinct heterogeneous sources, and stored under different formats. As the number of systems that utilize this heterogeneous data grows, the importance of data translation and conversion mechanisms increases greatly. In this paper we present a new translation system, based on schema-matching, aimed to simplify the intricate task of data conversion. We observe that in many cases the schema of the data in the source system is very similar to the that of the target system. In such cases, much of the translation work can be done automatically, based on the schemas similarity. This saves a lot of effort for the user, limiting the amount of programming needed. We define common schema and data models, in which schemas and data (resp.) from many common models can be represented. Using a rule-based method, the source schema is compared with the target one, and each component in the source schema is matched with a corresponding component in the target schema. Then, based on the matching achieved, data instances of the source schema can be translated to instances of the target schema. We show that our schema-based translation system allows a convenient specification and customization of data conversions, and can be easily combined with the traditional data-based translation languages.

1 Introduction

Data integration and translation is a problem facing many organizations that wish to utilize Web data. A broad spectrum of data is available on the Web in distinct heterogeneous sources, stored under different formats: a specific database vendor format, SGML or Latex (documents), DX formats (scientific data), Step (CAD/CAM data), etc. Their integration is a very active field of research, (see for instance, for a very small sample, [18, 12, 16, 15, 27, 24, 17, 3, 4]). A key observation is that, often, the application programs used by organizations can only handle data of a specific format. (e.g. Web browsers, like Netscape, expect files in HTML format, and relational databases expect relations). To enable a specific tool to manipulate data coming from various sources (e.g. use, in a relational system, data stored on the Web in HTML format), a translation phase must take place – the data (in the source format) needs to be mapped to the format expected by the application.

The naive way to translate data from one format to another is writing a specific program for each translation task. Examples are the Latex to HTML translator [22] or the HTML

*The work is supported by the Israeli Ministry of Science and by the Academy of Arts and Sciences

to text translator [1]. Writing such a program is typically a non trivial task which is often complicated by numerous technical aspects of the specific data sources that are not really relevant to the translation process (e.g. HTML or SGML parsing, or specific database access protocol). Recent works [4, 19] consider a more general framework which enables a more flexible translation between various models. The solution is based on using a common data model to which the source/target data is mapped, and providing a common translation language which enables the specification and customization of the translation task. This makes the introduction of new translations easier, but still very often requires a considerable programming effort whenever a new translation is to be defined[4].

The goal of this work is to design a mechanism for simplifying the specification of translations. The base observation is that, frequently, much of the structure of the source data is very similar to that of the target translated data, and many of the structure modifications to be performed by the translation process are rather standard and result from various differences between the schemas of the source and the target systems. We use here the general term *schema* to denote whatever way a data model chooses to model its data. For example, databases use schemas to model database instances; Structured documents often obey some grammar (e.g. Document Type Definition – DTD – in SGML and HTML); In other models such a definition may be partial (e.g. in semi-structured data [2]). The observation is that, in many translations, the schema of the target system is closely related to that of the source system – both schemas aim to represent the same data. This implies that a large part of the translation process can be done *automatically*, relaying on this (often standard) relationship, hence reducing the programming effort, and involving the user only in the specification of the "non standard" parts of the translation.

We built a data translation system, called *TranScm*, which implements the above idea. Given the schemas for the source and target data, the system examines the schemas and tries to find similarities/differences between them. This is done using a rule-based method, where each rule (1) defines a possible common matching between two schema components, and (2) provides means for translating an instance of the first to an instance of the second. The system has a set of build-in rules that handles most common cases, and that can be extended/adjusted/overridden by the user during the translation process. The system uses the rules and tries to find for each component of the source schema a unique "best matching" component in the target schema, or determine that the component should not be represented in the target. This is called the *matching process*. If the process succeeds, the data translation can be performed automatically using the translation facilities of the matching rules. There are two cases where the process may fail: (i) a component of the source schema cannot be matched with a target one using the current set of rules, (and the matching process can neither derive that the component should be just ignored), or (ii) a component of the source schema matches several components in the target schema, and the system cannot automatically determine which is the "best" match. For (i) the user can add rules to the system to handle the special component and describe the translation to be applied to it. For (ii) the user is asked to determine the best match. Then, based on the user's input, the matching process is completed and the translation is enabled.

Note that the schema-based data translation method that we propose is not aimed to *replace* the programming languages for data translation proposed in [4, 19], but rather to *complement* them. The idea is that rather than having to write a translation program for the whole data, much of the translation specification will be done automatically by the system, based on the schema matching, and the programmer will only need to supply some minimal additional code to handle the data components not covered the system. Hence the programming effort will be greatly simplified.

The focus of the work is on the system architecture and the modular use of schema-based match&translate rules, and not on the specific language used to define the rules. In fact, we provide a generic interface for rules and the presentation is independent of the specific language used to specify them. For implementation reasons we used in the prototype Java as the rule definition language, but if desired, the user can use declarative rule languages in the style of [4, 19], thus enabling logic-based inference of the properties and correctness of rules. This is beyond the scope of this paper.

Handling data and schemas from different models requires a common framework in which the different schema and data formats can be presented. For that we defined a *middleware* schema and data models in which the matching process and the data translation are performed. The schema model consists of graphs, and the data model consists of labeled forests and is similar to the one introduced in [4] and to the OEM and the tree models of [28, 12]. The difference with the OEM model is that we allow some nodes to be ordered. This is crucial for modeling data that might be ordered (e.g. structured documents). Each data source that is to be exposed to the Web community is expected to provide a mapping to/from the middleware format. As we shall see, the representation of each source inside the middleware is very close to the structure of data/schema in the source, so the implementation of such a mapping is fairly easy.

The paper is organized as follows. We start with a general overview of the system in Section 2. Then Section 3 presents the middleware data and schema models. In Section 4 we describe the match&translate rules used to determine the matching between the schema components and the data translation derived from the. We also explain the user interaction with the system and the means for customizing the translations. The system architecture and implementation are considered in Section 5. Finally, we conclude in Section 6 by considering related work.

