Foundational Data Modeling and Schema Transformations for XML Data Engineering

Reema Al-Kamha[1], David W. Embley[2], and Stephen W. Liddle[3]

[1] Informatics Department, Damascus University, Syria
[2] Department of Computer Science, Brigham Young University, USA
[3] Information Systems Department, Brigham Young University, USA

Abstract. As XML data storage and interchange become ubiquitous, analysts and data engineers increasingly need tools to model their data and map it to XML schemas and to reverse engineer XML documents and schemas in support of evolution and integration activities. For effective data management, model transformations require guarantees of properties of interest including guarantees of information and constraint-preservation, redundancy-free and compactness guarantees, and assurances about readability and maintainability. In this paper, we make foundational observations about XML data management, including conceptual modeling for XML data, transformations to and from XML Schema and XML data models, and transformation guarantees concerning properties of interest, and we provide resolutions for conceptual mismatches between XML data management and more traditional data management. Our implemented prototype tools show that these observations and insights can provide a strong foundation for XML data engineering.

1 Introduction

Because XML has become a standard for data representation, there is a need for a simple conceptual model for XML-based data engineering. But this is not enough—engineers also need a suite of design and development tools to map conceptual designs into implementable designs and to reverse-engineer legacy implementations to conceptual designs. In addition to facilitating these activities, the tools should guarantee certain desirable properties about generated implementations and should warn developers if such properties do not hold.

In building a suite of XML design and development tools, we face several interesting challenges. (1) Creating a conceptual model is a delicate balance between providing enough but not too many high-level conceptualizations without introducing low-level, implementation detail; making the model formal but easily understandable; and having a notation that is easily understood by developers and customers alike. (2) Once a conceptual model exists, the challenge becomes defining equivalence transformations to and from XML Schema—a nontrivial task because of the large conceptual mismatch. (3) Beyond just having transformations, XML data engineering demands certain guarantees. As a minimum, the translations must preserve information content and, to the extent possible,
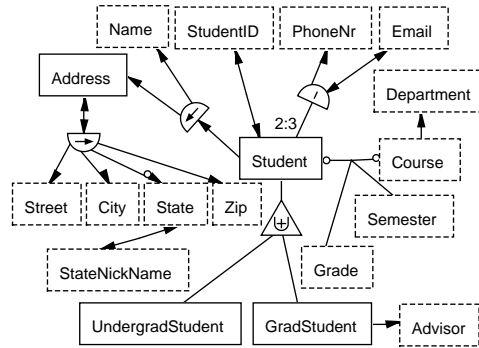
Fig. 1. Given C-XML Model Instance.

preserve constraints. These guarantees should also enable developers to guarantee that forward translations yield storage structures that are redundancy-free and thus free of update anomalies, and that reverse translations yield faithful and understandable conceptual models.

In a keynote address [4] Carey challenged the conceptual-modeling community to develop conceptual models usable in XML data engineering. Several researchers have contributed to making this challenge a reality. Many have attempted to create or define characteristics for conceptual models for XML (e.g., [5, 6, 10, 11, 12, 14, 15, 16]), but all have fallen short of capturing some interesting features of XML Schema. Some have attempted to provide transformations or design guidelines based on standard conceptual models or on XML-augmented conceptual models (e.g., [5, 10, 11]), but few address design properties such as being redundancy free or address them in a way to provide mapping guarantees, and none provide transformations for XML reverse-engineering.

In this paper, we build on our earlier work [2, 3, 8] and describe our implemented algorithms for conversions between a generic conceptual model and XML Schema. Based on these implementations, we explain how to meet the challenges of creating a suite of design and development tools for XML. In particular, we make the following contributions: (1) We argue for a few augmentations to traditional conceptual models to accommodate XML (Section 2). (2) We provide implemented equivalence transformations between XML-augmented conceptual models and XML Schema (Sections 3 and 4). (3) We show how to guarantee properties of interest in XML design and development (Sections 3 and 4).

