A Semantic Approach to Integrating XML and Structured Data Sources

Peter M^cBrien
Dept. of Computing, Imperial College,
180 Queen's Gate, London SW7 2BZ, pjm@doc.ic.ac.uk

Alexandra Poulovassilis
Dept. of Computer Science, Birkbeck College, University of London,
Malet Street, London WC1E 7HX, ap@dcs.bbk.ac.uk

Abstract

XML is fast becoming the standard for information exchange on the Internet. As such, information expressed in XML will need to be integrated with existing information systems, which are mostly based on structured data models such as relational, object-oriented or object/relational data models. This paper shows how our previous framework for integrating heterogeneous structured data sources can also be used for integrating XML data sources with each other and/or with other structured data sources.

In our approach, the constructs and transformations of modelling languages such as ER, XML etc. are defined in terms of the constructs and transformations of a lower-level graph-based data model. This allows constructs from multiple modelling languages to co-exist within the same intermediate schema, thus avoiding the need for a high-level common data model and the semantic mismatches that this can bring about. Transformations between schemas are expressed as sequences of primitive transformations and a key feature of them is that they are automatically reversible. This allows automatic translation of data, queries and updates between semantically equivalent or overlapping heterogenous schemas.

1 Introduction

The presentation-oriented nature of HTML has been widely recognised as being an impediment to building efficient search engines and query languages for information available on the WWW. This has led to the emergence of XML as a more effective means of describing the semantic content of WWW documents, with presentational information being specified using a separate language such as XSL. However, although much superior to HTML for describing document content, XML is still to some extent presentation-oriented in the sense that the structuring of the data for a particular application tends to be such as to suit the later presentation of the data. This is due to XML's hierarchical nature, with tags being nested inside each other, requiring document designers to make an *a priori* choice as to the ordering of the nesting. Whilst languages such as **DTD**s or **XML Schema** serve to structure XML documents, rather like a relational model structures relational databases, it is still the case that an essentially hierarchical model is being used.

For example, consider an application where a bank has records of customers, their accounts, and the bank site where the account is held. A site may have accounts belong to different customers and a customer may have accounts at several sites. Figure 2(a) shows how one might list details of customers in XML, detailing under each customer the account and the site of the account. Alternatively, Figure 2(b) shows how the same information could be listed by site, with details of accounts, and the customer holding the account listed under each site.

The example appears to be that of a many-many relationship between customers and sites. Such choices of ordering as made in Figures 2(a) and (b) do not arise in data models such as ER or UML, which are based on the classification of entities into types or classes, with relationships between them. For example, Figure 1 shows an ER model for the same data as in Figures 2(a) and (b). However, in XML the order of data can be significant and there may be semantic information embedded within XML documents which is assumed by applications but which is not deducible from the document itself. Looking at Figure 1(a) for example, it might be the case that the first account listed for any customer is to be used for charging any banking costs to e.g. charges for customer Jones will be made to account 4411 and not to account 6676.

This possibility that ordering may or may not be significant in XML means that it is desirable to use a semantic data model as the basis for integrating XML data sources with each other and/or with other structured data sources, rather than to use XML itself. Note that this does not preclude the use of XML as the data transfer mechanism, just that the modelling of multiple data sources requires something more structured.

The approach that we present in this paper extends our previous work on integrating structured data sources [20, 16, 15]. In this work, we have used as the common data model a low-level **hypergraph-based data model (HDM)**. One advantage of this HDM is that it separates the definition of the data sets from the definition of constraints on the values of these data sets. All data is held as either nodes representing sets of values, or edges representing relationships between these sets. Taking the ER schema in Figure 1 as an example, this means that the attributes and entities are represented as HDM nodes while the associations between attribute and entities, and the relationships between entities are represented as HDM edges with appropriate constraints on their cardinality. We will see later in the paper that this separation in the HDM of data sets and constraints on them is useful for modelling XML data, since ordering of XML elements can be represented by extra node and edge information, leaving any other constraints on the data unchanged.

Our previous work defined a set of primitive transformations for adding or deleting nodes, edges and constraints to or from an HDM schema. Higher-level modelling languages and primitive schema transformations for those languages are defined in terms of this lower-level HDM and its primitive transformations. Hence another advantage of using a low-level common data model is that it provides a unifying semantics for higher-level modelling constructs. In [16] we proposed a generic method for specifying the semantics of a higher-level modelling language in terms of the HDM, showing how the set of primitive transformations for the language can then be automatically derived from this specification. Transformations on higher-level modelling languages can be applied by a user to map between schemas expressed in the same or in different modelling languages. The use of a unifying underlying data model allows constructs from different modelling languages to be mixed within the same intermediate schema. Hence, using the HDM is *not* a substitute for using a standard data modelling language, such as the ER, relational, or UML models, but instead a common underpinning to facilitate the integration of data expressed in these standard languages.

1.1 Outline of the paper

We begin the paper in Section 2 by expanding upon our bank application example described above, which will serve as the running example for the remainder of the paper.

In Section 3 we extend our previous work to show how XML can be represented in the HDM. This leads to the first contribution of the paper — providing a common underpinning for structured data models and XML, and hence the possibility of transforming between them and integrating them.

In Section 4 we discuss how XML documents can be transformed into an ER representation, and from there into each other, thus providing a framework in which XML data sources can be integrated and queried in conjunction with each other and with other structured data sources. This leads to the second contribution of the paper — providing a method for transforming between ER and XML representations which allows complete control over whether the various elements of the XML representation have set or list-based semantics.

In Section 5 we discuss related work. We give our concluding remarks in Section 6

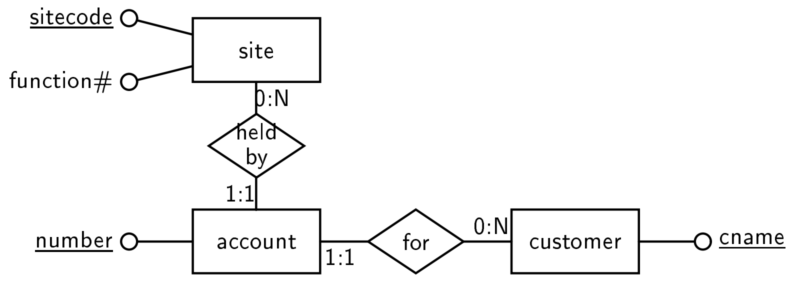


Figure 1: ER schema of the Bank database

2 Running Example

Figure 1 illustrates an ER schema for a simple banking application, where the customers of a bank hold accounts and accounts are managed at various sites where the bank has branches. When generating an XML representation of information held in such an ER schema, a programmer can structure the XML to suit the purpose of the application. For example, if a list of customers is required together with the location of each of their accounts, the XML description shown in Figure 2(a) may be used. If the application requires a list of the accounts held at each site, the XML description shown in Figure 2(b) may be used.

Apart from these variations in how the ER schema is navigated to produce a hierarchical data structure, there is also a choice as to whether to use XML elements or attributes. The examples given in Figures 2(a) and (b) use XML elements to represent data (as is usually the case in most of the literature) but for the 'leaf' nodes we could equally well use attributes. Figure 2(c) takes this approach for the same data as shown in Figure 2(a). XML (with DTDs) also supports a tuple-based representation of data as illustrated in Figure 2(d) where duplication of data is avoided.

3 Representing XML in the HDM

In [16] we showed how the HDM can represent a number of higher-level, structured modelling languages such as the ER, relational and UML data models. We also showed how it is possible to transform the constructs of one modelling language into those of another during the process of integrating multiple heterogeneous schemas into a single global schema[1]. By extending our work to specify how XML can be represented in the HDM, we are adding XML to the set of modelling languages whose schemas can be transformed into each other and integrated using our framework.

Structured data models typically have a set-based semantics *i.e.* there is no ordering on the extents of the types and relationships comprising the database schema, and no duplicate occurrences. XML's semi-structured nature and the fact that it is presentation-oriented means that lists need to be representable in the HDM, as opposed to just sets which were sufficient for our previous work on transforming and integrating structured data models. In particular, lists are needed because the order in which elements appear within an XML document may be significant to applications and this information should not be lost when transforming and integrating XML documents.

