Facilitating Web-Based OLAP By Converting XML Data To UML Diagrams

Mikael R. Jensen   Thomas H. Møller   Torben Bach Pedersen

Department of Computer Science, Aalborg University, Fr. Bajers Vej 7E, 9220 Aalborg Ø, Denmark

Abstract

To support analytical applications such as On-Line Analytical Processing (OLAP), the demand for data integration in modern enterprises is rapidly becoming larger as more and more information sources appear. In many situations a logical (rather than physical) integration of data is preferable since some data is inherently not suited for storing in a physically integrated data warehouse. Previous web-based data integration efforts have focused almost exclusively on the logical level of data models, creating a need for techniques focused on the conceptual level. Extensible Markup Language (XML) is fast becoming the new standard for data representation and exchange on the World Wide Web, e.g., in B2B e-commerce, making it necessary for data analysis tools to handle XML data as well as traditional data formats..

This paper presents algorithms for automatically constructing UML diagrams from XML data and shows how to use the diagrams for the conceptual design of (virtual) data warehouses based on web data. An integration architecture is also presented. Unlike earlier work, the focus of this paper is the support of (OLAP) systems based on web data.

1 Introduction

To support analytical applications such as On-Line Analytical Processing (OLAP), integration of web data sources is becoming increasingly important as more business-relevant data appear on the web, e.g., on B2B marketplaces. Also, enterprises cooperate more tightly with their partners, creating a need for integrating the information from several enterprises for use in an analytical setting. The *data warehousing* approach dictates a physical integration of data, enabling fast evaluation of complex queries, but also demanding great effort in keeping the data warehouse up to date.

Email address: {`mrj,thm,tbp`}`@cs.auc.dk` (Mikael R. Jensen   Thomas H. Møller   Torben Bach Pedersen).

However, some kinds of data are impossible, or not attractive, to physically store in a data warehouse, either because of legislation or because of the nature of the data, e.g., rapidly evolving information. Enabling integrated use of such data requires a logical rather than physical integration.

XML is increasingly used for data exchange on the Web, making it attractive to use XML for building data warehouses from web data. Previous approaches for integrating web-based data, particularly in XML format, have focused almost exclusively on data integration at the *logical* level of the data model, creating a need for techniques that are usable at the conceptual level which is more suitable for use by system designers and end users. The most wide-spread conceptual model is the Unified Modeling Language (UML) [19].

The contribution of this paper is threefold. First, algorithms for automatically constructing UML diagrams from XML data are presented, enabling fast and easy graphical browsing of XML data sources on the web. The algorithms capture important semantic properties of the XML data such as precise cardinalities and aggregation (containment) relationships between the data elements. These precise properties would be lost if an intermediate translation (XML to relational, followed by relational to UML) was used. Second, an architecture for integrating XML data at the conceptual level is presented. The architecture also supports relational data sources and is thus well suited to build data warehouses which are based partly on in-house relational data and partly on XML data available on the web. Third, the process of integrating the generated UML diagrams into a dimensional model is described. The focus of the data integration effort is the support of OLAP systems, meaning that some design choices are different that in a more general setting, e.g., the order of data in the XML documents is considered unimportant as our focus is on *aggregation* queries that destroy document order anyway. We assume that the XML data has a Document Type Description (DTD) to specify it's structure. More powerful frameworks for describing XML documents, most notably XML Schema [26] exist, but none are yet official W3C recommendations. Also, only DTDs are currently widely used in real-world XML documents. Also, even if XML Schema is used, the process of designing the integrated database will still benefit significantly from the graphical notation found in UML (and not in XML Schema).

We believe this paper to be the first to present algorithms that automatically construct UML diagrams directly from XML data while retaining important semantic information, allowing for an easy overview of the available data sources. Also, we believe it to be the first to consider the important issue of integration of web-based data sources at the conceptual level and to focus on OLAP systems.

A number of systems allows integration of structured and/or semi-structured data on the web. The systems YAT [1], Strudel [5], TSIMMIS [7], and Ozone [11] all use a semi-structured/XML data model, Garlic [22] uses an object data model, Cohera [8] uses the relational (SQL) data model, while Clio [18] supports combined object-relational and XML data. These systems all focus on web data integration at the *logical* level. In comparison, this paper focus on integration of web data on the conceptual level, supporting the *design* process better. Also, our focus is on OLAP systems rather than generic data integration. However, we do not consider data integration at the logical level to be unnecessary, as such systems are used for the *implementation* of the integrated databases. Indeed, a DW conceptually designed using our approach could be implemented using these systems. The issue of converting XML data into relational data is described in [4,6,21]. In comparison, this paper aims at capturing more semantics of the XML data by using a higher-level conceptual model rather than providing efficient querying of XML data using RDBMSes. Database design tools such as ERwin [3] can "reverse-engineer" relational structures into UML diagrams. However, semantic information such as precise cardinalities and containment type relationships invariably get lost in the translation process. In comparison, this paper translates XML data directly into UML diagrams, keeping this valuable information. A lot of work has been done on conceptual schema integration [2,12,13,14]. Compared to this, we only want to map *selected* parts, i.e., not all, of the source schemas into our target schema. Also, our focus on OLAP means that the output of the integration process is different, e.g., we want to create "useful" dimensions with hierarchies, not general schemas that capture all of the source data.

The remainder of the paper is organized as follows. Section 2 defines algorithms for automatically generating a UML diagram from an XML DTD. Section 3 presents an integration architecture that enables XML and relational data at the conceptual level for constructing a "virtual" DW. Section 4 describes the process of integrating UML diagrams into a dimensional model. Section 5 summarizes and points to topics for future research.

2 Generating UML Diagrams From XML Data

This section describes algorithms for automatically generating UML diagrams from XML data, enabling fast and easy browsing of XML data available on the Web. The structure of XML data is visualized by a UML diagram derived from a DTD describing the XML data source. Furthermore, suggestions for determining types for XML elements are briefly presented. The purpose of the algorithms is to build a UML diagram which is at least as general as the document described by a DTD, meaning that XML data conforming to the DTD must be expressible in terms of the derived UML diagram. The algorithms generate UML according to the OMG Unified Modeling Language Specification version 1.3 [19]. Only a subset of the modeling constructs described in this specification are used, namely classes, typed attributes, and cardinality- and role-specified associations and aggregations. The algorithms are designed such that the structure of the original XML document is preserved and visible in the UML diagram.

The UML diagram has been chosen as the way of modeling and visualizing the DTD because UML is a standardized conceptual data modeling language which is familiar to many designers and end users, and because UML is powerful enough to express any document described by a DTD. The diagram is a conceptual easy understandable way of visualizing the structure of the XML document and allows the user quickly to get a feeling of the content and the structure of the data available. This supports easy browsing of data made available through XML on the Web. The algorithms for deriving a UML diagram from a DTD can in addition to the nesting constructs of XML also handle recursion between elements (both direct and indirect recursion by either nesting or reference) as well as references between different elements (ID/IDREF(S) constructs).