2 C-XML: Conceptual XML

We show here how to extend traditional conceptual models for XML-based data engineering. In our prototype implementation, we have extended the conceptual modeling language OSM [7] for use with XML, resulting in *C-XML* (*Conceptual XML*). Figure 1 shows an example of a graphical rendition of a particular C-XML model instance for a small part of a student database.

All who have addressed the issue of creating a conceptual model for XML Schema argue for augmenting conceptual models with XML-like *sequence* and *choice* features. Although we disagree with previously-suggested ways of including *sequence* and *choice* features, we agree that both should be included. Some may argue that *sequence* and *choice* constitute low-level, implementation detail. We, and most others who have faced this problem, disagree. Order is a natural, high-level conceptualization among some entities (e.g. to represent component parts of addresses and start-end values for chromosome sequences), and alternatives are natural for others (e.g. to represent various forms of international addresses). We do, however, recommend only appropriate and conceptual use of *sequence* and *choice*—not the inappropriate and artificial use often seen in XML Schema because of its lack of modeling alternatives.

Fundamentally, a C-XML model instance is a hypergraph whose nodes are *object sets* and whose edges are *relationship sets*, which are often binary, but may be *n*-ary ($n > 2$). Beyond a generic hypergraph, C-XML provides notation to declare various *constraints* over object and relationship sets.

Rendered graphically, object sets are boxes, relationship sets are lines, and constraints are decorations. A dashed box denotes a *lexical* object set whose object values are printable types, and a solid box denotes a *non-lexical* object set whose object values are object identifiers. In Figure 1, for example, $Name$ and $StudentID$ are lexical object sets, and $Student$ and $Address$ are non-lexical object sets. A *participation constraint* (e.g., *2:3*) indicates how many times an object in a connected relationship may participate in the relationship set. C-XML uses decorations for common constraints: (1) an arrowhead specifies a *functional constraint* and (2) an "o" on a connection designates *optional participation*. A triangle represents a generalization/specialization relationship among object sets, and C-XML allows for these relationships to be constrained (e.g., $\cup$, $+$, and $\uplus$ respectively denote union, mutual-exclusion, and partition constraints).

A bounded half circle with a directional arrow represents a *sequence* structure. Sequenced child object sets connect to the curved side, and the parent object set connects to the flat side. The representation for the *choice* structure is similar, but instead of an arrow a vertical bar indicates *choice*. In Figure 1, for example, each student has an address that is in a one-to-one correspondence with an ordered 4-tuple ($Street$, $City$, $State$, Zip), and each student has two or three ways to be contacted, each of which is either a phone number or an email address. Students may share phone numbers, but email addresses are unique as indicated by the functional relationship from $Email$ to the *choice* structure.

C-XML satisfies the requirements for conceptual modeling for XML presented by others who have studied the problem of creating a conceptual model for XML [14, 15, 16]. These requirements include: a graphical notation, a formal foundation, structural independence, reflection of the mental model, *n*-ary relationship sets, views, logical-level mapping, cardinality for all participants, ordering, allowance for irregular and heterogeneous structure, and document-centric data. The property of having a formal foundation is particularly important, and we thus point out that a C-XML model instance is precisely an abstract represen-

tation for a particular set of predicates and predicate-calculus formulas. Each object set maps to a one-place predicate, and each n-ary relationship set maps to an n-place predicate. Each constraint maps to a closed predicate-calculus formula. A properly populated C-XML model is therefore a reformulation of a model in first-order theory [9].

3 Mapping C-XML to XML Schema

In the translation from C-XML to XML Schema we must consider several challenging issues.

– XML Schema has a hierarchical structure, while a particular conceptual-model instance may have no explicit hierarchical structure. Converting non-hierarchical structure to hierarchical structure presents some interesting challenges especially if we wish to be able to guarantee properties such as making the hierarchical structures as large as possible without introducing redundancy.
– XML Schema often does not mesh well with conceptual-modeling structures. Translations resulting in a valid XML-Schema instance sometimes need extra, unwanted artifacts to satisfy XML Schema's syntactic requirements.
– Because of XML-Schema limitations the translation sometimes cannot capture all the constraints of a C-XML model instance (e.g. some cardinality constraints). To preserve constraints in the translation, we add them as special comments that auxiliary constraint-checking software can read and enforce.
– A conceptual-model instance may contain a variety of conceptualizations: hypergraphs representing interrelated object and relationship sets; generalization/specialization hierarchies with union, mutual-exclusion, and partition constraints; hierarchies of *sequence* and *choice* structures; and mixed textual/conceptual structures. Translating these conceptualizations individually presents some challenges, and translating conceptual-model instances with a mixture of these conceptualizations is harder.

