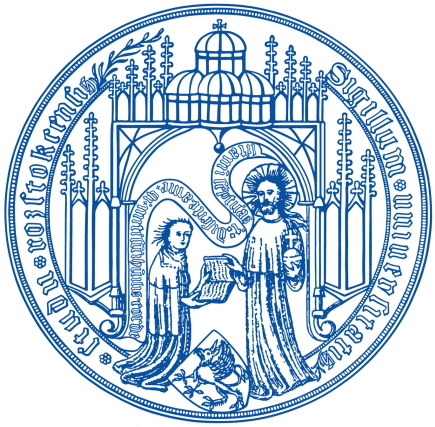
Model Driven Engineering für XML-Anwendungen

Diplomarbeit

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik



vorgelegt von:	Christian Schubert
Matrikelnummer:	1201875
geboren am:	19.01.1982 in Dresden
Erstgutachter:	Prof. Dr. Andreas Heuer
Zweitgutachter:	Prof. Dr. Clemens Cap
Betreuer:	Dr. Meike Klettke

Abgabedatum: 30.04.2008

Abstract

Der Entwurf und die Bearbeitung von XML-Schemata können durch Verwendung von Modellen erleichtert werden. In der hier vorgelegten Arbeit wurde das Konzept des Model Driven Engineering bzw. Model Driven Architecture aus dem Gebiet Softwareentwicklung auf die Entwicklung von XML-Schemata verschiedener Schemasprachen (XML-DTD, XML-Schema, Schematron) übertragen. Die Idee der Model Driven Architecture (MDA) ist, die Softwareentwicklung, also die Erzeugung von Programmcode, über Modelle zu formalisieren und durch einen zumindest teilautomatischen Transformationsprozess weniger fehleranfällig zu machen. Dabei wird zwischen so genannten plattformunabhängigen und plattformabhängigen Modellen unterschieden. Die plattformunabhängigen Modelle abstrahieren von der technischen Umsetzung, und sollen so für „Jedermann“ verständlich sein. Die plattformabhängigen Modelle beziehen sich dann dagegen auf bestimmte technische Plattformen. Sie geben dem Entwickler die Möglichkeit der Modellierung von plattformabhängigen Details, die für die Umsetzung in eine bestimmte Programmiersprache notwendig sind. In die Entwicklung von XML-Schemata übertragen entsprechen also die verschiedenen XML-Schemasprachen den Plattformen. So wurden in dieser Arbeit mehrere von bestimmten XML-Schemasprachen abhängige bzw. unabhängige Modelle entwickelt. Des Weiteren wurden eine Entwicklungsmethode und ein zu großen Teilen automatisierter Transformationsprozess beschrieben. Diese zeigen den Weg von den verschiedenen Modellen bis hin zur Codeerzeugung.

Inhaltsverzeichnis

Abbildungsverzeichnis **5**

Abkürzungsverzeichnis **7**

1 Einleitung **9**

2 Extensible Markup Language - XML **11**

3 Schemasprachen **13**
3.1 DTD . . . 13
3.1.1 Elemente . . . 14
3.1.2 Attribute . . . 15
3.1.3 Entities & Notations . . . 16
3.1.4 Bedingte Abschnitte . . . 17
3.2 XML-Schema . . . 17
3.2.1 Struktur von XML-Schema . . . 18
3.2.2 Typsystem . . . 19
3.2.3 Datentypen . . . 20
3.2.4 Elementdefinition und Elementeigenschaften . . . 22
3.2.5 Attributdefinition . . . 24
3.2.6 Integritätskonzept . . . 24
3.2.7 Modellierungsstile . . . 25
3.3 Schematron . . . 26
3.3.1 Funktionsweise . . . 27
3.3.2 Struktur . . . 28

4 Vergleich der Schemasprachen **33**
4.1 Erlernbarkeit & Benutzbarkeit . . . 33
4.2 Spracheigenschaften . . . 33
4.3 Unterstützung von Datenbankeigenschaften . . . 33
4.4 Ausdrucksstärke der XML-Schemasprachen . . . 34
4.5 Zusammenfassung . . . 35

5 Model Driven Architecture **37**
5.1 Einführung MDA . . . 37
5.2 Architektur . . . 37
5.3 MDA-Konzepte . . . 39
5.4 MDA-Entwicklungsprozess . . . 42
5.5 Vorteile und Probleme der MDA . . . 43

6 Adaption der MDA auf Entwurf für XML-Schemata **47**
6.1 Vorgehensweise . . . 47
6.2 Plattformunabhängiges Modell für XML-Schemata . . . 47
6.2.1 Vereinfachtes Entity Model for XML-Schema . . . 48
6.2.2 Eigenes Modell . . . 50
6.3 Plattformabhängige Modelle für XML-Schemata . . . 52
6.3.1 Plattformabhängige Modelle für das vereinfachte CoDEX-Modell . . . 53
6.3.2 Plattformabhängige Modelle für das eigene Modell . . . 57
6.4 Entwurfsmethode & Transformationsprozess für CoDEX-Modelle . . . 60
6.4.1 Entwurfsmethode & Transformationsprozess für die Plattform DTD . . . 60
6.4.2 Entwurfsmethode & Transformationsprozess für die Plattform XML-Schema . . . 63
6.4.3 Entwurfsmethode & Transformationsprozess für die Plattform Schematron . . . 66
6.5 Entwurfsmethode & Transformationsprozess für eigene Modelle . . . 68
6.5.1 Entwurfsmethode & Transformationsprozess für die Plattform DTD . . . 68
6.5.2 Entwurfsmethode & Transformationsprozess für die Plattform XML-Schema . . . 69
6.5.3 Entwurfsmethode & Transformationsprozess für die Plattform Schematron . . . 73

7 Evaluation und Ausblick **75**
7.1 Bewertung der Modelle und Entwurfsmethode . . . 75
7.2 Fazit & Ausblick . . . 75

Literaturverzeichnis **77**

Abbildungsverzeichnis

3.1 Wohlgeformtheit, Gültigkeit, brauchbare XML Dokumente . . . 13
3.2 Wohlgeformtheit, Gültigkeit, Gültigkeit gegen ein Schema, brauchbare XML Dokumente . 18
3.3 XML-Schema Typhierarchie . . . 20
3.4 XML-Dokument-Validierungsprozess . . . 28
3.5 Schematron Report in HTML-Frames . . . 29

4.1 Klassifikation von XML-Schemasprachen . . . 34
4.2 Klassifikation Ausdrucksstärke der XML-Schemasprachen . . . 34

5.1 OMG's Model Driven Architecture . . . 38
5.2 (Meta-) Modellebenen am Bsp. . . . 39
5.3 Prinzipielles MDA-Transformationsschema . . . 40
5.4 Mapping und Metamodelle für das Ausgangs- und das Zielmodell . . . 41
5.5 PIM-PSM-Code-Transformationen . . . 42
5.6 Beispiel CIM-PIM-PSM-Transformationsprozess . . . 43
5.7 Beispiel Vorgehensweise MDA . . . 44
5.8 PIM-zu-PSM-Transformation mit Mapping und Marking . . . 45
5.9 Phasen und Iterationen in der MDA . . . 46

6.1 vereinfachtes CoDEX-Beispiel . . . 48
6.2 Notation im vereinfachten CoDEX-Modell (kursiv + fett dargestellt Platzhalter) . . . 49
6.3 Eigenes Modell am Beispiel . . . 50
6.4 Notation im eigenen Modell (kursiv + fett dargestellt Platzhalter) . . . 51
6.5 Komponentensicht für eigenes Modell (Beispiel nicht vollständig) . . . 52
6.6 erweitertes Beispiel für DTDabhängiges Modell in CoDEX-Notation . . . 53
6.7 erweitertes Beispiel für das von XML-Schema abhängige Modell in CoDEX-Modell-Notation 54
6.8 erweitertes Beispiel für das von Schematron abhängige Modell in CoDEX-Notation . . . 55
6.9 Beispiel für eigenes DTD-Modell (wird fortgesetzt...) . . . 56
6.10 Beispiel für eigenes DTD-Modell (Fortsetzung) . . . 57
6.11 Beispiel für eigenes XML-Schema-Modell . . . 58
6.12 Beispiel für eigenes Schematron-Modell . . . 59
6.13 Transformation (-sschablonen) vom abhängigen CoDEX-DTD-Modell zu Code . . . 61
6.14 Transformation (-sschablonen) vom abhängigen CoDEX-DTD-Modell zu Code (Fortsetzung) 62
6.15 Modifikationen am CoDEX-XML-Schema-Modell . . . 63
6.16 Transformation (-sschablonen) vom abhängigen CoDEX-XML-Schema-Modell zu Code . . 64
6.17 Transformation (-sschablonen) für eigenes DTD-Modell . . . 68
6.18 Transformation (-sschablonen) für eigenes XML-Schema-Modell . . . 70

Abkürzungsverzeichnis

CASE	Computer Aided Software Engineering - Computergestützte Softwareentwicklung
CIM	Computation Independent Model
CoDEX	Conceptual Design and Evolution of XML schemas
CWM	Common Warehouse Model
DTD	(XML)- Document Type Definition
DSD	(XML)- Document Structure Description
EJB	Enterprise Java Beans
EMX	Entity Model for XML-Schema
GUI	Graphical User Interface (grafische Benutzeroberfläche)
HTML	Hypertext Markup Language
ISO	International Organization for Standardization
J2EE	Java Platform Enterprise Edition
JSON	Java Script Object Notation
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MOF	Meta Object Facility
MOF-QVT	Meta Object Facility - Query View Transformation
OCL	Object Constraint Language
OMG	Object Management Group
OMG IDL	Object Management Group Interface Definition Language
OO	Object Oriented
OOA	Object Oriented Analysis
OOD	Object Oriented Design
OOP	Object Oriented Programming
OOT	Object Oriented Testing
PI	Processing Instruction
PIM	Platform Independent Model - Plattformunabhängiges Modell
PM	Platform Model - Plattform Modell
PMP	Plattformmodellierungsphase
PSM	Platform Platform Specific Model - Plattformabhängiges Modell
SQL-DDL	Structured Query Language - Data Definition Language
UML	Unified Modeling Language
XSLT	Extensible Stylesheet Language (XSL) Transformation
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Kapitel 1

