Reihe Informatik
10 / 1999

Variationsmöglichkeiten bei der Transformation von UML-Basiskonzepten in Dokumenttypdefinitionen

Dunja Winkens Guido Moerkotte

Variationsmöglichkeiten bei der Transformation von UML-Basiskonzepten in Dokumenttypdefinitionen

Dunja Winkens
Lehrstuhl für Praktische Informatik I
Universität Mannheim
68131 Mannheim
Germany

Guido Moerkotte
Lehrstuhl für Praktische Informatik III
Universität Mannheim
68131 Mannheim
Germany

6. Dezember 1999

Zusammenfassung

Wir betrachten verschiedene Transformationsmöglichkeiten eines in UML gegebenen konzeptuellen Schemas in eine XML-Dokumenttypdefinition. Dabei konzentrieren wir uns auf die Basiskonstrukte, die in UML zur Verfügung stehen. Die verschiedenen Transformationsmöglichkeiten werden in Hinsicht auf den Erhalt der Semantik des konzeptuellen Schemas bewertet.

1 Einleitung und Motivation

Ursprünglich war die *Extended markup language* (XML [GP99, BPSM98]) als Alternative zu HTML oder sogar als dessen Nachfolger zur Beschreibung von Dokumenten gedacht. Der zugrunde liegende Kerngedanke von XML ist die Erweiterbarkeit um neue Markierungen — eine Eigenschaft, die HTML nicht bietet. Diese neugewonnene Flexibilität hat dazu geführt, XML nicht nur für die Beschreibung von Dokumenten zu verwenden, sondern auch für die Darstellung von Daten. Inzwischen liegt hier die eigentliche Bedeutung von XML.

Mit den neuen E-Anwendungsgebieten wie *electronic business*, *electronic commerce*, *electronic services* und so weiter gewinnt die Nutzung von XML zur Datendarstellung zunehmend an Dynamik. Die angeführten Anwendungsgebiete setzen oft die Möglichkeit des Austauschs von Daten über das Internet voraus. Die vormals eingeführten speziellen Formate für den elektronischen Datenaustausch (*electronic data interchange* (EDI) [GP99]) erwiesen sich als zu schwerfällig und auch zu teuer. Daraus resultiert ein geringerer Grad der Verbreitung, was wiederum die Effektivität des Einsatzes von EDI verringert. Mit XML steht hier nun

eine alternative Möglichkeit für den elektronischen Datenaustausch zur Verfügung. Für ein gegebenes Geschäftsfeld müssen die benötigten Daten ermittelt und ihre Darstellung in XML mittels der Dokumenttypdefinition fixiert werden. Verschiedene Organisationen[1] setzen sich mit der Normierung von Dokumenttypdefinitionen auseinander, um einheitliche Darstellungen von Geschäftsinformationen für verschiedene Geschäftsfelder mittels XML-Dokumenten zu erreichen.

Im Datenbankbereich kennt man das Problem der Modellierung von Daten bereits seit geraumer Zeit. Das weitverbreiteste Instrument der konzeptuellen, vom Datenmodell unabhängigen Modellierung von Daten ist das Entity-Relationship-Diagramm [Teo99]. Eine modernere Variante ist UML [RJB99]. Modellierungsmethoden unterstützen die Erstellung von ER- oder UML-Diagrammen. Solche Modellierungsmethoden sind für die Entwicklung von XML-Dokumenttypdefinitionen nicht bekannt und auch nicht sinnvoll. Letzteres kann man daran erkennen, dass Modellierungsmethoden für logische Datenbankmodelle (zum Beispiel das relationale Modell) keinerlei praktische Relevanz haben. Die übliche Vorgehensweise führt immer vom konzeptuellen Modell über das logische Modell zum physischen Modell [Teo99]. Dabei helfen Standardverfahren bei der Umsetzung von einem Modell in das andere. Beispielsweise lässt sich die Umsetzung von ER-Diagrammen in Relationenschemata mittels einfach zu befolgender Regeln beschreiben [Teo99].

Wir sehen eine für die Darstellung von Daten entwickelte XML-Dokumenttypdefinition (DTD) als ein logisches Modell an. Daher schlagen wir vor, zur Erstellung einer solchen DTD zunächst weiterhin ein konzeptuelles Modell mittels einschlägiger Methoden zu entwickeln. Für die Transformation des konzeptuellen Modells in das logische Modell, also eine DTD, gibt es verschiedene Alternativen. Im Folgenden untersuchen wir diese Alternativen für verschiedene Basiskonstrukte von UML. Wir beschäftigen uns zunächst mit Objektidentität, dann mit der Umsetzung von Klassen und Attributen. Einen breiten Raum nimmt die Behandlung der Umsetzung von Beziehungen in Anspruch, da diese in recht vielfältigen semantischen Ausprägungen existieren. Abschliessend diskutieren wir verschiedene Umsetzungsmöglichkeiten von Generalisierungshierarchien. Wir evaluieren die angesprochenen Transformationsmöglichkeiten mit Blick auf den Erhalt der im konzeptuellen Schema enthaltenen Semantik.

In den Kapiteln 2 und 3 wiederholen wir zunächst die Grundlagen von UML bzw. XML. Kapitel 4 enthält den Kern des Papiers. Kapitel 5 enthält Zusammenfassung und Ausblick.

2 UML

Die Unified Modeling Language (UML) ist eine Sprache zur Beschreibung komplexer Software-Systeme sowie zur Geschäftsprozessmodellierung. In diesem Abschnitt beziehen wir uns auf die UML 1.0, wie sie von Booch, Jacobsen und Rumbaugh in [RJB99] dargestellt wird.

Ein in der UML modelliertes System beinhaltet Informationen über seine statische sowie seine dynamische Struktur. Die statische Struktur definiert die Art von Objekten, die im System relevant sind, sowie ihre Beziehungen untereinander. Das dynamische Verhalten beinhaltet die Veränderung der Objekte und ihrer Beziehungen untereinander sowie die Kommunikation zwischen den Objekten während der Lebenszeit eines Systems. Bei einer statischen Betrachtungsweise, wie sie durch die Modellierung in XML vorgegeben ist, kann das Verhalten von Objekten nicht abgebildet werden. Wir konzentrieren uns im Rahmen die-

[1] Bspw. Oasis (http://www.oasis-open.org) und BizzTalk (http://www.BizzTalk.org)

ser Arbeit deshalb auf die folgenden statischen Kostrukte der UML zur Modellierung eines Systems, die an einem Beispiel (vgl. Abbildung 1) erläutert werden:

1. Klassen und Objekte
2. Attribute
3. Assoziation
4. Generalisierung

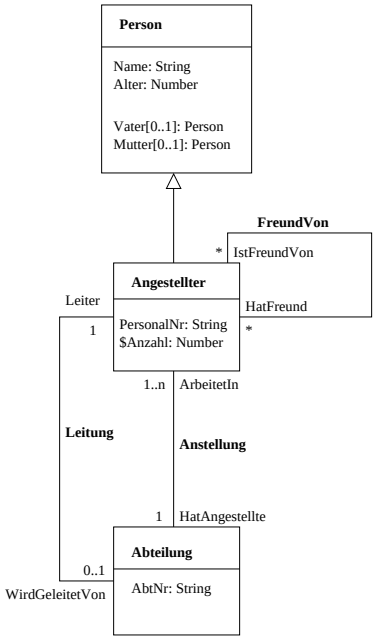


Abbildung 1: Statische Sicht eines UML-Modells für eine Firma

2.1 Klassen und Objekte

Eine *Klasse* ist ein Konzept, das zur Beschreibung einer Menge von Objekten mit ähnlichen Eigenschaften dient. In der *Klassendefinition* werden die Struktur, das Verhalten sowie die Beziehungen von Objekten der Klasse festgelegt. Ein Objekt besitzt eine *Identität*, die sie von anderen Objekten unterscheidet. In unserem Beispielmodell (vgl. Abbildung 1) werden die Klassen `Person`, `Angestellter` und `Abteilung` definiert.

