Enterprise Modeling with Conceptual XML

D.W. Embley,* Department of Computer Science
S.W. Liddle,* School of Accountancy and Information Systems
R. Al-Kamha,* Department of Computer Science
Brigham Young University, Provo, Utah 84602, U.S.A.
embley@cs.byu.edu, liddle@byu.edu, ra72@email.byu.edu

Abstract

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. We present a vision for Conceptual XML (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.

1 Introduction

A challenge[1] for modern enterprise modeling is to produce a simple conceptual model that

- works well with XML and XML Schema;
- abstracts well for conceptual entities and relationships;
- scales to handle both large data sets and complex object interrelationships;
- allows for queries and defined views via XQuery; and
- 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

*Supported in part by the National Science Foundation under grant No. IIS-0083127.

[1]As a note of interest, we mention that Michael Carey issued this challenge directly to the conceptual modeling community in his ER2003 keynote address in Chicago.

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- 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" [LEW00] 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 answering Carey's XML/conceptual-modeling challenge, our contributions in this paper 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 describes C-XML. Section 3 shows that C-XML is "model-equivalent" with XML Schema by providing mappings in both directions—from C-XML to XML Schema in Section 3.1 and from XML Schema to C-XML in Section 3.2. Section 4 describes C-XML views. Section 4.1 defines and illustrates both high-level object-set views and high-level relationship-set views. Section 4.2 explains how XQuery immediately applies to C-XML as a query language and thus, also immediately provides a view-definition mechanism. Section 4.3 further extends the usage of XQuery to integration mappings and supports views over an enterprise-wide federated database system. Although we present a solution for the central issues embodied in the challenge, it is clear that there are many closely related issues we must resolve to fully realize the potential of the approach. These include, for example, a formal foundation, query optimization, the need for a simple programming model that facilitates very high level programming, and updates that involve workflow and custom API calls. We believe that the approach we propose opens the door to a resolution of these issues. In Section 5 we briefly explain how C-XML lends itself to a solution. We report the status of our implementation and make a few concluding remarks in Section 6.

