2007-06-19

Conceptual XML for Systems Analysis

Reema Al-Kamha
Brigham Young University - Provo

CONCEPTUAL XML FOR SYSTEMS ANALYSIS

by

Reema Al-Kamha

A dissertation submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Brigham Young University

August 2007

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a dissertation submitted by

Reema Al-Kamha

This dissertation has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	David W. Embley, Chair
Date	Stephen W. Liddle
Date	Scott N. Woodfield
Date	Parris K. Egbert
Date	Eric G. Mercer

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Reema Al-Kamha in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

David W. Embley
Chair, Graduate Committee

Accepted for the Department

Parris K. Egbert
Graduate Coordinator

Accepted for the College

Thomas W. Sederberg
Associate Dean
College of Physical and Mathematical Sciences

ABSTRACT

CONCEPTUAL XML FOR SYSTEMS ANALYSIS

Reema Al-Kamha

Department of Computer Science

Doctor of Philosophy

Because XML has become a new standard for data representation, there is a need for a simple conceptual model that works well with XML-based development. In this research we present a conceptual model for XML, called C-XML, which meets this new need of systems analysts who store their data using XML. We describe our implementation of an automatic conversion from XML Schema to C-XML that preserves information and constraints. With this conversion, we can view an XML Schema instance graphically at a higher level of abstraction. We also describe our implementation of an automatic conversion from C-XML to XML Schema. Our conversion preserves information and constraints as long as we count the special C-XML comments that we insert in an XML-Schema instance to capture the constraints in C-XML that are not representable in XML Schema. In connection with defining C-XML and implementing conversions between C-XML and XML Schema, we are also able to make several insightful observations. We point out ways in which C-XML is more expressive than XML Schema, and we make recommendations for extending XML Schema. We also point out ways in which XML Schema is more expressive than

conceptual models, and we make recommendations for augmenting traditional conceptual models to better accommodate XML. The work accomplished in connection with this research establishes the basis for several fundamental activities in system analysis, design, development, and evolution.

ACKNOWLEDGMENTS

My foremost thank goes to my thesis adviser Dr. David W. Embley. Without him, this dissertation would not have been possible. I thank him for his patience and encouragement that carried me on through difficult times, and for his insights and suggestions that helped to shape my research skills. His valuable feedback contributed greatly to this dissertation.

I thank the rest of my dissertation committee members: Dr. Stephen W. Liddle, Dr. Scott N. Woodfield, Dr. Parris K. Egbert, and Dr. Eric G. Mercer. Their valuable feedback helped me to improve the dissertation in many ways.

I want to express my appreciation to Dr. Sandra Rogers for supporting me through my studies.

I would like to thank my parents who always encourage me to continue my studies. I also want to thank my husband, Dr. Ghassan Wazan, my four children, Wael, Yaman, Maya, and Ameer for their support and patience.

I thank all the BYU data-extraction research group members for their help.

Contents

List of Figures

Chapter 1

Introduction

XML has become a standard for data representation, especially for data that is exchanged on the web. XML Schema [44] is used to describe the structure and the content of XML data. Although XML Schema is useful for specifying and validating XML documents, systems analysts who wish to model their data using XML need a simple conceptual model to help improve modeling capabilities for XML-based development. This need for conceptual modeling arises because XML Schema was designed as an interchange language, not a conceptual modeling language. When it is applied to conceptual modeling, XML Schema over exposes analysts to low-level implementation details, and the structure of XML Schema forces a hierarchical view for all data, even when its natural structure is not hierarchical.

Furthermore, XML Schema is represented textually. Since the early 1970's, systems analysts have used graphical versions of conceptual models to aid in understanding and documenting essential characteristics of systems. This capability should be available to systems analysts who store their data using XML as well.

In this research we present *Conceptual-XML* (*C-XML*) which meets these new needs of systems analysts who wish to store their data using XML. C-XML is first and foremost a conceptual model, being fundamentally based on object-set and relationship-set constructs. Further, it has a graphical representation for all its conceptual components, and it has semi-structured textual attachments for each concept to provide low-level data-type details. Additionally, C-XML can represent each component and constraint in XML Schema. Hence, system analysts can use it to model their XML data.

We present our contribution of providing a conceptual model for XML Schema as a series of self-contained papers. Because each chapter is self contained, there is necessarily some amount of overlap in the introductory material of some of the chapters. Each chapter, however, contributes an important component of the whole.

Chapter 2, titled "Enterprise Modeling with Conceptual XML," is a motivational chapter that describes the "big picture" of what we are trying to accomplish. An open challenge is to integrate XML and conceptual modeling in order to satisfy large-scale enterprise needs. Because enterprises typically have many data sources using different assumptions, formats, and schemas, all expressed in—or soon to be expressed in—XML, it is easy to become lost in an avalanche of XML detail. This creates an opportunity for the conceptual modeling community to provide improved abstractions to help manage this detail. In this chapter, we present a vision for C-XML that builds on the established work of the conceptual modeling community over the last several decades to bring improved modeling capabilities to XML-based development. Building on a framework such as C-XML will enable better management of enterprise-scale data and more rapid development of enterprise applications.

Chapter 3, titled "Representing Generalization/Specialization in XML Schema," motivates the problem of representing conceptual *is-a* hierarchies in XML Schema. Generalization/specialization and its constraints are fundamental concepts in system modeling and design, but are difficult to express and enforce with XML Schema. This mismatch leads to unnecessary complexity and uncertainty in XML-based models. In this chapter, we describe how to translate various aspects of generalization/specialization from a conceptual model into XML Schema. We also explore what needs to be added to XML Schema to handle some aspects of this fundamental modeling construct. If XML Schema were to include our proposed constructs, it would be fully capable of faithfully representing generalization/specialization, thus reducing the complexity of the XML models that rely on generalization/specialization.

Chapter 4, titled "Augmenting Traditional Conceptual Models to Accommodate XML Structural Constructs," shows how C-XML raises the conceptual level of

abstraction of XML Schema. Although it is possible to present XML Schema graphically, such representations do not raise the level of abstraction for XML schemata in the same way traditional conceptual models raise the level of abstraction for data schemata. Traditional conceptual models, on the other hand, do not accommodate several XML-Schema structures. Thus, there is a need to enrich traditional conceptual models with new XML-Schema features. After establishing criteria for XML conceptual modeling, we propose an enrichment to represent the XML features missing in traditional models. We argue that our solution can be adapted generally for traditional conceptual models and show how it can be adapted for two popular conceptual models.

Chapter 5, titled "Translating XML Schema to Conceptual XML," explains how to translate any XML Schema instance to C-XML. The textual representation of XML Schema makes it difficult to understand, handle, and do further modification and thus difficult to use for systems modeling and design. Since conceptual data models have proven to be successful for representing data graphically at a higher level of abstraction, if we can represent an XML-Schema instance graphically, at a higher level of abstraction, we can conceptualize the XML-Schema instance. We propose a translation from XML Schema to a conceptual model that preserves information and constraints. Its implementation allows any XML-Schema instance as input and produces a C-XML model instance as output.

Chapter 6, titled "Translating Conceptual XML to XML Schema," proposes a translation from C-XML to an XML Schema. XML Schema is not a good modeling language for analysts because it lacks a two-dimensional visualization of the application data, and it fails to raise the level of abstraction beyond a mundane textual specification. C-XML resolves these issues, but raises another—the translation from C-XML to XML Schema. In this chapter, we propose a translation from a C-XML conceptual-model instance to an XML-Schema instance. Several issues arise in attempting to make the translation fully automatic and to make it capture the data contained within the conceptual model instance and the constraints imposed over this data. These issues include (1) converting nonhierarchical conceptual structures

to XML hierarchal structures, (2) resolving fundamental mismatches in constraint specification between the two languages, (3) translating mixtures of various conceptual structures (hypergraphs, *is-a* hierarchies, choice/sequence hierarchies), and (4) selecting from among alternative target translation possibilities. Our implementation resolves these issues, which allow it to take any valid C-XML model instance as input and produce an XML-Schema instance as output.

In Chapter 7, the conclusion, we summarize our accomplishments and give the status of our implementation. We also discuss some possibilities for future work.

Chapter 2

Enterprise Modeling with Conceptual XML

2.1 Introduction

A challenge [9] for modern enterprise modeling is to produce a simple conceptual model that: (1) works well with XML and XML Schema; (2) abstracts well for conceptual entities and relationships; (3) scales to handle both large data sets and complex object interrelationships; (4) allows for queries and defined views via XQuery; and (5) accommodates heterogeneity.

The conceptual model must work well with XML and XML Schema because XML is rapidly becoming the de facto standard for business data. Because conceptualizations must support both high-level understanding and high-level program construction, the conceptual model must abstract well. Because many of today's huge industrial conglomerations have large, enterprise-size data sets and increasingly complex constraints over their data, the conceptual model must scale up. Because XQuery, like XML, is rapidly becoming the industry standard, the conceptual model must smoothly incorporate both XQuery and XML. Finally, because we can no longer assume that all enterprise data is integrated, the conceptual model must accommodate heterogeneity. Accommodating heterogeneity also supports today's rapid acquisitions and mergers, which require fast-paced solutions to data integration.

We call the answer we offer for this challenge *Conceptual XML* (*C-XML*). C-XML is first and foremost a conceptual model, being fundamentally based on object-set and relationship-set constructs. As a central feature, C-XML supports high-level object-set and relationship-set construction at ever higher levels of abstraction. At

any level of abstraction the object and relationship sets are always first class, which lets us address object and relationship sets uniformly, independent of level of abstraction. These features of C-XML make it abstract well and scale well. Secondly, C-XML is "model-equivalent" [25] with XML Schema, which means that C-XML can represent each component and constraint in XML Schema and vice versa. Because of this correspondence between C-XML and XML Schema, XQuery immediately applies to populated C-XML model instances and thus we can raise the level of abstraction for XQuery by applying it to high-level model instances rather than low-level XML documents. Further, we can define high-level XQuery-based mappings between C-XML model instances over in-house, autonomous databases, and we can declare virtual views over these mappings. Thus, we can accommodate heterogeneity at a higher level of abstraction and provide uniform access to all enterprise data.

Besides enunciating a comprehensive vision for the XML/conceptual-modeling challenge [9], our contributions in this chapter include: (1) mappings to and from C-XML and XML Schema, (2) defined mechanisms for producing and using first-class, high-level, conceptual abstractions, and (3) XQuery view definitions over both standard and federated conceptual-model instances that are themselves conceptual-model equivalent. As a result of these contributions, C-XML and XML Schema can be fully interchangable in their usage over both standard and heterogeneous XML data repositories. This lets us leverage conceptual model abstractions for high-level understanding while retaining all the complex details involved with low-level XML Schema intricacies, view mappings, and integration issues over heterogeneous XML repositories.

We present the details of our contributions as follows. Section 2.2 describes C-XML. Section 2.3 shows that C-XML is "model-equivalent" with XML Schema by providing mappings between the two. Section 2.4 describes C-XML views. We report the status of our implementation and conclude in Section 2.5.

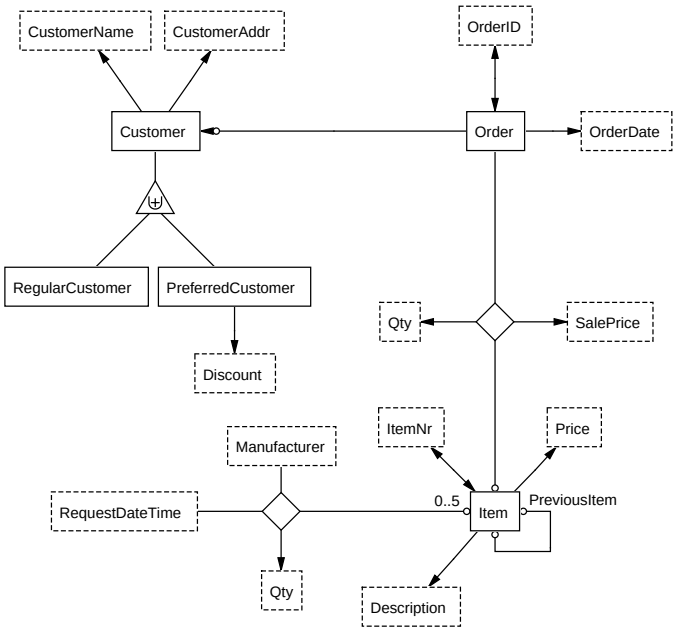


Figure 2.1: Customer/Order C-XML Model Instance.

2.2 C-XML: Conceptual XML

C-XML is a conceptual model consisting of object sets, relationship sets, and constraints over these object and relationship sets. Graphically a C-XML model instance M is an augmented hypergraph whose vertices and edges are respectively the object sets and relationship sets of M, and whose augmentations consist of decorations that represent constraints. Figure 2.1 shows an example.

In the notation, boxes represent *object sets*. An object set with a solid border is a nonlexical object set (e.g. *Customer*). An object set with a dashed border is a lexical object set (e.g. *OrderID*). With each object set we can associate a data frame (as we call it) to provide a rich description of its value set and other properties. A data frame lets us specify, for example, that *OrderDate* is of type *Date* or that *ItemNr* values must satisfy the value pattern "[A-Z]{3}-\d{7}". Lines connecting object sets are *relationship sets*; these lines may be hyper-lines (hyper-edges in hyper-graphs) with diamonds when they have more than two connections to object sets. Optional

or mandatory *participation constraints* respectively specify whether objects in a connected relationship may or must participate in a relationship set (an "o" on a connecting relationship-set line designates *optional* while the absence of an "o" designates *mandatory*). Thus, for example, the C-XML model instance in Figure 2.1 declares that an *Order* must include at least one *Item* but that an *Item* need not be included in any *Order*. Arrowheads on lines specify *functional constraints*. Thus, Figure 2.1 declares that an *Item* has a *Price* and a *Description* and is in a one-to-one correspondence with *ItemNr* and that an *Item* in an *Order* has one *Qty* and one *SalePrice*. In cases when optional and mandatory participation constraints along with functional constraints are insufficient to specify minimum and maximum participation, explicit *min..max* constraints may be specified. Triangles denote *generalization/specialization hierarchies*. We can constrain *is-a* hierarchies by partition ($\uplus$), union ($\cup$), or mutual exclusion ($+$) among specializations. Any object-set/relationship-set connection may have a role, but a role is simply a shorthand for an object set that denotes the subset consisting of the objects that actually participate in the connection.

2.3 Translations between C-XML and XML Schema

Many translations between C-XML and XML Schema are possible. In recent ER conferences, researchers have described varying conceptual-model translations to and/or from XML or XML DTD's or XML-Schema-like specifications. (See, for example, [12, 14, 26].) It is not our purpose here to argue for or against a particular translation. Indeed, we would argue that a variety of translations may be desirable. For any translation, however, we require information and constraint preservation. This ensures that an XML Schema and a conceptual instantiation of an XML Schema as a C-XML model instance correspond and that a system can reflect manipulations of the one in the other.

To make our correspondence exact, we need information- and constraint-preserving translations in both directions. We do not, however, require that translations be inverses of one another—translations that generate members of an equivalence class of XML Schema specifications and C-XML model instances are sufficient. In

Section 2.3.1 we present our C-XML-to-XML-Schema translation, and in Section 2.3.2 we present an XML-Schema-to-C-XML translation. In Section 2.3.3 we formalize the notions of information and constraint preservation and show that the translations we propose preserve information and constraints.

2.3.1 Translation from C-XML to XML Schema

We now describe our process for translating a C-XML model instance C to an XML Schema S_C. We illustrate our translation process with the C-XML model instance of Figure 2.1 translated to the corresponding XML Schema excerpted in Figure 2.2.

Fully automatic translation from C to S_C is not only possible, but can be done with certain guarantees regarding the quality of S_C. Our approach is based on our previous work [21], which for C generates a forest of scheme trees F_C such that (1) F_C has a minimal number of scheme trees, and (2) XML documents conforming to F_C have no redundant data with respect to functional and multivalued constraints of C. For our example in Figure 2.1, the algorithms in [21] will generate the following two nested scheme trees.

$(Customer, CustomerName, CustomerAddr, Discount$
$(Order, OrderID, OrderDate,$
$(Item, SalePrice, Qty)*)*)*$
$(Item, ItemNr, Description, Price,$
$(PreviousItem)*, (Manufacturer, RequestDateTime, Qty)*)*$

Observe that the XML Schema in Figure 2.2 satisfies these nesting specifications. *Item* in the second scheme tree appears as an element on Line 8 with *ItemNr*, *Description*, and *Price* defined as its attributes on Lines 28–30. *PreviousItem* is nested, by itself, underneath *Item*, on Line 18, and *Manufacturer*, *RequestDateTime*, and *Qty* are nested underneath *Item* as a group on Lines 13–15. The XML-Schema notation that accompanies these C-XML object-set names obscures the nesting to some extent, but this additional notation is necessary either to satisfy the syntactic requirements of XML Schema or to allow us to specify the constraints of the C-XML model instance.

```
 1: <?xml version="1.0" encoding="UTF-8"?>
 2: <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
 3:            elementFormDefault="qualified" attributeFormDefault="unqualified">
 4: <xs:element name="Document">
 5:     <xs:complexType>
 6:         <xs:choice minOccurs="0" maxOccurs="unbounded">
 7:             <xs:element ref="Customer"/>
 8:             <xs:element name="Item">
 9:                 <xs:complexType>
10:                     <xs:sequence>
11:                         <xs:element name="ItemMR" minOccurs="0" maxOccurs="5">
12:                             <xs:complexType>
13:                                 <xs:attribute name="Manufacturer" type="xs:string" use="required"/>
14:                                 <xs:attribute name="RequestDateTime" type="xs:date" use="required"/>
15:                                 <xs:attribute name="Qty" type="xs:positiveInteger" use="required"/>
16:                             </xs:complexType>
17:                         </xs:element>
18:                         <xs:element name="PreviousItem" minOccurs="0" maxOccurs="unbounded">
19:                             <xs:complexType>
20:                                 <xs:attribute name="ItemNr" type="xs:positiveInteger" use="required"/>
21:                             </xs:complexType>
22:                             <xs:keyref name="r1" refer="ItemKey">
23:                                 <xs:selector xpath="."/>
24:                                 <xs:field xpath="@ItemNr"/>
25:                             </xs:keyref>
26:                         </xs:element>
27:                     </xs:sequence>
28:                     <xs:attribute name="ItemNr" type="xs:positiveInteger" use="required"/>
29:                     <xs:attribute name="Description" type="xs:string" use="required"/>
30:                     <xs:attribute name="Price" type="xs:decimal" use="required"/>
31:                 </xs:complexType>
32:             </xs:element>
33:         </xs:choice>
34:     </xs:complexType>
35:     <xs:key name="OrderKey">
36:         <xs:selector xpath=".//Order"/>
37:         <xs:field xpath="@OrderID"/>
38:     </xs:key>
39:     <xs:key name="ItemKey">
40:         <xs:selector xpath=".//Item"/>
41:         <xs:field xpath="@ItemNr"/>
42:     </xs:key>
43: </xs:element>
44: <xs:element name="Customer" abstract="true"/>
45: <xs:element name="PreferredCustomer" substitutionGroup="Customer">
46:     <xs:complexType>
47:         <xs:group ref="CustomerDetails"/>
48:         <xs:attribute name="Discount" type="xs:string" use="required"/>
49:     </xs:complexType>
50: </xs:element>
51: <xs:element name="RegularCustomer" substitutionGroup="Customer">
52:     <xs:complexType>
53:         <xs:group ref="CustomerDetails"/>
54:     </xs:complexType>
55: </xs:element>
56: <xs:group name="CustomerDetails">
57:     <xs:sequence>
58:         <xs:element name="CustomerName" type="xs:string"/>
59:         <xs:element name="CustomerAddr" type="xs:string"/>
60:         <xs:element name="Order" minOccurs="0" maxOccurs="unbounded">
61:             <xs:complexType>
62:                 <xs:sequence>
63:                     <xs:element name="OrderItem" minOccurs="0" maxOccurs="unbounded">
64:                         <xs:complexType>
65:                             <xs:attribute name="Qty" type="xs:positiveInteger" use="required"/>
66:                             <xs:attribute name="SalePrice" type="xs:decimal" use="required"/>
67:                             <xs:attribute name="ItemNr" type="xs:positiveInteger" use="required"/>
68:                         </xs:complexType>
69:                         <xs:keyref name="r3" refer="ItemKey">
70:                             <xs:selector xpath="."/>
71:                             <xs:field xpath="@ItemNr"/>
72:                         </xs:keyref>
73:                     </xs:element>
74:                 </xs:sequence>
75:                 <xs:attribute name="OrderID" type="xs:positiveInteger" use="required"/>
76:                 <xs:attribute name="OrderDate" type="xs:date" use="required"/>
77:             </xs:complexType>
78:         </xs:element>
79:     </xs:sequence>
80: </xs:group>
81: </xs:schema>
```

Figure 2.2: XML Schema Excerpt for the C-XML Model Instance in Figure 2.1

As we continue, recall first that each C-XML object set has an associated data frame that contains specifications such as type declarations, value restrictions, and any other annotations needed to specify information about objects in the object set. For our work here, we let the kind of information that appears in a data frame correspond exactly to the kind of data constraint information specifiable in XML Schema. One example we point out explicitly is order information, which is usually absent in conceptual models, but is unavoidably present in XML. Thus, if we wish to say that *CustomerName* precedes *CustomerAddr*, we add the annotation "$\prec CustomerAddr$" to the *CustomerName* data frame and add the annotation "$\succ CustomerName$" to the *CustomerAddr* data frame. In our discussion, we assume that these annotations are in the data frames that accompany the object sets *CustomerName* and *CustomerAddr* in Figure 2.1.

Our conversion algorithm preserves all annotations found in C-XML data frames. This is where we obtain all the type specifications in Figure 2.2. We capture the order specification, $CustomerName \prec CustomerAddr$, by making *CustomerName* and *CustomerAddr* elements (rather than attributes) and placing them, in order, in their proper place in the nesting—for our example in Lines 58 and 59 nested under *CustomerDetails*.

In the conversion from C-XML to XML Schema we use attributes instead of elements where possible. An object set can be represented as an attribute of an element if it is lexical, is functionally dependent on the element, and has no order annotations. The object sets *OrderID* and *OrderDate*, for example, satisfy these conditions and appear as attributes of an *Order* element on Lines 75 and 76. Both attributes are also marked as "*required*" because of their mandatory connection to *Order* as specified by the absence of an "o" on their connection to *Order* in Figure 2.1.

When an object set is lexical but not functional and order constraints do not hold, the object set becomes an element with minimum and maximum participation constraints. *PreviousItem* in Line 18 has a minimum participation constraint of 0 and a maximum of *unbounded*.

Because XML Schema will not let us directly specify n-ary relationship sets ($n \geq 2$), we convert them all to binary relationship sets by introducing a tuple identifier. We can think of each diamond in a C-XML diagram as being replaced by a nonlexical object set containing these tuple identifiers. To obtain a name for the object set containing the tuple identifiers, we concatenate names of nonfunctionally dependent object sets. For example, given the n-ary relationship set for *Order*, *Item*, *SalePrice*, and *Qty*, we generate an *OrderItem* element (Line 63). If names become too long, we abbreviate using only the first letter of some object-set names. Thus, for example, we generate *ItemMR* (Line 11) for the relationship set connecting *Item*, *Manufacturer*, *RequestDateTime*, and *Qty*.

When a lexical object set has a one-to-one relationship with a nonlexical object set, we use the lexical object set as a surrogate for the nonlexical object set and generate a key constraint. In our example, this generates key constraints for *Order*/*OrderID* in Line 35 and *Item*/*ItemNr* in Line 39. We also use these surrogate identifiers, as needed, to maintain explicit referential integrity. Observe that in the scheme trees above, *Item* in the first tree references *Item* in the root of the second scheme tree and also that *PreviousItem* in the second scheme tree is a role and therefore a specific specialization (or subset) of *Item* in the root. Thus, we generate *keyref* constraints, one in Lines 69–72 to ensure the referential integrity of *ItemNr* in the *OrderItem* element and another in Lines 22–25 for the *PreviousItem* element.

Another construct in C-XML we need to translate is generalization/specialization. XML Schema uses the concept of *substitution groups* to allow the use of multiple element types in a given context. Thus, for example, we generate an abstract element for *Customer* in Line 44, but then specify in Lines 45–55 a substitution group for *Customer* that allows *RegularCustomer* and *PreferredCustomer* to appear in a *Customer* context. We model content that would normally be associated with the generalization by generating a *group* that is referenced in each specialization (in Lines 47 and 52). In our example, we generate the group *CustomerDetails* and nest the details of *Customer* such as *CustomerName*, *CustomerAddr*, and *Orders* under *CustomerDetails* as we do beginning in Line 56. Further, we can nest any information

that only applies to one of the specializations directly with that specialization; thus, in Line 48 we nest *Discount* under *PreferredCustomer*.

Finally, XML documents need to have a single content root node. Thus, we assume the existence of an element called *Document* (Line 4) that serves as the universal content root.

2.3.2 Translation from XML Schema to C-XML

We translate XML Schema instances to C-XML by separating structural XML Schema concepts (such as elements and attributes) from non-structural XML Schema concepts (such as attribute types and order constraints). Then we generate C-XML constructs for the structural concepts and annotate generated C-XML object sets with the non-structural information.

We can convert an XML Schema S to a C-XML model instance C_S by generating object sets for each element and attribute type connected by relationship sets according to the nesting structure of S. Figure 2.3 shows the result of applying our conversion process to the XML Schema instance of Figure 2.2. Note that we nest object and relationship sets inside one another corresponding to the nested element structure of the XML Schema instance. Whether we display C-XML object sets inside or outside one another has no semantic significance. The nested structure, however, is convenient because it corresponds to the natural XML Schema instance structure.

The initial set of generated object and relationship sets is straightforward. Each element or attribute generates exactly one object set, and each element that is nested inside another element generates a relationship set connecting the two. Each attribute associated with an element e always generates a corresponding object set a and a relationship set r connecting a to the object set generated by e. Participation constraints for attribute-generated relationship sets are always *1..** on the a side and are either *1* or *0..1* on the e side. Participation constraints for relationship sets generated by element nesting require a bit more work. If the element is in a *sequence* or a *choice*, there may be specific minimum/maximum occurrence constraints

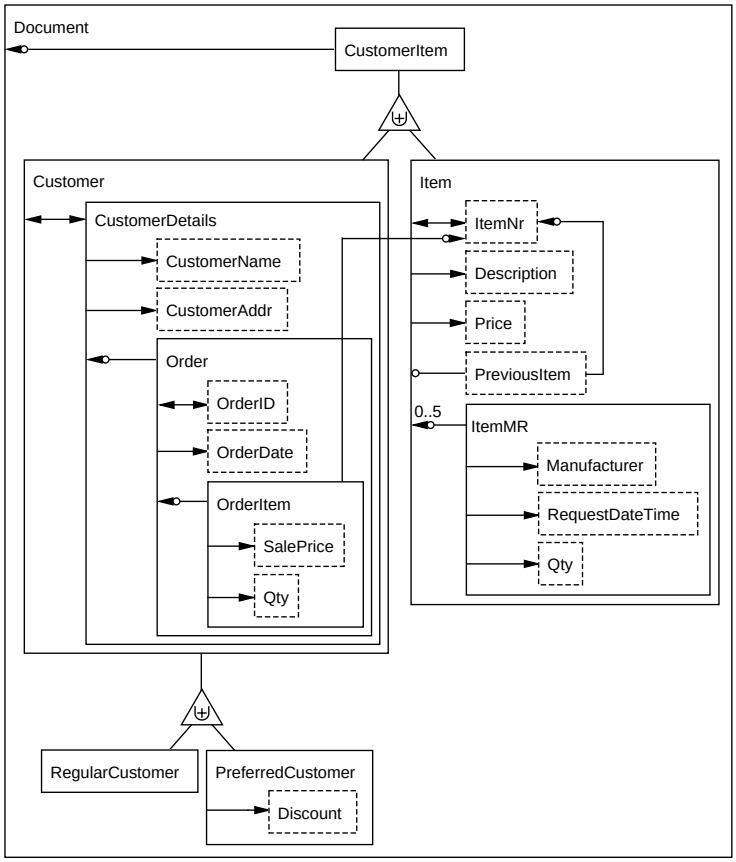


Figure 2.3: C-XML Model Instance Translated from Figure 2.2.

we can use directly. For example, according to the constraints on Line 60 in Figure 2.2 a *CustomerDetails* element may contain a list of 0 or more *Order* elements. However, an *Order* element must be nested inside a *CustomerDetails* element. Thus, for the relationship set connecting *CustomerDetails* and *Order*, we place participation constraints of *0..** on the *CustomerDetails* side, and *1* on the *Order* side.

In order to make the generated C-XML model instance less redundant, we look for certain patterns and rewrite the generated model instance when appropriate. For example, since *ItemNr* has a key constraint, we infer that it is one-to-one with *Item*. Further, the keyref constraints on *ItemNr* for *PreviousItem* and *OrderItem* indicate that rather than create two additional *ItemNr* object sets, we can instead relate *PreviousItem* and *OrderItem* to the *ItemNr* nested in *Item*. Another optimization is the treatment of substitution groups. In our example, since *RegularCustomer*

and *PreferredCustomer* are substitutable for *Customer*, we construct a generalization/specialization for the three object sets and factor out the common substructure of the specializations into the generalization. Thus, *CustomerDetails* exists in a one-to-one relationship with *Customer*.

Another complication in XML Schema is the presence of anonymous types. For example, the complex type in Line 5 of Figure 2.2 is a choice of 0 or more *Customer* or *Item* elements. We need a generalization/specialization to represent this, and since C-XML requires names for object sets, we simply concatenate all the top-level names to form the generalization name *CustomerItem*.

There are striking differences between the C-XML model instances of Figures 2.1 and 2.3. The translation to XML Schema introduced new elements *Document*, *CustomerDetails*, *OrderItem*, and *ItemMR* in order to represent a top-level root node, generalization/specializations, and decomposed n-ary relationship sets. If we knew that a particular XML Schema instance was generated from an original C-XML model instance, we could perform additional optimizations. For example, if we knew *CustomerDetails* was fabricated by the translation to XML Schema, we could observe that in the reverse translation to C-XML it is superfluous because it is one-to-one with *Customer*. Similarly, we could recognize that *Document* is a fabricated top-level element and omit it from the reverse translation; this would also eliminate the need for *CustomerItem* and its generalization/specialization. Finally, we could recognize that n-ary relationship sets have been decomposed, and in the reverse translation reconstitute them. The original C-XML to XML Schema translation could easily place annotation objects in the generated XML Schema instance marking elements for this sort of optimization.

2.3.3 Information and Constraint Preservation

To formalize information and constraint preservation for schema translations, we use first-order predicate calculus. We represent any schema specification (which for C-XML is a model instance and for XML is an XML Schema instance) in predicate calculus by generating an n-place predicate for each n-ary tuple container and a

closed formula for each constraint [19]. Using the closed-world assumption, we can then populate the predicates to form an interpretation. If all the constraints hold over the populated predicates, the interpretation is valid.

For any schema specification S_A of type A (e.g. S_{C-XML} or $S_{XMLSchema}$ in our discussion here) there is a corresponding valid interpretation I_{S_A} (i.e. a valid, populated model instance for a C-XML model instance or a conforming XML document for an XML Schema instance). We can guarantee that a translation T translates a schema specification S_A to a constraint-equivalent schema specification S_B by checking whether the constraints of the generated predicate calculus for the schema specification of type B imply the constraints of the generated predicate calculus for the schema specification of type A (i.e. by checking whether $Constraints(S_B^{PC}) \Rightarrow Constraints(S_A^{PC})$, where the superscript PC denotes that the schema is predicate calculus). A translation T that translates a schema specification S_A into a schema translation S_B induces a translation T' from an interpretation I_{S_A} for a schema of type A to an interpretation I_{S_B} for a schema of type B. We can guarantee that a T-induced translation T' translates any valid interpretation I_{S_A} into an information equivalent valid interpretation I_{S_B} by translating both of the corresponding valid interpretations to predicate calculus interpretations $I_{S_A^{PC}}$ and $I_{S_B^{PC}}$ and checking for information equivalence.

Definition 1 A translation T from schema specification S_A to a schema specification S_B *preserves information* if there exists a procedure P that for any valid interpretation I_{S_A} corresponding to S_A computes I_{S_A} from I_{S_B} where I_{S_B} is the interpretation corresponding to S_B induced by T. □

Definition 2 A translation T from schema specification S_A to a schema specification S_B *preserves constraints* if the constraints of S_B imply the constraints of S_A. □

Lemma 1 Let $I_{S_{C-XML}}$ be a valid interpretation for a populated C-XML model instance S_{C-XML}. There exists a translation T_{C-XML} that correctly represents $I_{S_{C-XML}}$ as a valid interpretation $I_{S^{PC}_{C-XML}}$ in predicate calculus.

Proof: We construct T_{C-XML} as follows. We generate 1-place predicates for object

sets (e.g. $OrderDate(x)$), n-place predicates for relationship sets (e.g. $Order\_OrderDate(x, y)$), and closed formulas corresponding to the constraints (e.g. $\forall x(\exists y Order\_OrderDate(x, y) \Rightarrow OrderDate(x))$). We then populate the predicates with corresponding constants representing each object and relationship in the model instance to obtain T_{C-XML}. Since T_{C-XML} includes all and only all objects in 1-place predicates and all and only all relationships in n-place predicates and represents all and only all constraints, T_{C-XML} correctly represents $I_{S_{C-XML}}$ as a valid interpretation $I_{S^{PC}_{C-XML}}$ in predicate calculus. See [19] for details. □

Lemma 2 Let $I_{S_{XMLSchema}}$ be an XML document that conforms to an XML Schema instance $S_{XMLSchema}$. There exists a translation $t_{XMLSchema}$ that correctly represents $I_{S_{XMLSchema}}$ as a valid interpretation $I_{S^{PC}_{XMLSchema}}$ in predicate calculus.

Proof: We construct $t_{XMLSchema}$ as follows. Similar to [19], we generate 1-place predicates for elements and attributes (e.g. $PreferredCustomer(x)$), 2-place predicates for each attribute of an element and for each element nested within another element (e.g. $PreferredCustomer\_Discount(x, y)$), and closed formulas corresponding to the constraints (e.g. $\forall x(PreferredCustomer(x) \Rightarrow Customer(x))$. We generate constants corresponding to the data in the document, and populate the 1- and 2-place predicates accordingly. Since $t_{XMLSchema}$ includes all and only all objects in 1-place predicates and all and only all relationships in n-place predicates and represents all and only all constraints, $t_{XMLSchema}$ correctly represents $I_{S_{XMLSchema}}$ as a valid interpretation $I_{S^{PC}_{XMLSchema}}$ in predicate calculus. □

Theorem 1 Let T be the translation described in Section 2.3.1 that translates a C-XML model instance S_{C-XML} to an XML Schema instance $S_{XMLSchema}$. T preserves information and constraints.

Proof: Let T' be the induced translation of T that translates a valid, populated model instance $I_{S_{C-XML}}$ for S_{C-XML} to an XML document $I_{S_{XMLSchema}}$ for $S_{XMLSchema}$. By Lemma 1, we can obtain $I_{S^{PC}_{C-XML}}$ as a valid interpretation for $I_{S_{C-XML}}$ in predicate calculus; similarly by Lemma 2, we can obtain $I_{S^{PC}_{XMLSchema}}$ as a valid interpretation for $I_{S_{XMLSchema}}$ in predicate calculus. According to Definition 1 we must show that

there is a procedure P that can construct each populated predicate in $I_{S^{PC}_{C-XML}}$ from $I_{S^{PC}_{XMLSchema}}$. The 1-place predicates map directly, but the n-place predicates are more interesting since $I_{S^{PC}_{XMLSchema}}$ has binary predicates decomposed from n-place predicates. To recover the original n-place predicates, we join the binary predicates and project the n required columns. According to Definition 2, we must also show that the constraints of $I_{S^{PC}_{XMLSchema}}$ imply the constraints of $I_{S^{PC}_{C-XML}}$. This requires a case analysis of the generated constraints. See [19] for a list of cases. □

Theorem 2 Let T be the translation described in Section 2.3.2 that translates an XML Schema instance $S_{XMLSchema}$ to a C-XML model instance S_{C-XML}. T preserves information and constraints.