2.2 Attribute

Bei der Definition einer Klasse werden die zugehörigen Attribute festgelegt. Im Rahmen dieser Arbeit nennen wir solche Attribute *Objektattribute*, um sie von den in XML definierten Elementattributen (vgl. Abschnitt 3.2) zu unterscheiden.

Ein *Objektattribut* hat einen Namen und einen Wert. In Objekten einer Klasse ist jedes Objektattribut jeweils mit einem Wert des entsprechenden Typs belegt. Es kann sich hierbei um *atomare Typen* oder um *komplexe Typen* handeln.

Objektattribute atomaren Typs haben keine Identität, d.h. sie können nur innerhalb von Objekten existieren. Weiterhin haben sie einen *Wertebereich*, aus dem sie Werte annehmen können. Es gibt Wertebereiche für *einfache Typen* (number, string, time) sowie *Aufzählungstypen* (enumeration).

In unserem Beispielmodell (vgl. Abbildung 1) werden für die Klasse `Person` die atomaren Objektattribute `Name` mit dem Typ `String` und `Alter` mit dem Typ `Number` definiert.

Komplexe Objektattribute sind vom Typ *Klasse*. Im Beispielmodell (vgl. Abbildung 1) werden für die Klasse `Person` die komplexen Objektattribute `Vater` und `Mutter` vom Typ `Person` definiert.

Die *Kardinalität (multiplicity)* von Objektattributen kennzeichnet ihre Wertigkeit. Ein Objektattribut ist genau einem Klassenelement zugeordnet und kann keinen, einen oder eine Menge von Werten annehmen. Es kann auch eine Anzahl möglicher Werte vorgegeben werden.

Im Beispielmodell (vgl. Abbildung 1) wird für das Attribut `Vater`, der Klasse `Person` die Kardinalität [0..1] eingeführt, d.h. eine Person kann einen oder keinen Vater haben.

Klassenattribute sind der Klasse zugeordnet, d.h. es gibt genau einen Attributwert für alle Objekte der Klasse. Im Beispielmodell (vgl. Abbildung 1) wird für die Klasse `Angestellter` das Klassenattribut `Anzahl` definiert.

2.3 Assoziation

Eine *Assoziation* beschreibt eine Beziehung zwischen zwei oder mehr Objekten. Dies umfasst die Informationen, die über eine solche Verbindung benötigt werden. Dazu gehören neben den beteiligten Klassen auch etwaige Attribute der Beziehung, die wir hier *Assoziationsattribute* oder *Beziehungsattribute* nennen.

Meist findet eine Assoziation zwischen Objekten zweier verschiedener Klassen statt (binär). Falls verschiedene Objekte derselben Klasse verbunden werden, spricht man von einer *rekursiven Beziehung*. Sind drei, bzw. n Klassen an der Beziehung beteiligt, so spricht man von *ternären*, bzw. *n-wertigen* Beziehungen.

Im Beispielmodell (vgl. Abbildung 1) wird die rekursive Beziehung `FreundVon` definiert. Die Beziehungen `Leitung` und `Anstellung` sind binäre Beziehungen.

Ein *Link* ist eine konkrete Instanz einer Assoziation, d.h. die Verbindung von Objekten der durch die Assoziation bestimmten Klassen. Eine solche Instanz verbindet im allgemeinen verschiedene Objekte, sie kann jedoch auch ein Objekt mit sich selbst verbinden.

Assoziationen haben einen Namen und gegebenenfalls Assoziationsattribute. Falls eine Assoziation Attribute hat, so ist sie wiederum eine Klasse, eine sogenannte *Assoziationsklasse*.

Die an einer Assoziation beteiligten Klassen nennt man *Assoziationsziele*. Sie haben üblicherweise einen Namen (*Rolle*) sowie eine Kardinalität. Die *Kardinalität* beschreibt, wieviele Objekte der entsprechenden Klasse eine Beziehung zu einem Objekt der gegenüberliegenden Klasse eingehen können.

Im Beispielmodell (vgl. Abbildung 1) hat die Assoziation `Anstellung` die Assoziationsziele `Angestellter` bzw. `Abteilung`. Sie nehmen die Rollen `Leiter` bzw. `WirdGeleitetVon` ein.

2.4 Generalisierung

Eine *Geralisierungsbeziehung* besteht zwischen einer *allgemeinen Klasse (Superklasse)* und einer *spezialisierten Klasse (Subklasse)*. Die spezialisierte Klasse enthält alle Informationen der allgemeinen Klasse (Objektattribute, Verhalten und Beziehungen) und erweitert sie um zusätzliche Informationen. Man spricht deshalb in diesem Zusammenhang auch von *Vererbung*.

Im Beispielmodell (vgl. Abbildung 1) ist ein `Angestellter` eine Spezialisierung einer `Person`. Er erbt alle Objektattribute von der Klasse `Person` (`Name` und `Alter`) und hat zusätzlich das Objektattribut `Personalnummer`.

3 XML

XML [BPSM98] ist eine Markup-Sprache, bei der einzelne XML-Elemente von Anfang- und Ende-Markierungen umschlossen sind. Ein XML-Dokument ist ein Baum aus solchen Elementen. Insbesondere hat es ein eindeutiges Wurzelelement und alle Elemente müssen korrekt, d.h. ohne Überschneidungen, ineinander geschachtelt sein. Erfüllt ein XML-Dokument diese Bedingung, so heißt es *wohlgeformt*.

Zusätzlich legen die *Dokumenttypdefinitionen (DTDs)* eine Grammatik fest, der das XML-Dokument entsprechen muss. Entspricht ein wohlgeformtes Dokument den angegebenen DTDs und erfüllt es einige weitere Bedingungen (s.u.), so heißt es gültig. Wohlgeformtheit und Gültigkeit werden von einem validierenden Parser überprüft.

Für das vorliegende Papier benötigen wir die XML-Konstrukte *Elementtyp*, *Element*, *Elementattribut* und *Entity*. Sie werden in den weiteren Abschnitten dieses Kapitels erläutert.

3.1 Elementtypen und Elemente

In einer DTD kann durch Elementtyp- und Attributlisten-Deklarationen (vgl. Abschnitt 3.2) die Element-Struktur eines gültigen XML-Dokuments beschränkt werden. Eine *Elementtyp-Deklaration* hat die folgende Form:

```
<!ELEMENT Name Inhaltsmodell>
```

Hier wird der Name und das sogenannte *Inhaltsmodell* festgelegt. Ein Elementtyp darf nicht mehr als einmal deklariert werden.

Das *Inhaltsmodell* legt fest, welche Elementypen als Kinder des Elements zugelassen sind. Es sind folgende Inhaltsmodelle zugelassen:

1. *Leer (EMPTY)*: Es können nur Zeichendaten, d.h. keine Elemente, in das deklarierte Element eingebettet werden. Eine Beispieldeklaration eines leeren Elements ist `<!ELEMENT Person EMPTY>`. Ein leeres Element wird im Dokument durch `<Person/>` ausgedrückt.

2. *Element-Inhalt (children)*: Es sind ausschliesslich Kindelemente, d.h. keine Zeichendaten zugelassen. Neben den Elementnamen der zugelassenen Kindelemente kann man weitere Eigenschaften in Form eines regulären Ausdrucks spezifizieren. Dabei bezeichnet ',' eine Sequenz, '|' eine Alternative, '*' eine beliebige Wiederholung, '+' eine mindestens einmalige Wiederholung und '?' ein optionales Element. Auf diese Weise erzeugte Ausdrücke können durch Klammerung rekursiv zusammengesetzt werden.