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

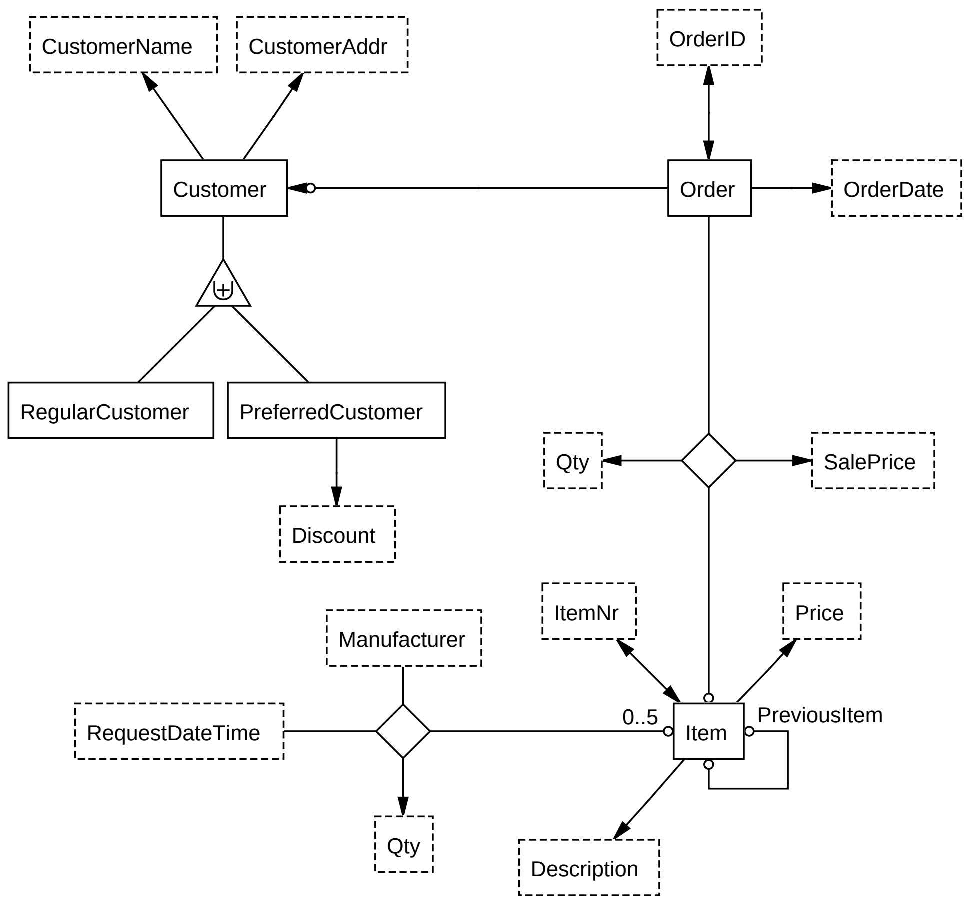


Figure 1: Customer/Order C-XML Model Instance.

augmentations consist of decorations that represent constraints. Figure 1 shows an example.

In the notation[2] boxes represent *object sets*—dashed if lexical (e.g. *CustomerName* in Figure 1) and not dashed if nonlexical because their objects are represented by object identifiers (e.g. *Order* in Figure 1). 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 (e.g. the relationship set among the object sets *Order*, *Item*, *Qty*, and *SalePrice* in Figure 1). 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 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*

[2]Although based on [EKW92], the particular notation we use to represent C-XML is not significant. The hypergraph foundation, however, is significant because it is more directly amenable to the requirements of XML and XML Schema. Thus, this choice simplifies translations, abstraction definitions, and view generation. We can translate standard ER model instances to this hypergraph notation by letting hypergraph nodes represent both entities and attributes, by letting hypergraph edges represent both relationships and entity/attribute connections, and by adding appropriate constraints to capture the restrictions imposed by an ER diagram.

constraints. Thus, Figure 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. The *0..5* participation constraint in Figure 1 specifies that there can be at most five delivery requests for items from manufacturers. Triangles denote *generalization/specialization hierarchies* (ISA hierarchies, subset constraints, or inclusion dependencies), so that in Figure 1 *RegularCustomer* and *PreferredCustomer* are subsets of object set *Customer*. We can constrain ISA hierarchies by partition ($\uplus$), union ($\cup$), or mutual exclusion ($+$) among specializations; thus in Figure 1 all customers are either regular or preferred, and no customer is both. 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. In Figure 1 *PreviousItem* is a role on the recursive relationship set connecting current items to previous items they replaced in the inventory.

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, [BGH00, CSF00, MLM01, dSMH01, EM01, EWH$^+$02, CLL02].) 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 3.1 we present our C-XML-to-XML-Schema translation, and in Section 3.2 we present a XML-Schema-to-C-XML translation. In Section 3.3 we formalize the notions of information and constraint preservation and show that the translations we propose preserve information and constraints.

3.1 Translation from C-XML to XML Schema

In this section we 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 1 translated to the corresponding XML Schema of Figure 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 [EM01], 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 1, the algorithms in [EM01] will generate the following two nested scheme trees.

$$\begin{array}{l}(Customer, CustomerName, CustomerAddr, Discount\\ \quad (Order, OrderID, OrderDate,\\ \qquad (Item, SalePrice, Qty)*)*)*\end{array}$$