2 System Overview

A typical scenario of the system's work is as follows. It receives as input two schemas, one of the data source and the other of the target. The two schemas are imported into the system and presented in the common schema model. The next step is matching. The system tries to find for every component in the source schema a corresponding component in the target schema (or determine that the component should not be represented in the target), using the rule-based process mentioned in the Introduction. Once the matching is completed (perhaps with the user's assistance), a data translation is possible. To perform the translation, a data instance of the source schema is imported to the common data model, and is "typed", i.e. every element in the data is attached a corresponding schema element as a type. Now, the system uses the match between the schema components, achieved in the previous step, to translate the data: Recall from the Introduction that every rule in the system has two components, the first defines a possible common matching between two components of schemas, and the second provides means for translating an instance of the

```
<!DOCTYPE  atricle [
<!ELEMENT  article   (title, authors, sections) >
<!ELEMENT  authors   (author+) >
<!ELEMENT  sections  (section*) >
<!ELEMENT  title     (#PCDATA) >
<!ELEMENT  author    (#PCDATA) >
<!ELEMENT  section   (section1 | section2) >
<!ELEMENT  section1  (title, body)) >
<!ELEMENT  section2  (picture, caption?) >
<!ELEMENT  body      (parag*) >
<!ELEMENT  picture   (#PCDATA) >
<!ELEMENT  caption   (#PCDATA) >
<!ELEMENT  parag     (#PCDATA) >
```

Figure 1: SGML DTD

first to an instance of the second. Every element of the source data is translated, using the translation function of the rule that matched its type with a type (component) of the target schema, to an instance of the target type. The resulting elements are "glued" together to form a valid instance of the target schema. Finally, the translated data is exported to the target application.

We demonstrate the above process with an example. We assume below some basic knowledge on SGML[23] and OODBs, and consider the translation of data between these formats. A brief introduction to SGML is presented in the Appendix. The example we use is a simplified version of the example described in [18]. (The full example can be handled similarly, the simplification is only for space reasons.) We ignore for now the representation of these formats in the middleware models (this will be considered in the next section) and concentrate on the matching and translation steps.

Consider the SGML DTD in Figure 1, the SGML document in Figure 2, and the OODB schema in figure 3. We would like to translate the SGML document (which is an instance of the mentioned DTD) to an instance of the OODB schema. Note that the DTD and the OO schema are quite similar. We will describe some possible matches between their components, that can be determined by an automatic rule-based system. For every such match we will detail the difference between the structure of the components, and suggest a possible translation function for mapping instances of the first to instances of the second.

- The *article* element in the SGML DTD is basically an ordered tuple. The most "similar" element in the OO schema is the *Article* class. (The two components have the same name, up to capital letters, and similar component, and as will be explained below, the structure of the components also match). A difference is that tuple attributes in OODBs are not ordered. The translation function is this case is rather simple – an *Article* instance in the OODB will be built from the input one by simply omitting the order information.

- The *section* element in the DTD describes a union type. In the OO schema, the most similar element seems to be the *Section* class: This class describes a 3-ary tuple, where the first and second attributes are similar (in name and structure) to the first and

```
< article >
< title > From structured Documents to Novel Query Facilities < /title >
< authors >
< author > V. Christophides < /author >
< author > S. Abiteboul < /author >
< author > S. Cluet < /author >
< /authors >
< sections >
< section >
< section1 >
< title > Introduction < /title >
< body >
< parag > Structured documents are central... < /parag >
< /body >
< /section1 >
< /section >
< section >
< section1 >
< title > SGML Preliminaries < /title >
< body >
< parag > In this section, we present... < /parag >
< parag > In order to define... < /parag >
< /body >
< /section1 >
< /section >
< section >
< section2 >
< picture > some bitmap < /picture >
< caption > A DTD for a document < /caption >
< /section2 >
< /section >
< /sections >
< /article >
```

Figure 2: SGML Document

```
class Article public type
        tuple (title      : string,
               authors    : list(Author),
               sections   : set(Section))

class Author : string;

class Section public type
        tuple (section1 : tuple (title : string,
                                 body  : set (string)),
               section2 : tuple(tmuna   : string,
                                koteret : string),
               tag      : string)
```

Figure 3: OODB Schema

second alternatives, resp., of the SGML *section* union type, and the third attribute is a 'tag'. Knowing that ODMG does not support union types, and that such a construct is often implemented by having a tuple containing the two alternatives, plus a tag attribute indicating which of the two alternatives is actually used, we can conclude that two structures of the above form match. The translation function is this case maps an the SGML section to a tuple in the OODB, filling the relevant attribute (according to the section type) and assigning some default value to the other one, and filling the *tag* attribute with the relevant type indication.

- The *authors* element in the DTD and the *authors* list in the OODB both represent collection of matching elements (*author* and *Author* resp.) The translation function can produce an OODB list of authors from the SGML element by taking the individual (translated) author elements and the grouping them into a list ordered by the order of occurrence in the file.

- Last, consider the *picture* and *caption* elements in the SGML DTD. From the above discussion we conclude that the *section2* elements of the two schemas potentially match, hence the *picture* and *caption* are likely to be matched with the *tmuna* and the *koteret* elements in the OODB schema (which are actually the Hebrew terms for *picture* and *caption*, resp.) But assuming that our computer does not contain a Hebrew dictionary, how can it decide which of the two is to be matched with each of the components? Note that we cannot use the structure of the attributes to determine the best match since they both have exactly the same structure. Hence the user is asked to determine the best match.

Now, assume that the system contains, among others, some generic matching rules that cover the above cases: Rule 1, matching ordered and unordered tuple-like structures, with an attached translation function as described above (the rule also handles the case where some attributes are omitted or added); rule 2, matching union types and tagged tuples, with an attached translation function as above; and rule 3, matching collections of matched components, again with a translation function as above. Then, the user input is added to

class name	oid	value
Article	paper	tuple(title : ”From Structured Documents...”, authors : list(VC, SA, SC), sections : set(sec1, sec2, sec3))
Author	VC	V. Christophides
Author	SA	S. Abiteboul
Author	SC	S. Cluet
Section	sec1	tuple(section1 : tuple(title : ”Introduction”, body : set(”Structured documents ...”)), section2 : tuple(tmuna : ””, koteret : ””), tag : ”section1”)
Section	sec2	tuple(section1 : tuple (title : ”SGML Preliminaries”, body : set(”In this section, we present. .”, ”In order to define...”)), section2 : tuple(tmuna : ””, koteret : ””) , tag : ”section1”)
Section	sec3	tuple(section1 : tuple (title : ””, body : set()), section2 : tuple(tmuna : some bitmap, koteret : ”A DTD for a document”), tag : ”section2”)

Figure 4: An OODB Instance

the system as an additional special rule indicating the match between the *picture* and *tmuna* (*caption* and *koteret*) elements, with a translation function which is the identity function (up to elements label). After the matching and translation process (using the extended set of rules) is completed we get an instance of the OODB schema (Figure 4), which is a natural translation of the source document.
This example is relatively simple. Now, let us complicate things a little bit.

