Konzeptionelle Modellierung von XML für die Integration

Rainer Conrad, Dieter Scheffner*
LFE Datenbanken und Informationssysteme
Institut für Informatik
HU Berlin

Zusammenfassung

Semistrukturierte Daten werden zunehmend im Internet in Form von XML–Dokumenten gespeichert bzw. übertragen. XML sieht es vor, daß die Dokumente durch die Element–Tags und deren Attribute mit Strukturinformationen und weiteren Meta–Daten angereichert werden können. Die Behandlung der Integration dieser im Internet verteilt liegenden Dokumente sowie der Integration von Datenbanken via XML–Dokumente wird immer wichtiger werden, da unterschiedlichste Strukturen mit XML definiert werden können und auch werden. Der Ansatz, der in diesem Beitrag vorgestellt wird, wird es ermöglichen, auf elegante Art und Weise die Integrationsergebnisse, die gerade im Bereich der föderierten Datenbanksysteme entwickelt worden sind, auf die Integration im Bereich XML direkt anzupassen. Die Grundidee besteht in der konzeptionellen Modellierung der Strukturen, die durch die DTD definiert werden.

1 Einleitung

Das Internet ist in den letzten Jahren zu einem riesigen verteilten heterogenen und wenig strukturierten Datenspeicher geworden. Die Entwicklung von XML (Extended Markup Language) [Wor98a] als deutlich abgespeckte Version von SGML (Standard Generalized Markup Language) [Mic95] liefert im Gegensatz zu HTML die Möglichkeit den Dokumenten mehr Strukturierung und mehr Informationen hinzuzufügen, da hier zu den eigentlichen Daten auch Meta–Daten in Form von selbst definierbaren Element–Tags und zusätzlichen Attributen hinzugefügt werden können. Durch die unterschiedliche Definition von Tag–Mengen und –Kombinationen, die die Gestaltung eines Dokumentes auf diese Strukturen einschränken, entstehen verschiedene Dokument–Typen. Insbesondere zur Gewinnung von Informationen werden Dokumente verschiedenster Typen angefragt werden, so daß die Informationsintegration von XML-Dokumenten einen hohen Stellenwert erhalten wird. Die entstehende Integrationsproblematik ähnelt der bei der Föderation von Datenbanken sehr. Für XML wird die Integration derzeit nur auf technischer Ebene und nur in eingeschränktem Maße unterstützt.

Unser Ansatz erlaubt es, XML–Dokument–Typen konzeptionell — also auf der Ebene eines Daten–Schemas — zu modellieren und größtenteils automatisch in Definitionen von Dokument–Typen, sogenannten DTDs, zu transformieren. Da damit auch das „Reverse–Engineering“ ermöglicht wird, kann auf Basis der konzeptionellen Modellierung die Integration der Dokument–Typen vorgenommen werden. Des weiteren wird so der Datenaustausch zwischen heterogenen Datenbanken mittels XML einfach ermöglicht.

* {rconrad|scheffne}@dbis.informatik.hu-berlin.de

Für die konzeptionelle Modellierung wird die Unified Modeling Language (UML) verwendet, die für diese Zwecke minimal so erweitert wurde, daß sich Eigenschaften von XML modellieren lassen, die sowohl beim Datenbankentwurf als auch beim Softwareentwurf nicht relevant sind. UML wurde zur konzeptionellen Modellierung einerseits verwendet, um keinen Bruch zur objektorientierten Entwicklung der Software entstehen zu lassen, die auf der Basis der XML–Dokumente arbeiten soll. Andererseits bietet sie die Möglichkeit der Erweiterung und Anpassung, so daß die Ausdrucksstärke der Zielsprachen abgebildet werden kann.

Im nächsten Abschnitt werden wir für das allgemeine Verständnis die wesentlichen Konstrukte von XML vorstellen und den Begriff der semistrukturierten Daten erläutern, der in diesem Kontext eine große Rolle spielt. Im ersten Schwerpunkt dieses Beitrags werden die konzeptionelle Modellierung mittels der UML, die notwendigen Erweiterungen und die Transformation in XML–DTDs vorgestellt. Im vierten Abschnitt wird als zweiter Schwerpunkt gezeigt, wie einerseits XML–Dokumente mittels der konzeptionellen Modellierung integriert werden können und wie andererseits der Datenaustausch zwischen beliebigen Datenquellen mit XML unterstützt werden kann.

2 Semistrukturierte Daten und XML

Die Behandlung semistrukturierter Daten erlangt durch das Internet eine immer größer werdende Bedeutung. Immer mehr Daten, die nicht voll strukturiert wie insbesondere in Datenbanken gehalten werden, sollen verarbeitet werden können. Zum besseren Verständnis wird in diesem einführenden Abschnitt zuerst der Begriff der semistrukturierten Daten näher erläutert, bevor dann auf die Konstrukte und Möglichkeiten von XML eingegangen wird.

2.1 Semistrukturierte Daten

Gerade die Entwicklung des Internets hat aufgezeigt, daß die Speicherung und der Austausch von Daten nicht nur strukturiert im Datenbanksinne erfolgt sondern häufig auch in textueller Form, bei der die Daten zum Teil von Meta–Daten und zum Teil von weiterem Text umgeben sind. Für diese Art von Daten wurde der Begriff der *semistrukturierten Daten* geprägt. In [SH99] wurden nach [Abi97, Dit98, HP99] folgende Merkmale, von denen mindestens eins gelten muß, formuliert. Zu diesen Merkmalen wird ergänzend angemerkt, welche Aspekte für XML gelten:

- Das Datenschema wird implizit in jedem Dokument und nicht in einem zentralen Katalog gehalten. — Jedes XML–Dokument kann mit einer Dokument–Typ–Definition (DTD), die im Dokumentkopf steht, ein Datenschema enthalten. Dieses Schema kann durch die Tag–Attribute bei jedem Datum weiter mit Meta-Daten angereichert werden. Dokumente ohne DTD sind erlaubt, so daß kein Datenschema explizit gehalten wird.
- Daten desselben „Typs“ können in der Struktur wechseln, z.B. durch differierende, fehlende oder zusätzliche Attribute. — XML strukturiert an dieser Stelle durch die DTDs. Allerdings sind Strukturwechsel insbesondere durch die Optionalität von Elementen und Attributen immer noch möglich und vermutlich deutlich häufiger zu finden als in vollstrukturierten Datenbanken.
- Angegebene Datentypen werden nicht als Integritätsbedingungen angesehen und werden dementsprechend nicht überprüft. — Derzeit gilt dies für XML uneingeschränkt.