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified" attributeFormDefault="unqualified">
<xs:element name="Document">
    <xs:complexType>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="Customer"/>
            <xs:element name="Item">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="ItemMR" minOccurs="0" maxOccurs="5">
                            <xs:complexType>
                                <xs:attribute name="Manufacturer" type="xs:string" use="required"/>
                                <xs:attribute name="RequestDateTime" type="xs:date" use="required"/>
                                <xs:attribute name="Qty" type="xs:positiveInteger" use="required"/>
                            </xs:complexType>
                        </xs:element>
                        <xs:element name="PreviousItem" minOccurs="0" maxOccurs="unbounded">
                            <xs:complexType>
                                <xs:attribute name="ItemNr" type="xs:positiveInteger" use="required"/>
                            </xs:complexType>
                            <xs:keyref name="r1" refer="ItemKey">
                                <xs:selector xpath="."/>
                                <xs:field xpath="@ItemNr"/>
                            </xs:keyref>
                        </xs:element>
                    </xs:sequence>
                    <xs:attribute name="ItemNr" type="xs:positiveInteger" use="required"/>
                    <xs:attribute name="Description" type="xs:string" use="required"/>
                    <xs:attribute name="Price" type="xs:decimal" use="required"/>
                </xs:complexType>
            </xs:element>
        </xs:choice>
    </xs:complexType>
    <xs:key name="OrderKey">
        <xs:selector xpath=".//Order"/>
        <xs:field xpath="@OrderID"/>
    </xs:key>
    <xs:key name="ItemKey">
        <xs:selector xpath=".//Item"/>
        <xs:field xpath="@ItemNr"/>
    </xs:key>
</xs:element>
<xs:element name="Customer" abstract="true"/>
<xs:element name="PreferredCustomer" substitutionGroup="Customer">
    <xs:complexType>
        <xs:group ref="CustomerDetails"/>
        <xs:attribute name="Discount" type="xs:string" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="RegularCustomer" substitutionGroup="Customer">
    <xs:complexType>
        <xs:group ref="CustomerDetails"/>
    </xs:complexType>
</xs:element>
<xs:group name="CustomerDetails">
    <xs:sequence>
        <xs:element name="CustomerName" type="xs:string"/>
        <xs:element name="CustomerAddr" type="xs:string"/>
        <xs:element name="Order" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="OrderItem" minOccurs="0" maxOccurs="unbounded">
                        <xs:complexType>
                            <xs:attribute name="Qty" type="xs:positiveInteger" use="required"/>
                            <xs:attribute name="SalePrice" type="xs:decimal" use="required"/>
                            <xs:attribute name="ItemNr" type="xs:positiveInteger" use="required"/>
                        </xs:complexType>
                        <xs:keyref name="r3" refer="ItemKey">
                            <xs:selector xpath="."/>
                            <xs:field xpath="@ItemNr"/>
                        </xs:keyref>
                    </xs:element>
                </xs:sequence>
                <xs:attribute name="OrderID" type="xs:positiveInteger" use="required"/>
                <xs:attribute name="OrderDate" type="xs:date" use="required"/>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:group>
</xs:schema>
```

Figure 2: XML Schema for the C-XML Model Instance in Figure 1

$$(Item, ItemNr, Description, Price,\\ \quad (PreviousItem)*, (Manufacturer, RequestDateTime, Qty)*)*$$

Observe that the XML Schema in Figure 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, as we explain in our continuing discussion, 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.

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 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 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 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. 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 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 non-functionally 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 names 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 Lines 35–38 and *Item*/*ItemNr* in Lines 39–42. 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 53). In our example, we generate the group *CustomerDetails*[3] 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.

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 now describe the structural concepts of XML Schema.

XML Schema [XML01] has *complex types* and *simple types*. Simple types are fundamentally strings: built-in types (e.g. *date*, *integer*), restrictions of built-in types (e.g. 4-digit integers), list types (e.g. *IDREFS*), and union types which combine multiple simple types into a single type (e.g. date or integer). Complex types define elements that can have attributes and one of three kinds of content: *simple* (content is defined by a simple type), *mixed* (content may be text interspersed with elements), or *complex* (content may include other elements). Further, complex content may be defined as *group*, *choice*, or *sequence* structures. Complex-content group structures are used to factor out a portion of a schema to be reused in multiple regions of a schema (for example, *CustomerDetails* in Figure 2 is defined in Line 56 and referenced at lines 47 and 53). Choice structures allow alternation (choose exactly one alternative from a given set). Sequence structures specify a list of elements, possibly restricted by minimum/maximum occurrence constraints.