- Assume first that the *article* element in the document is defined by

  $<!ELEMENT \quad article \quad (title,\ author^{+},\ section^{*})>$

  and thus the tags *<authors>*, *</authors>*, *<sections>*, and *</sections>* are omitted in the SGML document. In this case, the SGML article no longer includes a clear separation into three components, but is rather a sequence of many elements, starting with a a title element followed by several author elements and then some section elements. Nevertheless, when looking at the schema, it is fairly easy to see that sequence can be logically separated into three parts, and that the $author^+$ subsequence matches the *sections* attribute of the *Article* class, and the the $section^*$ subsequence matches the *authors* attribute. Thus the translation mechanism here first has to split the sequence into its logical sub-components, and only then proceed with the mapping described above. Similarly, if the definition of the *section* element is shortened by

$<!ELEMENT \quad section \quad (\, (title, parag^*) \mid (picture, caption?) \,) >$

then the *section*1, *section*2, and *body* elements no longer explicitly appear in the data – their tags are omitted – which would complicate the mapping to the OO image, if only the data itself was considered. But the logical structure of the data is still reflected in the schema and can thus be used in the translation process to split the file into its logical components.

- As another example, assume that the *Author* class in the OODB, rather than containing a simple string, is defined by

  $$\begin{aligned} class\ Author\ :\ tuple\ (&first\_name\ :\ string, \\ &last\_name\ :\ string, \\ &email\ :\ string) \end{aligned}$$

  Just by looking at the SGML schema (DTD), the system cannot determine how to break an SGML *author* string into the relevant components. The user needs to provide here a specific translation program for this element, based on the string semantics and the data analysis. Although some programming is needed here, the effort is limited to a small portion of the data, while the rest of the translation is derived automatically.

- Finally, assume that the user wants to move all figures to the end of the article, and perhaps also to omit some specific figures. To do that, the user can override Rule 3 above (the rule for matching and translating collections of matched components), so that for this particular collection the translation function reorders the (translated) components as required, and omits the specified elements. Again, some programming is needed to define the new translation function for this specific case, but still this is a very limited: The user only needs to specify the reordering of the collection, while the actual translation of the collection components is given automatically by the system.

There are cases where the differences between the schemas structure require a more complex matching and analysis, e.g. when the source schema includes nested collections or nested tuple structures (which is common in the OO model and in structured documents), and the target schema does not (e.g. when the target system is a relational database), or when the source schema includes references/links (typical for the OODBs and hypertext), and the target schema does not (e.g. a relational database or a simple non-hyperlinked textual document). Nevertheless, our experience shows that even in these cases the common mappings are rather standard. Continuing with the above examples, nested tuples are often represented in flat models by simply flattening the nesting and using a flat tuple containing all the leaf attributes (and sometimes additionally adding the name of the origin component as a prefix to the attribute name). Nested sets have two common representations in flat models: either simple unnesting, or giving each nested set an identifier and then using some auxiliary relation that record the relationship between the identifiers and the corresponding set elements. Similarly, references are often represented in a relational system (or a document) using keys that identify the referred element. Another common alternative in documents is to use an actual copy of the referred element instead of the link. In all these cases it is rather simple to define matching rules for each of the possible alternatives with a corresponding translation function.

Our system contains a large set of predefined rules covering the above cases and many other common cases we encountered in our experiments or found in the literature on data translation. When working with the system the user can add, if needed, additional rules to cover cases that are currently not handled by the system, define arbitrary new translations, or disable/modify/override existing rules to adjust the system to his needs. The system has a graphical interface that can display at each point the two schemas and the set of matches determined by the system rules (and the problems, if any, encountered in the matching process). Starting from this the user can add/disable/modify/override rules to obtain the desired matching and translation.

In the rest of the paper we describe the components of the system and how they are used. We start with the middleware data and schema models, and then continue with the match&translate rules.

3 The Data and Schema models

Handling data and schemas from different models requires a common framework in which the different schema and data formats can be presented. For that we defined a *middleware* schema and data models in which the matching process and the data translation are performed. Each data source that wishes to use the system is expected to provide a mapping of its data and schema to/from the middleware format. As we shall see, the models are fairly simple and the representation of each source inside the middleware is very close to the structure of data/schema in the source, so the implementation of such a mapping is fairly easy. Furthermore our system includes several import/export programs for some common data models (e.g. relational, OO, HTML, SGML, etc.) that can be used by the data sources.

3.1 The Data Model

The data model that we use is similar to the one of [4], and to the OEM and the tree models of [28, 12]. Data is represented by a forest with labeled nodes. A particularity here is that we allow an order to be defined on the children of some of the nodes. Order in an inherent component of some data structures, e.g. ordered tuples and lists. Similarly, textual data can either be described as a sequence of characters or words, or on a higher level by a certain parse tree; in both cases, the order of data element is important. Supporting order as part of the data model enables a natural representation of data coming from such sources [4]. As in [4, 28, 12] the labels on vertices can be used to represent schematic information and data values. To represent cyclic structures, leaves can have values that are the ids of other vertices in the forest, in which case the leaf basically describes a "pointer" to the vertex.

The main reason for the popularity of this kind of model is its simplicity and the fact that one can easily map anything into a graph/tree. To illustrate how data from different

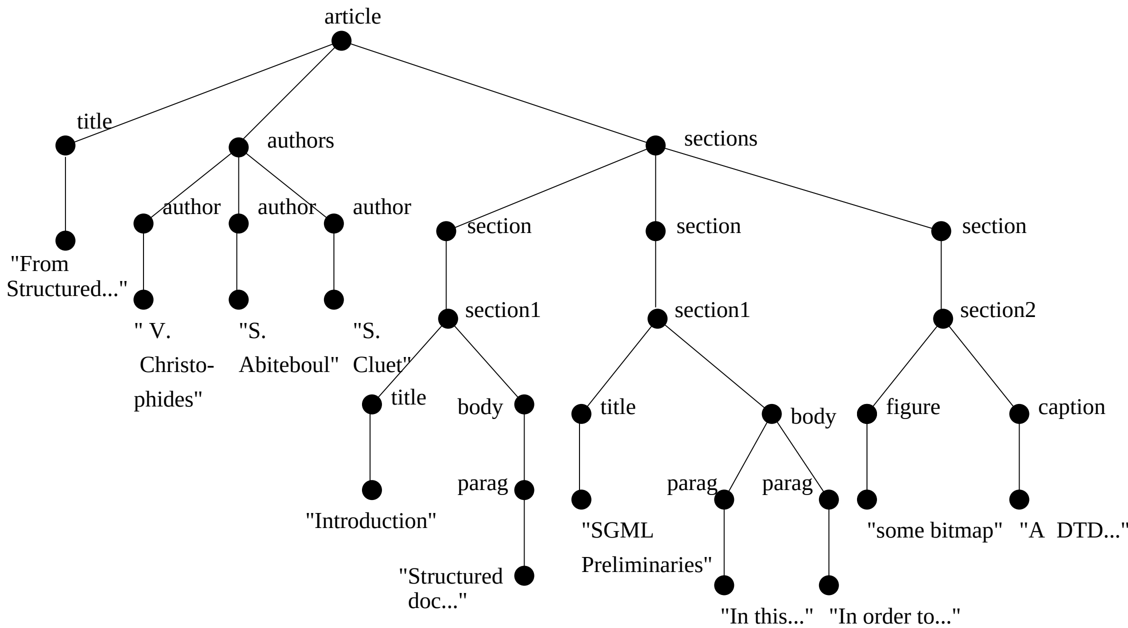

Figure 5: SGML file in the middleware data representation

sources is naturally represented in the middleware model we consider the representation of the SGML document and the OODB discussed in the previous section. (A formal definition of the model and additional examples of the representation of data from various sources in it can be found in [30].) An SGML document is basically represented by its pares tree, so the document in Figure 2 is described by the tree in Figure 5. Its variant, discussed in the previous section, with the *article* and *section* elements defined by