Es wäre aber denkbar, daß die XML–Parser in Zukunft Datentypen überprüfen. Des weiteren entsteht mit XML–Schema [Wor98b] eine Variante, in der auch Datentypen explizit betrachtet werden.

- Die Anzahl der (möglichen) Attribute und die Vielfalt ihrer internen Strukturierung ist sehr groß. — Dies läßt sich auch auf XML übertragen.
- Die Attribute oder die Strukturierung unterliegt häufigen Änderungen. — Durch die Möglichkeit der Festschreibung der Strukturen in den DTDs wird dieses verlangsamt. Es ist aber in einem neuen Dokument ohne weiteres möglich, Teile von DTDs neu– oder umzudefinieren.
- Zwischen Daten und dem Schema kann nicht immer deutlich unterschieden werden. — Insbesondere für die Tag–Attribute, in denen auch Daten enthalten sein können, gilt dies.
- Anfragen beziehen sich häufig auf ein ganzes Dokument oder Teile davon und nicht nur auf Attribute. — Die Texte der XML–Dokumente werden häufig als Ganzes oder zumindest in Teilen betrachtet, so daß Anfragen üblicherweise elementübergreifend sind.

Zusammenfassend läßt sich sagen, daß XML–Dokumente im allgemeinen semistrukturiert sind. XML bietet aber die Möglichkeit, semistrukturierte Daten besser zu strukturieren, da hier Meta–Daten in Form von Element–Tags und Attributen dem Text hinzugefügt werden können. Diese Meta–Daten lassen sich dann auch für das Layout eines Dokuments verwenden.

2.2 XML im Überblick

Ein wesentlicher Aspekt bei XML ist die Trennung von Struktur und Inhalt eines Dokuments von seinem Layout. Hierdurch erlangt XML eine besondere Bedeutung für den Daten– und Informationsaustausch bzw. deren Speicherung im Web.

XML–Dokumente besitzen sowohl eine logische als auch physische Struktur [Wor98a]. Physisch besteht die Struktur eines XML–Dokuments aus Einheiten (*entities*). Jedes Dokument besitzt genau eine Dokumenteinheit (*root entity*), die in keiner weiteren Einheit enthalten sein darf. Ansonsten bestehen zwischen den Einheiten Inklusionsabhängigkeiten. Die logische Struktur wird explizit durch Tags gekennzeichnet bzw. beschrieben und umfaßt Deklarationen, Elemente, Kommentare, Zeichensatzreferenzen (*character references*) und Verarbeitungsinstruktionen (*processing instructions*) [Wor98a]. Wesentlich für XML–Dokumente ist die Unterteilung der Dokumente in XML–Deklaration, Dokumenttyp–Deklaration und Dokumentinstanz.

Die XML–Deklaration beinhaltet Informationen über die XML–Version, die Zeichenkodierung (*character encoding*) und darüber, ob das Dokument externe Tag–Deklarationen benötigt, oder nicht.

Die Dokumenttyp–Deklaration beinhaltet eine Grammatik, die sogenannte *Document Type Definition (DTD)*, die die logische Struktur der mit Tags strukturierten Dokumentinstanz einschränkt und die Verwendung von vordefinierten Entities unterstützt. Es werden gültige (*valid*) und wohlgeformte (*well-formed*) Dokumente unterschieden. Für wohlgeformte Dokumente muß gelten, daß die Element–Tags ihrer Instanzen korrekt verschachtelt sein müssen. XML–Dokumente mit einer DTD sind gültige (*valid*) Dokumente, wenn sie wohlgeformt sind und der entsprechenden DTD genügen.

Im folgenden werden kurz die wesentlichen Konstrukte vorgestellt, die zum Aufbau einer DTD benötigt werden. Details werden an dieser Stelle weggelassen und bei der Transformation erläutert.

Das wesentliche Konstrukt ist das Element. Es deklariert eine Tag–Struktur mit Beginn– und Ende–Tag und die Struktur des Elementinhalts, der zwischen Beginn– und Ende–Tag stehen darf. Als Elementinhalt ist die Angabe von Elementnamen sowie Zeichensatzdaten (*character data*) (`#PCDATA`) erlaubt. Strukturiert wird der Elementinhalt durch eine beliebige Kombination von Sequenzen, Auswahlen (*choices*) und den Kardinalitäten (`*`, `+` und `?`). Zeichensatzdaten können entweder nur alleine als Elementinhalt stehen oder als beliebig häufige Auswahl mit Elementen. Die Elementdeklaration für die Speicherung eines Autors könnte z.B. wie folgt aussehen:

```
<!ELEMENT author
          ((firstname,middlename,lastname,suffix?) | organisation)
>
```

Ein Autor wäre so entweder eine Organisation oder eine Sequenz der drei Namensteile, optional (gekennzeichnet durch `?`) gefolgt von einem Suffix. In der Dokumentinstanz, in der ein Autor erfaßt werden soll, würde folgende Element–Tag–Struktur entstehen, bei der nur an den mit Punkten gekennzeichneten Stellen in Abhängigkeit von der Deklaration der entsprechenden Elemente weiterer Inhalt stehen kann (Die Zeilenumbrüche sind nicht zwingend):

```
<author>
<firstname>...</firstname>
<middlename>...</middlename>
<lastname>...</lastname>
<suffix>...</suffix>
</author>
```

Zu einem Element können Attribute in Form einer Attributliste deklariert werden. Die Attribute werden dann im Beginn–Tag in den Instanzen aufgeführt. Die Attributdeklarationen bestehen aus Attributnamen, Typ, der ein Aufzählungstyp sein kann, und einer Default–Deklaration. In der Default–Deklaration wird festgelegt, ob das Attribut konstant (`#FIXED`) ist, ob es immer angegeben werden muß (`#REQUIRED`) oder ob automatisch ein Wert generiert wird (`#IMPLIED`). Zusätzlich kann ein Defaultwert angegeben werden. Beispielsweise könnte man ein Attribut `Organisation` angeben, das konstant auf `HU Berlin` gesetzt wird, wenn die DTD nur für Veröffentlichungen der entsprechenden Universität gelten soll. `CDATA` ist der Built–In–Typ für Stringdaten:

```
<!ATTLIST author
          Organisation CDATA #FIXED "HU Berlin"
>
```

Neben diesen beiden Konstrukten, die die wesentlichen Strukturen festlegen, können Entities deklariert werden, mit denen man die DTDs strukturieren und in Hierarchien zerlegen kann. Eine Entity–Deklaration bindet ein Entity an einen Bezeichner, über den der Zugriff erfolgt. Entities dienen im allgemeinen der Textersetzung, die von einfacher Textsubstituierung bis zur Ersetzung von komplexen DTD–Strukturen reichen kann, wobei sich die Ersetzungsteile u.U. in externen Dateien befinden können. Damit können dann DTD–Hierarchien aufgebaut werden, bei denen auch ein Überdefinieren von Elementen möglich ist.