Proof: Like Theorem 1, the proof is by case analysis, showing how each XML Schema construct maps to C-XML. Again we use Lemmas 1 and 2 to provide predicate calculus interpretations, and then we need to show that (1) each predicate in the XML Schema interpretation can be constructed from those in the C-XML interpretation, and (2) each constraint in the XML Schema interpretation is implied by the constraints of the C-XML interpretation. □

2.4 C-XML Views

This section describes three types of views—simple views that help us scale up to large and complex XML schemas, query-generated views over a single XML schema, and query-generated views over heterogeneous XML schemas.

2.4.1 High-Level Abstractions in C-XML

We create simple views in two ways. Our first way is to nest and hide C-XML components inside one another [19]. Figure 2.3 shows how we can nest object sets inside one another. We can pull any object set inside any other connected object set, and we can pull any object set inside any connected relationship set so long as we leave at least two object sets outside (e.g. in Figure 2.1 we can pull *Qty* and/or *SalePrice* inside the diamond). Whether an object set appears on the inside

or outside has no effect on the meaning. Once we have object sets on the inside, we can implode the object set or relationship set and thus remove the inner object sets from the view. We can, for example, implode *Customer*, *Item*, and *PreferredCustomer* in Figure 2.3, presenting a much simpler diagram showing only five object sets and two generalization/specialization components nested in *Document*. To denote an imploded object or relationship set, we shade the object set or the relationship-set diamond. Later, we can explode object or relationship sets and view all details. Since we allow arbitrary nesting, it is possible that relationship-set lines may cross object- or relationship-set boundaries. In this case, when we implode, we connect the line to the imploded object or relationship set and make the line dashed to indicate that the connection is to an interior object set.

Our second way to create simple views is to discard C-XML components that are not of interest. We can discard any relationship set, and we can discard all but any two connections of an n-ary relationship set ($n > 2$). We can also discard any object set, but then must discard (1) any connecting binary relationship sets, (2) any connections to n-ary relationship sets ($n > 2$), and (3) any specializations and relationship sets or relationship-set connections to these specializations. Figure 2.4 shows an example of a high-level abstraction of Figure 2.1. In Figure 2.4 we have discarded *Price* and its associated binary relationship set, the relationship set for *PreviousItem*, and the connections to *RequestDateTime* and *Qty* in the n-ary relationship set involving *Manufacturer*. We have also hidden *OrderID*, *OrderDate*, and all customer information except *CustomerName* inside *Order*, and we have hidden *SalePrice* and *Qty* inside the *Order-Item* relationship set. Note that both the *Order* object set and the *Order-Item* relationship set are shaded, indicating the inclusion of C-XML components; that neither the *Item* object set nor the *Item-Manufacturer* relationship set are shaded, indicating that the original connecting information has been discarded rather than hidden within; and that the line between *CustomerName* and *Order* is dashed, indicating that *CustomerName* connects, not to *Order* directly, but rather to an object set inside *Order*.

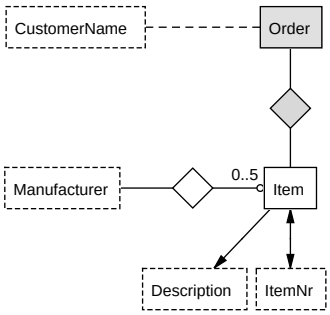


Figure 2.4: High-Level View of Customer/Order C-XML Model Instance.

Theorem 3 Simple, high-level views constructed by properly discarding C-XML components are valid C-XML model instances.

Proof: Clear based on the construction procedure. □

Corollary 1 Any simple, high-level view can be represented by an XML Schema.

Proof: Since each high level view is a valid C-XML model instance (Theorem 3), and since each C-XML model instance can be represented by an XML-Schema instance (Section 2.3.1), thus any simple, high-level view can be represented by an XML Schema. □

2.4.2 C-XML XQuery Views

We now consider the use of C-XML views to generate XQuery views. As other researchers have pointed out [8, 13], XQuery can be hard for users to understand and manipulate. One reason XQuery can be cumbersome is because it must follow the particular hierarchical structure of an underlying XML schema, rather than the simpler, logical structure of an underlying conceptual model. Further, different XML sources might specify conflicting hierarchical representations of the same conceptual relationship [8]. Thus, it is highly desirable to be able to construct XQuery views by generating them from a high-level conceptual model-based description. [13] describes an algorithm for generating XQuery views from ORA-SS descriptions. [8] also describes how to specify XQuery views by writing *conceptual XPath* expressions over a conceptual schema and then automatically generating the corresponding XQuery

```
define view CustomersByItemsOrdered
{    for $item in Item
     return
     <Item>
          {$item/ItemNr, $item/Description}
          {    for $customer in $item/Order/Customer
               return
               <Customer>
                    {$customer/CustomerName, $customer/CustomerAddr}
                    {    for $order in $customer/Order,
                              $item2 in $order/Item
                         where $item2 = $item
                         return
                         <Order>
                              {$order/OrderDate, $item2/Qty, $item2/SalePrice}
                         </Order>
                    }
               </Customer>
          }
     </Item>
}
```

Figure 2.5: C-XQuery View of Customers Nested within Items Ordered.

specifications. In a similar fashion, we can generate XQuery views directly from high-level C-XML views. In some situations a graphical query language would be an excellent choice for creating C-XML views [25], but in keeping with the spirit of C-XML we define an XQuery-like textual language called C-XQuery.

Figure 2.5 shows a high-level view written in C-XQuery over the model instance of Figure 2.1. We introduce a view definition with the phrase *define view*, and specify the contents of the view with FLWOR (for, let, where, order by, return) expressions [43]. The first *for $item in Item* phrase creates an iterator over objects in the *Item* object set. Since there is no top-level *where* clause, we iterate over all the items. Also, since C-XML model instances do not have "root nodes" the idea of context is different. In this case, *Item* defines the *Item* object set as the context of the path expression. For each such item, we return an *<Item>* ... *</Item>* structure populated according to the nested expressions.

C-XQuery is much like ordinary XQuery, with the main distinguishing factor that our path expressions are conceptual, and so, for example, they are not concerned with the distinction between attributes and elements. Note particularly that for the data fields, such as *ItemNr*, *CustomerName*, and *OrderDate*, we do not care

```
define view RecentNitrogenFertilizerCustomers
{    for $i in CustomersByItemsOrdered/Item
     where $i/Description = "Nitrogen Fertilizer"
     return
     <Customer>
          {    for $c in $i/Customer
               let $total := sum( for $o in $c/Order
                                  where $o/OrderDate > add-days(current-date(),-90)
                                  return $o/Qty * $o/SalePrice )
               return
               {$c/CustomerName, Total=$total}
          }
     </Customer>
}

for $c in RecentNitrogenFertilizerCustomers/Customer
where $c/total > 300
return
<PotentialThreatCustomer>
     {$c/CustomerName, $c/Total}
</PotentialThreatCustomer>
```

Figure 2.6: C-XQuery over the View of Customers Nested within Items Ordered.

whether the generated XML treats them as attributes or elements. A more subtle characteristic of our conceptual path expressions is that since they operate over a flat C-XML structure, we can traverse the conceptual-model graph more flexibly, without regard for hierarchical structure. Thus, we generalize the notion of a path expression so that the expression $A//B$ designates the path from A to B regardless of hierarchy or the number of intervening steps in the path [25]. This can lead to ambiguity in the presence of cycles or multiple paths between nodes, but we can automatically detect ambiguity and require the user to disambiguate the expression (say, by designating an intermediate node that fixes a unique path).

Given a view definition, we can write queries against the view. For the view in Figure 2.5, for example, the query in Figure 2.6 finds customers who have purchased more than $300 worth of nitrogen fertilizer within the last 90 days. To execute the query, we unfold the view according to the view definition and minimize the resulting XQuery. See [39] for a discussion of the underlying principles.

The view in Figure 2.6 illustrates the use of views over views. Indeed, applications can use views as first-class data sources, just like ordinary sources, and we can write queries against the conceptual model and views over that model. In any case,

we translate the conceptual queries to XQuery specifications over the XML Schema instance generated for the C-XML conceptual model.

Theorem 4 A C-XQuery view Q over a C-XML model instance C can be translated to an XQuery query Q_C over an XML Schema instance S_C.

Proof: Observe that by the definition of XQuery [43], any valid XQuery instance generates an underlying XML Schema instance. By Theorem 4, we thus know that for any C-XQuery view we retain a correspondence to XML Schema. In particular, this means we can compose views of views to an arbitrary depth and still retain a correspondence to XML Schema. □

2.4.3 XQuery Integration Mappings

To motivate the use of views in enterprise conceptual modeling, suppose through mergers and acquisitions we acquire the catalog inventory of another company. Figure 2.7 shows the C-XML for this assumed catalog. We can rapidly integrate this catalog into the full inventory of the parent company by creating a mapping from the acquired company's catalog to the parent company's catalog. Figure 2.8 shows such a mapping. In order to integrate the source (Figure 2.7) with the target (Figure 2.1), the mapping needs to generate target names in the source. In this example, *CatalogItem*, *CatalogNr*, and *ShortName* correspond respectively to *Item*, *ItemNr*, and *Description*. We must compute *Price* in the target from the *MSRP* and *MarkupPercent* values in the source, as Figure 2.8 shows. We assume the function *CatalogNr-to-ItemNr* is either a hand-coded lookup table, or a manually-programmed function to translate source catalog numbers to item numbers in the target. The underlying structure of this mapping query corresponds directly to the relevant section of the C-XML model instance in Figure 2.1, so integration is now immediate.

The mapping in Figure 2.8 creates a target-compatible C-XQuery view over the acquired company's catalog in Figure 2.7. When we now query the parent company's items, we also query the acquired company's catalog. Thus, the previous examples are immediately applicable. For example, we can find those customers who have

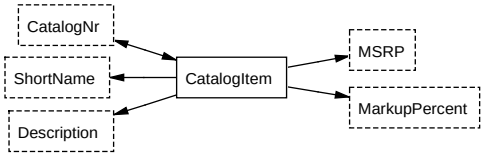


Figure 2.7: C-XML Model Instance for the Catalog of an Acquired Company.

```
define view CatalogItemToItem
{    for $cItem in CatalogItem
     let $itemNr := CatalogNr-to-ItemNr($cItem)
     let $price := $cItem/MSRP * (1 + $cItem/MarkupPercent)
     return
          <Item>
               <ItemNr>{$itemNr}</ItemNr>
               <Description>{$cItem/ShortName}</Description>
               <Price>{$price}</Price>
          </Item>
}
```

Figure 2.8: C-XQuery Mapping for Catalog Integration

ordered more than $300 worth of nitrogen fertilizer from either the inventory of the parent company or the inventory of the acquired company by simply issuing the query in Figure 2.6. With the acquired company's catalog integrated, when the query in Figure 2.6 iterates over customer orders, it iterates over data instances for both *Item* in Figure 2.1 and *CatalogItem* in Figure 2.8. (Now, if the potential terrorist has purchased, say $200 worth of nitrogen fertilizer from the original company and $150 worth from the acquired company, the potential terrorist will appear on the list, whereas the potential terrorist would have appeared on neither list before.)

We could also write a mapping query going in the opposite direction, with Figure 2.1 as the source and Figure 2.7 as the target. Such bidirectional integration is useful in circumstances where we need to shift between perspectives, as is often the case in enterprise application development. This is especially true because all enterprise data is rarely fully integrated.

In general it would be nice to have a mostly automated tool for generating integration mappings. In order to support such a tool, we require two-way mappings between both schemas and data elements. Sometimes we can use automated element matchers [6, 31] to help us with the mapping. However, in other cases the mappings

are intricate and require programmer intervention (e.g. calculating *Price* from *MSRP* plus a *MarkupPercent* or converting *CatalogNr* to *ItemNr*). In any case, we can write C-XQuery views describing each such mapping, with or without the aid of tools (e.g. [27]), and we can compose these views to provide larger C-XQuery schema mappings. Of course there are many integration details we do not address here, such as handling dirty data, but the approach of integrating by composing C-XQuery views is sound.

2.5 Concluding Remarks

We have offered Conceptual-XML (C-XML) as an answer to the challenge of modern enterprise modeling. C-XML is equivalent in expressive power to XML Schema (Theorems 1 and 2). In contrast to XML Schema, however, C-XML provides for high level conceptualization of an enterprise. C-XML allows users to view schemas at any level of abstraction and at various levels of abstraction in the same specification (Theorem 3), which goes a long way toward mitigating the complexity of large data sets and complex interrelationships. Along with C-XML, we have provided C-XQuery, a conceptualization of XQuery that relieves programmers from concerns about the often arbitrary choice of nesting and arbitrary choice of whether to represent values with attributes or with elements. Using C-XQuery, we have shown how to define views and automatically translate them to XQuery (Theorem 4). We have also shown how to accommodate heterogeneity by defining mapping views over federated data repositories and automatically translate them to XQuery.

Implementing C-XML is a huge undertaking. Fortunately, we have a foundation on which to build. We have already implemented tools relevant to C-XML that include graphical diagram editors, model checkers, textual model compilers, a model execution engine, and several data integration tools. We are actively continuing development of an Integrated Development Environment (IDE) for modeling-related activities. Our strategy is to plug new tools into this IDE rather than develop stand-alone programs. Our most recent implementation work consists of tools for automatic generation of XML normal form schemes. We are now working on the implementation

of the algorithms to translate C-XML to XML Schema, XML Schema to C-XML, and C-XQuery to XQuery.

Chapter 3

Representing Generalization/Specialization in XML Schema

3.1 Introduction

The scientific community has long recognized the importance of *generalization*—and its inverse, *specialization*—as a fundamental and highly useful modeling construct (see, for example, [36, 4]). Generalization/specialization is used broadly in conceptual models such as UML [7], and EER [40], and in description logics [3]. The main idea in generalization/specialization, also called the *is-a* relationship, is that one set, class, or concept is a subset of another. If A is a generalization of B (or equivalently, if B is a specialization of A), we say that B is a subset of A (B *is-a* A). In general, concepts form a hierarchy wherein a generalization may have many specializations, and a specialization may have many generalizations. It is often useful, however, to define constraints over generalization/specialization hierarchies. For example, we can declare two specializations of a common generalization to be *mutually exclusive*. We can also declare the specializations of a concept to be complete in the sense that their *union* contains all members of the generalization. If both of these constraints are present (a common occurrence), the specializations *partition* the generalization space. A less common constraint is the situation where a specialization constitutes the *intersection* of its multiple generalizations.

In this chapter we illustrate our examples using *Conceptual XML* (C-XML) [20] which is a conceptual model consisting of object sets, relationship sets, and

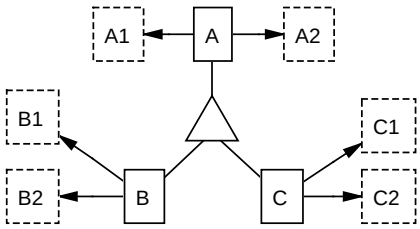


Figure 3.1: Generalization/Specialization in C-XML.

constraints over these object and relationship sets.[1] In C-XML, we represent object sets or concepts by writing names inside rectangles, with a solid border indicating a nonlexical concept and a dashed border indicating a lexical concept. In Figure 3.1, each nonlexical concept has two related lexical concepts whose relationship sets are functional, indicated by arrows (e.g., for each A, there is exactly one *A1*, but multiple A's may have the same *A1* value). In C-XML, a triangle denotes generalization/-specialization. For example, in Figure 3.1 the set of objects in B is a subset of the set of objects in A. C-XML allows modelers to constrain generalizations by writing a constraint symbol in a triangle. A plus symbol ($+$) in indicates that the specialization sets are mutually exclusive. A union symbol ($\cup$), specifies that the set of objects in the generalization is the union of the specialization object sets. A plus and union together ($\uplus$) specify that the specializations partition the generalization since there is both a union and a mutual-exclusion constraint. An intersection symbol ($\cap$) indicates that the members of the specialization object set constitute the intersection of the generalization object sets.

These simple definitions find many different, intricate, and complicating expressions in conceptual models and schema description languages. For example, a typical object-oriented "class" is a *type* rather than a mathematical set, and it uses the *inheritance* relationship and the notion of substitutability in place of the more general concept of generalization/specialization and simple *is-a* semantics. This leads

[1]The particular choice of conceptual model is not critical to this chapter, since the various conceptual models and description logics typically have very similar underlying generalization/specialization constructs.

to a potential mismatch between how we model the real world and how we implement information systems.[2]

XML Schema has rapidly become the method of choice for describing XML document structures. Since XML is the de facto standard for modern data interchange, it is important that we understand how to properly capture and enforce constraints on XML document structures. Thus, a number of researchers have studied how to transform conceptual models into XML Schema. A chapter in [10] describes how to translate a UML model instance into XML Schema. A chapter in [16] presents Relax NG and introduces how to translate from the Asset Oriented Modeling conceptual model into XML Schema. Yet another study shows how to translate Object Role Modeling into XML Schema [5]. In each case, the discussion is about translation in general and does not focus specifically on the problem of fully capturing all the semantics of generalization/specialization. In this chapter we deal with the full details of translating generalization/specialization and its constraints into XML Schema.

The remainder of the chapter proceeds as follows. In Section 3.2 we describe the mechanisms available in XML Schema to represent generalization/specialization. In Section 3.3 we show how to use those mechanisms to capture the semantics of certain forms of generalization/specialization and its constraints. Since XML Schema is incapable of fully representing all of the necessary semantics, in Section 3.4 we describe a relatively small but important set of augmentations that would allow XML Schema to do a complete job. We conclude in Section 3.5.

3.2 Generalization/Specialization Mechanisms in XML Schema

There are several mechanisms in XML Schema that support generalization/-specialization. The foundational information construct in XML, of course, is the *element*, which together with the *attribute* construct and element nesting is sufficient to represent all data structures. So the starting point for any translation from a conceptual model to XML is to map "concepts" to "elements." Relationships typically

[2]Thus some developers adopt the rule of thumb that class derivation (inheritance) should only be used when *is-a* also holds for the derivation relationship. But this rule is not applied universally.

map either to attributes or to nested elements. There are significant complications when we consider finer points like object identity, but the overall process of structure mapping is fairly clear-cut and generally intuitive.

However, once we have a basic structure encoded in XML, how can we capture generalization/specialization relationships and their constraints? We find three constructs in XML Schema that support various aspects of generalization/specialization: (1) derived types, (2) substitution groups, and (3) abstract elements. We consider each construct in turn.

3.2.1 Derived Types

In XML Schema, each element has a *type* that describes valid element content. Types come in two broad categories: *simple* and *complex*. One simple type can be derived from another by restriction. For example, *string* is a simple type, and we can specify a customized type, *GenericTLD*, as the set of strings that correspond to the generic top-level internet domains by restricting the *string* type as follows:

```
<xs:simpleType name="GenericTLD">
    <xs:restriction base="xs:string">
        <xs:enumeration value="com" />
        <xs:enumeration value="edu" />
        <xs:enumeration value="gov" />
        <xs:enumeration value="net" />
        <xs:enumeration value="org" />
        . . .
    </xs:restriction>
</xs:simpleType>
```

Similarly, complex types may be derived by restriction from a base type. Valid restrictions include those that increase the constraints on attributes or elements in the complex type in a way that is compatible with the base type. For example, an

optional element in the base type may be required in the derived type. Thus, the derivation of both simple and complex types by restriction results in a set of allowed values for the derived type that is a subset of the allowed values for the base type. This notion is similar to the conceptual *is-a* relationship.

Extension of complex types involves creating a derived complex type whose content model is a superset of its base type's content model. When we extend a complex type, we can add to the derived type extra attributes or elements in addition to those found in the content model of the base type as follows:

```
<xs:complexType name="A">
    <xs:sequence>
        <xs:element name="A1" type="xs:string" />
        <xs:element name="A2" type="xs:string" />
    </xs:sequence>
</xs:complexType>

<xs:complexType name="B">
    <xs:complexContent>
        <xs:extension base="A">
            <xs:sequence>
                <xs:element name="B1" type="xs:string" />
                <xs:element name="B2" type="xs:string" />
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

In this case, type B is derived by extension from type A. In addition to including *A1* and *A2* elements, B-type elements also include *B1* and *B2* elements.

The concept of a extension constitutes a form of generalization/specialization. The element that we derive from its type contains the common properties of attributes and elements. Thus, we consider it as a generalization element. The element that derives its type from the type of the generalization element inherits all properties of the generalization element and it also has additional properties of attributes and elements that do not already exist in the generalization element. Thus, we consider it as a specialization element.

3.2.2 Substitution Groups

In XML Schema, global elements can be organized into a *substitution group*, wherein a particular set of elements can be substituted for a named element called the *head element*. For example, if elements B and C were each declared to be substitutable for A by including the attribute *substitutionGroup="A"* in the declarations of elements B and C, then the meaning is that B or C may appear anywhere that A is required. The presence of a substitution group does not require use of the substitutable elements, nor does it preclude the use of the head element. It simply establishes a way for a set of elements to be used interchangeably.

The concept of a substitution group constitutes a form of generalization/specialization, though it is not identical to the natural subset notion of generalization/-specialization that corresponds to the *is-a* relationship described in the introduction. Instead, a substitution group defines an equivalence class of elements that can be used interchangeably. However, substitution groups can form hierarchies similar to *is-a* hierarchies, and we can construe them to denote a relationship much like *is-a*. Indeed, we argue that the use of a substitution group implies conceptual generalization/specialization in the sense that one concept (a substitutable element) is a special kind of another concept (the head element).

3.2.3 Abstract Elements and Types

It is possible to require the use of substitution for a particular element or type by declaring it to be *abstract*. An element declared to be abstract cannot be used in

an instance document—a non-abstract substitutable element must be used instead. Thus, declaring an element as abstract requires the specification of a substitution group. Similarly, declaring a type to be abstract requires the use of concrete types that extend the abstract type. In both cases, abstract elements are associated with concept hierarchies that are related to the conceptual *is-a* relationship.

3.3 Representing Generalization/Specialization in XML Schema

Given the foundational XML Schema mechanisms described in Section 3.2, we now turn our attention to how we can actually represent conceptual generalization/specialization in XML Schema. There are two cases of conceptual generalization/specialization that we are able to represent faithfully in XML Schema, two cases that are problematic, and two other cases that are not possible (directly). When a generalization/specialization hierarchy does not include multiple generalizations for any specialization (i.e., no concept has more than one parent concept), we are able to represent generalization/specialization relationships with partition and mutual-exclusion constraints in a straightforward manner as we show in Section 3.3.1. We are also able to represent union constraints and unconstrained generalization/specialization relationships, but as we describe in Section 3.3.2, these cases are more problematic. In Section 3.3.3 we discuss the cases that we cannot model reasonably in XML Schema, namely generalization/specialization relationships involving multiple generalizations.

3.3.1 Straightforward Cases

The two straightforward cases of generalization/specialization constraints are partition and mutual-exclusion. In both cases, we start by translating concepts into elements and attributes, and relationship sets into attributes and nested elements. The primary means for representing generalization/specialization in XML Schema is captured by the notion of substitution groups, so we have chosen to represent each generalization/specialization relationship with an XML Schema substitution group.

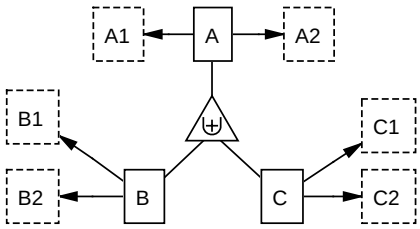


Figure 3.2: Generalization/Specialization Partition Constraint in C-XML.

We now describe how to represent partition and mutual-exclusion constraints in XML Schema.

Partition Constraints

Figure 3.2 shows a C-XML model instance where specialized concepts B and C form a partition of the general A concept. In set terminology, we say that $B \cup C = A$ and $B \cap C = \{\}$. Figure 3.3 shows our XML Schema translation of this model instance.

The translation from C-XML in Figure 3.2 to XML Schema in Figure 3.3 proceeds as follows. We begin by introducing *Document* as a root-level node that contains a sequence of A elements. We declare A as an abstract type whose content is defined by the complex type *Atype* (line 11). Since A is abstract, it cannot appear independently in an instance document—either B or C must be substituted. This serves the purpose of covering the union constraint (recall that partition is the combination of union and mutual exclusion), since A must necessarily be defined as the union of the set of B's and C's that actually appear in the instance document.

Atype declares that the content model of A includes exactly one *A1* element and exactly one *A2* element (lines 12-16). Furthermore, we define an object identifier attribute *OID* of type *ID* (line 17) that serves as a unique identifier for each A. XML Schema defines the special *ID* type to be unique across an entire document instance. Since B and C must be mutually exclusive, we can ensure that the sets are disjoint simply by providing a unique surrogate identifier for each element. A key point here is how we deal with the issue of object identity. How do we know whether a B element

```
 1: <?xml version="1.0" encoding="UTF-8"?>
 2: <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
 3:         elementFormDefault="qualified" attributeFormDefault="unqualified">
 4: <xs:element name="Document">
 5:   <xs:complexType>
 6:     <xs:sequence minOccurs="0" maxOccurs="unbounded">
 7:       <xs:element ref="A" />
 8:     </xs:sequence>
 9:   </xs:complexType>
10: </xs:element>
11: <xs:element name="A" type="Atype" abstract="true" />
12: <xs:complexType name="Atype">
13:   <xs:sequence>
14:     <xs:element name="A1" minOccurs="1" maxOccurs="1" />
15:     <xs:element name="A2" minOccurs="1" maxOccurs="1" />
16:   </xs:sequence>
17:   <xs:attribute name="OID" type="xs:ID" use="required" />
18: </xs:complexType>
19: <xs:element name="B" type="Btype" substitutionGroup="A" />
20: <xs:complexType name="Btype">
21:   <xs:complexContent>
22:     <xs:extension base="Atype">
23:       <xs:sequence>
24:         <xs:element name="B1" type="xs:string" minOccurs="1" maxOccurs="1" />
25:         <xs:element name="B2" type="xs:string"  minOccurs="1" maxOccurs="1" />
26:       </xs:sequence>
27:     </xs:extension>
28:   </xs:complexContent>
29: </xs:complexType>
30: <xs:element name="C" type="Ctype" substitutionGroup="A" />
31: <xs:complexType name="Ctype">
32:   <xs:complexContent>
33:     <xs:extension base="Atype">
34:       <xs:sequence>
35:         <xs:element name="C1" type="xs:string" minOccurs="1" maxOccurs="1" />
36:         <xs:element name="C2" type="xs:string" minOccurs="1" maxOccurs="1" />
37:       </xs:sequence>
38:     </xs:extension>
39:   </xs:complexContent>
40: </xs:complexType>
41: </xs:schema>
```

Figure 3.3: XML Schema Translation of C-XML in Figure 3.2.

and a C element represent the same A object? Since A, B, and C are all nonlexical, we need to associate object identifiers with them. Attribute *OID* serves this purpose, and because *OID* must be unique across all elements, we are guaranteed that no B element will have the same *OID* value as some C element. Hence we know that B and C are mutually exclusive (even if a B object and a C object share the same *A1* and *A2* values).

The remainder of Figure 3.3 accounts for the specialized structure of B and C, each with its own pair of related concepts. Elements B and C are members of the substitution group whose head element is A (lines 19 and 30). Both elements B

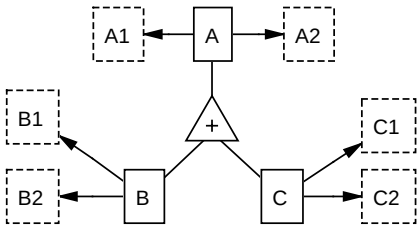


Figure 3.4: Generalization/Specialization Mutual-Exclusion Constraint in C-XML.

and C derive their content models by extension from the base *Atype* (lines 20-29 and 31-40 respectively).

Mutual-Exclusion Constraints

Figure 3.4 shows a C-XML model instance that is similar to the model instance in Figure 3.2, except that the partition constraint is replaced with the weaker mutual-exclusion constraint. The translation of the model instance to XML Schema is identical to the partition-constraint case with one exception. Since the C-XML model instance in Figure 3.4 does not force A to be the union of B and C, there may be A's present that are in neither B nor C. That is, we still have $B \cap C = \{\}$, but we no longer have $B \cup C = A$. Instead, we merely know that $B \cup C \subseteq A$. Thus we must allow instances of the A element to be directly present in the XML document instance. We accomplish this by repeating the same translation as before except we declare element A not to be abstract. The only thing that changes from Figure 3.3 is that on line 11 we write *abstract="false"* instead of *abstract="true"*.

Partition and mutual-exclusion constraints on generalization/specialization relationships are fairly straightforward to represent in XML Schema without introducing many artifacts. The two additional information-carrying elements in the XML Schema translation are the *Document* element, since XML requires a single root-level container element, and the *OID* attribute, which is necessary to capture object identity semantics. We now proceed to the more difficult union constraint and unconstrained generalization/specialization.

3.3.2 Problematic Cases in XML Schema

In contrast with the straightforward translation of partition and mutual-exclusion constraints from C-XML to XML Schema, unconstrained generalization/specialization and generalization/specialization with only a union constraint are more difficult to handle, and the mapping approach is not entirely satisfactory.

Generalization/Specialization without any Constraint

Figure 3.1 shows an unconstrained generalization/specialization relationship, where A is the general concept and B and C are specializations of A. In set notation we write $B \subseteq A$ and $C \subseteq A$. In particular, this allows for the possibility that the intersection of B and C is non-empty. And that is where the chief difficulty arises—how do we enforce object identity when $B \cap C \neq \{\}$? Figure 3.5 shows the best we can do with the available mechanisms in XML Schema to represent this case.

The differences between Figure 3.5 and Figure 3.3 are (1) element A is not abstract, thus relaxing the union constraint, and (2) the object identifier attribute *OID* is not of type *ID*, and so we do not enforce uniqueness, thus relaxing the mutual-exclusion constraint.

A, B, and C are still nonlexical concepts, and so they should have an identity in the corresponding XML Schema translation. We can argue that two elements in an XML document instance with the exact same values still have distinct identities because they are written separately in the XML document. Thus we can distinguish between the element instance written first in the document and the element instance written second. However, consider the case where an object is a member of both B and C. Since we have no combined type to represent a *B/C* element, we must write the element first as a B, and then using the same values for *OID*, *A1*, and *A2*, we must write the element as a C. Now the conceptual object that is a member of B and C is represented as two separate XML elements tied together by a common *OID* value. Besides introducing an update anomaly over *A1* and *A2*, we are in the unsatisfying position of not being able to enforce uniqueness of *OID*.