Beispiel:

```
<!ELEMENT Person (Name, Telefon*,
                  (Studienfach|(Ausbildung, Arbeitgeber)))>
```

Hier wird festgelegt, dass Personen einen Namen, beliebig viele Telefonnummern und ein Studienfach oder eine Ausbildung und gleichzeitig einen Arbeitgeber haben müssen. Ein Beispielelement ist

```
<Person>
    <Name>Anton Huber</Name>
    <Studienfach>Informatik</Studienfach>
</Person>
```

3. *Gemischter Inhalt (mixed)*: Elemente dieses Typs dürfen Zeichendaten (PCDATA) enthalten, die optional mit Kindelementen gemischt sind. Die Typen der Kindelemente können hier beschränkt werden, nicht jedoch ihre Reihenfolge oder ihre Anzahl.

4. *Jedes Element (ANY)*: Es kann jedes beliebige Element eingebettet werden, das in den DTDs deklariert ist.

3.2 Elementattribute

Elementattribute werden verwendet, um Elemente genauer zu spezifizieren. Eine *Attribut-Deklaration* bezieht sich immer auf einen angegebenen Elementtyp. Sie erfolgt in der DTD und legt für ein Element dieses Typs die zugelassenen Elementattribute, ihre Typen sowie Vorgaben für ihre Werte fest.

Als *Attribut-Vorgaben* können die Schlüsselworte REQUIRED, IMPLIED, FIXED sowie ein Defaultwert angegeben werden. Falls ein Elementattribut bei seiner Deklaration mit dem Schlüsselwort REQUIRED versehen ist, so muss dieses Attribut mit einem Wert versehen werden. Bei der Option IMPLIED ist es möglich, dieses Attribut nicht anzugeben.

Das Schlüsselwort FIXED kann nur in Kombination mit einem Defaultwert angegeben werden. In diesem Fall muss als Attributwert immer der Defaultwert angegeben werden. Wird das Attribut weggelassen, so wird der Defaultwert genommen.

In XML sind für Elementattribute unter anderem die Typen CDATA, ID, IDREF und IDREFS definiert.

1. *CDATA* ist ein Zeichenkettentyp. Er entspricht dem im Inhaltsmodell zugelassenen Typ PCDATA, wobei CDATA im Gegensatz zu PCDATA nicht geparsed wird.

2. Innerhalb eines gültigen XML-Dokuments müssen alle vergebenen ID-Werte unterschiedlich sein, so dass *ID-Attribute* der Identifizierung von Elementen dienen.

3. Als Wert eines *IDREF-Attributs* sind nur Zeichenketten zugelassen, die als ID-Wert eines beliebigen Elements des Dokuments vorkommen. Anzumerken ist hier, dass es ist nicht möglich ist, den Typ eines Elements, das über seine Identität referenziert wird, festzulegen. Der Parser prüft nur die Existenz des zugehörigen ID-Werts.

   Als Wert eines IDREFS-Attributs kann eine durch Leerzeichen getrennte Folge von Zeichenketten angegeben werden, die den Anforderungen von IDREF-Attributen genügen.

Bei der Erzeugung eines Elements in einem XML-Dokument werden die Werte in der *Attributspezifikation* innerhalb des Start-Tags mit Werten versehen. In Beispiel 1 werden zwei Elemente des Typs `Person` angelegt. Beide Elemente müssen ein `id-Attribut` haben. Der `Name` ist optional und wird beim zweiten Element weggelassen. In beiden Elementen wird für das Attribut `Beruf`, bzw. `Klasse` der Defaultwert, d.h. „*Schreiner*“ bzw. „*Person*“ angenommen. Im ersten Element wird das Attribut `Klasse` in der Attributspezifikation explizit angegeben, ein anderer Wert als “*Person*“ würde hier zu einem Fehler führen.

```
<!DOCTYPE Person> [                <Person id="person_1"
  <!ELEMENT Person EMPTY>                  Name="Theo Mustermann"
  <!ATTLIST Person                         Klasse="Person"/>
      id ID #REQUIRED              <Person id="person_2"/>
      Name CDATA
      Beruf (Schreiner, Metzger) "Schreiner"
      Klasse #FIXED "Person">
]>
```

Beispiel 1: Attribut-Deklaration und Verwendung

3.3 Entities

Entities sind eine Art Kürzel, sie haben einen Namen und einen Inhalt. Durch eine *Entity-Deklaration* (vgl. Beispiel 2) wird einem Namen eine Zeichenkette zugewiesen.

```
<!DOCTYPE Person> [                <Person id="person_1"
  <!-- Parameter-Entity -->                Name=&EinName;/>
  <!ENTITY % idAttribut            <Person id="person_2"/>
    "id ID #REQUIRED">

  <!-- Allgemeines Entity -->
  <!ENTITY einName "Theo Mustermann">

  <!ELEMENT Person>
  <!ATTLIST Person %idAttribut; Name CDATA>
]>
```

Beispiel 2: Attribut-Deklaration

Entities werden unterschieden in geparste und nicht geparste Entities, in interne und externe Entities sowie in allgemeine und Parameter-Entities. Im Rahmen dieser Arbeit werden nur interne, geparste Entities verwendet. Stösst der Parser auf ein solches, vorher deklariertes Entity, so ersetzt er das Entitity durch die zugehörige Zeichenkette.

Allgemeine und *Parameter-Entities* werden nach ihrer Verwendung unterschieden. Auf allgemeine Entities kann in XML-Dokumenten durch Einbetten des Namens in die Zeichen „&“ und „;“zugegriffen werden. Parameter-Entities sind nur innerhalb von DTDs zugreifbar, sie werden in die Zeichen „%“ und „;“ eingebettet (vgl. Beispiel 2).

4 Transformation

Ein Ziel der in dieser Arbeit vorgestellten Möglichkeiten zur Umsetzung von UML- in XML-Konzepte ist es, die Semantik der UML-Konzepte bestmöglichst zu erhalten. Dazu werden die semantischen Konzepte der UML auf DTD-Regeln abgebildet, so dass ein validierender Parser Verstösse gegen die Semantik möglichst aufdecken kann. Es darf nicht möglich sein, gültige XML-Dokumente zu erzeugen, die der Semantik der UML-Konzepte widersprechen.

Bei einer statischen Betrachtungsweise, wie sie durch die Modellierung in XML vorgegeben ist, geht es hierbei zunächst um die Konzepte *Klasse*, *Objekt* und *Identität* (vgl. Abschnitt 4.1). Danach werden Überlegungen zur Transformation von *Attributen* (vgl. Abschnitt 4.3) sowie von *Assoziationen* (vgl. Abschnitt 4.4) angestellt. In Abschnitt 4.5 gilt es die *Generalisierungshirarchie* geeignet umzusetzen.

4.1 Klassen und Objekte

Klassendefinitionen werden bei der Modellierung in XML auf Elementtypen abgebildet. Sie werden im Folgenden *Klassenelementtypen* genannt. Das Erzeugen einer Klasseninstanz in Form eines konkreten Objekts entspricht somit der Erzeugung eines Elements in einem XML-Dokument. Ein solches Element wird hier *Klassenelement* genannt.

Die Abbildung der Identität eines Objekts auf die eines Elements ist abhängig von der Speicherung der Elemente. Man kann Elemente auf folgende Weise speichern:

1. Alle Elemente werden in einer Datei gespeichert.

2. Elemente werden nach bestimmten Kriterien zusammengefasst und in einer Datei gespeichert. Ein Kriterium könnte hier beispielsweise der Elementtyp sein.

3. Jedes Element wird in einer eigenen Datei gespeichert.

Speichert man alle Elemente in einer Datei, so kann man den Elementen ein Attribut des Typs ID zuzuweisen (vgl. Beispiel 3). Die Identität eines Elements ist in diesem Fall gerade der Wert dieses Attributs. Die Eindeutigkeit der eingetragenen Werte wird durch einen validierenden Parser gewährleistet. So würde in Beispiel 3 das Erzeugen eines weiteren Elements mit dem Wert "*person_1*" oder "*person_2*" zu einer Fehlermeldung des Parsers führen.