It is assumed that the XML data is described by a DTD and that it is valid according to this DTD. We also assume that all ID-references (IDREFS referring to some IDs) from a given element is a reference to elements of one type only. This means that for an element having multiple references defined (using an IDREFS attribute), each reference must be to the same element type. This assumption has been made since it in common data models, like the UML model, E-R model, and relational model, is generally impossible to model references to multiple unspecified elements [1] . Also, this only occurs very rarely in well-designed XML documents. It is furthermore assumed that it is possible to determine the relationship between any two elements having an ID/IDREF(S) relationship. This means that for every element having an IDREF or IDREFS attribute defined, it is possible to determine the element type to which a reference exist. The XML 1.0 Recommendation [25] for XML data is used throughout this paper

As noted above, frameworks that are more powerful than DTDs for describing XML documents exist. However, none of these are yet official W3C recommendations and only DTDs are currently widely used in real-world XML documents, thus we have based our approach on DTDs. The most prominent of the alternative description mechanisms is XML Schema [26]. XML Schema is a powerful modeling language that can capture advanced properties of the data such as attribute type information, different types of relationships, precise cardinalities, and inheritance with features such as virtual and final classes. However, the use of XML Schema to describe the XML data does not render our approach useless, as the process of understanding the data sources and designing the integrated OLAP database will still benefit significantly from the graphical notation available in UML (and not in XML Schema). Indeed, the auto-generated UML diagrams will likely be of a higher quality since important properties such as precise cardinalities, attribute types, and aggregation/association relationships can be transferred easily to the generated UML diagram. Some features of XML Schema such as virtual and final classes cannot be captured in standard UML. However, it is unlikely that such advanced features will

[1] The only way is to make zero-to-many relationships between all entities in the model, a rather unsatisfactory solution.

be used in schemas that are available to external users as these features will usually only be useful in in-house development environments where inheritance is used to facilitate re-use.

2.1 Pre-processing of the DTD

Before generating the UML diagram corresponding to the document described by the DTD, the DTD is simplified in order to make further processing easier. This simplifying process is here termed DTD pre-processing. It is not the intention to create a DTD that is equivalent to the original DTD. Instead the transformations creates a DTD which can be "less strict" than the original DTD, meaning that XML data described by the original DTD is "at least" described by the transformed DTD. This assures that the diagram generated from the DTD will be as general and high-level as possible, while still retaining the essential features of the original DTD.

In our case, all that matters in an XML document is the position of an element relative to its siblings and the parent-child relationship between elements, and it is therefore safe to apply the transformations since they preserve these conditions. The pre-processing removes elements not accessible from the document root, removes elements declared EMPTY which have not declared an ATTLIST and simplifies element content specifications

The first step towards simplifying the DTD is to remove all elements not accessible from the root. It is safe to apply this procedure since any XML document conforming to the DTD cannot make use of this type of elements [25]. All elements declared EMPTY which does not have an ATTLIST declared can safely be removed from the DTD. It is possible to make use of this type of elements in an XML document, but the element can contain no data and are therefore of no interest [25].

Most of the complexity of a DTD stems from the complex definition of element content specifications. By applying certain transformations to a DTD it is possible to simplify the DTD element content specifications. The transformations used in the pre-processing are a slight change of the transformations presented in [21], consisting in that "+" operators are not transformed into "*" operators. This is not done since it is possible to capture the "+" operator semantics in the UML model. Only a subset of the transformations is shown here.

Flattenings	**Simplifications**	**Groupings**
$(e_1,\ e_2)^* \rightarrow e_1^*, e_2^*$	$e_1^{**} \rightarrow e_1^*$	$..., a^*, ..., a^*, ... \rightarrow a^*, ...$
$(e_1,\ e_2)? \rightarrow e_1?, e_2?$	$e_1^*? \rightarrow e_1^*$	$..., a^*, ..., a?, ... \rightarrow a^*, ...$
$(e_1 \mid e_2) \rightarrow e_1?, e_2?$	$e_1?^* \rightarrow e_1^*$	$..., a?, ..., a^*, ... \rightarrow a^*, ...$
	$e_1?? \rightarrow e_1?$	$..., a?, ..., a?, ... \rightarrow a^*, ...$
		$..., a, ..., a, ... \rightarrow a^+, ...$

The first set of transformations convert a nested definition into a flat representation, meaning that the binary operators "," and "$|$" do not appear inside any operator. The second set of transformations reduces many unary operators to a single unary operator. The last set of transformations group sub-elements having the same name. Note that some of the transformations destroy document order. This is unproblematic since we are focusing on OLAP queries that do not query the document order of the detail data but rather aggregate several detail data items into one aggregate result, thus destroying document order anyway.

Example 2.1 The specification $<$*!ELEMENT* x $((b|c)^*, a, (d?|(e?, (b, b)^*))^*, a) >$ can be reduced to $<$*!ELEMENT* x $(b^*, c^*, a^+, d^*, e^*) >$.

2.2 DTD and UML Data Model

This section defines the formal data model specifications used to describe the DTD and the UML diagram components. Both the DTD and the UML data model are defined by means of an environment based on sets. The two data model specifications are defined in order to give a precise description of the generation and modification of a UML diagram.

Notation In order to ease the readability of the algorithms presented, the following notation will be used. All sets are written in *italics* with initial capital letter. A capital letter is used to separate words of a set, e.g. *MySet*. All elements of sets are written in *italics* with small letters. An underscore character is used to separate words of an element, e.g. *my_element*. All functions are written in sans serif with small letters. A capital letter is used to separate words of a function, e.g. myFunction. When a set consists of elements that are tuples square brackets ("[" and "]") are used to address a certain tuple element, e.g. when an element is defined by *my_element* $=$ $($*my_tuple_element*$1,$ *MyTupleElement*$2)$ then *my_element*$[$*my_tuple_element*$1]$ denotes the first element in the tuple. Furthermore, when a tuple consists of elements which are sets (like the second tuple from the former example) then a subscript will denote a certain element within this set, e.g. *my_element*$[$*MyTupleElement*$2_j]$ denotes element j in the set *MyTupleElement*2. We now define the sets needed to describe an XML DTD.

ElementNames = { e $|$ e is an element name defined in the DTD }
∪ { CDATA, PCDATA }
AttributeNames = { a $|$ a is an attribute name defined in the DTD }
DTDContent = { (*element_name, ElementContentSpec, AttContentSpec*) $|$
element_name ∈ *ElementNames* }
ElementContentSpec = { (*element_name, modifier*) $|$
element_name ∈ *ElementNames* $\wedge$
modifier ∈ *Modifiers* \ {REQUIRED, IMPLIED}}

$$\begin{aligned}
\mathit{AttContentSpec} = \{(&\mathit{att\_name}, \mathit{att\_type}, \mathit{modifier}) \mid \\
&\mathit{att\_name} \in \mathit{AttributeNames} \ \wedge \\
&\mathit{att\_type} \in \mathit{AttributeTypes} \ \wedge \\
&\mathit{modifier} \in \mathit{Modifiers} \setminus \{ *, +, ?, 1 \} \}
\end{aligned}$$

$$\mathit{Modifiers} = \{ *, +, ?, 1, \text{REQUIRED}, \text{IMPLIED} \}$$

$$\mathit{AttributeTypes} = \{ \text{ID}, \text{IDREF}, \text{IDREFS}, \text{CDATA} \}$$

The set *ElementNames* contains all names of elements declared with *!ELEMENT* in the DTD including the pre-defined character data elements CDATA and PC-DATA. *AttributeNames* contains all names of attributes declared with *!ATTLIST* in the DTD. The *DTDContent* set is a set of three-tuples describing the contents and structure of the DTD. The first element in the tuple is the name of an element from the DTD. The second and third elements in the tuple are both sets which are defined below. The *ElementContentSpec* set describes the content specification for all elements defined in the DTD. The set contains a two-tuple for each element listed in the element content specification for some element in the DTD. A modifier is associated with each element in accordance to the modifier specified in the DTD. For each element declared in the DTD a list of attributes can be associated with the element. The set *AttContentSpec* contains three-tuples where the first element in the tuple is the name of the declared attribute, the second element is the type of the attribute and the third element is the modifier associated with the attribute. *Modifiers* is the set of modifiers used to describe the cardinality of elements and attributes specified in the DTD. All attributes defined in a DTD are typed. *AttributeTypes* is the set of attribute types.

Definition A *leaf element* is declared in the DTD to have character data, e.g., PC-DATA, as content.