Einleitung

Seit Ende des vergangenen Jahrtausends entwickelte sich die Extensible Markup Language XML zu einer der beliebtesten Textauszeichnungssprachen für strukturierte Daten und einem beliebten Datenaustauschformat. Um die Erstellung qualitativ hochwertiger und fehlerfreier XML-Dokumente zu unterstützen, wurden verschiedene sogenannte Schemasprachen entwickelt. Diese versuchen den korrekten Aufbau und zum Teil auch den Inhalt von XML-Dokumenten hinsichtlich einer speziellen Anwendung zu beschreiben. Dabei sind die verschiedenen Schemasprachen zum Teil auf sehr unterschiedliche Aspekte ausgerichtet worden. Die einen sind eher auf Kompaktheit, leichte Erlernbarkeit und Übersichtlichkeit bedacht, während andere auf möglichst vollständige und umfangreiche Strukturbeschreibungen abzielen. Wieder andere verfolgen den Ansatz XML-Dokumente auch semantisch zu beschränken bzw. zu überprüfen.

In dieser Arbeit soll für die drei zum Teil sehr unterschiedlichen Schemasprachen XML-DTD, XML-Schema und Schematron ein gemeinsames „Obermodell“ entwickelt werden. Aus diesem Modell soll es dann möglich sein, weitere Modelle für alle drei Schemasprachen abzuleiten. Diese ebenfalls zu spezifizierenden Modelle sollen dann mehr auf die jeweiligen Schemasprachen zugeschnitten sein. Im letzten Schritt soll es dann möglich sein, aus diesen zugeschnittenen Modellen den Quellcode für die entsprechenden Schemasprachen zu generieren. Dazu wird in dieser Arbeit eine Entwurfsmethode, also ein Vorgehensweise, entwickelt und der Transformationsprozess der Modelle in einander bzw. zu Quellcode beschrieben. Diese Strategie, der Codeerzeugung aus „Modellketten“ ist eine relativ alte Idee, neu angeregt in der Model Driven Architecture (MDA). Die MDA kommt aus dem Bereich der Softwareentwicklung und wurde 2000 erstmals durch die Object Management Group (OMG) vorgestellt. Dabei ist die MDA aber anders als z.B. CASE nicht auf vollständige automatische Erzeugung von Code ausgerichtet, da dies zumeist nicht möglich oder relativ schwierig ist. Stattdessen werden nur sinnvolle Anteile des Quelltextes, wie oftmals Strukturbeschreibungen oder Ähnliches, über Modelle automatisch erzeugt. Dadurch ist die MDA wesentlich flexibler und hat in den letzten Jahren wesentlich an Popularität gewonnen.

Dieses Konzept der MDA soll in dieser Arbeit also auf die Entwicklung von XML-Schemata übertragen werden. Dabei untergliedert sich die Arbeit in die folgenden Abschnitte. Im Kapitel 2 wird nur sehr kurz auf die XML eingegangen, weil diese bereits in zahlreichen Abhandlungen sehr umfassend und genau beschrieben wurde. In Kapitel 3 werden die verschiedenen XML-Schemasprachen vorgestellt und etwas ausführlicher beschrieben. Im folgenden Kapitel 4 werden die in Kapitel 3 vorgestellten Schemasprachen mit einander verglichen, um so Gemeinsamkeiten und Unterschiede für die verschiedenen Modelle heraus zu arbeiten. Mit der MDA wird dann in Kapitel 5 ein Konzept aus dem Bereich Softwareentwicklung vorgestellt. Die in diesem Kapitel vorgestellten Konzepte und Ideen stellen die Grundlage für das, dann im Hauptteil (Kapitel 6) der Arbeit beschriebene, Vorgehen beim modellgetriebenen Entwickeln von XML-Anwendungen dar. Kapitel 7 setzt sich dann kritisch mit den in Kapitel 6 vorgestellten Modellen und Vorgehensweisen aus einander und gibt einen Ausblick auf eventuell vorstellbare Ergänzungen und Weiterentwicklungen.

Kapitel 2

Extensible Markup Language - XML

Da bereits in vielen wissenschaftlichen Abhandlungen auf die Extensible Markup Language (XML) eingegangen wurde, soll diese hier nur kurz angesprochen werden. Die XML ist die Sprache, in der die Dokumente verfasst werden, die durch die Schemasprachen beschreiben werden sollen. Die Extensible Markup Language (XML) [W3CXML06] ist wie der Name schon sagt eine einfache, sehr flexible und erweiterbare Textauszeichnungssprache. Sie wurde von W3 Konsortium aktuell in der Version 1.1 (2. Auflage) spezifiziert. Die XML ist eine vereinfachte Untermenge der SGML (Standard Generalized Markup Language), die in einer Reihe von Designzielen [XMLidP] vor allem auf die Menschen-Lesbarkeit ausgerichtet ist. Sie wird vor allem zum Datenaustausch speziell über das Internet verwendet. Es sei auf weiterführende Literatur [W3CXML06], [XMLA03], [XMLC02] verwiesen.

Kapitel 3

Schemasprachen

`XML-Schemasprachen` stellen eine sehr gute Möglichkeit dar, den Inhalt von XML-Dokumenten zu beschreiben, reglementieren und strukturieren. Ziel all dieser Schemasprachen ist es eine Verbesserung der Qualität von XML-Dokumenten. Um brauchbare XML-Dokumente zu gewährleisten, wurden für die Qualität von XML-Dokumenten die Kriterien Gültigkeit und Wohlgeformtheit eingeführt. Dabei beschreibt die `Wohlgeformtheit` zunächst nur, dass die Dokumente dem grundlegenden Regelsatz der XML-Spezifikation [W3CXML06] genügen. Um `gültig` bzw. `valide` zu sein, muss das Dokument gegen ein Schema validiert werden. Dazu muss das Dokument zunächst wohlgeformt sein. Weiterhin müssen alle Elemente und Attribute im Schema definiert sein und sie müssen, gemäß ihrer Definition im Schema, im Dokument verwendet worden sein. Abbildung 3.1 zeigt den Zusammenhang zwischen wohlgeformten, gültigen und brauchbaren XML Dokumenten.

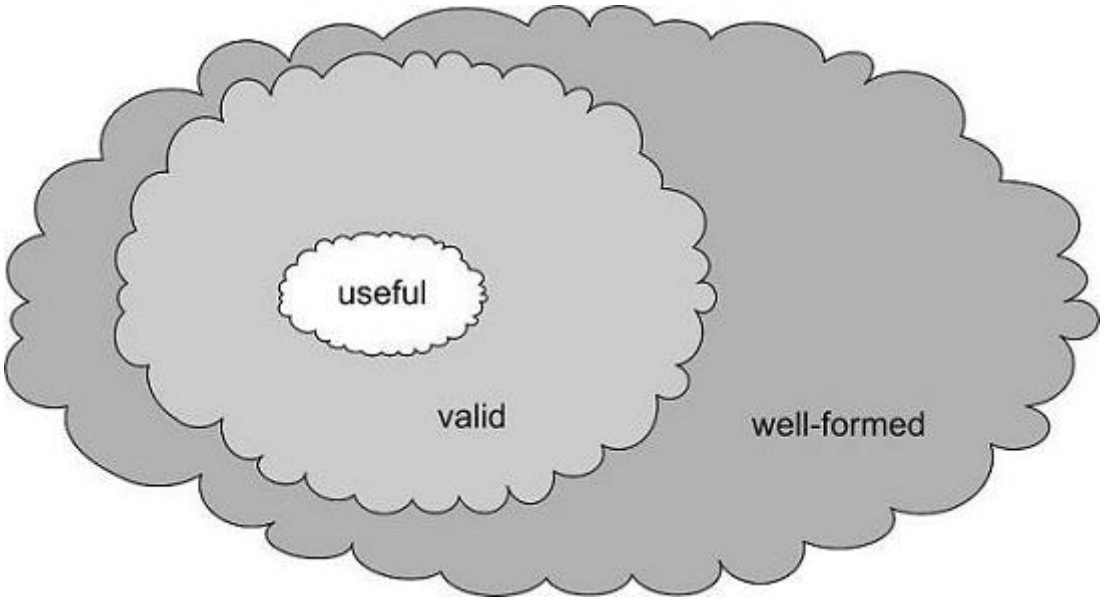


Abbildung 3.1: Wohlgeformtheit, Gültigkeit, brauchbare XML Dokumente [ZEN07a]

Für XML-Dokumente gibt es eine ganze Reihe von verschiedenen Schemasprachen. Dabei sind einige wie zum Beispiel DTD auf Einfachheit, Kompaktheit und gute Lesbarkeit ausgerichtet, andere wie XML-Schema wiederum haben den Anspruch XML-Dokumente möglichst genau, umfassend, detailliert und flexibel zu beschreiben. In den folgenden drei Abschnitten sollen die drei XML-Schemasprachen DTD (Document Type Definition)[W3CDTD06], XML-Schema [XMLS04] und Schematron [STRON04] vorgestellt werden.