3.1 Build Scheme-Tree Forest

Our translation starts by applying an algorithm to convert a conceptual-model hypergraph to a forest of scheme trees [13], which we refer to as the *HST* algorithm (Hypergraph-to-Scheme-Tree translation algorithm). By observing many-one cardinality constraints, this algorithm finds hierarchical structures, making them as large as possible without introducing potential redundancy in stored XML data instances. Observing mandatory/optional constraints, this algorithm also ensures that all values populating a C-XML model instance are representable in the XML data instance.

   An application of the HST algorithm to the C-XML model instance in Figure 1 generates the forest of scheme trees in Figure 2. The algorithm lets us build a root node in a scheme-tree beginning with any object set. By default it chooses
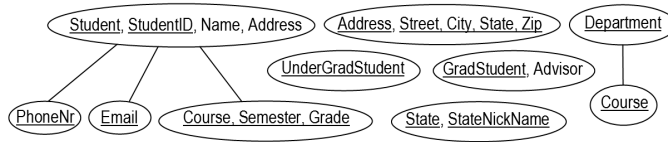
Fig. 2. Scheme Trees.

an object set that is the root of the largest natural hierarchy as determined by the one-many relationship sets. In Figure 1, *State* is one-many with *Address* which is one-many with *Student* and thus would be the default choice. Often, however, it is best to start with the most important node, measured as one with the maximum number of incident edges—*Student* in our example. A starting object set for a node becomes a key for the node. For our example, we start with *Student*, place it in the first root node as Figure 2 shows, and underline it to indicate that it is a key for the node. The HST algorithm then builds the rest of the scheme-tree forest in Figure 2.

The details are beyond the scope of this paper, but in essence, the HST algorithm ensures that each relationship instance appears at most once in any populated scheme tree it generates, which ensures the redundancy-free property, and thus also ensures that storage structures are free from update anomalies. At the same time, from a given starting node, it grows scheme trees as large as possible without introducing potential redundancy. This ensures that the representation does not introduce unnecessary trees, and thus ensures that the representation is compact [13].

3.2 Generate Nested Object Containers

After generating the forest of scheme trees, we construct the XML-Schema instance. Since each scheme-tree node denotes a set of tuples, we generate a container for the set of tuples as well as a container for an individual tuple. Each container requires a name, and although we could use arbitrary names or let the user select names, we attempt to select a reasonable name for each container automatically. Since a key for a set of tuples identifies individual tuples, we choose keys as names for individual tuples and plurals of these names for sets of tuples. For example, Figure 3 shows part of the XML-Schema instance generated for the *Student* scheme tree in Figure 2, and Figure 4 shows part of an XML document that complies with the XML-Schema instance in Figure 3.

Each container has some content. Thus, the container element has *complexType* content. The container element for a node must provide for a set of tuples. We introduce them with a *sequence* structure, even though the *sequence* structure has the extra, perhaps unwanted, constraint of requiring its children to be ordered. The *sequence* structure is the only choice that suffices in this case. The container element for an individual tuple, on the other hand, contains at most one instance of each object set and each child node of the tuple. We thus use the *all* structure for the elements representing the child nodes of the node being built, where possible. It is not possible when *sequence* and *choice* structures

```
<xs:element name="Students">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Student" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:sequence>
              <xs:element name="Name" />
              <xs:element name="Address">
                <xs:complexType>
                  <xs:attribute ref="AddressOID"
                    use="required" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:choice minOccurs="2" maxOccurs="3">
              <xs:element name="Emails">
...
            <xs:attribute name="StudentOID" type="xs:string"
              use="required" />
            <xs:attribute name="StudentID" use="required" />
          </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:key name="StudentOID-Key">
    <xs:selector xpath="./Student" />
    <xs:field xpath="@StudentOID" />
  </xs:key>
  <xs:key name="StudentID-Key">
...
</xs:element>
```

