XML Conceptual Modeling using UML

Rainer Conrad[1], Dieter Scheffner, and J. Christoph Freytag

Department of Computer Science
Humboldt-Universität zu Berlin, Germany,
{rainer.conrad|dieter.scheffner|freytag}@informatik.hu-berlin.de

Abstract. *The eXtensible Markup Language (XML) is increasingly finding acceptance as a standard for storing and exchanging structured and semi-structured information. With its expressive power, XML enables a great variety of applications relying on such structures - notably product catalogs, digital libraries, and electronic data interchange (EDI). As the data schema, an XML Document Type Definition (DTD) is a means by which documents and objects can be structured. Currently, there is no suitable way to model DTDs conceptually. Our approach is to model DTDs and thus classes of documents on the basis of UML (Unified Modeling Language). We consider UML to be the connecting link between software engineering and document design, i.e., it is possible to design object-oriented software together with the necessary XML structures. For this reason, we describe how to transform the static part of UML, i.e. class diagrams, into XML DTDs. The major challenge for the transformation is to define a suitable mapping reflecting the semantics of a UML specification in a DTD correctly. Because of XML's specific properties, we slightly extend the UML language in a UML-compliant way. Our approach provides the stepping stone to bridge the gap between object-oriented software design and the development of XML data schemata.*

1 Introduction

Because XML is widely accepted for storage and exchange of information, it enables a great variety of applications that depend on semistructured data within different fields of usage. This variety of applications demands various document structures. Document Type Definitions (DTDs) provide a grammar for creating document structures. They allow the designer to specify tailor-made structures, e.g. product catalogs, digital libraries, and electronic data interchange (EDI) in business-to-business e-commerce.

However, even the design of simple DTDs may cause difficulties, partly due to the textual form of the grammar itself. As we experienced in some cases, understanding DTDs is not intuitive. A DTD in its current textual form commonly lacks clarity and readability, therefore erroneous design and usage are inevitable.

As the data schema for documents, or objects respectively, XML DTDs suggest themselves to be modeled conceptually. The main goal of conceptual modeling is to separate the designer's intention from implementation details. In addition, the model's inherent visualization contributes to a better understanding

of the design. We propose to use a subset of UML for DTD design in particular and XML schemata in general. Using UML for document design, we are able to combine object-oriented software design with the XML document structures. Hence, conceptual modeling with UML helps to improve a redesign and to reveal possible structural weaknesses in the document design. Being extensible and adaptable, UML retains the expressiveness of target languages properly. One of XML's specific properties for which we extend UML in a compliant way, is the concept of order.

The structure of our paper is as follows: First, we give a brief overview of XML followed by a summary of related work. We then describe relevant modeling concepts of UML and its transformation into DTDs. With XML documents only having a static structure, we currently focus on the static model. Thereafter, we discuss our extensions necessary for the implementation dependent concepts of the modeling. Due to space limitation, our presentation focuses on the major aspects of our transformation model. An extended version of this paper can be found in [2].

2 XML

We briefly describe the XML concepts used for our transformation. First, we show how DTDs are arranged in the context of XML documents in order to give a better understanding of the DTD constructs afterwards.

2.1 XML documents

An XML document has a logical and a physical structure [10]. The physical structure of a document is made up of *entities* that are ordered hierarchically. The logical structure is explicitly characterized and described by "markups" which comprise declarations, elements, comments, character references, and processing instructions. Essentially, the document structure consists of an optional document type declaration containing the Document Type Definition (DTD), and a document instance. The purpose of a DTD is to provide a grammar for a class of documents.

2.2 XML DTD constructs

According to the XML specification, DTDs consist of markup declarations namely element declarations, attribute-list declarations, entity declarations, notation declarations, processing instructions, and comments [10]. As for these declarations, they are the elementary building blocks on which a DTD can be designed.

Element Type and Attribute-list Declarations make up the kernel of DTDs. Together, they declare the *valid* structures of a document instance, namely the nested element tags with their additional attributes. An element type declaration associates the element name with the element content. XML provides a

variety of facilities for the construction of the element content, namely *sequence* of elements, *choice* of elements, cardinality constructors (?, *, +), the types EMPTY, ANY, #PCDATA, and *mixed* content. *sequence* requires elements to have a fixed order, whereas *choice* expresses element alternatives. An EMPTY element has no content, whereas ANY indicates that the element can contain data of type #PCDATA or *any* other element defined in the DTD (comp. [5]). Mixed is useful when elements are supposed to contain character data (#PCDATA), optionally interspersed with child elements.