Thus we extend the notions of nodes and edges in HDM schemas (which respectively correspond to types and relationships in higher-level modelling languages) so that the extent of a node or edge may be either a set or a list. For reasons of space we refer the reader to our earlier work [20, 16, 15] for a full definition of the HDM. Here we give a simplified summary of it together with the extensions needed for representing XML:

[1]This integration process may be either 'bottom-up', as in federated architectures, or 'top-down', as in mediator architectures, and our approach is equally applicable to either.

```
⟨customer⟩
  ⟨cname⟩Jones⟨/cname⟩
  ⟨account⟩
    ⟨number⟩4411⟨/number⟩
    ⟨site⟩
      ⟨sitecode⟩32⟨/sitecode⟩
    ⟨/site⟩
  ⟨/account⟩
  ⟨account⟩
    ⟨number⟩6976⟨/number⟩
    ⟨site⟩
      ⟨sitecode⟩56⟨/sitecode⟩
      ⟨function⟩Business⟨/function⟩
    ⟨/site⟩
  ⟨/account⟩
⟨/customer⟩
⟨customer⟩
  ⟨cname⟩Frazer⟨/cname⟩
  ⟨account⟩
    ⟨number⟩8331⟨/number⟩
    ⟨site⟩
      ⟨sitecode⟩32⟨/sitecode⟩
    ⟨/site⟩
  ⟨/account⟩
⟨/customer⟩
```

(a) by customer, elements preferred

```
⟨site⟩
  ⟨sitecode⟩32⟨/sitecode⟩
  ⟨account⟩
    ⟨number⟩4411⟨/number⟩
    ⟨customer⟩
      ⟨cname⟩Jones⟨/cname⟩
    ⟨/customer⟩
  ⟨/account⟩
  ⟨account⟩
    ⟨number⟩8331⟨/number⟩
    ⟨customer⟩
      ⟨cname⟩Frazer⟨/cname⟩
    ⟨/customer⟩
  ⟨/account⟩
⟨/site⟩
⟨site⟩
  ⟨sitecode⟩56⟨/sitecode⟩
  ⟨function⟩Business⟨/function⟩
  ⟨account⟩
    ⟨number⟩6976⟨/number⟩
    ⟨customer⟩
      ⟨cname⟩Jones⟨/cname⟩
    ⟨/customer⟩
  ⟨/account⟩
⟨/site⟩
```

(b) by site, elements preferred

```
⟨customer cname=⟨Jones⟩⟩
  ⟨account number=⟨4411⟩⟩
    ⟨site sitecode=⟨32⟩/⟩
  ⟨/account⟩
  ⟨account number=⟨6976⟩⟩
    ⟨site sitecode=⟨56⟩ function=⟨Business⟩/⟩
  ⟨/account⟩
⟨/customer⟩
⟨customer cname=⟨Frazer⟩⟩
  ⟨account number=⟨8331⟩⟩
    ⟨site sitecode=⟨32⟩/⟩
  ⟨/account⟩
⟨/customer⟩
```

(c) by customer, attributes preferred

```
⟨customer cid=⟨c1⟩ cname=⟨Jones⟩/⟩
⟨customer cid=⟨c2⟩ cname=⟨Frazer⟩/⟩
⟨account aid=⟨a1⟩ number=⟨4411⟩ cid=⟨c1⟩ sid=⟨s1⟩/⟩
⟨account aid=⟨a2⟩ number=⟨6976⟩ cid=⟨c1⟩ sid=⟨s2⟩/⟩
⟨account aid=⟨a3⟩ number=⟨8331⟩ cid=⟨c2⟩ sid=⟨s1⟩/⟩
⟨site sid=⟨s1⟩ sitecode=⟨32⟩/⟩
⟨site sid=⟨s2⟩ sitecode=⟨56⟩ function=⟨Business⟩/⟩
```

(d) tuple-based

Figure 2: Example XML data files for the Bank database

A **schema** in the HDM is a triple (Nodes,Edges,Constraints). Nodes and edges are identified by their **schemes**, delimited by double chevrons ⟨⟨ ... ⟩⟩. The scheme of a node n consists of just the node itself, $\langle\langle n\rangle\rangle$. The scheme of an edge labelled l between nodes $n_1,\ldots,n_m$ is $\langle\langle l,n_1,\ldots,n_m\rangle\rangle$. Edges can also link other edges, so more generally the scheme of an edge is $\langle\langle l,s_1,\ldots,s_m\rangle\rangle$ for some schemes $s_1,\ldots,s_m$. Two primitive transformations are available for adding a node or an edge to an HDM schema, S, to yield a new schema:

addNode(s,q,i,c)

addEdge(s,q,i,c)

Here, s is the scheme of the node or edge being added and q is a query on S which defines the extent of s in terms of the extents of the existing schema constructs (so adding s does not change the information content of S)[2]. i is one of set or list, indicating the collection type of the extent of s, and c is a boolean condition on instances of S which must hold for the transformation to be applicable for that particular instance. Optionally, list may take an argument which determines the ordering of instances of s.

Often the argument c will simply be true, indicating that the transformation applies for all instances of S, and in our previous work i has always been set. In [15] we allowed q to take the special value void, meaning that s can not be derived from the other constructs of S. This is needed when a transformation pathway is being set up between non-equivalent schemas *e.g.* between a component schema and a global schema. For convenience, we use addNode(s,q,i) as a shorthand for addNode(s,q,i,true), addNode(s,q) for addNode(s,q,set,true), and expandNode(s) for addNode(s,void,set,true). We use similar abbreviations for adding edges.

There are also two primitive transformations for deleting a node or an edge from an HDM schema S, delNode(s,q,i,c) and delEdge(s,q,i,c), where s is the scheme of the node or edge being deleted and q is a query which defines how the extent of s can be reconstructed from the extents of the remaining constructs (so deleting s does not change the information content of S), and i the collection type of s. c is again a boolean condition on instances of S which must hold for the transformation to be applicable for that particular instance. Similar shorthands as for the add transformations are used. There are similarly two primitive transformations for adding/deleting a constraint from an HDM schema, addConstraint$(s, constraint)$ and delConstraint$(s, constraint)$.

Supporting a list collection type does not actually require the HDM to be extended and the above primitive transformations on the HDM can be viewed as 'syntactic sugar' for the primitive transformations we used in previous work. In particular, lists can be supported by introducing a reserved node order, the extent of which is the set of natural numbers. For any scheme s whose extent needs to be viewed as a list, an extra unlabelled edge $\langle\langle$_,s,order$\rangle\rangle$ is used whose instances assign an ordinality to each instance of s. The instances of $\langle\langle$_,s,order$\rangle\rangle$ do not necessary need to be numbered consecutively, and the ordering of instances of s can be relative to the ordering of instances other schemes.

3.1 Specifying XML in terms of the HDM

Table 1 summarises how the methodology we described in [16] can be used to build an HDM representation of an XML document. In particular:

1. An XML element may exist by itself and is not dependent on the existence of any other information. Thus, each XML element e is what we term a **nodal** construct [16] and is represented by a node $\langle\langle$xml:$e\rangle\rangle$ in the HDM[3]. Each instance of e in an XML document corresponds to an instance of the HDM node $\langle\langle$xml:$e\rangle\rangle$.

2. An XML attribute a of an XML element e may only exist in the context of e, and hence a is what we term a **nodal-linking** construct. It is represented by a node xml:e:a in the HDM, together with an associated unlabelled edge $\langle\langle$_,xml:e,xml:e:$a\rangle\rangle$ connecting the HDM node representing the attribute a to the HDM node representing the element e. A constraint states that each instance of the attribute is related to at least one instance of the element[4].

[2]We first developed our definitions of schema equivalence, schema subsumption and schema transformation in the context of an ER common data model [13, 14] and then applied them to the more general setting of the HDM [20]. A comparison with other approaches to schema equivalence and schema transformation can be found in [14].

[3]Because it is possible to have present within the same HDM schema constructs from schemas expressed in different higher-level modelling notations, higher-level constructs are distinguished at the HDM level by adding a prefix to their name. This prefix is xml for XML constructs and er for ER constructs.