We now define the sets for describing the UML diagram. The set *UMLClasses* describes a complete UML diagram. It contains three-tuples where the first element in the tuple is the name of the UML class. The two other elements are both sets, where *AttributeList* is the list of attributes associated with this particular class and *PointsTo* is a list of classes that this class links to.

$$\begin{aligned}
\mathit{UMLClasses} = \{ (&\mathit{class\_name}, \mathit{AttributeList}, \mathit{PointsTo}) \mid \\
&\mathit{class\_name} \in \mathit{ElementNames} \}, \text{ where}
\end{aligned}$$

$$\begin{aligned}
\mathit{AttributeList} = \{ (&\mathit{att\_name}, \mathit{modifier}, \mathit{data\_type}) \mid \\
&\mathit{modifier} \in \mathit{Modifiers} \setminus \{ *, +, \text{REQUIRED}, \\
&\quad \text{IMPLIED} \} \\
&\wedge \mathit{data\_type} \in \mathit{DataTypes} \}, \text{ and}
\end{aligned}$$

$$\begin{aligned}
\mathit{PointsTo} = \{ (&\mathit{class\_name}, \mathit{link\_type}, \mathit{source\_card}, \mathit{target\_card}, \mathit{link\_role}) \mid \\
&\mathit{class\_name} \in \mathit{ElementNames} \ \wedge \ \mathit{link\_type} \in \mathit{LinkTypes} \\
&\wedge \mathit{source\_card}, \mathit{target\_card} \in \mathit{Cardinalities} \\
&\wedge \mathit{link\_role} \in \mathit{AttributeNames} \cup \{\text{NULL}\}
\end{aligned}$$

The *AttributeList* set describes the name, modifier and data type of each attribute in a UML class. The modifier can either be 1 or ?, describing whether or not null values are allowed for this attribute. The *data_type* element holds type information for this variable. The *PointsTo* describes a relationship from one class to another class. *class_name* is the name of the class to which a link exist, *link_type* is the type of link and *source_card* and *target_card* is the cardinality specified for the source and the target of the link, respectively. *link_role* is the role of the link. A link role is a description of a link, containing a name and a direction. In this case only the name is used since the direction is implicit (the link is always from the class holding the *PointsTo* element to the class named in *class_name*). Only association type links are given a role, since the meaning of an aggregation type link is always clear (aggregation means "contained in"), and links of this type are thus assigned NULL as *link_role*.

DataTypes = { NUMERIC, DATE, TEXT, NULL }
LinkTypes = { AGGREGATION, ASSOCIATION }
Cardinalities = { 0..*, 1..*, 0..1, 1 }

DataTypes captures the four basic data types for UML diagram attributes. Data types are not applicable when generating the UML model, since the DTD contains no type information (see Section 2.6). The set *LinkTypes* describes the different types of relations that can exist between any pair of classes in the UML diagram. *Cardinalities* is the set of cardinalities used to describe the quantitative relationship between elements in the UML data model. The cardinalities of a relationship are given by specifying minimum and maximum cardinalities. The cardinality "1" is used as a shorthand for "1..1".

Functions We now define the functions used in the conversion from DTDs to UML diagrams. All functions are defined by specifying the domain of definition and range. "$\rightarrow$" and "$\hookrightarrow$" denotes total and partial functions, respectively.

$\mathsf{target} : \mathit{AttContentSpec} \hookrightarrow \mathit{ElementNames}$
$\mathsf{cardinality} : \mathit{Modifiers} \cup \mathit{AttContentSpec} \rightarrow \mathit{Cardinalities}$
$\mathsf{parent} : \mathit{UMLClasses} \hookrightarrow \mathit{UMLClasses}$

The target function is given an attribute specification and returns the name of an element. The function is used to pair IDREF(S) and IDs, and is only defined for the subset of *AttContentSpec* having an *att_type* of either IDREF or IDREFS. The cardinality function is given either an element from *Modifiers* or an element from *AttContentSpec* and returns the cardinality corresponding to the modifier or the cardinality of the attribute. An attribute declared as IMPLIED has cardinality 0..1, a REQUIRED attribute has cardinality 1, an IMPLIED IDREFS attribute has cardinality 0..*, a REQUIRED IDREFS attribute has cardinality 1..*. The modifiers *, + and ? has cardinality 0..*, 1..* and 0..1, respectively. The parent function is given an element from *UMLClasses* and returns the parent element of this element as specified by the nesting relationship in the DTD. The function is only defined for the elements having exactly one parent.

2.3 DTD to UML Conversion Algorithm

The algorithm for generating the UML diagram has been divided into two parts. The first part, `generateUMLClass`, is a subfunction that generates one element in *UMLClasses* which corresponds to a complete class in the final UML diagram. The other part of the algorithm, `generateUML`, calls the `generateUMLClass` subfunction repeatedly until all the elements of *UMLClasses* have been generated. The algorithms generate only valid UML diagrams. The only UML components generated by the algorithm `generateUMLClass` are classes with zero or more attributes, cardinality-specified aggregations between classes and, role- and cardinality-specified associations between classes (see above for a definition of association roles). Aggregations are always constructed in direction from parent to child and associations are always constructed in direction from the referring element to the element bearing the corresponding ID.

In general the structure of the UML diagram generated by the algorithm corresponds to the tree-structure of the DTD where a UML class is generated for each element. All elements having a parent/child relationship in the DTD are connected by an aggregation relationship in the UML diagram and ID-references between elements are modelled as associations between the corresponding UML classes.

GenerateUMLClass Algorithm

```
(1)  generateUMLClass($x_i \in \mathit{DTDContent}$):
(2)  Generate new element $e \in \mathit{UMLClasses}$ where
(3)     $e[\mathit{class\_name}] = x_i[\mathit{element\_name}]$,
(4)     $e[\mathit{AttributeList}] = \{(a, b, c) \mid a = x_i[\mathit{AttContentSpec}_j][att\_name]$,
(5)          $b = \text{cardinality}(x_i[\mathit{AttContentSpec}_j][\mathit{modifier}]),\ c = \text{NULL} \wedge$
(6)          $x_i[\mathit{AttContentSpec}_j][att\_type] \notin \{ \text{IDREF, IDREFS} \} \} \ \cup$
(7)       $\{ (\text{“value”}, 1, \text{NULL}) \mid x_i \text{ is a leaf element}\}$, and
(8)     $e[\mathit{PointsTo}] = \{(d, e, f, g, h) \mid$
(9)          $d = x_i[\mathit{ElementContentSpec}_j][\mathit{element\_name}]$,
(10)         $e = \text{AGGREGATION},\ f = 1$,
(11)         $g = \text{cardinality}(x_i[\mathit{ElementContentSpec}_j][\mathit{modifier}]),\ h = \text{NULL} \ \wedge$
(12)         $x_i$ is not a leaf element $\} \cup$
(13)      $\{ (d, e, f, g, h) \mid d = \text{target}(x_i[\mathit{AttContentSpec}_j])$,
(14)         $e = \text{ASSOCIATION},\ f = \text{cardinality}(x_i[\mathit{AttContentSpec}_j]),\ g = 0..*$,
(15)         $h = x_i[\mathit{AttContentSpec}_j][att\_name] \wedge$
(16)         $x_i[\mathit{AttContentSpec}_j][att\_type] \in \{\text{IDREF, IDREFS}\}\}$
```

The subfunction `generateUMLClass` works as follows: The function is given an element $x_i \in$ *DTDContent* and generates an element $e \in$ *UMLClasses* which is the representation of a UML class. e[*class_name*] is set to the name of the element in the DTD to which x_i corresponds. e[*AttributeList*] is a set of attribute names, modifiers and data types for all the non IDREF and IDREFS type attributes defined for the UML class corresponding to x_i. The data type is not applicable since it cannot be determined at this point. If the element x_i is a leaf element an attribute of name "value" is added to the class to hold the element's data. This means that all elements having their content defined as CDATA or PCDATA will have the attribute "value" to hold this character data.