```
 1: <?xml version="1.0" encoding="UTF-8"?>
 2: <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
 3:         elementFormDefault="qualified" attributeFormDefault="unqualified">
 4: <xs:element name="Document">
 5:   <xs:complexType>
 6:     <xs:sequence minOccurs="0" maxOccurs="unbounded">
 7:       <xs:element ref="A" />
 8:     </xs:sequence>
 9:   </xs:complexType>
10: </xs:element>
11: <xs:element name="A" type="Atype" abstract="false" />
12: <xs:complexType name="Atype">
13:   <xs:sequence>
14:     <xs:element name="A1" minOccurs="1" maxOccurs="1" />
15:     <xs:element name="A2" minOccurs="1" maxOccurs="1" />
16:   </xs:sequence>
17:   <xs:attribute name="OID" type="xs:string" use="required" />
18: </xs:complexType>
19: <xs:element name="B" type="Btype" substitutionGroup="A" />
20: <xs:complexType name="Btype">
21:   <xs:complexContent>
22:     <xs:extension base="Atype">
23:       <xs:sequence>
24:         <xs:element name="B1" type="xs:string" minOccurs="1" maxOccurs="1" />
25:         <xs:element name="B2" type="xs:string"  minOccurs="1" maxOccurs="1" />
26:       </xs:sequence>
27:     </xs:extension>
28:   </xs:complexContent>
29: </xs:complexType>
30: <xs:element name="C" type="Ctype" substitutionGroup="A" />
31: <xs:complexType name="Ctype">
32:   <xs:complexContent>
33:     <xs:extension base="Atype">
34:       <xs:sequence>
35:         <xs:element name="C1" type="xs:string" minOccurs="1" maxOccurs="1" />
36:         <xs:element name="C2" type="xs:string" minOccurs="1" maxOccurs="1" />
37:       </xs:sequence>
38:     </xs:extension>
39:   </xs:complexContent>
40: </xs:complexType>
41: </xs:schema>
```

Figure 3.5: XML Schema Translation of C-XML in Figure 3.1.

The alternative is even less satisfying. We could include a combined B/C type, but there would still be two major problems. First, there would be an exponential explosion of potential combinations (imagine having just 10 or 20 specializations of one general concept—the XML Schema would be unwieldy to say the least). Second, we would break the nice correspondence between substitution groups and generalization/specialization. So, for example, a combined B/C element would let us create B elements that do not directly relate to the B element which represents the B concept in our conceptual model. Iterating over the set of B elements would become needlessly difficult.

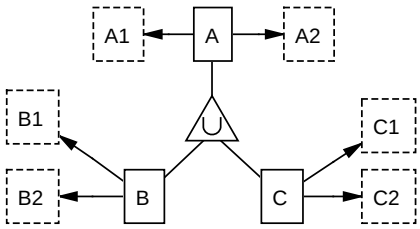


Figure 3.6: Generalization/Specialization Union Constraint in C-XML.

The advantages of our chosen approach are that it aligns more closely with the conceptual model structure, and we can enforce the appropriate constraints by post-processing outside of the ordinary XML Schema constraint enforcement mechanisms. Nonetheless, as we explore in Section 3.4, a fully satisfactory approach requires extensions to XML Schema.

Union Constraint

Figure 3.6 shows a C-XML model instance similar to the previous case except with a union constraint on the generalization/specialization relationship. For this case, our translation approach is similar to the unconstrained case in Figure 3.5 except we declare the element A to be abstract so that it cannot be instantiated directly in a document instance. With a union constraint, we know that $B \cup C = A$, and so we must prevent the situation where an A exists that is neither in B nor C. Specifying *abstract="true"* for A accomplishes this.

Unfortunately, our solution in this case suffers from the same problems we describe in Section 3.3.2, and so we are not fully satisfied with the outcome. Nonetheless, it is possible to faithfully represent the conceptual structures of our C-XML model instances in all these cases, even though we sometimes cannot enforce the constraints fully in XML Schema. The unifying mechanism of our C-XML-to-XML Schema translation of generalization/specialization is that we use substitution groups to represent the generalization/specialization hierarchy.

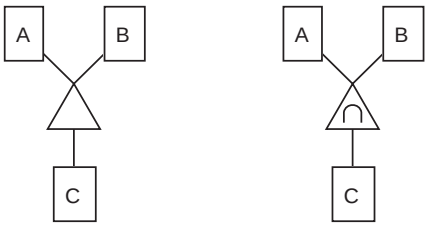


Figure 3.7: Multiple Generalizations in C-XML.

3.3.3 The Problem of Multiple Generalizations

The correspondence between generalization/specialization and substitution groups breaks down when we consider multiple generalizations. Figure 3.7 shows a simple C-XML model instance where concept C is a specialization of both A and B, so that $C \subseteq A$ and $C \subseteq B$. When we specify an intersection constraint on the generalization/specialization, we further require that $A \cap B = C$. As with the unconstrained generalization/specialization case, we must be able to handle the situation where an object is a member of more than one concept. Thus we will rely on the same *OID* mechanism as before.

However, for these cases, we simply have no way of specifying in XML Schema that an element is a member of two substitution groups. The philosophy of XML Schema 1.0 was to implement single inheritance only. Five years ago, one of the editors of the XML Schema standard acknowledged that this is a problem and that the working group might consider adding support for multiple inheritance in the future [17]. However, since inheritance combines the *is-a* construct with a code-reuse mechanism, it is not clear that simply adding multiple inheritance will resolve the problem of supporting conceptual generalization/specialization appropriately. We agree that extension from multiple types would be useful, but conceptually what we need even more is the ability for an element to participate in multiple substitution groups. (See [15] for ideas on this topic generated in a different but related context.)

3.4 Resolving the Conceptual Modeling Issues

There are two general approaches we can take to resolve the issues we have raised with respect to capturing conceptual generalization/specialization constructs in XML Schema. First, we could implement constraint-checking external to XML Schema to enforce the meaning of the conceptual model within corresponding XML documents. Alternatively, we could augment XML Schema with a few modest extensions that will support conceptual generalization/specialization directly.

3.4.1 Post-Processing to Enforce Constraints

Section 3.3.2 describes a somewhat unsatisfactory approach to mapping unconstrained and union-constrained generalization/specialization to XML Schema. What is missing in these cases is appropriate enforcement of object identity uniqueness. Consider the C-XML model instance in Figure 3.1 and its translation to XML Schema in Figure 3.5. If we were to add pragmas to the XML Schema instance to indicate that $B \subseteq A$ and $C \subseteq A$, a post-processor could examine the corresponding document instance and determine whether the object identities are all appropriate. The post-processor would need to verify the following: (1) *OID* is vertically unique across the generalization/specialization hierarchy (so, for example, there is no B element whose *OID* value is identical to some A element), and (2) when two *OID* values are the same in two sibling classes, they also share the same *A1* and *A2* values.

To implement multiple generalizations, we would need to take a somewhat different approach to laying out the C-XML concepts as XML elements. Instead of relying on substitution groups to map one-to-one with generalization/specialization, we would need to write pragmas to indicate the structure of the conceptual generalization/specialization relationships. So we might write in a specially-formatted comment, for example, that C is a specialization of both A and B, and if an intersection constraint were present we would also note that. A post-processor could readily parse the pragmas and check whether the specified constraints hold. However, since XML Schema would have no way of tying C directly to A and B, we would need to rewrite the schema so that any reference to A or B could be replaced by a C element

instead. In general, we could use this strategy to handle all generalization/specialization relationships and constraints.

Certainly the post-processor methodology has significant drawbacks; we now explore a cleaner approach.

3.4.2 Proposed Extensions to XML Schema

Perhaps the best way to implement conceptual generalization/specialization in XML Schema is to augment XML Schema with a few extensions. For multiple generalizations, it would be straightforward to extend the *substitutionGroup* attribute on a substitutable element so that it admits a list of multiple head elements. For example, to capture the generalization/specialization hierarchy of Figure 3.7, we could write the following:

<xs:element name="C" substitutionGroup="A,B" />

And if there were an intersection constraint present, we could note it with a distinguished keyword, for example:

<xs:element name="C" intersectionGroup="A,B" />

To capture the concept of union-constrained generalization/specialization, we need to mark head elements with the appropriate constraints. For example, we could mark the union constraint of Figure 3.6 in this manner:

<xs:element name="A" union="B,C" />

Similarly, mutual-exclusion and partition constraints could replace the word *union* with *mutex* and *partition*, respectively.

Finally, to cover the aforementioned cases and to handle unconstrained generalization/specialization, we would need to attach unique object identifiers uniformly to all elements in all substitution groups. We could do this by modifying XML Schema to automatically assert the existence of an *OID* attribute for all elements in a substitution group, including the head element(s) and all substitutable elements.

3.5 Conclusion

Generalization/specialization is an important structure in conceptual modeling, but it is often difficult to implement faithfully in XML Schema. This leads to XML Schema instances that are unnecessarily complex or that misrepresent the original semantics of a conceptual model. To compound the problem, it is often the case that inheritance combines the properties of the conceptual *is-a* relationship with the notion of code reuse. This can sometimes cause awkward structures that are implemented efficiently but do not correspond to a natural conceptual model.

The contributions of this chapter include the following:

- We have characterized the nature of conceptual generalization/specialization and have shown how it corresponds to structures in XML Schema.
- We have identified a small set of constructs that could augment XML Schema so that it would fully support conceptual generalization/specialization.

If our proposal were adopted, it would result in better alignment between conceptual models and corresponding XML Schema instances, and the resulting schemas would have the added benefit of being substantially less complex than the alternatives. Given the inherent complexity of enterprise application modeling and development, these advantages could be significant.

Chapter 4

Augmenting Traditional Conceptual Models to Accommodate XML Structural Constructs

4.1 Introduction

Recently, many organizations have begun to store their data using XML, and XML Schema has become the preeminent mechanism for describing valid XML document structures. Moreover, the number of applications that use XML as their native data model have increased. This increases the need for well-designed XML data models and the need for a conceptual model for designing XML schemas.

Several commercial tools provide support for graphically representing XML Schema structures. Visual Studio .NET [29] from Microsoft, Stylus Studio [38] from DataDirect Technologies, and XML Spy [37] from Altova all have their own proprietary methods for graphically representing XML structures. Each of them includes a graphical XML Schema editor that uses connected rectangular blocks to present the schema. Although these products provide visual XML Schema editing tools, they do not raise the level of abstraction because they only provide a direct view of an XML Schema document. Thus, these graphical representations do not serve the objective of conceptualizing XML Schema to be used in modeling and design.

In systems modeling and design, traditional conceptual models have proven to be quite successful for graphically representing data at a higher level of abstraction. Conceptual models represent components and their relationships to other components in the system under study in a graphical way, at a conceptual level of understanding.

Popular conceptual models that achieve these objectives are ER [11], extended ER models [40], and UML [7, 41].

XML Schema, however, introduces a few features that are not explicitly supported in these and similar conceptual models. The most important of these features include the ability to (1) order lists of concepts, (2) choose alternative concepts from among several, (3) declare nested hierarchies of information, (4) specify mixed content, and (5) use content from another data model.

The chapter makes the following contributions. First, it proposes conceptual representation for XML content structures that are not explicitly present in traditional conceptual models. Second, based on the underlying idea of the proposed representation, it suggests ways to represent missing XML content structures in two of the most popular conceptual models, ER and UML.

We present the details of our contributions as follows. Section 4.2 lists criteria an XML conceptual model should satisfy. Section 4.3 describes the structural constructs in XML Schema that are missing in traditional conceptual models. Section 4.4 explains how we model these features of XML Schema in a modeling language we call Conceptual XML (C-XML). Section 4.5 compares our proposal with other proposals for ways to extend some traditional conceptual models to represent some XML features and shows how to adapt C-XML representations for traditional conceptual models. Section 4.6 summarizes and draws conclusions.

4.2 XML Modeling Criteria

Lists of requirements for XML conceptual models have been presented in [42], [35], and [28]. Some of these requirements cover general goals of conceptual modeling, while others are specific to XML. General requirements include the following:

- *Graphical notation.* The notation should be graphical and should be user-friendly [28, 35, 42].

- *Formal foundation.* The conceptual model should have a formal foundation [28, 35, 42].

- *Structure independence.* The notation should ensure that the basics of the conceptual model are not influenced by the underlying structure, but reflect only the conceptual components of the data [28, 35, 42].

- *Reflection of the mental model.* The conceptual model must be consistent with a designer's mental conceptualization of objects and their interrelationships [35]. For example, there should be no distinction between element and attribute on the conceptual level, and hierarchies should not be required.

- *N-ary relationship sets.* The conceptual model should be able to represent n-ary relationship sets at the conceptual level [28].

- *Views.* It should be possible to transform the model to present multiple user views [28].

- *Logical level mapping.* There should be algorithms for mapping the conceptual modeling constructs to XML Schema [28, 42].

- *Constraints.* The conceptual model should support common data constraints such as cardinality and uniqueness constraints [35].

- *Cardinality for all participants.* The hierarchical structure of XML data restricts the specification of cardinality constraints only to nested participants; however, it should be possible to specify cardinality constraints for all participants at the conceptual level [28].

- *Ordering.* The conceptual model should be able to order a list of concepts [28, 35].

- *Irregular and heterogeneous structure.* The conceptual model should introduce constructs for modeling irregular and heterogeneous structure [28].

- *Document-centric data.* The conceptual model should be able to represent the mixed content and open content that XML Schema provides [28, 35, 42].

4.3 Missing Modeling Constructs

In this section we give an overview of the structural constructs in XML Schema that are missing in traditional conceptual models. We explain each and give a motivating example, which we also use in later sections to illustrate conceptual model augmentations.

The *sequence* structure specifies that the child concepts declared inside it must appear in an XML document in the order declared. Each ordered child concept can occur zero or more times within the sequence constrained by *minOccurs* and *maxOccurs* attributes. Likewise, the entire *sequence* itself can occur zero or more times. The default value for both *minOccurs* and *maxOccurs* is always *1*. The *sequence* construct may include several types of child constructs: *element*, *group*, *choice*, *sequence*, and *any*. Lines 15–23 in Figure 4.1 specify that in a complying XML document an element *School* contains a sequence of required *SchoolName*, *SchoolAddress*, and *SchoolID* elements, and an optional *SchoolMascot* element.

The *choice* structure specifies that for each choice only one of the child concepts declared within it can appear in an XML document. Each child concept in the *choice* can occur zero or more times within the choice constrained by *minOccurs* and *maxOccurs* attributes. Likewise, the entire *choice* itself can occur zero or more times. The default value for *minOccurs* and *maxOccurs* for both the entire choice and the component children is *1*. The *choice* construct may include several types of child constructs: *element*, *group*, *choice*, *sequence*, and *any*. In Figure 4.1, lines 47–55 specify that in a complying XML document an element *ContactInfo* contains one or two choices, and each choice contains either one *PhoneNumber*, one *Email*, or one *Fax*.

By default, structural constructs in XML Schema can contain child elements, but not text. To allow mixed content (child elements and text), XML Schema provides a *mixed* attribute that can be set to true. In Figure 4.1, lines 43–58 show an example of mixed content for a complex type. Setting *mixed* to true enables character data to appear between the child elements of *RecommendationLetter* in a complying XML document. Thus, the content of *RecommendationLetter* may, for example, be

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
   <xs:element name="StudentInfo">
      <xs:complexType>
         <xs:sequence>
            <xs:choice>
               <xs:element name="Name" type="xs:string"/>
               <xs:sequence>
                  <xs:element name="FirstName" type="xs:string"/>
                  <xs:element name="MiddleName" type="xs:string" minOccurs="0" maxOccurs="2"/>
                  <xs:element name="LastName" type="xs:string"/>
               </xs:sequence>
            </xs:choice>
            <xs:sequence maxOccurs="5">
               <xs:element name="School">
                  <xs:complexType>
                     <xs:sequence>
                        <xs:element name="SchoolName" type="xs:string"/>
                        <xs:element name="SchoolAddress" type="xs:string"/>
                        <xs:element name="SchoolID" type="xs:string"/>
                        <xs:element name="SchoolMascot" type="xs:string" minOccurs="0"/>
                     </xs:sequence>
                  </xs:complexType>
                  <xs:key name="schoolKey">
                     <xs:selector xpath=".//School"/>
                     <xs:field xpath="SchoolName"/>
                     <xs:field xpath="SchoolAddress"/>
                  </xs:key>
                  <xs:key name="schoolIDKey">
                     <xs:selector xpath=".//School"/>
                     <xs:field xpath="SchoolID"/>
                  </xs:key>
               </xs:element>
               <xs:element name="GraduationDate" minOccurs="0">
                  <xs:complexType>
                     <xs:sequence>
                        <xs:element name="Month" type="xs:string"/>
                        <xs:element name="Year" type="xs:string"/>
                     </xs:sequence>
                  </xs:complexType>
               </xs:element>
            </xs:sequence>
            <xs:element name="RecommendationLetter" minOccurs="0" maxOccurs="3">
               <xs:complexType mixed="true">
                  <xs:all>
                     <xs:element name="ProfessorName" type="xs:string"/>
                     <xs:element name="ContactInfo">
                        <xs:complexType>
                           <xs:choice maxOccurs="2">
                              <xs:element name="PhoneNumber" type="xs:string"/>
                              <xs:element name="Email" type="xs:string"/>
                              <xs:element name="Fax" type="xs:string"/>
                           </xs:choice>
                        </xs:complexType>
                     </xs:element>
                  </xs:all>
               </xs:complexType>
            </xs:element>
            <xs:any namespace="##other" minOccurs="0"/>
         </xs:sequence>
         <xs:attribute name="StudentNumber" type="xs:ID" use="required"/>
         <xs:anyAttribute namespace="##any"/>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

Figure 4.1: More Example of Choice/Sequence Structures in XML Schema.

"<RecommendationLetter> <ProfessorName> Dr. Jones </ProfessorName> recommends this student. Email <ContactInfo><Email>jones@ univ.edu </Email> </ContactInfo> with questions. </RecommendationLetter>".

The *any* and *anyAttribute* structures of XML Schema let designers reuse components from foreign schemata or namespaces. The *any* structure allows the insertion of any element belonging to a list of namespaces, and it can have *minOccurs* and *maxOccurs* attributes to define the number of occurrences of the *any* construct. The *anyAttribute* structure allows the insertion of any attribute belonging to a list of namespaces. Both *any* and *anyAttribute* can have *namespace* and *processContents* as attributes. The attribute *namespace* specifies the namespaces that an XML validator examines to determine the validity of an element in an XML document. The attribute *processContents* specifies how the XML processor should handle validation against the elements specified by the *any* or *anyAttribute*. In Figure 4.1, the *any* element in line 59 specifies that zero or more elements from any other namespace can appear after the *RecommendationLetter* element. Further, the *anyAttribute* specification in line 62 indicates that the *StudentInfo* element can have additional attributes from any namespace. When *processContents* is *strict*, the XML processor must obtain the schema for the required namespaces and validate the elements. When *processContents* is set to *lax*, the XML processor attempts the same processing as for *strict*, but ignores errors if validation fails. When *processContents* is *skip*, the XML processor does not attempt to validate any elements from the specified namespaces.

In XML Schema, it is possible to nest structural constructs, thus forming a hierarchy of nested constructs. In Figure 4.1, for example, *StudentInfo* has the attributes *StudentNumber* and *anyAttribute*, and it also contains the following structures in order: first, either a *Name* or a sequence of one *FirstName*, zero to two *MiddleName*'s, and one *LastName*; second, one to five sequences such that each sequence includes one *SchoolName*, one *SchoolAddress*, and an optional *GraduationDate* (the *GraduationDate* itself contains a *Month* followed by a *Year*); third, an

element *RecommendationLetter* that has two elements, *ProfessorName* and *ContactInfo* (*ContactInfo* in turn contains one to two choices such that in each choice either *PhoneNumber* or *Email* or *Fax* is specified); and fourth, an optional *any* element.

4.4 C-XML

In this section we propose an enrichment to represent XML Schema content structures that are usually missing in traditional conceptual models. Since hypergraphs provide a general representation for conceptual models, we begin with an augmented hypergraph whose vertices and edges are respectively object sets and relationship sets, and whose augmentations consist of decorations that represent constraints. A hypergraph foundation is amenable to the requirements of XML Schema, and thus this choice simplifies the correspondence between conceptual models and XML Schema. We call our representation Conceptual XML (C-XML).

We derive C-XML from OSM [19], a hypergraph-based conceptual model that defines structure in terms of *object sets* (or *concepts*), *relationship sets*, and *constraints* over these object and relationship sets. Figure 4.2 shows a C-XML model instance that corresponds to the XML schema of Figure 4.1. An object set with a solid border indicates a nonlexical concept, a dashed border indicates a lexical concept, and a double solid/dashed border indicates a *mixed* concept.[1] A shaded object set indicates a high-level object set that groups other object and relationship sets into a single object set. Lines connecting object sets are *relationship sets*. A *participation constraint* specifies how many times an object in a connected relationship may participate in a relationship set. For the most common participation constraints (*0:1*, *1:1*, *0:**, and *1:**), C-XML uses graphical notation as a shorthand: (1) an "o" on a connecting relationship-set line designates *optional participation*, while the absence of an "o" designates *mandatory*, and (2) an arrowhead specifies a *functional constraint*, limiting participation of objects on the tail side of the arrow to be at most one.

[1] In an XML document, the content string for a mixed concept might be interspersed among a number of child nodes. However, in C-XML the *mixed* concept does not explicitly specify how text and child elements can be interleaved. If the pattern for interspersing chunks of the string among child nodes matters, then the user must model text nodes explicitly (in combination with a sequence structure) rather than use the generic *mixed* construct.

The *sequence* structure representation must be able to specify concepts in a sequence in a particular order. Also, the representation must be able to specify the minimum and maximum numbers of occurrence of the whole sequence and of each child element within the sequence. For C-XML we let a bounded half circle with a directional arrow represent a sequence. The sequenced child concepts connect to the curved side, and the parent concept that contains the sequenced child concepts connects to the flat side. We place participation constraints for the entire sequence near the connection to the parent. We place participation constraints for each child near the curved side of the sequence symbol. Note that C-XML has participation constraints that represent the minimum and maximum number of occurrences of the sequence in the relationship set between the parent and the sequence. C-XML also allows participation constraints that represent the minimum and maximum allowed occurrences of the sequence in the relationship set between the sequence and each sequenced child concept.

The representation for *choice* is similar in appearance to the representation for *sequence*, but instead of an arrow we use a vertical bar to indicate choice.

For *any* and *anyAttribute* we use a high-level object set to indicate that it contains some content from another schema. XML Schema is not specific enough to designate which concept, and thus we cannot specify which concept. We therefore name these concepts "any". Conceptually, in C-XML whether the concept is an attribute or an element does not matter, and we do not distinguish between these cases.

We now evaluate C-XML with respect to the criteria for XML conceptual models in Section 4.2.

- *Graphical notation.* We have presented a sufficient graphical notation, but this is just one possibility among many.

- *Formal foundation.* OSM has a solid formal foundation in terms of predicate calculus (see Appendix A of [19]). In OSM, each object set maps to a one-place predicate, and each n-ary relationship set ($n \geq 2$) relationship set maps

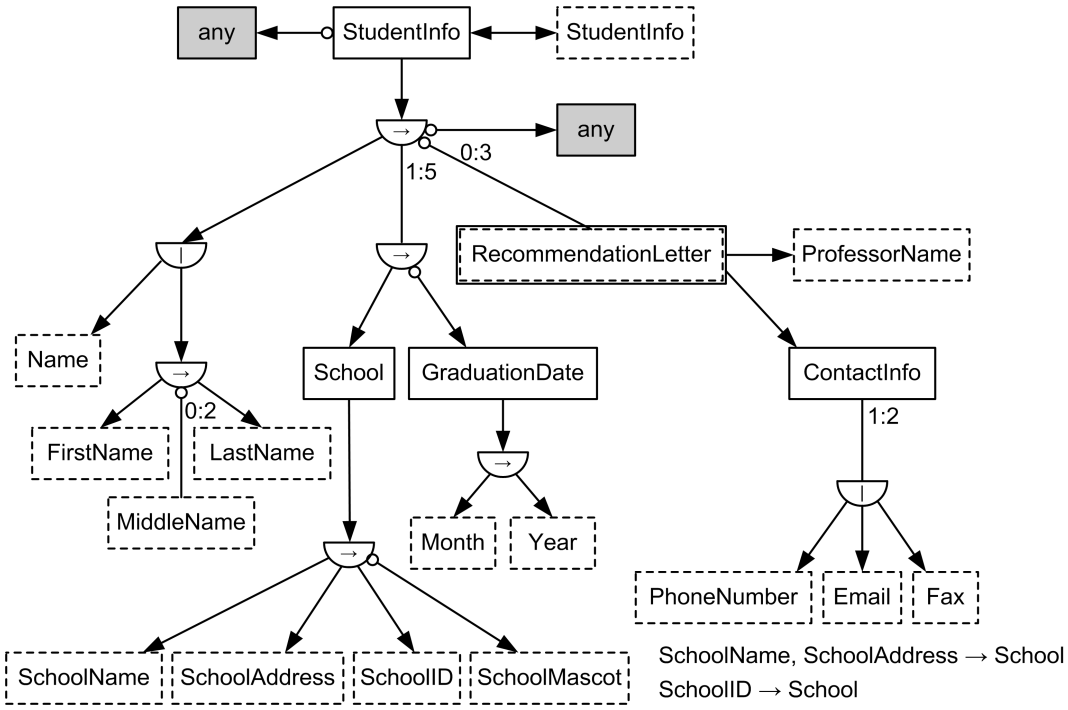


Figure 4.2: Sequence/Choice Structures for Figure 4.1.

to an n-place predicate. Each constraint (e.g. a participation constraint) maps to a closed predicate-calculus formula. In the appendix of this chapter we provide formal representations for the added features for C-XML: sequence, choice, mixed content, and general co-occurrence constraints.

- *Structure independence.* XML in general, and XML Schema in particular, are strongly hierarchical in nature. C-XML is capable of representing the hierarchical aspect of XML Schema, but C-XML is more general, flexible, and conceptual. For example, C-XML allows multiple sequence and choice structures to be associated directly with a single concept (XML Schema allows only one sequence or choice structure for the content of an element). Also, C-XML supports the intermixing of ordinary relationship sets with sequence and choice structures. From this conceptual structure, we can derive many possible hierarchical representations. Similarly, C-XML defines generalized versions of the concepts of

sequence, choice, and mixed content. C-XML provides a conceptual perspective that is structurally independent of XML Schema.

- *Reflection of the mental model.* Given its structure independence and generality, C-XML is well suited to reflect the mental model (design) of a modeler. C-XML can represent hierarchical and non-hierarchical structure. Conceptually, whether a concept is an attribute or an element does not matter, and C-XML does not distinguish between them. C-XML is also able to represent both sequences among related entities and non-sequences among related entities. Choices among alternative related entities are also possible, and choice is distinct from generalization/specialization so that neither is overloaded. C-XML supports mixed content and open content. Finally, C-XML provides for all XML cardinality constraints; indeed it provides for a very large spectrum of cardinality constraints [24] encompassing and going beyond those provided by XML.

- *N-ary relationship sets.* C-XML supports n-ary relationship sets, ($n \geq 2$).

- *Views.* High-level object sets constitute a formal view mechanism, as do high-level relationship sets [19]. As described above, C-XML also can represent both hierarchical and non-hierarchical views.

- *Logical level mapping.* We have implemented automatic conversions from XML Schema to C-XML and vice versa.

- *Constraints.* C-XML supports several kinds of constraints: set constraints, referential-integrity constraints, cardinality constraints, and general constraints.

- *Cardinality for all participants.* C-XML goes further than XML Schema, even allowing cardinality constraints for children of a sequence or choice.

- *Ordering.* C-XML explicitly supports ordering with its sequence construct.

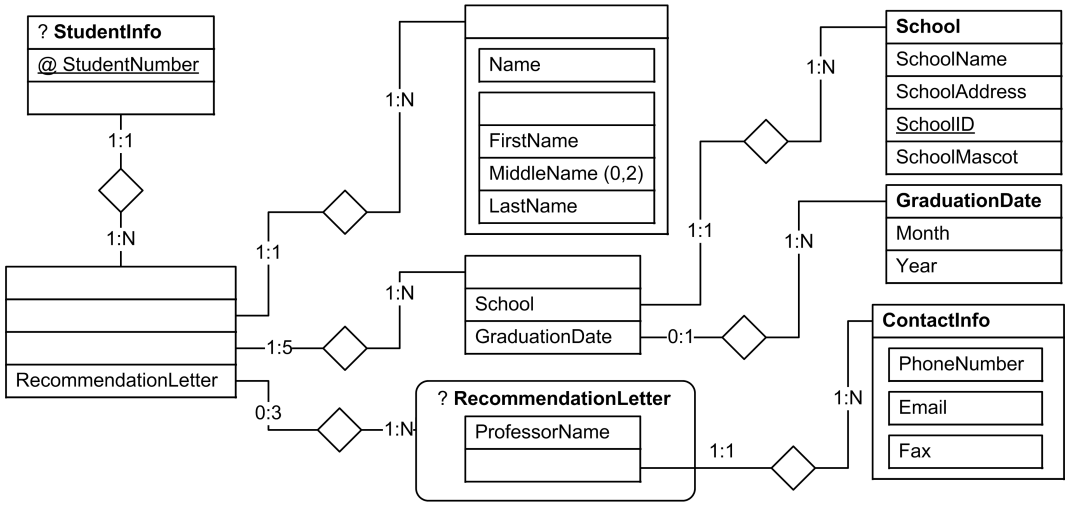


Figure 4.3: Best Representation of Figure 4.1 using XER Notation.

- *Irregular and heterogeneous structure.* The features that give C-XML its structure independence (described above) provide for the modeling of irregular and heterogeneous structure.

- *Document-centric data.* C-XML is able to represent both mixed content and open content.

4.5 Augmenting ER and UML

A number of conceptual modeling languages for XML Schema have been described in the literature. Sengupta and Mohan [33] and Necasky [28] present fairly recent surveys. As we explain in this section, however, most of these efforts do not support the full generality of XML Schema.

4.5.1 ER

Sengupta et al. [34] propose XER as an extension to the ER model for XML. Figure 4.3 shows an example of XER; in fact, it shows the best that can be done to represent the XML schema in Figure 4.1. As we will see, it does not capture all the concepts and constraints in the XML schema in Figure 4.1.

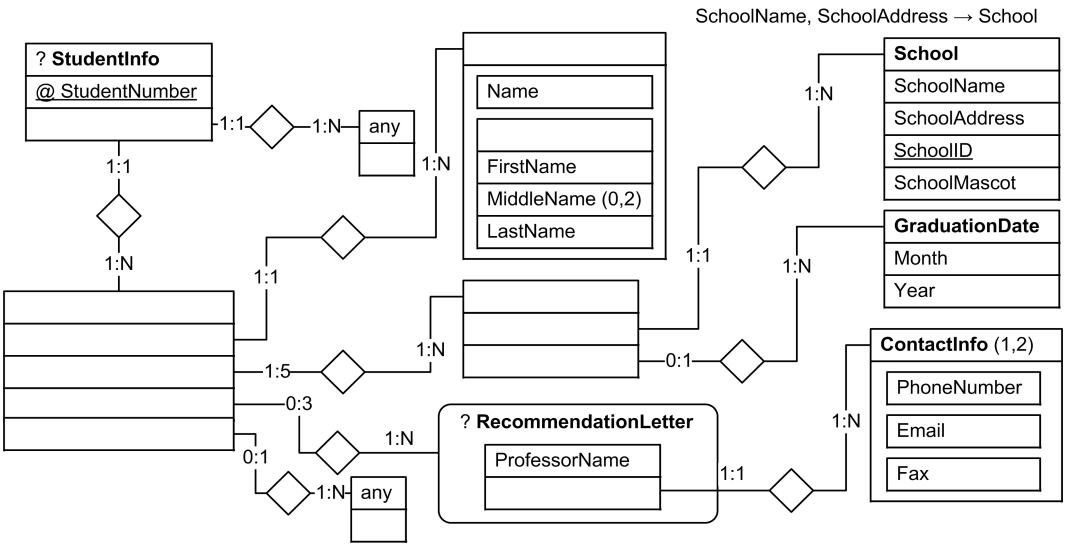


Figure 4.4: Possible Way to Represent XML Schema Document in Figure 4.1 in ER-XML.

XER represents an entity such as *StudentInfo* or *GraduationDate* using a rectangle with a title area giving the name of the entity and the body giving the attributes. For example, in Figure 4.1, *Month* and *Year* are sequenced elements nested under the element *GraduationDate*, so in Figure 4.3 *Month* and *Year* are represented as attributes for the *GraduationDate* entity. Multi-valued attributes are also allowed; their multiplicity constraints are in parentheses. *MiddleName*, for example, is a multi-valued attribute with a multiplicity *(0,2)*. XML attributes in an XER entity are prefixed with @, and key attributes are underlined. The attribute *StudentNumber* is a key in Figure 4.1, so in Figure 4.3 it appears as an underlined attribute with a prefix of @.

An XER entity can be ordered or unordered. Additionally, an XER entity can be mixed.

- Ordered Entity. XER entities are ordered by default from top to bottom. The ordered entity *GraduationDate* in Figure 4.3 indicates that its attributes are ordered first *Month*, then *Year*.

- Unordered Entity. An unordered entity is represented by placing a question mark (?) in front of the entity name. *StudentInfo* in Figure 4.3 is an unordered entity.

- Mixed Entity. A mixed entity is represented in XER using a rounded rectangle. *RecommendationLetter* in Figure 4.3 is a mixed entity.

XER relationships denote a connection between two or more entities, but in XER they can also denote that a complex entity contains a complex element as one of its sub-elements. When an entity E in XER has an attribute A and this attribute A by itself is an entity that contains other attributes, then A appears in the XER diagram twice, once as an attribute inside the entity E, and once as an entity A. In addition, there is a connection between the attribute A inside the entity E and the entity A. If *minA:maxA* is the participation constraint on A within E and *minE:maxE* is the participation constraint on E for A, *minA:maxA* appears on the side of the attribute A within E, and *minE:maxE* appears on the side of the entity A. For example, *RecommendationLetter* has two attributes *ProfessorName* and *ContactInfo*, but *ContactInfo* by itself is an entity. Thus, a relationship set appears between the attribute *ContactInfo* inside *RecommendationLetter* and the entity *ContactInfo*. A participation constraint of *1:1* appears on the side of the attribute *ContactInfo* inside *RecommendationLetter* to denote that *RecommendationLetter* has one *ContactInfo*, and a participation constraint of *1:N* appears on the *ContactInfo* entity side to denote that *ContactInfo* is for one or more *RecommendationLetter*s.[2]

XER represents the choice concept in XML Schema as a generalization/specialization. The generalization term in XER refers to the concept of having an entity that can have different specialization entities in an *is-a* relationship. XER represents a generalization using a covering rectangle containing the specialized XER entities. This, the authors claim in [34], is equivalent to using the "xs:choice" tag in XML Schema. In Figure 4.3 the rectangle representing the entity *ContactInfo* contains the rectangles of entities of choice elements *PhoneNumber*, *Email*, and *Fax*.

[2]Although ER more commonly uses look-across cardinality constraints, the designers of XER have chosen to use participation constraints [34].

Comparing the conceptual components for C-XML (e.g. Figure 4.1) and XER (e.g. Figure 4.3), we see that several constructs and constraints are missing in XER. First, XER lacks the ability to represent the minimum and the maximum occurrence of the whole sequence or choice within a containing entity when either of their values is more than *1*. For example, XER cannot represent the minimum and maximum occurrence of *1* to *2* for the *choice* within the entity *ContactInfo*. Second, XER has no representation for *any* and *anyAttribute* structures. For example, in Figure 4.3 the entity *StudentInfo* is missing the *anyAttribute*, and the sequence contained inside the *StudentInfo* entity does not have *any*. Third, XER has no representation for composite keys. For example, in Figure 4.3 the representation that *SchoolName* and *SchoolAddress* together constitute a key for the entity *School* is missing. Fourth, although XER has a representation for a single key, this representation only applies when the key for an entity is an attribute of that entity. The representation is not able to specify a key constraint for an entity within the context of another entity.

Beyond these omissions, we have several concerns about some representations in XER.

- Representing *choice* by generalization/specialization is problematic; the formal definition of *choice* differs from the formal definition of generalization/specialization. First, *choice* contains different types of alternative concepts, but all the specialized concepts in generalization/specialization hierarchies typically must have the same type. Second, in generalization/specialization hierarchies any specialized concept inherits relationship sets from its generalization concepts, while in *choice*, the alternative concepts do not inherit relationship sets. Third, the participation constraints for *choice* allow alternative concepts to appear more than one time, while in generalization/specialization hierarchies specialized concepts can appear at most once.

- In XER it is not clear from [34] whether it is possible to represent an entity without having a name for the entity. For Figure 4.3 we assume that we are able to represent an entity in XER with a null name. Also, in XER it is not

clear whether it is possible to have an empty slot in an entity to indicate that an attribute by itself is an entity without a name. We also assume for Figure 4.3 that we are able to do so in XER. From [34] it is not clear whether it is possible to have hierarchies of *choice* and *sequence* structures, but we assume that this is possible as Figure 4.3 shows.

- In XER when an entity has an attribute and this attribute is also an entity, the model instance in XER has an attribute and an entity with the same name. This redundancy might cause problems if XER developers are able to write the two names independently.

In light of these omissions and concerns, we extend XER, augmenting it with constructs and constraints that are missing and resolving our concerns. Figure 4.4 shows our suggested way of representing the schema in Figure 4.1 in ER-XML, our ER augmentation for XML. We add *any* and *anyAttribute* concepts to XER. We have chosen to add a representation of *any* and *anyAttribute* as entities with the name *any*. We also add minimum and maximum occurrence to *sequence* and *choice*, placing this minimum and maximum in parentheses in the name slot, following the name, if any, of the entity that declares the *sequence* or *choice*. We have chosen to add a representation for key constraints by allowing functional dependencies that must hold within entity sets or along paths of relationship sets. Thus, for example, as Figure 4.4 shows, we can specify the composite key *SchoolName*, *SchoolAddress* by the functional dependency *SchoolName*, *SchoolAddress* $\longrightarrow$ *School*. Although we use the same notation for *choice*, we do not consider the representation of *choice* in ER-XML to be a generalization concept. Finally, we do not repeat attribute names, writing the name only in the entity that represents the attribute.

4.5.2 UML

Conrad et al. [14] add features to UML to enable mappings from class diagrams to XML DTDs. Figure 4.5 shows an example; in fact, it shows the best that can be

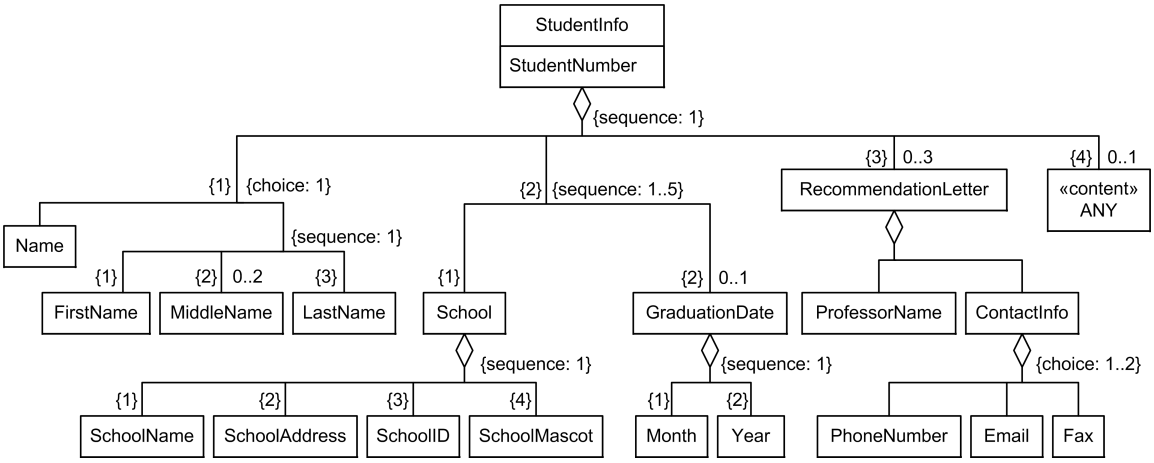


Figure 4.5: Best Representation of Figure 4.1 Using Conrad Notation.

done to represent the XML schema in Figure 4.1. Unfortunately, it does not capture all the concepts and constraints in the XML schema in Figure 4.1.

As described in [14], Conrad et al. augment UML aggregation so that it can be transformed into a *sequence* construct or a *choice* construct. The designation {*sequence*} specifies a left-to-right ordering of elements, and the designation {*choice*} specifies a choice among elements. For a sequence the first constituent element is marked as *1*, the second as *2*, and so forth. A *sequence* or *choice* construct may have cardinality to represent the minimum and maximum occurrence of the entire sequence or choice. For example, the class *ContactInfo* in Figure 4.5 has one to two choices {*choice* : 1..2} of the classes *PhoneNumber*, *Email*, and *Fax*. For an *any* structure, the notation in [14] uses the «*content*» stereotype.

Comparing the conceptual components for C-XML (e.g. Figure 4.1) and extended UML presented in [14] (e.g. Figure 4.5), we see that several constructs are missing. First, the extended UML in [14] lacks the ability to represent an *anyAttribute*. For example, in Figure 4.5 the class *StudentInfo* is missing the *anyAttribute*. Second, the extended UML in [14] lacks the ability to represent mixed content. In Figure 4.5 the class *RecommendationLetter* does not appear as having mixed content. Third, the extended UML in [14] lacks key constraints, although, in principle, we could specify key constraints using OCL (the constraint language of UML).

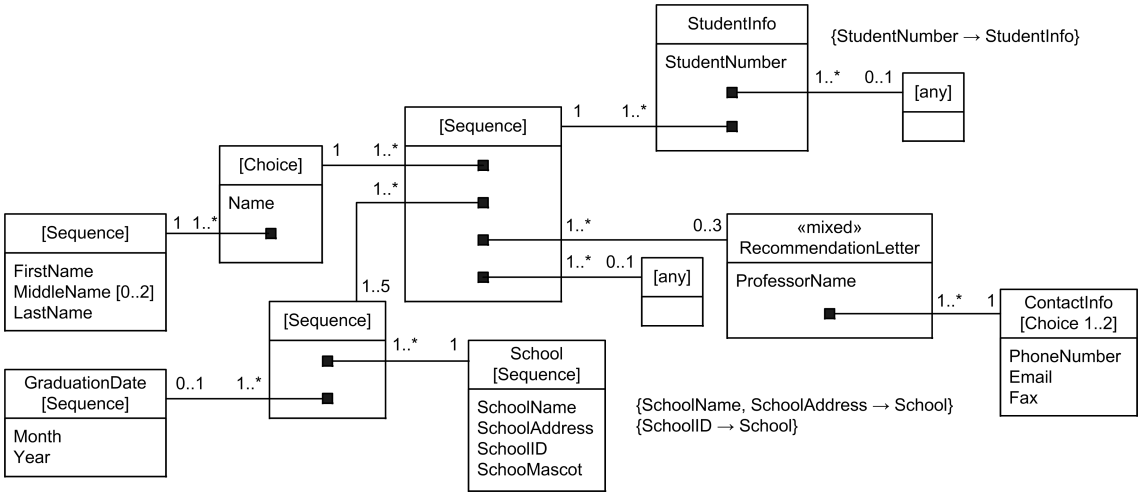


Figure 4.6: Possible Way to Represent XML Schema Document in Figure 4.1 in UML-XML.

Besides these omissions, we have concerns about the suggested representation of sequence and choice in [14]. The suggested representations can only be applied between classes, not between attributes. This is because Conrad et al. augment UML aggregation for *sequence* and *choice*. Since the aggregation in UML applies to classes, the notation forces attributes to be represented as classes. For example, to represent the *GraduationDate* class as a sequence of *Month* and *Year*, would-be attributes *Month* and *Year* must each become a class first.

To overcome these difficulties, we need to extend and adjust the representations in [14]. Figure 4.6 shows our suggested extensions and adjustments by rendering Figure 4.1 in UML-XML, our UML augmentation for XML.

- We have chosen to represent the *anyAttribute* as an associated class with the *any* content type rather than as a stereotype. For mixed content we use the «*mixed*» stereotype. The *RecommendationLetter* class in Figure 4.6 is an example.

- We suggest representing *sequence* and *choice* in a different way so that we do not force attributes to be represented as classes. When attributes in a class are ordered, we add the designation [*Sequence*] under the class name

to specify a top-to-bottom ordering of the attributes. We also add *minOccurs..maxOccurs*, if needed, to express participation different from the default. For example, in Figure 4.6, the designation [*Sequence*] is added under *GraduationDate*. Similarly, we allow designating a choice construct by adding [*Choice minOccurs..maxOccurs*], allowing *minOccurs .. maxOccurs* to be omitted when it is *1..1*, the default. For example, in Figure 4.6, the designation [*Choice 1..2*] is added under *ContactInfo*.

- We add notation to denote that a class contains an attribute and that this attribute is a class that contains other attributes. A connection appears that connects an empty slot indicating the presence of an attribute inside the class with the class containing other attributes. For example, we indicate that *ContactInfo* is an attribute inside the class *RecommendationLetter* by the connection inside *RecommendationLetter* that extends to *ContactInfo*. Note also that *ContactInfo* by itself is a class that contains attributes. A multiplicity of *1* is added to the *ContactInfo* class side and a multiplicity of *1..** is added to the *ContactInfo* attribute side in the *RecommendationLetter* class to denote that *ContactInfo* is for *1* or more *RecommendationLetter*s and each *RecommendationLetter* has one *ContactInfo*.

- For the case when a sequence or choice is a complex attribute inside a class *C*, the *sequence* or *choice* is represented as a class with no name but has the designation [*Sequence*] or [*Choice*], and we connect the empty slot inside the class *C* with the class that represents the *sequence* or *choice*. For example, the class *StudentInfo* has a complex sequence attribute. Further, this sequence by itself is a class that contains other attributes including another complex choice attribute and a complex sequence attribute.

- We can specify key constraints in UML by using OCL. However, since this is a common task, we have an alternative representation that we can add to a diagram. We have chosen to add a representation for key constraints by allowing

functional dependencies which must hold within classes or along paths of associations. Thus, for example, as Figure 4.6 shows, we can specify the composite key *SchoolName*, *SchoolAddress* by the functional dependency {*SchoolName*, *SchoolAddress* $\longrightarrow$ *School*}.

4.5.3 ER-XML, UML-XML, and C-XML

Comparing ER-XML, UML-XML, and C-XML, we make the following observations according to the criteria for XML conceptual modeling we described in Section 4.2. Criteria from Section 4.2 not listed here have equal validity among the three models (e.g. all three have a *graphical notation*).

- *Formal foundation.* C-XML has a solid formal foundation in terms of predicate calculus. ER-XML and UML-XML are respectively derived from XER as described in [34] and UML as described in [14]. There is no formal foundation for XER [28], and the underlying formalism of UML is not fully developed [22]. In principle both could have complete formal foundations.

- *Reflection of the mental model.* ER-XML distinguishes attributes from entities and UML-XML distinguishes attributes from classes. C-XML represents all concepts as object-set nodes in hypergraphs. Forcing attributes to be embedded within an entity/class has the disadvantage that a user of UML-XML or ER-XML has to decide before representing any concept whether it should be an attribute or entity/class. Distinguishing between an attribute and an entity/class is not necessary and may even be harmful as a mental-model conceptualization. Goldstein and Storey [23] showed that this can be a major source of errors in conceptual modeling.

- *Views.* Hypergraphs are typically more amenable to translations to various views and even alternate XML schemas such as normalized XML schemas. Further, although not discussed here, C-XML supports both high-level object sets and high-level relationship sets as first class concepts [19]. Neither ER-XML nor UML-XML supports high-level view constructs.

- *Logical level mapping.* We have implemented both a mapping from XML Schema to C-XML and vice versa [1, 2]. In principle mappings to and from XML Schema and ER-XML as well as UML-XML are possible.

- *Cardinality for all participants.* The nesting representation for ER-XML and UML-XML restricts the specification of cardinality constraints to only the nesting participants. C-XML specifies cardinality constraints for all participants, beyond even those supported by XML Schema.

4.6 Conclusion

In this chapter we discussed the structural constructs in XML Schema that are missing in traditional conceptual models. Our proposed solution is to enrich conceptual models with the ability to order a list of concepts, choose alternative concepts from among several, specify mixed content, and use content from another data model. We presented our solution using C-XML, and we showed that our solution can be adapted and used for the ER and UML languages.

We also presented requirements for conceptual modeling for XML. We based these requirements on those presented in [28], [35], and [42]. We evaluated C-XML against these requirements and showed that C-XML satisfies all of them, which makes C-XML a good candidate for a conceptual modeling language for XML. We also argued that ER-XML and UML-XML, our adaptations for ER and UML, also largely satisfy these requirements, but do not satisfy them as well as does C-XML.

We have implemented C-XML, and we have implemented conversions from XML Schema to C-XML and vice versa. Currently, we are working on a formal proof that our conversions to and from C-XML and XML Schema preserve information and constraints.

4.7 Appendix

4.7.1 Sequence

Figure 4.7 shows the schematic structure of a sequence. Exactly one parent object set connects to a sequence of n children, $n \geq 0$, with participation constraints on the several connections as Figure 4.7 shows. A sequenced child may be either an object set or a nested sequence or choice structure. In general, there may be many sequences in a model instance, and since we do not explicitly name sequence structures, we denote a particular sequence, the kth sequence, by $Sequence_k$. Let P be the name of the parent object set for $Sequence_k$, and let C_1, ..., C_n be the names of the n child object sets or nested sequences or choices that are sequenced within $Sequence_k$. To impose order, we introduce the unary predicate *Order*, which we can think of as an object set containing as many ordinal numbers as we need 1, 2, ... We denote the minimum and maximum cardinalities of $Sequence_k$ according to Figure 4.7. Let min and max be, respectively, the minimum and maximum number of occurrences of $Sequence_k$ allowed for an object in P. Let min_{C_i} and max_{C_i}, $1 \leq i \leq n$, be, respectively, the minimum and maximum number of allowed occurrences of C_i objects within $Sequence_k$. Let min' and max' be, respectively, the minimum and maximum number of occurrences of $Sequence_k$ sequences in the relationship set between P and $Sequence_k$. Finally, let min_{Seq_i} and max_{Seq_i}, $1 \leq i \leq n$, be, respectively, the minimum and maximum allowed occurrences of $Sequence_k$ in the relationship set between $Sequence_k$ sequences and C_i (i.e. the number of C_i objects that can be associated with $Sequence_k$ for a given order position). For $Sequence_k$, we have the following object sets, relationship sets, and constraints.

- Object Sets:
  - $P(x)$
  - $Sequence_k(x)$
  - $Order(x)$
  - $C_1(x)$, ..., $C_n(x)$

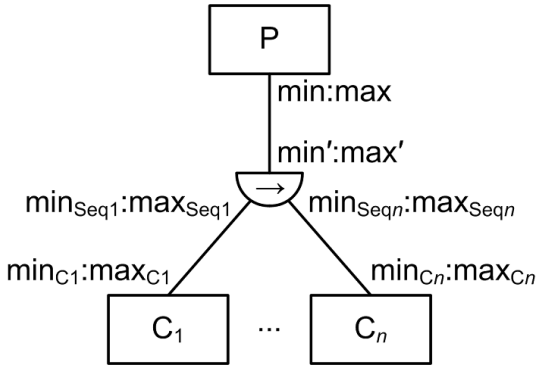


Figure 4.7: Sequence Structure in C-XML.

- Relationship Sets:
  - $P(x)$ *contains* $Sequence_k(y)$
  - $C_1(x)$ *has* $Order(1)$ *in* $Sequence_k(y)$
  - ...
  - $C_n(x)$ *has* $Order(n)$ *in* $Sequence_k(y)$
- Referential Integrity:
  - $\forall x \forall y(P(x)$ *contains* $Sequence_k(y) \Rightarrow P(x) \wedge Sequence_k(y))$
  - $\forall x \forall y(C_1(x)$ *has* $Order(1)$ *in* $Sequence_k(\mathrm{y}) \Rightarrow C_1(x) \wedge Order(1) \wedge Sequence_k(y))$
  - ...
  - $\forall x \forall y(C_n(x)$ *has* $Order(n)$ *in* $Sequence_k(y) \Rightarrow C_n(x) \wedge Order(n) \wedge Sequence_k(y))$
- Participation Constraints:
  - $\forall x(P(x) \Rightarrow \exists^{\geq min} y(P(x)$ *contains* $Sequence_k(y))) \wedge$
    $\forall x(P(x) \Rightarrow \exists^{\leq max} y(P(x)$ *contains* $Sequence_k(y)))$
  - $\forall x(Sequence_k(x) \Rightarrow \exists^{\geq min'} y(P(y)$ *contains* $Sequence_k(x))) \wedge$
    $\forall x(Sequence_k(x) \Rightarrow \exists^{\leq max'} y(P(y)$ *contains* $Sequence_k(x)))$
  - $\forall x(Sequence_k(x) \Rightarrow$
    $\exists^{\geq min_{Seq_1}} y_1(C_1(y_1)$ *has* $Order(1)$ *in* $Sequence_k(x))$

$\wedge$ ... $\wedge$

$\exists^{\geq min_{Seq_n}} y_n(C_n(y_n)\ has\ Order(n)\ in\ Sequence_k(x)))$

$\wedge$

$\forall x(Sequence_k(x) \Rightarrow$

$\exists^{\leq max_{Seq_1}} y_1(C_1(y_1)\ has\ Order(1)\ in\ Sequence_k(x))$

$\wedge$ ... $\wedge$

$\exists^{\leq max_{Seq_n}} y_n(C_n(y_n)\ has\ Order(n)\ in\ Sequence_k(x)))$

– $\forall x(C_1(x) \Rightarrow \exists^{\geq min_{C_1}} y(C_1(x)\ has\ Order(1)\ in\ Sequence_k(y))) \wedge$
$\forall x(C_1(x) \Rightarrow \exists^{\leq max_{C_1}} y(C_1(x)\ has\ Order(1)\ in\ Sequence_k(y)))$
$\wedge$ ... $\wedge$
$\forall x(C_n(x) \Rightarrow \exists^{\geq min_{C_n}} y(C_n(x)\ has\ Order(n)\ in\ Sequence_k(y))) \wedge$
$\forall x(C_n(x) \Rightarrow \exists^{\leq max_{C_n}} y(C_n(x)\ has\ Order(n)\ in\ Sequence_k(y)))$

4.7.2 Choice

The schematic structure of a choice is similar to sequence (see Figure 4.7). Exactly one parent object set connects to a group of n children, $n \geq 0$, and the participation constraints for choice are similar to those for sequence. As with sequence, children of a choice may be either object sets or nested sequence or choice structures. In general, there may be many choices in a model instance, and since we do not explicitly name choice structures, we denote a particular choice, the kth choice, by $Choice_k$. Let P be the name of the parent object set for $Choice_k$, and let C_1, ..., C_n be the names of the n child object sets or nested sequences or choices that are alternatives for $Choice_k$. Let min and max be, respectively, the minimum and maximum number of occurrences of $Choice_k$ allowed for an object in P. Let min_{C_i} and max_{C_i}, $1 \leq i \leq n$, be, respectively, the minimum and maximum number of allowed occurrences of C_i objects within $Choice_k$. Let min' and max' be, respectively, the minimum and maximum number of occurrences of $Choice_k$ choices in the relationship set between P and $Choice_k$. Finally, let min_{Cho_i} and max_{Cho_i}, $1 \leq i \leq n$, be, respectively, the minimum and maximum allowed occurrences of $Choice_k$ choices in the

relationship set between $Choice_k$ and C_i. For $Choice_k$, we have the following object sets, relationship sets, and constraints.

- Object Sets:
  - $P(x)$
  - $Choice_k(x)$
  - $C_1(x)$, ..., $C_n(x)$
- Relationship Sets:
  - $P(x)$ *contains* $Choice_k(y)$
  - $C_1(x)$ *is alternative for* $Choice_k(y)$
  - ...
  - $C_n(x)$ *is alternative for* $Choice_k(y)$
- Referential Integrity:
  - $\forall x \forall y(P(x)$ *contains* $Choice_k(y) \Rightarrow P(x) \wedge Choice_k(y))$
  - $\forall x \forall y(C_1(x)$ *is alternative for* $Choice_k(\mathrm{y}) \Rightarrow C_1(x) \wedge Choice_k(y))$
  - ...
  - $\forall x \forall y(C_n(x)$ *is alternative for* $Choice_k(y) \Rightarrow C_n(x) \wedge Choice_k(y))$
- Participation Constraints:
  - $\forall x(P(x) \Rightarrow \exists^{\geq min} y(P(x)$ *contains* $Choice_k(y))) \wedge$
    $\forall x(P(x) \Rightarrow \exists^{\leq max} y(P(x)$ *contains* $Choice_k(y)))$
  - $\forall x(Choice_k(x) \Rightarrow \exists^{\geq min'} y(P(y)$ *contains* $Choice_k(x))) \wedge$
    $\forall x(Choice_k(x) \Rightarrow \exists^{\leq max'} y(P(y)$ *contains* $Choice_k(x)))$
  - $\forall x(Choice_k(x) \Rightarrow$
    $(\exists^{\geq min_{Cho_1}} y_1\ (C_1(y_1)$ *is alternative for* $Choice_k(x)) \wedge$
    $\exists^{\leq max_{Cho_1}} z_1\ (C_1(z_1)$ *is alternative for* $Choice_k(x))) \vee$

$\neg\exists w_1\ (C_1(w_1)\ \textit{is alternative for}\ Choice_k(x)))$

$\wedge\ ...\ \wedge$

$\forall x(Choice_k(x) \Rightarrow$

$(\exists^{\geq min_{Chon}} y_n(C_n(y_n)\ \textit{is alternative for}\ Choice_k(x)) \wedge$

$\exists^{\leq max_{Chon}} z_n(C_n(z_n)\ \textit{is alternative for}\ Choice_k(x))) \vee$

$\neg\exists w_n\ (C_n(w_n)\ \textit{is alternative for}\ Choice_k(x)))$

– $\forall x(C_1(x) \Rightarrow \exists^{\geq min_{C_1}} y(C_1(x)\ \textit{is alternative for}\ Choice_k(y))) \wedge$

$\forall x(C_1(x) \Rightarrow \exists^{\leq max_{C_1}} y(C_1(x)\ \textit{is alternative for}\ Choice_k(y)))$

$\wedge\ ...\ \wedge$

$\forall x(C_n(x) \Rightarrow \exists^{\geq min_{C_n}} y(C_n(x)\ \textit{is alternative for}\ Choice_k(y))) \wedge$

$\forall x(C_n(x) \Rightarrow \exists^{\leq max_{C_n}} y(C_n(x)\ \textit{is alternative for}\ Choice_k(y)))$

- Mutual-Exclusion Constraints:

– $\forall x(Choice_k(x) \Rightarrow$

$(\exists y_1(C_1(y_1)\ \textit{is alternative for}\ Choice_k(x)) \wedge$

$\neg\exists y_2(C_2(y_2)\ \textit{is alternative for}\ Choice_k(x))\ \wedge \ldots \wedge$

$\neg\exists y_n(C_n(y_n)\ \textit{is alternative for}\ Choice_k(x)))$

$\vee \ldots \vee$

$(\neg\exists y_1(C_1(y_1)\ \textit{is alternative for}\ Choice_k(x))\ \wedge \ldots \wedge$

$\neg\exists y_{i-1}(C_{i-1}(y_{i-1})\ \textit{is alternative for}\ Choice_k(x)) \wedge$

$\exists y_i(C_i(y_i)\ \textit{is alternative for}\ Choice_k(x)) \wedge$

$\neg\exists y_{i+1}(C_{i+1}(y_{i+1})\ \textit{is alternative for}\ Choice_k(x))\ \wedge \ldots \wedge$

$\neg\exists y_n(C_n(y_n)\ \textit{is alternative for}\ Choice_k(x)))$

$\vee \ldots \vee$

$(\neg\exists y_1(C_1(y_1)\ \textit{is alternative for}\ Choice_k(x))\ \wedge \ldots \wedge$

$\neg\exists y_{n-1}(C_{n-1}(y_{n-1})\ \textit{is alternative for}\ Choice_k(x)) \wedge$

$\exists y_n(C_n(y_n)\ \textit{is alternative for}\ Choice_k(x))))$

4.7.3 Mixed Content

Formally, marking an object set P as *mixed* is a template for creating a relationship set to a lexical object set $Text$ of type string: $P[1]$ *contains* $[1{:}*]Text$. The string associated with an object in a mixed object set may be interspersed among direct child elements.

4.7.4 Generalized Co-Occurrence:

A generalized co-occurrence constraint $A_1, ..., A_n \rightarrow B_1, ..., B_m$ is shorthand for an ordinary co-occurrence constraint written over a high-level relationship set connecting the object sets $A_1, ..., A_n, B_1, ..., B_m$. If the subgraph that connects these object sets is unique, we can derive the corresponding high-level relationship set automatically. Otherwise, in addition to specifying the co-occurrence constraint, the user must also specify the derived relationship set using Prolog-like syntax (e.g., $r(A, B)$:- $r_1(A, X), r_2(X, B)$). The formal definition of co-occurrence constraints appears in Appendix A of [19].

Chapter 5

Translating XML Schema to Conceptual XML

5.1 Introduction

Recently, many organizations have begun to store their data using XML, and XML Schema has become the preeminent mechanism for describing valid XML document structures. Moreover, the number of applications that use XML as their native data model has increased. This increases the need for well-designed XML data models and the need for a conceptual model for designing XML schemas. The objective in this chapter is to translate any given XML-Schema instance to Conceptual XML (C-XML). We describe in detail the translation of all XML-Schema components to conceptual model (C-XML) elements.

The basic translation strategies are straightforward.[1] In translating from XML Schema to C-XML, elements and attributes become object sets. Elements that have simple types become lexical object sets, while elements that have complex types become nonlexical 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.

Following [45], we discuss the following components of XML Schema in the sections indicated: *schema* (Section 5.2.1), *element* (Section 5.2.2), *attribute* (Section 5.2.3), identity-constraint definitions (*key*, *unique*, and *keyref*) (Section 5.2.4), *simpleType* (Section 5.2.5), *complexType* (Section 5.2.6), *attributeGroup* (Section 5.2.7), *all* (Section 5.2.8), *sequence* (Section 5.2.9), *choice* (Section 5.2.10), *any* (Section 5.2.11),

[1] Some require a deep understanding of C-XML, the OSM Object-Relationship Model, and XML Schema. We assume that the reader has the necessary background.

anyAttribute (Section 5.2.11), *group* (Section 5.2.12), simple-content complex type (Section 5.2.13), and complex-content complex type (Section 5.2.14).

In the translation we are concerned about conceptualizing XML Schema, and thus we are only interested in representing the conceptual components of XML Schema. The XML Schema components *annotation*, *notation*, *include*, *redefine*, and *import* are not conceptual constructs, but we could include them in trivial ways. For example, we could map *annotation* and *notation* as comments. Since *include* and *import* add schemas to the XML-Schema instance, we could insert the actual added schemas to the XML-schema instance and then map the content of that XML-Schema instance to C-XML. Since the *redefine* component redefines simple and complex types, groups, and attribute groups from an external schema, we can simply insert the newly defined components in the XML-Schema instance and then map the new content of that XML-Schema instance to C-XML.

5.2 Translation Details

In this section we explain in detail the translation of each conceptual component in XML Schema to C-XML. For each component we give (1) the component declaration definition from [45], (2) an XML-Schema instance exemplifying the component, and (3) a C-XML instance illustrating the translation. As necessary to complete the discussion of some components, we give additional XML-Schema instance examples and their translation to C-XML. To explain the translation and to show that all component parts are translated, we list each component part, explain its translation, and show its translation in the given examples. When the content part in the component definition contains other XML-Schema components, we explain these other components in the sections in which their definition appears.

5.2.1 Schema

Figure 5.1 shows definition details for *schema* which is the root element of any document that conforms to W3C XML Schema. A *schema* component serves two purposes: (1) it establishes default values and (2) it records basic meta information. We use the default specification in a preprocessing step and then discard these statements. We store the basic meta information directly as header meta information in C-XML.

In the following, we give the translation details of the component parts of *schema*.