Bildet man Gruppen von Elementen und speichert diese gemeinsam in einer Datei ab, so ist der *URI (uniform resource identifier)* zur Identifizierung einer Gruppe geeignet. Weist man den Elementen ein Attribut des Typs ID zu, so ist die Identifizierung innerhalb einer Gruppe durch einen validierenden Parser gewährleistet (vgl. Beispiel 4). Die Identität eines Elements innerhalb des modellierten Systems besteht nun aus seinem URI in Kombination mit dem Wert des ID-Attributs.

Speichert man jedes Element in einer separaten Datei, so ist die Identität eines Elements sein URI und somit eindeutig.

Bei den letzten beiden Verfahren sind Aspekte der Referenzierung mittles XPath [CD99], XLink [MD98a] und XPointer [MD98b] zu berücksichtigen. Im Rahmen dieser Arbeit wird von der Speicherung aller erzeugten Elemente in einer Datei ausgegangen.

```
<!DOCTYPE Firma [                    <!-- Datei Firma: -->
 <!-- Wurzelelement Firma:-->        <Firma>
 <!ELEMENT Firma                      <Person id="person_1"/>
     (Person*, Abteilung*)>           <Person id="person_2"/>
                                      <Abteilung id="abteilung_1"/>
 <!-- Klasse Person:-->              <Abteilung id="abteilung_2"/>
 <!ELEMENT Person>                  </Firma>
 <!ATTLIST Person id ID #REQUIRED>

 <!-- Klasse Abteilung:-->
 <!ELEMENT Abteilung>
 <!ATTLIST Abteilung id ID #REQUIRED>
]>
```

Beispiel 3: Speichern aller Elemente in einer Datei

4.2 Resumee

Klassentypen aus einem UML-Modell werden bei der Modellierung in XML auf Elementtypen (Klassenelementtypen) abgebildet. Objekte des Modells entsprechen demnach Elementen (Klassenelementen) in einem XML-Dokument. Ausgehend von der Speicherung aller erzeugten Elemente in einer Datei, wird die Identität eines Klassenelements durch das Aufnehmen eines Elementattributs vom Typ ID in den Klassenelementtyp erreicht.

4.3 Attribute

Wie im letzten Abschnitt beschrieben, werden Klassentypen der UML in XML auf Klassenelementtypen abgebildet. Objektattribute können in XML prinzipiell auf zwei Arten dargestellt werden.

1. Sie können als Elementattribute (vgl. Abschnitt 4.3.1) modelliert werden oder
2. man erzeugt für jeden Objektattributtyp einen eigenen Elementtyp, der hier *Attributelementtyp* genannt wird. Elemente eines Attributelementtyps heissen entsprechend *Attributelemente*. Sie müssen in irgendeiner Form den Klassenelementen zugeordnet werden, wobei es zwei unterschiedliche Verfahren gibt:
   (a) Einbetten der Attributelemente in ein Klassenelement (vgl. Abschnitt 4.3.2)
   (b) Referenz in den Attributelementen auf das Klassenelement (vgl. Abschnitt 4.3.3)

4.3.1 Objektattribute als Elementattribute

Bildet man Objektattribute auf Elementattribute ab (vgl. Beispiel 5), so ist man auf die in der XML-Spezifikation vorgesehenen Datentypen für Attribute beschränkt (vgl. Abschnitt 3.2). Im allgemeinen wird hier der Typ CDATA, d.h. ein Zeichenkettentyp in Frage kommen. Die Typintegrität kann im Allgemeinen nicht durch den Parser geprüft werden, so sind beispielsweise keine numerischen Typen vorgesehen.

```
<!DOCTYPE Personen [                  <Datei: Personen -->
 <!-- Wurzelelement Personen:-->      <Personen>
 <!ELEMENT Personen (Person*)>          <Person id="person_1"/>
 <!-- Klasse Person:-->                 <Person id="person_2"/>
 <!ELEMENT Person>                    </Personen>
 <!ATTLIST Person id ID #REQUIRED>
]>

<!DOCTYPE Abteilungen> [              <!-- Datei: Abteilungen -->
 <!-- Wurzel Abteilungen:-->          <Abteilungen>
 <!ELEMENT Abteilungen                  <Abteilung id="abteilung_1"/>
              (Abteilung*)>             <Abteilung id="abteilung_2"/>
 <!-- Klasse Abteilung:-->            <Abteilungen>
 <!ELEMENT Abteilung>
 <!ATTLIST Abteilung id ID #REQUIRED>
]>
```

Beispiel 4: Typweise Speicherung

Gewährleistet ist bei dieser Modellierung jedoch, dass ein Klassenelement nur die in den DTDs festgelegten Attribute enthält und dass die Attribute ihrem Klassenelement eindeutig zugeordnet sind. Weiterhin können Attributwerte in der Attributspezifikation nicht doppelt belegt werden.

```
<!DOCTYPE Person> [                  <Person id="p_1"
  <!ELEMENT Person>                          Name="Theo Mustermann"
  <!ATTLIST Person                           Alter="31"/>
     id ID #REQUIRED                 <Person id="p_2"
     Name CDATA #REQUIRED                    Name="Peter Schmidt"
     Alter CDATA #REQUIRED                   Alter="42"/>
     Klasse CDATA #FIXED "Person">
]>
```

Beispiel 5: Modellierung von Objektattributen als Elementattribute

Es können hier für Objektattribute die Kardinalitäten [0..1], [1..1], [0..*] sowie [1..*] festgelegt werden. Durch Hinzunehmen oder Weglassen der Option REQUIRED legt man die untere Grenze auf 0 oder 1 fest. Weiterhin kann man Attribute durch vorher festzulegende Zeichen trennen, wodurch eine Mehrwertigkeit des Attributs erreicht würde. Dies ist jedoch keine saubere Lösung. Deweiteren gibt es keine Möglichkeit, die Anzahl an möglichen Werten nach oben zu beschränken.

Konstante Klassenattribute können hier wie folgt abgebildet werden: Versieht man ein Elementattribut mit der Option FIXED und gibt einen Defaultwert an, so wird der Wert für alle erzeugten Elemente auf den Defaultwert gesetzt. Das Überschreiben in einem Element würde zu einem Fehler führen, so dass der Wert somit für alle Elemente gleich ist. Da der Attributwert beim Erstellen der DTD angegeben werden muss, können nur konstante Klassenattribute auf diese Weise modelliert werden. In Beispiel 5 wird das Klassenattribut `Klasse` auf diese Weise modelliert, da der Attributwert hier beim Erstellen der DTD feststeht.

4.3.2 Eingebettete Attributelemente

Bettet man die Attributelemente in die Klassenelemente ein (vgl. Beispiel 6), so wird über das Inhaltsmodell festgelegt, dass ein Klassenelement nur die in der DTD vorgegebenen Attribute enthalten kann. Gleichzeitig ist die eindeutige Zuordnung eines Attributelements zu seinem Klassenelement gesichert.

```
<!DOCTYPE Person [                    <Person id="p_1">
  <!ELEMENT Person                      <Name>Theo Mustermann</Name>
              (Name, Alter)>            <Alter>31</Alter>
  <!ATTLIST Person                    </Person>
              id ID #REQUIRED>
  <!ELEMENT Name  (#PCDATA)>
  <!ELEMENT Alter (#PCDATA)>
]>
```

Beispiel 6: Eingebettete Attributelemente

Typintegrität kann auch hier nicht erreicht werden. Da der in ein Attributelement eingetragene Wert ein Zeichentyp ist, kann beispielsweise für ein Attribut „Alter“ eine beliebige Zeichenkette eingetragen werden.

Die Kardinalität von Attributen kann bei dieser Modellierung uneingeschränkt durch das Inhaltsmodell (vgl. Abschnitt 3.1) festgelegt werden. Durch Aufzählen der Elemente und entsprechendes Anfügen von „?“, „+“ bzw. „∗“ kann die Kardinalität entsprechend variiert werden.

