An Approach to Heterogeneous Data Translation based on XML Conversion

Paolo Papotti and Riccardo Torlone

Dipartimento di Informatica e Automazione
Università Roma Tre
{papotti,torlone}@dia.uniroma3.it

Abstract. In this paper, we illustrate a preliminary approach to the translation of Web data between heterogeneous formats. This work fits into a larger project whose aim is the development of a tool for the management of data described according to a large variety of formats used on the Web and the (semi)automatic translation of schemes and instances from one model to another. Data translations operate over XML representations of instances and rely on a uniform representation of models that we call metamodel. The metamodel shows structural diversities and dictates the needed transformations. Complex translation can be derived by combining a number of predefined basic functions performing XML transformations expressed in XQuery. Practical examples are provided to show the effectiveness of the approach.

1 Introduction

Very often, data cooperation and interchange between different organizations is made difficult by the fact that little or no advance standardization exists and data is stored under different formats in distinct heterogeneous sources [1]. Therefore the need arises for an integrated management of heterogeneous data descriptions that allows for easy and flexible data *translation* from a format to another [6]. This problem is related to, but different from, the problems of data *integration* [4] and schema matching [20]. Recently, various aspects of the data translation problem has been largely studied in the context of the relational model [9, 10] or in more general settings [16, 18, 19]. However, it is widely recognized that a general solution able to cope the large diversity of the various formats available is a very difficult task [5].

In this framework, the final goal of our research project is the development of a tool for the management of data available on the Web described according to a large variety of formats and models and the (semi)automatic translation of schemes and instances from one model to another. The tool can be seen as an implementation of the "ModelGen" operator proposed by Bernstein in the context of *Model Management Systems* [5].

In principle, the set of models managed by the tool should include the majority of the formats used to represent data in Web-based applications: semi-structured models, schema languages for XML, specific formats for e.g. scientific

data, and even traditional conceptual data models. Actually, the set of models is not fixed a priori in the environment we have in mind. A new model M should be definable by the user at run-time and translations for M should be derived by the system with limited user intervention.

Recently, we have proposed a tool for the management and the automatic translation of schemes between the majority of formats and models used to represent data in Web-based applications [21, 22].

The approach relies on a novel notion of *metamodel*, expressed in XML, that embeds, on the one hand, the main primitives adopted by different schema languages for XML [15] and, on the other hand, the basic constructs of traditional database conceptual and logical models [13].

This metamodel provide a uniform representation of models that allows the identification of differences between primitives used in the various models. Then, translations are automatically derived by combining a set of predefined and standard translations between individual primitives.

In this paper, we present a preliminary approach that, building on the translation derived at scheme-level, aims at generating a corresponding translation at instance-level. This translation operates over serialized XML representations of data. XML data is then transformed to agree with the constructs allowed in the target model. Finally, it is deserialized into the specific syntax of the target. The transformation phase is performed by combining a number of predefined basic functions expressed in XQuery [11]. A number of practical examples are presented to show the effectiveness of the approach.

The rest of the paper is organized as follow. In Section 2 we provide a general overview of our approach to model management. In Section 3 we present a new technique for data translation and in Section 4 we illustrate a complete example of translation. Finally, in Section 5 we discuss some open issues and sketch future direction of research.

2 A metamodel approach to Model Management

Let us first clarify our terminology. In our framework, we identify four levels of abstractions. At the bottom level we have actual *data* (or *instances*) organized according to a variety of (semi) structured formats (relational tables, XML, HTML, scientific data format and so on). At the second level we have *schemes*, which describe the structure of the instances (a relational schema, a DTD, an XML schema or one of its dialects [15], etc.). Then, we have different formalisms for the description of schemes, which we call *models* hereinafter (e.g., the relational model, the XML schema model or even a conceptual model like the ER model). Finally, we use the term *metamodel* to mean a general formalism for the definition of the various models.