The name of the attribute list must match the name of the corresponding element. The list of attribute declarations consists of the attribute names, their types and default declarations.

Entity declarations serve the reuse of DTD fragments and text as well as the integration of unparsed data. An entity declaration binds an entity to an identifier. Being external entities, unparsed entities always have notation references.

Notation declarations provide a name for the format of an unparsed entity. They might be used as reference in entity declarations, and in attribute-list declarations as well as in attribute specifications.

Processing instructions (PIs) play an important role while checking integrity constraints of valid document instances. PIs have to be checked while parsing a document instance. The XML parser validates the document instance first and consumes the processing instructions known to XML. Then an application can handle more specific PIs.

3 Related Work

In this paper, we examine the mapping of UML to XML. However, we are aware of other data models for semistructured data. Since semistructured data are often regarded to be "schemaless", most of the following relevant models apply to instances and do not give an instance independent description of the data. We discuss some of these models, namely the Object Exchange Model, the Document Object Model, and the XML Data Model as described by the W3C.

The Object Exchange Model (OEM) was developed for the exchange of data between heterogeneous sources at Stanford University before it became a model for semistructured data [4]. OEM represents the hierarchical structure of document data using graphs. The graph based approach makes it possible to query document instances. Such queries handle various types of corresponding data from different documents. However, documents often are clustered in document classes like product catalogs, books, etc. OEM does not take this into account.

With the idea of providing an API for manipulating document instances, the W3C has defined the Document Object Model (DOM) [9]. A programmer equipped with DOM is in a position to create, navigate the structure, add, modify, or delete elements and content of any documents. DOM proposes an

implementation neutral and language independent API that assigns elements and attributes to interface definitions formulated in OMG's IDL. XML's logical structure parts are handled as objects in the same way. These objects are arranged in a graph to reflect the document structure. As a side effect, the DOM comes up with an instance based model that defines the logical structure of documents.

To visualize the structures of documents, W3C designed the XML Data Model [3]. The lexical structure is represented by a tree. The tree is extended by additional links representing references within the tree, thus generating a graph. The node types correspond to XML's logical structure. Being very simple and providing no more than a baseline, the XML Data Model supports conceptual modeling of document instances.

The models reviewed above only deal with modeling on instance level. Apart from this, modeling of more general structures on schema level, namely document classes, is desirable. This would facilitate a more abstract view on document structures according to database modeling, or software engineering respectively. By conceptually modeling DTDs with UML, the focal point of our approach is the schema level.

Our approach should not be confused with the XML Metadata Interchange (XMI) format [7]. The XMI describes the exchange format for UML class diagrams in an XML style. This way, XMI enables the exchange of schema information for cooperative work. However, our interest focuses on the development of DTDs by means of modeling of UML class diagrams and their mapping into XML DTDs.

4 Relevant Modeling Concepts and their Transformation

According to Rumbaugh, Jacobson, and Booch, *the Unified Modeling Language (UML) is a general-purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system* [8, p. 3]. For this reason, UML is widely accepted. Furthermore, UML incorporates an easy-to-understand graphical notation and visualization of interrelations, and correlations supporting the modeling of static structure and dynamic behavior of systems.

For our transformation approach, relevant constructs are those of UML's Static View and Model Management View. The Static View consists of classes and their relationship such as association, generalization, and various kinds of dependencies. The Model Management View describes the organization of the model itself, i.e., a model consists of a set of packages which in return are made up of model elements, e.g., classes as well as packages, recursively.

In the following we describe the UML constructs while explaining the transformation of these constructs into DTD fragments. Considering the pure conceptual view, this chapter provides an intuitive transformation into DTDs. For requirement reasons, it is the designer's responsibility to tune the result. Ad-

ditionally, the next chapter addresses the implementation dependent extensions which integrate tuning aspects into the diagrams.

4.1 Classes

Classes, characterizing objects with the same properties, consist of a class name, attributes, and methods. All objects must satisfy the constraints given by their class. Analogously, the content of an XML element is constrained by its type declaration. Thus, we transform UML classes into XML element type declarations.

The class names become the names of the element types. The attributes are transformed into element content description. This is motivated by considering class attributes to be the composite parts of a class. Therefore, dealing with attributes is analogous to aggregations as described below. However, UML aggregations do not support order, thus we determine the order to be from top to bottom.