In XML werden im wesentlichen *parsed* und *unparsed* Entities unterschieden. Parsed Entities können in globale und Parameter–Entities untergliedert werden. Letzteres spielt für die Verwendung von Entities zur DTD–Strukturierung eine untergeordnete Rolle.

Folgendes Beispiel zeigt ein Parameter–Entity (spezifiziert durch `%`) mit einfacher Textersetzung:

```
<!ENTITY % list  "ol | ul | dl">
```

Hiermit kann die häufiger benötigte Elementinhalt–Spezifikation, die eine Alternative zwischen verschiedenen Listenarten verkörpert, an die Variable `list` gebunden werden. Des weiteren wird mit Hilfe von Entities auch die Einbindung weiterer Dateien ermöglicht, die keine XML–Struktur besitzen müssen:

```
<!ENTITY % article  SYSTEM "./article.tex"  NDATA latex>
```

Hierbei handelt es sich um eine Deklaration eines unparsed Entities, wobei durch `NDATA latex` der Notationstyp, mit dem dieses Entity bearbeitet werden soll, benannt wird. Dieser Typ muß in einer Notation–Deklaration deklariert werden, bei der eine System–Datei bestimmt wird, die die Daten des Entities bearbeiten kann.

```
<!NOTATION latex  SYSTEM "./latex2xml">
```

3 Konzeptionelle Modellierung von XML-Dokumenten

Nachdem im vorigen Abschnitt ein Überblick über XML gegeben wurde, sollen im weiteren die Konstrukte der UML-Klassendiagramme kurz vorgestellt werden, die für die Modellierung von XML-Dokumenten benötigt werden. Darauf wird dann auf die Transformation in XML-DTDs eingegangen. An manchen Stellen sind Erweiterungen von UML, wie sie die UML selbst auch vorsieht, notwendig. Die Notwendigkeit und die Form der Erweiterungen sowie deren Transformation werden an den entsprechenden Stellen erläutert. Mit der konzeptionellen Modellierung und der Transformation wird das wesentliche Fundament für die Integration im Rahmen von XML-Dokumenten gelegt. Eine ausführliche Version dieses Abschnitts ist in [CS99] zu finden.

3.1 Die für XML benötigten Konstrukte des UML-Datenmodells

Die Ausführungen dieses Abschnitts basieren auf [RJB99, EN99]. Die graphische Notation wird in diesem Abschnitt nur kurz beschrieben. Im nächsten Abschnitt ist sie bei den Beispielen zur Transformation zu sehen. Das Basis-Konstrukt sind die Klassen mit ihren Attributen und Methoden. Auf die Methoden werden wir hier nicht weiter eingehen, da der operationale Teil nicht Thema dieses Beitrags ist. Klassen werden graphisch durch ein Rechteck symbolisiert, bei dem der Name, die Attribute und die Methoden untereinander aufgeführt und durch einen waagerechten Strich optisch voneinander getrennt werden. Klassen können durch verschiedene Konstrukte miteinander verbunden sein. Einerseits kann dies durch Aggregationen und Kompositionen erfolgen. Dies sind Teil-von-Beziehungen. Die Komposition stellt eine spezielle Art der Aggregation dar, bei der die einzelnen Teile abhängig sind. Das heißt, daß sie nicht ohne ihr übergeordnetes Objekt existieren können. Graphisch wird beides durch eine Raute an der übergeordneten Klasse gekennzeichnet. Bei der Komposition ist sie ausgefüllt.

Des weiteren wird die Generalisierung, die häufig auch als Spezialisierung bezeichnet wird, als Modellierungskonstrukt angeboten. Sie wird durch ein Dreieck symbolisiert, dessen Spitze an der generelleren Klasse hängt und dessen Grundseite mit den spezielleren Klassen verbunden ist. An dieser Stelle besitzt die UML einen ihrer sogenannten *semantic variation points*, bei dem die Semantik des Konstrukts von der Zielsprache abhängt. Neben diesen speziellen Beziehungen gibt es mit der Assoziation eine allgemeine Beziehung. Sie wird durch eine einfache Verbindungslinie zwischen den Klassen gekennzeichnet.

Des weiteren können zwischen verschiedenen Konstrukten Abhängigkeiten dargestellt sowie zur Modularisierung Packages definiert werden, die einen Teilausschnitt einer Modellierung enthalten.

3.2 Transformation von UML in XML–DTDs

Klassen & Attribute Jede nicht–abstrakte Klasse kann als DTD–Element interpretiert werden, dessen Name der Klassenname ist. Für Digitale Bibliotheken beispielsweise kann ein Kapitel als Objekt modelliert werden, welches dann in ein Element transformiert wird, so daß jedes Kapitel mit einem Kapitel–Element–Tag umschlossen wird.

Üblicherweise trennt man in der konzeptionellen Modellierung die Daten, die gespeichert werden sollen, nicht von ergänzenden beschreibenden Daten (Meta–Daten). Da XML mit den Element–Tag–Attributen explizit die Möglichkeit bietet, Meta–Daten zu den eigentlichen Daten hinzuzufügen, soll diese Unterscheidung auch explizit in der Modellierung sichtbar werden. Für beide Arten von Daten wäre in UML, wie auch in anderen Datenmodellen, das Konstrukt des Attributs geeignet. Da ein Attribut aber auch als Aggregation eines weiteren Objekts darstellbar ist, bietet sich hier der Ansatzpunkt zur Unterscheidung. Meta–Daten sollen als Attribute und Daten als Aggregationen von Objekten modelliert werden. Dadurch ist es wiederum einfach möglich, zu den Daten weitere Meta–Daten zu ergänzen. Eine ausführlichere Behandlung der Aggregationen erfolgt nachdem die Transformation der Klassen mit den Attributen abgeschlossen ist.