In this framework, a *translation* is defined as follows: given two models M_1 and M_2 represented by the metamodel, a set of data D_1 (the *data source*) of a scheme S_1 (the *source scheme*) for M_1 (the *source model*), a *translation* of D_1

(S_1) into M_2 is a set of data D_2 (the *data target*) of a scheme S_2 (the *target scheme*) for M_2 (the *target model*) containing the same information as D_1.[1]

Our approach relies on a *metamodel* notion made of a set of *metaprimitives*. Each metaprimitive captures one basic abstraction principle used in some data model [21]. Examples of metaprimitives are: class, attribute, base type, relationship, sequence, generalization, disjoint union, key, foreign key, and so on. In this framework, a model is defined as a set of *primitives*, each of which is classified according to a metaprimitive of the metamodel. For instance the relational model offers the *table* primitive which is an instance of the metaprimitive *relationship* over basic domains.

In [21, 22] we have proposed a technique and a tool for the management of XML based data model (that is, data models expressed in XML) and the translation of *schemes* from one model to another. The scheme translation technique makes use of an internal concept, called *supermodel*, which is used by the system as a reference for the translations. Intuitively, a supermodel is a model (that is, like the other models, an instance of the metamodel) maintained automatically by the system that "subsumes" each other model [21]. The translation process of a scheme can be then seen as composed of a number of steps. First, the scheme is expressed in an internal representation. We are using XML since it is a widely accepted standard for data exchange and allows the description of information at different levels of abstraction. Second, the scheme is translated into the supermodel. This is actually a trivial task since, by definition, every scheme of any model is a scheme of the supermodel. Then, the scheme is transformed by translating primitives used in the source scheme that are not allowed in the target model. This is clearly the more involved step. Therefore, the scheme we have obtained is converted into a format compatible with the target model, but still in the internal representation, and finally translated into the specific syntax of the target model. Again, this last phase is rather trivial.

Note that, with this approach, it suffices to define translations from the supermodel to every other model in order to implement all the possible translations between models. It follows that the number of required translations is linear in terms of the number of models, instead of quadratic, as it would be if the process had to be specified for each pair of models. As the number of primitives is limited, it is possible to predefine a number of basic translations, which can be composed to build more complex translations.

As a first concrete example of scheme translation, let us consider the `Order` XML Schema reported on the left hand side of Figure 1 (clearly, it does not require an XML conversion) and assume that we need to convert this scheme into a DTD. Assume that the metamodel of reference contains the following metaprimitives: element, attribute, ordered sequence, unordered sequence, choice, base types, cardinality, inheritance, key constraint, and foreign key constraint. Then, the corresponding scheme in the supermodel is reported on the right hand side of

[1] We stress the fact that we are interested into the translation of a data source into a different representation rather than the derivation of a mapping between heterogenous data sources.

the same figure . In this step the various primitives have been converted into the corresponding metaprimitives. For instance, the primitive *all* of XML Schema has been turned into the metaprimitive *unordered sequence*.

Fig. 1. An XML Schema and its representation in the supermodel

The left hand side of Figure 2 shows the scheme of the supermodel produced by the tool as the translation of the scheme of Figure 1 into the DTD model. The final target scheme is reported on the right hand side of the same figure. Note that, for instance, the unordered sequence used to define the structure of the element *destination* of the source has been transformed into an ordered sequence, since unordered sequences are not representable by a DTD.

Fig. 2. The translation of the scheme in Figure 1

The transformation of the cardinality from 0:10 to 0:N is a clear example of a "semantic loss" due to the limited expressiveness of the target model. In this case, the system stores information about the loss externally, in a file associated with the target scheme. We call this extra information the *residual* of the scheme. With this solution, it is possible to reverse the translation using the residual.

In the rest of the paper, we illustrate an approach to *data* translation that, building on the above translation scheme, aims to generate a corresponding translation at instance-level.

3 Data translation

According to the schema translation process, data translation requires a number of phases. First, data is automatically serialized into XML preserving the original structure (a simple example is given in Figure 3). Then, XML data is transformed into a structure that matches with the target scheme (expressed in XML format) produced by the scheme translation process. Finally, data is transformed into the final format according to the specific syntax of the target model. The first and the last phases are rather easy, usually supported by systems, and not always needed when source and/or target are already represented in XML. Therefore, they will not discussed further. We concentrate now our attention on the transformation phase where data, expressed in XML, is restructured according to the target model.