```
<schema
 id = ID
 attributeFormDefault = (‘qualified’ | ‘unqualified’) : ‘unqualified’
 blockDefault = (‘#all’ | List of (‘extension’ |‘restriction’
                | ‘substitution’)) : ‘‘’’
 elementFormDefault = (‘qualified’ | ‘unqualified’) : ‘unqualified’
 finalDefault = (‘#all’ | List of (‘extension’ | ‘restriction’)) : ‘‘’’
 targetNamespace = anyURI
 version = token
 xml:lang = language>
 Content: ((include | import | redefine | annotation)*,
          (((simpleType | complexType | group | attributeGroup)
          | element | attribute | notation), annotation*)*)
</schema>
```

Figure 5.1: Schema Declaration.

- *id*. The *id* plays no role in the translation. We simply discard it.
- *attributeFormDefault*. The preprocessing step for *attributeFormDefault* consists of considering each *attribute* in the XML-Schema instance. If no value is assigned to the *form* attribute, we assign it the default value specified in *attributeFormDefault*.
- *elementFormDefault*. The preprocessing step for *elementFormDefault* consists of considering each *element* in the XML-Schema instance. If no value is assigned to the *form* attribute, we assign it the default value specified in *elementFormDefault*.
- *blockDefault* and *finalDefault*. The preprocessing step for both *blockDefault* and *finalDefault* consists of considering each *element* and *complexType* in the XML-Schema instance. If no value is assigned to the *block* or *final* attribute, we assign it the default value specified in its corresponding attribute *blockDefault* or *finalDefault*.
- *targetNamespace*, *version*, and *xml:lang*. The value of the attribute *targetNamespace* is recorded in the specified place in the C-XML instance for *targetNamespace*. The value of the attribute *version* is recorded in the specified place for *version*. The value of the attribute *xml:lang* is recorded in the specified place for *xml:lang*.

5.2.2 Element

Figure 5.2 shows definition details for *element* in XML Schema. Figure 5.3 shows an example using *element* in an XML-Schema instance, and Figures 5.4–5.6 show a C-XML translation of Figure 5.3. Figures 5.7 and 5.8 show an additional example to demonstrate *abstract* and *substitutionGroup* attributes in the definition of *element*.

```
<element id = ID
 abstract = boolean : 'false'
 block = ('#all' | List of ('extension' | 'restriction' | 'substitution'))
 default = string
 final = ('#all' | List of ('extension' | 'restriction'))
 fixed = string
 form = ('qualified' | 'unqualified')
 maxOccurs = (nonNegativeInteger | 'unbounded') : 1
 minOccurs = nonNegativeInteger : 1
 name = NCName
 nillable = boolean : 'false'
 ref = QName
 substitutionGroup = QName
 type = QName>
 Content: (annotation?, ((simpleType | complexType)?, (unique | key | keyref)*))
</element>
```

Figure 5.2: Element Declaration.

We translate an element in XML Schema to an object set that has the same name as the element in XML Schema. In the following, we give the translation details of the component parts of *element*.

- *id*. The *id* is not a conceptual component; it exists implicitly. Thus we do not map it to C-XML.
- *name*. The *name* of the element becomes the name of the object set representing that element in C-XML. For example, the object set in C-XML that represents the element B declared in Line 6 in Figure 5.3 has the name B in Figure 5.4.
- *minOccurs* and *maxOccurs*. The default value for both *minOccurs* and *maxOccurs* is 1. Whether by default or explicitly given, *minOccurs* and *maxOccurs* become participation constraints. They are assigned to the connection between

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
 3    <xs:element name="A">
 4       <xs:complexType>
 5          <xs:sequence>
 6             <xs:element name="B" type="BType"/>
 7             <xs:element ref="C" minOccurs="1" maxOccurs="3"/>
 8          </xs:sequence>
 9       </xs:complexType>
10    </xs:element>
11    <xs:complexType name="BType">
12       <xs:sequence>
13          <xs:element name="B1" type="xs:string" minOccurs="1" maxOccurs="1"/>
14          <xs:element name="B2" type="B2type" minOccurs="0" maxOccurs="2"/>
15          <xs:element name="B3" minOccurs="1" maxOccurs="1">
16             <xs:simpleType>
17                <xs:restriction base="B2type">
18                   <xs:maxLength value="2"/>
19                </xs:restriction>
20             </xs:simpleType>
21          </xs:element>
22       </xs:sequence>
23    </xs:complexType>
24    <xs:element name="C">
25       <xs:complexType>
26          <xs:choice>
27             <xs:element name="C1" type="xs:string" fixed="xy"/>
28             <xs:element name="C2" type="xs:string"/>
29          </xs:choice>
30       </xs:complexType>
31    </xs:element>
32    <xs:simpleType name="B2type">
33       <xs:restriction base="xs:string">
34          <xs:enumeration value="EW"/>
35          <xs:enumeration value="L"/>
36       </xs:restriction>
37    </xs:simpleType>
38 </xs:schema>
```

Figure 5.3: Example of *Element* Structure in XML Schema.

the relationship set and the component that represents the containing element. For example, *minOccurs="0"* and *maxOccurs="2"* for *B2* within the sequence structure in Line 14 become the participation constraint *0:2* which is near the sequence side of the connection between the sequence construct and *B2* in Figure 5.4. In C-XML, for the most common participation constraints (*0:1*, *1:1*, *0:**, and *1:**), we can use the special notation that obviates the need for a participation constraint. *Optional* or *mandatory participation* respectively specify whether objects in a connected relationship may or must participate in a relationship set (an "o" on a connecting relationship-set line designates optional while the absence of an "o" designates mandatory). Arrowheads on lines specify *functional constraints*, which limits participation of objects on the tail side of an arrow to be at most one. Thus, for example, the *minOccurs="0"* of *B2* within the sequence yields an optional constraint for the sequence in the connection

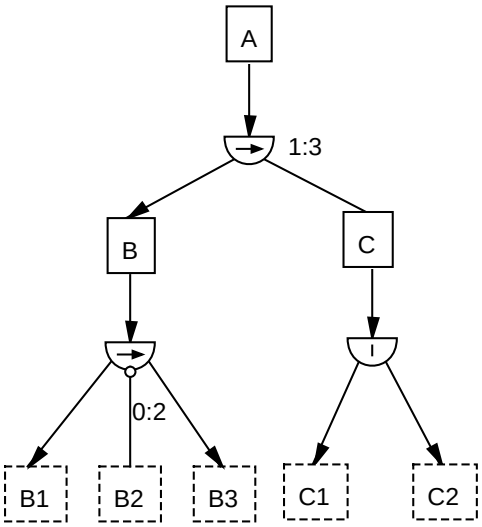


Figure 5.4: Translated C-XML Model Instance of Figure 5.3

between the sequence construct and *B2* in Figure 5.4. The constraints *minOccurs="1"* and *maxOccurs="1"* for *B3* within the sequence structure in Line 15 become a mandatory constraint on the sequence side and functional constraint on the object set *B3* in the connection between the sequence construct and *B3* in Figure 5.4.

- *ref*. The *ref* attribute in XML Schema provides for the use or reuse of a global element in an XML-Schema instance in another local place inside the same XML-Schema instance. In the translation to C-XML, we have an object set that represents the global element. The object set connects into the overall structure in the same way the local element that contains the *ref* attribute connects. For example, element C in Line 7 refers to the global element C that appears in Line 24. In the C-XML instance, the global element C becomes an object set that takes on the relationship set and the constraints the local element in Line 7 has. Thus, the object set C will be connected to the sequence structure (which represents the sequence starting in Line 5 nested under the element A in Line 3). The *minOccurs="1"* and *maxOccurs="3"* in Line 7 appear as a participation constraint on the sequence structure side of the connection to C in C-XML.

- *block*, *default*, *final*, *fixed*, *nillable*, and *form*. These attributes are added to the associated data frame of the object set that represents the element. For example, Figure 5.5 shows the data frame associated with the object set *C1* in

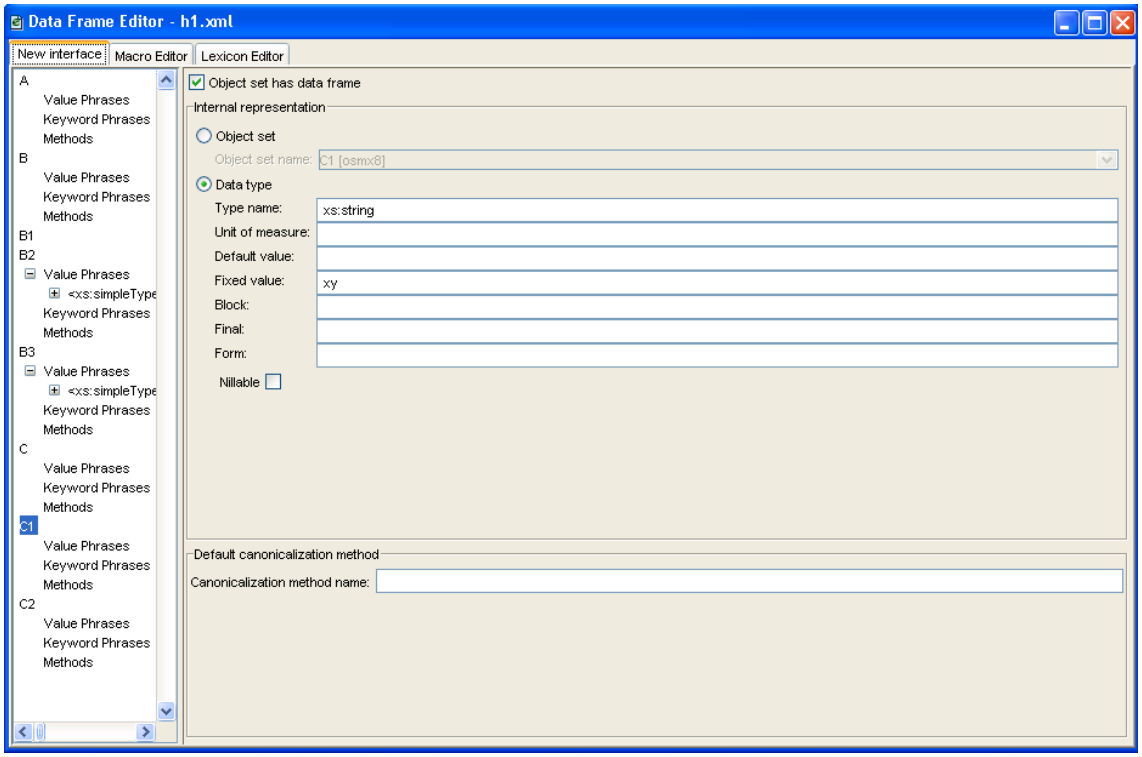


Figure 5.5: Data Frame for Element *C1* in Figure 5.3

Line 27. Observe, for example, that the *fixed* attribute has the value *"xy"*, which appears in the *Fixed value* field in Figure 5.5.

- *type*. The attribute *type* in an element declaration can be any one of the following.
  - *built-in data type*. The element becomes a lexical object set in C-XML. The type is specified in the type name field of the data frame for the associated object set. For example, in Figure 5.3, the element *B1* in Line 13, which has the built-in type *xs:string*, becomes a lexical object set in Figure 5.4. The type *xs:string* of the element *B1* is specified in the type name field of the data frame that associates with the object set *B1* as Figure 5.6 shows.
  - *simple data type*. The element becomes a lexical object set in C-XML, and the type is specified in the value phrase inside the associated data frame as explained in Section 5.2.5. For example, the data frame associated with

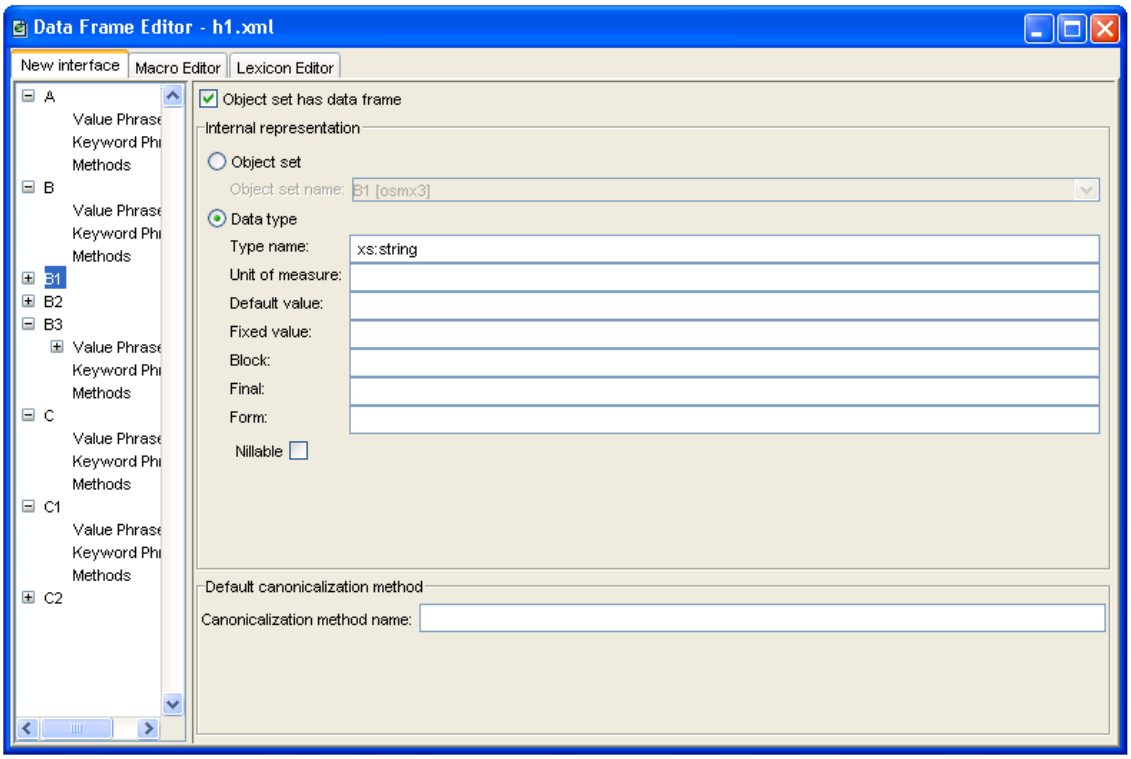


Figure 5.6: Data Frame for Element *B1* in Figure 5.3

the object set *B2* contains the text in Lines 32–37 in its value phrase field, except that we omit the name attribute and its value.[2]

– *complex data type.* The element becomes a nonlexical object set. The connection between the object set and its children is determined according to the content of the *complexType* structure.[3] For example, the element *B* in Line 6 in Figure 5.3 has the attribute type *BType* which references the global complex type *BType* beginning on Line 11 that contains a sequence of the elements *B1*, *B2*, and *B3*. The element *B* becomes a nonlexical object set, and the sequence structure that appears in definition of *BType* connects the object set *B* with its children *B1*, *B2*, and *B3*.