The name of an attribute provides the name for the element type in content specification. We notice, the semantics of the UML attribute construct and XML content model for elements diverge. In UML attribute names are mandatory whereas the attribute types are optional. In contrast, an element content only consists of type names. There are no access names bound to these type names as in programming languages. For this reason, attribute names imply their attribute type names in our approach. If there is no class representing a suitable declaration for an attribute type, the attribute type is assumed to be an element whose content type is `#PCDATA`. Multiplicity specifications of attributes are mapped into cardinality specifications (with specifiers `?`, `*`, `+`) used for element content construction, e.g., [0..1] maps into `?`. Currently, class attributes and initial values are not supported. Figure 1 depicts a simple example of an author class with its transformation result.

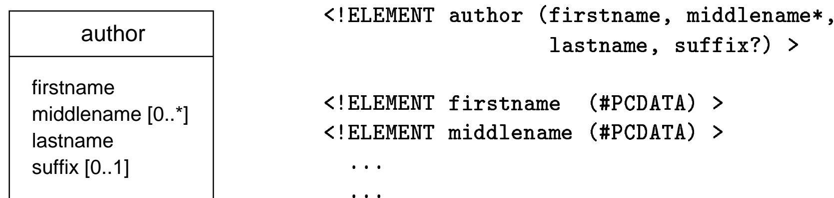


Fig. 1. Sample transformation

4.2 Aggregation and Composition

An aggregation specifies a whole-part relationship between an aggregate — a class that represents the whole — and a constituent part. Aggregation is a more

general form of composition where constituent parts directly depend on the whole part; they cannot exist independently. Composition mainly applies to attribute composition described above.

The only DTD construct that expresses a part-of relationship is the element content. XML element types can appear in several element contents. In general, it is impossible to restrict this appearance to exactly one content. Therefore, the concept of aggregation might be the predominant form that applies to XML element's content.

An aggregation might be refined with multiplicity. This multiplicity specification is semantically as rich as the cardinality specification of elements in element type content defined in XML. Thus, all forms of cardinality can be handled. Figure 2 shows an alternative modeling of Figure 1 using aggregation. The order in the sequence is implicit given by the notation itself.

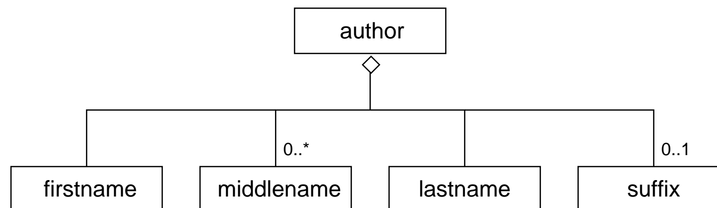


Fig. 2. Simple Aggregation

4.3 Generalization

The generalization concept defines a relationship between classes to build a taxonomy of classes: One class is a more general description of a set of other classes.

Figure 3 shows the class person specialized as employee or student, respectively. In addition, UML allows to constrain generalizations as follows: If subclasses have no common instances a comment with the keyword {disjoint} expressing the constraint has to be placed near the generalization symbol. Otherwise, the generalization is assumed to be overlapping which can be made explicit by providing the comment {overlapping}. If each instance shall be assigned to at least one subclass, the generalization must be constrained to {complete}. The counterpart of {complete} is {incomplete} being the default. Being orthogonal to one another, each pair reflects a different aspect of constraint.

Our approach adopts the idea of the transformation of the generalization construct into the Relational Model. To reflect the constraints' characteristics in overlapping generalizations, we propose to transform the general class into a superelement, i.e., the element type originating from the superclass (Figure 3). Such a superelement receives an additional "artificial" ID attribute declared `#REQUIRED`. Similarly, each subelement is augmented by a `#REQUIRED` IDREF