Im Beispiel 6 muss beim Erzeugen eines Elements des Typs Person ein Name und ein Alter angegeben werden.

Klassenattribute können auf diese Weise nicht sinnvoll umgesetzt werden. Würde man ein Klassenattribut als eingebettetes Attributelement modellieren, so müsste ein solches Klassenattributelement in alle Klassenelemente eingebett werden. Dies ist durch die ungewollte Redundanz sehr fehleranfällig, da nicht gewährleistet werden kann, dass alle eingebetteten Klassenattributelemente mit demselben Wert belegt werden.

Prinzipiell ist es bei dieser Modellierung möglich, dass Attributelemente auch in andere Klassenelementtypen eingebettet werden. Dies kann jedoch nur beabsichtigt geschehen, da es in einer DTD festgelegt werden muss.

Es kann nicht verhindert werden, dass ein Attributelement als Wurzelelement definiert wird. Da wir uns hier aber darauf beschränken, dass alle Elemente in einer Datei gespeichert werden, würde auf diese Weise nur genau dieses Attributelement erzeugt werden können, so dass ein solcher Fehler schnell auffällt.

4.3.3 Referenz in den Attributelementen auf das Klassenelement

Neben dem Einbetten gibt es die Möglichkeit, Attributelemente ausserhalb der Klassenelemente zu erzeugen. Die Zuordnung findet in diesem Fall über Referenzierung statt. Da Objektattribute nur von genau einer Klasse referenziert werden dürfen, können die Referenzen nur mit der oberen Grenze 1 in die Attributelemente aufgenommen werden (s.a. Abschnitt 4.4.2). Man kann demnach für die Anzahl der Attributelemente pro Klassenelement weder eine obe-

re noch eine untere Grenze angeben, so dass nur mehrwertige, optionale Objektattribute auf diese Weise modellierbar sind.

In Beispiel 7 wird das Attribut `Name` der Klasse `Person` auf diese Weise modelliert. Es kann hier nicht verhindert werden, dass mehrere Namen pro Person erzeugt werden.

```
<!DOCTYPE Person [                    <Person id="p_1"/>
  <!ELEMENT Person EMPTY>             <Name refPerson="p_1">
  <!ATTLIST Person                      Peter Schmidt
      id       ID #REQUIRED>          </Name>
  <!ELEMENT Name (#PCDATA)>
  <!ATTLIST Name
      refPerson IDREF #REQUIRED>
]>
```

Beispiel 7: Referenzierte Attributelemente

Zur Referenzierung können hier nur Referenzattribute eingesetzt werden. Würde man in die Attributelemente ein Referenzelement einbetten, so müsste das Inhaltsmodell `mixed` gewählt werden. In diesem Fall könnte man allerdings beliebig viele Referenzelemente einbetten, wodurch die eindeutige Zuordnung eines Attributelements zu seinem Klassenelement verloren ginge.

Typintegrität kann auch bei dieser Modellierung nicht gewährleistet werden, da mittels IDREF-Attributen beliebige Elemente referenzierbar sind (vgl. Abschnitt 3). Es ist in jedem Fall sinnvoll, eine Benennung der Referenzattribute zu wählen, aus der der Typ des referenzierten Elements hervorgeht. So ist im Beispiel 7 beim Erzeugen eines Attributelements `Name` der Typ des zu referenzierenden Klassenelements aus der Benennung ($refPerson$) ersichtlich.

Auch Klassenattribute lassen sich auf diese Weise modellieren. Zunächst wählt man einen vorgebenen Wert für die Identität des Klassenattributelements. Weiterhin wird das IDREF-Attribut im Klassenelement als FIXED definiert, wobei der Defaultwert die Identität des Klassenattributelements ist. Alle Klassenelemente zeigen auf diese Weise auf dasselbe Klassenattributelement, welches allerdings nun auch definiert werden muss. In dem eingangs vorgestellten UML-Modell (vgl. Abbildung 1) ist dies eine sinnvolle Umsetzung für das Klassenattribut `Anzahl` der Klasse `Angestellte`.

Eine weitere Variante zur Modellierung von Klassenattributen besteht darin, alle Objektattribute einer Klasse zu einem einzigen Attributelementtyp zusammenzufassen. Die Objektattribute können dann innerhalb dieses Attributelementtyps wie in diesem Kapitel beschrieben umgesetzt werden. Das Attributelement wird dann eingebettet oder referenziert. Eine sinnvolle Anwendung ist bei der Abbildung einer Vererbungshirarchie (vgl. Abschnitt 4.5.2) dargestellt.

4.3.4 Resumee

Für Objektattribute werden in XML Elementtypen eingeführt. Über das Inhaltsmodell kann dabei jede beliebe Kardinalität vorgeben werden.

Konstante Klassenattribute werden als Elementattribute mit der Option FIXED modelliert, wobei der Attributwert als Defaultwert eingetragen wird. Er muss somit in der DTD festgelegt werden.

Klassenattribute werden als eigener Elementtyp umgesetzt. Die Klassenelemente erhalten hierbei ein Referenzattribut des Typs IDREF mit der Option FIXED und der ID des Klassenattributelements als Defaultwert. Es kann hierbei entweder für jedes Klassenattribut ein eigener Klassenelementtyp eingeführt werden oder alle Klassenattribute werden in einem Klassenattributelement zusammengefasst.

4.4 Assoziation

Zur Abbildung von Assoziationen ist es erforderlich, die betroffenen Klassenelemente einander in irgendeiner Form zuzuordnen. Dazu hat man die folgenden Möglichkeiten

1. Einbetten eines der Klassenelemente in das andere (vgl. Abschnitt 4.4.1)
2. Referenzieren innerhalb der Klassenelemente (vgl. Abschnitt 4.4.2)
3. Assoziationsklassenelement (vgl. Abschnitt 4.4.3)

Falls für eine Assoziationsbeziehung Attribute zu modellieren sind, so wird in den folgenden Abschnitten darauf eingegangen, welchem Element sie zugeordnet werden können. Bei allen vorgestellten Modellierungen, können die Attribute innerhalb der Elemente, denen sie zugeordnet sind wie in Abschnitt 4.3 beschrieben modelliert werden.

4.4.1 Einbetten eines der Klassenelemente in das andere

Bettet man ein Element in das andere ein, so ist die eindeutige Zuordnung des Kindelements zum Vaterelement gewährleistet. Die obere Grenze für Kardinalität auf Seiten des Kindelements ist somit auf 1 beschränkt. Lässt man in einer DTD das eingebettete Element auch als Kindelement des Wurzelelements zu, so bildet man eine optionale Beziehung auf Seiten des Kindelements ab, ansonsten wird eine untere Grenze von 1 auf Seiten des Kindelements erreicht.

Auf Seiten des Vaterelements kann durch das Inhaltsmodell (vgl. Abschnitt 3.1) die Kardinalität uneingeschränkt festgelegt werden.

Die Typintegrität ist hier durch das Inhaltsmodell gewährleistet, es können nur Elemente des angegebenen Typs eingebettet werden.

Falls für eine Assoziationsbeziehung Attribute zu modellieren sind, so sind sie dem eingebetteten Element zuzuordnen. Über das Inhaltsmodell des Kindelements kann bei nicht optionalen Beziehungen die Kardinalität der Beziehungsattribute beliebig vorgegeben werden. Ist die modellierte Beziehung optional, so können nur optionale Beziehungsattribute umgesetzt werden.

Auch rekursive Beziehungen sind auf diese Weise umsetzbar, allerdings nur, wenn sie optional sind. Bei nicht optionalen, rekursiven Beziehungen hätte man keine Möglichkeit zum Abbruch der Rekursion.

