Schema-Based Data Translation

Extended Abstract

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

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

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 and development, (see for instance, for a very small sample, [16, 10, 14, 13, 26, 22, 15, 4, 3]). 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 [20] or the HTML 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 [3, 17] 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[25].

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 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 handle 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.

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 [3, 17], 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 [3] and to the OEM and the tree models of [27, 10]. 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. 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[30].

In the following sections we provide a general overview of the system (Section 2), and briefly present the middleware data and schema models and the match&translate rules (Section 3). We conclude in Section 4 by considering related work. All the figures appear at the end of the paper.

For lack of space the presentation is rather informal and is mainly based on examples and intuitive explanations. The full definitions of the various components and a detailed description of the system architecture and implementation can be found in [30, 31].

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 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[21] and OODBs, and consider the translation of data between these formats. The example we use is a simplified version of the example described in [16]. (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 components of the schemas, 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 names, 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 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 out 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 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. There are cases where the differences between the schemas structure require a more complex matching. 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 translations, or disable/modify 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/change/modify rules to obtain the desired matching and translation.

3 The System Components

In the rest of the paper we briefly describe the components of the system. We start with the middleware data and schema models, and continue with the match&translate rules. For lack of space the presentation is informal. The full definitions and examples on the representation of data from common data sources (e.g. relational, OO, HTML, SGML, etc.) in these models, and on the definition of various rules, appear in [30].

The Data Model The data model we use is similar to the one introduced in [3], and to the OEM model and the tree models of [27, 10]. Data is represented by a forest with labeled nodes. A particularity 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 [3]. As in [3, 27, 10] the labels on vertices can be used to represent data values and schematic information. 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 Schema Model Schemas are modeled by labeled graphs, where some of the nodes may be ordered[1]. Each of the vertices 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, 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 from this type, (5) are the children of a data vertex of this type ordered or not (this, together with the ordering, if exists, on the components of the type in schema graph, determines the possible types for the children) (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 arbitrary 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 an SGML element definition of the form

$$<!ELEMENT\ \ article\ \ (title, author*, section+)>$$

For translation purposes, it is sometimes convenient to make it explicit in the schema that an article is composed of three components, a *title* part , an *author** part (that is composed of a sequence of *author* elements), and a *section*$^+$ part (that is composed of non empty sequence of *section* elements). Note however that a typical data tree (parse tree) of an *SGML* document of this type, does not really contain the *author** and *section*$^+$ nodes. In the parse tree, the *article* root vertex typically has one *title* child followed by sequences of *author* elements followed by a sequence of *section* elements, and the authors and sections are not explicitly grouped. This is different, for example, from OO data where the common forest representations explicitly include *all* the type constructors. So an article object in an OODB will be represented by a vertex having three children, one for the title, one for the list of authors

[1] 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).

and one for the list of sections, (and the last two vertices will in turn have children representing the sequences of authors and sections resp.) Item (8) in the labeling is used to reflects this difference between the representation of data in these models.

In the full version of the paper we define what does it mean for a data forest to *conform* to a schema graph, and the *domain* a schema graph is then defined to be the set of all data forests 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.

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 ($Decendents$, $Translation$) of a rule r.

We distinguish between three types of rules: *local* rules where the $Decendents$ are direct children of the matched schema vertices (e.g. when we map one tuple type to another and the translation process translates each of the direct attributes and then glues them together to form an output tuple), *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)[2], 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. Observe that 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.

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.

[2]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

Once the systems determines the matching between the source and the target schema graphs (perhaps with the user's assistance to resolve ambiguities or deal with components that cannot be matched suing the current set of rules), 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. 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. Finally, the resulting translated data instance is exported to target application.

4 Related Work

Many works on data translation focus on the translation of specific formats. Some examples are the Latex to HTML translator [20], the HTML to text translator [1], and mappings between structured documents and object oriented databases in [4] and [16]. Some works [3, 17, 15, 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. [3, 25], for example, uses datalog-style rules for this matter. In [17] 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 [18], and demonstrated by the WOL language of [19]. 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. [12, 11, 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. [23, 24], 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 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.

The work presented in this paper is a part of a larger project, WWWDAG[28], 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.

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. Correspondence and translation for heterogeneous data. In *Proc. of*

the ACM SIGMOD Conf. on Management of Data, San Jose, California, 1995.

[4] 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.

[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, December 1986.

[10] 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.

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

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

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

[14] 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.

[15] 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.

[16] 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.

[17] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion!, 1997. Submitted for publication.

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

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

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

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

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

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

[24] 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.

[25] Pini Mogilevsky. W3trans translation system, 1996. Available from `www.math.tau.ac.il/pinim/w3trans-home.html`.

[26] 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`.

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

[28] O. Shmuely, G. Elber, T. Milo, C. Beeri, Y. Sagiv, and N. Tishby. Intelligent utilization of heterogeneous information sources and digital libraries.

[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/`.

```
<!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

```
<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

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