[4]We use some short-hand for expressing cardinality constraints, makecard$(\langle\langle l, s_1, s_2\rangle\rangle, \{l_1 \ldots u_1\}, \{l_2 \ldots u_2\})$, which denotes the following cardinality constraint on the edge $\langle\langle l, s_1, s_2\rangle\rangle$:

$$(\forall i_1 \in s_1 \,.\, l_1 \leq |\{y|\langle x,y\rangle \in \langle\langle l, s_1, s_2\rangle\rangle \wedge x = i_1\}| \leq u_1) \wedge (\forall i_2 \in s_2 \,.\, l_2 \leq |\{x|\langle x,y\rangle \in \langle\langle l, s_1, s_2\rangle\rangle \wedge y = i_2\}| \leq u_2)$$

Higher Level Construct		Equivalent HDM Representation	
Construct	**element (Elem)**	Node	⟨⟨xml:e⟩⟩
Class	nodal, set		
Scheme	⟨⟨e⟩⟩		
Construct	**attribute (Att)**	Node	⟨⟨xml:e:a⟩⟩
Class	nodal-linking, constraint, list	Edge	⟨⟨_,xml:e,xml:e:a⟩⟩
		Links	⟨⟨xml:e⟩⟩
Scheme	⟨⟨e, a⟩⟩	Cons	makeCard(⟨⟨_,xml:e,xml:e:a⟩⟩, $\{0..1\}, \{1..N\}$)
Construct	**nest-list (List)**	Edge	⟨⟨_,xml:e,xml:e_s⟩⟩, ⟨⟨_, ⟨⟨_, xml:e, xml:e_s⟩⟩, order⟩⟩
Class	linking, constraint, list	Links	⟨⟨xml:e⟩⟩, ⟨⟨xml:e_s⟩⟩
Scheme	⟨⟨e, e_s⟩⟩	Cons	makeCard(⟨⟨_,⟨⟨_,xml:e,xml:e_s⟩⟩,order⟩⟩, $\{1..1\}, \{0..N\}$)
Construct	**nest-set (Set)**	Edge	⟨⟨_,xml:e,xml:e_s⟩⟩
Class	linking, set	Links	⟨⟨xml:e⟩⟩, ⟨⟨xml:e_s⟩⟩
Scheme	⟨⟨e, e_s⟩⟩		

Table 1: Specifying XML constructs in the HDM

3. XML allows any number of elements $e_1, \ldots, e_n$ to be nested within an element e. This nesting of $e_1, \ldots, e_n$ is represented by a set of edges ⟨⟨_,xml:e,xml:e_1⟩⟩, ..., ⟨⟨_,xml:e,xml:e_n⟩⟩. Each such edge is an individual **linking** construct, with scheme ⟨⟨_,xml:e,xml:e_s⟩⟩.

   Each such edge may have list or set semantics. For list semantics, there is an extra edge between the edge ⟨⟨_,xml:e,xml:e_s⟩⟩ and the node order, and a cardinality constraint which states that each instance of ⟨⟨_,xml:e,xml:e_s⟩⟩ is related to precisely one instance of order.

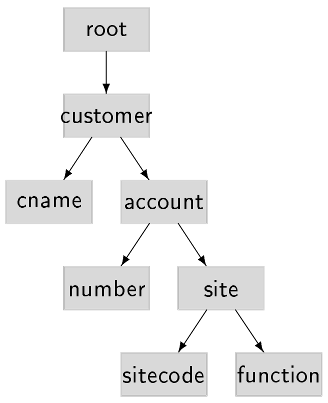
4. XML allows plain text to be placed within a pair of element tags. If a DTD is present, then this text is denoted as PCDATA. We thus assume there is an HDM node called pcdata whose extent consists of plain text instances. An element e can then be associated with a fragment of plain text by means of an edge ⟨⟨_,e,pcdata⟩⟩.

To illustrate this representation of XML documents in the HDM, we show in Figure 3 the HDM schemas corresponding to the XML documents of Figure 2 (we have now shown the constraints and the links that each edge has to the order node). In the HDM schema (d), the common child node cid between site and customer, and aid between site and account, arise if we assume the presence of a DTD in XML document (d) with ID attributes cid and aid on customer and account, and corresponding IDREF attributes on site.

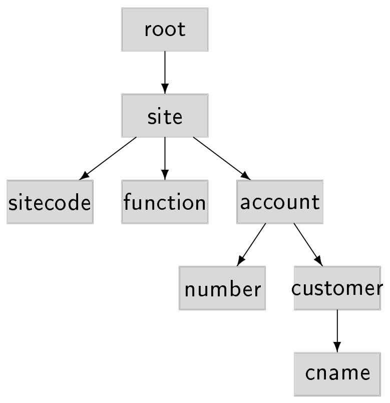
In general, elements within an XML document may be repeated, with identical attributes and nested elements within them. Hence elements and attributes are uniquely identified by their position within the document. In representing an XML document as an instance of an HDM schema, we thus assume that all nodes are assigned unique identifiers which are generated in some way from their position within the document. For example, the XML elements customer, cname and account of Figure 2(a) can be represented by the following instance of the schemes ⟨⟨root⟩⟩, ⟨⟨customer⟩⟩, ⟨⟨cname⟩⟩, ⟨⟨pcdata⟩⟩, and ⟨⟨account⟩⟩ of the HDM schema in Figure 3(a), where r1, c1, c2, cn1, cn2, a1, a2, a3 are unique identifiers:

⟨⟨root⟩⟩={r1}
⟨⟨customer⟩⟩={c1,c2}
⟨⟨cname⟩⟩={cn1, cn2}
⟨⟨pcdata⟩⟩={Jones, Frazer}
⟨⟨account⟩⟩={a1,a2,a3}

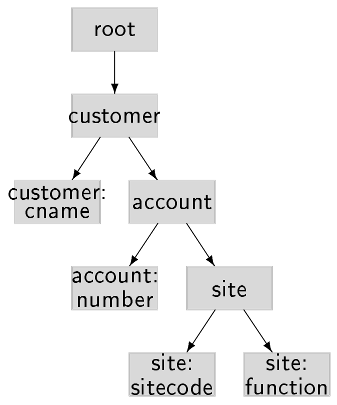
The nesting relationships between these elements are then represented by the following instance of the schemes ⟨⟨_,root,customer⟩⟩, ⟨⟨_,⟨⟨_,root,customer⟩⟩,order⟩⟩, ⟨⟨_,customer,cname⟩⟩, ⟨⟨_,⟨⟨_,customer, cname⟩⟩,order⟩⟩, ⟨⟨_,cname,pcdata⟩⟩, ⟨⟨_,⟨⟨_,cname,pcdata⟩⟩,order⟩⟩, ⟨⟨_,customer,account⟩⟩, and ⟨⟨_,⟨⟨_, customer,account⟩⟩,order⟩⟩



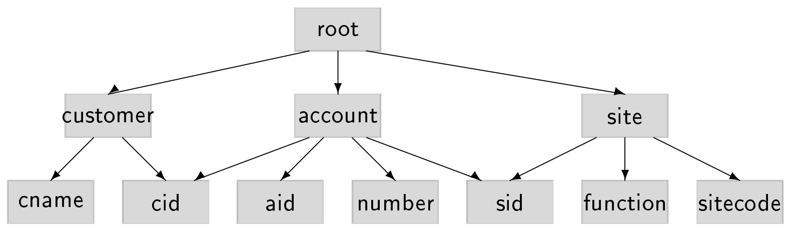
(a) HDM for Figure 2(a)



(b) HDM for Figure 2(b)



(c) HDM for Figure 2(c)



(d) HDM for Figure 2(d)

Figure 3: HDM schemas for the XML Documents (a)-(d)

of Figure 3(a). This states that the one root document r1 of the XML document contains two ordered customers c1 and c2, that c1 contains, in order, one cname cn1 and two accounts a1 and a2, and that c2 contains, in order, one cname cn2 and one account a3:

⟪_,root,customer⟫= {⟨r1,c1⟩, ⟨r1,c2⟩}
⟪_,⟪_,root,customer⟫,order⟫={⟨⟨r1,c1⟩,1⟩, ⟨⟨r1,c2⟩,2⟩}
⟪_,customer,cname⟫={⟨c1,cn1⟩, ⟨c2,cn2⟩}
⟪_,⟪_,customer,cname⟫,order⟫={⟨⟨c1,cn1⟩,1⟩, ⟨⟨c2,cn2⟩,1⟩}
⟪_,cname,pcdata⟫={⟨cn1,Jones⟩, ⟨cn2,Frazer⟩}
⟪_,⟪_,cname,pcdata⟫,order⟫={⟨⟨cn1,Jones⟩,1⟩, ⟨⟨cn2,Frazer⟩,1⟩}
⟪_,customer,account⟫={⟨c1,a1⟩, ⟨c1,a2⟩, ⟨c2,a3⟩}
⟪_,⟪_,customer,account⟫,order⟫={⟨⟨c1,a1⟩,2⟩, ⟨⟨c1,a2⟩,3⟩, ⟨⟨c2,a3⟩,2⟩}

An HDM representation which assumed set-based as opposed to list-based semantics for element containment would not contain the schemes ⟪_,⟪_,root,customer⟫,order⟫, ⟪_,⟪_,customer, cname⟫,order⟫, ⟪_,⟪_,cname,pcdata⟫,order⟫ and ⟪_,⟪_,customer,account⟫,order⟫, and would have the same extents as above for the remaining schemes.

An HDM representation of the entire XML document of Figure 2(a) would list and order all the elements in a similar way to the above fragments.