There are more structural details we need to consider. For example, attributes can be declared in *attribute groups* (like complex content groups, this permits reuse of attribute declarations). Attributes can only occur once, so there are no explicit maximum participation constraints. By default, attributes are optional, but they can be marked as required. Attributes are unordered within an element. Complex content groups can come in three flavors: *all*, *choice*, and *sequence*. And elements may be declared substitutable for each other, creating equivalence classes among the elements. Further, elements may be declared abstract, indicating that they cannot be directly instantiated, but instead a substitutable element must be used.

Understanding these structures, 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

[3]In general, we name such a group by *Details* concatenated with the abstract element.

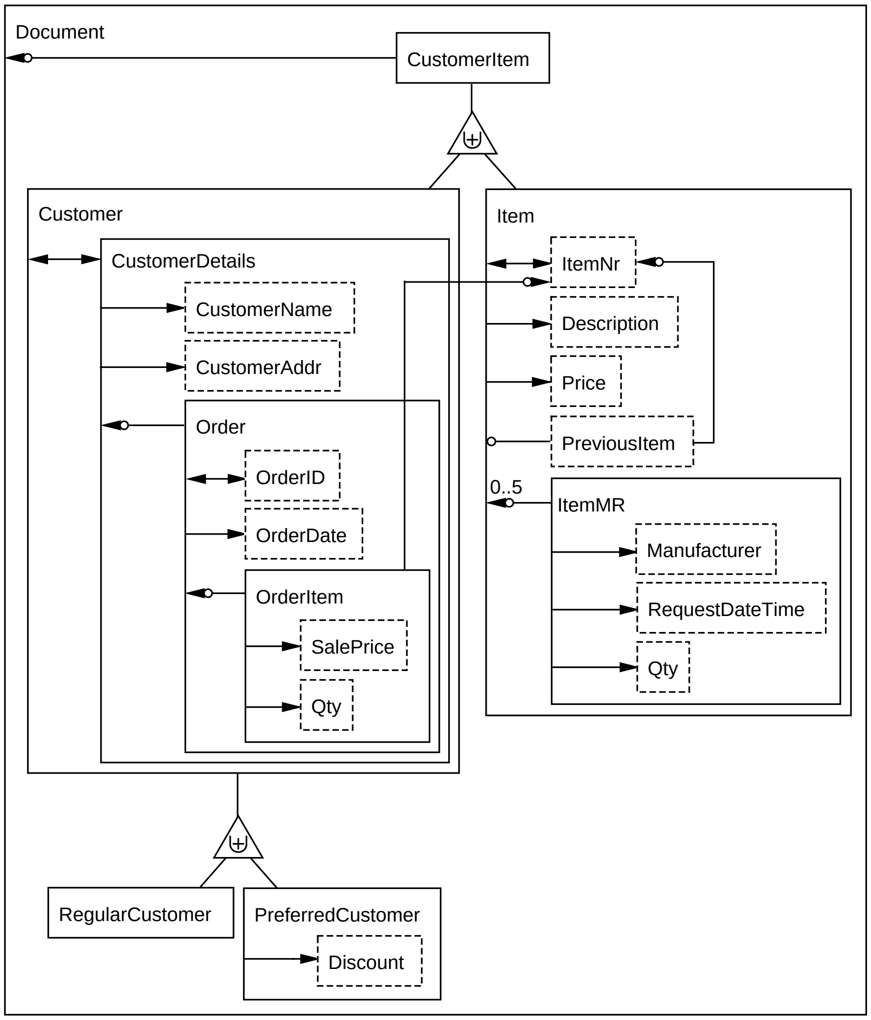


Figure 3: C-XML Model Instance Translated from XML Schema Instance of Figure 2.

according to the nesting structure of S. Figure 3 shows the result of applying our conversion process to the XML Schema instance of Figure 2.[4] 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 we can use directly.

[4]The particular graphical layout is human-created. Our algorithm only supplies the C-XML structure. We have used automated layout algorithms like AGLO, but humans generally need to adjust the output anyway.

For example, according to the constraints on Line 60 in Figure 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 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 1 and 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.

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 [EKW92]. 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 $I_{S_{C-XML}^{PC}}$ in predicate calculus.
Proof(*Sketch*): 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 $I_{S_{C-XML}^{PC}}$ in predicate calculus. See [EKW92] 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 $I_{S_{XMLSchema}^{PC}}$ in predicate calculus.
Proof(*Sketch*): We construct $t_{XMLSchema}$ as follows. Similar to [EKW92], 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 $I_{S_{XMLSchema}^{PC}}$ in predicate calculus. □

Theorem 1 Let T be the translation described in Section 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 (*Sketch*): 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 $I_{S_{C-XML}^{PC}}$ as a valid interpretation for $I_{S_{C-XML}}$ in predicate calculus; similarly by Lemma 2, we can obtain $I_{S_{XMLSchema}^{PC}}$ 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 $I_{S_{C-XML}^{PC}}$ from $I_{S_{XMLSchema}^{PC}}$. 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 [EKW92] for a list of cases. □

Theorem 2 Let T be the translation described in Section 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 (*Sketch*): 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. □

4 C-XML Views

This section describes three types of views—simple views that help us scale up to large and complex XML schemas (Section 4.1), query generated views over a single XML schema (Section 4.2), and query generated views over heterogeneous XML schemas (Section 4.3).

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 [EKW92]. Figure 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 (e.g. in Figure 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 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 4 shows an example of a high-level abstraction of Figure 1. In Figure 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

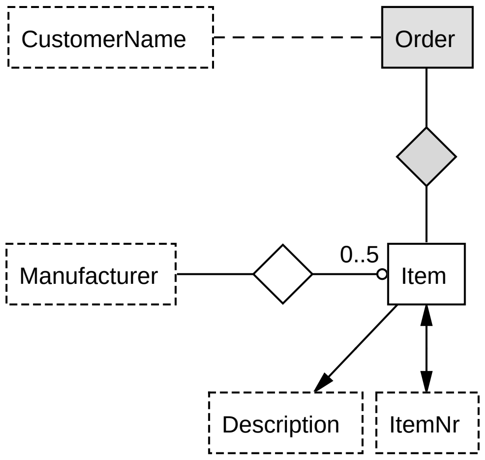


Figure 4: High-Level View of Customer/Order C-XML Model Instance.

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*.

Theorem 3 Simple, high-level views constructed by properly discarding C-XML components are valid C-XML model instances.
Proof: Straightforward. (See [BE03] for explanatory details.) □

Corollary 1 Any simple, high-level view can be represented by an XML Schema.
Proof: By Theorem 3 any simple, high-level view V is a C-XML model instance C. Thus, by Theorem 1, we can represent C as an XML Schema. □

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 [CHdSM03, CLL03], 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 [CHdSM03]. Thus, it is highly desirable to be able to construct XQuery views by generating them from a high-level conceptual model-based description. [CLL03] describes an algorithm for generating XQuery views from ORA-SS descriptions. [CHdSM03] also describes how to specify XQuery views by writing *conceptual XPath* expressions over a conceptual schema and then automatically generating the corresponding XQuery 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 [LEW00], but in keeping with the spirit of C-XML we define an XQuery-like textual language called C-XQuery.

Figure 5 shows a high-level view written in C-XQuery over the model instance of Figure 1.[5] We introduce a view definition with the phrase *define view*, and specify the contents of the view with

[5]All the XQuery samples in this paper were tested using BEA's excellent LiquidData product. In the case of C-XQuery, we first translated to normal XQuery and then tested.