Fig. 3. Generated XML-Schema Instance.

```
<Students>
  <Student StudentOID="Student1" StudentID="0000-1">
    <Name>Alice</Name>
    <Address AddressOID="Address1"/>
    <Emails>
      <Email Email="alice@university.edu/>
      <Email Email="alice@gmail.com/>
    <Emails>
    <Course-Semester-Grades>
      <Course-Semester-Grade
        Course="CS100" Semester="Fall" Grade="A"/>
      ...
    </Course-Semester-Grades>
  </Student>
  <Student StudentOID="Student2" StudentID="0000-2">
    <Name>Bob</Name>
  ...
</Students>
```

Fig. 4. Complying XML Document.

are present, as they are for *Student* in our example, and thus in Figure 3 we introduce the elements of the *Student* tuple with a *sequence* structure.

The tuple itself consists of the key object set (or object sets in case of a compound key)—*Student* in our example; other object sets in the node being constructed—*StudentID*, *Name*, and *Address* in our example; and the container element for child nodes of the node being constructed—*PhoneNrs*, *Emails*, and *Course-Semester-Grades* in our example. We generate the object sets as attributes, when possible, and otherwise as elements, and we generate the

child-node containers as elements. Thus, as Figure 3 shows, *Name* and *Address* are elements since they appear under a *sequence* structure, and *StudentID* and *StudentOID* are attributes.

Each non-lexical object set has an "OID" attribute, which allows us to explicitly represent the entity. If designers prefer not to have explicit representations for non-lexical entities, they can "lexicalize" the C-XML model instance before transforming it to an XML-Schema instance. To lexicalize a non-lexical object set, we replace it with a lexical object set with which it has a one-to-one correspondence or by a group of lexical object sets with which it has a one-to-one correspondence. To retain original names for generating more pleasing tag names for XML documents, we can rename the lexicalized nodes in a special way: *X (of Y)* where *X* is the name of the lexical object set and *Y* is the name of the non-lexical object set. Thus, in our example we would have *StudentID (of Student)*, *StudentID (of UnderGradStudent)*, and *StudentID (of GradStudent)*. With this construction we would retain the names *Students* and *Student* for node and tuple containers, and we would omit the attribute *StudentOID*, retaining *StudentID* as an attribute and as the key.

Each XML-Schema instance must have a single root element. When the number of scheme trees in the generated forest is one, we do not generate a root element because the container element for the set of tuples for the root node in that scheme tree can serve as the root element. When the number of scheme trees in the generated forest is more than one, we generate a root element, call it *Root*, and nest the elements beneath it that represent the sets of tuples for each generated scheme tree.

3.3 Add Constraints

The structure of the generated scheme trees plus the optional constraints of the input C-XML model instance dictate the cardinality constraints. Every XML-Schema element declaration specifies its cardinality with respect to its parent as a *minOccurs* value and a *maxOccurs* value, and every XML-Schema attribute declaration specifies whether it is "required" or "optional" in its *use* attribute. Since there is exactly one instance of the container element for a set of tuples for a node, the assigned values for *minOccurs* and *maxOccurs* are both *1*, the default values. A set of individual tuples may contain one or more tuples. Thus, for a container for individual tuples, the *minOccurs* value is *1* and *maxOccurs* is *unbounded*. The *Student* element in Figure 3, for example, shows these cardinalities. The value for *use* is *required* unless the conceptual model constrains the *use* to be *optional*. In Figure 3, for example, the *use* for the attribute for *StudentID* is *required*.