[2]We note that in OSM on which C-XML is built that the value expression is usually a regular expression that generates the possible instance values. In C-XML we accept other well defined syntaxes, but the purpose remains the same—to generate the possible instance values.

[3]See Section Section 5.2.6, which explains *complexType*.

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
 3    <xs:element name="Person">
 4       <xs:complexType>
 5          <xs:sequence>
 6             <xs:element name="FirstName" type="xs:string"/>
 7             <xs:element name="LastName" type="xs:string"/>
 8          </xs:sequence>
 9       </xs:complexType>
10    </xs:element>
11    <xs:element name="Student" substitutionGroup="Person"/>
12    <xs:element name="Professor" substitutionGroup="Person"/>
13 </xs:schema>
```

Figure 5.7: Example Using the Attribute *substitutionGroup* in *Element*.

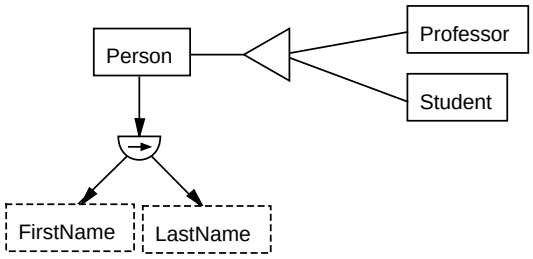


Figure 5.8: Translated C-XML Model Instance of Figure 5.7

- *abstract* and *substitutionGroup*. When an element in XML Schema has a *substitutionGroup* attribute, the element can be substituted for the element mentioned in the *substitutionGroup*. Thus, a *substitutionGroup* becomes a generalization/specialization relationship set in C-XML, where the generalized object set is the element in XML-Schema instance given as the value for the *substitutionGroup* attribute and the specialized object set is the element in the XML-Schema instance given as the value of the *name* attribute. For example, in Line 11 of Figure 5.7, *Person* is the value for the *substitutionGroup* attribute and thus, *Person* is the generalized object set and *Student* is the specialized object set. When several *name* attributes share the same *substitutionGroup* attribute, they become a group of specialized object sets under a common generalized object set. For example, in Line 11 in Figure 5.7, *Person* is the value for the *substitutionGroup* attribute for element *Student*, and in Line 12 *Person* is the value for the *substitutionGroup* attribute for element *Professor*; thus, *Person* is the generalized object set in a generaliztion/specialization and each of *Student* and *Professor* becomes a specialized object set grouped under *Person*.

  The *abstract* attribute controls whether the element may be used directly in an instance document. The default value for the *abstract* attribute is *false*. When

the value of *abstract* attribute is set to *true*, the element must be substituted through a substitution group in the instance document. In this case we add a union constraint to the generalization/specialization, which in C-XML forces every value present in the generalization to also be present in at least one of its specializations.

5.2.3 Attribute

Figure 5.9 shows definition details for *attribute* in XML Schema. Figure 5.10 shows an example using *attribute* in an XML-Schema instance, and Figures 5.11 shows a C-XML translation of Figure 5.10.

```
<attribute id = ID
 default = string
 fixed = string
 form = (‘qualified’ | ‘unqualified’)
 name = NCName
 ref = QName
 type = QName
 use = (‘optional’ | ‘prohibited’ | ‘required’): ‘optional’>
 Content: (annotation?, (simpleType?))
</attribute>
```

Figure 5.9: Attribute Declaration.

We translate an attribute in XML Schema to a lexical object set that has the same name as the *attribute* in XML Schema. In the following, we give the translation details of the component parts of *attribute*.

- *id*. As before, this is implicit—not mapped.
- *name*. The *name* of the attribute becomes the name of the object set representing that attribute in C-XML. For example, the object set in C-XML that represents the attribute *Language* declared in Line 5 in Figure 5.10 has the name *Language* in Figure 5.11.
- *default*, *fixed*, and *form*. These attributes are added to the associated data frame of the object set that represents the attribute.
- *use*. The default value of the attribute *use* is *optional*. When an attribute is optional, we introduce a *0:1* participation constraint. The participation constraint

```
 1 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
 2    <xs:element name="Employee">
 3       <xs:complexType>
 4          <xs:attribute ref="Status" use="optional"/>
 5          <xs:attribute name="Language" type="LanguageType"/>
 6          <xs:attribute name="EmployeeNumber" use="required">
 7             <xs:simpleType>
 8                <xs:restriction base="xs:string">
 9                   <xs:length value="9"/>
10                </xs:restriction>
11             </xs:simpleType>
12          </xs:attribute>
13       </xs:complexType>
14    </xs:element>
15    <xs:attribute name="Status" type="xs:string"/>
16    <xs:simpleType name="LanguageType">
17       <xs:restriction base="xs:string">
18          <xs:enumeration value="English"/>
19          <xs:enumeration value="Latin"/>
20       </xs:restriction>
21    </xs:simpleType>
22 </xs:schema>
```

Figure 5.10: Example of *Attribute* Structure in XML Schema.

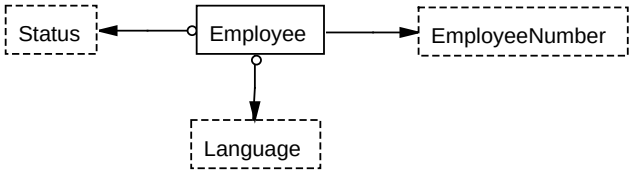


Figure 5.11: Translated C-XML Model Instance of Figure 5.10.

of the attribute is for the object set representing the containing element of the attribute in the relationship set between the lexical object set that represents the attribute and the object set that represents the containing element of the attribute. When *use* is *prohibited* the participation constraint is *0:0*, and when *use* is *required* the participation constraint is *1:1*. In C-XML diagrams we can, of course, use optional and functional notation for relationship sets to denote the participation constraints as Figure 5.11 shows. Thus, for example, attribute *Status* in Line 4 in Figure 5.10 whose *use* value is *optional* yields the optional constraint for the object set *Employee* for the binary relationship set between *Employee* and *Status*, and the attribute *EmployeeNumber* in Line 6 whose *use* value is *required* yields the mandatory constraint.

- *ref*. This is the same as for *element* except that the reference must be an attribute. Thus, we handle this attribute in exactly the same way we handle *ref* in Section 5.2.2. For example, attribute *Status* in Line 4 of Figure 5.10 refers to the global attribute *Status* that appears in Line 15. In the C-XML instance,

the global attribute *Status* becomes a lexical object set that takes the binary relationship set and the constraints that the local element in Line 4. Thus, the lexical object set *Status* will be connected to the object set *Employee* (which represents the element *Employee* in Line 2). The *optional* value for the *use* attribute in Line 4 appears as a participation constraint on the *Employee* side of the connection to *Status* in C-XML.

- *type*. This is the same as for *element* except that the type can only be *built-in* or *simple*. Thus, we handle this attribute in exactly the same way we handle *built-in data type* and *simple data type* in Section 5.2.2. For example, the type *xs:string* of the attribute *Status* in Line 15 of Figure 5.10 is specified in the type name field of the data frame that associates with the lexical object set *Status*. As another example, the element *Language* in Line 5 in Figure 5.10 has the type *LanguageType* which references the global simple data type *LanguageType* in Line 16. In the translation to C-XML, the data frame associated with the object set *Language* contains the text in Lines 16–21 in its value phrase field except that we omit the name attribute and its value.

5.2.4 Key, Unique and Keyref

Figure 5.12 shows definition details for *key*, *unique*, and *keyref* in XML Schema and also definition details for *selector* and *field*, which are embedded in each *key*, *unique*, and *keyref* declaration. The *selector* in XML Schema defines the element on which a uniqueness constraint or key reference constraint (for *keyref*) applies within the defined scope. The *field* in XML Schema defines the elements or attributes that are constrained to be unique within the defined scope. XPath expressions identify the location of elements or attributes. Figure 5.13 shows an example using *key*, *unique*, and *keyref* in an XML-Schema instance. In the example, the *key* component in Lines 16–19 constrains the element *Student* under the element *School* to have unique *StudentID* values. The *unique* component in Lines 45–48 constrains the element *Student* under the element *Class* to have unique *NickName* values. The *key* component in Lines 49–53 constrains the element *Student* under the element *Class* to have unique values for the pair *FirstName* and *LastName*. The *keyref* component in Lines 54–58 specifies that the set of pairs (*StudentFirstName*, *StudentLastName*) is a subset of the set of pairs (*FirstName*, *LastName*).[4]

[4]We assume that this is the intended definition. Definitions in the XML Schema literature are not clear and the validator we tried validated an XML document even though the set of

```
<unique
 id = ID
 name = NCName>
 Content: (annotation?, (selector, field+))
</unique>

<key
 id = ID
 name = NCName>
 Content: (annotation?, (selector, field+))
</key>

<keyref
 id = ID
 name = NCName
 refer = QName>
 Content: (annotation?, (selector, field+))
</keyref>

<selector
 id = ID
 xpath = a subset of XPath expression>
 Content:(annotation?)
</selector>

<field
 id = ID
 xpath = a subset of XPath expression>
 Content:(annotation?)
</field>
```

Figure 5.12: Identity-Constraint Declaration.

We translate *key* and *unique* components in XML Schema to C-XML as generalized co-occurrence constraints as follows. On the left-hand side of the generalized co-occurrence constraint, we list the name of the element that embeds the *key* or *unique* component and the names of elements or attributes in the *field* part of the *key* or *unique* component. The right-hand side of the co-occurrence constraint consists of the name of the element in the *selector* part of the *key* or *unique* component. For example, in Figure 5.14, which shows the C-XML translation of Figure 5.13, we have the following generalized co-occurrence constraints:

pairs ($StudentFirstName$, $StudentLastName$) is not a subset of the set of pairs ($FirstName$, $LastName$). We assume that the validator is in error.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
   <xs:element name="City">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="School"  maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="School">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="Class" maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:complexType>
      <xs:key name="StudentIDKey">
         <xs:selector xpath="./Class/Student"/>
         <xs:field xpath="@StudentID"/>
      </xs:key>
   </xs:element>
   <xs:element name="Class">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="Student" maxOccurs="40">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element name="FirstName" type="xs:string"/>
                     <xs:element name="LastName" type="xs:string"/>
                     <xs:element name="NickName" type="xs:string" minOccurs="0"/>
                  </xs:sequence>
                  <xs:attribute name="StudentID" type="xs:string"/>
               </xs:complexType>
            </xs:element>
            <xs:element name="ReadingContest">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element name="StudentFirstName" type="xs:string"/>
                     <xs:element name="StudentLastName" type="xs:string"/>
                     <xs:element name="EarnedPoints" type="xs:integer"/>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
      </xs:complexType>
      <xs:unique name="StudentNickNameUnique">
         <xs:selector xpath="./Student"/>
         <xs:field xpath="NickName"/>
      </xs:unique>
      <xs:key name="StudentNameKey">
         <xs:selector xpath="./Student"/>
         <xs:field xpath="FirstName"/>
         <xs:field xpath="LastName"/>
      </xs:key>
      <xs:keyref name="StudentNameKeyref" refer="StudentNameKey">
         <xs:selector xpath="./ReadingContest"/>
         <xs:field xpath="StudentFirstName"/>
         <xs:field xpath="StudentLastName"/>
      </xs:keyref>
   </xs:element>
</xs:schema>
```

Figure 5.13: Example of Identity Constraint Structures in XML Schema.

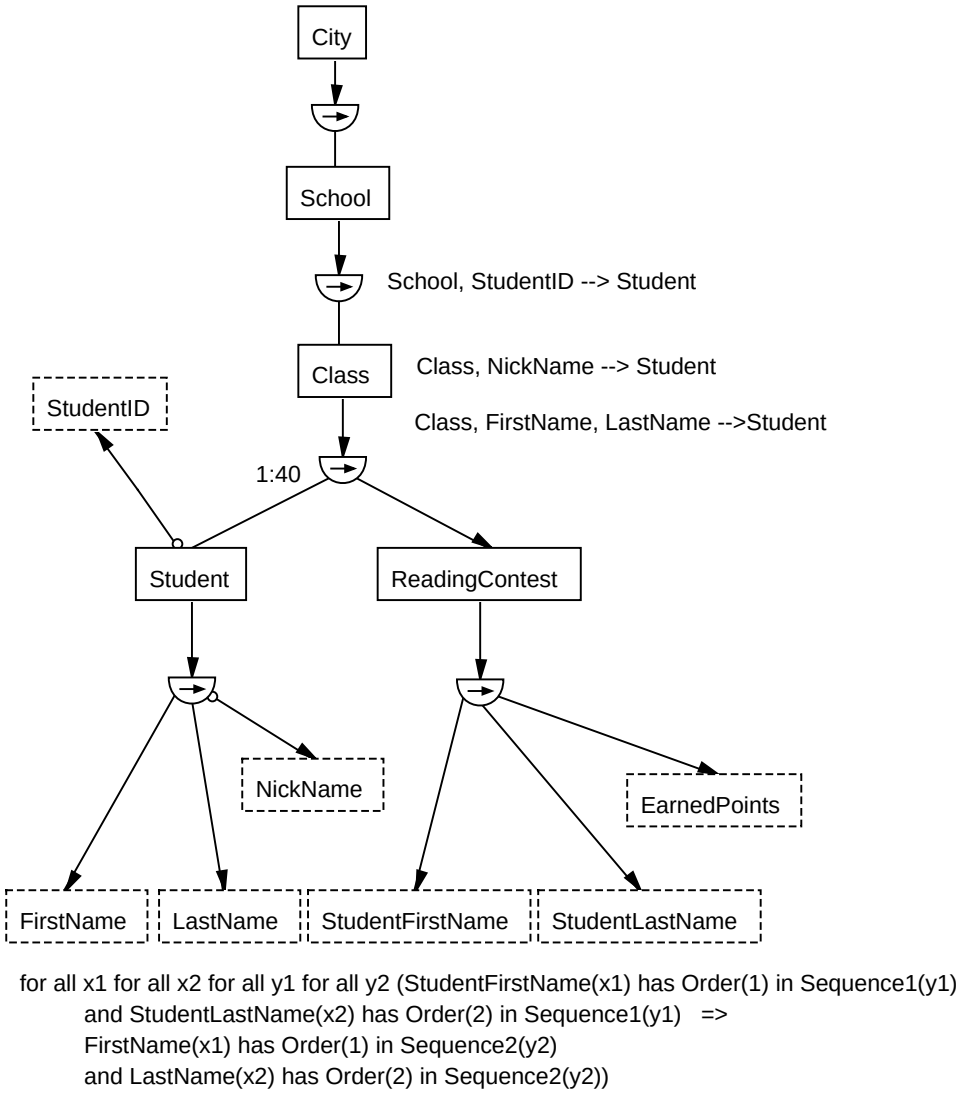


Figure 5.14: Translated C-XML Model Instance of Figure 5.13.

School, StudentID $\longrightarrow$ *Student* (for the *key* constraint in Lines 16–19),

Class, NickName $\longrightarrow$ *Student* (for the *unique* constraint defined in Lines 45–48), and

Class, FirstName, LastName $\longrightarrow$ *Student* (for the *key* constraint defined in Lines 49–53).

Multiple paths for a generalized co-occurrence constraint are possible. This happens when a C-XML hypergraph is cyclic. In this case, C-XML requires a high-level relationship set that gives a unique path over which the generalized co-occurrence

constraint holds. Since the *selector* in XML Schema uniquely defines the scope, we can translate this scope designation directly into an appropriate high-level relationship set designator for the generated co-occurrence constraint. Indeed, all generalized co-occurrence constraints associate with a high-level relationship set independent of whether the path is ambiguous. We list the names of all relationship sets over the path that leads from the object sets in the left hand side of the generalized co-occurrence constraint to the object sets in its the right hand side. For example, we specify the derived relationship set for *School, StudentID* $\longrightarrow$ *Student* as the relationship set that connects the object set *School* with a *Sequence* (the one that is between *School* and *Class*), the relationship set that connects that *Sequence* with the object set *Class*, the relationship set that connects the object set *Class* with a *Sequence* (the one that is between *Class*, *Student*, and *ReadingContest*), and the relationship set that connects that *Sequence* with the object set *Student*, and finally the relationship set that connects the object set *Student* with the object set *StudentID*.

We translate *keyref* in XML Schema to C-XML as a general constraint specifying that the n-tuples in the referencing part of the *keyerf* are a subset of the n-tuples in the referenced part. For example, in Figure 5.14, which shows the C-XML translation of Figure 5.13, we have the following general constraints:

$\forall x_1 \forall x_2 \forall y_1 \forall y_2 (StudentFirstName(x_1)\ has\ Order(1)\ in\ Sequence_1(y_1) \wedge$
$StudentLastName(x_2)\ has\ Order(2)\ in\ Sequence_1(y_1) \Longrightarrow$
$FirstName(x_1)\ has\ Order(1)\ in\ Sequence_2(y_2) \wedge$
$LastName(x_2)\ has\ Order(2)\ in\ Sequence_2(y_2))$.

In the following, we give the translation details of the component parts for *key*, *unique*, and *keyref*.

- *id*. As before, this is implicit—not mapped.
- *name*. This attribute represents the name of the *key*, *unique*, or *keyref* component in an XML-Schema instance, and we do not map it.
- *refer*. This attribute only appears in *keyref* to represent the name of the *key* or *unique* constraint referred by the *keyref*, and we do not map it.

5.2.5 Simple Type

Figure 5.15 shows definition details for *simpleType* in XML Schema. Simple types use one of the three declaration methods: **restriction** to add new constraints to a data type, **list** to define lists of values, and **union** to combine disparate data types

```
<simpleType
 id = ID
 final = (‘#all’ | (‘list’ | ‘union’ | ‘restriction’))
 name = NCName>
 Content: ( annotation ?, (restriction | list | union))
</simpleType>

<list
 id = ID
 itemType = QName>
 Content: (annotation ?, (simpleType ?))
</list>

<union
 id = ID
 memberTypes = List of QName>
 Content: (annotation ?,( simpleType *))
</union>

<restriction
 id = ID
 base = QName>
 Content: (annotation ?, ( simpleType ?, ( minExclusive |
           minInclusive | maxExclusive | maxInclusive | totalDigits |
           fractionDigits | length | minLength | maxLength | enumeration |
           whiteSpace | pattern )*))
</restriction>
```

Figure 5.15: Simple Data Type Content Declaration.

into a single data type. Figure 5.16 [5] shows an example of using a *simpleType* structure in XML Schema. Lines 18–23 show an example of a declaration by *restriction* where a simple type called *sizebynumber* is derived (by restriction) from the built-in type *positiveInteger* to create an integer between 8 and 72. Lines 32-35 show an example of creating a new list type *sizelistbynumber* based on the *simpleType sizebynumber* to define a list of values of *sizebynumber*. Lines 14–16 show an example of creating a *union* type from two simple types *sizebynumber* and *sizebystringname*. The union type represents the size as an integer between 8 and 72 or one of the strings *small*, *medium* or *large*.

We translate a *simpleType* to C-XML by adding all the content of the simple type tags to the value phrase inside the associated data frame of the element or the

[5] Lines 18–30 are taken with modification from $http://www.smrtx.com/RS/xsd\_union.htm$.

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 3    <xs:element name="root">
 4       <xs:complexType>
 5          <xs:choice>
 6             <xs:element ref="size"/>
 7             <xs:element ref="MonitorSize"/>
 8             <xs:element ref="document"/>
 9          </xs:choice>
10       </xs:complexType>
11    </xs:element>
12    <xs:element name="size" type="sizebynumber"/>
13    <xs:element name="MonitorSize">
14       <xs:simpleType>
15          <xs:union memberTypes="sizebynumber sizebystringname"/>
16        </xs:simpleType>
17     </xs:element>
18     <xs:simpleType name="sizebynumber">
19        <xs:restriction base="xs:positiveInteger">
20           <xs:minInclusive value="8"/>
21           <xs:maxInclusive value="72"/>
22        </xs:restriction>
23     </xs:simpleType>
24     <xs:simpleType name="sizebystringname">
25        <xs:restriction base="xs:string">
26           <xs:enumeration value="small"/>
27           <xs:enumeration value="medium"/>
28           <xs:enumeration value="large"/>
29        </xs:restriction>
30     </xs:simpleType>
31     <xs:element name="document" type="mysizelist"/>
32     <xs:simpleType name="sizelistbynumber">
33        <xs:list itemType="sizebynumber">
34        </xs:list>
35     </xs:simpleType>
36     <xs:simpleType name="mysizelist">
37        <xs:restriction base="sizelistbynumber">
38           <xs:length value="2"/>
39        </xs:restriction>
40     </xs:simpleType>
41 </xs:schema>
```

Figure 5.16: Example of Using *Simple Type* in XML Schema.

attribute. In the translation to C-XML, we distinguish between the following two cases.

- When an element or an attribute embeds or references a simple type, which is by itself created from a *built-in type*, the simple type is specified in the value phrase inside the associated data frame for the object set which represents that element or attribute by adding all the content of the simple-type tags, except that we omit the *name* attribute and its value.[6] For example, in Figure 5.16 the element *size* in Line 12 references the *simpleType sizebynumber* in Lines 18–23, and since this simple type is created by restriction from the *base xs:positiveInteger* which

[6]The *name* attribute only appears when the *simpleType* is declared globally in order to reference that *simpleType* through its *name* from other places inside the XML-Schema instance.

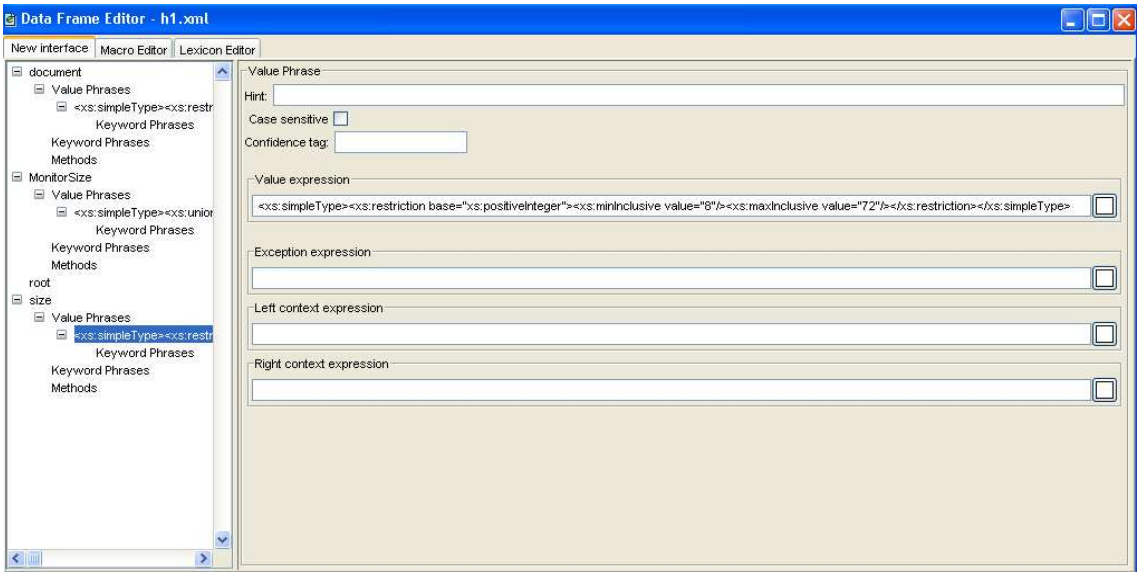


Figure 5.17: Data Frame for Element *size* in Figure 5.16

```
<xs:simpleType>
   <xs:union>
      <xs:simpleType>
         <xs:restriction base="xs:positiveInteger">
            <xs:minInclusive value="8"/>
            <xs:maxInclusive value="72"/>
         </xs:restriction>
      </xs:simpleType>
      <xs:simpleType>
         <xs:restriction base="xs:string">
            <xs:enumeration value="small"/>
            <xs:enumeration value="medium"/>
            <xs:enumeration value="large"/>
         </xs:restriction>
      </xs:simpleType>
   </xs:union>
</xs:simpleType>
```

Figure 5.18: Text Representing the Simple Type for *MonitorSize* in Figure 5.16.

```
<xs:simpleType>
   <xs:restriction>
      <xs:simpleType>
         <xs:list>
            <xs:simpleType>
               <xs:restriction base="xs:positiveInteger">
                  <xs:minInclusive value="8"/>
                  <xs:maxInclusive value="72"/>
               </xs:restriction>
            </xs:simpleType>
         </xs:list>
      </xs:simpleType>
      <xs:length value="2"/>
   </xs:restriction>
</xs:simpleType>
```

Figure 5.19: Text Representing the Simple Type for *mysizelist* in Figure 5.16.

is a *built-in* type, the data frame associated with the object set *size* contains the text in Lines 18–23 in its value phrase field as it appears in Figure 5.17.

- When an element or an attribute X embeds or references a simple type T, which itself is created from another *simpleType* S, in the translation to C-XML we unfold the T by first obtaining all the content of the S tag except the attribute name and its value and inserting that content into one line below where the *base* or *itemType* or *memberTypes* attribute is mentioned. Second, we omit the *base* or *itemType* or *memberTypes* attribute and its value in T. Third, we repeat these first two steps until the representation of T contains a built-in value for a *base* or *itemType* or *memberTypes* attributes. Fourth, the unfolded representation of the *simpleType* belonging to X is specified in the value phrase inside the associated data frame for the object set X. For example, in Figure 5.16, the element *MonitorSize* in Line 13 embeds a *simpleType* in Lines 14–16. This *simpleType* is derived from the two simple types *sizebynumber* and *sizebystringname* that are mentioned inside *memberTypes*. Each of *sizebynumber*(Lines 18–23) and *sizebystringname*(Lines 24–30) by itself is a *simpleType*. Thus, the text in Figure 5.18 is added to the value phrase inside the associated data frame of the element *MonitorSize*.

  The element *document* in Line 31 references the simple data type *mysizelist*. The *simpleType mysizelist* (Lines 31–35) is derived from the *simpleType sizelistbynumber* (Lines 18–22). Thus, the text in Figure 5.19 is added to the value phrase inside the associated data frame of the element *document*.

5.2.6 Complex Type

In XML Schema when an element includes or references a *complexType*, child elements and/or attributes are expected. We translate these child elements and attributes to object sets in C-XML and relate them to other object sets via relationship sets defined according to where the *complexType* declaration occurs in an XML-Schema instance. The connection between the object set and its children is determined according to the content of the *complexType*.

We have illustrated these translations in many of the examples we have already introduced, such as Figure 5.3 and 5.4. In Figure 5.3 the element A in Line 3 includes a *complexType* in Line 4 which itself includes a sequence of B and C. In the translation in Figure 5.4 the object set A connects to the object sets B and C via *sequence*. The element C in Line 24 embeds a *complexType* in Line 25 that represents a choice of *C1*

and *C2*. In the translation in Figure 5.4 the object set *C* is connected to the object sets *C1* and *C2* via *choice*.

Figure 5.20 shows the definition details for *complexType* in XML Schema. In the following we give the translation details of the component parts of *complexType*.

```
<complexType
 id = ID
 abstract = boolean : 'false'
 block = ('#all' | List of ('extension' | 'restriction'))
 final = ('#all' | List of ('extension' | 'restriction'))
 mixed = boolean : 'false'
 name=NCName>
 Content: (annotation?, (simpleContent | complexContent |
          ((group | all | choice | sequence)?,
          ((attribute | attributeGroup)*, anyAttribute?))))
</complexType>
```

Figure 5.20: Complex Data Type Declaration.

- *id*. As always, this is implicit—not mapped.
- *abstract*. The *abstract* attribute in a *complexType* definition only appears when the complex type is declared globally. The default value of this attribute is *false*. When the value is set to *true*, an element cannot use this complex type directly but may only derive other complex types from this one. Figure 5.22 shows a C-XML translation of Figure 5.21. In the translation to C-XML, the element that references an *abstract complexType* is represented as a nonlexical object set. In Figure 5.22 this element is *Email*. The object set is connected to other object sets that represent the elements or attributes nested under the *abstract complexType*. In Figure 5.22 this includes *Title*, *Date*, and *Message*. In addition, if nested extension elements are present, we allow any one of these to be chosen and associated with the *complexType*. Thus, in the translation from XML Schema to C-XML we add a *choice* structure. The *choice* structure connects to each extension of the *complexType*. As Figure 5.22 shows, there is a *choice* of *EmailTo* and *EMailFrom* in the C-XML translation of Figure 5.21.
- *block* and *final*. We add these attributes to the associated data frame of the object set representing the element that includes or references a *complexType*.

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
 3    <xs:element name="Document">
 4       <xs:complexType>
 5          <xs:sequence>
 6             <xs:element name="Email" type="eMailCommon" maxOccurs="unbounded"/>
 7          </xs:sequence>
 8       </xs:complexType>
 9    </xs:element>
10    <xs:complexType name="eMailCommon" abstract="true">
11       <xs:sequence>
12          <xs:element name="Title" type="xs:string"/>
13          <xs:element name="Date" type="xs:date"/>
14          <xs:element name="Message" type="xs:string"/>
15       </xs:sequence>
16    </xs:complexType>
17    <xs:complexType name="eMailOutGoing">
18       <xs:complexContent>
19          <xs:extension base="eMailCommon">
20             <xs:sequence>
21                <xs:element name="EmailTo" type="xs:string" maxOccurs="unbounded"/>
22             </xs:sequence>
23          </xs:extension>
24       </xs:complexContent>
25    </xs:complexType>
26    <xs:complexType name="eMailInComing">
27       <xs:complexContent>
28          <xs:extension base="eMailCommon">
29             <xs:sequence>
30                <xs:element name="EmailFrom" type="xs:string" maxOccurs="unbounded"/>
31             </xs:sequence>
32          </xs:extension>
33       </xs:complexContent>
34    </xs:complexType>
35 </xs:schema>
```

Figure 5.21: Sample of Using Abstract in Complex Type in XML Schema.

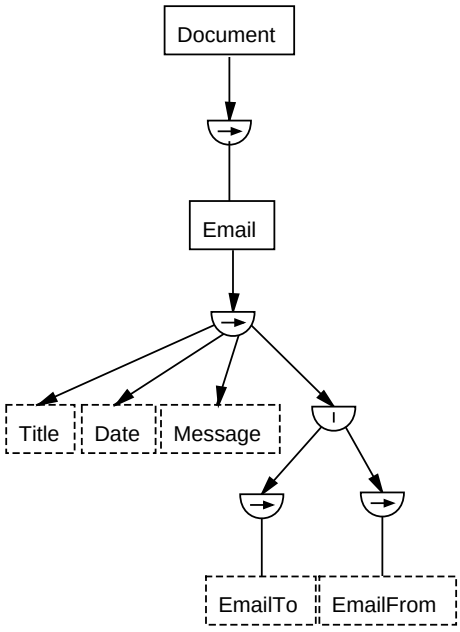


Figure 5.22: Translated C-XML Model Instance of Figure 5.21.

- *mixed*. When *mixed* is set to *true*, the object set representing the element including or referencing that *complexType* in C-XML is mixed. Figure 5.23 shows an example of mixed content of a complex type. In Figure 5.23, setting *mixed* to *true* enables character data to appear between the child-elements of *Order* in a complying XML document. Thus, the content of *Order* may, for example, be "The order for <Name>Pat</Name> identified by <OrderID>123</OrderID> was shipped on <ShipDate>2006-07-10</ShipDate>." Figure 5.24 shows a translation example.

```
<xs:element name="Order">
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="OrderID" type="xs:positiveInteger"/>
      <xs:element name="ShipDate" type="xs:date"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figure 5.23: Sample of Mixed Content in XML Schema.

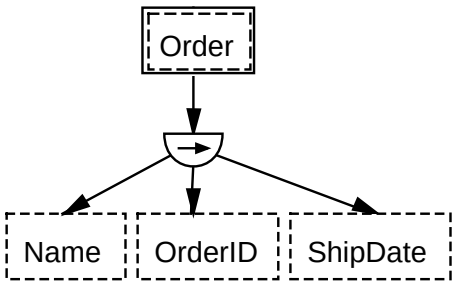


Figure 5.24: Mixed Content Structure Corresponding to Figure 5.23.

- *name*. The *name* attribute in a *complexType* definition only appears when the complex type is declared globally to be referenced from other local places inside the XML-Schema instance. In our translation of *complexType* to C-XML, we do not map the *name* attribute of the *complexType* in XML Schema to C-XML.

5.2.7 AttributeGroup

Figure 5.25 shows definition details for *attributeGroup* in XML Schema. The *attributeGroup* declaration merely lets us reference a global group of attribute declarations. Thus, to do the translation to C-XML, we simply replace the referencing attribute declaration by the attribute declarations in the referenced *attributeGroup* declaration. We then map these attribute declarations as explained in Section 5.2.3.

```
<attributeGroup id = ID
 name = NCName
 ref = QName >
 Content:(annotation?, ((attribute | attributeGroup)*, anyAttribute?))
</attributeGroup>
```

Figure 5.25: AttributeGroup Declaration.

5.2.8 All

Figure 5.26 shows definition details for *all* in XML Schema. Figure 5.27 shows an example using *all* in an XML-Schema instance, and Figure 5.28 shows a C-XML translation of Figure 5.27.

```
<all
 id = ID
 maxOccurs = 1 : 1
 minOccurs = (0 | 1) : 1>
 Content:(annotation?, element*)
</all>
```

Figure 5.26: All Declaration.

We translate an *all* structure in XML Schema to C-XML by representing the elements embedded in the *all* as object sets. In the translation, a binary relationship set appears between the object set of the containing element of the *all* structure and each of the object sets of the children elements of the *all* structure.

In the following, we give the translation details of the component parts of *all*.

- *id*. As always, this is implicit—not mapped.