3.1 DTD

Die `Document Type Definition` (DTD) ist, wie bereits zuvor erwähnt, eine einfache und sehr kompakte Schemasprache, die sich auf wenige Beschreibungselemente beschränkt. Sie ist dadurch relativ einfach

und schnell zu erlernen und sehr übersichtlich und gut (menschen-)lesbar. Die DTD ist direkt in der W3C-Recommandation für XML [W3CDTD06] beschrieben, obwohl sie selbst im Gegensatz zu anderen Schemasprachen keine XML-Syntax besitzt. Somit ist möglich, die DTD direkt im zu beschreibenden XML-Dokument mit anzugeben (interne DTD). Es ist aber auch möglich, die DTD in eine externe Datei auszulagern und mittels `<!DOCTYPE Rootelement SYSTEM 'datei.dtd'>` auf diese Datei im Prolog des XML-Dokument zu verweisen (externe DTD). Die Grundbausteine der DTD sind Elementdeklaration, Attributdeklaration zu Elementen, Entitiesdeklaration, Notationsdeklaration, Processing Instructions (PI) und Kommentare. Auf die Hauptbestandteile Elemente und Attribute und auf Entities wird noch genauer in den folgenden Abschnitten eingegangen. `Processing Instructions` sind Anweisungen für die Verarbeitung von Dokumenten an die verarbeitenden Programme. Sie sind wie folgt gekennzeichnet: `<? PI-Name PI-Anweisung ?>`. Dabei ist darauf zu achten, dass der Name der PI nicht „xml“ ist. Der gebräuchlichste Einsatz von PIs ist die Zuweisung von Stylesheets zu XML-Dokumenten [KLME02]. `Kommentare` können dazu verwendet werden, Struktur in Dokumente zu bringen, um so die Lesbarkeit für den Anwender zu erhöhen. Die Syntax für Kommentare ist wie folgt: `<!-- Kommentar -->`. Kommentare können bei der Verarbeitung von verschiedenen XML-Prozessoren oder anderen Applikationen ausgewertet werden.

3.1.1 Elemente

Der wichtigste Baustein einer DTD ist die `Elementdeklaration`. Hier wird beschrieben, welche Elemente im XML-Dokument wie oft auftreten dürfen und wie diese Elemente strukturiert werden können. Folgende Grammatik [W3CDTD06] gibt die Syntax der Elementdefinition an, wobei „S“ hier für Leerzeichen und „Name“ für den Elementnamen, der gewissen Restriktionen des XML-Namensraum unterliegt, steht:

```
[45] elementdecl  ::=    '<!ELEMENT' S  Name  S  contentspec  S? '>'
[46] contentspec  ::=    'EMPTY' | 'ANY' | Mixed | children
[47] children     ::=    (choice | seq) ('?' | '*' | '+')?
[48] cp     ::=    (Name | choice | seq) ('?' | '*' | '+')?
[49] choice ::=    '(' S? cp ( S? '|' S? cp )+ S? ')'
[50] seq     ::=    '(' S? cp ( S? ',' S? cp )* S? ')'
[51] Mixed ::=  '(' S? '#PCDATA' (S? '|' S? Name)* S? ')*' | '(' S? '#PCDATA' S? ')'
```

Dabei wird semantisch in der Elementdefinition neben dem Namen des Elementes auch der mögliche Inhalt der Elemente mittels verschiedener Inhaltsmodelle definiert:

- `#PCDATA` für Zeichendaten (Parsed Character DATA)
- komplexe Elemente mit Kindelementen:
  - `A,B` für Reihenfolgen von Elementen (Sequenz: erst Element A dann Element B)
  - `A|B` für Alternativen (Element A oder B)
  - `(...)` für Gruppierungen
- `Quantifizierer` von Subelementen:
  - `A` ist kein Quantifizierer angegeben muss Element auftreten
  - $A^{?}$ für Kennzeichnung von optionalen Elementen (0..1 mal)
  - A^{+} für Subelemente die mindestens 1mal auftreten (1..n mal)
  - A^{*} für Subelemente die beliebig oft auftrete dürfen (0..n mal)
- `EMPTY` definiert leere Elemente
- `ANY` definiert Elemente mit gemischtem Inhalt (Kombination beliebiger Elemente und Zeichendaten)

Weiterhin ist es, wie schon in der Grammatik ersichtlich, möglich so genannte `Mixed Content` Elemente, eine Kombination von Zeichendaten mit Kindelementen anzugeben, wobei in der Deklaration des Elementes darauf zu achten ist, dass die #PCDATA-Angabe für Zeichendaten zuerst erscheinen muss. Beispiel: `(!ELEMENT D (#PCDATA|A|B|C)*)`. Bei einer Mixed-Content-Deklaration ist auch darauf zu achten, dass ein Subelement nur einmal vorkommen darf. Folgendes Beispiel-DTD zeigt die Bandbreite der Möglichkeiten auf:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Definition vom Element 'hotel' mit Subelemente 'name', 'kategorie' usw.-->
<!-- Subelement 'zimmer' mit '+' definiert mindestens ein Zimmer benötigt -->
<!ELEMENT hotel (name, kategorie, zimmer+, adresse, beschreibung?,
hotelausstattung*,anfahrt?)>
```

```
<!-- Ende der Definition vom Element 'hotel'-->
<!ATTLIST hotel hid NMTOKEN #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT kategorie EMPTY>
<!ATTLIST kategorie sterne (1|2|3|4|5) #REQUIRED>
<!ELEMENT zimmer (zimmernr, ausstattung*)>
<!ATTLIST zimmer zimmertyp CDATA #IMPLIED>
<!ELEMENT adresse (ort, ((plz, strasse) | postfach), adresszusatz?)>
<!ELEMENT beschreibung (#PCDATA | name | ort | strasse | ausstattung)*>
<!ELEMENT hotelausstattung (#PCDATA)>
<!ELEMENT anfahrt ANY>
<!ELEMENT zimmernr (#PCDATA)>
<!ELEMENT ausstattung (#PCDATA)>
<!ELEMENT ort (#PCDATA)>
<!ELEMENT plz (#PCDATA)>
<!ELEMENT strasse (#PCDATA)>
<!ELEMENT postfach (#PCDATA)>
<!ELEMENT adresszusatz (#PCDATA)>
<!ELEMENT hotels (hotel*)>
```

3.1.2 Attribute

Elemente in XML-Dokumenten können `Attribute` besitzen, die im jeweiligen Start-Tag des Elementes angegeben werden. In DTDs wird die Definition der Attribute in den zugehörigen Attributlisten der Elemente vorgenommen. Grammatikalisch wird das wie folgt angegeben:

```
[52] AttlistDecl    ::= '<!ATTLIST' S  Name  AttDef* S? '>'
[53] AttDef         ::= S Name S AttType S DefaultDecl
[54] AttType        ::= StringType | TokenizedType | EnumeratedType
[55] StringType     ::= 'CDATA'
[56] TokenizedType  ::= 'ID'|'IDREF'|'IDREFS'|'ENTITY'|'ENTITIES'|'NMTOKEN'|'NMTOKENS'
[57] EnumeratedType ::=  NotationType | Enumeration
[58] NotationType   ::= 'NOTATION' S '(' S? Name (S? '|' S? Name)* S? ')'
[59] Enumeration    ::= '(' S? Nmtoken (S? '|' S? Nmtoken)* S? ')'
[60] DefaultDecl    ::= '#REQUIRED' | '#IMPLIED' | (('#FIXED' S)? AttValue)
```

Die Bausteine der Attributdeklaration haben folgende Bedeutung:

- 'Name' der Attributname
- 'AttType' der Attributtyp:

`CDATA`	für Character DATA also Zeichendaten
`ID`	für dokumenteindeutige ID also Identifizierer
`IDREF/IDREFS`	zum Verweis auf eine oder mehrere IDs anderer Attribute
`ENTITY/ENTITIES`	zur Definition von Entitäten
`NMTOKEN/NMTOKENS`	für „name token“
`(wert1|wert2|wert3|...)`	Aufzählung erlaubter Werte

- 'DefaultDecl' für Use bzw. Vorgabewerte:

`'defautwert'`	für Defaultwerte von Attributen
`#REQUIRED`	für Attribute die angegeben werden müssen
`#IMLIED`	für optionale Attribute
`#FIXED 'fixwert'`	für fixe also vorgegebene Werte von Attributen

Das heißt: Im Anschluss an den Elementnamen werden die für dieses Element zugelassenen Attribute spezifiziert. Im einfachsten Fall sind das der eindeutige Attributname, ein Datentyp (gemäß der Produktionen [55] und [56]) und die Angabe, ob es sich um ein optionales (#IMPLIED) oder zwingend anzugebendes (#REQUIRED) Attribut handelt. Darüber hinaus können Attribute mit Vorgabewerten belegt und zusätzlich konstant (#FIXED) sein. `ID` und `IDREF(S)` bieten einen einfachen Mechanismus zur Referenzierung. Ein validierender Prozessor prüft, ob die angegebenen Referenzen gültig sind. Während IDREF auf eine ID verweist, kann mit IDREFS gleich eine ganze Liste von Verweisen angegeben werden. Zu beachten ist das die IDs im Dokument (global) eindeutig sein müssen. IDREF(S)-Attribute

müssen den selben Wert wie eine ID eines Attributes des Dokumentes haben, um auf diese zu verweisen. Die Verweisziele müssen existieren, andernfalls wird durch einen XML-Prozessor ein Fehler gemeldet. Folgendes Bsp. verdeutlicht den ID/IDREFS-Mechanismus:

```
<!-- DTD -->
<!ELEMENT verkaufsobjekt (bezeichnung)>
<!ATTLIST verkaufsobjekt auktionator IDREF #REQUIRED>
<!ELEMENT person (name, vorname)>
<!ATTLIST person id ID #REQUIRED>
...
<!-- XML Dokument -->
<verkaufsobjekt auktionator=´p01´>
   <bezeichnung>Ming-Vase</bezeichnung>
</verkaufsobjekt>
<verkaufsobjekt auktionator=´p05´>
   <bezeichnung>Barockflügeltür</bezeichnung>
</verkaufsobjekt>