3.2 Primitive Transformations on XML

In [16] we describe how, once the constructs of some higher-level modelling language have been defined in terms of the HDM constructs, this definition can be used to *automatically derive* the necessary set of primitive transformations on schemas expressed in that language. Thus, from the specification of XML given in Table 1 and described in the previous section, the primitive

Transformation on XML	Equivalent Transformation on HDM
$\mathsf{renameElem_{xml}}(e,e')$	$\mathsf{renameNode}(\langle\langle \mathsf{xml}{:}e\rangle\rangle,\langle\langle \mathsf{xml}{:}e'\rangle\rangle)$
$\mathsf{addElem_{xml}}(e,q)$	$\mathsf{addNode}(\langle\langle \mathsf{xml}{:}e\rangle\rangle,q)$
$\mathsf{delElem_{xml}}(e,q)$	$\mathsf{delNode}(\langle\langle \mathsf{xml}{:}e\rangle\rangle,q)$
$\mathsf{renameAtt_{xml}}(a,a')$	$\mathsf{renameNode}(\langle\langle \mathsf{xml}{:}e{:}a\rangle\rangle,\langle\langle \mathsf{xml}{:}e{:}a'\rangle\rangle)$
$\mathsf{addAtt_{xml}}(e,a,q_{att},q_{assoc})$	$\mathsf{addNode}(\langle\langle \mathsf{xml}{:}e{:}a\rangle\rangle,q_{att})$; $\mathsf{addEdge}(\langle\langle \_,\mathsf{xml}{:}e,\mathsf{xml}{:}e{:}a\rangle\rangle,q_{assoc})$; $\mathsf{addConstraint}(x \in \langle\langle \mathsf{xml}{:}e{:}a\rangle\rangle \rightarrow \langle \_,x\rangle \in \langle\langle \_,\mathsf{xml}{:}e,\mathsf{xml}{:}e{:}a\rangle\rangle)$
$\mathsf{delAtt_{xml}}(e,a,q_{att},q_{assoc})$	$\mathsf{delConstraint}(x \in \langle\langle \mathsf{xml}{:}e{:}a\rangle\rangle \rightarrow \langle \_,x\rangle \in \langle\langle \_,\mathsf{xml}{:}e,\mathsf{xml}{:}e{:}a\rangle\rangle;)$ $\mathsf{delEdge}(\langle\langle \_,\mathsf{xml{:}e},\mathsf{xml}{:}e{:}a\rangle\rangle,q_{assoc})$; $\mathsf{delNode}(\langle\langle \mathsf{xml}{:}e{:}a\rangle\rangle,q_{att})$
$\mathsf{addList_{xml}}(\langle\langle e,e_s\rangle\rangle,q,p)$	$\mathsf{addEdge}(\langle\langle \_,\mathsf{xml}{:}e,\mathsf{xml}{:}e_s\rangle\rangle,q,\mathsf{list}(p))$
$\mathsf{delList_{xml}}(\langle\langle e,e_s\rangle\rangle,q,p)$	$\mathsf{delEdge}(\langle\langle \_,\mathsf{xml}{:}e,\mathsf{xml}{:}e_s\rangle\rangle,q,\mathsf{list}(p))$
$\mathsf{addSet_{xml}}(\langle\langle e,e_s\rangle\rangle,q)$	$\mathsf{addEdge}(\langle\langle \_,\mathsf{xml}{:}e,\mathsf{xml}{:}e_s\rangle\rangle,q)$
$\mathsf{delSet_{xml}}(\langle\langle e,e_s\rangle\rangle,q)$	$\mathsf{delEdge}(\langle\langle \_,\mathsf{xml}{:}e,\mathsf{xml}{:}e_s\rangle\rangle,q)$

Table 2: Derived transformations on XML

```
⟨site⟩
  ⟨sitecode⟩32⟨/sitecode⟩
  ⟨account⟩
    ⟨number⟩4411⟨/number⟩
    ⟨customer⟩
      ⟨cname⟩Jones⟨/cname⟩
    ⟨/customer⟩
  ⟨/account⟩
⟨/site⟩
⟨site⟩
  ⟨sitecode⟩32⟨/sitecode⟩
  ⟨account⟩
    ⟨number⟩8331⟨/number⟩
    ⟨customer⟩
      ⟨cname⟩Frazer⟨/cname⟩
    ⟨/customer⟩
  ⟨/account⟩
⟨/site⟩
⟨site⟩
  ⟨sitecode⟩56⟨/sitecode⟩
  ⟨function⟩Business⟨/function⟩
  ⟨account⟩
    ⟨number⟩6976⟨/number⟩
    ⟨customer⟩
      ⟨cname⟩Jones⟨/cname⟩
    ⟨/customer⟩
  ⟨/account⟩
⟨/site⟩
```

Figure 4: Transformed XML

transformations shown in Table 2 can be automatically derived. The left-hand column of this table gives the names and arguments of the XML transformations and the right-hand column gives their implementation as sequences of the primitive transformations on the underlying HDM representation.

To illustrate the use of these primitive transformations on XML, we give below a sequence of primitive transformations that transform the document in Figure 2(a) to that in Figure 4. The HDM schema representation of the former is shown in Figure 3(a) and of the latter in Figure 3(b).

$$\mathsf{addList_{xml}}(\langle\langle \mathsf{root},\mathsf{site}\rangle\rangle,\{\langle \mathsf{r1},x\rangle \mid \langle x\rangle \in \langle\langle \mathsf{site}\rangle\rangle\},\ \mathsf{after}(\langle\langle \mathsf{root},\mathsf{customer}\rangle\rangle))$$
$$\mathsf{addList_{xml}}(\langle\langle \mathsf{site},\mathsf{account}\rangle\rangle,\{\langle x,y\rangle \mid \langle y,x\rangle \in \langle\langle \mathsf{account},\mathsf{site}\rangle\rangle\},\ \mathsf{after}(\langle\langle \mathsf{site},\mathsf{function}\rangle\rangle))$$

$\mathsf{addList_{xml}}(\langle\langle \mathsf{account,customer}\rangle\rangle, \{\langle x,y\rangle \mid \langle y,x\rangle \in \langle\langle \mathsf{customer,account}\rangle\rangle\}, \mathsf{after}(\langle\langle \mathsf{account,number}\rangle\rangle))$
$\mathsf{delList_{xml}}(\langle\langle \mathsf{account,site}\rangle\rangle, \{\langle x,y\rangle \mid \langle y,x\rangle \in \langle\langle \mathsf{site,account}\rangle\rangle\}, \mathsf{after}(\langle\langle \mathsf{account,customer}\rangle\rangle))$
$\mathsf{delList_{xml}}(\langle\langle \mathsf{customer,account}\rangle\rangle, \{\langle x,y\rangle \mid \langle y,x\rangle \in \langle\langle \mathsf{account,customer}\rangle\rangle\}, \mathsf{after}(\langle\langle \mathsf{customer,cname}\rangle\rangle))$
$\mathsf{delList_{xml}}(\langle\langle \mathsf{root,customer}\rangle\rangle, \{\langle \mathsf{r1},x\rangle \mid \langle x\rangle \in \langle\langle \mathsf{customer}\rangle\rangle\}, \mathsf{before}(\langle\langle \mathsf{root,site}\rangle\rangle))$

Notice that the above transformation consists of a 'growing phase' where new constructs are added to the XML model, followed by a 'shrinking phase' where the constructs now rendered redundant are removed. This is a general characteristic of schema transformations expressed within our framework.

The availability of a transformation pathway from one schema to another allows queries expressed on one schema to be automatically translated onto the other. For example, the following query on Figure 3(a) finds the names of customers with accounts at site 32[5]:

$$\{n \mid \langle r,c\rangle \in \langle\langle \mathsf{root,customer}\rangle\rangle \wedge \langle c,n\rangle \in \langle\langle \mathsf{customer,name}\rangle\rangle \wedge \langle c,a\rangle \in \langle\langle \mathsf{customer,account}\rangle\rangle \wedge \\ \langle a,s\rangle \in \langle\langle \mathsf{account,site}\rangle\rangle \wedge \langle s,sc\rangle \in \langle\langle \mathsf{site,sitecode}\rangle\rangle \wedge sc = 32\}$$

By substituting deleted constructs appearing in this query by their restoring query *i.e.* by the 3rd argument of the $\mathsf{delList_{xml}}()$ transformations above, it is possible to translate the query into the following equivalent query on Figure 3(b) (see [15] for a general discussion of query/update/data translation in our framework):

$$\{n \mid \langle r,c\rangle \in \{\langle \mathsf{r1},x\rangle \mid \langle x\rangle \in \langle\langle \mathsf{customer}\rangle\rangle\} \wedge \langle c,n\rangle \in \langle\langle \mathsf{customer,name}\rangle\rangle \wedge \\ \langle c,a\rangle \in \{\langle x,y\rangle \mid \langle y,x\rangle \in \langle\langle \mathsf{account,customer}\rangle\rangle\} \wedge \\ \langle a,s\rangle \in \{\langle x,y\rangle \mid \langle y,x\rangle \in \langle\langle \mathsf{site,account}\rangle\rangle\} \wedge \langle s,sc\rangle \in \langle\langle \mathsf{site,sitecode}\rangle\rangle \wedge sc = 32\}$$