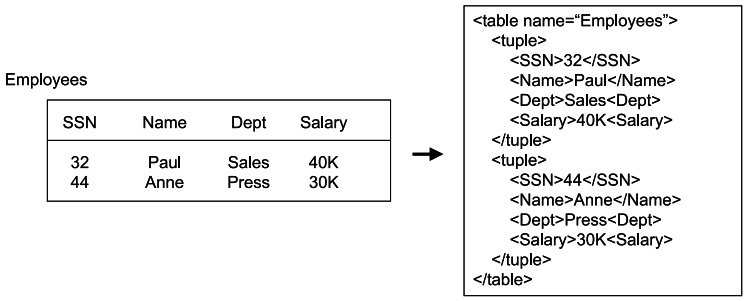

Fig. 3. An example of serialization

3.1 An approach to model translation

The transformation method proceeds by analyzing the scheme S_s of the input data in the supermodel (see above). For each primitive C used in S_s, the system verifies whether the corresponding metaprimitive $\mathcal{C}$ is allowed in the target model. If this is not the case, it tries to convert instances of $\mathcal{C}$ into a format of another primitive (or a set thereof) available in the target model.

This work is supported by a set of predefined basic procedures p that implements rather standard translations between constructs. Each of these procedures has indeed two components: a schema-level function f_S, which performs translations of metaprimitives, and a function f_I, which operates at instance level by transforming actual data according to the translations operated by f_S. Specifically, these functions must satisfy the following consistency criterium: given a procedure $p[f_S, f_I]$, for each scheme S and each instance I of S, it is the case that $f_I(I)$ is an instance of $f_S(S)$. Both f_S and f_I generate residual information (that is, components that have been lost in the translation), as explained above. Representatives of such procedures will be presented in more detail in Section 3.2.

The technique is specified in the algorithm reported in Figure 4. This algorithm generates the target scheme and a transaction t, made of a sequence of functions f_I, that translates any instance of the source scheme into a valid instance for the target scheme.

Algorithm 1
Input: *A scheme S_s of a model M_s, the residual m_s of S_s (if available), a library of procedures $L = \{p_1[f_S^1, f_I^1], \ldots, p_k[f_S^k, f_I^k]\}$, and the target model M_t*
Output: *A transaction t, a scheme S_t for M_t, and the residual m_t of S_t*
begin
(1) Set a temporary scheme S to the source scheme S_s;
(2) Set t to the empty transaction;
(3) ***while*** *there is a primitive C in S such that the corresponding metaprimitive $\mathcal{C}$ is not allowed in M_t* ***do***
(4) ***if*** *there exists a procedure $p_i[f_S^i, f_I^i]$ in L such that f_S^i translates $\mathcal{C}$ to a metaprimitive (or a set thereof) allowed in M_t*
(5) ***then*** ** direct translation **
(6) $S = f_S^i(S)$; ** apply f_S^i to S **
(7) *add to m_t the residual generated by f_S^i;*
(8) $t = t, f_I^i$; ** append f_I^i to t **
(9) ***else***
(10) ***if*** *there exists a procedure $p_i[f_S^i, f_I^i]$ in L such that f_S^i translates $\mathcal{C}$ to a metaprimitive (or a set thereof) <u>not</u> allowed in M_t* ***and*** *the analysis of m_t prevents infinite loops;*
(11) ***then*** ** try to find an intermediate translation **
(12) $S = f_S^i(S)$; ** apply f_S^i to S **
(13) *add to m_t the residual generated by f_S^i;*
(14) $t = t, f_I^i$; ** append f_I^i to t **
(15) ***else***
(16) *abort the translation and notify the user;*
end while
(17) $S_t = S$; ** S becomes the target scheme **
end

Fig. 4. The translation algorithm

There are a number of important aspects to point out about this algorithm. First, in step (4) it may happen that more than one procedure available in L can perform the needed translation. For instance, it is well known that there are several ways to translate generalizations into other primitives. A possible solution in this case is the introduction of a request for user intervention in order to make a choice between the various possibilities. Also, ambiguity can be solved by introducing a (partial) preference order between procedures.