<person id=´p05´>
   <name>Meyer</name> <vorname>Hans</vorname>
</person>
<person id=´p01´>
   <name>Müller</name> <vorname>Franz</vorname>
</person>
```

Der ID/IDREFS-Mechanismus ist nicht mit dem Schlüsselkonzept in Datenbanken gleich zu setzen. Die Schwächen [ZEN07] des ID/IDREFS-Verfahren sind:

- IDs in einem virtuellen Bereich, Bereich nicht partitionierbar, Folge: Namenskonflikte
- Verweisziele nicht einschränkbar (nicht-sinnvolle Verwiese möglich)
- Mechanismus nicht dokumentübergreifend

Auf Entities und Notations wird im folgenden Abschnitt noch eingegangen, deshalb sei hier nur noch kurz auf die `Aufzählungstypen` eingegangen. Sie bilden eine sehr einfache aber auch sehr beschränkte Möglichkeit anwenderdefinierte Datentypen anzugeben. Aufzählungstypen erlauben eine einfache Aufzählung der möglichen Werte, die das jeweilige Attribut annehmen kann. Folgendes Beispiel zeigt die Verwendung von Aufzählungstypen anhand der Aufzählung der möglichen Werte für das Attribut 'Wochentag' mit dem Defaultwert 'Mo' für Montag:

```
<!ATTLIST Meeting Wochentag (Mo|Di|Mi|Do|Fr|Sa|So) 'Mo'>
```

3.1.3 Entities & Notations

Mit DTDs ist es möglich sich eigene `Entities` zu definieren. Dies sind Ausdrücke, die wie in HTML mit '&' bzw. '%' beginnen und mit ';' enden. Die XML-Prozessoren analysieren den zwischen '&' und ';' stehenden Ausdruck und ermitteln den zugehörigen Ersetzungstext. Entities sind also eine Art Makro. Es lassen sich so zur Erstellungszeit bekannte Ausdrücke oder auch der Inhalt von ganzen Dateien einbinden, so dass der Ersetzungstext auch erst zur Aufrufzeit ermittelt werden kann. Die externen Dateien müssen dabei keine vollständigen XML-Dokumente sein. Durch Entities ist es damit möglich, Definitionen in eigene Dateien auszulagern um diese mehrfach wiederverwenden zu können. Im Gegensatz zu HTML gibt es in XML nur sehr wenige vordefinierte Entities: '<' für ' < ', '>' für ' > ', '&' für ' & ', '&quot' für ' " ', ''' für ' ' '. Diese Entities können immer verwendet werden. Jeder XML-Parser kennt diese Entities.

Man unterscheidet mehrere `Entitytypen`. Als erstes die Sonderklasse der Zeichenentities (ASCII-, Unicodezeichen). Diese werden nicht in der DTD festgelegt werden. Alle anderen Entities werden innerhalb einer DTD definiert. Man unterteilt weiter zwischen globalen Entities und Parameter-Entities. Globale Entities können, wie Zeichenentities, im Elementinhalt und in Attributen verwendet werden und beginnen mit '&'. Parameter-Entities werden in externen DTDs verwendet und starten mit '%'. Dann wird noch zwischen internen Entities und externen Entities unterschieden. Bei internen DTDs wird der Ersetzungstext direkt angegeben und bei externen wird, um den Ersetzungstext zu bestimmen, auf eine

andere Ressource zugegriffen. Zum Schluss gibt es noch geparste und ungeparste Entities. Erstere enthalten Text, Markup und eventuell noch weitere Entities und müssen geparst werden, während letztere ungeparst an die Anwendung übergeben werden. Ungeparste Entities stellen eine Möglichkeit dar, andere Dateiformate einzubinden, ohne dass diese Binärdaten nach den Markup durchsucht werden. Ein Beispiel wäre die Einbindung von Bildern:

```
<!ELEMENT penthouse (#PCDATA)>
<!ATTLIST penthouse view ENTITY #IMPLIED>
<!ENTITY fronview_pent SYSTEM "frontview.gif" NDATA myImage>
<!NOTATION myImage SYSTEM "image/gif">

<penthouse view="fronview_pent">
    Haus in Malibu
</penthouse>
```

`Notation`-Deklarationen haben die folgende Syntax:

```
[82] NotationDecl ::=     '<!NOTATION' S Name S (ExternalID | PublicID) S? '>'
[75] ExternalID   ::=     'SYSTEM' S SystemLiteral
                          | 'PUBLIC' S PubidLiteral S SystemLiteral
[83] PublicID     ::=     'PUBLIC' S PubidLiteral
```

Wie im obigen Beispiel schon zu sehen war, wird die 'Notation'-Deklaration dazu benutzt, um: „einer Notation einen Namen für die Verwendung in Entity- und Attributlisten-Deklarationen und in Attribut-Spezifikationen sowie einen externen Identifizierer für die Notation, der es dem XML-Prozessor oder seinem Anwendungsprogramm erlaubt, ein Hilfsprogramm zu finden, das in der Lage ist, Daten in der gegebenen Notation zu verarbeiten“ [XMLD02]. Das heißt, dass Notaions Verarbeitungshinweise für Daten, die nicht direkt vom XML-Prozessor verarbeitet werden können, an interpretierende Applikationen sind. Mittels dieser Anweisungen können externen Anwendungen Details über die referenzierten Daten mitgeteilt werden. Die Verwendung der definierten Notations erfolgt wie im obigen Beispiel für Entities ersichtlich, mit der Angabe einer 'ExternalID' (siehe obige Grammatik) und dem Signalwort 'NDATA' plus dem definierten Notaionsnamen.

3.1.4 Bedingte Abschnitte

`Bedingte Abschnitte` können nur in externen DTDs in Abhängigkeit von einem Schlüsselwort in die logische Struktur einer DTD eingebunden bzw. ausgeschlossen werden. Das Schlüsselwort 'INCLUDE' gibt an, dass der Inhalt des bedingten Abschnitts Teil der DTD ist, wo hingegen das Schlüsselwort 'IGNORE' den auszuschließenden Inhalt eines bedingten Abschnittes anzeigt. Der Inhalt eines ignorierten Abschnitts wird geparst, indem alle Zeichen nach der Klammer '[', die auf das Schlüsselwort 'IGNORE' folgt, ignoriert werden bis das passende Endzeichen auftritt. Parameterentities werden während dieses Vorgangs nicht erkannt. Sehr sinnvoll wird dieser Mechanismus im Zusammenspiel mit Parameterentities. Im folgenden Beispiel lassen sich durch einfaches Vertauschen der Parameterentities '`% entwurf`' und '`% fertig`' in den Entitydefinition, '`kommentare`' zum Element '`buch`' ein- bzw. ausschalten:

```
<!ENTITY % entwurf 'INCLUDE' >
<!ENTITY % fertig 'IGNORE' >

<![%entwurf;[
<!ELEMENT buch (kommentare*, titel, inhalt, anhaenge?)>
]]>
<![%fertig;[
<!ELEMENT buch (titel, inhalt, anhaenge?)>
]]>
```

3.2 XML-Schema

`XML-Schema` abgekürzt auch XSD (XML Schema Definition) ist im Gegensatz zu XML- DTD eine XML-Dokumentbeschreibungssprache, die selbst der XML Struktur entspricht. Da also jedes XML-Schema selber ein XML-Dokument ist, ist es möglich auch das Schema selber wieder durch ein Schema, also ein Metaschema, zu beschreiben. XML Schema ist die aktuell am meisten genutzte XML-Schemasprache und die aktuelle Empfehlung vom W3-Konsortium zur Definition der Struktur von XML-Dokumenten.

XML Schema bildet zusammen mit der XML-Spezifikation selbst und den XML-Namensräumen die Basis der weiteren W3C-XML-Sprachstandards. Die XML-Schema-Spezifikation ist im Allgemeinen wesentlich komplexer als die DTD-Spezifikation, da sie von sich aus schon eine ganze Reihe einfacher Datentypen, sogenannter „Built-In-Typen“, mit sich bringt. Des Weiteren lassen sich mit XML Schema auch komplexere und selbst definierte Datentypen erstellen, Integritätsbedingungen darstellen und Namensräume spezifizieren und nutzen. XML Schema erlaubt es also Inhaltsmodelle für Elemente, Attributstrukturen und wiederverwendbare Inhalte zu definieren. Durch die umfassenden Beschreibungsmöglichkeiten führen aber selbst kleinere Strukturen schnell zu sehr umfangreichen Beschreibungsdokumenten, die weniger gut menschenlesbar sind und somit zur Auswertung technischer Hilfsmittel bedürfen. Die Definition des W3C, an der sich auch die folgenden Abschnitte orientieren, ist in die drei Teile Einführung („Primier“)[XMLSa04], Strukturen („Structures“)[XMLSb04] und Datentypen („Datatypes“)[XMLSc04] untergliedert, wobei die Einführung ein informaler Beschreibungsteil mit vielen Beispielen ist.

XML Schema wird verwendet, um die Qualität von XML-Dokumenten zu erhöhen bzw. zu gewährleisten. Abbildung 3.2 erweitert die in Kapitel 3 vorgestellte Abbildung 3.1 und zeigt die Verfeinerung und oft damit einhergehende Qualitätsverbesserung von validen zu schemavaliden Dokumenten. Dabei bezeichnen valide Dokumente im Gegensatz zu schemavaliden Dokumenten hier, Dokumente die „nur“ gegen eine DTD valide sind. Da XML Schemata meist nicht Bestandteile eines XML-Dokumentes sind,

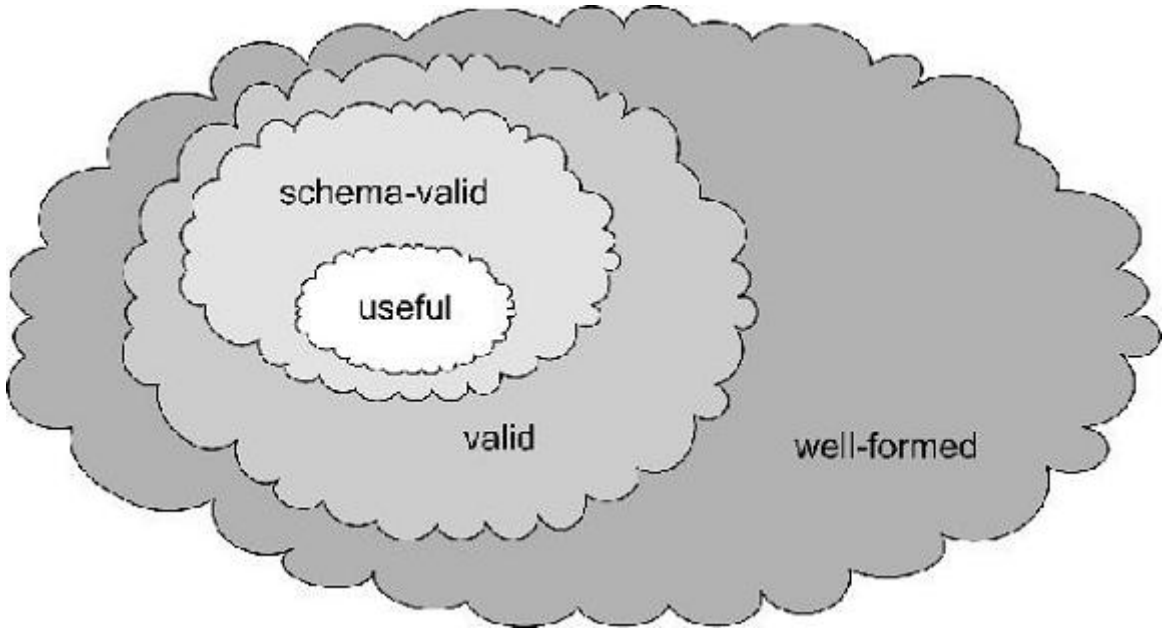


Abbildung 3.2: Wohlgeformtheit, Gültigkeit, Gültigkeit gegen ein Schema, brauchbare XML Dokumente [ZEN07b]

wird Zuordnung von XML Schemata zu den Dokumenten mittels der Attribute ’schemaLocation’ bzw. ’noNamespaceSchemaLocation’, die im Wurzelelement des XML-Dokumentes angegeben werden, vorgenommen. Wie immer eines der beiden Attribute angegeben werden muss, muss auch die Zuordnung des XML-Namensraumes im Wurzelelement des XML-Dokumentes erscheinen.

3.2.1 Struktur von XML-Schema

Ein XML Schema besteht primär aus Definitionen und Deklarationen. Mit erstem erzeugt man neue einfache oder komplexe Datentypen, mit zweitem beschreibt man das Inhaltsmodell von Elementen und Attributen. Nach [XMLSb04] besteht ein XML Schema aus insgesamt 13 sogenannter Schemakomponenten, die in drei Gruppen unterteilt werden. Die erste Gruppe bilden die primären Komponenten, welche da wären:

- Definitionen von einfachen Typen
- Definitionen von komplexen Typen
- Attribut-Deklarationen
- Element-Deklarationen

Hierbei können die ersten zwei genannten, also die Typdefinition Namen besitzen, die letzen zwei müssen sogar Namen besitzen.

Die zweite Gruppe sind die sekundären Komponenten:

- Definitionen von Attributgruppen
- Definitionen von Identitätseinschränkungen
- Definitionen von Modellgruppen
- Notations-Deklarationen

In dieser Gruppe müssen alle Komponenten einen Namen haben.

Die dritte Gruppe besteht aus den nicht selbstständigen Hilfskomponenten:

- Anmerkungen
- Elementgruppen
- Partikel
- Wildcards
- Attribut-Verwendungen

Diese Komponenten sind Bestandteile anderer Komponenten und sind somit nicht kontextunabhängig.

3.2.2 Typsystem

XML-Schema definiert zurzeit 44 sogenannte `Buit-In-Types`. Die Übersicht vorgegebener Datentypen kann aus Abbildung 3.3 entnommen werden. Die Datentypen lassen sich mit folgenden Eigenschaften unterscheiden:

- atomare versus aggregierte Typen
- primitive versus abgeleitete Typen
- vorgegebene versus anwenderdefinierte Typen
- einfache versus komplexe Typen

Ein `atomarer` Datentyp ist die kleinste Informationseinheit. Die beinhaltende Information ist nicht mehr weiter unterteilbar, d.h. nicht mehr weiter in andere Bestandteile strukturiert. Ein Beispiel: Der Datentyp 'date' erlaubt kein Einzelzugriff auf Jahr, Monat oder Tag. `Aggregierte` Datentypen hingegen setzen sich entweder aus mehreren Werten eines oder mehrerer verschiedener Datentypen zusammen. Diese Typen nennt man Vereinigungstypen. Oder sie setzen sich aus mehreren Werten eines atomaren Datentyps zusammen. Das sind die Listentypen, z.B. drei Werte des Datentyps 'decimal': '3', '14.5', '12.5'.

`Primitive` Datentypen existieren unabhängig von anderen Datentypen und sind in der Typhierarchie nur den sogenannten Urtypen 'anyType' und 'anySimpleType' untergeordnet (z.B. 'float'). `Abgeleitete` Datentypen bauen auf anderen Datentypen auf, sind also von deren Definition abhängig. Ein Beispiel ist der Datentyp 'integer' dessen Wertebereich der mathematischen Menge der ganzen Zahlen entspricht und somit eine Ableitung bzw. eine einschränkende Spezialisierung des Typs 'decimal' ist.

`Vordefinierte` Datentypen sind alle die, die in XML Schema Part 2 [XMLSc04] definiert sind. Es sind im Allgemeinen die Datentypen definiert, die auch aus gängigen Programmiersprachen bekannt sind. `Anwenderdefinierte` Typen werden, wie der Name schon sagt, vom Anwender selbst definiert bzw. festgelegt.

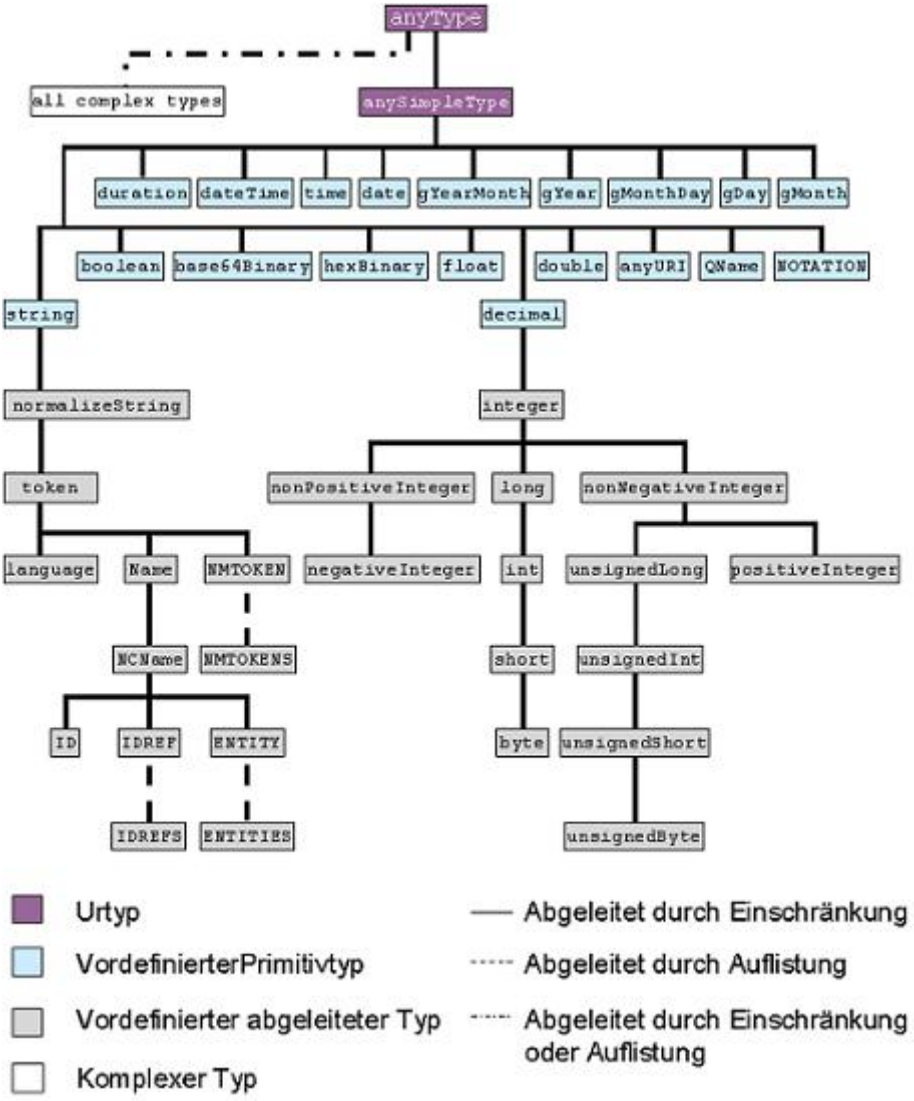


Abbildung 3.3: XML-Schema Typhierarchie
[XMLSc04]

3.2.3 Datentypen

Wie schon erwähnt, unterscheidet man hinsichtlich der Datentypen auch zwischen einfachen und die komplexen Datentypen. `Einfache Datentypen` zeichnen sich dadurch aus, dass sie nur Zeichendaten enthalten, also weder Subelemente noch Attribute besitzen. Dabei können diese Zeichendaten selbst verschiedenen Typs sein. Es werden die in Abschnitt 3.2.2 angegebenen vordefinierten Standarddatentypen wie 'string', 'boolean', 'float', 'integer', 'decimal', 'time', 'date', usw. als auch selbst definierte Datentypen unterstützt.

`Komplexe Datentypen` sind dagegen strukturierte Typen, die ausschließlich durch ihr Inhaltsmodell definiert werden. Sie können aus einfachen Datentypen abgeleitet werden. So ist auch möglich einfachen Datentypen Attribute zuzuordnen. Es sei darauf hingewiesen, dass so auch komplexe Typen ('complexType') mit einfachen Inhalt ('simpleContent') möglich sind. Dazu ein kurzes Beispielfragment:

```
<xsd:element name="Hotel">
 <xsd:complexType>
  <xsd:simpleContent>
   <xsd:extension base="xsd:string">
    <xsd:attribute name="Hotelname" type="xsd:string"/>
   </xsd:extension>
  </xsd:simpleContent>
 </xsd:complexType>
```

```
</xsd:element>
```

Da die Definition des Typs durch das Inhaltsmodell vorgenommen wird, ist die Benennung des Typs vom Namen der typisierten Instanz unabhängig. Syntaktisch erfolgt die Typbildung durch die Benennung des 'complexType'-Elements durch das Attribut 'name'. Um die mehrfache Verwendung eines solchen Typen zu ermöglichen, muss seine Definition zwingend auf einer Ebene erfolgen, die für alle nutzenden Elemente erreichbar ist. Daraus folgt, dass diese Definitionen auf der obersten Ebene, meist direkt unterhalb des Wurzelknotens, positioniert werden.

In XML-Schema gibt es die Möglichkeit, eigene Datentypen durch Einschränkung (Restriction), Erweiterung (Extension), Vereinigung, Listung und Substitution anderer Datentypen zu definieren. Als Basis dafür dient immer mindestens ein bereits definierter (vordefinierter oder auch selbstdefinierter) Datentyp.

`Einschränkungen` (Restrictions), die durch das XSD-Element `<xsd:restriction base="basistyp">` dargestellt wird, können zur Einschränkung eigener und auch vordefinierter Datentypen verwendet werden. Dabei ist der abgeleitete Datentyp dem Originaltyp meist sehr ähnlich. Die Deklarationen sind jedoch eingeschränkt, d.h. die Werte des abgeleiteten Typs entsprechen einer Untermenge der Werte des ursprünglichen Typs. Einschränkungen können über den Gebrauch von Facetten wie Länge ('length'), minimale Länge ('minLength'), maximale Länge ('maxLength'), Patterns ('pattern'), Aufzählung ('enumeration') usw. vorgenommen werden. Eine vollständige Liste über die einfachen Datentypen und die jeweilig anwendbaren Facetten zeigen die Tabellen B1.a. und B1.b.im Anhang B von XML Schema Part 0[XMLSa04]. Folgendes Beispiel zeigt die Definition eines eigenen Integerdatentyps 'myInteger', der nur Zahlen zwischen 1000 und 9999 zulässt, durch die Einschränkung des Basisdatentyps 'xsd:interger' mit den Facetten 'xsd:minInclusive' und 'xsd:maxInclusive':

```
<xsd:simpleType name="myInteger">
 <xsd:restriction base="xsd:integer">
  <xsd:minInclusive value="1000"/>
  <xsd:maxInclusive value="9999"/>
 </xsd:restriction>
</xsd:simpleType>
```

Bei der `Erweiterung` (Extension), die durch das XSD-Element `<xsd:extension base="basistyp">` angezeigt wird, wird die Definition eines Basistypen durch weitere zusätzliche Informationen (Elemente, Attribute) ergänzt. Im Beispiel wird der Typ 'Person', der die Kindelemente 'Name' und 'Titel' hat, um das Element 'Vorname' zum Typ 'erwPerson' erweitert:

```
<xsd:complexType name="Person">
 <xsd:sequence>
  <xsd:element name="Name" type="xsd:string"/>
  <xsd:element name="Titel" type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="erwPerson">
 <xsd:complexContent>
  <xsd:extension base="Person">
   <xsd:sequence>
    <xsd:element name="Vorname" type="xsd:string"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

Wie schon bei der Gegenüberstellung von atomaren und aggregierten Typen kurz erwähnt, gibt es für die einfachen Datentypen ('xsd:simpleTypes') die speziellen Datentypen Listen ('xsd:list') und Vereinigung ('xsd:union'). `Listentypen` bestehen aus einer Aneinanderreihung von Werten eines atomaren Datentypen. Der als Basis verwendete atomare Datentyp wird im Attribut 'itemType' des XSD-Elementes 'xsd:list' angegeben. In XML-Schema gibt es bereits die drei vordefinierten Listendatentypen 'NMTOKEN', 'IDREFS' und 'ENTITIES'. Das folgende Beispiel zeigt die Definition eines eigenen Listentyps 'Verweiseliste' als eine Liste von Strings:

```
<xsd:simpleType name="Verweiseliste">
  <xsd:list itemType="xsd:String"/>
</xsd:simpleType>
```

Ein XML-Dokumentfragment mit vier Elementen durch Leerzeichen getrennt könnte dann so aussehen: `<verweisliste>www.xyz.de www.abc.de www.123.de www.klm.com</verweisliste>`.

Dagegen setzen sich `Vereiningungstypen` aus einer Menge von Ausgangsdatentypen zusammen. Die verwendeten Basisdatentypen werden im Attribut 'membertypes' des XSD-Elementes 'xsd:union' aufgezählt. Das Beispiel zeigt die Definition eines Datentyps 'Jeansgroesse' als Vereinigung der Datentypen 'Nummerngroesse' und 'Stringgroesse':

```
<xsd:element name="Jeansgroesse">
  <xsd:simpleType>
    <xsd:union memberTypes="Nummerngroesse Stringgroesse" />
  </xsd:simpleType>
</xsd:element>

<xsd:simpleType name="Nummerngroesse">
  <xsd:restriction base="xsd:positiveInteger">
    <xsd:maxInclusive value="48"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="Stringgroesse">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="xs"/>
    <xsd:enumeration value="s"/>
    <xsd:enumeration value="m"/>
    <xsd:enumeration value="l"/>
    <xsd:enumeration value="xl"/>
    <xsd:enumeration value="xxl"/>
  </xsd:restriction>
</xsd:simpleType>
```

Als eine letzte Möglichkeit Datentypen zu definieren, soll hier die `Ersetzung` (Substitution) vorgestellt werden. Mit Hilfe der Substitution können Elemente durch andere Elemente ersetzt werden. Die Elemente werden hierzu sogenannten Substitutionsgruppen zugeordnet. Die Elemente dieser Gruppe können dann anstatt eines speziellen benannten Elements, des „Head-Elements“ verwendet werden. Das „Head-Element“ wird im Attribut 'substitutionGroup' des Ersetzungselementes angegeben. Im Beispiel wird definiert, dass das Element 'customer' durch das Element 'kunde' ersetzt werden kann.

```
<xsd:element name="customer" type="customertype"/>
<xsd:element name="kunde" substitutionGroup="customer"/>
```

Diese Definition erlaubt dann sowohl 'customer'- als auch 'kunde'-Elemente in den XML-Dokumentinstanzen. Es sei noch darauf verwiesen, dass die Substitution mit Hilfe des Attributes 'block' in XSD-Elementen verhindert werden kann. Dazu muss man im vor Ersetzung zu schützendem Element im Attribut 'block' den Wert 'substitution' mit angeben bzw. 'block' auf den Wert 'all' setzen.

3.2.4 Elementdefinition und Elementeigenschaften

Die Elementdefinition in XML-Schema beginnt mit dem Wurzelknotenelement 'xsd:schema'. Dabei hat das Schema-Element die folgenden Attribute:

- 'xmlns:xsd': (optionale) Angabe des Namensraums (targetNamespace)
- 'elementFormDefault'
- 'attributeFormDefault'

Das Attribut 'elementFormDefault' kann den Namensraumzwang für Elemente des Instanzdokuments lockern, 'qualified' gibt an, dass sich die Elemente des Dokuments immer im Zielnamensraum befinden

müssen, 'unqualified' hingegen erlaubt das Vorkommen von Elementen auch außerhalb dieses Namensraums. Der Defaultwert ist 'unqualified'. Das Attribut 'attributeFormDefault' ist analog zu 'elementFormDefault' für Attribute. Danach folgen dann die eigentlichen Element bzw. Attributdefinitionen und -deklarationen unterhalb des Wurzelelements. Elementdefinitionen werden durch das XSD-Element 'element' angezeigt. XML Schema bringt von sich aus schon zahlreiche vordefinierte Elementtypen mit und bietet darüber hinaus auch die Definition eigener Typen. Das folgende Beispiel zeigt eine einfache Deklaration eines `Elementes` mit dem Namen 'elementname' und dem Typen 'xsd:integer':

```
<xsd:element name="elementName" type="xsd:integer"/>"
```

Elemente können Eigenschaften haben, die über entsprechende Attribute definiert werden können. Dazu gehören unter anderem auch Elementname, Elementtyp und Häufigkeitsbeschränkungen ('Occurrence Constraints'). Die wichtigsten Elementeigenschaften sind:

- `name` Name des Elementes
- `type` Typzuordnung (vordefinierte oder selbstdefinierte Typen)
- `maxOccurs` maximale Auftretenshäufigkeit (Werte von '0' bis 'unbounded' möglich), Defaultwert ist '1'
- `minOccurs` minimales Auftretenshäufigkeit (als Werte alle positiven ganzen Zahlen) möglich, Defaultwert ist '1'
- `fixed` definiert einen konstanten Wert an
- `default` definiert einen Standarddefaultwert an
- `abstract` für abstrakte Typen, nur zur Strukturierung des Schemaentwurfs, als Basis von Spezialisierungen, Defaultwert ist 'false'
- `final` verbietet die Ableitung von diesem Element
- `id` eineindeutige schemaweite Kennzeichnung des Elements durch eindeutige Zeichenkette
- `nillable` erlaubt Null-Werte im Instanzdokument ('true' oder 'false'), Defaultwert 'false'
- `ref` Referenz auf andere Elementdeklaration zur Übernahme der dort spezifizierten Definitionen

Ähnlich wie in DTDs unterstützt XML-Schema die Inhaltsmodelle: unstrukturierter Inhalt (Zeichendaten), explizit benannte Kindelemente, beliebige Kindelemente, leeres Inhaltsmodell, gemischtes Inhaltsmodell, Alternativen, Sequenzen und Mixed Content.