e[*PointsTo*] is a set of links to each of the elements within the content specification for the element corresponding to x_i. For each element in x_i's element content specification (i.e. elements nested beneath x_i) a link is created; if the elements *att_type* is IDREF(S) the link is an association, whereas it is an aggregation otherwise. The link type and cardinality is determined by the functions linkType and cardinality. For each element in the points-to set two cardinalities are specified; a cardinality associated with e[*class_name*] (*source_card*) and a cardinality associated with the class pointed to by e[*class_name*] (*target_card*). The cardinality *source_card* for links caused by attributes of types different from IDREF and IDREFS is always defined as 1 since elements in XML are always uniquely nested. If an attribute is of type IDREF(S) then *source_card* is equal to the output of the function cardinality. The cardinality *target_card* is for attributes of type IDREF and IDREFS always set to 0..* since it in XML documents described by DTDs not can be controlled how many elements are allowed to refer to an element having an ID attribute. A link can be from one element to itself indicating a recursive definition.

The link types are either aggregations or associations. Aggregations are used whenever a link to a nested element is established (modeling the "consists of" parent-child relationship) and associations are used when modeling an ID-reference relationship between two elements. A link role is supplied for each association type link, containing the name of the IDREF or IDREFS attribute causing the link. This helps resolving the conceptual meaning of association type links in the UML diagram, provided, of course, that the IDREF and IDREFS attributes are given meaningful names in the DTD.

GenerateUML Algorithm

```
(1)    generateUML:
(2)        $UMLClasses = \bigcup_i \{$ generateUMLClass($x_i \in$ DTDContent) $\}$
```

The algorithm works as follows: The set *UMLClasses* is generated by uniting the result of calling the `generateUMLClass` subfunction on every $x_i \in$ *DTDContent*.

To reduce the number of classes in the UML diagram *UMLClasses* is processed by the algorithm `postProcessUML`, seen below. The number of classes is reduced by removing all leaf elements having only one parent and no association relationships. The attributes and data contained within the leaf element is moved from the leaf element to its immediate parent, thereby making it possible to remove the now empty leaf element.

The algorithm works as follows: An element $e_i \in$ *UMLClasses* is a candidate for removal if e_i is a leaf element having one parent and no incoming or outgoing references. When a candidate element has been found, it is checked if the link from e_i's parent to e_i has a target cardinality of 1 or 0..1. If it has, e_i can be safely removed. The reason for only removing classes having this cardinality is that classes that are part of a "many" relationship are always modelled as separate classes in UML.

When an element e_i destined for removal has been located, the names of the attributes defined for the element are changed. The new attribute names are a concatenation of the name of the class, a dot (".") and the old attribute name (in the algorithm "$\circ$" is the concatenation operator). Finally, all the attributes from the about-to-be removed class are given a modifier corresponding to the class' cardinality and are then copied to the parent.

Post-processing Algorithm

```
(1)  postProcessUML:
(2)  $\forall e_i \in$ UMLClasses that are leaf elements with 1 parent and 0 refs:
(3)    if (parent($e_i$)[$PointsTo_j$][$class\_name$] $==$ $e_i$[$class\_name$]) $\wedge$
(4)       (parent($e_i$)[$PointsTo_j$][$target\_card$] $\in$ { 1, 0..1 }) then
(5)         $\forall a_m \in e_i[AttributeList]$ :
(6)            $a_m[att\_name] = e_i[class\_name] \circ "." \circ a_m[att\_name]$
(7)         if (parent($e_i$)[$PointsTo_j$][$target\_card$] $== 0..1$) then
(8)            $\forall z_k \in e_i[AttributeList]$ :   $z_k[modifier]$ = ?
(9)         endif
(10)        parent($e_i$)[$AttributeList$] = parent($e_i$)[$AttributeList$] $\cup$
               $e_i[AttributeList]$
(11)        parent($e_i$)[$PointsTo$] = parent($e_i$)[$PointsTo$] $\setminus$
               { parent($e_i$)[$PointsTo_j$] }
(12)        UMLClasses = UMLClasses $\setminus \{e_i\}$
(13)   endif
```

This section illustrates through an example how the previous algorithms work. It is assumed that the document used in the example is located at the address *http://www.componentheaven.com/parts.xml*.

Consider the following DTD, based on the Electronic Component Information Exchange (ECIX) QuickData Architecture [24], a project dedicated to the design of standards for B2B technical information exchange of component information for traditional electronic components. The root element of this DTD is the `class` element.

```
<!ELEMENT class ((ec, device)*) >
<!ATTLIST class name CDATA #REQUIRED >
<!ELEMENT ec (unitprice, pincount, gatecount, textdesc?) >
<!ATTLIST ec id ID #REQUIRED usedWithin IDREFS #IMPLIED
             name CDATA #REQUIRED >
<!ELEMENT device (textdesc, unitprice, device?) >
<!ATTLIST device id ID #REQUIRED name CDATA #REQUIRED >
<!ELEMENT unitprice (number, price) >
<!ELEMENT pincount (#PCDATA) >
<!ELEMENT gatecount (#PCDATA) >
<!ELEMENT textdesc (#PCDATA) >
<!ELEMENT number (#PCDATA) >
<!ELEMENT price (#PCDATA) >
```

When pre-processing the DTD according to the transformations described in Section 2.1 the element content specification for element "class" is reduced to `<!ELEMENT class (ec*, device*) >`.

We now proceed to construct the sets describing the DTD as defined above. Only an overview is given here, the details can be seen in Appendix A.

A UML diagram describing the DTD can be built directly from the sets describing the UML diagram. The UML diagram for the DTD in this example is shown in Figure 1. Notice the labelled arrow on the association type link from "ec" to "device", which is a representation of the *link_role* element associated with this link. The labelled arrow indicates the direction and name of this link, making it easier to understand the relationship between the two classes.

The UML diagram in Figure 1 has one class for each element specified in the DTD. To reduce the total number of classes the `postProcessUML` algorithm is applied to the *UMLClasses* set. The UML diagram resulting from the application of `postProcessUML` is shown in Figure 2. As can be seen from the figure, four classes have been removed by the post processing algorithm. This is caused by the "inlining" of simple leaf classes, resulting in a diagram that is more meaningful and