$<!ELEMENT\ \ article\ \ (title,\ author^{+},\ section^{*})>$ and

$<!ELEMENT\ \ section\ \ (\ (title, parag^{*})\ |\ (picture, caption?)\ )>$

is represented by the tree of Figure 6. Observe that the tree here is more flat, reflecting the fact that some of the logical tags are now missing in the data. In the two trees, all the nodes are ordered to reflect the order of elements in the file. The OODB of Figure 4 is described by the forest of Figure 7, where the only ordered node is the *authors*. The $\&i$, $i = 1 \dots 6$ represent the ids of *author* and *section* vertexes pointed by the leafs.

3.2 The Schema Model

Schemas are modeled by labeled graphs, where some of the nodes may be ordered. We chose to use a graph rather than a forest, as in the data model case, to simplify the description of recursive types. This however is not a significant issue and a similar forest-based representation can be defined (by having leaves pointing to other vertices, as done in the data-forest case).

Each vertex in the schema graph represents a schema element (type), and the children represent its possible components. The labeling of a vertex describes the name of the element, some of the element properties, and information on the relationship between the element and its components. This includes information on (1) whether this is a *root* type,

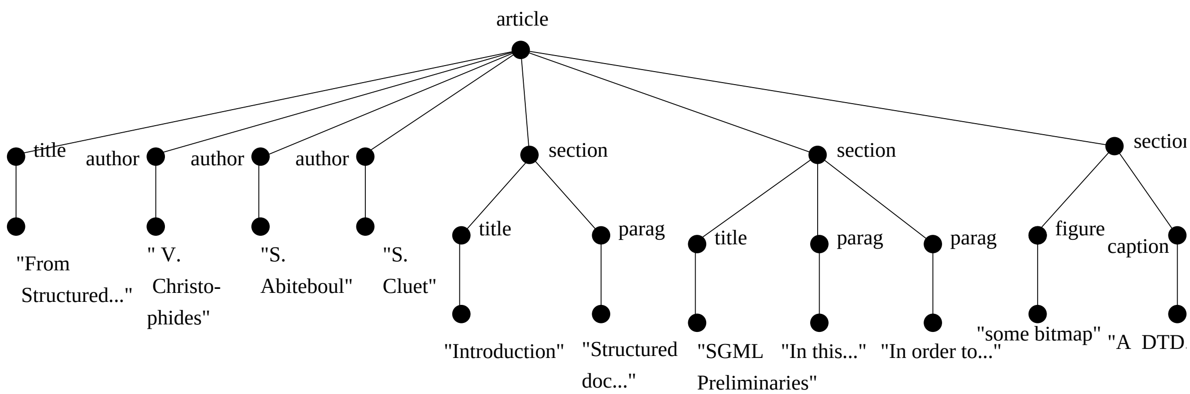

Figure 6: Second SGML file in the middleware data representation

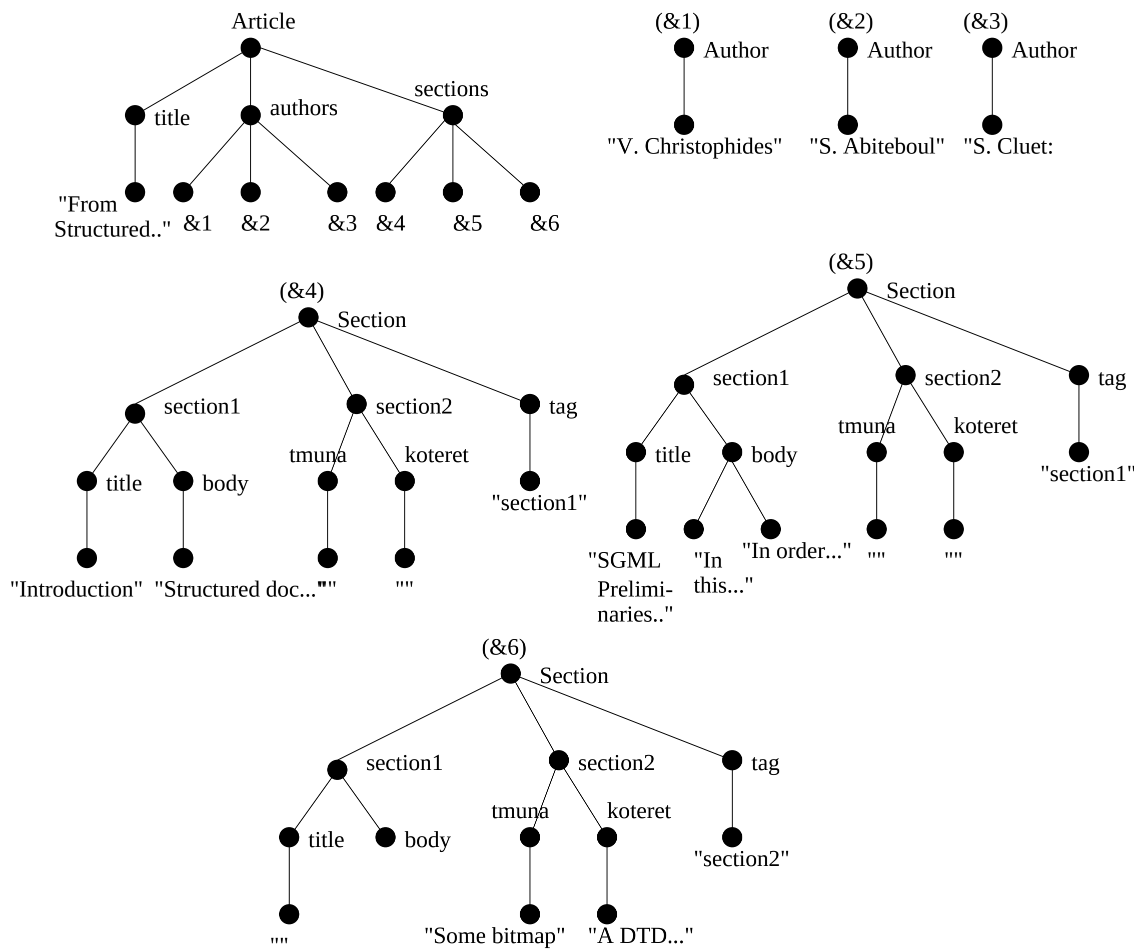

Figure 7: OODB in the middleware data representation

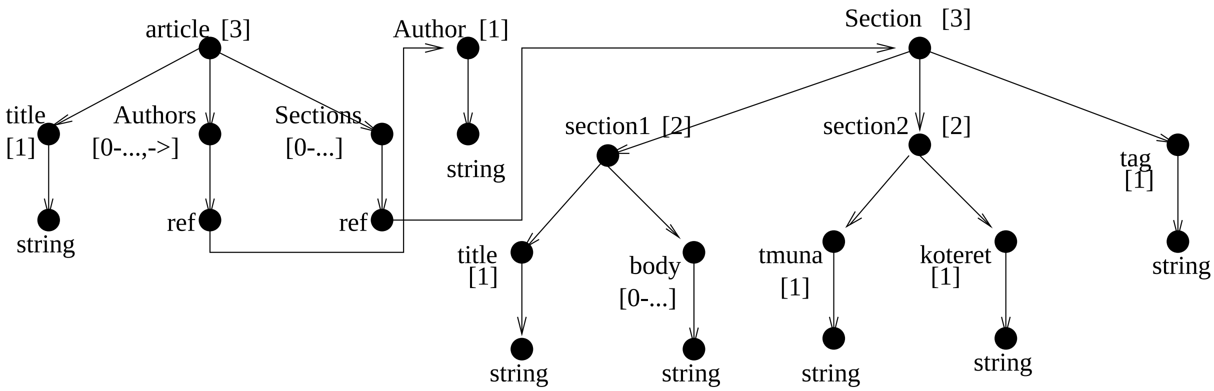

Figure 8: OODB schema in the middleware schema representation

i.e. whether roots of the data forest can be assigned this type, (2) what are the possible labels of data vertices of this type (for leaf vertices this will determine the possible domain of data values), (3) can a data vertex of this type be referenced by other vertices (i.e. the vertex id can be the value of some leaf node in the forest), (4) what is the allowed number (range) of children of a data vertex of this type, (5) are the children of a data vertex of this type ordered or not (6) are some of the component types optional (this is useful for describing union types and optional attributes), (7) is the sub-tree rooted at a node of this type allowed to have arbitrary structure (useful to describe semi-structured data[2]), and (8) whether vertices of this type actually appear in the data graph or are just "virtual".

To understand the last point, consider the second SGML definition of the *article* element $<!ELEMENT\ \ article\ \ (title,\ author*,\ section+)>$ As explained in the previous section, for translation purposes, it is convenient to make it explicit in the schema that an article is composed of three logical components, a *title* part, an $author^*$ part, and a $section^+$ part. Note however that the data tree for this SGML document (Figure 6, does not really contain the $author^*$ and $section^+$ nodes. Item (8) in the labeling is used to reflects this fact.