```
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 5: C-XQuery View of Customers Nested within Items Ordered.

FLWOR (for, let, where, order by, return) expressions [XML03]. 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 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 [LEW00]. 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 5, for example, the query in Figure 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 the view according to the view definition and minimize the resulting XQuery as Figure 7 shows. See [TH04] for a discussion of the principles underlying this process.

The view in Figure 6 illustrates the use of views within 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.

```
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 6: C-XQuery over the View of Customers Nested within Items Ordered.

```
for $item in document(URL)/Document/Item,
     $c in document(URL)/Document/RegularCustomer
let $daysAgo90 := xf:add-days(xfext:date-from-dateTime(xf:current-dateTime()), -90)
where $item/@Description = "Nitrogen Fertilizer" and
     some $o in $c/Order satisfies
     ( $o/@OrderDate > $daysAgo90 and some $oi in $o/OrderItem satisfies
        ($oi/@ItemNr = $item/@ItemNr) ) and
     sum(for $o in $c/Order, $oi in $o/OrderItem
          where $o/@OrderDate > $daysAgo90 and $oi/@ItemNr = $item/@ItemNr
          return $oi/@Qty * $oi/@SalePrice) > 300
return
<PotentialThreatCustomer>
     { $c/CustomerName }
     <Total>{ sum( for $o in $c/Order, $oi in $o/OrderItem
                    where $o/@OrderDate > $daysAgo90
                         and $oi/@ItemNr = $item/@ItemNr
                    return $oi/@Qty * $oi/@SalePrice) }</Total>
</PotentialThreatCustomer>
```

Figure 7: XQuery Corresponding to Figure 6.