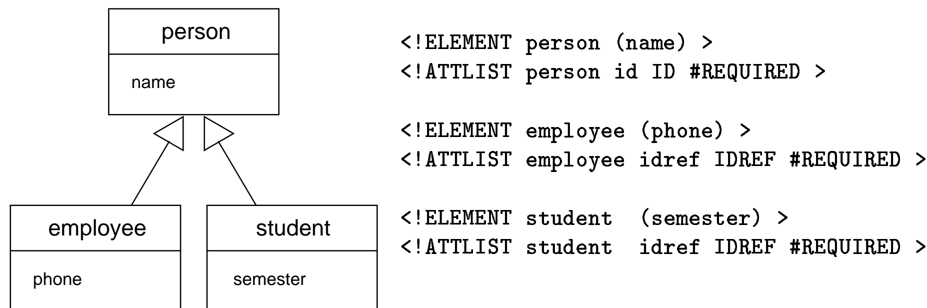


Fig. 3. Generalization transformation

attribute. Thus, a subelement instance references its general component. Note, the proposed transformation naturally applies to any generalization specified as {overlapping} and {incomplete}. To ensure the {complete} constraint in this context, we must prevent unreferenced superelement instances, therefore an additional integrity check of a document is necessary.

For generalizations specified as disjoint, we transform each class into its referring element type. Hereby, the instances are disjoint by nature. If such a generalization is further constrained by {complete}, the element type declaration of the superelement must be omitted in order to deny an element instance of that kind.

4.4 Association

Associations are relationships that describe connections among class instances. An association is a more general relationship than aggregation or generalization. A role may be assigned to each class taking part in an association, making the association a directed link. Associations are transformed into links between instances of two or more document classes in the sense of XML Linking Language (XLink) [11]. Because of the draft status of XLink, at this point, we only introduce some first ideas.

A simple link expresses a binary association. The following part of an attribute-list declaration for the referencing element is useful for representing an association.

```
xlink:type    (simple)    #FIXED "simple"
xlink:href    CDATA       #IMPLIED
xlink:title   CDATA       #IMPLIED
xlink:show    (new|replace|embed|undefined)  #FIXED "replace"
xlink:actuate (onLoad|onRequest|undefined)   #FIXED "onRequest"
```

For n-ary associations and for binary associations with multiplicity n:m, we suggest to use extended links. The element of xlink:type `extended` could contain

element types of xlink:types `locator`, `arc`, `resource`, and `title`. The usage of them and their attributes depends on the kind of association. A `locator`-type or a `resource`-type element type is necessary to specify the resource instances related to the association. `arc`-typed elements direct the link. A `title`-typed element type provides a name for the link.

4.5 Packages

A package is a concept for combining several concepts such as classes and packages into groups. Nesting, generalization, and import is applicable to packages such that packages may contain other packages, one or more packages can be the specialization of other packages, and packages can import other ones. Actually, packages also have names.

The only concept for modularization that XML offers is the entity concept. We distinguish complex and simple entities. Complex entities are always a set of complete markup declarations that again may contain entity declarations themselves, thus nesting of entities is supported. This nesting corresponds to the nesting of packages.

The packages appearing in a generalization relationship are transformed into single files. The specific subpackage declares an external entity (<!ENTITY % *superpkg* SYSTEM " *superpkg*.dtd">) and resolves the entity reference (%*superpkg-name*;) in its DTD. Parts of a specific subpackage overwrite parts of the superpackage by declaring these corresponding parts as entities of the same name. The entity reference of the overwriting part must be processed first, because an entity reference cannot be overwritten once it is resolved.

The usage of the import dependency between two packages leads to semantic losses at transformation into DTDs. The non-transitive extension of the namespace is not manageable in XML. Name clashes must be avoided. The transformation of the import dependency is similar to the transformation of the generalization concept.

5 Implementation dependent Concepts

We have to extend UML to take advantage of all facets that DTD concepts offer. Additionally, these extensions enable the tuning of the resulting models for guiding the transformation process. It is important for the extensions to be UML compliant and as minimal as possible.

5.1 Classes

We extend UML with diverse class and attribute stereotypes. They are useful for implementation dependent modeling of element attributes, element content, entity declarations, and notation declarations.

Meta Data Attributes. Up to now in our transformation a class attribute refers to element content. XML, however, allows additionally to constrain an element type by means of (element) attributes, i.e. attributes providing additional (meta) information for the element itself. In order to model such *meta data attributes*, we mark them ≪meta≫. Thus, we are able to distinguish between "normal" data attributes and meta data attributes.

Meta data attributes are transformed into attributes of the element's attribute-list. Specifically, the name of a class attribute and the type are transformed into their equivalent counterpart. UML does not constrain types. Thus, we can use specific XML attribute types like `CDATA`, `ID`, and even enumeration types.