Any sequence of primitive transformations $t_1; \ldots ; t_n$ is automatically reversible by the sequence $t_n^{-1}; \ldots ; t_1^{-1}$, where the inverse of an add transformation is a del transformation with the same arguments, and vice versa. Thus, the reverse transformation from Figure 3(b) to Figure 3(a) can be automatically derived to be:

$\mathsf{addList_{xml}}(\langle\langle \mathsf{root,customer}\rangle\rangle, \{\langle \mathsf{r1},x\rangle \mid \langle x\rangle \in \langle\langle \mathsf{customer}\rangle\rangle\}, \mathsf{before}(\langle\langle \mathsf{root,site}\rangle\rangle))$
$\mathsf{addList_{xml}}(\langle\langle \mathsf{customer,account}\rangle\rangle, \{\langle x,y\rangle \mid \langle y,x\rangle \in \langle\langle \mathsf{account,customer}\rangle\rangle\}, \mathsf{after}(\langle\langle \mathsf{customer,cname}\rangle\rangle))$
$\mathsf{addList_{xml}}(\langle\langle \mathsf{account,site}\rangle\rangle, \{\langle x,y\rangle \mid \langle y,x\rangle \in \langle\langle \mathsf{site,account}\rangle\rangle\}, \mathsf{after}(\langle\langle \mathsf{account,customer}\rangle\rangle))$
$\mathsf{delList_{xml}}(\langle\langle \mathsf{account,customer}\rangle\rangle, \{\langle x,y\rangle \mid \langle y,x\rangle \in \langle\langle \mathsf{customer,account}\rangle\rangle\}, \mathsf{after}(\langle\langle \mathsf{account,number}\rangle\rangle))$
$\mathsf{delList_{xml}}(\langle\langle \mathsf{site,account}\rangle\rangle, \{\langle x,y\rangle \mid \langle y,x\rangle \in \langle\langle \mathsf{account,site}\rangle\rangle\}, \mathsf{after}(\langle\langle \mathsf{site,function}\rangle\rangle))$
$\mathsf{delList_{xml}}(\langle\langle \mathsf{root,site}\rangle\rangle, \{\langle \mathsf{r1},x\rangle \mid \langle x\rangle \in \langle\langle \mathsf{site}\rangle\rangle\}, \mathsf{after}(\langle\langle \mathsf{root,customer}\rangle\rangle))$

This reverse transformation can now be used to translate queries on Figure 3(b) to queries on Figure 3(a).

We finally observe the similarity of Figure 4 to the document in Figure 2(b), which in fact has the same schema, Figure 3(b). It is not possible to capture the non-duplication of site 32 in Figure 2(b) using the above XML-to-XML transformation. It is however possible to transform Figure 2(a) to Figure 2(b) going via the ER model of Figure 1, and we discuss how in Section 4 below.

3.3 Additional information supplied by a DTD

A DTD constrains an XML schema, limiting which elements may appear in a document and the order in which they appear. In [6] DTDs are treated as an integral component of XML. We choose here to separate the definition of XML from the definition of DTDs for XML documents in order to facilitate future incorporation into our framework of the emerging XML Schema standard [10] instead of DTDs. Figure 5 shows DTDs for the XML examples in Figure 2.

```
⟨!ELEMENT bank (customer)∗⟩
⟨!ELEMENT customer (cname, account∗)⟩
⟨!ELEMENT cname (#PCDATA)⟩
⟨!ELEMENT account (number, site)⟩
⟨!ELEMENT number (#PCDATA)⟩
⟨!ELEMENT site (sitecode, function?)⟩
⟨!ELEMENT sitecode (#PCDATA)⟩
⟨!ELEMENT function (#PCDATA)⟩
```

(a)

```
⟨!ELEMENT bank (site)∗⟩
⟨!ELEMENT site (sitecode, function?, account∗)⟩
⟨!ELEMENT sitecode (#PCDATA)⟩
⟨!ELEMENT function (#PCDATA)⟩
⟨!ELEMENT account (number, customer)⟩
⟨!ELEMENT number (#PCDATA)⟩
⟨!ELEMENT customer (cname)⟩
⟨!ELEMENT cname (#PCDATA)⟩
```

(b)

```
⟨!ELEMENT bank (customer)∗⟩
⟨!ELEMENT customer (account∗)⟩
⟨!ATTLIST customer cname CDATA #REQUIRED⟩
⟨!ELEMENT account (site)⟩
⟨!ATTLIST account number CDATA #REQUIRED⟩
⟨!ELEMENT site EMPTY⟩
⟨!ATTLIST site sitecode CDATA #REQUIRED⟩
⟨!ATTLIST site function CDATA #IMPLIED⟩
```

(c)

```
⟨!ELEMENT bank (customer | account | site)∗⟩
⟨!ELEMENT customer EMPTY⟩
⟨!ATTLIST customer cid ID #REQUIRED⟩
⟨!ATTLIST customer cname CDATA #REQUIRED⟩
⟨!ELEMENT site EMPTY⟩
⟨!ATTLIST site sid ID REQUIRED⟩
⟨!ATTLIST site sitecode CDATA #REQUIRED⟩
⟨!ATTLIST site function CDATA #IMPLIED⟩
⟨!ELEMENT account EMPTY⟩
⟨!ATTLIST account aid ID #REQUIRED⟩
⟨!ATTLIST account number CDATA #REQUIRED⟩
⟨!ATTLIST account sid IDREF #REQUIRED⟩
⟨!ATTLIST account cid IDREF #REQUIRED⟩
```

(d)

Figure 5: DTDs for the Bank XML examples

Higher Level Construct		Equivalent HDM Representation	
Construct	**required (REQ)**	Links	⟨⟨xml:e:a⟩⟩
Class	constraint	Cons	makeCard(⟨⟨_,xml:e,xml:e:a⟩⟩, $\{1..N\}$, $\{1..N\}$)
Scheme	⟨⟨e, a⟩⟩		
Construct	**fixed (FIX)**	Links	⟨⟨xml:e:a⟩⟩
Class	constraint	Cons	makeCard(⟨⟨_,xml:e,xml:e:a⟩⟩, $\{1..1\}$, $\{1..N\}$); makeCard(⟨⟨xml:e:a⟩⟩, $\{1..1\}$)
Scheme	⟨⟨e, a⟩⟩		
Construct	**identity(ID)**	Edge	⟨⟨xml:a,xml:e,id⟩⟩
Class	linking, constraint, list	Links	⟨⟨xml:e⟩⟩, ⟨⟨id⟩⟩
Scheme	⟨⟨e, a⟩⟩	Cons	makeCard(⟨⟨xml:a,xml:e,id⟩⟩, $\{1..1\}$, $\{0..1\}$)
Construct	**identity(IDREF)**	Edge	⟨⟨xml:a,xml:e,id⟩⟩
Class	linking, constraint, list	Links	⟨⟨xml:e⟩⟩, ⟨⟨id⟩⟩
Scheme	⟨⟨e, a⟩⟩	Cons	makeCard(⟨⟨xml:a,xml:e,id⟩⟩, $\{1..1\}$, $\{0..N\}$)

Table 3: Specifying !ATTLIST DTD constraints in the HDM

3.3.1 Handling the !ATTLIST DTD construct

The general form of a DTD attribute definition is
⟨!ATTLIST element_name attribute_name attribute_type attribute_constraint⟩
and Table 3 summarises how ATTLIST DTD constraints are represented in the HDM. We see

[5]We do not consider in this paper the issue of translating between different query languages, and assume a 'neutral' intermediate query notation, namely set comprehensions, into which queries submitted to local or global schemas can be translated as a first step.

that for the most part DTD constructs translate into constraints on the HDM nodes and edges representing the XML constructs. The one exception is the representation of ID, IDREF and IDREFS attributes. In more detail:

1. An attribute constraint of #IMPLIED is assumed by our representation of XML attributes in Table 1 since the attributes there are optional. Other attribute types serve to restrict this default constraint as follows:
   - #REQUIRED has a mandatory constraint added to the edge between a pair of element and attribute nodes.
   - #FIXED has a mandatory constraint added, and in addition makes the attribute node single valued.
2. Table 1 is correct for the CDATA and NMTOKEN attribute types in that no further constraints are needed in the HDM. For the attribute type NMTOKENS, the constraint on the edge $\langle\langle \_,\mathsf{xml}{:}e,\mathsf{xml}{:}e{:}a\rangle\rangle$ in Table 1 would be modified so that each instance of the element e can be associated with $0..N$ instead of $0..1$ instances of the attribute a. For the other DTD attribute types, the following alterations are needed:
   - An attribute of type ID identifies an element. Since there is a document-wide name space for element identifiers we use a single HDM node id whose extent consists of all instances of attributes which are element identifiers (as opposed to using a different node for each such attribute). There is a one-to-one association between the element e and id, forcing each instance of e to be associated with exactly one instance of id, and each instance of id to be associated with no more than one instance of e.
   - An attribute of type IDREF is similarly associated with the id node, with the difference that the constraint for id is now many-valued, since any number of elements can have an IDREF instance pointing to the same element. For attribute type IDREFS (not shown in Table 3) the cardinality of e would be $1..N$ instead of $1..1$.