Another point is in step (10): a scheme translation function f_S could translate a metaprimitive $\mathcal{C}$ into a metaprimitive that is not allowed in M_t. The rationale here is that if we are not able to translate directly into M_t, we try to translate $\mathcal{C}$ into an intermediate metaprimitive that is not allowed in the target model but for which there could exist a translation towards the target. Consider for instance the translation from the Entity-Relationship model into a DTD representation. In this case, generalizations can be first translated into relationships (which are not directly representable in a DTD). Then, relationships can be easily translated into elements and attributes of a DTD. It is easy to see however that, proceeding in this way, we can enter into infinite loops. In order to prevent this situation, the method verifies whether the selected procedure introduces a metaconstruct that has been previously deleted. This can be done by analyzing the residual generated until that point.

It's important to note that a procedure translating a metaprimitive does not always requires a data translation. Assume, for example, that we need to translate a scheme S with a *cardinality* constraint of type $(1, 10)$ to a model that allows only cardinalities of the form $(1, 1)$, $(1, n)$ and $(0, n)$. In this case S needs to be modified but this change does not affect data. On the other hand, many metaprimitives need data manipulation, like the creation of identifiers.

As a final comment, we note that this main algorithm can be improved in several points. In particular, a final optimization step can be introduced on the output transaction by eliminating redundant or useless functions and by finding a better execution order. This is subject of current work.

3.2 Basic procedures

In this section we illustrate some examples of basic procedures used by the Translation Algorithm reported in Figure 4. They are used within the supermodel, where the system matches models definition and selects metaprimitives to be transformed, as described in Algorithm 1. We recall that each procedure is composed by two functions, one operates at scheme level and the other at instance level.

1. **Nesting of complex elements**. This procedure nests elements according to referential integrity constraints between them.
   f_S: it nests an element definition E_1 into another element definition E_2, deletes the corresponding integrity constraint, and stores the performed translation in the residual.

f_I: it groups and nests instances of E_1 into the corresponding instance of E_2 and deletes the reference between them.

2. **Unnesting of complex elements**. The procedure flats nested elements and introduces integrity constraints between them.
   f_S: it unnests a complex element E_1 nested into another element E_2 by moving the definition of E_1 at the same level of E_2 and introducing a foreign key between them. It also stores the performed translation in the residual.
   f_I: it moves instances of E_1 outside the instance of E_2.
3. **Key creation**. It generates identifiers for elements making use of Skolem functors [14] if the element contains at least an atomic element or an attribute, otherwise making use of counters.
   f_S: it adds a key constraint K to an element E and stores the performed translation in the residual.
   f_I: it invents a value for each instance of E using either a Skolem functor or a counter, and assigns it to the instance as unique identifier.
4. **Add/Remove namespaces**. This pair of procedures add/remove information on the domain of the names used in a scheme.
   f_S: it adds/deletes the namespace definition, retriving/storing this information from/in the residual.
   f_I: it does nothing on instances.
5. **Cardinality range extension**. As we have said in Section 2, cardinalities are used at different levels of precision in the various models. This procedure changes the actual value of a cardinality to an undefined value and has no effect on instances.
   f_S: it changes the cardinality definitions from a number, different from 0 or 1, to the undefined value N and stores information on the old values in the residual.
   f_I: it does nothing on instances.
6. **Cardinality range restriction**. Differently from the previous procedure, this procedure implies some involved transformation on the instances. As an example, consider the transformations needed to convert an n-ary relationship to a binary one.
   f_S: it changes values that express cardinality in the element definition and stores information on the deleted values in the residual.
   f_I: it applies transformation on the instances of the elements with the modified cardinality, grouping and splitting element instances according to the new values.
7. **Transformation of ordered sequences in unordered ones**. The procedure performs the translation adding a new attribute that codes the order.
   f_S: it changes the ordered sequence definition to unordered, introduces a new attribute and stores the performed translation in the residual.
   f_I: it adds an atomic element that takes an integer coding the original position on the element in the ordered sequence.
8. **Transformation of generalization hierarchies**. The procedure removes generalizations and translates them in other primitives.
   f_S: there are several ways to translate generalizations (e.g., using relation-

ships or grouping elements) and the user can choose the preferred one. The procedure stores information on the performed translation and on the removed elements in the residual.
f_I: it performs modification on the instances according to the choice done at scheme level.