Multiplicity specifications and initial values yield default declarations. For attributes, we restrict multiplicity to [0..1]. This specification translates into either an attribute with the default declaration `#IMPLIED` or an attribute with a given default value. Both forms can be distinguished, because an `#IMPLIED` attribute must not have a default value, but attributes with default must have one. If attribute multiplicity is not explicit, the attribute is assumed to be `#REQUIRED`. Being valid for each instance of the corresponding class, class attributes refer to `#FIXED` attributes.

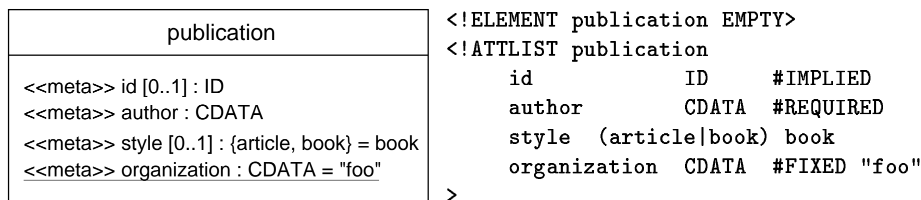


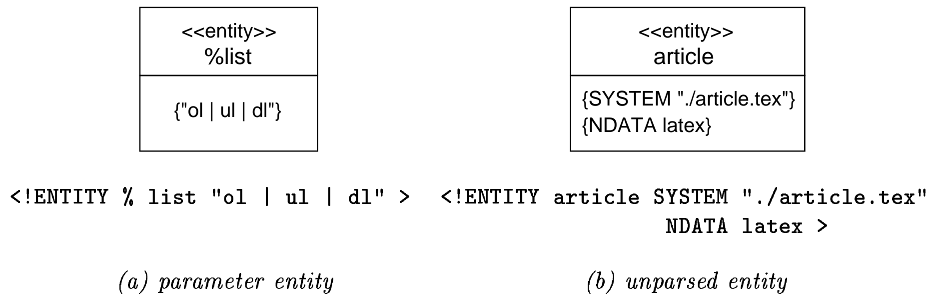
Fig. 4. An example transformation

Figure 4 depicts a simple example of a publication class with its transformation result. Note, in contrast with XML, a UML class diagram emphasizes the logical interrelation of element type declarations with their attribute-list declarations.

The Content Stereotype. To model the element content types `ANY` and `#PCDATA`, we create the ≪content≫ stereotype (see Figure 5). There is no need to model the content type `EMPTY`. It is up to the designer to use a UML comment.

Fig. 5. Class stereotype content

The Entity Stereotype. The ≪entity≫ stereotype refers to parsed and unparsed simple entities. In order to distinguish global entities and parameter entities on modeling, we only separate them by name, i.e., parameter entities, only visible on DTD level, get the prefix % (see Figure 6 (a)).



(a) parameter entity *(b) unparsed entity*

Fig. 6. Simple entities

Complex entities are always a set of complete markup declarations. In contrast, simple entities are represented by classes stereotyped ≪entity≫. Being either an *entity value* or an external identifier, the *entity definition* as well as the reference to the corresponding notation declaration is a UML comment as shown in Figure 6 (b).

The Notation Stereotype. In contrast to other markup declarations, notation declarations have a special meaning. Notations refer to helper applications which are capable to process data. A notation declaration binds an external identifier for a given notation to a name that can be used as a reference in entity declarations, and in attribute specifications of attribute-list declarations.

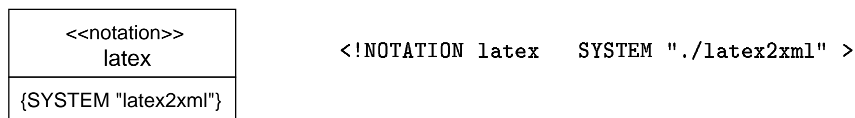


Fig. 7. The notation declaration

Referring to helper applications, a notation suggests itself to be a ≪utility≫ class. For this reason, we introduce the stereotype ≪notation≫ to the class latex in the example of Figure 7. The external application required is identified by its external identifier {SYSTEM "latex2xml"}.

5.2 Aggregation

In Section 4.2 we proposed a standard transformation for aggregations using element content. Nonetheless, to model XML content structures like *sequence*, *choice*, and *mixed* properly, still causes some problems.