```
 1 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
 2    <xs:element name="Person">
 3       <xs:complexType>
 4          <xs:all>
 5             <xs:element name="Address" type="xs:string"/>
 6             <xs:element name="ContactInfo" type="xs:string" minOccurs="0"/>
 7          </xs:all>
 8       </xs:complexType>
 9    </xs:element>
10 </xs:schema>
```

Figure 5.27: Example of *All* Structure in XML Schema.

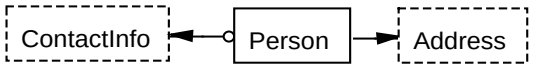


Figure 5.28: Translated C-XML Model Instance of Figure 5.27

- *minOccurs* and *maxOccurs*. As usual, we translate *minOccurs* and *maxOccurs* to participation constraints. In the *all* structure, the minimum number of occurrences must be *0* or *1* and the maximum number must be *1*. The default value for *minOccurs* is *1*. Further, each child element of the *all* structure can only occur between *0* and *1* times. In translating from XML Schema to C-XML, if the *minOccurs* of the *all* structure is *0*, the final *minOccurs* of each element embedded in the *all* structure will be *0*. If the *minOccurs* of the *all* structure is *1*, the final *minOccurs* of each element embedded in the *all* structure will remain the same. The final *maxOccurs* of each element embedded in the *all* structure will be *1*. Thus, for example, in Figure 5.28, the binary relationship sets are both functional. Since *minOccurs* in Line 6 in Figure 5.27 is *0*; the binary relationship that appears between *Person* and *ContactInfo* is optional on the *Person* side.

5.2.9 Sequence

Figure 5.29 shows definition details for *sequence* in XML Schema. Figure 5.30 shows an example of *sequence* in XML Schema where an element *ParentOrGuardian* contains one or two sequences of an optional *PreferredName* string, one *FirstName* string, zero to two *MiddleName* strings, and one *LastName* string. Figure 5.31 shows a C-XML translation of Figure 5.30.

We translate a sequence structure in XML Schema to C-XML as follows. In C-XML, a bounded half circle with a directional arrow represents a sequence. The sequenced child concepts connect to the curved side, and the parent concept that

```
<sequence id = ID
 maxOccurs = (nonNegativeInteger | 'unbounded' ) : 1
 minOccurs = nonNegativeInteger : 1>
 Content: (annotation?, (element | group | choice | sequence | any)*)
</sequence>
```

Figure 5.29: Sequence Declaration.

```
 1 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
 2    <xs:element name="ParentOrGuardian">
 3       <xs:complexType>
 4          <xs:sequence minOccurs="1" maxOccurs="2">
 5             <xs:element name="PreferredName" type="xs:string" minOccurs="0" maxOccurs="1"/>
 6             <xs:element name="FirstName" type="xs:string"/>
 7             <xs:element name="MiddleName" type="xs:string" minOccurs="0" maxOccurs="2"/>
 8             <xs:element name="LastName" type="xs:string"/>
 9          </xs:sequence>
10       </xs:complexType>
11    </xs:element>
12 </xs:schema>
```

Figure 5.30: Example of *Sequence* Structure in XML Schema.

contains the sequenced child concepts connects to the flat, bounded side. In the following, we give the translation details of the component parts of *sequence*.

- *id*. As always, this is implicit—not mapped.
- *minOccurs* and *maxOccurs*. The participation constraints for the entire sequence are for the connection to the parent concept. The participation constraints for each of the component children of the sequence are for the connection to the sequence symbol. The default value for *minOccurs* and *maxOccurs* for both the entire sequence and for the component children of the sequence is *1*. As Figure 5.31 shows, for *minOccurs="1"* and *maxOccurs="2"* of the *sequence* element in Line 4 we write the participation constraint *1:2* near the connection

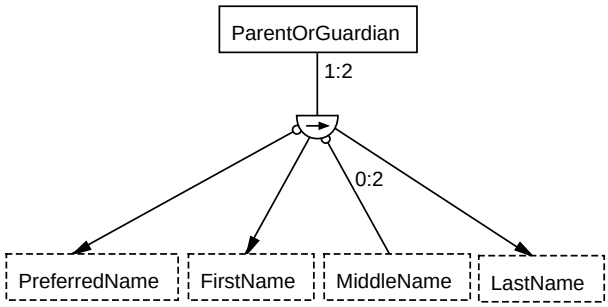


Figure 5.31: Translated C-XML Model Instance of Figure 5.30.

to *ParentOrGuardian*, and for *minOccurs="0"* and *maxOccurs="2"* of the child component *MiddleName* in Line 7 we write the participation constraint *0:2* near the sequence symbol.

5.2.10 Choice

Figure 5.32 shows definition details for the *choice* structure in XML Schema. Figure 5.33 shows an example of *choice* in an XML-Schema instance where an element *Phone* contains either one *HomePhone* or one *CellPhone* but not both. Figure 5.34 shows a C-XML translation of Figure 5.33.

```
<choice id = ID
 maxOccurs = (nonNegativeInteger | 'unbounded' ) :1
 minOccurs = nonNegativeInteger : 1>
 Content: (annotation?, (element | group | choice | sequence | any)*)
</choice>
```

Figure 5.32: Choice Declaration.

The translation of the *choice* structure in XML Schema is almost identical to the translation for *sequence*. Like *sequence*, we use a bounded half circle for choice in C-XML. Instead of the arrow, however, we use a bar to indicate the *choice* structure. In the following, we give the translation details of the component parts of *choice*.

- *id*. As always, this is implicit—not mapped.
- *minOccurs* and *maxOccurs*. The participation constraints for the entire *choice* are for the connection to the parent concept. The participation constraints for each of the component children of the *choice* are for the connection to the sequence symbol. The default value for *minOccurs* and *maxOccurs* for

```
 1 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
 2    <xs:element name="Phone">
 3       <xs:complexType>
 4          <xs:choice>
 5             <xs:element name="HomePhone" type="xs:string"/>
 6             <xs:element name="CellPhone" type="xs:string"/>
 7          </xs:choice>
 8       </xs:complexType>
 9    </xs:element>
10 </xs:schema>
```

Figure 5.33: Example of *Choice* Structure in XML Schema.

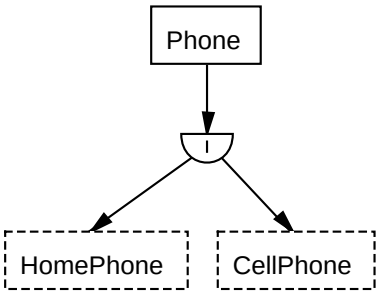


Figure 5.34: Translated C-XML Model Instance of Figure 5.33.

both the entire *choice* and for the component children of the *choice* is *1*. As Figure 5.34 shows, the default value for both *minOccurs* and *maxOccurs* of the *choice* element in Line 4 which is *1* is translated to a functional edge from *Phone* to the *choice* component, and the default *minOccurs="1"* and *maxOccurs="1"* of the component child *HomePhone* of the *choice* is the functional edge from the *choice* component to *HomePhone*.

5.2.11 Any and AnyAttribute

Figures 5.35 and 5.36 show definition details for *any* and *anyAttribute* respectively in XML Schema. Figure 5.37 shows an example of *any* and *anyAttribute* in an XML-Schema instance. The example shows a schema that describes a *Customer* element containing a *FirstName* and a *LastName* element in sequence and a *CustomerID* attribute. Additionally, the two *any* elements specify that zero or more elements from the *urn:xmlns:25hoursaday-com:customer* namespace can appear after the customer's name elements followed by zero or more elements from any other namespace. The attribute *anyAttribute* specifies that the *Customer* element can have attributes from any namespace. Figure 5.38 shows a C-XML translation of Figure 5.38.

```
<any id = ID
 maxOccurs = ( nonNegativeInteger | 'unbounded') : 1
 minOccurs = nonNegativeInteger : 1
 namespace = (('##any' | '##other') | List of (anyURI |
             ('##targetNamespace' | '##local'))) : '##any'
 processContents = ('lax' | 'skip' | 'strict') : 'strict' >
 Content: (annotation?)
</any>
```

Figure 5.35: Any Declaration.

```
<anyAttribute id = ID
 namespace = ((‘##any’ | ‘##other’) | List of (anyURI |
             (‘##targetNamespace’ | ‘##local’))) : ‘##any’
 processContents = (‘lax’ | ‘skip’ | ‘strict’) : ‘strict’>
 Content: (annotation?)
</anyAttribute>
```

Figure 5.36: AnyAttribute Declaration.

We translate occurrences of *any* and *anyAttribute* in XML Schema to high-level object sets to indicate they contain some content from elsewhere. XML Schema is not specific enough to designate which object set. We thus cannot specify which object set. We therefore name these object sets “any”. Conceptually, in C-XML, whether the object set is an attribute or an element does not matter, and we do not distinguish between these cases. In the following, we give the translation details for the component parts for *any* and *anyAttribute*.

```
 1 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
              xmlns:cust="urn:xmlns:25hoursaday-com:customer"
              targetNamespace="urn:xmlns:25hoursaday-com:customer"
              elementFormDefault="qualified">
 2    <xs:element name="Customer">
 3       <xs:complexType>
 4          <xs:sequence>
 5             <xs:element name="FirstName" type="xs:string"/>
 6             <xs:element name="LastName" type="xs:string"/>
 7             <xs:any namespace="##targetNamespace" processContents="strict" minOccurs="0"
                       maxOccurs="unbounded"/>
 8             <xs:any namespace="##other" processContents="lax" minOccurs="0"/>
 9          </xs:sequence>
10          <xs:attribute name="CustomerID" type="xs:integer"/>
11          <xs:anyAttribute namespace="##any" processContents="skip"/>
12       </xs:complexType>
13    </xs:element>
14 </xs:schema>
```

Figure 5.37: Example of the *Any* and *anyAttribute* Structures in XML Schema.

- *id*. As always, this is implicit—not mapped.
- *namespace*. This attribute specifies the namespaces that an XML validator examines to determine the validity of an element in an XML instance, and it is not a conceptual construct. Thus, we do not map it to C-XML. We can, however, record it as a comment in C-XML.

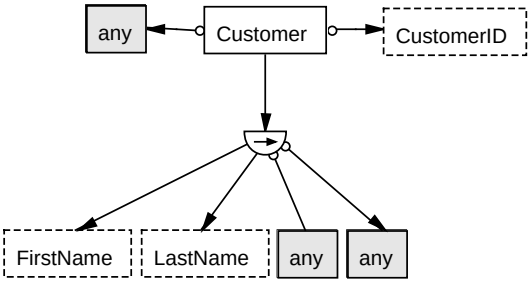


Figure 5.38: The Translated C-XML Model Instance of Figure 5.37.

- *processContents*. This attribute specifies how the XML processor should handle validation against the elements specified by the *any* or *anyAttribute*, and is not a conceptual construct. Thus, we do not map it to C-XML. We can, however, record it as a comment in C-XML.

- *minOccurs* and *maxOccurs*. The default value for both *minOccurs* and *maxOccurs* is *1*. As usual, in the translation to C-XML, *minOccurs* and *maxOccurs* become participation constraints and are assigned to the connection of the object set that represents the containing element of the *any* structure. For example, the *minOccurs="0"* and *maxOccurs="unbounded'* for *any* in Line 7 becomes *0:** represented by the many-many relationship set with an optional descriptor on the sequence designator.

5.2.12 Group

Figure 5.39 shows definition details for the *group* structure in XML Schema. Figure 5.40 shows an example of using a *group* structure in XML Schema. A global group named *group1* in Line 19 has a sequence of two elements, *A1* and *A2*. The group is referenced in Line 7 and Line 14. Another global group named *group2* in Line 25, which has elements *B1* and *B2* that can appear in any order, is referenced in Line 17. Figure 5.41 shows a C-XML translation of Figure 5.40.

We translate the *group* structure in XML Schema to C-XML by representing the content of the *group* only once and linking it in with a relationship set connection for each reference to it. If the *group* is a *sequence* or a *choice*, that *sequence* or *choice* represents the *group*. If the *group* is an *all*, only the elements embedded in the *all* structure are translated to C-XML as object sets. Each object set representing an element embedded in the *all* structure is connected through a binary relationship set to the object set representing the containing element of the *group*.

```
<group id = ID
 maxOccurs = (nonNegativeInteger | 'unbounded' ) : 1
 minOccurs = nonNegativeInteger : 1
 name = NCName
 ref = QName >
 Content: (annotation?, (all | choice | sequence)?)
</group>
```

Figure 5.39: Group Declaration.

- *id*. As always, this is implicit—not mapped.

- *name*. Since the *name* attribute can only appear in the global group declaration and serves only to provide a reference name for the group, and since we unfold the content of the global group in our translation to C-XML, we do not map the name of the group in XML Schema to C-XML.

- *ref*. The *ref* attribute is used in XML Schema to reuse a global group in an XML-Schema instance from other local places inside the same XML-Schema instance. In the translation to C-XML, we unfold the content of the global group. The unfolded content connects into the overall structure in the same way the local group which contains the *ref* attribute in the XML-Schema instance connects. For example, the global group *group1* in Line 19 is referenced from the two local groups in Line 7 and Line 14. In the C-XML instance, the global group *group1* is represented via the sequence structure embedded in the *group*. The sequence connects into the sequence representation in Line 6 in the same way the local group in Line 7 connects. Also, the sequence connects into another object set *E1* that represents the element *E1* in the same way that the local group in Line 14 connects. The global group *group2* in Line 25 is referenced from the local group in Line 17. In the C-XML instance, the global group *group2* in Line 25 is represented by the object sets *B1* and *B2*, which represent the elements *B1* in Line 27 and *B2* in Line 28, which are embedded in the *all* structure beginning in Line 26. The object sets *B1* and *B2* connect to the object set *E2* in the same way the local group, *group2* in Line 17, connects.

- *minOccurs* and *maxOccurs*. The *minOccurs* and *maxOccurs* attributes of the group can only appear for local groups referencing the global groups. The *minOccurs* and *maxOccurs* attributes may not appear as attributes for a global group. Also, a *sequence*, *choice*, or *all* structure embedded in a global group may

```
 1 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
 2    <xs:element name="Document">
 3       <xs:complexType>
 4         <xs:choice minOccurs="1" maxOccurs="unbounded">
 5            <xs:element name="E1" type="E1Type"/>
 6            <xs:sequence>
 7               <xs:group ref="group1" minOccurs="1" maxOccurs="2"/>
 8            </xs:sequence>
 9            <xs:element name="E2" type="E2Type"/>
10         </xs:choice>
11       </xs:complexType>
12    </xs:element>
13    <xs:complexType name="E1Type">
14       <xs:group ref="group1" minOccurs="2" maxOccurs="5"/>
15    </xs:complexType>
16    <xs:complexType name="E2Type">
17       <xs:group ref="group2" minOccurs="1" maxOccurs="3"/>
18    </xs:complexType>
19    <xs:group name="group1">
20       <xs:sequence>
21          <xs:element name="A1" type="xs:string"/>
22          <xs:element name="A2" type="xs:string" minOccurs="0" maxOccurs="2"/>
23       </xs:sequence>
24    </xs:group>
25    <xs:group name="group2">
26       <xs:all>
27          <xs:element name="B1" type="xs:string"/>
28          <xs:element name="B2" type="xs:string" minOccurs="0" maxOccurs="1"/>
29       </xs:all>
30    </xs:group>
31 </xs:schema>
```

Figure 5.40: Example of the *Group* Structure in XML Schema.

not have *minOccurs* and *maxOccurs* attributes. In the translation to C-XML, when the global group embeds a sequence or choice structure, the *minOccurs* and *maxOccurs* of the local group that references the global group become participation constraints for the C-XML component that represents the containing element of the local group in the XML-Schema instance. Note that the containing element could be an object set, a *sequence*, or a *choice*. In Line 7, the *minOccurs="1"* and *maxOccurs="2"* of the local group, *group1*, are translated to the participation constraint *1:2* for the *sequence* beginning in Line 6. In Line 14, *minOccurs="2"* and *maxOccurs="5"* of the local group, *group1*, are translated to the participation constraint *2:5* for the object set *E1* that represents the element *E1* in Line 9. When the global group embeds an *all* structure, each *minOccurs* and *maxOccurs* of the local group that references that global group, is multiplied by each *minOccurs* and *maxOccurs* of the elements embedded in the *all* structure. These multiplication results become participation constraints in the relationship set between each element embedded in the *all* structure and the object set representing the containing element of the local group referencing that global group. In Line 17, *minOccurs="1"* and *maxOccurs="3"* of the

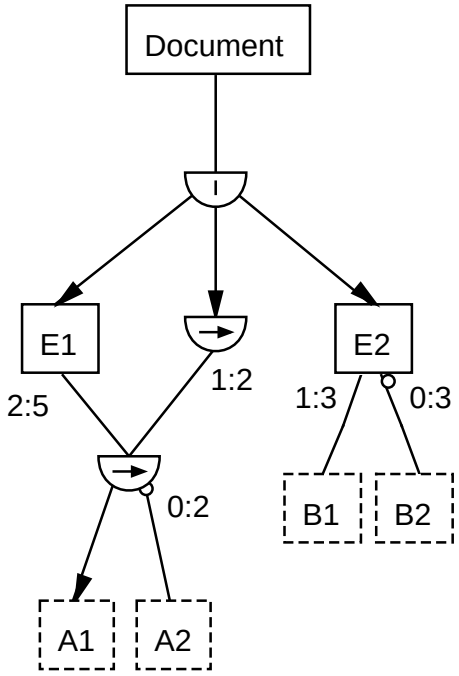


Figure 5.41: Translated C-XML Model Instance of Figure 5.40

local group, *group2*, are respectively multiplied by *minOccurs="1"* and *maxOccurs="1"* for the element *B1* in Line 27 to get the result *minOccurs="1"* and the *maxOccurs="3"*, which translates into the participation constraint *1:3* for the the object set *E2* in the binary relationship set between *E2* and *B1*. Also, in Line 17, *minOccurs="1"* and *maxOccurs="3"* of the local group, *group2*, are respectively multiplied by *minOccurs="0"* and *maxOccurs="1"* for the element *B2* in Line 28 to get the result *minOccurs="0"* and *maxOccurs="3"*, which translates into the participation constraint *0:3* near the object set *E2* in the binary relationship set between *E2* and *B2*.

5.2.13 Simple Content Complex Type

Figure 5.42 shows a declaration of *simpleContent complexType* in XML Schema. As always the *id* is implicit—not mapped. As Figure 5.42 shows, in XML Schema we provide the content of *simpleContent* either by *restriction* or *extension*. We explain the translation of these elements respectively in the subsections below.

```
<simpleContent
 id = ID>
 Content: (annotation?, (restriction | extension))
</simpleContent>
```

Figure 5.42: Simple Content Declaration.

Derivation by Restriction of Simple Content

Figure 5.43 shows definition details for *restriction* of *simpleContent* for *complexType* in XML Schema. *Restriction* allows the addition of constraints to both the attributes and the text nodes in simple content elements. Restrictions reduce the set of valid instances and structures. To restrict attributes, XML Schema provides modifiers for attribute definitions. The controls applied to the attributes must result in a restriction of their definitions. For example, a value may be fixed, or an attribute that was optional may become either required or prohibited.

```
<restriction
 id = ID
 base = QName>
 Content: (annotation?, (simpleType?, (minExclusive | minInclusive |
          maxExclusive | maxInclusive | totalDigits | fractionDigits |
          length | minLength | maxLength | enumeration | whiteSpace |
          pattern)*)?, ((attribute | attributeGroup)*, anyAttribute?))
</restriction>
```

Figure 5.43: Restriction on simpleContent Content Declaration.

In Figure 5.44, Line 28 shows an element *MensSize* that references the complex type *MensSizeType*. *MensSizeType* (defined in Lines 29–36) is a simple content complex type derived by restriction from another simple content complex type *ShoeSizeType* (Lines 20–27), which in turn is also a simple content complex type derived by extension from *SizeType* (Lines 12–18). *SizeType* provides the base, *integer*. *MensSize* restricts the *integer* values to be greater than 6. Also, *MensSize* has an additional restriction on the attribute *HeelHeight* listed in the base *ShoeSizeType* that removes *HeelHeight* from *MensSize* by setting its use to *prohibited*. Figure 5.45 shows the translation of Figure 5.44.

In the following, we give the translation details of the component parts for *restriction* of *simpleContent* for *ComplexType*.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="Document">
      <xs:complexType>
         <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="MensSize"/>
            <xs:element ref="ShoeSize"/>
            <xs:element ref="WomensSize"/>
         </xs:choice>
      </xs:complexType>
   </xs:element>
   <xs:complexType name="SizeType">
      <xs:simpleContent>
         <xs:extension base="xs:integer">
            <xs:attribute name="Country" type="xs:string"/>
         </xs:extension>
      </xs:simpleContent>
   </xs:complexType>
   <xs:element name="ShoeSize" type="ShoeSizeType"/>
   <xs:complexType name="ShoeSizeType">
      <xs:simpleContent>
         <xs:extension base="SizeType">
            <xs:attribute name="Width" type="xs:string" />
            <xs:attribute name="HeelHeight" type="xs:integer"/>
         </xs:extension>
      </xs:simpleContent>
   </xs:complexType>
   <xs:element name="MensSize" type="MensSizeType"/>
   <xs:complexType name="MensSizeType">
      <xs:simpleContent>
         <xs:restriction base="ShoeSizeType">
            <xs:minExclusive value="6"/>
            <xs:attribute name="HeelHeight" type="xs:integer" use="prohibited"/>
         </xs:restriction>
      </xs:simpleContent>
   </xs:complexType>
   <xs:element name="WomensSize" type="WomensSizeType"/>
   <xs:complexType name="WomensSizeType">
      <xs:simpleContent>
         <xs:extension base="ShoeSizeType">
            <xs:attribute name="Fashion" type="xs:string"/>
            <xs:attribute name="Quality" type="xs:string"/>
         </xs:extension>
      </xs:simpleContent>
   </xs:complexType>
</xs:schema>
```

Figure 5.44: Examples of Simple Content Nested under Complex Type in XML Schema.

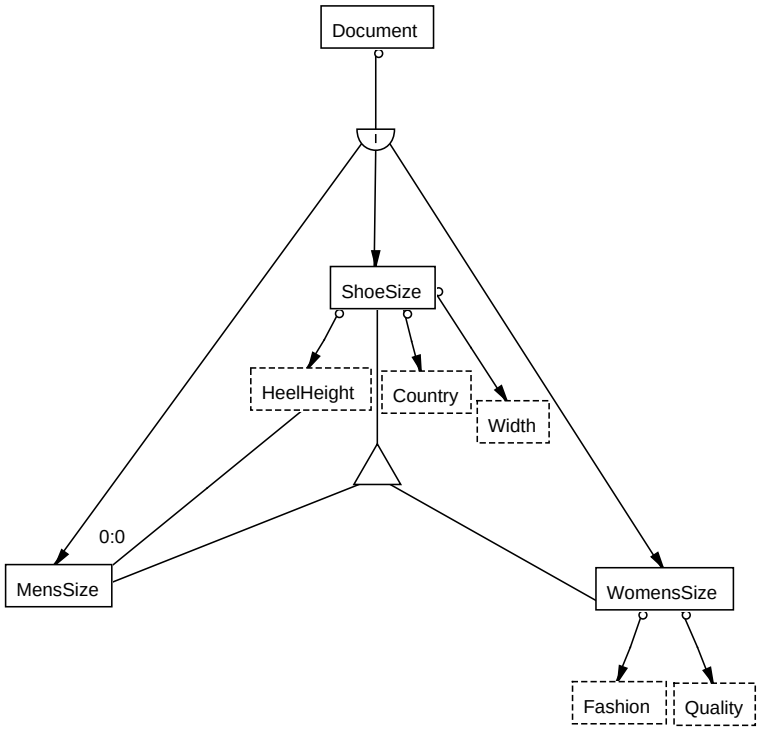


Figure 5.45: Translated Simple Content Instance of Figure 5.44.

- *id*. As always, this is implicit—not mapped.

- *base*. When the value of the *base* attribute in a *restriction* of *simple content complex type* T_1 matches the value of a *name* attribute of a simple content complex type T_2 and there is an element E_1 whose type is T_1 and an element E_2 whose type is T_2, in the translation to C-XML E_1 becomes a nonlexical specialized object set S_1 of the nonlexical object set S_2 that represents E_2. In Figure 5.44, $T_1 = MensShoeType$, $T_2 = ShoeSizeType$, $E_1 = MensSize$, $E_2 = ShoeSize$, $S_1 = MensSize$, and $S_2 = ShoeSize$. Thus, as Figure 5.45 shows, the object set *MensSize* is a specialization of the object set *ShoeSize*. S_1 connects via binary relationship sets to each lexical object set that represents one of the restricted attributes nested under T_1. Thus, as Figure 5.45 shows, $MensSize$ connects to $HeelHeight$. The restriction on the values of the element E_1 whose type is T_1 appears in the the data frame associated with E_1. We take the content of the *restriction* component nested under *simpleContent* in T_1 and delete from that content any *attribute* declarations and add to the beginning of that content an open tag of *simpleType* and add to the end of that content a closed tag of *simpleType*. We unfold the *base*; if the *base* contains another *base*

with a built-in value, then we add this value as the *base* value of our constructed *simpleType*. Otherwise, we repeat the unfolding step until we get a *base* with a built-in value. Thus, to specify that the value of *MensSize* has an integer value greater than 6, we add the text in Figure 5.46 to the value phrase inside the associated data frame of the element *MensSize*. In the text, we get the built-in value *xs:integer* of the *base* by tracing from the *base* in the complex type *MensSizeType* (Lines 29–36) through the *base* in *ShoeSizeType* (Lines 20–27) to the *base* in *SizeType* (Lines 12–18).

```
<xs:simpleType>
   <xs:restriction base="xs:integer">
      <xs:minExclusivevalue="6"/>
   </xs:restriction>
</xs:simpleType>
```

Figure 5.46: The text Representing the Simple Type for *MensSize* in Figure 5.44

Derivation by Extension of Simple Content

Figure 5.47 shows definition details for *extension* of *simpleContent* for *complexType* in XML Schema. The *base* attribute specifies the type to extend, and the list of attributes to add to the content is given as *attribute*, *attributeGroup*, and *anyAttribute* embedded in the *extension*.

```
<extension
 id = ID
 base = QName>
 Content:(annotation?,((attribute | attributeGroup)*, anyAttribute?))
</extension>
```

Figure 5.47: Extension on a simpleContent Content Declaration.

Figure 5.44 shows two examples of derivation by extension of the simple content complex type. The first example (Lines 37–45) shows an element *WomensSize* (Line 37) that references the complex type *WomensSizeType* (Line 38). *WomensSizeType* is a simple content complex type derived by extension from another simple content complex type *ShoeSizeType* (Lines 20–27). *ShoeSizeType* references another element *ShoeSize* (Line 19). In a complying XML document, *WomensSize* has the

same attributes, *Width* and *Height*, that *ShoeSize* has, but *WomensSize* also has attributes *Fashion* and *Quality* that are nested under the extension content in the complex type *WomensSizeType*. Ultimately, *WomensSize* has the type *integer* as specified in *SizeType*. The second example (Lines 19–27) shows an element *ShoeSize* (Line 19) that references the complex type *ShoeSizeType*. In this case, however, although *ShoeSizeType* (Lines 20–27) is a simple content complex type derived by extension from another simple content complex type *SizeType* (Lines 12–18), *SizeType* is not referenced by any element in the schema instance. Thus, *ShoeSize* needs to include the attribute *Country* that *SizeType* has. *ShoeSize* also has the attributes, *Width* and *HeelHeight*, that are nested under *ShoeSizeType*. *ShoeSize* has the type *integer*, the *base* in *SizeType*. Figure 5.45 shows the translation of Figure 5.44.

In the following, we give the translation details of the component parts for *extension* of *simpleContent* for *complexType*.

- *id*. As always, this is implicit—not mapped.
- *base*. We distinguish between two situations:
  - When the value of the *base* attribute in an *extension* of *simple content complex type* T_1 matches the value of a *name* attribute of a simple content complex type T_2 and there is an element E_1 that references T_1 and there is an element E_2 that references T_2, then in the translation to C-XML, the element E_1 that references T_1 becomes a nonlexical specialized object set S_1 of the object set S_2 that represents the element E_2 referencing T_2. In Figure 5.44, $T_1 = WomensSizeType$, $T_2 = ShoeSizeType$, $E_1 = WomensSize$, $E_2 = ShoeSize$, $S_1 = WomensSize$, and $S_2 = ShoeSize$. Thus, as Figure 5.45 shows, the object set *WomensSize* is a specialization of the object set *ShoeSize*. S_1 connects via binary relationship sets to each lexical object set that represents one of the attributes that are nested under the *extension* in T_1. Thus, as Figure 5.45 shows, *WomensSize* connects to *Fashion* and *Quality*. The built-in type of the object set that represents the element referencing T_1 is specified in the type name field of the data frame that associates with the object set. We get the built-in type by unfolding the *base* in T_1. If it contains a *base* with a built-in value, then we add this value to the type name field in the data frame. Otherwise, we repeat the unfolding step until we get a *base* with a built-in value. Thus, to get the type for *WomensSize*, we trace from the *base* in *WomensSizeType*

through the *base* in *ShoeSizeType* to the *base* in *SizeType* and set the type to *xs:integer*.

– When the value of the *base* attribute in an *extension* of *simple content complex type* T_1 matches the value of a *name* attribute of a simple content complex type T_2, and there is an element E_1 that references T_1 and there is no element that references T_2, in the translation to C-XML, the object set S_1 for E_1 connects via binary relationship sets to each lexical object set that represents one of the attributes nested under T_2. In Figure 5.44, T_1 = *ShoeSizeType*, T_2 = *SizeType*, E_1 = *ShoeSize*, and S_1 = *ShoeSize*. Thus, as Figure 5.45 shows, *ShoeSize* connects to the lexical object set *Country*. The object set S_1 also connects via binary relationship sets to each lexical object set that represents one of the attributes nested under T_1. Thus, as Figure 5.45 shows, the non-lexical object set *ShoeSize* connects to the lexical object sets *Width* and *HeelHeight*. As before, we get the built-in type for the object set S_1 by unfolding the *base* in T_1.

5.2.14 Complex Content Complex Type

Figure 5.48 shows a declaration of *complexContent complexType* in XML Schema. Figure 5.49 shows an example of using *complexContent* in an XML-Schema instance, and Figure 5.50 shows a translation of Figure 5.49. In the following, we give the translation details of the component parts of *complexContent*.

```
<complexContent
 id = ID
 mixed = boolean>
 Content:(annotation?,(restriction | extension))
</complexContent>
```

Figure 5.48: Complex Content Declaration.

- *id*. As always, this is implicit—not mapped.
- *mixed*. The default value for this attribute is *false*. The value of the mixed attribute of a complex content component in the extension of a complex content complex type T_1 must match the value of the mixed attribute of the complex content T_2 that matches the name in the base of T_1. In Figure 5.49, T_1 =

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
 3    <xs:element name="OrderForm">
 4       <xs:complexType>
 5          <xs:choice>
 6             <xs:element name="Order" type="OrderType"/>
 7             <xs:element name="BookOrder" type="BookOrderType"/>
 8          </xs:choice>
 9       </xs:complexType>
10    </xs:element>
11    <xs:complexType name="OrderType" mixed="true">
12       <xs:sequence>
13          <xs:element name="Name" type="xs:string"/>
14          <xs:element name="OrderID" type="xs:positiveInteger"/>
15          <xs:element name="ShipDate" type="xs:date"/>
16       </xs:sequence>
17    </xs:complexType>
18    <xs:complexType name="BookOrderType">
19       <xs:complexContent mixed="true">
20          <xs:extension base="OrderType">
21             <xs:sequence>
22                <xs:element name="BookName" type="xs:string"/>
23                <xs:element name="Author" type="xs:string"/>
24                <xs:element name="PublicationDate" type="xs:date"/>
25             </xs:sequence>
26          </xs:extension>
27       </xs:complexContent>
28    </xs:complexType>
29 </xs:schema>
```

Figure 5.49: Example of Extension of Complex Content under Complex Type in XML Schema.

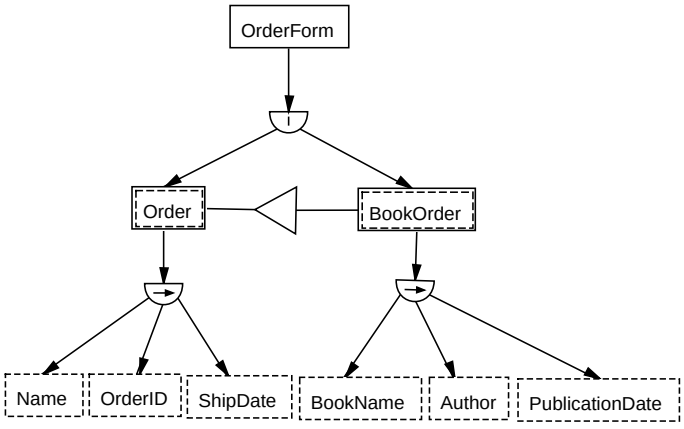


Figure 5.50: Translated C-XML Model Instance of Figure 5.49

BookOrderType (Lines 18–28), $T_2 = OrderType$ (Lines 11–17); the mixed attribute of both is set to *true*. When the value of the attribute *mixed* is set to *true* in the complex content component of a complex type T_1, in the translation to C-XML the object set E_1 that references the complex type T_1 becomes mixed. In Figure 5.49, $T_1 = BookOrderType$, $E_1 = BookOrder$. Thus, as Figure 5.50 shows, the object set *BookOrder* is mixed.

As Figure 5.48 shows, in XML Schema we provide the content of *complexContent* either by *restriction* or *extension*. We explain the translation of these elements respectively in the subsections below.

Derivation by Restriction of Complex Content

Figure 5.51 shows definition details for *restriction* of *complexContent* for *complexType* in XML Schema. *Restriction* allows the addition of new constraints to both the attributes and child elements in complex content structures. The new content is entirely described under restriction and must represent a restriction of what was allowed by the base content model (i.e., any content valid for the restricted type must also be valid for the base type).

```
<restriction
 id = ID
 base = QName>
 Content: (annotation?,(group | all | choice | sequence)?,
          ((attribute |attributeGroup)*,anyAttribute?))
</restriction>
```

Figure 5.51: Restriction on Complex Content Declaration.

Figure 5.52 shows an XML Schema example using *complexType* by *restriction*. An element *Canadian-Address* in Line 7 has a *complexType*, *Canadian-AddressType*. The *complexType*, *Canadian-AddressType*, in Lines 19–30 is a *complexType* derived by restriction from another *complexType*, *AddressType*. The *complexType AddressType* in Lines 11–18 has a sequence of the following elements: *Street*, *City*, *PostalCode*, and *Country*. The complex type *Canadian-AddressType* in Lines 19–30 has the same elements that complex type *AddressType* in Lines 11–18 has with additional restrictions. *Street* in Line 13 is optional, while in Line 23 it is required. *PostalCode* in Line 15 is

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
 3    <xs:element name="Document">
 4       <xs:complexType>
 5          <xs:choice>
 6             <xs:element name="Address" type="AddressType"/>
 7             <xs:element name="Canadian-Address" type="Canadian-AddressType"/>
 8          </xs:choice>
 9       </xs:complexType>
10    </xs:element>
11    <xs:complexType name="AddressType">
12       <xs:sequence>
13          <xs:element name="Street" type="xs:string" minOccurs="0"/>
14          <xs:element name="City" type="xs:string"/>
15          <xs:element name="PostalCode" type="xs:integer"/>
16          <xs:element name="Country" type="xs:string"/>
17       </xs:sequence>
18    </xs:complexType>
19    <xs:complexType name="Canadian-AddressType">
20       <xs:complexContent>
21          <xs:restriction base="AddressType">
22             <xs:sequence>
23                <xs:element name="Street" type="xs:string"/>
24                <xs:element name="City" type="xs:string"/>
25                <xs:element name="PostalCode" type="xs:positiveInteger"/>
26                <xs:element name="Country" type="xs:string" fixed="Canada"/>
27             </xs:sequence>
28          </xs:restriction>
29       </xs:complexContent>
30    </xs:complexType>
31 </xs:schema>
```

Figure 5.52: Example of Restriction of Complex Content under Complex Type in XML Schema.

integer, while in Line 25 it is *positiveInteger*. *Country* in Line 16 can be any string, while in Line 26 it is fixed to be *Canada*.

In the following, we give the translation details of the component parts of the derivation of *complexContent complexType* by *restriction*.

- *id*. As always, this is implicit—not mapped.