Das `freie Inhaltsmodell`, das beliebigen Inhalt aus zugelassenen Elementen und Zeichendaten ermöglicht, wird durch die Angabe des Typs 'anyType' realisiert. Zur Darstellung des `leeren Inhaltsmodells` wird der ein leeres XSD-'complexType'-Element wie folgt verwendet:

```
<xsd:element name="elementName">
   <xsd:complexType/>
</xsd:element>
```

Das `Alternativen-Inhaltsmodell` wird durch 'xsd:choice' umgesetzt. Dabei werden zwischen dem Startag und dem Endetag von 'xsd:choice' die alternativen Dokumentstrukturen angegeben. Ein Beispielschemafragment, bei dem bei einer Adresse entweder Strasse und Postleitzahl oder ein Postfach angegeben wird:

```
<xsd:complexType name="adresse">
  <xsd:sequence>
    <xsd:element name="ort" type="ort"/>
      <xsd:choice>
        <xsd:sequence>
          <xsd:element name="plz" type="plz"/>
          <xsd:element name="strasse" type="strasse"/>
        </xsd:sequence>
        <xsd:element name="postfach" type="postfach"/>
      </xsd:choice>
    <xsd:element name="adresszusatz" type="adresszusatz" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

Im obigen Beispiel war auch schon die Realisierung von `Sequenzen` zu erkennen, die durch 'xsd:sequence' definiert werden. Dazu werden die Elemente in ihrer Reihenfolge zwischen dem Start- und dem Endetag von 'xsd:sequence' angegeben. Des Weiteren gibt es noch das XSD-Element 'xsd:all' das Elementsequenzen definiert, die in beliebiger Reihenfolge auftreten dürfen. Es sei noch mal darauf hingewiesen, dass bei Angabe des Attributs ' mixed='true' ' in '<xsd:complexType ...>' auch möglich ist, `Mixed Content` (Zeichendaten mit Kindelementen gemischt) ähnlich wie in DTDs anzugeben. Wobei aber beachtet werden muss, dass bei Mixed Content in XML-Schema die Elementsequenzen anders behandelt werden als in DTDs. Denn in den XML-Dokumenteninstanzen müssen Elementsequenzen bei XML-Schema wirklich in der Reihenfolge auftreten, in der sie deklariert wurden.

3.2.5 Attributdefinition

`Attribute` werden durch das XSD-Element 'xsd:attribute' deklariert und durch die Schachtelungsstruktur den jeweiligen Elementen zugeordnet. Attributdeklarationen beinhalten einen Namen ('name') und einen einfachen vordefinierten oder selbstdefinierten Typen ('type'). Weiterhin besitzen sie eine Angabe ('use'), wie das Attribut verwendet werden soll. Dabei sind folgende Werte möglich 'required' für notwendige bzw. geforderte Attribute, 'optional' für optionale Attribute, 'prohibited' für verboten. Zusätzlich kann man entweder mit der Angabe 'fixed' einen Fixwert definieren, oder mit 'default' Standarddefaultwert festlegen. Eine Übersicht über die Möglichkeiten des XML-Schemaelementes 'xsd:attribute' zeigt folgende Definition [XMLSa04]Kapitel 3.2.2:

```
<attribute
  default = string
  fixed = string
  form = (qualified | unqualified)
  id = ID
  name = NCName
  ref = QName
  type = QName
  use = (optional | prohibited | required) : optional
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, simpleType?)
</attribute>
```

3.2.6 Integritätskonzept

Ein interessantes Konzept ist das `Integritätskonzept` von XML-Schema, das über das von DTD hinausgeht. Es können damit Eindeutigkeitsbedingungen definiert werden, aber auch Schlüsselbedingungen und Schlüsselreferenzen angegeben werden. Diese sind vergleichbar mit den Konzepten aus dem Datenbankbereich.

```
<xsd:key name="hotel_key">
    <xsd:selector xpath="/hotels/hotel"/>
    <xsd:field xpath="@hotelID"/>
</xsd:key>

<xsd:keyref name="hotelcatalogue_keyref" refer="hotel_key">
    <xsd:selector xpath="/hotelcatalogue/countrys/hotels"/>
    <xsd:field xpath="@hID"/>
</xsd:keyref>
```

Wie im Beispiel zu erkennen, erfolgt die Angabe des Schlüssel über das Schemaelement 'xsd:key', das über das Attribut 'name' benannt werden muss. Mittels XPath-Ausdrücken im Schemaelement 'xsd:selector' wird das Element bzw. die Elementmenge ausgewählt, die als Schlüssel agieren soll. Mit 'xsd:field' wird relativ zur 'selector'-Elementdefinition angegeben, auf welchem Elementwert oder Attribut die Schlüsseleigenschaften Eindeutigkeit und „Nicht-NULL-barkeit“ gelten soll. Es sei noch darauf hingewiesen, dass auf jeden Fall immer die Elemente 'xsd:selector' und 'xsd:field' deklariert werden müssen, auch in der angegebenen Reihenfolge. Dabei können im Gegensatz zum 'selector'-Element mehrere 'field'-Elemente angegeben werden. Analog im Aufbau und Struktur zu 'xsd:key' gibt es das 'xsd:unique'-Element, das nur Eindeutigkeit auf den spezifizierten Teilen fordert.

Mit 'xsd:keyref' kann man dann wie im obigen Beispiel mittels dem 'refer'-Attribut auf definierte Schlüssel verweisen. Dies erfolgt über die Angabe des Namens des Schlüssel im 'refer'-Attribut. Auch hier wird analog zum 'key'-Element mit XPath-Ausdrücken zur Auswahl der Elemente bzw. Attribute gearbeitet.

3.2.7 Modellierungsstile

In XML-Schema ist es möglich Typen sowohl global als auch lokal zu definieren. Der Vorteil der `globalen Definition` der Typen ist, dass sie benannt werden und somit einen Bezeichner besitzen. Dies ermöglicht die Wiederverwendung von Typdefinitionen. Anders ist das dagegen bei `lokaler Typdefinition`. Bei dieser werden die Typen, direkt in den dazugehörigen Elementen oder Attributen beschrieben und werden daher nicht extra benannt. Deshalb sind diese Typdefinitionen aufgrund mangelnder Bezeichner nicht wieder verwendbar. Des Weiteren kann man zur (Wieder-) Verwendung entweder Referenzen auf globale Element- und Attributdeklarationen einsetzen, oder in Element- und Attributdeklarationen auf globale Typen verweisen.

Daraus lassen sich drei Modellierungsstile ableiten. Der erste Modellierungsstil ist der `Russian Doll (Matrjoschka) Design`, der mit lokalen Typdefinitionen arbeitet. Dadurch entsteht selbst bei kleineren XML-Dokumentbeschreibungen schnell eine tiefe Schachtelung. Es entsteht im Vergleich zu anderen Stilen oft auch mehr Code, da keine Wiederverwendung von Typdefinitionen und Deklarationen möglich ist.

```
<xsd:element name="hotelname" type="xsd:string"/>
<xsd:element name="adresse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="plz">
        <xsd:simpleType>
          <xsd:restriction base="xsd:nonNegativeInteger">
            <xsd:totalDigits value="5" fixed="true"/>
          </xsd:restriction>
       </xsd:simpleType>
      </xsd:element>
      <xsd:element name="ort" type="xsd:string"/>
      <xsd:element name="strasse" type="xsd:string"/>
      <xsd:element name="hausnummer">
       <xsd:simpleType>
          <xsd:restriction base="xsd:nonNegativeInteger">
            <xsd:totalDigits value="4"/>
          </xsd:restriction>
       </xsd:simpleType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Der zweite Stil wird `Salami Slice Design` genannt. Bei ihm erfolgen die Element- und Attributdeklarationen global. Die Wiederverwendung wird hier durch das 'ref'-Attribut der jeweiligen Elemente realisiert. Das 'ref'-Attribut verweist dann also auf die globale Element- oder Attributdeklaration.

```
<xsd:element name="hotelname" type="xsd:string"/>
<xsd:element name="plz">
  <xsd:simpleType>
    <xsd:restriction base="xsd:nonNegativeInteger">
      <xsd:totalDigits value="5" fixed="true"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
<xsd:element name="ort" type="xsd:string"/>
<xsd:element name="strasse" type="xsd:string"/>
<xsd:element name="hausnummer">
  <xsd:simpleType>
    <xsd:restriction base="xsd:nonNegativeInteger">
      <xsd:totalDigits value="4"/>
```

```
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>

  <xsd:element name="adresse">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="plz"/>
        <xsd:element ref="ort"/>
        <xsd:element ref="strasse"/>
        <xsd:element ref="hausnummer"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
```