We observe that since XML Schema has a hierarchical structure, we can only capture participation constraints in the conceptual model instance for the parent element. By default, the nesting structure in an XML-Schema instance makes the minimum participation constraint for a child element be *1* and the maximum constraint be *unbounded*. XML Schema provides no way to capture any constraint other than this default constraint. We can, however, capture constraints

```
<xs:keyref name="UnderGradStudentOID-Keyref"
    refer="StudentOID-Key">
  <xs:selector
    xpath="./UnderGradStudents/UnderGradStudent" />
  <xs:field xpath="@UnderGradStudentOID" />
</xs:keyref>
<xs:keyref name="GradStudentOID-Keyref"
    refer="StudentOID-Key">
  <xs:selector
    xpath="./GradStudents/GradStudent" />
  <xs:field xpath="@GradStudentOID" />
</xs:keyref>
```

Fig. 5. Generated Subset Declarations.

that differ from the default in special *pragma* comments. We prefix pragma comments with *C-XML* so that we can know to process them if we wish to enforce the constraint or if we wish to restore the original C-XML model instance from the XML-Schema instance. So that we can know what constraint to enforce or restore, we write the constraint formally using predicate-calculus syntax. (All constraints in C-XML have equivalent predicate-calculus expressions [7].) For example, to declare that the participation of *Course* in the *n*-ary relationship set in Figure 1 among *Student*, *Course*, *Semester*, and *Grade* is optional, we write the comment

$$forall\ x(Course(x) => exists\ [0:^{*}] <y,\ z,\ w>$$
$$(Course(x)Student(y)Semester(z)Grade(w)))$$

which establishes the constraint that each element x in *Course* may have zero or more tuples $<y,\ z,\ w>$ in the relationship set among *Course*, *Student*, *Semester*, and *Grade*.

For each key within a node we determine the uniqueness constraints and express them in the generated XML-Schema instance. Every key for each node is unique within the container element for that node. Keys within child nodes, however, are only known to be unique within their parent node. *Course-Semester-Grade* tuples, for example, are only unique for each *Student*. Figure 3 shows the generated key constraint for *StudentOID*. Its *selector* declaration is *./Student* since its declaration is within *Students*, and its *field* declaration is @*StudentOID* since *StudentOID* is an attribute.

Generalization/specialization is not a native construct in XML Schema. Nevertheless, with judicious use of XML-Schema's *keyref* constraint, we can make XML Schema enforce basic generalization/specialization constraints. The main idea in generalization/specialization is that a generalization object set is a superset of each of its specialization object sets. Using *keyref* enables us to specify that the set of values in a specialization element is a subset of the set of values in a generalization element. Figure 5 shows these basic subset declarations for *UnderGradStudent* and *GradStudent*. Observe that these declarations force the set of *UnderGradStudentOID*s and the set of *GradStudentOID*s to be a subset of the set of *StudentOID*s.

When a generalization/specialization hierarchy has a *union*, a *mutual-exclusion*, or an *intersection* constraint, we generate a special pragma comment

to capture the constraint. Since our example in Figure 1 has a partition constraint, we generate both a *union* constraint and a *mutual-exclusion constraint* as follows.

> forall x(StudentOID(x) => GradStudentOID(x)
>   or UnderGradStudentOID(x)
> forall x(GradStudentOID(x) =>
>   not UnderGradStudentOID(x))

3.4 Property Guarantees

Having explained the basic idea of our translation of a C-XML model instance to an XML-Schema instance, we now show that the translation preserves both information and constraints. We also show that the translation is redundancy-free and is reasonably compact.

Definition 1 *Let T be a transformation from a model instance M to a model instance M' that not only derives M' from M but also derives a populated model instance M'_p from a populated model instance M_p. T preserves information if for any properly populated model instance M_p, there exists an inverse transformation T^{-1} that maps $T(M_p)$ to M_p such that $T^{-1}(T(M_p)) = M_p$.*

Theorem 1 *The transformation from C-XML to XML Schema preserves information. (We have omitted all proofs due to space constraints.)*

Definition 2 *Let C be the constraints of a model instance M, and let C' be the constraints of a model instance M' obtained from M by a transformation T. T preserves constraints if $C' \implies C$.*

Theorem 2 *Allowing for pragma constraints, the transformation from C-XML to XML Schema preserves constraints.*

Definition 3 *A value or object instance i in a populated model instance M_I is* redundant *with respect to a set of constraints C if i is uniquely determinable from C and the values and object instances in M_I other than i.*

If, for example, we store the *StateNickName* with every student's address, then for any two student instances S_1 and S_2 whose address is in the same state, we can uniquely determine the state nickname of S_1 from the information in S_2's address and from the constraint that *StateNickName* and *State* are in a one-to-one correspondence.