To illustrate things we present below a few examples. (A formal definition of the schema model and additional examples of the representation of various schemas can be found in [30], for lack of space this is omitted here.) The schema graph of the OODB database from Figures 7 is presented in Figure 8, and the schema graph of the SGML document from Figure 6 is presented in Figure 9.[1]

The empty circles represent "virtual" element, i.e. elements that do not actually appear in the data, while the full circles represent "real" elements. The labeling of a vertex includes the element name (for "real" elements) and some additional information listed in the square brackets. Data instances of an element will have the element name as a label, or, if this a base type name (e.g. Int, String), then the data element will be labeled with a value of the corresponding domain. The keyword *Ref* (Figure 8) is used to denote leaf data vertices that "point" to other vertices, (i.e. have the pointed vertex id as a label). The first element in the square brackets indicates the number (range) of children that a vertex of this type

[1]The schema of the SGML document from Figure 5 is basically the same except that all the "virtual" elements become regular.

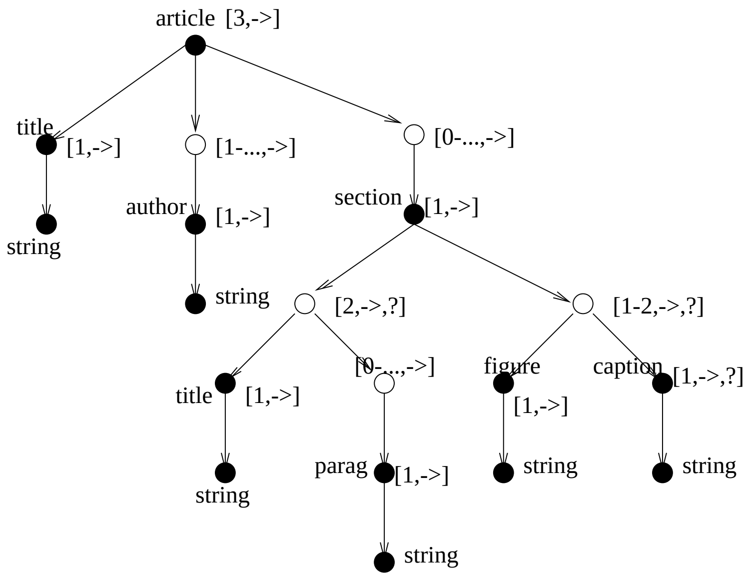

Figure 9: SGML schema in the middleware schema representation

can have. The $\rightarrow$ indicates that the node is ordered. So for example, the $[1 - \ldots, \rightarrow]$ next to the *authors* vertex in Figure 8 means that a data vertex of this type is ordered and can have one or more children. The possible type for the children is determined by the children of the vertex in the schema graph. [2] The ? sign denotes optionality. So for example, the ? next to the *caption* vertex in Figure 9 means that a data vertex of this type is optional, i.e there may be data instances where it appears, and others where it does not. Similarly, the ? next to the two children of the *section* vertex in this graph, (together with the fact that the section vertex is declared to have a single child), reflects the fact that this is a union type, i.e. a choice between two possible types of the children.

A schema graph defines a set of data instances that *conform* to it. Intuitively a data forest F conforms to a schema graph G if each of the vertices $v \in F$ can be assigned a type, i.e. a vertex $t \in G$, s.t. v satisfies the requirements of t, as described by t's labeling. Note however that "virtual" types appear explicitly only in the schema and do not have corresponding vertices in the data. For translation purpose, it is useful to make the full structure explicit in the data as well. So rather than looking at the data forest F we will look at an "explicit" version of it:

Definition 3.1 *An* explicit *version of a data forest F, is a data forest F' with some of its nodes marked as "virtual", s.t. F is obtained from F' by identifying all the virtual vertices with their parent, preserving the order of all the outgoing edges.*

For example, SGML tree in Figure 5, with the *authors*, *section*, *section*1, *section*2, and *body* vertices marked as virtual and their labels omitted, is an explicit version of the tree of Figure 6.