3.3.2 Handling the !ELEMENT DTD construct

The general form of a DTD element definition is:
⟨!ELEMENT element_name ⟨content_pattern⟩⟩

The ⟨content_pattern⟩ places constraints on the extents of the edges representing XML element nesting. The definition of set-based element nesting in Table 1 allows the instances of nested elements $e_1,\ldots,e_n$ of an element e to appear in any order, which corresponds to the DTD content pattern ANY. A DTD occurrence operator m, such as the $*$ in $(e_1,\ldots,e_n)*$, will generate extra cardinality constraints on the links between the edges representing the nesting of elements and the node order. We refer to the absence of an occurrence operator as $m = \mathsf{none}$. For content patterns of the form $(e_1 \ldots e_n)m$ we map m to a cardinality constraint by applying the following rules, where $\mathsf{mapOcc}(\mathsf{none}) = 1..1$, $\mathsf{mapOcc}(?) = 0..1$, $\mathsf{mapOcc}(+) = 1..N$, and $\mathsf{mapOcc}(*) = 0..N$:

- A single element $(e_s)m$ generates the cardinality constraint:
  $\mathsf{makeCard}(\{x \mid \langle y,x\rangle \in \langle\langle \_,\langle\langle \_,\mathsf{xml}{:}e,\mathsf{xml}{:}e_s\rangle\rangle,\mathsf{order}\rangle\rangle\}, \mathsf{mapOcc}(m))$
- The choice pattern $(e_1 \mid \ldots \mid e_n)m$ generates the cardinality constraint:
  $\mathsf{makeCard}(\{x \mid \langle y,x\rangle \in \langle\langle \_,\langle\langle \_,\mathsf{xml}{:}e,\mathsf{xml}{:}e_1\rangle\rangle,\mathsf{order}\rangle\rangle \vee \ldots \vee$
  $\langle y,x\rangle \in \langle\langle \_,\langle\langle \_,\mathsf{xml}{:}e,\mathsf{xml}{:}e_n\rangle\rangle,\mathsf{order}\rangle\rangle\}, \mathsf{mapOcc}(m))$
- The sequence pattern $(e_1,\ldots,e_n)m$ generates the cardinality constraint:
  $\mathsf{makeCard}(\{x \mid \langle y,x\rangle \in \langle\langle \_,\langle\langle \_,\mathsf{xml}{:}e,\mathsf{xml}{:}e_1\rangle\rangle,\mathsf{order}\rangle\rangle, \mathsf{mapOcc}(m)\}) \vee \ldots \vee$
  $\mathsf{makeCard}(\{x \mid \langle y,x\rangle \in \langle\langle \_,\langle\langle \_,\mathsf{xml}{:}e,\mathsf{xml}{:}e_n\rangle\rangle,\mathsf{order}\rangle\rangle, \mathsf{mapOcc}(m)\})$

For pairs of elements e_s, e_t) appearing within a sequence $(\ldots,e_s,\ldots,e_t,\ldots)$, we may also infer the following pair of constraints:
$\langle x,y\rangle \in \langle\langle \_,\langle\langle \_,\mathsf{xml}{:}e,\mathsf{xml}{:}e_s\rangle\rangle,\mathsf{order}\rangle\rangle \rightarrow \exists z.\langle x,z\rangle \in \langle\langle \_,\langle\langle \_,\mathsf{xml}{:}e,\mathsf{xml}{:}e_t\rangle\rangle,\mathsf{order}\rangle\rangle \wedge y < z$
$\langle x,z\rangle \in \langle\langle \_,\langle\langle \_,\mathsf{xml}{:}e,\mathsf{xml}{:}e_t\rangle\rangle,\mathsf{order}\rangle\rangle \rightarrow \exists y.\langle x,y\rangle \in \langle\langle \_,\langle\langle \_,\mathsf{xml}{:}e,\mathsf{xml}{:}e_s\rangle\rangle,\mathsf{order}\rangle\rangle \wedge y < z$

4 Transforming between ER Models and XML Documents

In [16] we showed how ER schemas can be represented in the HDM, and we refer the reader to that paper for details. Here we recall that ER entity classes are nodal constructs represented by HDM nodes, ER relationships are linking constructs represented by an HDM edge and a cardinality constraint, and ER attributes are nodal-linking constructs represented by a node, an edge linking this node to the node representing the attribute's entity class, and a cardinality constraint. The primitive transformations on ER schemas consist of the operations add, del, expand, contract and rename on the constructs Entity, Attribute or Relationship.

4.1 Automated generation of a canonical ER schema from XML

Using the sets of primitive transformations on XML and ER schemas, an XML document or set of documents can be automatically transformed into a 'canonical' ER schema by applying the rules given below. For ease of reading, we have specified these rules at the level of the HDM, and the primitive transformations on the HDM, rather than at the higher level of the XML and ER constructs. As with the example in Section 3.2 above, the transformation consists of a 'growing phase' where new ER constructs are added to the schema, followed by a 'shrinking phase' where the XML constructs now rendered redundant are removed:

1. Each node representing an XML element (*i.e.* is named xml:e for some e and is not id, order or pcdata) generates an ER entity class of the same name e, with a key attribute also named e (this explicit key attribute is needed because there is no implicit notion of unique object identifiers in the ER model):

   addNode(⟪er:e⟫,⟪xml:e⟫)
   addNode(⟪er:e:e⟫,⟪xml:e⟫)
   addEdge(⟪_,er:e,er:e:e⟫, $\{\langle x,x\rangle \mid x \in$ ⟪xml:e⟫$\}$)
   addConstraint(makeCard(⟪_,er:e,er:e:e⟫,1..1,1..1))

   Note that for the purposes of this rule, the root of the document should also be considered as a node, so there will always be a root entity in the ER model. Each instance of the root entity will represent a distinct XML document.

2. Each node representing an XML attribute (*i.e.* is named xml:e:a for some e and a) generates an attribute of the ER entity class e[6]:

   addNode(⟪er:e:a⟫,⟪xml:e:a⟫)
   addEdge(⟪_,er:e,er:e:a⟫, ⟪_,xml:e,xml:e:a⟫)
   addConstraint(copyCard(⟪_,xml:e,xml:e:a⟫, ⟪_,er:e,er:e:a⟫))

3. Each node linked by an edge to ⟪pcdata⟫ has an attribute added called pcdata whose extent will consist of the instances of ⟪pcdata⟫ with which instances of the node are associated:

   addNode(⟪er:e:pcdata⟫, $\{x \mid \langle \_, x\rangle \in$ ⟪_,xml:e,xml:e:pcdata⟫$\}$)
   addEdge(⟪_,er:e,er:e:pcdata⟫, ⟪_,xml:e,xml:e:pcdata⟫)
   addConstraint(copyCard(⟪_,xml:e,xml:e:pcdata⟫, ⟪_,er:e,er:e:pcdata⟫))

4. Each nesting edge between two XML elements generates an edge between the two corresponding ER entity classes representing a relationship between them:

   addEdge(⟪_,er:e,er:e_s⟫, ⟪_,xml:e,xml:e_s⟫)

[6]Here, the function copyCard() translates the constraint on an XML edge to the same constraint on an ER edge.

For list-based nestings, each instance of the XML nesting edge will be linked to the order node. This information can be represented as an attribute order of the new ER relationship:

addEdge(⟨⟨_,⟨⟨_,er:e,er:e_s⟩⟩,order⟩⟩, ⟨⟨_,⟨⟨_,xml:e,xml:e_s⟩⟩,order⟩⟩)

Any constraint on the XML nesting edge should also be copied over onto the new ER relationship.