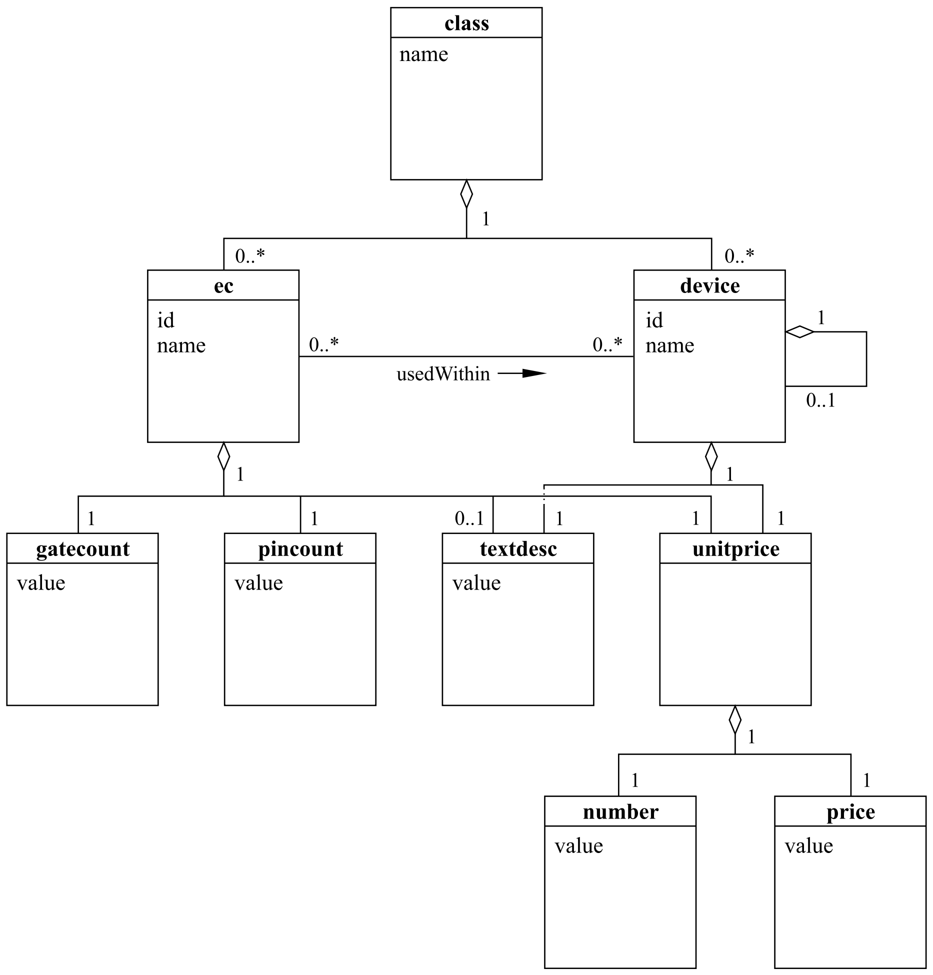

Figure 1. UML Diagram Produced by `generateUML` Algorithm

easier to understand for humans.

2.6 Determining Attribute Data Types

In the generation of the UML diagram, a set of attributes is associated with each of the classes, as described above. Since the UML diagram is generated directly from the XML DTD it is not possible to determine data types for the attributes at that point, and they are therefore typed with NULL. However, before constructing a DW, data types has to be determined for the attributes related to the post-processed UML diagram. We now discuss some of the problems associated with type determination and briefly present suggestions on how to resolve these problems.

One problem in determining attribute types is that XML 1.0 is a typeless language, so no data type declaration exists in the DTD or in the XML document. [25] The type space for XML is simply string values [25], but this information is to vague to be of any practical use in a DW system (e.g. when wanting to use aggregation

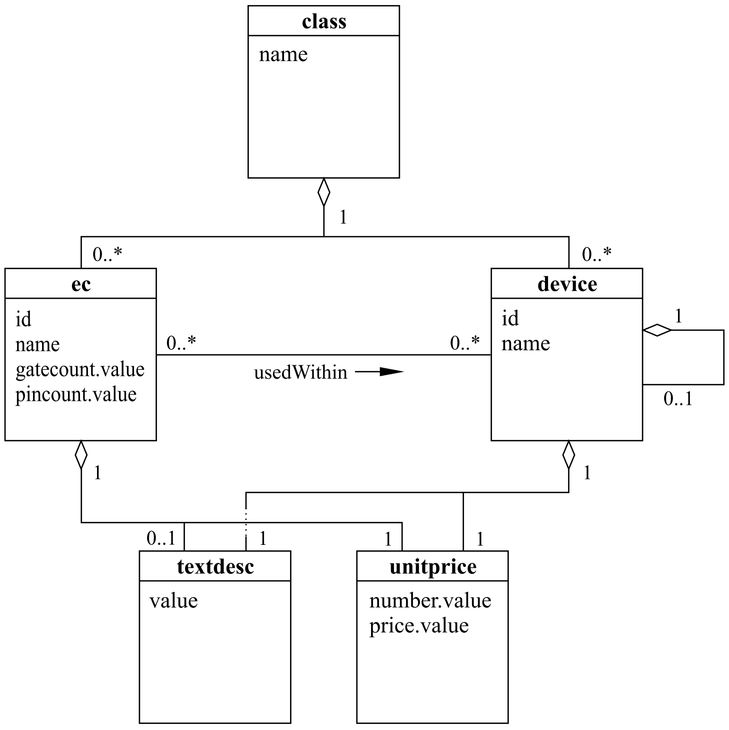

Figure 2. UML diagram after post-processing

functions other than count). Therefore the type space is extended as described above by the set *DataTypes* (having members NUMERIC, DATE, etc.). One way to assign type information to the attributes is to *sample* from the XML document. For each of the attributes in the UML diagram a number of samples can be done from the data contained within the corresponding XML tags, thereby making it possible with a certain probability to make a reasonable type determination.

Example 2.2 Sampling As an example of this approach consider the XML fragment `<price>123</price>`. By using sampling, the attribute price could be assigned the type NUMERIC.

The *designer* can also assist in determining types. It is fair to assume that the designer has some a priori knowledge about the data used for the analysis. By presenting the attributes of concern, the designer can make type assignments to these attributes based on a priori knowledge. Another way to resolve types is to simply provide a *lexical analysis* of the element declarations in the DTD, in order to make qualified guesses of types based on certain keywords.

Example 2.3 Lexical Analysis By using this approach, an element named “amount” defined as PCDATA could be assumed to contain data of type NUMERIC.

These methods can be combined in order to get the best possible assignment of types. As an example of a combined approach consider the following DTD fragment along with a fragment of an XML document conforming to the DTD:

DTD:	**XML document:**
`<!ELEMENT date (#PCDATA) >`	`<date>01/01/01</date>`

By applying the sampling method and lexical analysis of element definitions the element `date` could be assigned the type DATE. However, even though these methods can be used to provide type bindings for the attributes, complications exists. As described in Section 3, the only data initially fetched from the data source is the DTD. Not knowing the actual structure or size of the data makes it hard to make a sound statement about the statistical value of the sampling. Furthermore, having only a query interface to the data makes it impossible to sample at random, since determining a random position in the XML document is not possible. A way of determining types unambiguously is to read the entire XML document. This is not considered as an option due to the possibly very large documents that must be analyzed. It is beyond the scope of this paper to go further into the problems and solutions associated with type determination. Instead a function determineType is assumed, that assigns types from the set *DataTypes* to each of the elements in the set *AttributeList* associated with each of the UML classes.

3 Integration Architecture

This section describes a general system architecture for integrating XML and relational data sources in a web-based (virtual) DW. The architecture makes the use of XML data transparent to the data analysis tool and thereby to the end user.

XML Data Sources Before the designer can construct a DW based on XML data, the designer needs to know the structure of the data to be used. One way of obtaining this knowledge is to fetch the entire XML document from the Web. This solution is rather unsatisfactory, since fetching the entire XML document can take a considerable amount of time. Another solution is to consider the logical structure of the XML document, described by the DTD. Fetching only the DTD saves time, since the DTD is small in size compared to the document it describes. A potential problem associated with this approach is that the DTD can be overly complex, giving no contribution to the designer's understanding of the structure of the XML data. What is needed is an easy understandable and fast way of communicating the structure of an arbitrary XML data source to the designer.

To accomplish this goal, the DTD is transformed into a Unified Modeling Language (UML) diagram as described in the previous section. Visualizing the structure of XML documents in a graphical way is much easier for the designer to understand than a context free grammar such as the DTD. By describing the structure of XML data sources using a high-level conceptual graphical modeling language, each XML data source on the Web becomes an easy accessible and comprehensible database to both designers and end users. The use of DTDs as a basis for deriving the graphical representation of the data source supports fast browsing of the available data, due to the small size of the DTD. This corresponds to the traditional use of the Web where browsing is probably the most important mean of finding the desired information.