Sequence. UML lacks the support of ordering of classes, attributes etc. explicitly, because it is not necessary to have aggregations or compositions ordered in programming objects. By default, we consider an aggregation to be transformed into XML *sequence*. This implies an implicit order given by the model. To model a sequence explicitly, the comment {sequence} may be used. We are able to specify the order of elements with the help of comments (see Figure 8). The first element is marked as {1}, the second as {2}, and so forth. Moreover, a sequence may obtain cardinality, e.g. {sequence : 1} (see Figure 8). In case of the trivial cardinality (1), both the colon and the 1 can be dropped.

We emphasize that UML propagates the comment {ordered} for usage in aggregations. In contrast to class sequences, the specifier {ordered} only applies to the order of objects of the same class. Hereby, {ordered} appears with cardinality greater than one.

Choice. Unfortunately, there is no appropriate concept available in UML that tackles the problem of handling alternative classes which are members of other classes. Often, choices originate from generalizations as shown in Section 5.3. If nevertheless a choice should be modeled, e.g., to be closer to the implementation, it is expressible with a clumsy structure of some helper classes and a generalization.

Choices describe alternative element contents that are mutual exclusive. Principally, the {xor} constraint is applicable for this kind of content. However, it conflicts with the repetitive use of the choice constructor due to the cardinality specification. Therefore, we propose the {choice} constraint of which the use is analogous to the mapping of the sequence construct. Figure 8 shows an application example taken from ETDML DTD [6]. With this semantics, we are given an elegant method to describe an element's *choice* content. The only difference between *sequence* and *choice* is found in the keywords *sequence* and *choice* which describe the type of aggregation. Analogous to *sequence*, *choice* can have its own cardinality — in Figure 8 it is expressed by {choice : 0..*} and this directly refers to the * in `( p | citation | table | mm )*`.

It is easy to combine the concepts of sequence and choice with one another in any way that is necessary for the model. Figure 8 illustrates that the structure of model schema follows the grammar rule provided by the element definition naturally, and is therefore easy to understand on the one hand, and on the other hand easy to handle while designing a model.

Mixed Content. The *mixed content* is subdivided into simple mixed content, and complex mixed content. We speak of complex mixed content when elements of that type of content may contain `#PCDATA`, optionally interspersed with child

```
<!ELEMENT chapter
   ( head?,  ( p | citation | table | mm )*, section* )
>
```

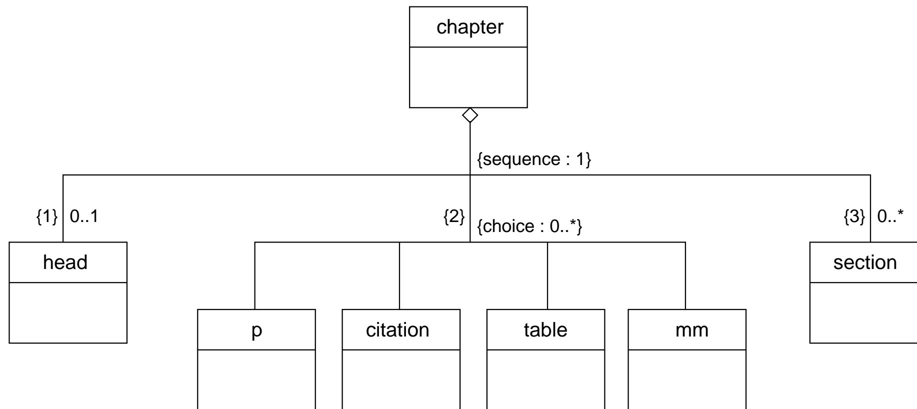


Fig. 8. A more complex example taken from ETDML DTD

elements. Using the *choice* modeling concept introduced above, the *mixed content* has to be expressed by a choice with cardinality `0..*`. Appearing once within the choice, the ≪content≫ class #PCDATA is the crucial feature which makes *mixed content* different from *choice content*. Simple mixed content is only made up of `#PCDATA`.

5.3 Generalization

The mapping of the generalization concept reviewed in Section 4.3 results in different element types corresponding to their classes. For some DTDs, it might be desirable to collapse the generalization structure into one class definition. Therefore, we adapt the idea of universal relations for element type declarations. To express this transformation, the generalization specification obtains a {discriminator} constraint. A single element type combines all variant properties found in the generalization, i.e., all element attributes as well as the element contents have to be merged in one attribute-list declaration or one element content specification, respectively.