Theorem 3 *Let C be a canonical C-XML model instance whose declared functional constraints are its functional edges and whose declared multivalued constraints are its non-functional edges. The transformation from C-XML to XML Schema for C yields an XML-Schema instance whose complying XML documents have no redundant value or object instances with respect to the functional and multivalued constraints declared in C.*

Observe that we have not claimed redundancy-free with respect to the inclusion dependencies in a generalization/specialization hierarchy. Indeed, because of the way we store and reference object identifiers or values in generalization/specialization hierarchies, every value or object in a generalization is stored redundantly. XML system developers can avoid this redundancy by collapsing all generalization/specialization hierarchies to their roots before generating XML-Schema instances. In Figure 1 collapsing the generalization/specialization hierarchy to the root consists of discarding the object sets *UnderGradStudent* and *GradStudent* and attaching *Advisor* optionally through a functional relationship set to *Student*. Implicitly, those students who have advisors are graduate students and those who do not have advisors are undergraduate students. The disadvantage of this approach is that the generalization/specialization constraints (the partition in our example) are lost as are all the specialization names.

As it turns out, the HST algorithm does not always produce the fewest number of scheme trees both because a developer may choose to start the algorithm at a node that does not initially carve out the largest natural hierarchy and because of some pathological cases that occur only when the hypergraph is cyclic. Normally, however, the HST algorithm yields the fewest number of scheme trees and thus yields a compact representation.

Although not provable, we believe that XML-Schema instances obtained by transforming C-XML model instances are reasonably readable and thus maintainable. As Figure 4 indicates, they provide reasonable tags for both sets of objects (e.g. *Students*) and individual objects (e.g. *Student*), and they appropriately nest associated information nicely inside of the scope of objects (e.g. *Address*, *Email*, and *Course-Semester-Grades* inside the scope of student).

4 Mapping XML Schema to C-XML

The basic translation strategies for mapping XML Schema to C-XML are straightforward, although some parts of the translation require some sophisticated manipulation. In the translation, elements and attributes become object sets. Elements that have simple types become lexical object sets, while elements that have complex types become non-lexical object sets. Attributes become lexical object sets since they always have a simple type. Built-in data types and simple data types for an element or an attribute in XML Schema are specified in the data frame associated with the object set representing the element or the attribute. XML parent-child connections among elements and XML element-attribute connections both become binary relationship sets in C-XML. The constraints *minOccurs* and *maxOccurs* translate directly to participation constraints in C-XML.

Unfortunately, not everything is straightforward. Translations for keys, extension, restriction, substitution groups, and mixed content are all quite interesting. The translation also involves a myriad of detail extending to over 40 pages in [1]. One general observation about the translation is that it is often difficult and sometimes impossible to express modeling constraints of interest.

– XML Schema provides no good way to nest elements that are not natural sequences of, not natural alternatives of, and not functionally dependent on a parent. Although designers usually declare sequences for these nestings, it is difficult to tell whether a *sequence* is merely an artifact, required by XML Schema, or whether it is a conceptual sequence, as address elements would be. Since a translation algorithm does not know whether a sequence is meaningful, however, we must faithfully generate all declared sequences. We can, and do, handle the special case of a *sequence* of one nested element.
– Generalization/specialization, especially large hierarchies with complex constraints, is difficult and sometimes impossible to specify [3].
– Some functional constraints are difficult or impossible to capture.
– Lexical elements cannot be parent elements. In such cases, we must introduce another object set.

Because some unwanted artifacts appear and because some constraints are either difficult or impossible to declare, one interesting possibility is to reverse-engineer an XML Schema instance to an C-XML instance, add the missing constraints, remove unwanted artifacts, and then regenerate a more desirable XML Schema instance.