9. **Add generalization**. This procedure adds the definition of a generalization making use of residual information of the scheme (if any).
f_S: it adds a generalization between elements making use of information stored in the residual.
f_I: it adds a generalization instance for each set of element instances that share the same identifier.

```
Scheme unnnesting
Input: A scheme Ss with nested elements and its residual
Output: A scheme St without nested elements and its residual
begin
   set S to the empty scheme;
   for each element ex in Ss do
      if ex is a complex element
         then
            case ex of
               is nested in a complex element ec:
                  copy ex in S outside ec;
                  add to ex a foreign key kr for ec;
               is not nested:
                  copy ex in S;
            end case
         else  *\ ex is atomic *\
            case ex of
               is nested in a complex element ec:
                  copy ex in S inside ec;
               is not nested:
                  copy ex in S;
            end case
   end for
   eliminate intermediate elements;
   St = S; \* S becomes the target scheme *\
end
```

```
Data unnnesting
Input: An instance Is with nested data, the scheme Ss of Is
Output: An instance It without nested data
begin
   set I to the empty instance;
   for each element ex in Is do
      if ex is a complex element
         then
            case ex of
               is nested in a complex element ec:
                  copy each occurrence of ex in I outside ec;
                  add to each occurrence of ex the key of ec;
               is not nested and contains an atomic element ea:
                  copy each occurrence of ex in I;
            end case
         else  *\ ex is atomic *\
            copy in I each occurrence of ea;
   end for
   eliminate intermediate elements;
   It = I; \* I becomes the target instance *\
end
```

Fig. 5. An example of basic procedure

As a concrete example, we now present in more detail the unnesting procedure. Unnesting is a rather common issue in data conversion: how to flat a nested scheme arises when, for instance, we need to store XML data into a relational database. This problem has been largely debated in the literature [12]. Here, we just show intuitive algorithms, based on combination of elementary operations over XML data. All the complex elements must contain an identification key, or have to preliminary be processed by the key creation procedure. With this approach the unnesting translation is completely reversible: system just needs to apply the nesting procedure to returns the original scheme and data. The first function, in the left hand side of Figure 5, takes as input a scheme S_s and outputs a scheme S_t, where nested elements are converted into flat ones. The second, in right hand side of the same figure, works on data: takes as input an instance I_s of the scheme S_s and outputs an instance I_t of the scheme S_t.

4 An example of translation

In this section we present a complete translation from the XML Schema model to the relational model. The input instance and the corresponding scheme are reported in Figure 6.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name = "Dept" >
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="DeptName" type="xsd:string"/>
        <xsd:element name="CreationDate" type="xsd:date"/>
        <xsd:element name = "Emps" >
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name = "Emp"
                      maxOccurs="unbounded">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="EmpID"
                            type="xsd:integer"/>
                    <xsd:element name="EmpName"
                            type="xsd:string"/>
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

```
<Dept>
  <DeptName>Storage</DeptName>
  <CreationDate>1999-01-07</CreationDate>
  <Emps>
    <Emp>
      <EmpID>37</EmpID>
      <EmpName>Paul</EmpName>
    </Emp>
    <Emp>
      <EmpID>48</EmpID>
      <EmpName>Andrew</EmpName>
    </Emp>
  </Emps>
</Dept>
```