- *base*. When the value of the base attribute in a restriction of *complex content complex type* T_1 matches the value of a name attribute of a complex type T_2 and there is an element E_1 whose type is T_1 and an element E_2 whose type is T_2, in the translation to C-XML, E_1 becomes a nonlexical specialized object set S_1 of the nonlexical object set S_2 that represents E_2. In Figure 5.52, T_1 = *Canadian-Address*, T_2 = *AddressType*, E_1 = *Canadian-Address*, E_2 = *Address*, S_1 = *Canadian-AddressType*, and S_2 = *Address*. Thus, as Figure 5.53 shows, the object set *Canadian-Address* is a specialization of the object set *Address*. The object set S_1 connects to object sets that represent elements or attributes nested under the restriction content in T_1. Thus, as Figure 5.53 shows, the

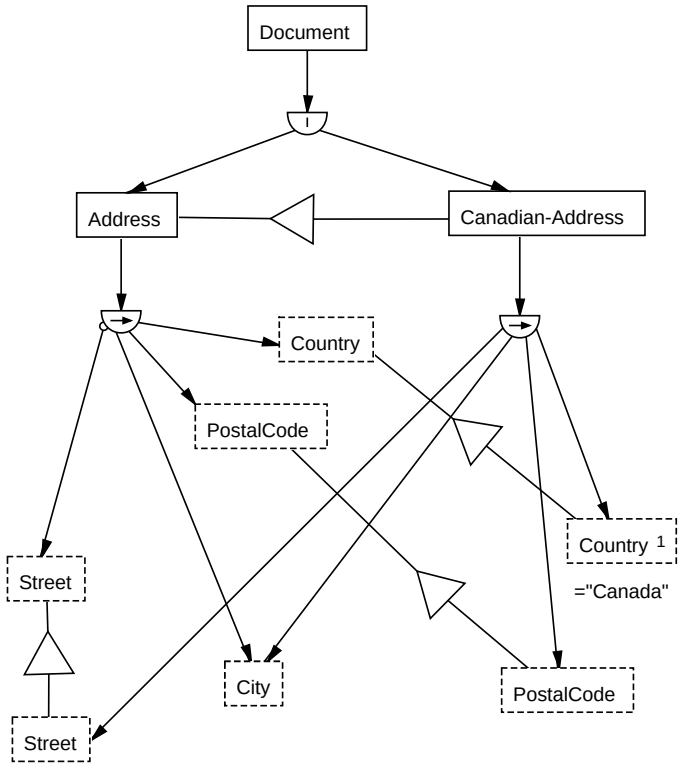


Figure 5.53: Translated C-XML Model Instance of Figure 5.52

object set *Canadian-Address* connects to the object sets *Street*, *City*, *PostalCode*, and *Country* via a sequence. The object set S_2 connects to object sets that represent elements or attributes nested under the extension content in T_2. Thus, as Figure 5.53 shows, the object set *Address* connects to the object sets *Street*, *City*, *PostalCode*, and *Country* via a sequence. Each element or attribute that has restrictions in T_2 becomes a specialization object set that represents its corresponding element or attribute in T_2. Thus, as Figure 5.53 shows, generalization/specialization constraints appear between the object set *Street* that connects by sequence to the object set *Address* and the object set *Street* that connects by sequence to the object set *Canadian-Address*, between the object set *PostalCode* that connects by sequence to the object set *Address* and the object set *PostalCode* that connects by sequence to the object set *Address*, and between the object set *Country* that connects by sequence to the object set *Address* and the singleton object set *Country* (fixed to the object "Canada") that connects by sequence to the object set *Canadian-Address*.

Derivation by Extension of Complex Content

Figure 5.54 shows definition details for the derivation of *complexContent complexType* by *extension* in XML Schema which allows the addition of elements and attributes to the *base* type. Figure 5.49 shows an example using derivation of *complexContent complexType* by *extension*, where an element *BookOrder* in Line 7 has complex type *BookOrderType*. *BookOrderType* in Lines 18–28 is a complex type that embeds a *complexContent* derived by extension from another *complexType*, *OrderType*. The *OrderType* in Lines 11–17 has a sequence of three elements: *Name*, *OrderID*, and *ShipDate*. *BookOrderType*, in addition being an extension of *OrderType*, it also has a sequence of three elements *BookName*, *Author*, and *PublicationDate*. Figure 5.50 shows a C-XML translation of Figure 5.49.

```
<extension
 id = ID
 base = QName>
 Content: (annotation?, ((group | all | choice | sequence)?,
          ((attribute | attributeGroup)*, anyAttribute?)))
</extension>
```

Figure 5.54: extension on Complex Content Declaration.

- *id*. As always, this is implicit—not mapped.
- *base*. When the value of the *base* attribute in an extension of *complex content complex type* T_1 matches the value of a name attribute of a complex type T_2 and there is an element E_1 whose type is T_1 and an element E_2 whose type is T_2, in the translation to C-XML E_1 becomes a nonlexical specialized object set S_1 of the nonlexical object set S_2 that represents E_2. In Figure 5.49, T_1 = *BookOrderType*, T_2 = *OrderType*, E_1 = *BookOrder*, E_2 = *Order*, S_1 = *BookOrder*, and S_2 = *Order*. Thus, as Figure 5.50 shows, the object set *BookOrder* is a specialization of the object set *Order*. The object set S_1 connects to object sets that represent elements or attributes nested under the extension content in *BookOrderType*. Thus, as Figure 5.50 shows, the object set *BookOrder* connects to the object sets *BookName*, *Author*, and *PublicationDate* via a sequence. The object set S_2 connects to object sets that represent elements

or attributes nested under the extension content in *OrderType*. Thus, as Figure 5.50 shows, the object set *Order* connects to the object sets *Name*, *OrderID*, and *ShipDate* via a sequence.

5.3 Conclusion

In this chapter we discussed in detail the translation of XML Schema to C-XML. For each component of XML Schema, we gave a suitable mapping for it and for each of its attributes. Thus, we can automatically convert any XML-Schema instance to a C-XML conceptual-model instance. Hence, we can view the structural components graphically in a two-dimensional layout, and we can access type details for each component by displaying its data frame as a filled-in form. We therefore can view an XML-Schema instance graphically and at a higher level of abstraction.

Chapter 6

Translating Conceptual XML to XML Schema

6.1 Introduction

XML, XML Schema, and the web have precipitated a change in information systems design. Many designers have shifted their work from traditional conceptual models to XML Schema to create XML-based applications. But XML Schema suffers from the same problem as other low-level data-definition languages such as SQL—it lacks a sufficiently intuitive two-dimensional visualization of object sets, relationship sets, and constraints for modeling real-world data, and it does not raise the level of abstraction beyond a linear list of textual statements with which designers can envision their data. Designers need conceptual models to graphically represent XML data at a higher level of abstraction, but they also need to be able to translate their conceptual models to XML Schema.

Elsewhere, we [Chapter 4] and others [14, 34] have adapted and defined suitable conceptual models for use with XML. Here we present a methodology to automatically translate a conceptual model to W3C XML Schema. The methodology guarantees that the XML-Schema instance resulting from the translation of a conceptual-model instance properly captures and represents both the data contained within the conceptual-model instance and, to the extent possible, the constraints imposed over the data.

The particular conceptual model we consider here for translation is C-XML [Chapter 2, Chapter 4]. C-XML has a rich set of modeling constructs, and it is fully compatible with XML Schema. Indeed, we can represent any XML-Schema instance in C-XML and preserve both containers for its data instances and all its constraints [Chapter 5]. C-XML is also equivalent in modeling power to both ER-XML [Chapter 4] and UML-XML [Chapter 4], which are both fundamentally grounded in C-XML. Thus, by translating C-XML to XML Schema, we also provide a way to translate ER-XML and UML-XML to XML Schema.

A number of other researchers have studied how to transform conceptual models into XML Schema. Both Carlson [10] and Routledge, et al. [32] describe how to translate a UML model instance into an XML Schema instance. In another study Bird, et al. explain how to translate Object Role Modeling into XML Schema [5]. A study by Pigozzo and Quintarelli [30] describes how to translate an ER model instance into an XML Schema instance. Comparing our approach to translation with these approaches, we observe the following.

- Their work explains how to translate one of the traditional conceptual models, UML, Object Role Modeling, or ER, to XML Schema. These traditional conceptual models have a number of limitations (see Chapter 4) when it comes to describing some of the conceptual structures that XML Schema provides; they are unable to (1) order lists of concepts, (2) choose alternative concepts from among several, (3) declare nested hierarchies of information, (4) specify mixed content, and (5) use content from another data model. We use an augmented conceptual model (C-XML) that includes the common data-modeling features found in traditional conceptual models, as well as these conceptual structures found in XML Schema.

- Their work fails to address and discuss many translation details. They only discuss the translation to XML Schema in a general way without walking the reader step-by-step through the process of generating a valid XML Schema instance.

- They fail to provide implementations for their work. Our implementation fully translates any C-XML model instance into a valid XML-Schema instance.

In the translation from C-XML to W3C XML Schema we must consider the following challenging issues:

- XML Schema has a hierarchical structure, while a particular conceptual-model instance may have no explicit hierarchal structure. Converting non-hierarchial structure to a hierarchial 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 artifacts to satisfy XML Schema's syntactic requirements (e.g. a sequence among

data items when the conceptual-model instance has no requirement for a sequence). Also, because of XML-Schema limitations the translation sometimes cannot capture all the constraints of a C-XML model instance (e.g. some cardinality constraints).

- The conceptual-model instance may contain a variety of conceptualizations: hypergraphs representing interrelated object and relationship sets; generalization/speialization hierarchies with union, mutual-exclusion, and partition constraints; hierarchies of sequence and choice structures; and mixed textual/conceptual structures. Translating these conceptualizations individually is a challenge, and translating conceptual-model instances with a mixture of these conceptualizations presents an even greater challenge.

- Often multiple translations are possible. Deciding on a default translation is sometimes difficult because the alternatives are equally reasonable. Having the user decide whenever there is an alternative is likely to be overburdensome. Having the user choose broadbased defaults in advance may help, but does not allow for fine-grained tuning. In our discussion here we point out the alternatives, but our implementation only allows the user to have fine-grained control over a few of them.

We address these issues and present our contribution of translating from C-XML to XML Schema as follows. Section 6.2 gives the translation when C-XML has basic conceptual structures, which include object sets and binary and n-ary relationship sets. Section 6.3 presents the translation of generalization/specialization hierarchies. Section 6.4 presents the translation when the conceptual model instance has sequence and choice structures. Section 6.5 summarizes, gives the status of our implementation, and considers future work.

6.2 Basic Conceptual Structures

In this section we present our translation from a basic C-XML model instance to an XML-Schema instance. A basic C-XML model instance contains object sets and binary and n-ary relationship sets. An XML-Schema instance generated from a basic C-XML model instance must provide for the objects and relationships representable in the C-XML object and relationships sets and must capture all possible C-XML model-instance constraints.

C-XML is a hypergraph-based conceptual model that defines structure in terms of *object sets*, *relationship sets*, and *constraints* over these object and relationship sets.

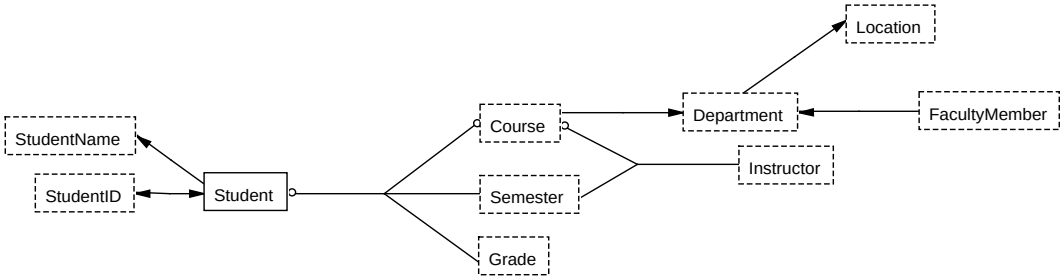


Figure 6.1: Basic C-XML Model Instance.

Figure 6.1 shows an example. An object set with a solid border is a nonlexical object set (e.g. *Student* in Figure 6.1); an object set with a dashed border is a lexical object set (e.g. *Course* in Figure 6.1). With each object set we can associate a data frame to provide a rich description of its value set and other properties. Lines connecting object sets are *relationship sets* (e.g. the line connecting *Student* and *Name* and the hyper-line connecting the object sets *Student*, *Course*, *Grade*, and *Semester* in Figure 6.1). A *participation constraint* specifies how many times an object in an object set may participate in relationships in a connected relationship set. For the most common participation constraints (*0:1*, *1:1*, *0:**, and *1:**), C-XML uses graphical notation as a shorthand. An "o" on a connecting relationship-set line designates *optional participation*, while the absence of an "o" designates *mandatory* participation. Thus, for example, the C-XML model instance in Figure 6.1 declares that an *Instructor* must teach at least one *Course* in some *Semester* but that a *Course* need not be taught by any *Instructor* in a specific *Semester*. An arrowhead specifies a *functional constraint*, limiting participation of objects on the tail side of the arrow to participate at most once. Thus, Figure 6.1 declares that a *Student* has one *Name* and that the *Department* has one *Location*. It also declares that the *Student* object set is in a one-to-one correspondence with the *StudentID* object set.

Our translation method starts by applying an algorithm, labeled HST, to convert a conceptual-model hypergraph to a forest of scheme trees [21].[1] By observing many-one cardinality constraints, the HST algorithm finds hierarchical structures, making them as large as possible without introducing redundancy. By observing mandatory/optional constraints, the algorithm also ensures that all values populating a C-XML model instance can be represented in instance trees complying with the scheme-tree forest.

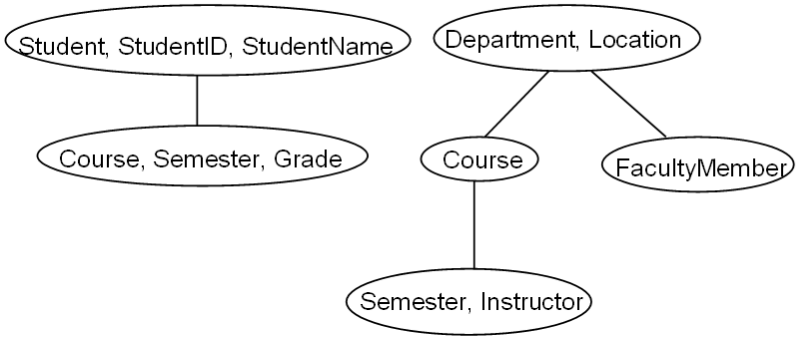
[1]To avoid the long explanation, each time we refer to the algorithm in this chapter, we designate it the *HST* algorithm (Hypergraph-to-Scheme-Tree translation algorithm).

An application of the HST algorithm to the C-XML model instance in Figure 6.1 generates the forest of scheme trees in Figure 6.2(a). Starting with *Student*, the HST algorithm adds *StudentID* and *Name* in the same node as *Student*. Both functionally depend only on *Student* and cannot cause redundancy. Since, *Course-Semester-Grade* triples do not functionally depend on *Student*, we form a node with these triples and make it a child node of the root node. The algorithm then begins building a new scheme tree starting with *Department*. The algorithm adds *Location* in the same node, as it functionally depends on *Department*. Since *Course* does not functionally depend on *Department*, *Course* becomes a child node of the root node *Department*. Because the relationship between *Department* and *Course* is one-many, we may continue along relationship-set lines and add an additional node below the *Course* node. We thus add in a new child node consisting of *Semester* and *Instructor*. Since *FacultyMember* does not functionally depend on *Department*, *FacultyMember* becomes a child node of the root node *Department*. We thus obtain the scheme tree forest in Figure 6.2(a). Figure 6.2(b) shows the textual representation of this scheme-tree forest.

The HST algorithm allows several degrees of freedom in forming a scheme-tree forest. Depending on where the algorithm starts, the algorithm can generate different scheme trees. If we start first with *Department*, the algorithm would continue on from *Course* by adding the node (*Student, Semester, Grade*)* below the node whose only element is *Course*. This is satisfactory in the sense that the scheme tree permits no redundancy, but may not be satisfactory for a user who may expect to view the data about the courses students have taken from a student's perspective. Our implementation allows designers to choose starting nodes. For our example, we assume that a designer first chooses *Student* as a starting node, and then after the algorithm produces the first scheme tree, chooses *Department* as the next starting node.

The HST algorithm also permits alternative node configurations for non-functional n-ary ($n \geq 3$) relationship sets. There is no redundancy penalty, for example, for splitting the node (*Semester, Instructor*)* into two nodes (*Semester,* (*Instructor*)*)*. Our current implementation does not give designers this flexibility, choosing instead by default, to group object sets into a fewer number, rather than a greater number, of nodes.

After generating the scheme tree forest, we construct the XML-Schema instance. We explain how the construction works bottom-up as follows. We first explain how to translate an individual object set. We next explain how to translate an



(a)

(Student, StudentID, StudentName, (Course, Semester, Grade)*)*

(Department, Location, (Course, (Semester, Instructor)*)*, (FacultyMember)*)*

(b)

Figure 6.2: Generated Forest of Scheme Trees for Figure 6.1.

individual node in a scheme tree. Since an individual node may have children, which are included in the translation of an individual node, the translation leads to the translation of an individual scheme tree. We then explain how to create a root node for the generated XML-Schema instance, which ties all the scheme trees together. Finally, we explain how to add global uniqueness constraints.

Individual Object Sets

In our implementation, we usually translate a lexical object set to an XML-Schema attribute, and we always translate a nonlexical object set to an XML-Schema element. An alternative to translating lexical object sets as attributes is to translate them as elements. Sometimes, when the structure dictates, we must use elements for lexical objects sets, but our preference is to use attributes whenever possible. A nonlexical object-set element has a *complexType* container for its content. As part of the content, we choose to generate an *OID* attribute to represent the objects within the nonlexical object set. If a lexical object set appears more than once in the scheme-tree forest, we declare the translated XML-Schema attribute globally in the generated XML-Schema instance, and we reference it using the attribute *ref* for each local appearance in the XML-Schema instance. If a nonlexical object set appears

```
<xs:attribute name="Grade" type="xs:string"/>

                (a)

<xs:attribute name="Course" type="xs:string"/>

                (b)

<xs:attribute ref="Course"/>

                (c)

<xs:element name="Student">
   <xs:complexType>
      ...
      <xs:attribute name="StudentOID">
   </xs:complexType>
</xs:element>

                (d)
```

Figure 6.3: Translation of Several Individual Object Sets.

more than once in the scheme-tree forest, we declare the attribute *OID* nested within the translated XML-Schema element for that nonlexical object set globally in the generated XML-Schema instance, and we reference it using the attribute *ref* for each local appearance in the XML-Schema instance.

To complete the declaration for an object set, we obtain information from its associated data frame. Every object set in a C-XML model instance has an associated data frame that contains the fields *Type name*, *Default value*, *Fixed value*, *Block*, *Final*, *Form*, and *Nillable*. When a field F has a value v, we add to the translated attribute or element declaration the qualifying attribute for F and the value v.

Figure 6.3 shows how we translate several individual object sets. Figure 6.3a shows the translation of the lexical object set *Grade* as a simple attribute declaration; its *name* is its object set name, and its *type* is *string*. Figure 6.3 also shows the translation of the object set *Course*, which appears twice in the scheme-tree forest. Figure 6.3b shows its global definition, and Figure 6.3c shows its local definition with a *ref* attribute, which references the global definition. Figure 6.3d shows the translation of the nonlexical object set *Student* as an element with *complexType* content. Observe that its *OID* attribute has the name *StudentOID*, a concatenation of the name of the nonlexical object set and "OID". We form all *OID* names by concatenating the object set name and "OID".

Nodes

Since each scheme-tree node denotes a set of tuples, we must have a container for the set as well as a container for an individual tuple. Each container requires a name. 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. (We always form plurals by adding the letter *s* even when adding an *s* does not yield the proper English plural.) Thus, for example, since the key for the *Department-Location* node in Figure 6.2 is *Department*, we name the container for its individual tuples *Department* and the container for its set of tuples *Departments*. Similarly, since the key for the *Semester-Instructor* node in Figure 6.2 is the composite key consisting of *Semester* and *Location*, we name the container for its individual tuples *Semester-Instructor* and the container for its set of tuples *Semester-Instructors*. If a node has multiple, minimal keys, we choose one arbitrarily except that we prefer nonlexical keys over lexical keys. Thus, for example, we choose *Student* and *Students* as the names for the *Student-StudentID-StudentName* node in Figure 6.2.

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 available to us; we cannot choose the *all* structure because it allows its children to appear at most one time within the *all* structure. The container element for an individual tuple, on the other hand, contains at most one instance of each object set and each child of the tuple. We thus use the *all* structure for the elements representing the child nodes of the node being built. If a node contains a nonlexical object set, it is usually the key for the node. If so, its translation, which is an element with *complexType* content containing an *OID* attribute to represent its objects, serves as the container for the individual tuple. If not, an *OID* attribute alone is sufficient to represent the nonlexical object set. We add any such *OID* attributes along with attributes for lexical object sets in the *complexType* content of the container element for an individual tuple. Figure 6.4, for example shows the construction of the *Student* scheme tree in Figure 6.2a. Since *Student* is nonlexical (see Figure 6.1) and is a key for the root node (see Figure 6.2), *Student* becomes the container for the individual tuples of the root node. The *complexType*

```
<xs:element name="Students">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Student" maxOccurs="unbounded">
        <xs:complexType>
          <xs:all>
            <xs:element name="Course-Semester-Grades">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Course-Semester-Grade" minOccurs="0" maxOccurs="unbounded">
                    <xs:complexType>
                      <xs:attribute ref="Course" use="required"/>
                      <!--C-XML: forall x (Course(x) =>
                          exists [0:*] <y, z, w> (Course(x)Student(y)Semester(z)Grade(w)))-->
                      <xs:attribute ref="Semester" use="required"/>
                      <xs:attribute name="Grade" type="xs:String" use="required"/>
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:all>
          <xs:attribute name="StudentOID" type="xs:string" use="required"/>
          <xs:attribute name="StudentID" type="xs:string" use="required"/>
          <xs:attribute name="StudentName" type="xs:string" use="required"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figure 6.4: Portion of XML-Schema Instance that Represents the Content of the first scheme tree.

structure under *Student* provides for the information for a single student. The parent structure for *Student* is *Students*, which provides for the set of students. The node has one child node, the *Course-Semester-Grade* node, which we formulate as another node nested under the *all* structure of the *Student* element. The remaining lexical object sets, *StudentID* and *Name*, become attributes nested under the *Student* structure.

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. The default value for both *minOccurs* and *maxOccurs* is "1". Every XML-Schema attribute declaration specifies whether it is "required" or "optional" in its *use* attribute. The default value for the attribute *use* is *optional*. We now explain how we obtain the values for *minOccurs*, *maxOccurs*, and *use* for (1) the container element for a set of tuples for a node, (2) the container element for an individual tuple for a node, and (3) attributes inside a node.

1. *Container elements for a set of tuples.* 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. In Figure 6.4, neither the element *Students* and nor the element *Course-Semester-Grades* has *minOccurs* or *maxOccurs*, which indicates that the default values must hold.

2. *Container elements for an individual tuple for a node.* Normally, a set of individual tuples may contain one or more tuples. Thus, *minOccurs* is "1" and *maxOccurs* is "unbounded", unless the conceptual model constrains the set to have a different minimum or maximum number of tuples. For example, the *minOccurs* value for *Student* in Figure 6.4 is "1" (the default), and the *maxOccurs* value for *Student* is "unbounded". On the other hand, the *minOccurs* value for *Course-Semester-Grade* in Figure 6.4 is "0" and the *maxOccurs* value for *Course-Semester-Grade* is "unbounded". We obtain the value "0" for the *minOccurs* by observing that the node *Course-Semester-Grade* is a child node for the node *Student-StudentID-StudentName*; then since *Student* has an optional connection to the object sets *Course*, *Semester*, and *Grade* the *minOccurs* value is zero.

3. *Attributes inside a node.* Normally, the value for *use* is "required" unless the conceptual model constrains the *use* to be "optional". For example, in Figure 6.4, the *use* for the attributes *StudentID* and *Name* is "required" because in Figure 6.1 *Student* has mandatory constraints in the relationship sets between *Student* and *StudentID* and between *Student* and *Name*. The *Course-Semester-Grade* container that includes *Course*, *Semester*, and *Grade* has *minOccurs="0"* to account for the optional participation of the *Course-Semester-Grade* relationship set. Each instance of the relationship set, however, requires a connection among *Student*, *Course*, *Semester*, and *Grade* and thus the *use* for each of the attributes *Course*, *Semester*, and *Grade* is "required".

Since XML Schema has a hierarchial structure, we can only capture participation constraints in the conceptual model instance for parent elements. By default, the nesting structure in an XML-Schema instance makes the participation constraint for a child element within a parent element have a minimum constraint of one and a maximum constraint of unbounded. XML Schema provides no way to capture any constraint other than this default constraint. Thus, we capture constraints that differ from the default in a special comment. We prefix special 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 an equivalent predicate-calculus expression [18].) For example, to declare that the participation of *Course* in the n-ary relationship set in Figure 6.1 among *Students*, *Course*, *Semester*, and *Grade* is optional, we write the comment that appears in Line 13 of Figure 6.4. The comment

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

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

Root Elements

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. We choose to give the root element the name *Root*, but a user may rename it, choosing a name that represents the entire conceptual model.

When we introduce a root element, we must nest the elements beneath the root element that represent the sets of tuples for each generated scheme tree. Thus, the root element is of *complexType*. Since there is no reason to sequence the scheme trees in any particular order and since there is only one instance of a scheme tree, we choose the *all* structure to nest the elements representing the scheme trees. Within the *all* structure, we nest the container element for the set of tuples for the root node of each generated scheme tree.

In our implementation we choose to declare the container element for the set of tuples for the root node of each generated scheme tree by using the *ref* attribute. We declare the entire content of the container element for the set of tuples for the root node of each generated scheme tree globally under the *schema* element and outside the *Root* element. The *ref* attribute references the container element for the set of tuples for the root node of each generated scheme tree. For example, the generated forest in Figure 6.2 has two scheme trees. Thus, as Figure 6.5 shows, we generate the root element *Root* and declare elements *Students* and *Departments* under the *all*

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="Root">
      <xs:complexType>
         <xs:all>
            <xs:element ref="Students"/>
            <xs:element ref="Departments"/>
         </xs:all>
      </xs:complexType>
   </xs:element>
   <xs:element name="Students">
      <xs:complexType>
       ...
      </xs:complexType>
      ...
   </xs:element>
   <xs:element name="Departments">
      <xs:complexType>
      ...
      </xs:complexType>
      ...
   </xs:element>
</xs:schema>
```

Figure 6.5: The Root Element for the Generated XML-Schema Instance.

structure using the *ref* attribute. The *ref* attribute references the global declarations *Students* and *Departments* under the *schema* element.

An alternative way of nesting each generated scheme tree is to place the entire content of the container element for the set of tuples for the root node of each generated scheme tree directly inside the *all* structure. We prefer referencing the scheme trees and declaring them outside the *Root* because this provides a quick and easy way for the reader to see the major structures of the XML-Schema instance.

Uniqueness Constraints

For each generated forest, we need to 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. We determine the scope of uniqueness of a key K within a node N by computing the functional closure of K, K^+, in the C-XML model instance. By definition of K being a key, K^+ includes the object sets of N. In addition, it may include the object sets in ancestor nodes of N. The *scope* of a key K is the highest level ancestor node A such that K^+ includes all object sets in all ancestor nodes of the node N in which K appears up to and including node A. A equal to N is common and happens when K^+ does not include the object sets in its parent. Because we construct scheme trees hierarchically, A is

Node	*Key*	*Functional Closure*	*Scope Element*
Student-StudentID-StudentName	StudentOID and StudentID	Student, StudentID, StudentName	Students
Course-Semester-Grade	Course-Semester-Grade	Course, Semester, Grade, Department, Location	Course-Semester-Grades
Department-Location	Department	Department, Location	Departments
Course	Course	Course, Department, Location	Departments
FacultyMember	FacultyMember	FacultyMember, Department, Location	Departments
Semester-Instructor	Semester-Instructor	Semester, Instructor	Semester-Instructors

Table 6.1: Keys for Figure 6.2.

often the root node of a scheme tree. In our example, we compute the closure for each key in each node in each scheme tree as Table 6.1 shows.

From Table 6.1, we can immediately create a key declaration as follows. We create the key structure within the scope element since the values of the key have to be unique within that element. Thus, for example, we create the key structure for *FacultyMembers* in *Departments* and the key structure for *Semester-Instructor* in *Semester-Instructors*. We list the key components in the *field* attributes in the key structure. When a key component is an attribute in an XML-Schema instance, it is prefixed with "@". Thus, the field component of the key declaration for "StudentOID" is "@StudentOID" and for "StudentID" is "@StudentID", and the field component for "Semester-Instructor" are the field components "@Semester" and "@Instructor". The *selector* field in the key structure specifies the scope. Since the key structure resides in the designated scope element, the path "." at the beginning of the *xpath* in the *selector* denotes the scope element. The remainder of the *xpath* consists of the container names down to, and including, the individual tuple for the key. Thus, for example, the *xpath* for the selector for *Semester-Instructor* is "./Semester-Instructor", and the *xpath* for the *FacultyMember* is "./Department/FacultyMembers/FacultyMember". The key structure has a *name* attribute—we concatenate the name of the key and *Key* with a hyphen between them. In our example there are seven keys to declare. As a complete example of one of them, Figure 6.6 shows the key structure for *FacultyMember* nested in the element *Departments*.

```
<xs:element name="Departments">
   <xs:complexType>
   ...
   </xs:complexType>
   <xs:key name="FacultyMember-Key">
      <xs:selector xpath="./Department/FacultyMembers/FacultyMember"/>
      <xs:field xpath="@FacultyMember"/>
   </xs:key>
</xs:element>
```

Figure 6.6: Generated Key Structure.

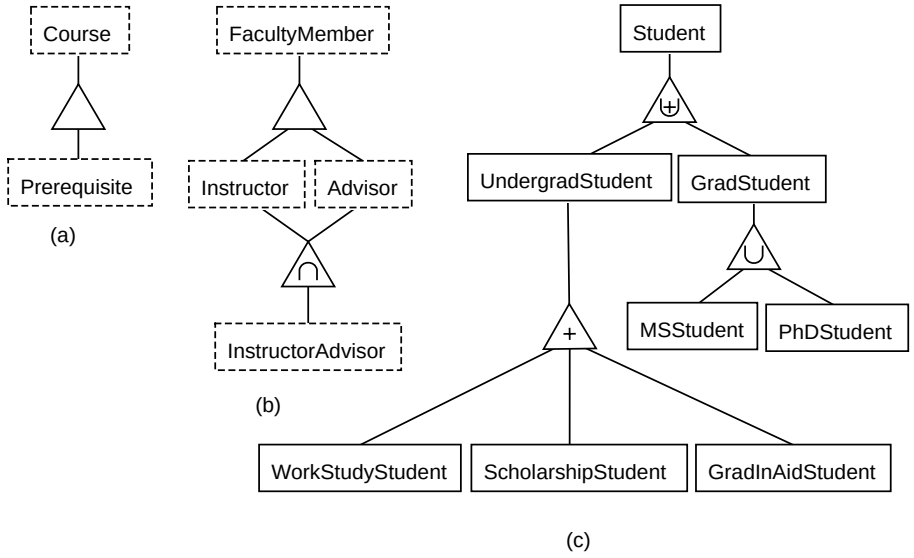


Figure 6.7: Several C-XML Generalization/Specialization Hierarchies.

6.3 Generalization/Specialization

Figure 6.7 shows several C-XML generalization/specialization hierarchies. The main idea in generalization/specialization is that a generalization object set is a superset of each of its specialization object sets. We choose to model this central idea using referential integrity constraints. In XML Schema, referential integrity constraints are declared by a *keyref* structure. 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.

Since XML-Schema requires us to make *keyref* declarations outside the scope of referenced key declarations, we create the *keyref* structure within the scope of the XML-Schema root element. Consider, for example, the *keyref* structure for *Prerequisite*. We list the *keyref* components in the *field* attributes in the *keyref* structure.

When a *keyref* component is an attribute in an XML-Schema instance, it is prefixed with "@". The field component of the *keyref* declaration for "Prerequisite", for example, is "@Prerequisite". The *selector* field in the *keyref* structure specifies the scope. Since the *keyref* structure resides in the root element, the path "." at the beginning of the *xpath* in the *selector* denotes the root element. The remainder of the *xpath* consists of the container names down to, and including, the individual tuple for the *keyref*. Thus, for example, the *xpath* for the selector for the element *Prerequisite* is "./Prerequisites/Prerequisite". The *refer* attribute in a *keyref* structure specifies the corresponding key definition. Thus, for example, the *refer* attribute for *Prerequisite* is *Course-Key*. To construct the *name* attribute, we concatenate the name of the *keyref*'s corresponding element with the string "-Keyref", Figure 6.9 shows the *keyref* structure for *Prerequisite*. The *keyref* declaration in Lines 10–13 references the *key* declaration in Lines 25–28. Hence, values for the *Prerequisite* attribute declared in Line 35 must be a subset of values for the *Course* attribute declared in Line 20.

When the generalization/specialization hierarchy has a *union* constraint, the set of objects in the generalization is a union of the specialization object sets. In XML Schema, there is no way to declare this constraint, but we can insert a comment line in an XML-Schema instance to indicate that the *union* constraint holds. For the generalization G with specializations S_1, ..., S_n we insert the comment

<!--C-XML: forall x (G(x) => (S_1*(x) or ... or* S_n*(x))-->*.

For the union constraint in Figure 6.7(c), for example, we write the constraint

<!--C-XML: forall x (GradStudentOID(x) =>
(MSStudentOID(x) or *PhDStudentOID(x))-->*.

When the generalization/specialization hierarchy has a *mutual-exclusion* constraint, the pairwise intersection of the specialization object sets must be empty. Since there is no way to declare this constraint in an XML-Schema instance, we insert a comment line to indicate that the *mutual-exclusion* constraint holds. For the generalization G with specializations S_1, ..., S_n we insert the comment

<!--C-XML: forall x (S_1*(x) => not* S_2*(x) and not* S_3*(x) and ... and not* S_n*(x))*
and ... and forall x (S_{n-1}*(x) => not* S_n*(x))-->*.

For the mutual-exclusion constraint in Figure 6.7(c), for example, we insert the comment

*<!--*C-XML*: forall x (WorkStudyStudentOID(x) => not ScholarshipStudentOID(x)*
and not GradInAidStudentOID(x)) and
forall x (ScholarshipStudentOID(x) => not GradInAidStudentOID(x))-->.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
   <xs:element name="Root">
      <xs:complexType>
         <xs:all>
           <xs:element ref="Courses"/>
           <xs:element ref="Prerequisites"/>
         </xs:all>
      </xs:complexType>
      <xs:keyref name="Prerequisite-Keyref" refer="Course-Key">
        <xs:selector xpath="./Prerequisites/Prerequisite"/>
        <xs:field xpath="@Prerequisite"/>
      </xs:keyref>
   </xs:element>
   <xs:element name="Courses">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="Course" minOccurs="1" maxOccurs="unbounded">
               <xs:complexType>
                  <xs:attribute name="Course" use="required"/>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
      </xs:complexType>
      <xs:key name="Course-Key">
         <xs:selector xpath="./Course"/>
         <xs:field xpath="@Course"/>
      </xs:key>
   </xs:element>
   <xs:element name="Prerequisites">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="Prerequisite" minOccurs="1" maxOccurs="unbounded">
               <xs:complexType>
                  <xs:attribute name="Prerequisite" use="required"/>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
```

Figure 6.8: XML-Schema Instance for the Translated Content for Figure 6.7(a).

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="Root">
      <xs:complexType>
         <xs:all>
           <xs:element ref="Students"/>
           ...
         </xs:all>
      </xs:complexType>
      <xs:keyref name="UndergradStudentOID-Keyref" refer="StudentOID-Key">
        <xs:selector xpath="./Students/Student"/>
        <xs:field xpath="@UndergradStudentOID"/>
      </xs:keyref>
      <xs:keyref name="GradStudentOID-Keyref" refer="StudentOID-Key">
        <xs:selector xpath="./Students/Student"/>
        <xs:field xpath="@GradStudentOID"/>
      </xs:keyref>
      <!--C-XML:forall x (StudentOID(x) => UndergradStudentOID(x) or GradStudentOID(x))-->
      <!--C-XML:forall x (UndergradStudentOID(x) => not GradStudentOID(x))-->
   </xs:element>
   <xs:element name="Students">
      <xs:complexType>
      ...
      </xs:complexType>
      <xs:key name="StudentOID-Key">
         <xs:selector xpath="./Student"/>
         <xs:field xpath="@StudentOID"/>
      </xs:key>
   </xs:element>
   <xs:element name="UndergradStudents">
   ...
   </xs:element>
   <xs:element name="GradStudents">
   ...
   </xs:element>
</xs:schema>
```

Figure 6.9: Translation of a Partition Constraint.