5. We also need to handle the association of nodes and edges in the HDM to the ⟨⟨id⟩⟩ node:

   - Each node ⟨⟨xml:e⟩⟩ linked to ⟨⟨id⟩⟩ where the cardinality constraint is such that the edge cannot be multivalued (*i.e.* the link represents an ID attribute) generates a new ER attribute a whose extent consists of those instances of ⟨⟨id⟩⟩ that are linked to the corresponding entity class:

     addNode(⟨⟨er:e:a⟩⟩, $\{\langle x\rangle \mid \langle \_, x\rangle \in$ ⟨⟨xml:a, xml:e, id⟩⟩$\}$)
     addEdge(⟨⟨_,er:e,er:e:a⟩⟩, ⟨⟨xml:a, xml:e,id⟩⟩)
     addConstraint(makeCard(⟨⟨_,er:e,er:e:a⟩⟩,1..1,0..1))

   - Each node ⟨⟨xml:e⟩⟩ linked to ⟨⟨id⟩⟩ where the cardinality constraint is such that the edge can be multivalued (*i.e.* the link represents an IDREF or IDREFS attribute) generates an ER relationship between the ER entity e and the "parent" ER entity e_p with the same ID attribute:

     addEdge(⟨⟨_,er:e,er:e_p⟩⟩, $\{\langle x, z\rangle \mid \langle x, w\rangle \in$ ⟨⟨xml:a, xml:e, id⟩⟩ $\wedge \langle z, w\rangle \in$ ⟨⟨xml:a, xml:e_p, id⟩⟩$\}$)
     addConstraint(makeCard(⟨⟨_,er:e,er:e_p⟩⟩,1..1,0..N))

6. As a result of the above add transformations, all the XML constructs are rendered semantically redundant and can finally be progressively removed from the schema by applying a sequence of del transformations.

The result of applying the above rules to the four HDM schemas representing the XML documents of Figure 2 is shown in Figure 6. We observe that each differs to a greater or lesser extent from the ER schema of Figure 1 which we assumed was used to generate the original XML documents. We now study how to derive the original ER schema.

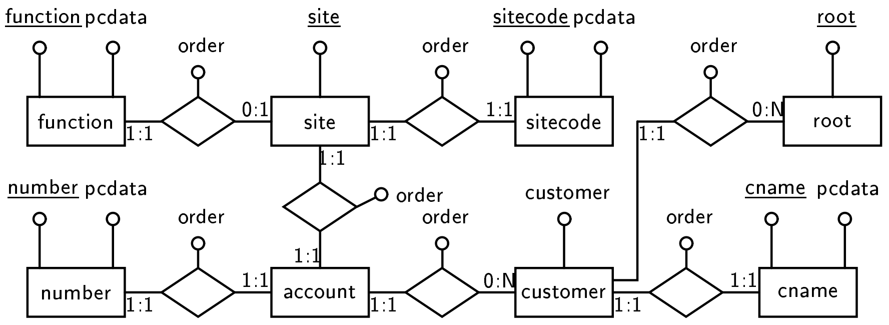
4.2 Manual refinement of canonical ER schemas

Comparing Figures 6(a) and (b) with Figure 1, we notice the following characteristics of using XML to represent ER data, and hence possible refinements that may be made to the canonical ER schemas automatically generated by the method described in the previous subsection:

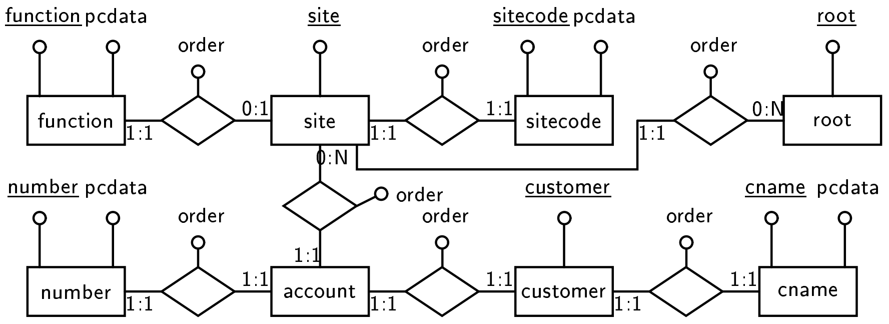
1. It is often the case that the ordering of elements in XML is not semantically significant, in which case the order attribute in the ER schema can be ignored. For example, if the order of site and number within account is not significant in Figure 2(a), then the order attribute in Figure 6(a) can be removed from the relationship between account and number, and from that between account and site, as follows:

   contractAttribute$_{\text{er}}$(⟨⟨⟨⟨_,account,number⟩⟩,order⟩⟩)
   contractAttribute$_{\text{er}}$(⟨⟨⟨⟨_,account,site⟩⟩,order⟩⟩)

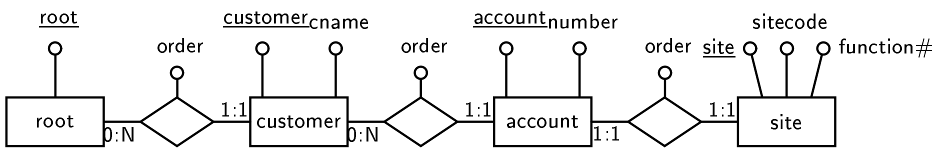
   In contrast if, as discussed in the introduction, the order of accounts within customers was significant, then the order attribute must remain on ⟨⟨_,customer,account⟩⟩.



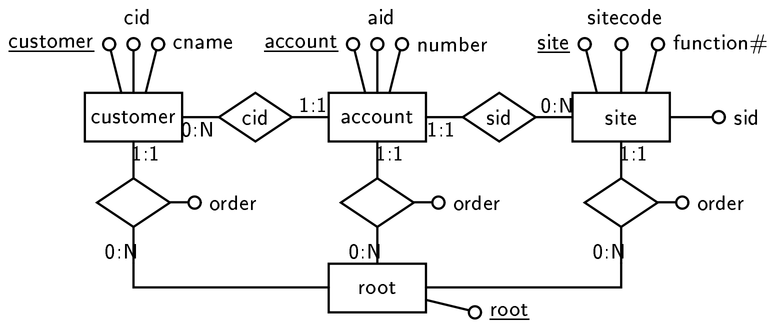
(a)



(b)



(c)



(d)

Figure 6: Canonical ER representations of the XML documents (a)-(d)

2. Using XML elements to represent attributes in the original ER schema has resulted in entity classes being created in the new ER schema.

   For both Figures 6(a) and (b) we see that function, number, cname and sitecode all have this property. The reason that this has occurred is because XML documents contain additional information regarding the position of constructs within the document, which is reflected by the order information being stored in the corresponding ER edges and the generation of a unique identifier for each XML element. If this ordering information need not be preserved in the ER schema (*i.e.* rule (1) above has been applied), these entity classes can be transformed into attributes, using a standard pattern of transformations which we give below for the case of the sitecode entity class:

```
addAttribute_er(⟪site,sitecode⟫, {⟨x, z⟩ | ⟨x, y⟩ ∈ ⟪_,site,sitecode⟫ ∧
        ⟨y, z⟩ ∈ ⟪sitecode,pcdata⟫}, copyCard(⟪_,site,sitecode⟫,⟪site,sitecode⟫))
contractAttribute_er(⟪sitecode,pcdata⟫)
contractAttribute_er(⟪sitecode,sitecode⟫)
contractRelationship_er(⟪_,site,sitecode⟫)
contractEntity_er(⟪sitecode⟫)
```

The result of applying these transformations to function, number, cname and sitecode entities in Figure 6(a) results in the ER schema shown in Figure 6(c) *i.e.* the ER schema generated from the XML document of Figure 2(c).

3. The order in which elements are nested within each other in an XML document effects the cardinality of the relationships between the corresponding ER entity classes.

   For example, in Figure 6(a) sites are repeated for each customer, and thus each site entity is associated with only one account (there will be two sites with sitecode 32 in our example, each with one account). This can be corrected by changing the key of site from the automatically generated, position-related, key to be sitecode instead. This involves renaming the site entity to site', creating a new site entity with the new sitecode key, copying the attributes and relationships across from site' to site, and finally removing site':

```
renameEntity_er(⟪site⟫,⟪site'⟫)
addEntity_er(⟪site⟫,{x | ⟨_, x⟩ ∈ ⟪site',sitecode⟫})
addAttribute_er(⟪site,sitecode⟫,{⟨x,x⟩ | ⟨x⟩ ∈ ⟪site',sitecode⟫},1..1,1..1)
addAttribute_er(⟪site,function⟫,
        {⟨x,y⟩ | ⟨z,x⟩ ∈ ⟪site',sitecode⟫ ∧ ⟨z,y⟩ ∈ ⟪site',function⟫},
        copyCard(⟪site',function⟫,⟪site,function⟫))
addRelationship_er(⟪_,site,account⟫,
        {⟨x,y⟩ | ⟨z,x⟩ ∈ ⟪site',sitecode⟫ ∧ ⟨z,_,y⟩ ∈ ⟪_,site',account⟫},
        copyCard(⟪_,site',account⟫,⟪_,site,account⟫))
contractAttribute_er(⟪site',function⟫)
contractAttribute_er(⟪site',sitecode⟫)
contractAttribute_er(⟪site',site⟫)
contractRelationship_er(⟪_,site',account⟫)
contractEntity_er(⟪site'⟫)
```

As already discussed, Figure 6(c) is an intermediate stage of the transformations applied to Figure 6(a) and only changing the key of entity classes from the automatically generated position-related key to another attribute remains to be done.