Attribute einer Klasse werden in Attribute des Elements unter Erhaltung ihres Namens transformiert. Der Typ muß möglicherweise in einen XML–Typ transformiert werden (`string` $\longrightarrow$ `CDATA`). Beispielsweise kann ein Kapitel eine ID als Attribut haben, die zu Referenzierungszwecken im Element als Attribut verwendet werden kann. Besitzt ein Attribut einen Initialwert kann dieser für XML als Defaultwert übernommen werden. Nicht–optionale Attribute müssen mit `#REQUIRED` gekennzeichnet werden. Die Attribut–Multiplizität `[0..1]` wird in `#IMPLIED` übersetzt, da solche Attribute in XML nicht angegeben werden müssen und sie auch keinen Defaultwert erhalten. Klassenattribute, die für eine Klasse konstant sind, werden als `#FIXED` gesetzt. Sichtbarkeitsangaben werden in XML zur Zeit nicht berücksichtigt und finden somit bei der Transformation keine Beachtung.

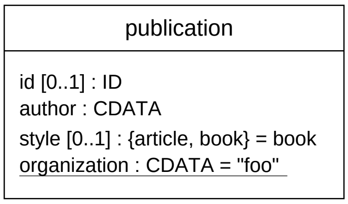
Abschließend müssen ein paar UML–Erweiterungen ergänzt werden. UML bietet als potentielle Erweiterung die Möglichkeit der Stereotypisierung an, mit der die Semantik beliebiger UML–Konstrukte beispielsweise für bestimmte Anwendungsgebiete genauer gefaßt bzw.

differenziert werden. Stereotype werden in französische Anführungszeichen (doppelte spitze Klammern) gesetzt.

Für die Modellierung von XML benötigt unser Ansatz drei neue Stereotype für Klassen: `<<entity>>`, `<<notation>>` und `<<content>>`. Der Stereotyp wird hier über dem Klassennamen innerhalb des Rechtecks geschrieben. Mit `<<entity>>` lassen sich einfache Entities ausdrücken, in dem der Entity–Wert als Kommentar in dem Attributblock ergänzt wird. Damit wird ermöglicht, daß die Textersetzung, die durch die Verwendung von Entities vorgenommen wird, als Abhängigkeit explizit gekennzeichnet werden kann. Alternativ kann der Entity–Name als Variable mit seinem Wert auch als freier Kommentar ohne graphisches Mittel in ein Package oder das globale Schema geschrieben werden.

Der Stereotyp `<<notation>>` wird aus dem bestehenden Stereotyp `<<utility>>` abgeleitet und drückt einen Notation–Typ aus. Das Programm, das die korrekte Zugehörigkeit von Dateien zu diesem Typ überprüfen soll, wird als Kommentar in dem Attributblock aufgeführt. Der Stereotyp `<<content>>` wird erst im nächsten Abschnitt über Aggregationen beschrieben, da er dafür erst benötigt wird.

In Abbildung 3.1 wird ein Beispiel gezeigt, in dem eine Klasse mit mehreren Attributen in ein Element mit seiner Attributliste transformiert wird. Die Attribute sind so gewählt, daß verschiedene Transformationsvarianten zum Tragen kommen.

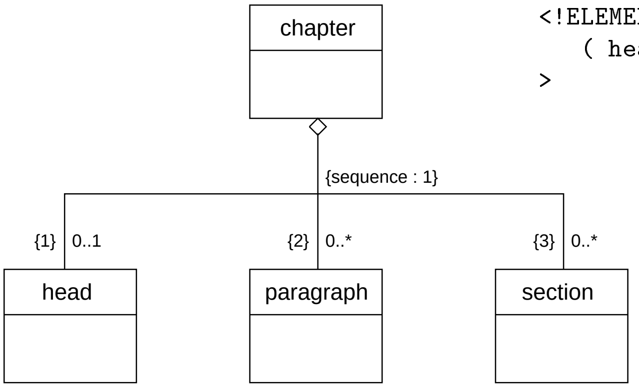



```
<!ELEMENT publication EMPTY>
<!ATTLIST publication
          id              ID     #IMPLIED
          author          CDATA  #REQUIRED
          style  (article|book)  book
          organization    CDATA  #FIXED "foo"
>
```

Figure 3.1 *Transformation einer Klasse mit Attributen*

Aggregationen Aggregationen als Teil-von-Beziehung lassen sich passend als Element–Inhalt übersetzen. Da sich im Element–Inhalt die Daten wiederfinden, wurde die Modellierungsmöglichkeit gewählt, Daten als Aggregationen zu modellieren. Jede Klasse, die Teil einer übergeordneten ist, findet sich in XML namentlich in der Spezifikation des Element–Inhalts wieder. Die Multiplizitäten sind in XML einfach über `?` (`[0..1]`), `+` (`[1..*]`) und `*` (`[0..*]`)auszudrücken.

Bei der konzeptionellen Modellierung existiert üblicherweise weder eine Reihenfolge der Teile, die aggregiert werden, noch die explizite Möglichkeit, beliebige Schachtelungen von Sequenzen und Auswahlen auszudrücken. Dieses wird jedoch für XML benötigt. Hier behelfen wir uns einer kleinen Modellierungsergänzung, indem an die entsprechenden Stellen Kommentare stehen, die für die Transformation bedeutungsvoll sind. Soll eine Aggregation eine Folge von Objekten (Sequenz) ausdrücken wird als Kommentar {`sequence`} und im Falle einer Auswahl {`choice`} an die Stelle geschrieben, an der sich die Verbindungslinie zu den untergeordneten Klassen auftrennt. Schachtelungen von Sequenzen und Auswahlen lassen sich so über einen entsprechenden Baum ausdrücken. Diese Vorgehensweise wird in [CS99] über eine zwischengeschaltete Objekte, die generalisiert werden, hergeleitet. Die Reihenfolge bei Sequenzen wird über eine Numerierung als Kommentare beschrieben, sofern sie nicht der intuitiven (von-oben-nach-unten und von-links-nach-rechts) entspricht.




```
<!ELEMENT chapter
   ( head?,  paragraph*, section* )
>
```

Figure 3.2 *Transformation einer Aggregation als Sequenz*

Beispielsweise kann wie in Abbildung 3.2 ein Kapitel so modelliert werden, daß es in folgender Reihenfolge aus optional einem Kopf, einer beliebigen Anzahl von Paragraphen und einer beliebigen Anzahl von Sektionen besteht. Die Aggregation wird dann in den entsprechenden Element–Inhalt übersetzt.