[2] Observe that since the data trees of SGML files are ordered, all the vertices in the schema graph of these files are ordered as well.

Now we can refine the notion of *conformness* described above and say that a data forest F *conforms* of a schema graph G, if F has an explicit version F' and a type assignment h, assigning nodes in G to the vertices of F', s.t. each vertex $v' \in F'$ satisfies the requirements of its assigned type, a described by labeling of $h(v')$ in G, and in particular v' is virtual iff $h(v')$ is. (For formal definition see [30]).

The explicit version F' of a data forest F and its type assignment h are used to determine the data translation, as explained in the next section.

To continue with the above example, the tree of Figure 6 conforms to the schema in Figure 9 due to an explicit version with a structure as in Figure 5, and with the natural type assignment.

4 Match & Translate Rules

Schema matching is the process of matching vertices of the source schema graph with vertices of the target schema graph. The matching achieved is then used for translating instances of the first schema to instances of the second.

For that we use rules. Each rule has of two components, one is in charge on the matching and the other on the translation. The matching part consists of two basic functions: A *Match* function that given two vertices, v_1 in the input schema graph and v_2 in the target schema graph, examines the labeling of the vertices and determines if they "possibly" match. The match is conditional of the matching of the components of the vertices (i.e. their descendents in the schema graph) as determined by the second function, the *Decendents* function. For each pair v_1, v_2 of input and output schema vertices, the function *Decendents* returns two sets of descendents, of v_1 and v_2 resp. (and possibly also a set of constraints) that need to be matched (in a way that satisfies the constraints) in order for v_1 and v_2 to match. The translation part consists of a *Traslation* function that is in charge on the translation of instances of matched types (according to the rule). We use $r.Match$ (resp. $r.Decendents$, $r.Translation$) to denote the *Match* function (resp. *Decendents*, *Translation*) of a rule r. We say that two schema vertices v_1, v_2 match, if there exists some rule r for which $r.Match(v_1, v_2)$ is true.

As a simple example, consider Rule 1 used in Section 2 to match ordered and unordered tuple-like structures. The *match* function of the rule simply compares the names of the two elements (using a built-in dictionary to detect synonyms) and the number of children they can have. The *Decendents* function returns the sets of direct children of the two vertices. The constraint on the allowed matchings for these descendents depends on how close we want the two structures to be: for example, if we want to consider only cases where all the input attributes are represented in the output, we can require the matching on the descendents to be total. If we allow some of the attributes to be omitted, we may allow partial matchings, and possibly constrain the minimal number of (or the specific) attributes that must match. The *translation* function here simply construct a data vertex representing the target tuple (with label as indicated in the target schema), and then attach the translated descendents as children.

We distinguish between three types of rules: *local* rules where the *Decendents* are direct children of the matched schema vertices (as in Rule 1 discussed above); *semi-local* rules where the *Decendens* can be non direct children (e.g. when a nested tuple is mapped to a flat one and the translation takes the leaf attributes of the nested input tuple and glues them together to form a flat tuple)[3]; and *global* rules where the translation function handles the whole subtree rooted at the vertex (i.e. performs a global translation, rather than a recursive one as in the previous cases), in which case the *Decendents* function return the empty set. As we shall see below, global rules are very useful for customizing the translation – The user can add to the system global rules defining special treatment for specific subtrees in the data, while the rest of the data is handled in a standard manner by the other predefined rules of the system.

Rules have priorities, and whenever two vertices can be matched by several rules, we will be interested in the highest priority rule. In the matching process we attach to vertices in the input schema a vertex of the output schema together with a (highest priority) rule that supports the matching of the two vertices.

Definition 4.1 *Given a set of rules R, we say that two schema graphs G_1, G_2* match *w.r.t R, if it is possible to define a partial mapping μ from vertices $v_1 \in G_1$ to pairs (v_2, r) of vertices $v \in G_2$ and rules $r \in R$ s.t. the roots of G_1 are mapped to roots of G_2, and for every vertex $v_1 \in G_1$ with $\mu(v_1) = (v_2, r)$ the following holds.*

1. *r is the best possible matching rule, i.e. $r.Match(v_1, v_2)$ holds and there is no other rule r' with priority $\geq$ of r s.t. $r'.Match(v_1, v_2)$ holds.*
2. *The descendents are properly (and non-ambiguously) matched, i.e.*
   - (a) *The mapping μ when restricted to $r.Decendents(v_1)$ maps the descendents of v_1 to members of $r.Decendents(v_2)$ (satisfying the constraints, if exists, on the allowed matchings), and*
   - (b) *for every $v_1' \in r.Decendents(v_1)$ with $\nu(v_1') = (v_2', r')$ there is no other vertex $v_2'' \in Decenedents(v_2)$ and rule $r'' \in R$ with priority $\geq$ of r', s.t. $r''.Match(v_1', v_2'')$ holds. And conversely, there is no other vertex $v_1'' \in Decendents(v_1)$ and rule $r'' \in R$ with priority $\geq$ of r', s.t. $r''.Match(v_1'', v_2')$ holds.*

If the schema graphs have several roots, then we also require non-ambiguity in the mapping of the roots, ala 2b above.

4.1 User Interaction

There are two cases where the matching may fail: (i) a component of the source schema cannot be matched with a target one using the current set of rules, (and the matching process can neither derive that the component should be just ignored), or (ii) a component of the source schema matches several components in the target schema, and the system cannot automatically determine which is the "best" match.

An example for the first case is when a vertex v_1 can be matched with only one vertex v_2 by a single rule r that requires a total matching on the *Decendents* of v_1 and v_2, but

[3]Note that in this case the descendents of the input vertex are non-direct while those of the output vertex are direct. An example where both descendents are non-direct is when one nested tuple is mapped to another nested tuple having a different internal structure but with matching leaf attributes

some of v_1 descendents cannot be matched with any of v_2 descendents by any of the given rules. An example for the second case is when some descendent of v_1 can be matched by the same rule with two distinct descendents of v_2, and there is no other higher priority rule to break the tie. In fact, this was exactly the case considered in Section 2, when the *figure* (*caption*) element could be match with both the *tmuna* and *koteret* elements, and the system could not determine automatically which is the best match.

Our system has a graphical interface that can display at each point the two schemas and the set of matches determined by the system rules. When the matching fails, the system displays to the user the maximal partial matching satisfying the above conditions, and highlights the schema components where the matching failed. Starting from this the user can add/disable/modify/override rules to obtain the desired matching and translation.