```
for $item in document(URL)/Document/Item
return
<Item>
    {$item/@ItemNr, $item/@Description}
    {
        for $customer in document(URL)/Document/Customer
        where some $o in $customer/Order satisfies
            ( some $oi in $o/OrderItem satisfies ($oi/@ItemNr = $item/@ItemNr) )
        return
        <Customer>
            {$customer/CustomerName, $customer/CustomerAddr}
            {
                for $order in $customer/Order,
                    $orderItem in $order/OrderItem
                where $item/@ItemNr = $orderItem/@ItemNr
                return
                <Order>
                    {$order/@OrderDate, $orderItem/@Qty, $orderItem/@SalePrice}
                </Order>
            }
        </Customer>
    }
</Item>
```

Figure 8: XQuery Generated for C-XML Query in Figure 5 over XML Schema in Figure 2.

Figure 8 shows the generated XQuery corresponding to the C-XQuery view of Figure 5 (with respect to the XML Schema instance of Figure 2).

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 (*Sketch*): By Theorem 1 we can translate C to S_C. Using the principles highlighted in this section we can translate Q to Q_C. Following the correspondence of C to S_C, we rewrite each conceptual path of Q as an XQuery path expression in Q_C, often with additional constraints in *where* clauses. We also designate fields as elements or attributes in Q_C where necessary. Figure 8 gives the XQuery translation of the C-XQuery view in Figure 5. □

Observe that by the definition of XQuery [XML03], 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.

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 9 shows the C-XML of the acquired company's 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 in Figure 9 to the parent company's catalog in Figure 1. Figure 10 shows the mapping we need. In order to integrate the source (Figure 9) with the target (Figure 1), the mapping needs to generate the same names found in the target. In this example, *CatalogItem*, *CatalogNr*, and *ShortName* correspond respectively to *Item*, *ItemNr*, and *Description*. We must compute *Price* in the target from the

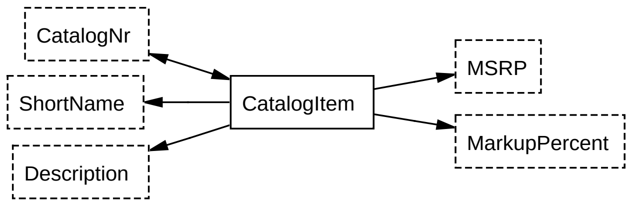


Figure 9: C-XML Model Instance for the Catalog of an Acquired Company.