Relational Data Sources An integration of XML and relational data is necessary since an enterprise typically stores different types of transactional data in relational databases. As is the case for XML data, a conceptual model of the structure of the relational data is needed, in order for the designer to make qualified statements about the data to be used. It is not the focus of this paper to describe conceptual models for relational data and it is therefore assumed that UML diagrams describing the relational data can be made available to the designer. Otherwise, database design tools such as ERwin [3] can aid in the process by doing reverse engineering of relational schemas into UML diagrams.

General System Architecture Figure 3 illustrates the architecture which enables the use of XML data and/or relational data in a virtual DW for performing decision support using existing OLAP tools.

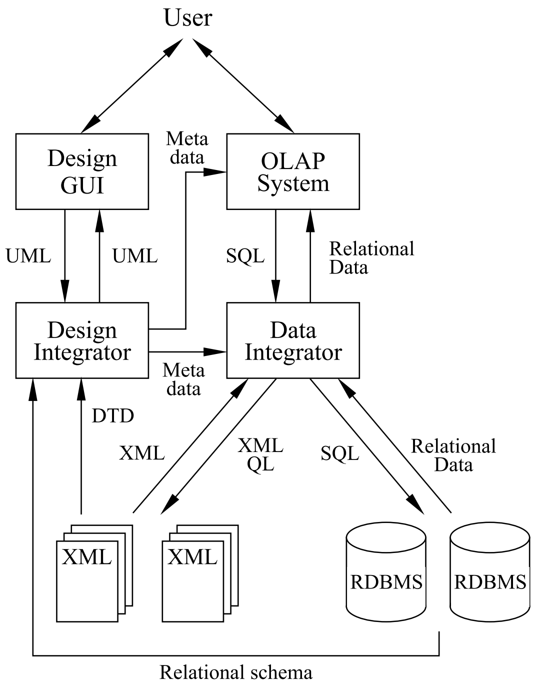

Figure 3. Architecture

The first part of the data integration process is the conceptual integration and OLAP design based on the UML diagrams, resulting in a conceptual model of the DW. The DW model (termed a UML snowflake diagram) is constructed by the designer through a graphical user interface (GUI). The UML snowflake diagram is constructed in a semi-automatic way from UML diagrams describing the data sources, which are generated by the Design Integrator. The construction process is described in the next section. The integration/design phase is similar to the “discovery” phase described in the Clio project [18], but is different because we focus on OLAP rather than general data integration and because a conceptual rather than a logical model is used. Finally, the UML snowflake diagram is transformed into relational structures by the Data Integrator, and are then made available to the OLAP System.

At run-time, the data sources are presented to the OLAP tools as relations, creating a relational multidimensional view (a star or snowflake schema [10]) over the data sources. There are several good reasons for using the relational model for the run-time system rather than the more powerful and general (conceptual or logical) models normally used for data integration such as object-oriented models (ODMG, UML) and semi-structured models (XML, OEM). First, almost all data analysis tools, e.g., OLAP tools, can access data using SQL, while only very few analysis tools can use XML query languages for data access. Second, no widely used query languages exist for conceptual models such as UML, so no OLAP tools would be able to use the constructed DW. Third, the distinguishing features of special XML query languages, e.g., querying document order, is not interesting w.r.t. OLAP queries. Fourth, OLAP queries are relatively simple, making the translation from SQL to XML query languages easier. Fifth, the OLAP schemas are sufficiently simple to be captured adequately by the relational model. This makes the relational model the obvious choice for the run-time system. The central component in the run-time systems is the Data Integrator which enables both XML data and relational data to be accessed using SQL queries through a standard interface like Open Database Connectivity (ODBC). Integrating XML this way makes the use of XML transparent to the end user since the OLAP tool cannot tell the difference between "true" relational structures and relational structures generated from XML data.

At query time, the user query the OLAP System using an OLAP query language native to the OLAP System, e.g. MultiDimensional eXpressions (MDX) [15]. These queries are transformed into one or more SQL queries by the OLAP System, used to query the relational structures generated from the UML Snowflake Diagram. The Data Integrator examines the SQL queries delivered by the OLAP System and transforms these queries into a series of queries expressed in either an XML query language or SQL, depending on whether the data originates from an XML source or a relational database management system (RDBMS). The results of these transformed queries (XML from XML data sources and relational tuples from RDBMSs) are then transformed into relational tuples fitting the relational schema generated from the UML snowflake diagram.

4 OLAP Schema Integration

Many data applications in data warehousing and electronic commerce require merging, coalescing and transforming data from multiple diverse sources into a new schema. At the core of all integration tasks lies the representation, understanding, and manipulation of schemas describing and structuring data. *Database integration* (or *global schema design*) is a process that takes several, possibly heterogeneous, schemas and integrates them into a view that provides a uniform interface for all the schemas [2]. If the local schemas are specified in different data models, they may be transformed into a common model before integration is performed.

All the schema integration methodologies investigated in [2] take the goal of the schema integration activity to be the construction of a single consensus schema. This is consistent with the ANSI/SPARC three-schema architecture [23] in which the conceptual schema is a unified representation of the whole domain.

Our focus, on the contrary, is on OLAP systems (i.e. *multidimensional* schemas) rather than generic data integration. This focus simplifies the schema integration process, since a single consensus schema created from multiple schemas is not the goal. Instead, the schema of interest is composed of several *partial* component schemas and has a rather specific structure, i.e., dimensions with hierarchies, meaning that integration process can be much more focused. Thus, identifying *equivalence of concepts* (e.g. behavioral, mapping, and transformational [2]) across the component schemas, such that schema equivalences are preserved, is not the focus. Moreover, the integration process is simplified by the fact that updates are not considered over the multidimensional schema, thereby yielding a *unidirectional* system [16], where the only operational goal is to query, through the multidimensional view, data stored under the local schemas constituting the view.

In [9] we present methods to construct a conceptual-level multidimensional view (termed a *UML snowflake diagram (USD)* [9]) that captures the desired multidimensional structure including *measures* that capture the properties we want to aggregate, e.g., sales, and dimensions with hierarchies that capture the ways we want to filter and group data, e.g., by time. The USD is constructed from XML and relational data sources, where each data source is described/visualized by a UML diagram, thereby easing the understanding and design process. One of the reasons that we have chosen UML as the common design model, is that UML is a semantically rich enough to describe the constructs present in XML and relational data so that valuable semantic information is not lost in the transformation. Notice that other formalisms for describing data sources with the aim of preserving semantics exist, e.g. the *hypergraph data model* [12,13]. The similarity with the work presented in [12,13] and our work is that both approaches uses a semantically rich *design* model, which is transformed into an appropriate *runtime* model, e.g. the relational model. The main difference is our focus on OLAP. In [14] it is furthermore demonstrated how hypergraphs can be used to model XML documents.

The designer constructs the USD by choosing the desired elements from various sources. Several source UML classes can be used to create one single USD class, and one source UML class can be used to create multiple USD classes. Informally, each class in the USD is constructed by choosing one class in a UML diagram as the *foundation* for the USD class. If the data described by this foundation class is not sufficient, the class can be extended by adding attributes from other classes, either from the same diagram or from UML classes originating from other data sources. In this way, a conceptual integration (forming a multidimensional view) of several possibly heterogeneous data sources is accomplished.