To solve problem (1) the user can add rules to the system to handle the special component and describe the translation to be applied to it. For (ii) the system asks the user to determine the best match. The user input is then added to the system as a new rule with higher priority than that of the rule causing the ambiguity. Now, when the matching process is restarted and reaches the problematic node, it will be matched using the new rule (which is now the highest priority possible rule) with the unique target node specified by the rule, hence resolving the ambiguity. The system maintains the set of rules as a list, and the priority of a rule is reflected by its relative position in the list.

Besides adding new rules, the user can also disable, modify, or override existing rules. Consider for example the SGML-to-OODB translation discussed in section 2, and assume we want to override Rule 3 (the rule for matching and translating collections of matched components) so that some specific collections are given a special treatment (for example, when translating the list of sections we may want to move all figures to the end). To override a rule r we can insert a new rule r' with a higher priority a *Match* criteria that covers a subset of the cases handled by r, and with translation function appropriate for this subset. Since the matching process always chooses the highest relevant priority rule, the new rule will override the old one, for all the specified elements.

Finally observe that, by looking at the matching derived by the system, the user can also see which data components are not involved in the translation: the input schema vertices that are not mapped by the matching to any of the output vertices.

4.2 Translation

Once the systems determines the matching between the source and the target schema graphs (perhaps with the user's assistance), the translation of instances of the first schema to instances of the second is enabled.

To perform the translation, a data instance of the source schema is imported to the common data model, and is "typed", i.e. every data vertex is attached a corresponding schema vertex as its type. Recall however that, to facilitate the translation, we want to use the full logical structure of the data. Hence, we first transform the input into an explicit version, and consider the type assignment for the explicit data forest. Now, the system uses the matching between the input and target schema vertices, computed in the previous step, to translate the data forest by applying recursively from top to bottom the translation functions of the rules attached to the type of the vertices. The resulting forest is an *explicit* instance of the target schema. To obtain a "real" forest, the virtual nodes are glued to their parents (as in Definition 3.1). Finally, the resulting translated data instance is exported to target application.

We conclude this section with two remarks:

Combining Schema-based and Data-based Translation Recent works [4, 19] propose specialized programming languages, targeted for specifying data translations. The schema-based approach that we present here is not aimed to to replace these languages but rather to complement them. The idea is that rather than having to write a translation program for the whole data, much of the translation specification will be done automatically by the system, based on the schema matching, and the programmer will only need to supply some minimal additional code to handle the data components not covered the system.

In terms of our system, this means adding some new rules with a translation function programmed in one of the above languages. In fact, in a previous project we have implemented the data translation language of [4] and the prototype can be extended to enable the execution of translations specified in the language. Typically, such rules are *global* rules where the translation function handles the whole subtree rooted at the vertex, while the rest of the data is handled in a standard manner by the other predefined rules of the system.

Typing The translation process constructs an output data forest. Before exporting the data to the target application, the system checks that the forest indeed conforms to the output schema. Note that this test can be spared if the individual rules are guaranteed to be correct, in the sense that, in each rule, the translation function is guaranteed to generate a legal instance of the output type, if given a legal instance of the input type and a correct translation for the *Decendents*. Our system contains a large set of build-in rules for which correctness, in the above sense, has been verified[30]. When new rules are added, (or when existing rules are modified), the user can either declare them to be "correct", in the sense that their correctness has been checked and proved, or else the type checking has to be enabled at run time to test the translated data before it is exported.

5 Architecture and Implementation

In this section we present the architecture of the TranScm system and the current status of the prototype. The TranScm system is composed of five main components:

- a *rule base* consisting of a large set of predefined rules covering all the above cases and many other common cases we encountered in our experiments or found in the literature on data translation. (A full list of the available rules can be found in [30, 31].)

- a *matching* module in charge on the matching of the input and output schemas w.r.t to the current set of rules. The matching algorithm works in a top down fashion starting from the root nodes and going down, following the conditions in Definition 4.1, and taking at most time polynomial in the size of the schemas and the rules.

- a *typing* module that, given a data forest and a schema graph, tests that the data conforms to the schema, constructing an explicit version of the data forest, together with a type assignment for the vertices. It is possible to show that in the worst case the process can take time exponential in the size of the input (the problem is NP complete), but for a large class of schemas, covering most common data models, a polynomial algorithm exists [11], and this is what we use here.

- a graphical user interface that can display the two schemas and the set of matches determined by the system rules (and the problems, if any, encountered in the matching process), and assists the user in adding/disable/modifying/overriding rules to obtain the desired matching and translation. The interface can also display the input/target data forests and the typing computed for their nodes.

- an extendible library of import/export programs for connecting to external sources and importing/exporting data and schema into/from the system.

The TranScm system is a part of a larger project, WWWDAG[10], that aims to develop tools for the utilization of digital libraries available through the Web. Our system is used to translate data found on the Web to the formats expected by the applications that are part of WWWDAG. The TranScm system is written in Java and its first version is currently fully operational. It can be used in an interactive mode or via an API, and includes all the features discussed above as well as some import/export programs (SGML, HTML, O_2 database, and WWWDAG relational data). We are currently working on enhancing the user interface and plan to add additional import/export modules and work on performance and optimization issues.

6 Related Work

We conclude by considering related work. Many works on data translation focus on the translation of specific formats. Some examples are the Latex to HTML translator [22], the HTML to text translator [1], and mappings between structured documents and object oriented databases in [3] and [18]. Some works [4, 19, 17, 6, 29] generalize this approach and consider mappings between various data models. However most of them relay on the data and not on schema. The input data is converted to some middleware model, where it is transformed or integrated with some target models. This is often done using some translation language. The language should be powerful enough to capture a variety of translations, and may be quite complex. [4], for example, uses datalog-style rules for this matter. In [19] the model is more general, and allows the representation of schemas, but still, the translation program should be written manually, and the translation language is intricate.

The closest to our approach is the one presented in [20], and demonstrated by the WOL language of [21]. This work also considers schema-based data translations. However, in their approach, the translation program depends on the specific characteristics of the input schema (e.g. specific labels and typing in the schema), and every two schemas should be manually assigned mappings. In our system, the translation rules are in a sense more generic. The system contains a large set of predefined generic rules that are based only on common properties of schemas in the middleware model, and not on the characteristic of a specific input schema. The user can however add specific rules for customizing the matching/translation. For that one can use, for example, the languages mentioned above.