Falls das eingebettete Element keine Identität erhält, so kann es nicht von anderen Elementen referenziert werden, so dass diese Modellierung für schwache Entities, ebenso wie für exklusive part-of-non-shared-Beziehungen sehr vorteilhaft ist.

Ein Problem tritt auf, wenn für das Kindelement mehr als eine Beziehung zu modellieren ist. So kann ohne Redundanz maximal eine solche Beziehung durch Einbetten realisiert werden. Würde man in unserem Modellbeispiel die Beziehungen `Leitung` und `Anstellung`

durch Einbetten realisieren, so müsste zunächst `Angestellter` in `Abteilung` eingebettet werden. Wollte man nun gleichzeitig den leitenden Angestellten in `Abteilung` einbetten, so ist zum einen der Angestellte doppelt definiert, zum anderen ist nicht ersichtlich, welcher der eingebetteten Angestellten der leitende Angestellte ist. Bettet man die `Abteilung` in den leitenden Angestellten ein, so ist auch hier dieser doppelt definiert, nämlich als Vaterelement der Abteilung und innerhalb der Abteilung als Angestellter.

In unserem Beispielmodell (vgl. Abbildung 1) kommt Einbetten für die Assoziation `Anstellung` in Frage (vgl. Beispiel 8).

```
<!DOCTYPE Firma [                    <Firma>
  <!ELEMENT Firma (Abteilung*)>        <Abteilung>
  <!ELEMENT Abteilung                    <Angestellter/>
      (Angestellter, Angestellter*)>     <!-- weitere Angestellte -->
  <!ELEMENT Angestellter EMPTY>        </Abteilung>
]>                                   </Firma>
```

Beispiel 8: Assoziation - Einbetten

4.4.2 Referenzieren innerhalb der Klassenelemente

Ein Element steht bei der Modellierung von Assoziationen durch Referenzen in einer Beziehung zu einem zweiten Element, wenn entweder das erste eine Referenz auf das zweite enthält oder umgekehrt.

Prinzipiell ist es möglich, Referenzen auf zwei Arten umzusetzen, nämlich durch *Referenzattribute*, d.h. ein IDREF- oder IDREFS-Attribut oder über *Referenzelemente*. Referenzelemente sind leere Elemente, die nur ein einziges Attribut des Typs IDREF besitzen. Bettet man sie in eine Klasse ein, so haben sie die Funktion von Referenzen.

Falls in diesem Abschnitt von Referenzen gesprochen wird, so gelten die Überlegungen für beide Varianten gleichermassen. Auf Unterschiede wird explizit eingegangen.

Eine *einwertige Referenz* ist ein Referenzattribut des Typs IDREF, bzw. ein Referenzelement, das über das Inhaltsmodell in die referenzierende Klasse höchstens einmal eingettet wird. Eine *mehrwertige Referenz* ist analog ein Referenzattribut des Typs IDREFS, bzw. ein Referenzelement, das über das Inhaltsmodell in die referenzierende Klasse mehrmals eingettetet wird.

Von einer *einseitigen Referenz* sprechen wir, wenn nur ein beteiligter Klassenelementtyp eine Referenz erhält, bei einer *beidseitigen Referenz* erhalten beide Seiten eine Referenz.

Modelliert man eine Beziehung zwischen Klassenelementen durch Referenzierung, so hat man in Abhängigkeit der abzubildenden Kardinalität verschiedene Möglichkeiten:

1. Eine *obere Grenze für eine Klasse* kann prinzipiell nur durch einseitiges Referenzieren gewährleistet werden (vgl. Abbildung 2a) und b)).

   Bei Referenzelementen kann die obere Grenze für die referenzierende Seite über das Inhaltsmodell uneingeschränkt festgelegt werden.

   Bei Referenzattributen ist nur 1 als obere Grenze modellierbar, in diesem Falle muss die Referenz einwertig sein (vgl. Abbildung 2a))

2. Es ist nicht möglich, für *beide Klassen eine obere Grenze* festzusetzen, da für die referenzierte Seite nicht nachprüfbar ist, wieviele Referenzen auf sie zeigen.

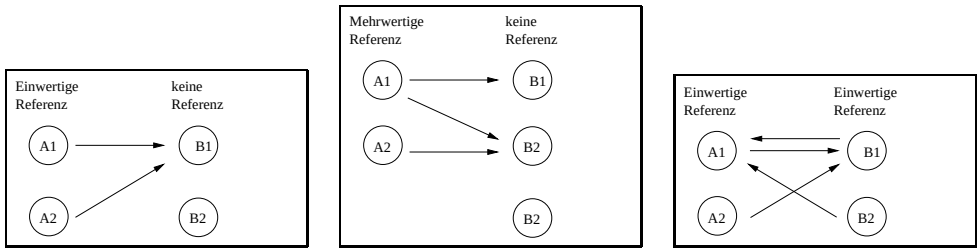
3. Eine *untere Grenze für eine Klasse* kann prinzipiell nur modelliert werden, indem sie selber eine Referenz enthält. Deshalb kann bei einseitiger Referenzierung (vgl. Abbildung 2a) und b)) nur für die referenzierende Seite eine untere Grenze angegeben werden.

   Bei Referenzattributen ist hier durch Hinzunehmen oder Weglassen der Option REQUIRED nur 1 bzw. 0 als untere Grenze möglich.

   Über das Inhaltsmodell ist bei Referenzelementen jede untere Grenze, d.h. insbesondere auch 1, modellierbar.

4. Sollen *beide Klassen eine untere Grenze* erhalten, so muss beidseitige Referenzierung (vgl. Abbildung 2c)) gewählt werden.

   Auch hier ist bei Referenzattributen nur 1 als untere Grenze modellierbar, während für Referenzelemente jede beliebige untere Grenze möglich ist.



a) Einseitige, einwertige Referenzen

b) Einseitige, mehrwertige Referenzen

c) Beidseitige, einwertige Referenzen

Abbildung 2: Assoziation - Referenzen

Typintegrität kann bei dieser Modellierung nicht gewährleistet werden, da mittels IDREF-Attributen beliebige Elemente referenzierbar sind (vgl. Abschnitt 3). Es ist in jedem Fall sinnvoll, eine Benennung der Rerenzattribute bzw. Referenzelemente zu wählen, aus der der Typ des referenzierten Elements bzw. seine Rolle hervorgeht.

Bei beidseitiger Referenzierung ist es nicht möglich, Attribute einer Beziehung ohne Semantikverlust abzubilden. Hier gäbe es Mehrdeutigkeiten, falls zwei Elemente sich gegenseitig referenzieren.

Bei einseitiger, einwertiger Referenzierung können Attribute einer Assoziationsbeziehung dem referenzierenden Klassenelement zugeordnet werden. Bei optionalen Beziehungen sind jedoch auch hier nur optionale Beziehungsattribute auf diese Weise modellierbar.

Eine angemessenere Abbildung von Beziehungsattributen wird durch den Einsatz von Referenzelementen erreicht. Beziehungsattribute werden hier den Referenzelementen zugeordnet, wobei durch das Inhaltsmodell der Referenzelemente beliebige Kardinalitäten umsetzbar sind.

Auch rekursive Beziehungen sind durch Referenzen modellierbar. Ein Element kann dann auch eine Beziehung zu sich selbst eingehen. Bezüglich Kardinalität gelten bei rekursiven Beziehungen dieselben Betrachtungen wie oben. Beidseitiges Referenzieren wird hier realisiert,

indem für jede Rolle der Klasse ein eigener Referenztyp bzw. ein eigenes Referenzattribut eingeführt wird. Als Bennenung für die Referenzen sollten die Rollen gewählt werden.

N-wertige Beziehung, bei denen n grösser als zwei ist, können wie folgt durch einseitige Referenzierung modelliert werden. Es wird ein Referenzelementtyp in eine der an der Beziehung beteiligten Klassen eingebettet. Dieser Referenzelementtyp erhält Referenzen auf die übrigen Klassenelemente, die an der Beziehung beteiligt sind. Bezüglich Kardinalitäten, Typintegrität, Beziehungsattributen und rekursiven Beziehungen gelten dieselben Betrachtungen wie bei binären Beziehungen.