Generalisierung Für die Transformation der Generalisierung sind zwei wesentliche Implementierungsmöglichkeiten zu betrachten. Diese seien anhand der Transformation der Generalisierung in das Relationenmodell motiviert. In einem Fall werden sämtliche Klassen der Generalisierungshierarchie in einer Relation gespeichert, so daß jedes Attribut, das in der Hierarchie Verwendung findet, in dieser Relation steht und daß die von einer Klasse nicht benötigten Attribute auf NULL gesetzt sind. Die Zugehörigkeit zu einer Klasse innerhalb der Hierarchie wird über Indikator–Attribute angezeigt. Diese Variante ist insbesondere dann interessant, wenn sich spezielle Klassen auf Instanzenebene überlappen. Bei der anderen Variante wird jede Klasse in eine eigene Relation transformiert. Hierbei existieren zwei Möglichkeiten. Zum einen können alle Attribute, die vererbt werden, in die Relation mit aufgenommen werden, und zum anderen können die vererbten Attribute nur in der generelleren Klasse bleiben, so daß eine Referenz auf das entsprechende Objekt der generelleren Klasse benötigt wird. Diese Varianten lassen sich analog von Relationen auf Elemente übertragen.

Um die Möglichkeit zu bieten, diese Transformationen bei der Modellierung zu unterscheiden, werden zu den in UML üblichen Kommentaren bei der Generalisierung ({`disjoint`}, {`overlapping`}, {`complete`} und {`incomplete`}), die die Beziehungen der Objektmengen zueinander charakterisieren, folgende zwei hinzugefügt: {`discriminator`} und {`scattered`}. {`discriminator`} deutet darauf hin, daß Diskriminator–Attribute für die Zuordnung zu einer Klasse benötigt werden, wenn die Hierarchie in ein Element übersetzt wird, während {`scattered`} auf das verstreute Speichern der Daten verweist. Wenn keiner dieser beiden Kommentare steht, wird als Defaultfall der angenommen, daß die Daten eines Objekts in genau dem entsprechenden Element zu finden sind.

Des weiteren muß noch ein Spezialfall in der Generalisierung betrachtet werden: Abstrakte Klassen, die keine eigene Objektklasse sondern die Zusammenfassung von Klassen repräsentieren, werden nicht als Element übersetzt. Hier können die alternativen Klassen, die zusammengefaßt worden sind, als Alternativen in einem einfachen Entity transformiert werden, welches dann zur Schreibvereinfachung genutzt werden kann, wenn diese abstrakte Klasse als

Aggregat einer anderen Klasse dient. Dann wird bei der entsprechenden Übersetzung dieses Aggregats nur der Entity–Name anstelle aller Alternativen angegeben.

In diesem Abschnitt soll eine abstrakte Klasse, die aus anderen Klassen generalisiert wurde, als Beispiel dienen. Abbildung 3.3 zeigt dieses für das Zusammenfassen von verschiedenen Schriftarten. Das Transformationsergebnis ist ein Entity, welches die Namen der verschiedenen Elemente als Alternative auflistet. Dieses Entity wird dann im Elementinhalt verwendet, wenn die abstrakte Klasse als Aggregat dient. Ein weiteres Beispiel zur Generalisierung folgt im Zusammenhang mit den Abhängigkeiten.

```
<!ENTITY  % richtext "em | strong | tt | ... " >

<!ELEMENT em      ... >
<!ATTLIST em      typ CDATA #FIXED "emphasize" >

<!ELEMENT strong  ... >
<!ATTLIST strong  typ CDATA #FIXED "strong" >

<!ELEMENT tt      ... >
<!ATTLIST tt      typ CDATA #FIXED "type writer" >
```

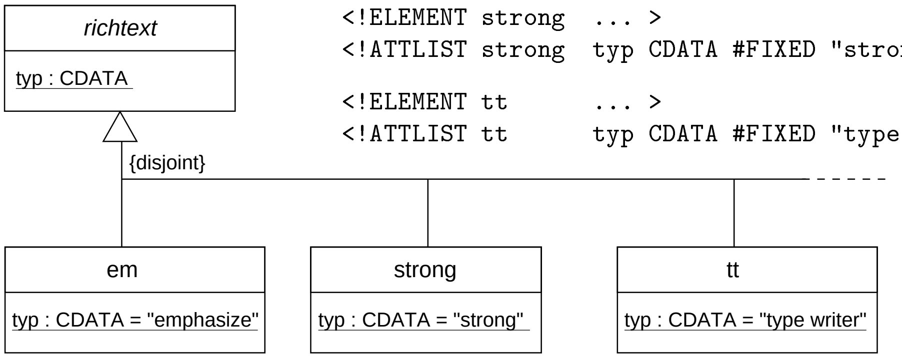


Figure 3.3 *Abstract class modeling*

Assoziationen Die allgemeinen Beziehungen, die zwischen Objektklassen definiert werden können, werden sich in Zukunft mit den Mechanismen von XLink [Wor98c] und XPointer [Wor98d] in XML ausdrücken lassen. Da zur Zeit nur die Working Drafts vom März 1998 publik sind, wollen und können wir über die genaue Transformation noch keine Aussagen treffen. Dies wird Teil einer späteren Veröffentlichung sein. An dieser Stelle sei nur soviel gesagt, daß XLink die Möglichkeit bietet, auf andere Dokumente zu verweisen, wobei über HTML hinausgehende Link–Typen definiert sein werden, so daß z.B. bijekte Links definierbar sind. XPointer hingegen ergänzt XLink durch die Möglichkeit genaue Stellen in Dokumenten anzugeben, wobei im Gegensatz zu HTML nicht eine Sprungadresse angegeben werden wird sondern einen Bereich, der aus einem oder mehreren Element–Tags oder aus einem bestimmten Textabschnitt bestehen kann. Zu dem sind über das Zusammenspiel von XLink und XPointer nicht nur Sprünge wie in HTML sondern auch das Hereinladen der bestimmten Bereiche möglich.

Abhängigkeiten Im Rahmen der Generalisierung werden sowohl Aggregationen als auch Attribute vererbt. Attribute stellen, wie oben beschrieben, kein Problem dar. Die Vererbung der Aggregation weist hingegen bei dre Transformation ähnliche Probleme auf wie die Aggregation alleine. Für XML ist die Gestaltung von Auswahlen, Sequenzen und deren Reihenfolge von Bedeutung. Intuitiv wird bei der Transformation der Elementinhalt der vererbten Aggregation in einer Sequenz vor den Elementinhalt der Aggregation des erbenden Elements geschrieben.

Soll die Modellierung der Vererbung eine Auswahl oder eine andere Reihenfolge ausdrücken, kann dies als Kommentar an dem Abhängigkeitssymbol zwischen den beiden Aggregationen gekennzeichnet werden. das Symbol ist ein Pfeil mit gestrichelter Linie, wobei der Pfeil auf die zu vererbende Aggregation zeigt.