A related subject is *schema transformation*. Works in this area mainly concentrated on restructuring of source *schemas* into a target ones (and not of the conversion of *data* instances of the schemas). See [9] for a survey of schema merging and translation techniques. Several works, e.g. [13, 6], consider aspects of merging schemas of source databases. Others, e.g. [7, 8, 5] consider translation of schemas from one model to another. A target schema is created by a series of manipulations on the source schema. [7] for example, introduces a meta-schema model, in which many schemas can be presented. Using the schema meta-model and a rule-based method, the source schema is restructured to become a schema in the target model. The output can then be mapped to the external "real world" format. These ideas are implemented in the MDM system, described in [8]. These works and others, e.g. [25, 26], address the problem of information capacity, namely determining whether it is possible to represent instances of the source schema by instances of the target schema, in a unique way, and vice versa. [7] proves that some schema transformations preserve information capacity. Note, however, that in our context, the user may sometimes need to export data to a specific target schema that does not preserve information capacity.

Most of the works in this area do not consider the underlying data. After a schema is transformed, there is still a need to translated the underlying data. Our work on the other hand concentrates on data translation. It does not deal with the schema transformation, but rather assumes that both the source and target schemas are given as input, and suggests an (partly) automated data translation, based on matching between the schemas. Combining our system with the works that deal with schema transformation could be very beneficial, and we plan to study the issue in future work.

References

[1] The html2text translator, 1996. Available by from `http://www.scanline.com/html2txt/`.

[2] S. Abiteboul. Querying semi-structured data. In *Proc. Int. Conf. on Database Theory ICDT 97*, 1997.

[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, S. Cluet, and T. Milo. Correspondence and translation for heterogeneous data. In *Proc. Int. Conf. on Database Theory (ICDT)*, 1997.

[5] R. Abu-Hamdeh, J.R. Cordy, and T.P. Martin. Schema translation using structural transformation. In *CASCON'94, IBM Centre for Advanced Studies 1994 Conference*, pages 202–215, November 1994.

[6] R. Ahmed, P. De Smedt, W. Du, W. Kent, M.A. Ketabchi, W. Litwin, A. Rafii, and M.C. Shan. The pegasus heterogeneous multidatabase system. *IEEE Computer*, 24(12):19–27, 1991.

[7] P. Atzeni and R. Torlone. Schema translation between heterogeneous data models in a lattice framework. In *Sixth IFIP TC-2 Working Conference on Data Semantics (DS-6)*, Atlanta, Georgia, 1995.

[8] P. Atzeni and R. Torlone. Mdm : a multiple-data-model tool for the management of heterogenous database schemes. In *Proceedings of the ACM Sigmod International Conference, Exhibition Section*, 1997.

[9] C. Batini, M. Lenzerini, and S.B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surverys*, 18(4):323–364, Dec. 1986.

[10] C. Beeri, G. Elber, T. Milo, Y. Sagiv, N. Tishby, D. Konopniki, and P. Mogilevski. Websuite – a tool suite for harnessing web data. In *To appear in WebDB'98*, 1998.

[11] C. Beeri and T. Milo. Schemas for semi-structured data. Technical report.

[12] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, San Diego, 1996.

[13] P. Buneman, S. Davidson, and Anthony Kosky. Theoretical aspects of schema merging. In *Proc. Extending Database Technology*, 1992.

[14] P. Buneman, S. Davidson, Anthony Kosky, and M. VanInwegen. A basis for interactive schema merging. In *Proc. Hawaii International Conference on System Sciences*, 1992.

[15] M.J. Carey et al. Towards heterogeneous multimedia information systems : The Garlic approach. Technical Report RJ 9911, IBM Almaden Research Center, 1994.

[16] T.-P. Chang and R. Hull. Using witness generators to support bi-directional update between objact-based databases. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, San Jose, California, May 1995.

[17] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The tsimmis project: Integration of heterogeneous information sources. In *Proceedings of IPSJ Conference*, pages 7–18, Tokyo, Japan, October 1994.

[18] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Minneapolis, Minnesota, 1994.

[19] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion! In *SIGMOD'98, to appear*, 1998.

[20] Susan Davidson, Peter Buneman, and Anthony Kosky. Semantics of database transformations. Technical Report MS-CIS-95-25, University of Pennsylvania, 1995.

[21] Susan B. Davidson and Anthony S. Kosky. Wol : A language for database transformations and constrains. In *Proc. of the 13th International Conference on Data Engineering*, pages 55–65, April 1997.

[22] Nikos Drakos. The latex2html translator, 1996. Available from `cbl.leeds.ac.uk/nikos/tex2html/doc/latex2html.ps`.

[23] C.F. Goldfarb. *The SGML Handbook*. Calendon Press, Oxford, 1990.

[24] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, 1996.

[25] R.J. Miller, S. Y.E. Ioannidis, and R. Ramakrishnan. The use of information capacity in schema integration and translation. In *VLDB*, 1993.

[26] R.J. Miller, S. Y.E. Ioannidis, and R. Ramakrishnan. Schema equivalence in heterogeneous systems: Bridging theory and practice. *Information Systems*, 19:3–31, 1994.

[27] 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`.

[28] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Int. Conference on Data Engineering*, 1995.

[29] G. Wiederhold. Forward : Intelligent integration of information. *Journal of Intelligent Information Systems*, 6(2/3):281–291, May 1996.

[30] Sagit Zohar. Schema-based data translation, 1997. M.Sc Thesis, Tel-Aviv University.

[31] Sagit Zohar. The transcm system, 1997. Available from `http://www.math.tau.ac.il/~sagit/tranScm/`.

Appendix – SGML

We give below a brief description of the main principles of the SGML standard. *Standard General Mark-up Language (SGML)* is a language for building structured documents. To define a document logical structure, SGML adds descriptive tags in document instances.

An SGML document contains two parts : (i) a prologue including a Document Type Definition (DTD), which is a set of grammar rules specifying the document structure, and (ii) a document instance, which contains the information of the document, structured according to the DTD. To highlight the logical structure of the document, SGML uses descriptive tags. So the document instance contains both the document text and the tags reflecting its structure. The non-terminals in the grammar are called *elements*. Each element is composed of other elements, terminals and connectors. The connectors (constructors) that can be used include, for example, the aggregation constructor (",") which implies order between elements, the choice connector ("|") which provides an alternative in the type definition (union type), the optional indicator ("?"), the plus sign ("+"), which indicates one or more occurrences of an elements, or the asterisk ("*") which indicates zero or more occurrences of an element. The model also allows references.

To illustrate things, the following definition declares that a *section* element can either contain a *title* element followed by an arbitrary number of *parag* elements, or it can contain a *picture* element, possibly followed by a *caption* element.

$$<!ELEMENT \quad section \quad (\ (title, parag^*) \mid (picture, caption?)\ )>$$

. Note that common *Hyper Text Mark-up Language (HTML)* format is a specialization of SGML: All HTML documents have the same DTD part, which means that they follow the same grammar. A full definition of the SGML standard is given in [23].