With application dependent processing instructions to check integrity constraints, we support attribute alternatives. Attribute alternatives require attributes to have null values which are produced by `#IMPLIED` default declarations. In our approach discriminator attributes indicate the class an instance belongs to. For example,
`uml:discriminator:author (author-person|author-org) #REQUIRED`
contained in the attribute-list declaration of author supplies the discriminator attribute for the generalization in Figure 9.

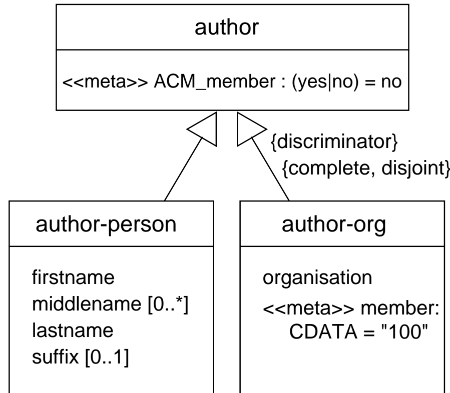


Fig. 9. Generalization using discriminator

The following processing instruction handles variant attributes of the subclasses; the suffix of the qualified attribute (`author:author-org`) must match a value of the discriminator attribute:

```
<?UML:DISCRIMINATOR author
    <!ATTLIST author:author-org  member CDATA "100">
?>
```

The merging of element contents depends on the generalization constraints as presented in the table below; cont_i denotes the element content of the transformed class i, with “1” representing the superclass:

	incomplete	complete
disjoint	cont_1,(cont_2|cont_3)?	cont_1,(cont_2|cont_3)
overlapping	cont_1,cont_2?,cont_3?	cont_1,((cont_2,cont_3?)|cont_3)

6 Conclusion

To bridge the gap between object-oriented software design and the development of XML data schemata, we use essential parts of static UML to model XML data schemata. The focal point of our approach is to find a suitable mapping from UML into XML DTDs. Being close to the real world, UML concepts for DTD design improve the design process and clarify the understanding of DTD semantics.

Starting from the abstract modeling, we develop an almost complete mapping. To exploit all DTD constructs, we slightly extended UML, thus providing a more flexible modeling approach. All proposed extensions are UML compliant and as minimal as possible.

For future work, the transformation of UML concepts must be validated in case studies. Furthermore, we plan to adapt our approach to XML Schema. That

is a DTD based method that allows the user to design DTDs with additional constraints. In previous work we applied our conceptual modeling approach to information integration [1]. We plan to pursue the UML-XML mapping approach in the context of information integration. For practical purposes, a CASE tool adopting the transformation is desirable. So far, we ignored the dynamic aspects when defining UML classes. We further have to investigate if and how class specific dynamic specifications are treated.

We believe that a combination of UML design with a UML to XML mapping is important enough to support document design in XML, thus influencing future developments of XML.

References

1. R. Conrad and D. Scheffner. Conceptual Modeling of XML for Integration Purposes (in German). In R.-D. Kutsche, U. Leser, and J.C. Freytag, editors, *4. Workshop "Föderierte Datenbanken"*. TU Berlin, 1999. Forschungsberichte des Fachbereichs Informatik.
2. R. Conrad and D. Scheffner. XML Conceptual Modeling using UML. Technical report, Institute of computer science, HU Berlin, 2000. Extended version; to appear.
3. World Wide Web Consortium. The XML Data Model. `http://www.w3.org/XML/Datamodel.html`, Jan 2000.
4. Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, pages 436–445. Morgan Kaufmann, 1997.
5. Michel Goosens and Janne Saarela. A Practical Introduction to SGML. In *TUGboat*, volume 16(3), 1995.
6. Neill Kipp, Zaodat Rahman, and Marc Bjorklund. Electronic Thesis and Dissertation Markup Language Version 2. Technical Report Version 2.0 Beta, Virginia Polytechnic Institute and State University, December 1998.
7. OMG. XML Metadata Interchange (XMI). Technical report, OMG, Oct 1998.
8. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison Wesley, 1999.
9. World Wide Web Consortium. Document Object Model (DOM) Level 1 Specification, Version 1.0. Technical Report REC-DOM-Level-1-19981001, W3C, October 1998.
10. World Wide Web Consortium. Extensible Markup Language (XML), Version 1.0. Technical Report REC-xml-19980210, W3C, February 1998.
11. World Wide Web Consortium. XML Linking Language (XLink). Technical Report WD-xlink-2000021, W3C, February 2000.