Durch eine Abhängigkeit kann des weiteren die Zugehörigkeit einer Entity–Stereotyp–Klasse bzw. eines Attributs zu einer Notation–Stereotyp–Klasse verdeutlicht werden. Im ersten Fall erhält man ein einfaches parsed Entity, das einer Datei entspricht, die dem entsprechenden Notation–Typ angehört. Im zweiten Fall erhält man ein Attribut, dessen Wert von dem Notation–Typ ist.

In Abbildung 3.4 ist beispielsweise die Abhängigkeit von Aggregationen bei der Generalisierung zu sehen. Durch den Kommentar {`choice`} an der Abhängigkeit wird ausgedrückt, daß im Gegensatz zur üblichen Vererbung die beiden Aggregationen im Elementinhalt von `B` nur als Alternativen zur Verfügung stehen.

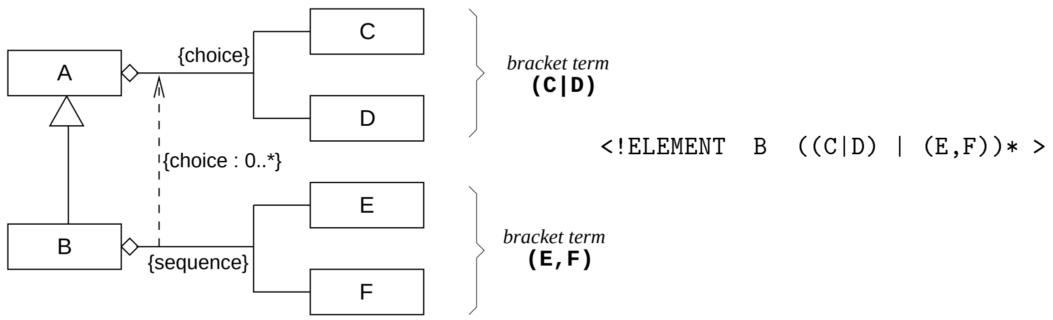


Figure 3.4 *Vererbung der Aggregation mit einer Abhängigkeit*

Packages Packages dienen der Strukturierung und Modularisierung. Dieses wird in XML durch die Verbindung von komplexen Entities und von der Aufteilung in mehrere Dateien übernommen. Ein Package, das nur Objektklassen, Klassen des Stereotyps `<<entity>>` und Beziehungen dazwischen enthält, wird in ein komplexes Entity übersetzt, das die entsprechenden Elemente, Attributlisten und einfachen Entities beinhaltet.

Da für XML keinen Hiding–Mechanismus existiert, wird jedes XML–Konstrukt automatisch exportiert, so daß auf jedes Konstrukt von jeder Stelle zugegriffen werden kann. Nebenbei gesagt, ist dies für Dokumente in gewissem Rahmen durchaus sinnvoll. Allerdings wären auch Fälle denkbar, wo man z.B. den Elementinhalt `ANY`, der eine beliebige Mischung aus Element–Tags und Charakter–Daten erlaubt, nur für eine lokale Menge von Element–Tags verwenden möchte. Da dies aber (noch) nicht vorgesehen ist, ist der Hiding–Mechanismus für die Modellierung uninteressant. Dementsprechend können wir ohne Einschränkung erlauben, daß es Beziehungen über Package–Grenzen gibt. Zur Vereinfachung dieser Modellierung führen wir die virtuelle Klasse als Modellierungskonstrukt ein, die durch ein gestricheltes Rechteck symbolisiert wird. Dem Klassenname muß der Packagenamen durch einen Punkt getrennt vorangestellt werden. Hierzu sei noch erwähnt, daß Klassennamen aus besagten Gründen innerhalb einer gesamten DTD immer eindeutig sein müssen.

Die Generalisierung von Packages ist eine sehr gute Methode, Hierarchien von Packages aufzubauen, die in gleichartige Hierarchien von DTD–Dateien umgesetzt werden können. Hiermit ist auch das Überladen von Klassen, Aggregationen und Attributen möglich, die entsprechend in ein Überladen von Elementen, Elementinhalten und Attributen transformiert wird. Das Problem der multiplen Vererbung wird durch die explizite Nummerierung in Kommentaren gelöst.

Die Modellierung des Imports von anderen Packages, läßt sich auf XML–Ebene nicht umsetzen. Zum einen fehlen Hiding–Mechanismen und zum anderen ist ein selektiver Import in XML zwar möglich aber nicht sinnvoll.

Alle Transformationsteile, die in diesem Abschnitt beschrieben worden sind, werden in den Tabellen 1 und 2 in Anhang A zusammengefaßt aufgeführt.

4 Integration

Nachdem mit der konzeptionellen Modellierung von XML-Dokumenten die Grundlagen gelegt worden sind, ist es jetzt ohne Schwierigkeiten möglich, die bekannten Ergebnisse zur Schema-Integration für den Bereich der XML-Dokumente anzupassen. Im folgenden wird gezeigt, wie dieses für die Integration von verschiedenen XML-Dokumenten und für die Integration von Datenbanken, deren Daten via XML ausgetauscht werden sollen, erfolgen kann.

4.1 Integration von XML-Dokumenten

Der übliche Fall, der für die Integration von XML-Dokumenten zu erwarten ist, wird der sein, bei dem in verschiedenen Datenquellen, die über das Internet verstreut liegen können, Dokumente liegen, die z.T. anhand derselben und z.T. anhand verschiedener DTDs aufgebaut sind. Der Fall, daß pro Datenquelle nur eine DTD als Basis dient, ist nur ein angenehmer Spezialfall.

Mit Hilfe der Ergebnisse aus den vorherigen Abschnitten lassen sich alle DTDs konzeptionell mit UML modellieren. Als Nebeneffekt kann man dabei eine Verbesserung der DTD-Hierarchie einer Datenquelle erhalten, wenn so einige wiederverwendbare DTD-Teile identifiziert werden können. Diese so gewonnene objektorientierte Modellierung kann man dann verwenden, um mit den bekannten Methoden der Schema-Integration, wie sie z.B. in [Con97] zusammengefaßt sind, selbige inklusive der Anfragetransformation vorzunehmen. Abbildung 1 stellt diesen Sachverhalt dar.