In unserem Beipielmodell (vgl. Abbildung 1) kann die Assoziation `Anstellung` nicht ohne Semantikverlust durch Referenzierung abgebildet werden. Sie hat auf beiden Seiten eine untere Grenze und auf einer Seite eine obere Grenze.

Dasselbe gilt für die Assoziation `Leitung`, da hier auf beiden Seiten obere und untere Grenzen zu modellieren sind.

In Beispiel 9 wird die rekursive Beziehung `FreundVon` durch einseitige mehrwertige Referenzierung mittels Referenzattributen realisiert. Sie könnte ebenfalls als beidseitige Referenzierung abgebildet werden, da hier keine untere Grenze festgelegt ist. Auch könnte man Referenzelemente einsetzen.

```
<!DOCTYPE Firma [                    <Firma>
  <!ELEMENT Firma (Angestellter*)>     <Angestellter id="a_1"/>
  <!ELEMENT Angestellter EMPTY>        <Angestellter id="a_2"
  <!ATTLIST Angestellter                      hatFreund="a_1"/>
          id ID REQUIRED               <Angestellter id="a_3"/>
          hatFreund IDREF>           </Firma>
]>
```

Beispiel 9: Assoziation - Referenzen

4.4.3 Assoziationsklassenelement

Bei der Modellierung von Assoziationen durch ein Assoziationsklassenelement erhält dieses für jede an der Beziehung beteiligte Klasse eine Referenz. Elemente stehen in einer Beziehung zu anderen Elementen, wenn es ein Assoziationsklassenelement gibt, das Referenzen auf diese Elemente enthält.

Auch hier können die Referenzen durch Referenzattribute oder Referenzelemente umgesetzt werden.

Es kann hier *weder eine obere noch eine untere Grenze* für eine Klasse angegeben werden (vgl. Abbildung 3), da nicht überprüft werden kann, von wievielen Assoziationsklassenelementen eine Klasse referenziert wird.

Die Überlegungen bezüglich rekursiver Beziehungen, Benennung der Referenzen und Typintegrität sind dieselben wie bei der Modellierung mittels Referenzieren innerhalb der Klassenelemente (vgl. Abschnitt 4.4.2). Attribute werden hier dem Assoziationsklassenelement zugeordnet. Da Redundanzen von einem validierenden Parser nicht aufgedeckt werden, sind jedoch Mehrdeutigkeiten möglich. Beziehungsattribute sind deshalb nicht ohne Semantikverlust auf diese Weise modellierbar.

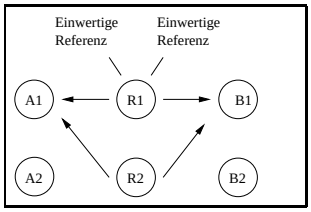


Abbildung 3: Assoziationsklassenelement

Die Assoziationen `Anstellung` und `Leitung` können nicht ohne Semantikverlust durch Assoziationselemente abgebildet werden, da es keine Möglichkeit für obere oder untere Grenzen gibt.

In Beispiel 10 wird die rekursive Beziehung `FreundVon` durch Referenzattribute innerhalb eines Assoziationsklassenelements abgebildet.

```
<!DOCTYPE Firma [                    <Firma>
  <!ELEMENT Firma                      <Angestellter id="a_1"/>
     (Angestellter*, FreundVon*)>      <Angestellter id="a_2"/>
  <!ELEMENT Angestellter EMPTY>        <Angestellter id="a_3"/>
  <!ATTLIST Angestellter               <FreundVon
          id ID REQUIRED>                   hatFreund="a_1"
  <!ELEMENT FreundVon EMPTY>                istFreundVon="a_2"/>
  <!ATTLIST FreundVon                </Firma>
          hatFreund    IDREF
          istFreundVon IDREF>
]>
```

Beispiel 10: Assoziationsklassenelement

4.4.4 Resumee

Aufgrund der Problematik von IDREF-Attributen (vgl. Abschnitt 3) ist nur die Modellierung von Assoziationen durch Einbetten ohne Semantikverlust möglich. Es können hier jedoch nur binäre 1:M Beziehungen abgebildet werden. Auf Seiten des Kindelements ist eine optionale oder obligatorische Beziehung mit oberer Grenze 1 umsetzbar. Auf Seiten des Vaterelements ist jede beliebige Kardinalität möglich. Besonders bei schwachen Entities oder exklusiven part-of-non-shared-Beziehungen ist diese Modellierung am geeignetsten.

Shared-Beziehungen, n-wertige oder N:M-Beziehungen sind nur über Referenzen modellierbar. Bei Assoziationselementen sind keine Kardinalitätsrestriktionen möglich, ebensowenig eine verlustfreie Umsetzung von Beziehungsattributen.

Bei Referenzierung innerhalb der Klassenelemente sind folgende Kardinalitäten abbildbar. Es kann nur auf einer Seite eine obere Grenze angegeben werden, wobei man auf einseitige Referenzierung beschränkt ist. Ebenso ist die verlustfreie Abbildung von Beziehungsattributen nur bei einseitiger Referenzierung mittels Referenzelementen möglich.

Zur Umsetzung einer unteren Grenze für eine Klasse, muss diese eine Referenz auf die anderen an der Beziehung beteiligten Elemente erhalten.

4.5 Generalisierung

Bei der Modellierung von Vererbung müssen alle Objektattribute und Beziehungen der Superklasse auch in der Subklasse enthalten sein. Zunächst kann man die gemeinsamen Objektattribute zusammenfassen und sie beiden Klassen zuordnen. Eine Möglichkeit hierzu bieten Parameterentities (vgl. Abschnitt 4.5.1). Man könnte auch einen Attributelementtyp einführen, in dem alle Objektattribute der Superklasse zu einem Attributelement zusammengefasst sind (vgl. Abschnitt 4.5.2).

Eine andere Möglichkeit ist es, in das Subklassenelement nur die Attribute und Beziehungen aufzunehmen, die im Rahmen der Spezialisierung hinzukommen. In diesem Fall muss man gewährleisten, dass es für jedes Subklassenelement genau ein Superklassenelement gibt, welches den zugehörigen Superklassenteil enthält. Zu einem Superklassenelement darf es höchstens ein Subklassenelement geben. Im Falle einer abstrakten Superklasse muss es genau ein Subklassenelement geben. Wir haben es hier demnach mit zwei oberen Grenzen zu tun. Da dies durch Referenzen nicht abzubilden ist (vgl. Abschnitt 4.4.2 und 4.4.3), kommt hier nur das Einbetten in Frage (vgl. Abschnitt 4.5.3).

Wir beschränken uns in diesem Kapitel auf Einfachvererbung. Zukünftige Arbeiten werden auch die Umsetzungen für Mehrfachvererbung betrachten.

4.5.1 Vererbung durch Parameterentities

Man kann für die Attributliste und die Kindelemente des Superklassenelements Parameterentities definieren. Sie werden dann in die Element-Deklaration der Subklasse zusätzlich zu den dort zu modellierenden Attributen, bzw. Kindelementen eingesetzt (vgl. Beispiel 11).

Da alle Objektattribute und Beziehungen der Superklasse über ihre Attributliste bzw. ihre Kindelemente modelliert sind, ist gewährleitet, dass alle Objektattribute und Beziehungen auch in der Subklasse enthalten sind.

4.5.2 Vererbung durch ein Attributelement

Fasst man alle Attribute und Beziehungen zu einem Attributelement zusammen, so gilt es zu gewährleisten, dass ein solches Attributelement genau einem Super- bzw. Subklassenelement zugeordnet ist. Weiterhin muss jedes Super- und Subklassenelement genau ein solches Attributelement enthalten. Da wir es hier auf beiden Seiten mit einer oberen Grenze zu tun haben, kann dies nur durch Einbetten realisiert werden (vgl. Beispiel 12).