The first step in schema integration involves processing component schemas in some sequence. In general, the number of schemas considered for integration of each step can be $n \geq 1$. When constructing the USD the integration-processing strategy is *n-ary iterative* [2]; that is, first the OLAP DB designer constructs the *fact class* containing the *measures* of interest and thereafter each *dimension*, i.e., the data characterizing the measures, is created individually of each other. The strength of the n-ary iterative strategy w.r.t. OLAP lies in that dimensions can be constructed individually of each other, thereby simplifying the overall integration process; partial component schemas are merged first, creating intermediate integrated schemas (a whole hierarchically organized dimension), that at the final step is integrated with the constructed fact class. Additionally, the same source data is often used (in different ways) for several dimensions which is not supported by the global schema design approach that only represents the same data once, i.e., has no redundancy. In [9] we describe how to construct a UML snowflake diagram from various UML diagrams. A set of different operations can be performed on the component diagrams in order to construct the USD; three such operations are described below. Other operations include removal of classes and higher-level operations, see [9] for additional details.

Splitting a Class. One characteristic of the USD is that hierarchy levels in a dimension are explicit, unlike a relational "star schema" [10]. Each dimension defines a containment hierarchy, where higher levels can be thought of as containing lower levels, i.e., a hierarchy of generalization. It is possible that a single UML class constitutes an entire dimension, e.g., consider a UML class with the attributes *id, name, city,* and *country*. However, if we want to be use thus as a hierarchically organized dimension with, say, two levels, customer and country, the UML class would have to be split in two. We allow this since id *functionally determines* country. In order to split a class it is required that an attribute in the "lower" class n functionally determines an attribute in the "higher" class $n + 1$, such that any member in class n only has one "parent" in class $n + 1$. Thus, the resulting cardinalities from class n to class $n + 1$ is "1-*" to "1".

Adding an Attribute to a Class. USD classes often consists of a number of attributes originating from a single class in some data source. However, it is often desirable to add additional attributes to the USD. The way attributes are added to a USD class is through mapping queries [17], where explicit join conditions are established across the diagrams used. The method we have presented in [9] resembles the *data walk* construct found in Clio [17,18], the difference being that we work on static structure diagrams whereas Clio uses data instances and that our OLAP focus allows for a more directed process. If a UML class S_0 is extended with additional attributes from another UML class S_n, it is required that a path of UML classes $S_0, S_1, ..., S_n$ ($n \geq 0$) exists, where the cardinality of the relationship between S_0 and S_n is many-to-one. It is not required that the path from S_0 to S_n is located entirely within one UML diagram, making it possible to include attributes from different data sources. Furthermore, the attributes used to create new links between classes must have *modifier* equal to 1, in order to avoid NULL values in comparison expression. Notice that since we work with static structure diagrams, we can not ensure that a many-to-one relationship actually exists between the involved UML classes, because these relationships are not “physically” present (in form of either associations or aggregations). Thus, this property must also be checked at run-time.

Discarding an Attribute. Often a UML class contains more information (i.e. attributes) than actually needed for the analysis. If this case, it is possible to discard the attributes not needed. Of course, attributes used to establish relationships between UML classes (e.g. when adding an attribute) can not be discarded.

By using the operations described above, a USD can be “tailored” for the specific purpose. As an example, if a designer wants to merge two classes i and j this can be accomplished by adding the attributes from class i to j, discarding the attributes in class i, and discarding class i. Notice that this “merging operation” is different from merging in the traditional sense, where classes are merged if they are semantically equivalent; here classes are merged if they together contains the information for e.g. a single hierarchy level. In this way the USD constitutes an “addresslist” of attributes and their relationships, such that the data elements present in the USD (and *only* the elements present) can be fetched at runtime from the involved sources and transformed into the structure described by the USD.

In Figure 4, the class “fact” is the root of the USD. It has three dimensions, as indicated by the three associations between class “fact” and classes “ec”, “customer”, and “day,” namely the “component”, “customer”, and “time” dimension, respectively. The “customer” dimension contains a single hierarchy with levels “customer” and “country.” The “time” dimension contains two hierarchies, where “day”-“week”-“year” and “day”-“month”-“quarter”-“year” are the hierarchy levels for the two hierarchies. This way, the time hierarchy links dates to either week-year or month-quarter-year, respectively. The diagram has three *measures*, numbers that can be aggregated to higher levels, namely *sales_price*, *cost*, and *profit*.

Here, the “fact” class and the dimensions “customer” and “time” were constructed from relational data whereas the “component” dimension was constructed from XML data. With reference to Figure 2, the component dimension is constructed from the classes “class” and “ec”, illustrating that partial component schemas can be used to create hierarchically organized dimensions. Furthermore, it can be seen that the classes in Figure 2 can be used directly as a dimension without any trans-

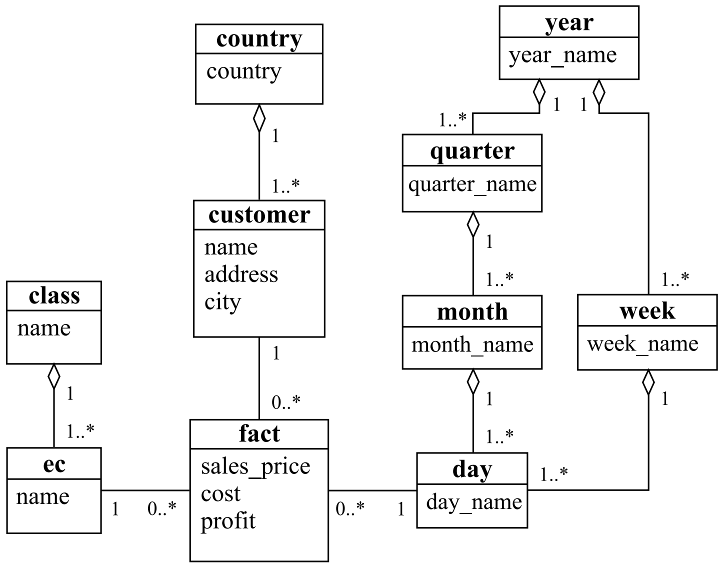

Figure 4. UML Snowflake Diagram

formations on the classes (except removing non-interesting attributes in the "ec" class). This is actually one of the strengths of XML w.r.t. OLAP; often XML data can be used directly as hierarchically organized dimensions. In [9] we describe a set of design patterns that can be used when XML is integrated for OLAP analysis.

5 Conclusions and Future Work

Motivated by the increasing need for OLAP on distributed data sources and the increased use of XML documents for exchanging information on the Web, this paper considers data integration at the conceptual level, with the purpose of building virtual web-based data warehouses.

First, algorithms for automatically constructing UML diagrams from XML data were presented, enabling fast and easy graphical browsing of XML data sources on the web. The algorithms captured important semantic properties of the XML data such as precise cardinalities and aggregation (containment) relationships between the data elements. Second, an architecture for integrating XML data at the conceptual and logical levels was presented. The architecture also supported relational data sources, making it well suited for building data warehouses which are based partly on in-house relational data and partly on XML data available on the web. Third, the process of integrating the generated UML diagrams into a dimensional model, termed the UML snowflake diagram, was described. This work is part of a project that investigates technologies for building data warehouses based on web data.

This paper improves on previous work on integration of web-based data by focusing on data integration at the conceptual rather than the logical level. The presented approach also improved over intermediate translations of XML to UML (XML to relational, followed by relational to UML). Another distinguishing feature is the focus on OLAP.

In future work, the first immediate step to be taken is to finish the implementation of a prototype utilizing the aspects described in this paper. A very important aspect of the implementation to investigate efficient query processing techniques such as query translations and data caching. Storing higher-level summaries of the data can speed up query processing considerably. We are currently working on these issues and the results so far are promising. Furthermore, if XML Schema advances to a W3C Recommendation it would be interesting to consider how to precisely use this richer formalism for instead of using DTDs. Other aspects of XML, such as whether preservation of document order is of relevance to OLAP analysis could also be investigated.

References

[1] Abiteboul, S. et al. Tools for Data Translation and Integration. *IEEE Data Engineering Bulletin* 22(1), pp. 3-8, 1999.