Die Integration wird jedoch einige Probleme aufwerfen. Neben der Tatsache, daß mit keiner bisher existierenden Methode alle Integrationsprobleme gelöst werden kann, werden im folgenden einige identifizierte potentielle Problembereiche aufgeführt, die sich aus der Basis XML ergeben. An einigen Stellen mußte UML erweitert werden, um die Modellierungen XML-konformer zu gestalten. Hier stellt sich die Frage, inwieweit diese Erweiterungen bzgl. der Schema-Integrationsmethoden anpaßbar sind. Da die Erweiterungen nicht schwerwiegend sind, nehmen wir an, daß dies ohne größeren Aufwand möglich ist. Reihenfolgen beispielsweise spielen auf der Ebene des integrierten Schemas keine Rolle, da diese die Anfrageergebnisse nicht verändern. Dies wird erst interessant für die Anfragetransformation zu den lokalen Schemata, falls diese Anfragen direkt auf den Dokumenten ausgeführt werden sollen.

Ein großes Problem, welches aber mehr XML-inhärent ist, besteht in der Frage, welche Daten man als Daten und welche man als Meta–Daten modelliert. Da hier die Entscheidungen zwischen verschiedenen Modellierungen garantiert auseinander gehen, muß dieser Fall unbedingt

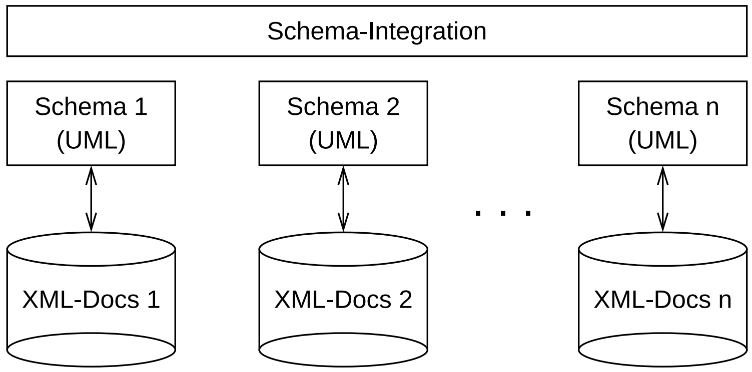


Abbildung 1: Integration von XML-Dokumenten

betrachtet werden. Eine Lösungsmöglichkeit könnte in Analogie zu Schema-SQL [LSS96] entwickelt werden. Bei Schema-SQL als Anfragesprache wird der Übergang zwischen den Metadaten des Schemas und den Daten fließend gemacht, so daß beides beliebig miteinander kombinierbar ist.

Der operationelle Teil wird noch Schwierigkeiten aufwerfen, da noch keine Standards für Anfragesprachen existiert. Neben den Bestrebungen der Gruppe um Andreas Heuer eine Anfragesprache namens IRQL [HP99] zu entwickeln, die die Fähigkeiten von Information-Retrieval-Sprachen mit denen von normalen Datenbank-Anfragesprachen verknüpft, wird vom W3C die XML-QL [Wor98e] entwickelt. In Ergänzung zu diesem Beitrag und dem zugrundeliegenden Bericht [CS99] soll als Vorstufe zur Lösung dieser Probleme die konzeptionelle Modellierung um den operationellen Teil erweitert werden.

Auf die zusätzlich entstehenden Probleme, wenn man bei der Integration nicht nur XML-Dokumente sondern auch „normale“ Datenbanken betrachten muß, werden wir im nächsten Abschnitt noch etwas näher eingehen, da dort die Verbindung zwischen Datenbanken und XML-Dokumenten betrachtet wird.

4.2 Integration mit Hilfe von XML

Die Datenbankintegration, bei der die Daten per XML ausgetauscht werden, weil die beteiligten Datenbanken z.B. über das Internet verstreut liegen, läßt sich mit Hilfe der konzeptionellen Modellierung der XML-DTDs analog behandeln. Durch die konzeptionelle Modellierung der DTDs und der Datenbanken existieren mehrere Schemata, bei denen zwischen den einzelnen Datenbank-Schemata und den Schemata zu den DTDs Schema-Integrationen vorgenommen werden müssen. Abbildung 2 verdeutlicht diesen Sachverhalt.

Vermutlich werden hier durch die unterschiedlichen Herangehensweisen bei der Modellierung von Datenbanken und XML-Dokumenten an einzelnen Stellen Schwierigkeiten auftreten. Nicht zuletzt, weil einmal UML mit den Erweiterungen für XML und einmal ohne verwendet wird, könnten sich Inkonsistenzen ergeben. Da die Erweiterungen jedoch möglichst gering gehalten wurden und sich größtenteils auf Aspekte beziehen, die in Datenbanken so nicht darstellbar sind, werden diese Schemata bei der Integration vermutlich eine Ergänzung des Datenbankschemas darstellen.

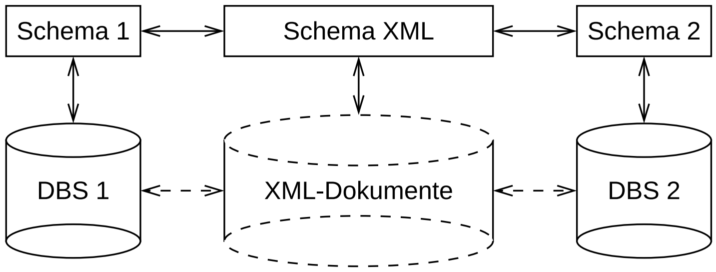


Abbildung 2: Integration mit XML-Dokumenten

5 Abschließende Bemerkungen

In diesem Beitrag haben wir gezeigt, daß sich die Integration von XML-Dokumenten unter Zuhilfenahme der konzeptionellen Modellierung fast nahtlos in die Datenbankintegration einfügt. Dieses gilt sowohl für die Integration von verschiedenen XML–Dokumenten als auch von Datenbanken, die via XML ihre Daten austauschen sowie von Datenbanken und XML-Dokumenten. Dafür wurde auf der Basis von UML ein Vorschlag entwickelt, wie XML-Dokumente konzeptionell modelliert werden. Zur Abrundung wurde zuvor kurz in XML eingeführt sowie der Bezug zu semistrukturierten Daten hergestellt.

Bezüglich der Integration auf Basis der konzeptionellen Modellierung wurden einige potentielle Problemfelder herauskristallisiert, die sich vermutlich ohne größeren Aufwand in den Griff bekommen lassen. Dies werden genauere Untersuchungen zeigen, die noch ausstehen.