Finally, the ‘flat’ representation of information in the XML document of Figure 2(d) results in the ER schema of Figure 6(d). This is the closest to Figure 1, with both the same attributes present and the same cardinality constraints. If they are not required for applications, we can simply drop the id nodes using contractAttribute$_{\mathsf{er}}$(⟪site,id⟫). We can also apply the key transformation rules to change the key from the position-related key to another attribute.

In summary, we have shown in this section how four XML documents $d_1, \ldots, d_4$ can be automatically transformed into ER schemas $er_1, \ldots, er_4$ which may then be manually transformed into a single ER schema er. To transform an XML document d_i to another XML document d_j the forward transformation $d_i \rightarrow er_i \rightarrow er$ can be applied, followed by the reverse transformation $er \rightarrow er_j \rightarrow d_j$.

5 Related Work

XML has received much attention as a language for the exchange of information on the Internet, and there has been much work on translation between XML and structured data models and between different XML formats. [1] describes the whole area and the issues involved

Considering first the issue of translating between different XML document formats, a common approach is to use one of the many proposed XML query languages to build a new XML document as a view of another XML document. A survey of five XML languages is presented in [5]. XSL [8] is perhaps the most established of these languages, being used in several tools concerned with the presentation of XML into HTML. In common with other approaches to representing XML or semi-structured data, *e.g.* [19, 3, 2, 4, 21, 9, 1], we use a graph-based data model in which to represent XML data (the HDM) which supports the notion of unique identifiers for XML elements. One difference with our approach is how the ordering information found in XML documents is represented. Our use of the order node in the HDM graph allows list semantics to be preserved with the data if desired. On the other hand, if only set semantics for the XML data are needed the links to order can be ignored and a set-based representation results. This is possible because of the support of *nested* edges in the HDM. These arose naturally in [20] in order to model attributes of relationships in the ER model. Generally, the provenance of the HDM as a common data model for structured as opposed to semi-structured or XML data gives it a rather different flavour to data models which have an XML or semi-structured data provenance *e.g.* the presence of constraints as well as structure in an HDM schema, and our use of nodal, liking and nodal-linking constructs to represent XML documents in Section 3.1 as opposed to just nodes and edges.

Considering the issue of translating between structured data models and XML, this has also received considerable attention in the literature. YAT [7] and SilkRoute [11] allow XML documents to be materialised from relational databases. Quilt [11] both translates between XML document formats and can materialise relational data in XML form. For the reverse process of translating XML into relational form, [12] represents all XML data in one table and studies the performance of using relational databases for querying this data. [22] considers the translation in both directions between XML and the relational model, and commercial tools with more limited capabilities exist which also provide this functionality *e.g.* Oracle8i. [2] considers the translation between SGML and OO data models, using correspondence rules between constructs expressed in the different data models. [18, 4] also consider the translation between SGML and OO data models, in this case via virtual graphs.

The approach that we have proposed here is a general method for translating between structured and XML representations of data and is not tied to any specific structured data model. Through the use of the HDM and our previous work on providing a methodology for using the HDM to represent any of the common structured data models [16], we have provided a transformation and integration route between XML and any structured data model. The specification of XML in the HDM has allowed the set of primitive transformations on XML to be automatically derived in terms of sequences of primitive transformations on the HDM. Both high and low-level transformations are automatically reversible in our framework because they require the specification of a constructing/restoring query for add/del transformations. These two-way transformation pathways between schemas can be used to auomatically translate data, queries and updates in either direction. The work on schema translation and matching [2, 18, 4] could be utilised to enhance our framework by automatically or semi-automatically deriving the constructing/restoring query in add/del transformations where possible.

6 Concluding Remarks

We have developed our previous work which supports the integration of structured data sources to also handle XML documents. As such, we have provided a framework which allows the free movement of data and queries between structured and XML data sources. We have demonstrated how XML documents can be automatically transformed into ER schemas, and have discussed how to further transform such schemas so that they are semantically closer to the original source database schema that the XML documents may have been generated from. This restructuring is based on the application of well-understood schema transformation rules that might be used in any database integration exercise.

In [15] we developed a second distinguishing feature of our framework, namely that our schema transformations are automatically reversible. Thus, schema transformations set up a two-way transformation pathway between pairs of schemas. These pathways can be used to automatically translate data, queries and updates in either direction between two semantically equivalent or overlapping schemas, for example between a global schema and a component schema.

The process of restructuring the ER schema would be a more complex task for data sources where the information has never been held in a structured form *i.e.* 'really' semi-structured data. In [23] techniques are presented for discovering structured associations from such data, and we are currently studying how our framework can be adapted to use such techniques.

In a longer version of this paper [17] we show how to represent in the HDM the additional information available in an XML DTD, if present. We are also studying how the approach proposed in this paper may be adapted to use XML Schema definitions [10] in place of DTDs, which will enable some of the restructuring rules for ER models to be automatically inferred.

Acknowledgements

This work has been supported by EPSRC grants GR/N 35915 and GR/N 38107, entitled "Automatic Generation of Mediator Tools for Heterogeneous Database Integration"

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web.* Morgan-Kaufmann, 2000.

[2] S. Abiteboul, S. Cluet, and T. Milo. Correspondence and translation for heterogeneous data. In *Proceedings of ICDT'97*, 1997.

[3] S. Abiteboul, *et al.* The Lorel query language for semistructured data. *Journal on Digital Libraries*, 1(1), 1997.

[4] C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *Proceedings of ICDT'99*, 1999.

[5] A. Bonifati and S. Ceri. Comparative analysis of five XML query languages. *SIGMOD RECORD*, 29(1):68–79, 2000.

[6] T. Bray, J. Paoli, and C.M. Sperberg-McQueen. Extensible markup language (XML) 1.0. Technical report, W3C, February 1998.

[7] V. Christophides, S. Cluet, and J. Siméon. On wrapping query languages and efficient XML integration. *SIGMOD RECORD*, 29(2):141–152, 2000.

[8] J. Clark. XSL transformations (XSLT specification). Technical report, W3C, 1999.

[9] A. Deutsch, M. Fernández, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proceedings of 8th International World Wide Web Conference*, 1999.

[10] D.C. Fallside. XML schema part 0: Primer; W3C working draft. Technical report, W3C, April 2000.

[11] M. Fernández, W-C. Tan, and D. Suciu. SilkRoute: Trading between relations and XML. In *Proceedings of 9th International World Wide Web Conference*, 2000.

[12] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. *Bulletin of the Technical Committee on Data Engineering*, 22(3):27–34, September 1999.

[13] P.J. McBrien and A. Poulovassilis. A formal framework for ER schema transformation. In *Proceedings of ER'97*, volume 1331 of *LNCS*, pages 408–421, 1997.

[14] P.J. McBrien and A. Poulovassilis. A formalisation of semantic schema integration. *Information Systems*, 23(5):307–334, 1998.

[15] P.J. McBrien and A. Poulovassilis. Automatic migration and wrapping of database applications — a schema transformation approach. In *Proceedings of ER99*, volume 1728 of *LNCS*, pages 96–113. Springer-Verlag, 1999.

[16] P.J. McBrien and A. Poulovassilis. A uniform approach to inter-model transformations. In *Advanced Information Systems Engineering, 11th International Conference CAiSE'99*, volume 1626 of *LNCS*, pages 333–348. Springer-Verlag, 1999.

[17] P.J. McBrien and A. Poulovassilis. A semantic approach to integrating XML and structured data sources. Technical Report 30/11/2000, Birkbeck College and Imperial College, London, November 2000.

[18] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *Proceedings of VLDB'98*, 1998.

[19] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of ICDE'95*, 1995.

[20] A. Poulovassilis and P.J. McBrien. A general formal framework for schema transformation. *Data and Knowledge Engineering*, 28(1):47–71, 1998.

[21] R.Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *Proceedings of WebDB*, 1999.

[22] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and objectives. In *Proceedings of the 25th VLDB Conference*, pages 302–314, 1999.

[23] K. Wang and H. Liu. Discovering structural association of semistructured data. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):353–371, 2000.