```
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>
}
```

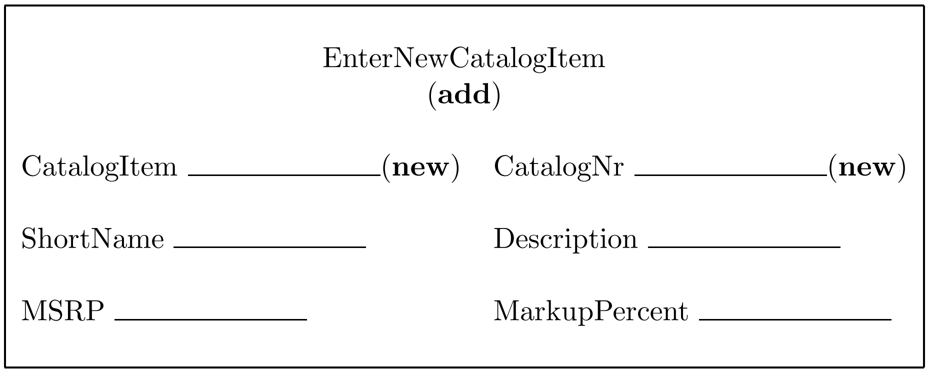
Figure 10: C-XQuery Mapping over the C-XML Model Instance in Figure 9

MSRP and *MarkupPercent* values in the source, as shown in Figure 10. We assume the predicate *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 1, so integration is now immediate.

The mapping in Figure 10 creates a C-XQuery view over the acquired company's catalog in Figure 1. 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 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 6. With the acquired company's catalog integrated, when the query in Figure 6 iterates over customer orders, it ultimately iterates over data instances for both *Item* in Figure 1 and *CatalogItem* in Figure 10. (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 1 as the source and Figure 9 as the target. Such bidirectional integration is useful in circumstances where we need to shift between perspectives often, 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 [RB01, BE03] to help us with the mapping. As illustrated in Figure 10, 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. [MHH00]), and we can compose these views to provide larger



EnterNewCatalogItem
(**add**)

CatalogItem __________(**new**)	CatalogNr __________(**new**)
ShortName __________	Description __________
MSRP __________	MarkupPercent __________

Figure 11: A Form Program Specification.

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.

Theorem 5 A C-XQuery view Q over a C-XML model instance C of an external, federated XML Schema can be translated to an XQuery query Q_C over an XML Schema instance S_C.
Proof: Similar to Theorem 4. □

5 C-XML Issues—Future Work

In this section we briefly discuss some of the issues, in addition to the ones we have already discussed, that need to be resolved to fully realize the potential impact of the C-XML approach to enterprise modeling. We focus on providing a simple high-level programming model based on C-XML. Done well, this can resolve a number of issues, including (1) the need for very high level declarative programming, (2) the need for expressing complex updates through workflow specifications, and (3) the need for API layers over both conceptual data declarations and conceptual workflow specifications.

Given a conceptual model like C-XML, one way to provide very high level declarative languages that is likely to work well in large enterprises is to use form specifications [Emb89]. When a customer orders items, for example, the customer usually fills out a form; a form specification language would allow us to directly use a new-order form as a program specification from which code can be generated. Figure 11 gives an example of a program specification for the view in Figure 9 that can be used to add new catalog items. The programmer-added annotations "(**add**)" and "(**new**)" direct the processing so that it adds a new item and catalog number and associates their values with other appropriate (possibly new or possibly already existing) data. We claim, and intend to show in future work, that we can convert the program specification in Figure 11 into code in much the same way we convert C-XQuery to code (as shown earlier in this paper).

Specifying a simple database update with a single form might be sufficient, but this simplicity will not scale up. In general, we need a way to express complex updates through workflow specifications. In our conceptual modeling work, we have defined both an object-behavior model and an object-interaction model to go along with our object-relationship model [EKW92]. Basically, an object in an object set may be either active or passive. Active objects behave according to their state transition diagrams. Groups of objects interact to work together. A group of interacting

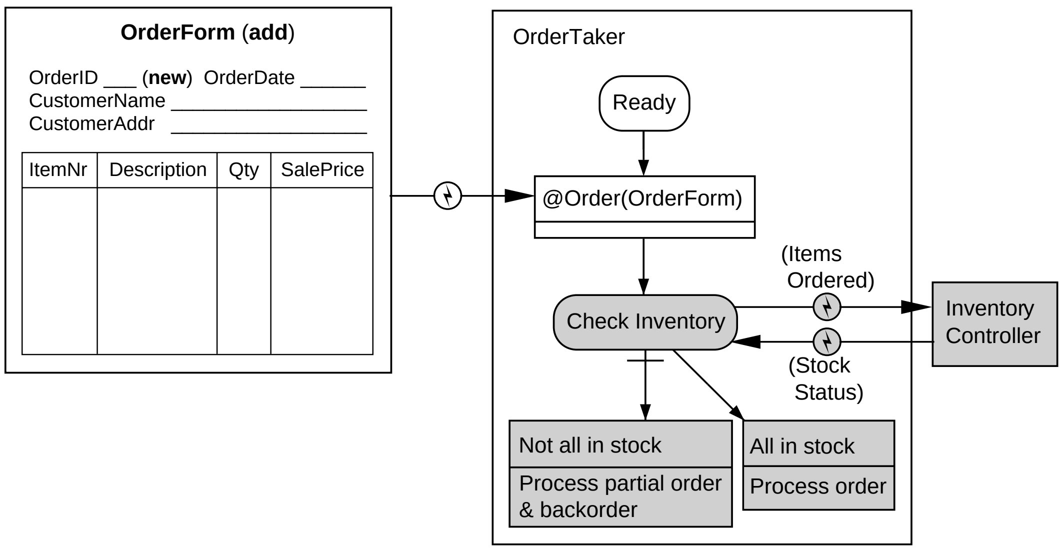


Figure 12: C-XML Workflow Diagram for Order Process.