Fig. 6. An XML schema and one of its instances

The source scheme is transformed in the supermodel scheme reported in the left hand side of Figure 7. The system applies to this scheme the Translation Algorithm reported in Figure 4.

```
TA source="XSD">
iment name="Dept">
equence occurs="1:1">
element name="DeptName" type="string" occurs="1:1"/>
element name="creationDate" type="date" occurs="1:1"/>
element name="Emps" occurs="1:1">
sequence occurs="1:1">
 element name="Emp" occurs="1:N">
   sequence occurs="1:1">
    element name="EID" type="integer" occurs="1:1"/>
    element name="EName" type="string" occurs="1:1"/>
   </sequence>
 </element
</sequence>
/element>
equence>
ement>
TA>
```

```
<META source="Relational">
 <element name="Dept" occurs="0:N">
   <attribute name="DeptName" occurs="1:1" type="string"/>
   <attribute name="CreationDate" occurs="1:1" type="string"/>
   <attribute name="Dept-New-Key" type="key" occurs="1:1"/>
 </element>

 <element name="Emp" occurs="0:N">
   <attribute name="Dept-Emp-Key" type="string">
     <keyref name="Dept-Emp-Key-Est" refer="Dept-New-Key"/>
   </attribute>
   <attribute name="EmpOrder" occurs="1:1" type="integer"/>
   <attribute name="EmpID" occurs="1:1" type="string"/>
   <attribute name="EmpName" occurs="1:1" type="string"/>
 </element>
</META>
```

Fig. 7. The translation of the scheme in Figure 6 in the Supermodel

The system performs three main transformations:

– the creation of a key for the elements not having it;
– the unnesting of elements using the procedure described in Section 3.2;
– the transformations of ordered sequences into unordered ones (that become tables in the target model);

The result scheme is reported on the right hand side of Figure 7. At the end of the algorithm, the system renames primitives generating the target scheme in relational format reported in Figure 8. Finally, the t translation is applied to the source instance I_s, generating the target data in Figure 8, ready to be serialized into a database.

```
<database>
 <table name="Dept">
  <tuple>
   <field name="DeptName" occurs="1:1" type="string"/>
   <field name="CreationDate" occurs="1:1" type="string"/>
   <field name="Dept-New-Key" type="key" occurs="1:1"/>
  </tuple>
 </table>

 <table name="Emp">
  <tuple>
   <field name="Dept-Emp-Key" type="string">
      <keyref name="Dept-Emp-Key-Est" refer="Dept-New-Key"/>
   </field>
   <field name="Emp-New-Key" type="key" occurs="1:1" />
   <field name="EmpOrder" occurs="1:1" type="integer" />
   <field name="EmpID" occurs="1:1" type="string" />
   <field name="EmpName" occurs="1:1" type="string" />
  </tuple>
 </table>
</database>
```

```
<Dept>
 <tuple>
  <DeptName>Storage</DeptName>
  <CreationDate>1999-01-07</CreationDate>
  <Dept-New-Key>sk1(Storage,1999-01-07)</Dept-New-Key>
 </tuple>
</Dept>

<Emp>
 <tuple>
  <Dept-Emp-Key>sk1(Storage,1999-01-07)</Dept-Emp-Key>
  <Emp-New-Key>sk2(37,Paul)</Emp-New-Key>
  <EmpOrder>1</EmpOrder>
  <EmpID>37</EmpID>
  <EmpName>Paul</EmpName>
 </tuple>
 <tuple>
  <Dept-Emp-Key>sk1(Storage,1999-01-07)</Dept-Emp-Key>
  <Emp-New-Key>sk2(48, Andrew)</Emp-New-Key>
  <EmpOrder>2</EmpOrder>
  <EmpID>48</EmpID>
  <EmpName>Andrew</EmpName>
 </tuple>
</Emp>
```

Fig. 8. The final result of the translation of the scheme and the instance in Figure 6

5 Discussion and future work

In this paper, we have presented an approach to the translation of Web data between heterogeneous formats. This translation operates over a XML representation of data and is derived by combining a number of predefined basic functions performing XML transformations.

It should be said that a number of conceptual aspects related to data translations have not been addressed in this paper. In particular, the analysis of *translation quality*. In [2, 3] we have proposed several properties that "good" translations should enjoy. The more relevant are *correctness* and *minimality*. The former establishes that the output is valid in the target model, the latter expresses the fact that does not exist shorter translations. We are currently studying how these properties can be verified in the framework presented here.