Theorem 4 *The transformation from XML Schema to C-XML preserves information.*

Theorem 5 *The transformation from XML Schema to C-XML preserves constraints.*

A main use of the transformation from XML Schema to C-XML is to reverse-engineer an XML Schema instance. We believe that the result is quite usable. We capture each XML concept as an object set and each nested connection among concepts as relationship set, and we also capture all representable constraints. Further, we believe that the representation is appropriately abstract and sufficiently high level in the usual way in which conceptual models are abstract and high level. Thus, we can reasonably claim that this reverse-engineering transformation can aid in understandability and thus maintainability and evolvability.

5 Concluding Remarks

We have implemented automatic conversions between C-XML and XML Schema that preserve information and constraints, that have guaranteed redundancy-free and compactness properties, and that yield reasonably understandable results and thus provide for maintainability and evolvability. Our prototype implementations and these observations and insights provide a solid theoretical foundation for XML data engineering.

References

1. R. Al-Kamha. *Conceptual XML for Systems Analysis*. Phd dissertation, Brigham Young University, Department of Computer Science, June 2007.
2. R. Al-Kamha, D.W. Embley, and S.W. Liddle. Representing generalization/specialization in XML schema. In *Proceedings of the Workshop on Enterprise Modeling and Information Systems Architectures (EMISA'05)*, pages 250–263, Klagenfurt, Austria, October 2005.
3. R. Al-Kamha, D.W. Embley, and S.W. Liddle. Augmenting traditional conceptual models to accommodate XML structural constructs. In *Proceedings of the 26th International Conference on Conceptual Modeling (ER2007)*, Auckland, New Zealand, November 2007. (in press).
4. M. Carey. Enterprise information integration—XML to the rescue! In *Proceedings of the 22nd International Conference on Conceptual Modeling (ER2003)*, page 14, Chicago, Illinois, October 2003. (keynote address).
5. M. Choi, J. Lim, and K. Joo. Developing a unified design methodology based on extended entity-relationship model for XML. In *Proceedings of the International Conference on Computational Science—ICCS 2003*, St. Petersburg, Russia and Melbourne, Australia, June 2003.
6. R. Conrad, Deiter Scheffner, and J.C. Freytag. XML conceptual modeling using UML. In *Proceedings of the Ninteenth International Conference on Conceptual Modeling (ER2000)*, pages 558–571, Salt Lake City, Utah, October 2000.
7. D.W. Embley, B.D. Kurtz, and S.N. Woodfield. *Object-oriented Systems Analysis: A Model-Driven Approach*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.
8. D.W. Embley, S.W. Liddle, and R. Al-Kamha. Enterprise modeling with conceptual XML. In *Proceedings of the 23rd International Conference on Conceptual Modeling (ER2004)*, pages 150–165, Shanghai, China, November 2004.
9. H.B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, Inc., Boston, Massachussets, 1972.
10. L. Feng, E. Chang, and T. Dillon. A semantic network-based design methodology for XML documents. *ACM Transactions on Information Systems*, 20(4):390–421, October 2002.
11. H. Liu, Y. Lu, and Q. Yang. XML conceptual modeling with XUML. In *Proceedings of the 28th International Conference on Software Engineering*, pages 973–976, Shanghai, China, May 2006.
12. M. Mani. Conceptual models for XML. In *Proceedings of the Second International XML Database Symposium (XSym 2004)*, pages 128–142, Toronto, Canada, August 2004.
13. W.Y. Mok and D.W. Embley. Generating compact redundancy-free XML documents from concptual-model hypergraphs. *IEEE Transactions on Knowledge and Data Engineering*, 18(8):1082–1096, August 2006.
14. M. Nečaský. Conceptual modeling for XML: A survey. In *Proceedings of the Annual International Workshop on Databases, Texts, Specifications and Objects (DATESO 2006)*, pages 45–53, Desna, Czech Republic, April 2006.
15. A. Sengupta and E. Wilde. The case for conceptual modeling for XML. Technical report, Wright State University (WSU) and Swiss Federal Institute of Technology (ETH), February 2006.
16. E. Wilde. Towards conceptual modeling for XML. In *Proceedings of the Berliner XML Tage 2005 (BXML2005)*, pages 213–224, Berlin, Germany, September 2005.