4.5.3 Vererbung durch Einbetten

Bettet man das Superklassenelement in das Subklassenelement ein, so kann über das Inhaltsmodell des Subklassenelementtyps gewährleistet werden, dass jedes Subklassenelement genau ein Superklassenelement enthält. Auch kann kein Superklassenelement in mehr als einem Subklassenelement eingebettet sein. Ist die Superklasse abstrakt, so bettet man den Superklassenelementtyp nicht in das Wurzelelement, sondern nur in die Subklasse ein (vgl. Beispiel 13). Das Subklassenelement erhält keine eigene Identität, seine Identität ist die des eingebetteten Superklassenelements.

```
<!DOCTYPE Firma [
 <!ENTITY % AttListPerson
             "id ID #REQUIRED">
 <!ENTITY % AttributePerson
             "Name, Alter">

 <!ELEMENT Firma
              ((Person|Angestellter)*)>

 <!ELEMENT Person (%AttributePerson;)>
 <!ATTLIST Person %AttListPerson;>

 <!ELEMENT Angestellter
           (%AttributePerson;,
           Personalnummer)>
 <!ATTLIST Angestellter %AttListPerson;
           hatFreund IDREF>

 <!ELEMENT Name EMPTY>
 <!ELEMENT Alter EMPTY>
 <!ELEMENT Personalnummer EMPTY>
]>
```

```
<Firma>
  <Person id="p_1">
    <Name/>
    <Alter/>
  </Person>
  <Angestellter id="a_1">
    <Name/>
    <Alter/>
    <Personalnummer/>
  </Angestellter>
</Firma>
```

Beispiel 11: Vererbung durch Parameterentities

Bettet man das Subklassenelement in das Superklassenelement ein, so kann ebenfalls gewährleistet werden, dass ein Subklassenelement nur innerhalb eines Superklassenelements existiert (vgl. Beispiel 14). Im Inhaltsmodell des Wurzelelements wird ein Subklassenelement nicht zugelassen, sondern nur innerhalb der Superklasse. Falls die Superklasse abstrakt ist, so darf das Einbetten eines Subklassenelements nicht optional sein. Das Subklassenelement erhält keine eigene Identität, seine Identität ist die des übergeordneten Vaterelements.

4.5.4 Resumee

Bei allen vorgestellten Umsetzungen von Vererbung ist gewährleistet, dass alle Attribute und Beziehungen eines Superklassenelements auch in dem Subklassenelement vorhanden sind. Das intuitive Verständnis einer Vererbungshirarchie, d.h. *ein Subklassenelement ist ein Superklassenelement*, ist beim Einbetten eines Subklassenelements in ein Superklassenelement am besten abgebildet. Bettet man ein Superklassenelement in ein Subklassenelement ein, so entspricht dies dem Verständnis *ein Subklassenelement hat ein Superklassenelement*. Mit der Modellierung durch Zusammenfassen der Attribute zu einem Attributelement erreicht man das Verständnis *ein Subklassenelement hat dieselben Attribute wie ein Superklassenelement*. Aus der Modellierung mittels Parameterentities ist auf den ersten Blick kein Zusammenhang zwischen Super- und Subklasse ersichtlich.

```
<!DOCTYPE Firma [
 <!ELEMENT Firma
      ((Person|Angestellter)*)>

 <!ELEMENT Person
      (AttributePerson)>
 <!ELEMENT AttributePerson
      (Name, Alter)>
 <!ATTLIST AttributePerson
      id ID #REQUIRED>

 <!ELEMENT Angestellter
      (AttributePerson,
      AttributeAngestellter)>
 <!ELEMENT AttributeAngestellter
      (Personalnummer)>
 <!ATTLIST AttributeAngestellter
       hatFreund IDREF>

 <!ELEMENT Name EMPTY>
 <!ELEMENT Alter EMPTY>
 <!ELEMENT Personalnummer EMPTY>
]>
```

```
<Firma>
  <Person>
    <AttributePerson
                id="p_1">
      <Name/>
      <Alter/>
    </AttributePerson>
  </Person>
  <Angestellter>
    <AttributePerson
                 id="a_2">
      <Name/>
      <Alter/>
    </AttributePerson>
    <AttributeAngestellter>
      <Personalnummer/>
      <Alter/>
    </AttributeAngestellter>
  </Angestellter>
</Firma>
```

Beispiel 12: Vererbung durch ein Attributelement

```
<!DOCTYPE Firma [
 <!ELEMENT Firma
             ((Person|Angestellter)*)>

 <!ELEMENT Person (Name, Alter)>
 <!ATTLIST Person id ID #REQUIRED>

 <!ELEMENT Angestellter
           (Person,
            Personalnummer)>
 <!ATTLIST Angestellter
           hatFreund IDREF>
 <!-- Attributelemente -->
]>
```

```
<Firma>
  <Person id="p_1">
    <Name/><Alter/>
  </Person>
  <Angestellter>
    <Person id="a_2">
      <Name/><Alter/>
    </Person>
    <Personalnummer/>
  </Angestellter>
</Firma>
```

Beispiel 13: Vererbung durch Einbetten der Superklasse

```
<!DOCTYPE Firma [
 <!ELEMENT Firma
             ((Person)*)>

 <!ELEMENT Person
        (Name, Alter, Angestellter?)>
 <!ATTLIST Person id ID #REQUIRED>

 <!ELEMENT Angestellter Personalnummer)>
 <!ATTLIST Angestellter
           hatFreund IDREF>
 <!-- Attributelemente -->
]>
```

```
<Firma>
  <Person id="p_1">
    <Name/><Alter/>
  </Person>
  <Person id="a_2">
    <Name/><Alter/>
    <Angestellter>
      <Personalnummer/>
    </Angestellter>
  </Person>
</Firma>
```

Beispiel 14: Vererbung durch Einbetten der Subklasse

5 Zusammenfassung und Ausblick

Die Umsetzung verschiedener objektorientierter Konstrukte, wie sie in UML unterstützt werden, in XML-DTDs wurden diskutiert und evaluiert. Bei der Evaluation stand der Grad des Erhalts der Semantik des konzeptuellen UML-Schemas im Vordergrund. In weitergehenden Arbeiten planen wir die Kommodität der Umsetzungen für die Anfragebearbeitung (beispielsweise mittels XQL [RLS98]) zu berücksichtigen.

Danksagung Wir danken Herrn Kanne, Herrn Helmer und Herrn Westmann für zahlreiche fruchtbare Diskussionen.

Literatur

[BPSM98] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (xml) 1.0. Technical report, World Wide Web Consortium, 1998. W3C Recommendation 10-Feb-98.

[CD99] J. Clark and S. DeRose. XML path language (XPath). Technical report, World Wide Web Consortium, 1999. W3C Working Draft 9 July 1999.

[GP99] C. Goldfarb and P. Prescod. *XML Handbuch.* Prentice Hall, 1999.

[Mar98] M. Marchiori. Ql'98 - the query languages workshop. Technical report, W3C, Dec. 1998. http://www.w3.org/TandS/QL/QL98.

[MD98a] E. Maler and S. DeRose. XML linking language (XLink). Technical report, World Wide Web Consortium, 1998. W3C Working Draft 3-March-1998.

[MD98b] E. Maler and S. DeRose. Xml pointer language (xpointer). Technical report, World Wide Web Consortium, 1998. World Wide Web Consortium Working Draft 03-March-1998.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, Reading, Massachusetts, USA, 1 edition, 1999.

[RLS98] J. Robie, J. Lapp, and D. Schach. XML query language (XQL). In *[Mar98]*, 1998.

[Teo99] T. Teorey. *Database Modeling and Design.* Morgan Kaufmann, 1999.