A partition constraint imposes both union and mutual exclusion conditions. We thus represent a partition constraint by inserting special C-XML comments for both union and mutual exclusion. Figure 6.9 shows the translation of the part of the generalization/specialization hierarchy in Figure 6.7(c) partitioning the *Student* object set into the *UndergradStudent* and *GradStudent* object sets. The *keyref* declaration in Lines 10–13 ensures that the set of *UndergradStudentOID* values is a subset of the set of values of *StudentOID* and the set of values of *GradStudentOID* is a subset of the set of the *StudentOID* values. We declare the union part of the partition constraint in Line 18 and the mutual-exclusion constraint in Line 19.

When an object set in a generalization/specialization hierarchy has multiple generalizations, the values of the specialization object set should be a subset of the values in each generalization object set. Figure 6.7(b) shows an example with multiple generalizations for the object set *InstructorAdvisor*, whose generalization object sets are *Instructor* and *Advisor*. Using two *keyref* declarations, we are able to constrain the values of objects in the specialization object set also to be values of objects in each generalization object set. In addition, when an object set in a generalization/specialization hierarchy has multiple generalizations with an *intersection* constraint, as is the case in Figure 6.7(b), the set of objects in the specialization must be exactly the intersection of the generalization object sets. Since in an XML-Schema instance there is no way to express this constraint, we insert a comment in the XML-Schema instance to declare it. Thus, for our example in Figure 6.7(b), we would add the comment

<!--C-XML: forall x (Instructor(x) and Advisor(x) =>
InstructorAdvisor(x))-->.

When we intermix generalization/specialization with other C-XML structures, we must be careful to ensure that we can continue to use *keyref* in XML Schema to express subset/superset constraints. We can if we ensure that every generalization element has a key structure. Hence, when we apply the HST algorithm, each generalization object set must be a key for its node. When forming a node within a scheme tree, if a generalization object set is to be added to the node and the object set is not a key within that node, the algorithm builds a new scheme tree starting with that generalization object set. This ensures that the generalization object set is a key inside the root node for the generated scheme tree, and also ensures that the XML-Schema instance has a key structure for the *keyref* structure.

An application of the HST algorithm to the C-XML model instance in Figure 6.10 generates the forest of scheme trees in Figure 6.11. Observe that the second

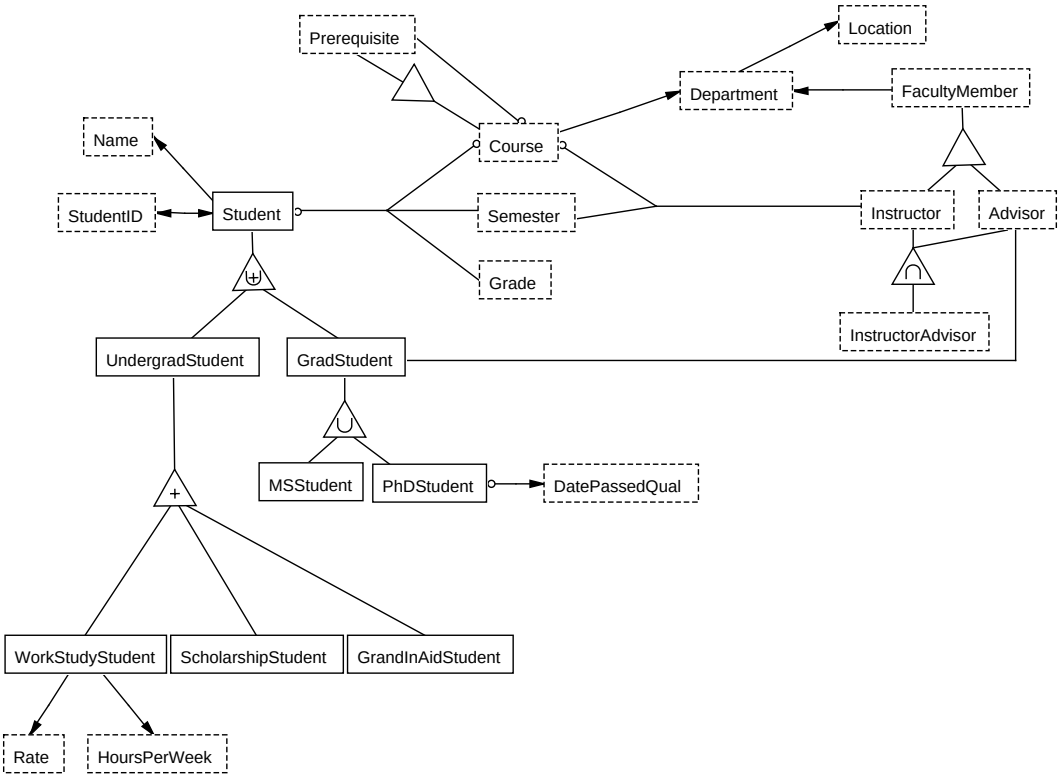


Figure 6.10: C-XML Model Instance Combining Basic Conceptual Structures and Generalization/Specialization Hierarchies.

```
(Student, StudentID, StudentName, (Course, Semester, Grade)*)*
(Department, Location, (Course,(Prerequisite)*)*, (FacultyMember)*)*
(WorkStudyStudent, HoursPerWeek, Rate)*
(Advisor, (GradStudent)*)*
(PhDStudent, DatePassedQual)*
(UndergradStudent)*
(MSStudent)*
(ScholarshipStudent)*
(GrandInAidStudent)*
(InstructorAdvisor)*
(Instructor, (Course, Semester)*)*
```

Figure 6.11: Generated Scheme Tree for Figure 6.10.

scheme tree (*Department, Location,* (*Course,*(*Prerequisite*)*)*, (*FacultyMember*)*)* does not include the node *Semester-Instructor* under *Course* as does the scheme tree in Figure 6.2. If we add the node *Semester-Instructor* under *Course*, the generalization object set *Instructor* is not a key within this node. Instead, we generate a new scheme tree (*Instructor,* (*Course, Semester*)*)* where the generalization object set *Instructor* is a key.

After generating the forest of scheme trees, we first construct the XML-Schema instance as explained in Section 6.2. We then add *keyref* structures under the *Root* element of the generated XML-Schema instance as explained in Section 6.3.

6.4 Sequence and Choice

Figure 6.12 shows an example of a C-XML model instance that contains sequence and choice structures along with some other C-XML structures. Initially, it might appear that because C-XML sequence and choice structures are essentially XML-Schema sequence and choice structures, the translation should be straightforward. Unfortunately, two major differences make the translation far less than straightforward. These differences include (1) our desire to translate so that the resulting XML-Schema instance allows no possible redundancy and (2) complications that arise because C-XML permits richer sequence and choice structures and richer constraints in sequence and choice structures than XML Schema. For redundancy, for example, observe that in Figure 6.12 if two students have the same address, and if we write corresponding XML-Schema structures with *Address* nested under *Student* and *City*, *State*, and *ZipCode* nested under *Address*, then the *City*, *State*, and *ZipCode* values will appear redundantly, once for each shared address. For richer structures, for example, observe that in Figure 6.12 the object set *Student* is a parent of two sequence/choice hierarchies. Moreover, observe also in Figure 6.12 that in a C-XML model instance an object set that is a child of a sequence or choice structure can connect to other object sets via binary or n-ary relationship sets. Neither of these structures is possible in XML-Schema. For richer constraints, unlike XML Schema, C-XML allows for sequence and choice structures to have a minimum and maximum number of occurrences within the relationship set between the parent and the sequence or choice. It also allows for a sequence's or a choice's child to have a minimum and maximum number of occurrences within the relationship set between the sequence or choice and the child. The optional constraint near the *State* side in the relationship set between the sequence and the *State* in Figure 6.12 is an example of a child having a declared minimum occurrence of zero.

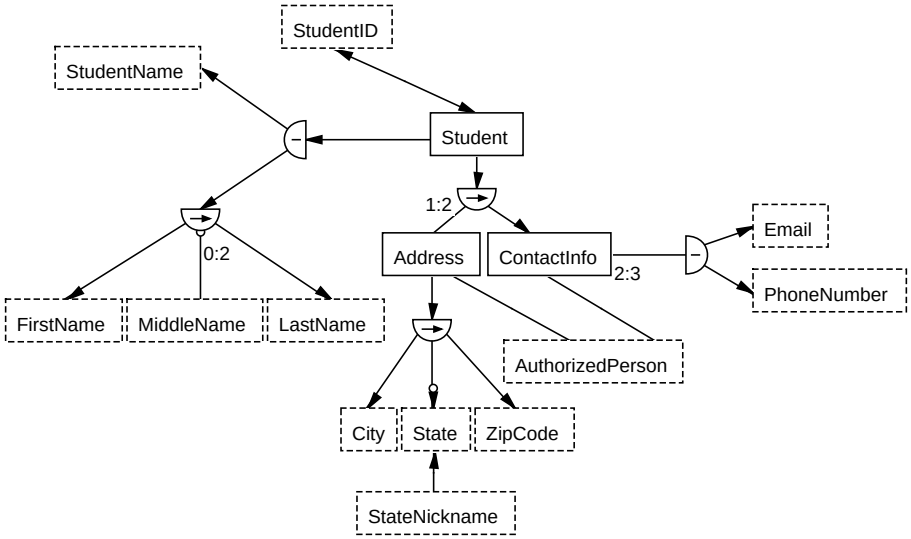


Figure 6.12: Sample Sequence and Choice Structures.

All these difficulties cause the translation of sequence and choice structures to differ markedly from standard XML Schema sequence and choice structures. The basic idea for translation is to consider sequence and choice structures as ordinary relationship sets. This lets us run the HST algorithm in the usual way, which leads to a resolution of both the redundancy problem and the structural problems. To resolve the constraint problems, the system keeps track of the original sequence and choice structures so that they can be appropriately inserted, and it keeps track of the original minimum and maximum occurrences so that, when possible, they can be appropriately inserted, and, when not possible, they can be recorded as special C-XML comments.

To do the translation correctly, we first replace sequence and choice structures between a parent object set P and a child object set C with a binary relationship set that connects P and C. Figure 6.13 shows Figure 6.12 with binary relationship sets in place of sequence and choice structures. We compute the participation constraints between P and C by considering the constraints on the path of one or more sequence and choice structures between P and C.

To compute the participation constraint for the parent side, we multiply the minimums of the participation constraints along the path from parent to child. Similarly, we multiply the maximums to compute the maximum. For the child side we do the same—multiply minimums together for the minimum and maximums together for the maximum. To obtain the participation constraint of 0:2 on the *Student* side

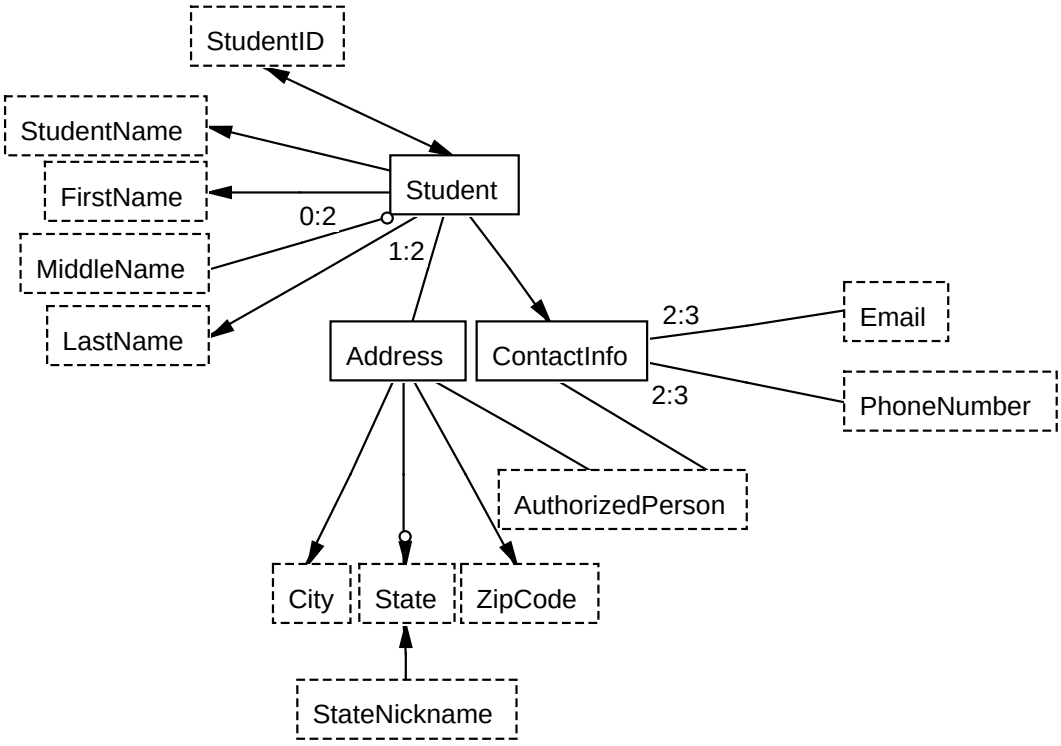


Figure 6.13: Sequence and Choice Structures Replaced with Binary Relationship Sets.

in the binary relationship set between *Student* and *MiddleName*, for example, we multiply respective minimums and maximums in Figure 6.12 of 1:1 (for *Student* in the relationship set between the *Student* and the choice), 1:1 (for the choice structure in the relationship set between the choice and the sequence structures), and 0:2 (for the sequence structure in the relationship set between the sequence structure and the object set *MiddleName*). Similarly, to obtain the participation constraint of 1:* on the *MiddleName* side in the binary relationship set between *Student* and *MiddleName*, we multiply respective minimums and maximums of 1:* (for *MiddleName* in the relationship set between the sequence structure and the object set *MiddleName*), 1:* (for the sequence structure in the relationship set between the choice and sequence structures), and 1:* (for the choice structure in the relationship set between *Student* and the choice structure). (Multiplying "*" by anything greater than 0, of course, results in "*".)

Next, we apply the HST algorithm. When applying the HST algorithm, we must be sure to make it possible to reinsert the sequence and choice structures. We do so by making each object set in the C-XML model instance that is a parent of

```
(Student, StudentID, StudentName, FirstName, LastName, ContactInfo,
    (MiddleName)*, (Address)*)*
(Address, City, State, ZipCode, (AuthorizedPerson)*)*
(ContactInfo, (Email)*, (PhoneNumber)*, (AuthorizedPerson)*)*
(State, (StateNickname)*)*
```

Figure 6.14: Generated Forest of Scheme Trees for Figure 6.12.

a sequence or choice structure the starting object set for a scheme tree. Figure 6.14 shows the forest of scheme trees generated for Figure 6.12.

After generating the forest of scheme trees, we construct the XML-Schema instance. Figure 6.15, as an example, shows the translation of the scheme tree (*Student, StudentID, StudentName, FirstName, LastName, ContactInfo,* (*MiddleName*)*, (*Address*)*)*. The algorithm is basically the same as the algorithm in Section 6.2. The insertion of sequence and choice structures, however, makes the translation different in the following ways.

- Since sequence and choice structures cannot nest attributes, the system translates any lexical object set in a C-XML model instance that is a child of a sequence or choice structure and that would otherwise be translated to an attribute as an XML-Schema element rather than as an attribute. For example, in the C-XML model instance in Figure 6.12, the lexical object sets *StudentName*, *FirstName*, and *LastName* are translated to elements instead of attributes in Lines 8, 10, and 22 in Figure 6.15.

- Because we will be introducing sequence and choice structures, we replace the *all* structure that is nested within the container element for an individual tuple in a node by a sequence structure. For example, in Figure 6.14 the object set *Student* is the container element for an individual tuple for the node *Student-StudentID-StudentName-FirstName-LastName-ContactInfo* in the scheme tree (*Student, StudentID, StudentName, FirstName, LastName, ContactInfo,* (*MiddleName*)*, (*Address*)*)*, and it is a parent of a choice and sequence structure. In this case the *all* structure that we used to nest the content for an individual tuple in a node in the subsection *Nodes* in Section 6.2 cannot work because the *all* structure can only nest elements and cannot nest sequence or choice structures. Thus, we replace the *all* structure with a sequence structure. In Figure 6.15, for example, we declare a sequence in Lines 6 through 37. Notice that the elements *StudentName*, *FirstName*, and *LastName* that were attributes in Section 6.2 are now nested under this sequence.

```
<xs:element name="Students">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="Student" maxOccurs="unbounded">
            <xs:complexType>
               <xs:sequence>
                  <xs:choice>
                     <xs:element name="StudentName" type="xs:string"/>
                     <xs:sequence>
                        <xs:element name="FirstName" type="xs:string"/>
                        <xs:element name="MiddleNames">
                           <xs:complexType>
                              <xs:sequence>
                                 <xs:element name="MiddleName" minOccurs="0" maxOccurs="2">
                                    <xs:complexType>
                                       <xs:atribute name="MiddleName" type="xs:string" use="required"/>
                                    </xs:complexType>
                                 </xs:element>
                              </xs:sequence>
                           </xs:complexType>
                        </xs:element>
                        <xs:element name="LastName" type="xs:string"/>
                     </xs:sequence>
                  </xs:choice>
                  <xs:sequence>
                     <xs:element name="Address" maxOccurs="2">
                        <xs:complexType>
                           <xs:attribute ref="AddressOID"/>
                        </xs:complexType>
                     </xs:element>
                     <xs:element name="ContactInfo">
                        <xs:complexType>
                           <xs:attribute ref="ContactInfoOID"/>
                        </xs:complexType>
                     </xs:element>
                  </xs:sequence>
               </xs:sequence>
            </xs:complexType>
            <xs:attribute name="StudentOID" type="xs:string" use="required"/>
            <xs:attribute name="StudentID" type="xs:string" use="required"/>
         </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Figure 6.15: XML-Schema Instance for *Student*.

- Since we are guaranteed that children of sequence or choice structures are nested with their parent in the same scheme tree, we can properly reinsert sequence and choice structures in an XML-Schema instance for the elements that are children of sequence or choice structures in the original C-XML model instance. For example, in Figure 6.15, elements *FirstName* (Line 10), *MiddleNames* (Line 11), and *LastName* (Line 22) are nested within a sequence structure (Lines 9–23). The sequence structure itself and the element *StudentName* (Line 8) are nested within the choice structure (Lines 7–24). *Address* (Lines 26–30) and *ContactInfo* (Lines 31–35) are nested within another sequence structure (Lines 25–36). (Since both *Address* and *ContactInfo* appear in multiple scheme trees, we define them by referring to global elements.)

Unlike XML Schema, C-XML allows for sequence and choice structures to have a minimum and maximum number of occurrences within the relationship set between the parent and the sequence or choice. It also allows for a sequence's or a choice's child to have a minimum and maximum number of occurrences within the relationship set between the sequence or choice and the child. For both of these cardinality constraints, the default minimum constraint is one and the maximum is unbounded. XML-Schema provides no way, in general, to capture these non-default cardinality constraints. Thus, when these cardinalities differ from the default, we capture them in special C-XML comments. For example, to declare that the participation constraint for the child *State* is optional, we can write the comment

<!--*C-XML*: *forall* x (*State*(x) =>
exists [0:$*$] y (*State*(x) *has Order*(2) *in Sequence-k*(y))) -->.

This comment declares that the minimum cardinality for the second child in the kth sequence, which is *State*, is zero. (To keep track of which of the potentially many sequences the constraint references, we give each sequence a number, k in this example).

Although there is no way, in general, to capture these cardinality constraints in XML Schema there is an alternative in one special case when the cardinality constraint for the child is optional. In our example the cardinality constraint for the child *State* of the sequence structure that is connected to *Address* is optional. In this case, we can replace the optional constraint with a mandatory constraint and make the child *State* a specialization of a new object set which we also call *State*. The generated forest of scheme trees will be the same as in Figure 6.14. The *State* in the scheme tree (*Address, City, State, ZipCode,* (*AuthorizedPerson*)*)*, however, is the specialized *State*, and the *State* in the scheme tree (*State,* (*StateNickname*)*)* is the generalized

State. To allow for the subset constraint to be represented by *keyref* as we do for any generalization/specialization, we must (and always can) make the generalization a key. In our example, *State* is a key in the scheme tree (*State,* (*StateNickname*)*)*, and we add a *keyref* from *State* in the address to *State* for state nickname.

6.5 Conclusion

In this chapter we discussed in detail the translation of C-XML to XML Schema. Our prototype implementation, which corresponds to our discussion in this chapter, can automatically convert any C-XML conceptual-model instance to an XML-Schema instance. Our goals for the translation were (1) to make the translation information and constraint preserving and (2) to make the translation reasonably pleasing for the end user.

Our translation preserves information. Our translation provides a place for each object instance of any C-XML object set. The HST algorithm guarantees that each object set finds its way into a scheme tree, and our translation from scheme tree to XML Schema guarantees that any object instance will have a place for its representation in an XML-Schema document. Similarly, our translation provides a place for each relationship instance of any C-XML relationship set. The HST algorithm guarantees that each relationship set appropriately finds its way into a scheme tree, either within a node or as a parent-child link between nodes. Furthermore, our translation algorithm guarantees that any relationship-set instance will have an appropriate place for its representation in an XML-Schema document—appropriate in the sense that a relationship will be represented either as a direct connection between an element and the element's attribute or a direct connection between an element and one or more of its nested elements.

Our translation preserves constraints as long as we count the comments that we insert to capture the C-XML constraints that XML Schema is not able to represent. We preserve superset/subset constraints of a generalization/specialization hierarchy by using *keyref* declarations, which guarantee that each specialization instance is also an instance of its generalization(s). We can, however, only capture constraints on generalization/specialization hierarchies (union, mutual-exclusion, partition, and intersection constraints) in special C-XML comments. We preserve sequence or choice structures that appear in C-XML model instances, by properly inserting them in XML-Schema instances over their child elements. We preserve uniqueness constraints by deriving them from given cardinality constraints and expressing them in the generated XML-Schema instance using *key* declarations. From the data frame associated

with each object set in C-XML, we are able to obtain, either directly or by default, type information for instances as well as other information needed to complete the declaration of XML-Schema attributes or elements. Finally, we preserve participation constraints for any object and relationship sets and for sequence and choice structures that appear in a C-XML model instance by expressing them directly in an XML-Schema instance via *minOccurs* and *maxOccurs* or *use*, or by expressing them in special C-XML comments.

Although our translation preserves information and constraints, it does introduces two artifacts (unnecessary structures and constraints needed to satisfy the syntax requirements of XML Schema). (1) XML Schema requires a root element. (2) The translation sometimes imposes a sequence among data items when the conceptual-model instance has no requirement for a sequence. This is because *all* in XML Schema is limited to only nest elements that appear at most one time, but the translation sometimes requires the nesting of elements that appear more than once.

Although whether something is pleasing is subjective and varies from one user to another, we believe that our translation is likely to be pleasing to the end user for the following reasons. (1) Our translation generates an XML-Schema instance that allows no redundancy and is reasonably compact. (2) Our translation allows the user to choose the starting nodes to form a scheme-tree forest so that users can guarantee that important application concepts can be at or near the top of nested XML structures. (3) The structure of the generated XML-Schema instance makes XML documents that validate with respect to the generated XML-Schema easy to read. We obtain this easy-to-read structure by providing and appropriately naming a container element for an individual tuple and for a set of tuples for each node in each scheme tree. (4) We generate an *OID* attribute to explicitly represent objects within the nonlexical object sets. Thus, since objects in lexical object sets also have an explicit representation, every object instance in any XML document that complies with a generated XML-Schema instance will have an explicit representation in the document.

Chapter 7

Conclusions and Future Work

In this research we have introduced a conceptual model for XML, called C-XML. C-XML is an answer for the new need for system analysts who wish to store their data using XML—the need for a simple conceptual model that works well for XML-based development.

This work has produced several contributions (Section 7.1) and has led to several observations and insights (Section 7.2). The work accomplished also establishes the basis for interesting future work (Section 7.3).

7.1 Contributions

In this research we have extended the conceptual modeling language OSM, resulting in the development and implementation of C-XML. We argued that C-XML is a good candidate for a conceptual modeling language for XML because. It satisfies the requirements for conceptual modeling for XML presented by others who have studied the problem [28, 35, 42]. 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.

We have implemented an automatic conversion from XML Schema to C-XML that preserves information and constraints. Thus, we can view an XML Schema instance graphically at a higher level of abstraction. We have also implemented an automatic conversion from C-XML to XML Schema. Our translations preserve information and constraints as long as we count the C-XML comments that we insert in an XML-Schema instance to capture the constraints in C-XML that are not representable in XML Schema. We also explained how to apply the hypergraph-to-scheme-tree translation algorithm to generate XML-Schema instances whose complying XML documents do not have redundancy and are reasonably compact.

Based on our implemented translations, we have explored the equivalence of C-XML and XML Schema. We discovered that both our translation from XML Schema to C-XML and from C-XML to XML Schema are injective. Therefore, reverse translations exist. We implemented these translations as well. We also discovered, however, that our basic translations to and from C-XML and XML Schema are not inverses of one another. Not only does an application of one after the other not yield the original model instance, but continued successive application of one after the other actually diverges, yielding ever more object sets in C-XML and ever more complex-type containers in XML Schema. We, of course, can stop the divergence at any point in time (usually after the first conversion) by using the reverse translations based on our injective translations instead.

7.2 Observations and Insights

As a result of studying and implementing translations to and from C-XML and XML-Schema, we offer the following insights and recommendations.

- C-XML is more expressive in several ways than is XML Schema for the following reasons.
  - XML Schema sometimes artificially imposes a sequence among elements in an XML-Schema instance when there is no need for a sequence. This is because the *all* structure in XML Schema is limited to only nest elements that appear at most one time.
  - XML Schema does not support union, mutual-exclusion, or partition constraints for generalization/specialization hierarchies. It also does not support an intersection constraint for multiple generalizations.
  - Because XML Schema has a hierarchial structure, it only allows the specification of participation constraints for parent elements. It does not provide for participation constraints for child elements except for the default—one-many.

  We recommend extending XML Schema in two ways.
  - Extend the *all* structure so that it can nest elements that appear more than one time and so that it can nest sequence and choice structures.

  - Extend XML Schema so that it provides for a declaration of union, mutual-exclusion, and partition constraints for generalization/specialization hierarchies, and also provides for a declaration of intersection constraints for multiple generalizations.

- XML Schema is more expressive in several ways than is the hypergraph-based conceptual model from which we built C-XML. Hypergraph-based OSM consists of lexical and non-lexical object sets; binary and n-ary (n>2) relationship sets; generalization/specialization with union, mutual-exclusion, partition, and intersection constraints, and several cardinality constraints—functional (including multidimensional functions whose domains consist of multiple object-sets), optional/mandatory participation, and min-max bounded participation constraints (for non-functional relationship sets). Hypergraph-based OSM, like other traditional conceptual models, does not support the following features that XML Schema supports: *sequence*, *choice*, *mixed-content*, and *any* and *anyAttribute* structures.

  We recommend enriching conceptual model languages by giving them the ability to: (1) order lists of concepts, (2) choose alternative concepts from among several, (3) specify mixed content, and (4) use content from another data model.

7.3 Future Work

It would be interesting to mathematically prove properties about the transformations included within the C-XML/XML-Schema mappings. Our implemented translations constitute constructive proofs of equivalence and non-equivalence under various assumptions, but these constructive proofs are complex. The mathematical proof proposed as future work would be based on mathematical model theory.

To make our prototype tools practical, we need to build interface tools to work synergistically with users. We imagine that these tools would have standard graphical interfaces and would provide drop-down menus and tools to facilitate user interactions.

The translation work accomplished in connection with this research establishes the basis for several fundamental activities in system analysis, design, development, and evolution. We thus could continue with this work in several directions.

- *XML Database Design and Development.* The conversion from C-XML to XML Schema constitutes a standard design and development paradigm for XML databases. We could integrate our work with the work of others to establish a complete workbench for XML database design and development.

- *Reverse Engineering.* The conversion from XML Schema to C-XML constitutes a standard kind of reverse engineering. We can reverse engineer XML-based systems to understand, maintain, and evolve them.

- *Integration.* The conversion between C-XML and XML Schema allows us to handle integration at the conceptual level, rather than at the XML-Schema level. Thus, to integrate two XML repositories each described by an XML schema, we first translate each XML schema to C-XML. We then integrate the two C-XML model instances to create an integrated C-XML model instance, and we then translate the integrated C-XML model instance back into an XML-Schema instance to create an integrated XML schema for the original XML schemas.

Bibliography

[1] R. Al-Kamha. *Translating XML Schema to Conceptual XML*. Technical Report, Computer Science Department, Brigham Young University, November 2006.

[2] R. Al-Kamha. *Translating Conceptual XML to XML Schema*. Technical Report, Computer Science Department, Brigham Young University, May 2007.

[3] F. Baader and W. Nutt. Basic description logics. In F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors, *The Description Logic Handbook*, chapter 2, pages 43–95. Cambridge University Press, Cambridge, UK, 2003.

[4] C. Batini, S. Ceri, and S.B. Navathe. *Conceptual database design: an Entity-relationship approach*. Benjamin-Cummings, Redwood City, California, USA, 1992.

[5] L. Bird, A. Goodchild, and T. Halpin. Object role modelling and XML-schema. In *Proceedings of the Ninteenth International Conference on Conceptual Modeling (ER2000)*, pages 309–322, Salt Lake City, Utah, October 2000.

[6] J. Biskup and D.W. Embley. Extracting information from heterogeneous information sources using ontologically specified target views. *Information Systems*, 28(3):169–212, 2003.

[7] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, 1999.

[8] S.D. Camillo, C.A. Heuser, and R. dos Santos Mello. Querying heterogeneous XML sources through a conceptual schema. In *Proceedings of the 22nd International Conference on Conceptual Modeling (ER2003), Lecture Notes in Computer Science 2813*, pages 186–199, Chicago, Illinois, October 2003.

[9] M. Carey. Enterprise information integration—XML to the rescue! In *Proceedings of the 22th International Conference on Conceptual Modeling (ER2003),*

Lecture Notes in Computer Science 2813, page 14, Chicago, Illinois, October 2003.

[10] D. Carlson. *Modeling XML Applications with UML: Practical e-Business Applications*. Addison-Wesley, Menlo Park, California, 2001.

[11] P.P. Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[12] Y.B. Chen, T.W. Ling, and M.L. Lee. Designing valid XML views. In *Proceedings of the 21st International Conference on Conceptual Modeling (ER'02)*, pages 463–477, Tampere, Finland, October 2002.

[13] Y.B. Chen, T.W. Ling, and M.L. Lee. Automatic generation of XQuery view definitions from ORA-SS views. In *Proceedings of the 22nd International Conference on Conceptual Modeling (ER2003), Lecture Notes in Computer Science 2813*, pages 158–171, Chicago, Illinois, October 2003.

[14] R. Conrad, D. Scheffner, and J.C. Freytag. XML conceptual modeling using UML. In *Proceedings of the Ninteenth International Conference on Conceptual Modeling (ER2000)*, LNCS 1920:558–571, 2000.

[15] E. M. Dashofy. Issues in generating data bindings for an XML schema-based language. In *Proceedings of the of the Workshop on XML Technologies and Software Engineering (XSE2001)*, Toronto, Ontario, Canada, May 2001.

[16] B. Daum. *Modeling Business Objects with XML Schema*. Morgan Kaufmann, San Francisco, California, 2003.

[17] L. Dodds. Reconstructing DTD best practice, June 2000. http://www.xml.com/pub/a/2000/06/xmleurope/schemas.html.

[18] D.W. Embley. *Object Database Development: Concepts and Principles*. Addison-Wesley, Reading, Massachusetts, 1998.

[19] 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.

[20] 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.

[21] D.W. Embley and W.Y. Mok. Developing XML documents with guaranteed 'good' properties. In *Proceedings of the 20th International Conference on Conceptual Modeling (ER2001)*, pages 426–441, Yokohama, Japan, November 2001.

[22] A Formal Semantics for UML, October 2006. http://www.cs.queensu.ca/ stl/internal/uml2/MoDELS2006/.

[23] R.C. Goldstein and V.C. Storey. Some findings on the intuitiveness of entity-relationship constructs. In *Proceedings of the Eighth International Conference on Entity-Relationship Approach (ER'89)*, pages 9–23, Toronto, Canada, October 1989. North-Holland.

[24] S.W. Liddle, D.W. Embley, and S.N. Woodfield. Cardinality constraints in semantic data models. *Data & Knowledge Engineering*, 11(3):235–270, 1993.

[25] S.W. Liddle, D.W. Embley, and S.N. Woodfield. An active, object-oriented, model-equivalent programming language. In M.P. Papazoglou, S. Spaccapietra, and Z. Tari, editors, *Advances in Object-Oriented Data Modeling*, pages 333–361. MIT Press, Cambridge, Massachusetts, 2000.

[26] M. Mani, D. Lee, and R.R. Muntz. Semantic data modeling using XML schemas. In *Proceedings of the 20th International Conference on Conceptual Modeling (ER2001)*, pages 149–163, Yokohama, Japan, November 2001.

[27] R. Miller, L. Haas, and M.A. Hernandez. Schema mapping as query discovery. In *Proceedings of the 26th International Conference on Very Large Databases (VLDB'00)*, pages 77–88, Cairo, Egypt, September 2000.

[28] M. Necasky. Conceptual modeling for XML: A survey. In *Proceedings of the DATESO 2006 Annual International Workshop on Databases, Texts, Specifications and Objects (DATESO 2006)*, pages 40–53, Desna, Czech Republic, April 2006.

[29] Visual Studio.NET, Microsoft. http://www.msdn.microsoft.com/vstudio.

[30] P. Pigozzo and E. Quintarelli. An algorithm for generating XML schemas from ER schemas. In *Proceedings of the Thirteenth Italian Symposium on Advanced Database Systems (SEBD2005)*, pages 192–199, Brixen-Bressanone, Italy, June 2005.

[31] E. Rahm and P.A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10:334–350, 2001.

[32] N. Routledge, L. Bird, and A. Goodchild. UML and XML schema. In *Proceedings of the Thirteenth Australasian Database Conference (ADC2002)*, pages 157–166, Melbourne, Australia, January 2002.

[33] A. Sengupta and S. Mohan. *Formal and Conceptual Models for XML Structures—The Past, Present, and Future*. Technical Report 137–1, Indiana University, Information Systems Department, Bloomington, Indiana, April 2003.

[34] A. Sengupta, S. Mohan, and R. Doshi. XER — extensible entity relationship modeling. In *Proceedings of XML 2003*, Philadelphia, Pennsylvania, December 2003.

[35] A. Sengupta and E. Wilde. *The Case for Conceptual Modeling for XML*. Technical Report No. 242, Computer Engineering and Networks Laboratory, ETH Zurich, February 2006.

[36] J. M. Smith and D. C. P. Smith. Database abstractions: Aggregation and generalization. *ACM Trans. Database Syst.*, 2(2):105–133, 1977.

[37] XMLSpy, Altova. http://www.xmlspy.com.

[38] Stylus Studio. http://www.stylusstudio.com/xml_schema_editor.html.

[39] I. Tatarinov and A. Halevy. Efficient query reformulation in peer data management systems. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, 2004. (to appear).

[40] T.J. Teorey, D. Yang, and J.P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys*, 18(2):197–222, June 1986.

[41] UML 2.0 superstructure specification, August 2005.

[42] E. Wilde. Towards conceptual modeling for XML. In *Proceedings of the Berliner XML Tage 2005 (BXML2005)*, pages 213–224, Berlin, Germany, September 2005.

[43] XQuery 1.0: An XML Query Language, November 2003. http://www.w3.org/TR/xquery/.

[44] XML Schema Part 0: Primer: W3C Recommendation, May 2001. http://www.w3.org/TR/xmlschema-0/.

[45] XML Schema - structures quick reference, 2003. http://www.xml.dvint.com/docs/SchemaStructuresQR-2.pdf.