[2] Batini, C et. al. A Comparative Analysis of Methodologies for Database Schema Integration, *ACM Computing Surveys*, 18(4):323–364 , 1986.

[3] Computer Associates. ERwin Brochure. www.cai.com/products/alm/erwin/erwin_pd.pdf. Current as of July 7th, 2001.

[4] Deutsch, A. et al. Storing Semistructured Data with STORED. In Proceedings of *SIGMOD Conference*, pp. 431–442, 1999.

[5] Fernandez, M. F. et al. Declarative Specification of Web Sites with Strudel. *VLDB Journal* 9(1), pp. 38–55, 2000.

[6] Florescu D. and D. Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin* 22(3), pp. 27–34, 1999.

[7] Garcia-Molina H. et al. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems* 8(2), pp. 117–132, 1997.

[8] Hellerstein, J. M. et al. Independent, Open Enterprise Data Integration. *IEEE Data Engineering Bulletin*, 22(1), pp. 43–49, 1999.

[9] Jensen, M. R., T. H. Møller, and T. B. Pedersen. Specifying OLAP Cubes On XML Data. In *Proceedings of the Thirteenth International Conference on Scientific and Statistical Database Management*, to appear, 2001.

[10] Kimball, R. *The Data Warehouse Toolkit*, Wiley, 1996.

[11] Lahiri, T et. al. Ozone: Integrating Structured and Semistructured Data. In *Proceedings of the Seventh International Workshop on Database Programming Languages*, pp. 297–323, 1999.

[12] McBrien, P. et al. A Uniform Approach to Inter-model Transformations. In *Proceedings of The Eleventh International Conference on Advanced Information Systems Engineering*, pp. 333–348, 1999.

[13] McBrien, P. et al. A Formalisation of Semantic Schema Integration. *Information Systems* 23(5):307–334, 1998.

[14] McBrien, P. et al. A Semantic Approach to Integrating XML and Structured Data Sources. In *Proceedings of The Thirteenth International Conference on Advanced Information Systems Engineering*, pp. 330–345, 2001.

[15] Microsoft Corporation. SQL Server 2000 Analysis Services Whitepaper, 2000.

[16] Miller, R. J. et. al The Use of Information Capacity in Schema Integration and Translation. In *Proceedings of the Nineteenth International Conference on Very Large Databases*, pp. 120–133, 1993.

[17] Miller, R. J. et. al. Data-Driven Understanding and Refinement of Schema Mappings. In Proceedings of the *SIGMOD Conference*, 2001.

[18] Miller, R. J. et al. The Clio Project: Managing Heterogeneity. *SIGMOD Record* 30(1):78–83, 2001.

[19] Object Management Group. *Unified Modeling Language Specification 1.3*, www.rational.com/uml/resources/documentation/index.jsp. Current as of July 7th, 2001.

[20] Pinnock, J. et. al. *Professional XML*, Wrox Press, 2000.

[21] Shanmugasundaram, J. et. al. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of the Twentyfifth International Conference on Very Large Databases*, pp. 302–314, 1999.

[22] Roth, M. T. et al. The Garlic Project. In *Proceedings of SIGMOD Conference*, pp. 557, 1996.

[23] Sheth. A. et. al. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases, *ACM Computing Surveys*, 22(3):183–236, 1990.

[24] Silicon Integration Initiative *The Electronic Component Information Exchange QuickData Architecture*, www.si2.org/ecix/. Current as of July 7th, 2001.

[25] World Wide Web Consortium *Extensible Markup Language (XML) 1.0 (Second Edition)*, W3C Recommendation, www.w3.org/TR/2000/REC-xml-20001006, Oct. 6 2000. Current as of July 7th, 2001.

[26] World Wide Web Consortium *XML Schema*, W3C Candidate Recommendation, www.w3.org/XML/Schema.html. Current as of July 7th, 2001.

A Detailed Example

This section gives the details of the example given in Section 2.5. The sets describing the DTD are constructed as follows:

ElementNames = { class, ec, device, unitprice, name, pincount, gatecount,
textdesc, number, price, CDATA, PCDATA }

AttributeNames = { id, usedWithin, name }

The set *DTDContent* is a three-tuple of the element name, the element's content specification and the element's attribute specification. The second and third element in the tuple, which are the sets *ElementContentSpec* and *AttributeContentSpec* are defined in Section 2.

DTDContent = { (class, {(ec, *), (device, *), (name, 1)},
{(name, CDATA, REQUIRED)}),
(ec, {(unitprice, 1), (pincount, 1), (gatecount, 1), (textdesc, ?)},
{(id, ID, REQUIRED), (usedWithin, IDREFS, IMPLIED),
(name, CDATA, REQUIRED)}),
(device, {(textdesc, 1), (unitprice, 1), (device, ?)},
{(id, ID, REQUIRED), (name, CDATA, REQUIRED)}),
(unitprice, {(number, 1), (price, 1)}, {}),
(pincount, {(PCDATA, 1)}, {}),
(gatecount, {(PCDATA, 1)}, {}),
(textdesc, {(PCDATA, 1)}, {}),
(number, {(PCDATA, 1)}, {}),
(price, {(PCDATA, 1)}, {})}

When the algorithm `generateUML` is applied to the sets describing the DTD, it calls the subfunction `generateUMLClass` on every element $x_i \in$ *DTDContent*. For each x_i an element $e_i \in$ *UMLClasses* is generated. Thus when called on the "device" element from *DTDContent* (hereafter denoted x_{device}) the subfunction will generate an element which is hereafter denoted e_{device}.

e_{device} is constructed as follows:

e_{device}[*class_name*] = x_{device}[*element_name*] = "device"
e_{device}[*AttributeList*] = { $(\text{id}, 1, \text{NULL})$, $(\text{name}, 1, \text{NULL})$ }
e_{device}[*PointsTo*] = {("textdesc", AGGREGATION, 1, 1, NULL),
("unitprice", AGGREGATION, 1, 1, NULL)}
("device", AGGREGATION, 1, 0..1, NULL)}

The e_{device} class has two attributes defined, corresponding to the attribute "id" of type ID and the attribute "name" of type CDATA defined for this element in the DTD. The element has three links to the three classes "textdesc", "unitprice", and "device". The links are all aggregation type links since they are all children of the "device" element in the DTD. The link from e_{device} to "device" is a link from e_{device} to itself, indicating that "device" is recursively defined. All aggregation type links are given a *link_role* of NULL since there is no confusion about the meaning of aggregation type links.

As another example, the *UMLClasses*-element e_{ec} corresponding to the more complex XML element "ec" is specified below.

e_{ec} = $($"ec", $\{($id$, 1, \text{NULL}), ($name$, 1, \text{NULL})\}$,
$\{$("unitprice", AGGREGATION, 1, 1, NULL),
("pincount", AGGREGATION, 1, 1, NULL),
("gatecount", AGGREGATION, 1, 1, NULL),
("textdesc", AGGREGATION, 1, 0..1, NULL),
("device", ASSOCIATION, 0..*, 0..*, "usedWithin")$\})$

As above, an aggregation type link exist for all children of this element. In addition, an association type link exist to the "device" class because of the IDREFS attribute "usedWithin" declared for element "ec". This link is given the *link_role* "usedWithin" which is the name of the IDREFS attribute causing the link. As an example of the specification of a leaf element, the element $e_{gatecount} \in$ *UML-Classes* is shown below.

$e_{gatecount}$ = $($"gatecount", $\{$("value", 1, NULL$\}$, $\{\})$

The attribute "value" defined for the leaf element $e_{gatecount}$ is used to hold the character type data contained within the "gatecount" element. Since a leaf element is defined as being any element having a content specification of some character data type (e.g. CDATA or PCDATA) all leaf elements will carry the "value" attribute to hold this data.