From a practical point of view, we are currently extending our tool (whose preliminary version has been presented in [22]) with the proposed approach for

data translation. Different implementations are under development. One of these solutions is a combination of XQuery and DOM. The tool manages schemes translations between models using DOM representations and performs XML data transformations by means of iterative queries expressed in XQuery over materialized temporary results. The tool is completely modular, so we are also implementing basic procedures in XSLT [8]. At the moment, the tool is able to fully translate schemes and data between various formats (XML Schema and some of its dialects [15], DTD, Entity-Relationship, Relational) and we are extending the tool with other models for Web data (Araneus [17] and WebML [7]).

We intend to improve the technique presented in this paper. We are particularly interested in the introduction of an optimization phase to be performed over data translation, which is clearly the most expensive task.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.
2. 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), Atalanta*, pages 218–227, 1995.
3. P. Atzeni and R. Torlone. Management of Multiple Models in an Extensible Database Design Tool. In *Fifth International Conference on Extending Database Technology (EDBT '96), Lecture Notes in Computer Science 1057*, Springer–Verlag, pag. 79–95, 1996.
4. C. Batini, M. Lenzerini, and S. B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. In *ACM Computing Surveys 18(4)*, pages 323–364, 1986.
5. P. A. Bernstein. Applying Model Management to Classical Meta Data Problems. *CIDR 2003* (available at `http://www-db.cs.wisc.edu/cidr/`)
6. P. A. Bernstein, A. Y. Levy, and R. A. Pottinger. A Vision for Management of Complex Models. *SIGMOD Record*, 29(4):55–63, December 2000.
7. S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera. *Designing Data-Intensive Web Applications*. Morgan Kaufmann, 2003.
8. J. Clark. XSL transformations (XSLT) specification. W3C Document, November 1999. http://www.w3.org/
9. R. Fagin, P. G. Kolaitis, and L. Popa. Data exchange: getting to the core. In *22nd ACM Symposium on Principles of Database Systems, San Diego*, pages 90–101, 2003.
10. R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. In *9th Int. Conference on Database Theory, Italy*, pages 207–224, 2003.
11. M. Fernandez, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 Data Model. W3C Document, November 2003. http://www.w3.org/
12. D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3): 27–34, 1999.
13. R.B. Hull and R. King. Semantic database modelling: survey, applications and research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

14. R. Hull and M. Yoshikawa. ILOG: Declarative Creation and Manipulation of Object Identifiers. In *16th International Conf. on Very Large Data Bases, Brisbane*, pages 455–468, 1990.
15. D. Lee and W. W. Chu. Comparative Analysis of Six XML Schema Languages. *SIGMOD Record*, 29(3):76–87, 2000.
16. S. Melnik, E. Rahm, and P. A. Bernstein. Rondo. A Programming Platform for Generic Model Management. In *ACM SIGMOD International Conference on Management of Data, San Diego*, pages 193–204, 2003.
17. P. Merialdo, P.Atzeni, and G. Mecca. Design and development of data-intensive web sites: The araneus approach. *ACM Trans. on Internet Technology*, 3(1): 49–92, 2003.
18. L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernndez, and R. Fagin. Translating Web Data. In *28th International Conf. on Very Large Data Bases, Hong Kong*, pages 598–609, 2002.
19. R. Pottinger and P. A. Bernstein. Merging Models Based on Given Correspondences. In *29th International Conf. on Very Large Data Bases, Berlin*, pages 826–873, 2003.
20. E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.
21. R. Torlone and P. Atzeni. A Unified Framework for Data Translation over the Web. In *Second International Conference on Web Information System Engineering (WISE 2001), Kyoto*, IEEE Computer Society Press, pages 350–358, 2001.
22. R. Torlone and P. Atzeni. Chameleon: an Extensible and Customizable Tool for Web Data Translation. In *29th International Conference on Very Large Data Bases (VLDB 2003), Berlin*, pages 1085–1088, 2003.