objects, including objects outside the implemented software system, constitutes a workflow. Workflows and form specifications meld well together because we can use form specifications at interface points and input/output points along a workflow and use a workflow specification for routing and control. Similar to high-level abstractions in C-XML, we have also defined high-level, first-class abstractions for states, transitions, and interactions so that workflow developers can work at various levels of abstraction (see Chapters 4 and 5 of [EKW92]). Figure 12 shows an example. We represent interactions with arrows marked by a circled lightning bolt, states with rectangles having rounded corners, and transitions as divided rectangles where the upper region describes the trigger, and the lower region gives the action of the transition. In Figure 12, an *OrderForm* object passes itself to an *OrderTaker*, triggering a high-level activity to check inventory status for the ordered items (high-level states, transitions, and interactions are shaded, indicating a view that hides lower-level details). When an *InventoryController* receives a list of items, it indicates the current stock status (details hidden here). If all items are in stock, the order taker processes the order normally, but if some item is out of stock, the order taker processes the exception (indicated by a bar across the arrow) by fulfilling a partial order and backordering the out-of-stock items. Again, we claim, and intend to show in future work, that we can convert the program specification in Figure 12 into code (we have already implemented a similar model execution engine [LEW00]).

There is much more to say, and what we have said only hints at the possibilities, but it should be clear that high-level conceptual modeling in something like C-XML and workflow-augmented C-XML opens the door to the possibility of (1) enabling very high level declarative programming, (2) expressing updates through workflow specifications, and (3) providing the kind of API layers we need to appropriately abstract away the myriad of detail in which we often become entangled when working with large enterprise systems. Thus, we can take one step closer to removing the imposition of arcane notation on decision makers and requiring programmers to grovel in low-level details.

6 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 (Theorem 5).

Implementing C-XML is a huge undertaking. Even implementing a proof-of-concept prototype is a daunting task for small, university research group. However, we have a long history of developing prototype-class tools supporting our models, and we have a foundation on which to build. Implemented tools relevant to C-XML include graphical diagram editors, model checkers, textual model compilers, a model execution engine, and several data integration tools. Our current implementation uses the Java framework for portability. We are actively continuing development of an Integrated Development Environment (IDE) for modeling-related activities. Our current strategy is to plug new tools into this IDE rather than developing 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.

References

[BE03] J. Biskup and D.W. Embley. Extracting information from heterogeneous information sources using ontologically specified target views. *Information Systems*, 28(3):169–212, 2003.

[BGH00] 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.

[CHdSM03] 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.

[CLL02] 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.

[CLL03] 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.

[CSF00] R. Conrad, Deiter Scheffner, and J.C. Freytag. XML conceptual modeling using UML. In *Proceedings of the Ninteenth International Conference on Conceptual Modeling (ER2000)*, Salt Lake City, Utah, October 2000. 558–571.

[dSMH01] R. dos Santos Mello and C.A. Heuser. A rule-based conversion of a DTD to a conceptual schema. In *Proceedings of the 20th International Conference on Conceptual Modeling (ER2001)*, pages 134–148, Yokohama, Japan, November 2001.

[EKW92] 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.

[EM01] 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.

[Emb89] D.W. Embley. NFQL: The natural forms query language. *ACM Transactions on Database Systems*, 14(2):168–211, June 1989.

[EWH$^+$02] R. Elmazri, Y.-C. Wu, B. Hojabri, C. Li, and J. Fu. Conceptual modeling for customized xml schemas. In *Proceedings of the 21st International Conference on Conceptual Modeling (ER'02)*, pages 429–443, Tampere, Finland, October 2002.

[LEW00] 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.

[MHH00] 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.

[MLM01] 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.

[RB01] E. Rahm and P.A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10:334–350, 2001.

[TH04] 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).

[XML01] XML Schema Part 0: Primer: W3C Recommendation, May 2001. URL: http://www.w3.org/TR/xmlschema-0/.

[XML03] XQuery 1.0: An XML Query Language, November 2003. URL: http://www.w3.org/TR/xquery/.