Ein wesentlicher Aspekt, der ebenfalls in näherer Zukunft behandelt werden soll, ist der operationelle Teil der konzeptionellen Modellierung von XML. Hier werden mehrere Diagrammarten von UML Eingang finden. Dies werden zumindest die Interaktionsdiagramme, die sich in Sequenz- und Kollaborationsdiagramme aufteilen, die Zustands- und die Aktivitätsdiagramme sein. Damit soll ermöglicht werden, typische Anfragen an eine Menge von XML-Dokumenten zu modellieren, die einerseits Aspekte von normalen Datenbankanfragesprachen und andererseits von Information-Retrieval- Sprachen besitzen.

Insgesammt läßt sich abschließend zusammenfassen, daß mit unserem Ansatz der konzeptionellen Modellierung von XML-Dokumenten einerseits sich die DTDs und ihre Hierarchien fehlerfreier und besser gestalten lassen und andererseits die Möglichkeit entstanden ist, die bestehenden Ergebnisse der Schema–Integration auf diesen Bereich zu adaptieren.

Literatur

[Abi97] S. Abiteboul. Querying semi-structured data. In F.N. Afrati and P. Kolaitis, editors, *Database Theory - ICDT 97, 6th international Conference, LNCS 1186*, pages 1–18, Delphi, Greece, 1997. Springer Verlag.

[Con97] Stefan Conrad. *Föderierte Datenbanksysteme — Konzepte der Datenintegration.* Springer-Verlag, Berlin, 1997. Habilitationsschrift.

[CS99] R. Conrad and D. Scheffner. XML Conceptual Modeling using UML. Technical report, Institut für Informatik, HU Berlin, 1999. Erscheint demnächst.

[Dit98] K. Dittrich. Von der Datenboutique zum Informationskaufhaus: Datenbanktechnologie für globale Anfragedienste. Vortragsmanuskript, Kolloquiumsvortrag, TU Berlin, 1998, 1998.

[EN99] S. Eckstein and K. Naumann. Konzeptioneller Entwurf mit der Unified Modeling Language. Technical Report Informatik Berichte 99-03, Abteilung Informationssysteme, TU Braunschweig, 1999.

[HP99] A. Heuer and D. Priebe. IRQL – yet another language for querying semistructured data? Technical Report Preprint CS-01-99, Universität Rostock, Fachbereich Informatik, 1999.

[LSS96] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian. Schemasql: A language for interoperability in relational multidatabase systems. In T.M. Vijayaraman, A.P. Buchmann, C. Mohan, and N.L. Sarda, editors, *Proc. of the 22nd Int. Conf. on Very Large Data Bases (VLDB 96)*. Morgan Kaufmann Publishers, 1996.

[Mic95] Michel Goosens, and Janne Saarela. A Practical Introduction to SGML. In *TUGboat*, volume 16(3), 1995.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

[SH99] Gunter Saake and Andreas Heuer. *Datenbanken: Implementierungstechniken*. MITP-Verlag, Bonn, 1999.

[Wor98a] World Wide Web Consortium. Extensible Markup Language (XML), Version 1.0. Technical Report REC-xml-19980210, W3C, February 1998.

[Wor98b] World Wide Web Consortium. Schema for Object-oriented XML. Technical Report NOTE-SOX-19980930, W3C, September 1998.

[Wor98c] World Wide Web Consortium. XML Linking Language (XLink). Technical Report WD-xlink-19980303, W3C, March 1998.

[Wor98d] World Wide Web Consortium. XML Pointer Language (XPointer). Technical Report WD-xptr-19980303, W3C, March 1998.

[Wor98e] World Wide Web Consortium. XML-QL: A Query Language for XML. Technical Report NOTE-xml-ql-19980819, W3C, August 1998.

A Die Transformation im Überblick

Im folgenden werden tabellarisch die UML-Konstrukte und deren Transformation in XML aufgeführt.

UML	XML
Klasse	Element-Typ-Deklaration + Attribut-Listen-Deklaration
Klassenname	Element-Typ-Name + Attribut-Listen-Name
Klassen-Stereotype	
$\odot$ <<entity>>	einfaches Entity (Text in Kommentar)
$\odot$ <<notation>>	Notation-Deklaration
$\odot$ <<content>>	Element-Inhalt (ANY und #PCDATA)
Attribute einer Klasse	Deklaration in Attributliste
Attributname	Attributname
Attributtyp	Attributtyp
Aufzählungstyp	Attribut-Enumeration mit Token
Generelle Typen	spezifische Attributtypen
Initialwert	Defaultwert
Attribut-Multiplizität [0..1]	#IMPLIED-Attribut
Klassen-Attribut	#FIXED-Attribut
ohne Kennzeichnung	#REQUIRED-Attribut
Aggregation	Element-Inhalt
Multiplizität	Kardinalität im Element-Inhalt
1 (Default-Wert)	(Default-Wert)
0..1	?
1..*	+
0..*	*
$\odot$ Kommentar {sequence}	Sequenz
$\odot$ Kommentar {choice}	Choice
$\odot$ Nummern in Kommentar	Reihenfolge

Tabelle 1: Zusammenfassung der Transformationen ($\odot$ bei UML-Erweiterungen) Teil 1

UML	XML
Generalisierung von Klassen	Spezialisierung von Elementen
$\odot$ Kommentar {`discriminator`}	Ein Element für alle Klassen
$\odot$ Kommentar {`scattered`}	Elemente „normal" transformieren + IDREF
ohne Kommentar	jedes Element bekommt alles vererbt
Vererbung der Attribute	Zusammenfassung der Element-Attribute
Vererbung der Aggregation	Kombination der Element-Inhalte
abstrakte Klasse + Komm {`disjoint`}	Alternativen in einfachem (Parameter-) Entity
Assoziationen	XLink & XPointer
Abhängigkeiten	
Aggregationen bei Generalisierung	Objekt in genau dem Element
$\odot$ Kommentar {`choice`}, {`sequence`} oder Nummer	s. Aggregation (sequence ist Default)
zw. Stereotypen Entity $\longrightarrow$ Notation	Entity $\longrightarrow$ Notation
zw. Stereotypen Attribut $\longrightarrow$ Notation	Attribut $\longrightarrow$ Notation
Packages	komplexes Entity
Package-Name	Entity-Name
Schachtelung	Modularisierung
Generalisierung	DTD-Hierarchien
Multiple Vererbung	
$\odot$ Nummer-Kommentare	Reihenfolge
Package für Datei	parsed Entity (Instanz)
Variable im Freitext	einfaches Entity
Schema-Name	Dokumenttyp-Name

Tabelle 2: Zusammenfassung der Transformationen ($\odot$ bei UML-Erweiterungen) Teil 2