Schließlich gibt es noch den sogenannten `Venetian Blind Design`. Er ist hinsichtlich der Wiederverwendung von Code die beste Variante. Der Venetian Blind Design nutzt globale Typdefinitionen, und legt damit die Basis für Typerweiterungen und Mehrfachnutzung.

```
  <xsd:simpleType name="hotelname_type">
    <xsd:restriction base="xsd:string"/>
  </xsd:simpleType>

  <xsd:simpleType name="plz_type">
    <xsd:restriction base="xsd:nonNegativeInteger">
      <xsd:totalDigits value="5" fixed="true"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="ort_type">
    <xsd:restriction base="xsd:string"/>
  </xsd:simpleType>

  <xsd:simpleType name="strasse_type">
    <xsd:restriction base="xsd:string"/>
  </xsd:simpleType>

  <xsd:simpleType name="hausnummer_type">
    <xsd:restriction base="xsd:nonNegativeInteger">
      <xsd:totalDigits value="4"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:complexType name="adresse_type">
    <xsd:sequence>
      <xsd:element name="plz" type="plz_type"/>
      <xsd:element name="ort" type="ort_type"/>
      <xsd:element name="strasse" type="strasse_type"/>
      <xsd:element name="hausnummer" type="hausnummer_type"/>
    </xsd:sequence>
  </xsd:complexType>
```

3.3 Schematron

Ein ganz anderer Ansatz, als die zuvor beschriebenen Schemasprachen, ist die Schemasprache Schematron. `Schematron` wurde zunächst in einem persönlichen Projekt von Rick Jelliffe entwickelt. Mit der Zeit beteiligten sich immer mehr und mehr Entwickler an dem Projekt. Inzwischen ist Schematron seit Mai 2006 offizieller ISO-Standard. Schematron ist eine in XML-Struktur gehaltene Sprache zum Validieren von XML-Dokumenten mittels XSLT. Schematron versteht sich selbst dabei als eine Ergänzung zu den zuvor angesprochenen grammatikbasierten Schemasprachen wie DTD und XML-Schema. Schematron

benutzt XPath-Ausdrücke, um Behauptungen (assert) aufzustellen und Beschreibungen von Fehlerzuständen (reports) anzugeben. Dieses Regelwerk aus positiven und negativen Bedingungen bildet das Schema in Schematron. Rick Jelliffe verfolgte bei der Entwicklung von Schematron mehrere Designziele [HACK], [NOE]:

- Beschreibungen der Fehler in natürlicher Sprache
- Diagnose der Fehler in Dokumenten statt einfacher Zurückweisung
- leicht erlernbar durch Verwendung vorhandener Hilfsmittel (XPath und XSLT) und nur weniger Beschreibungselemente
- einfache Implementierung durch Aufsetzen auf XSLT
- Entwicklung einer Architektur, die GUI-Entwicklungen unterstützt

Das folgende Beispiel zeigt ein XML-Fragment und ein minimales Schematron-Schema:

```
<aktiengesellschaft>
  <name>XYZ - AG</name>
  <teilhaber anteil="55">
    <name>Karl Muster</name>
  </teilhaber>
  <teilhaber anteil="20">
    <name> Firma ABC</name>
  </teilhaber>
  <teilhaber anteil="25">
    <name>EFG-Invest</name>
  </teilhaber>
</aktiengesellschaft>

<schema>
  <pattern name="Anteilsverteilung">
    <rule context="aktiengesellschaft">
      <assert test="sum(teilhaber/@anteil) = 100">
        Die Summe aller Anteile muss 100 ergeben!
      </assert>
    </rule>
  </pattern>
</schema>
```

Im XML-Fragment wird die 'XYZ-AG' beschrieben, die mehrere Teilhaber mit unterschiedlichen Anteilsmengen hat. Im Schema wird eine Regel definiert, die überprüft, ob die Summe der Anteile der Teilhaber gleich 100 ist.

3.3.1 Funktionsweise

Schematron arbeitet in erster Linie musterbasiert und regelorientiert. Das bedeutet, dass bei Regelverletzungen oder bei Erkennung von bestimmten Mustern Fehlermeldungen erzeugt werden. Diese weisen den User auf einen Fehler, ein bestimmten meist nicht gewünschten Zustand oder ein (Fehler-)Muster hin. Diese Fehlermeldungen und Hinweise werden in einem Report gesammelt, der am Ende dann also alle Fehlermeldungen enthält. Ist ein XML-Dokument also fehlerfrei bezüglich eines Schematron-Schema, so enthält der Fehlerreport nach dem Validierungsprozess keine Fehlermeldungen.

Schematron operiert wie bereits erwähnt auf der Grundlage von XSLT. Die `Validierung` eines XML-Dokumentes mit einem Schematron-Schema erfolgt in mehreren Schritten. Abbildung 3.4 zeigt den Validierungsprozess eines XML-Dokumentes mittels Schematron. Das Schematron-Schema wird in einem XSLT-Prozessor mit Hilfe von einem Meta-Stylesheet dem sogenannten Schematron-Präprozessor, in ein gültiges XSLT-Dokument (Validierungs-Stylesheet) umgewandelt. Der `Schematron-Präprozessor` wird meist mit 'schematron.xslt' bezeichnet und ist die Transformationsdatei, die die Regelvorlagen zur Umwandlung des Schematron-Schemas in das Validierungs-Stylesheet enthält. Diese Datei baut dabei zumeist auf der Vorlage von Rick Jelliffe auf und kann recht einfach erweitert bzw. verändert werden. Das entstandene `Validierungs-Stylesheet` wird dann auf das zu validierende XML-Dokument angewendet. Das

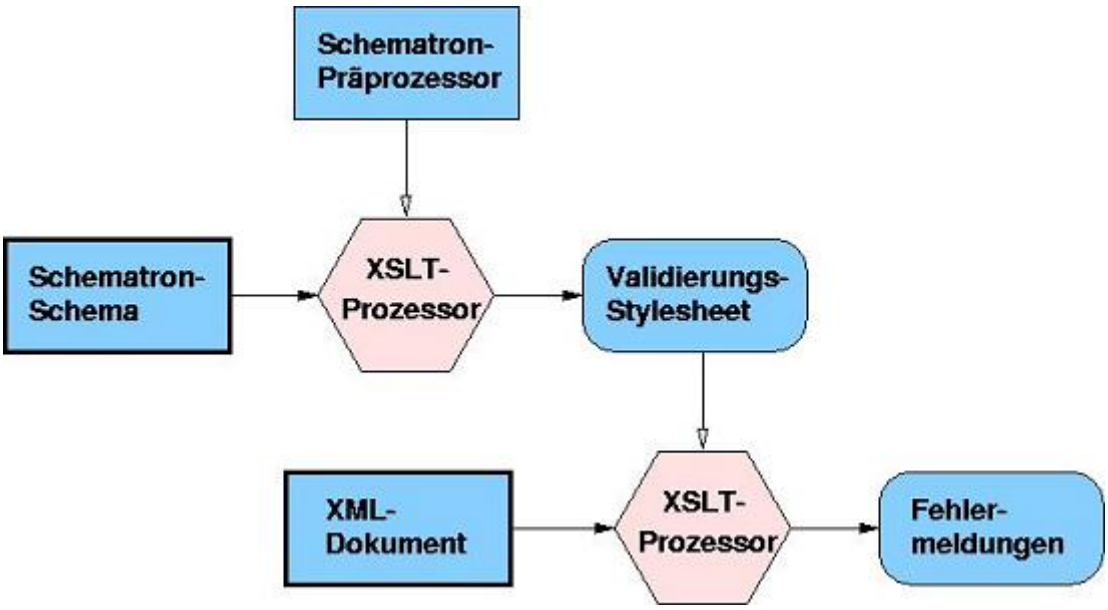


Abbildung 3.4: XML-Dokument-Validierungsprozess
[BECK02]

Resultat dieser XSL-Transformation kann je nach Validierungs-Stylesheet eine XML-, Text- oder auch HTML-Datei sein. Dieses Datei wird dann auch `Report` bzw. Fehlerreport genannt. Dieser zweistufige Validierungsprozess besitzt das Problem, dass die Fehlermeldungen keinen Hinweis darauf enthalten, wo der Fehler im XML-Dokument auftritt. Deshalb wird empfohlen, den Prozess zu daraufhin zu erweitern, dass Verweise auf die XML-Quelle im Schema-Report angegeben werden. Dazu wird der Validierungsprozess leicht modifiziert [BECK02]:

- Erzeugen eines HTML-Abbildes des XML-Dokumentes mit internen Verweiszielen
- Formatierung der Fehlermeldungen als HTML-Datei mit entsprechenden Verweisen in das HTML-Abbild
- Anzeige beider HTML-Dateien (Fehlerreport, Abbild XML-Dokumentes) innerhalb von Frames

Abbildung 3.5 zeigt einen Schematron Report in HTML-Frames. Der obere Frame zeigt die Fehlermeldungen zu den im XML-Dokument gefundenen Fehlern. Bei Klick auf eine Fehlermeldung „springt“ der untere Frame auf die Stelle des HTML-Abbildes des XML-Dokumentes, wo der Fehler auftritt. Im Beispiel wird bei Klick auf die Fehlermeldung „Das Attribut ref ist für Zutaten in der Zutatenliste nicht erlaubt.“ auf die fehlerhafte Stelle im XML-Dokument `<zutat ref="wasser" menge="viel">Wasser</zutat>` verwiesen.

3.3.2 Struktur

Um dem Ziel der Einfachheit bzw. leichten Erlernbarkeit nachzukommen, beschränkt sich Schematron auf einige wenige Grundelemente. Folgende Übersicht zeigt die Grundstruktur und die wichtigsten Beschreibungselemente von Schematron (nach [STRONe04], [BECK02]):

```
- <schema>
  - <title>?
  - <ns>*
  - <phase>*
    - <active>+
  - <pattern>+
    - <rule>+
       - <extends>*
       - (<assert> | <report>)+
  - <diagnostics>?
    - <diagnostic>+
```

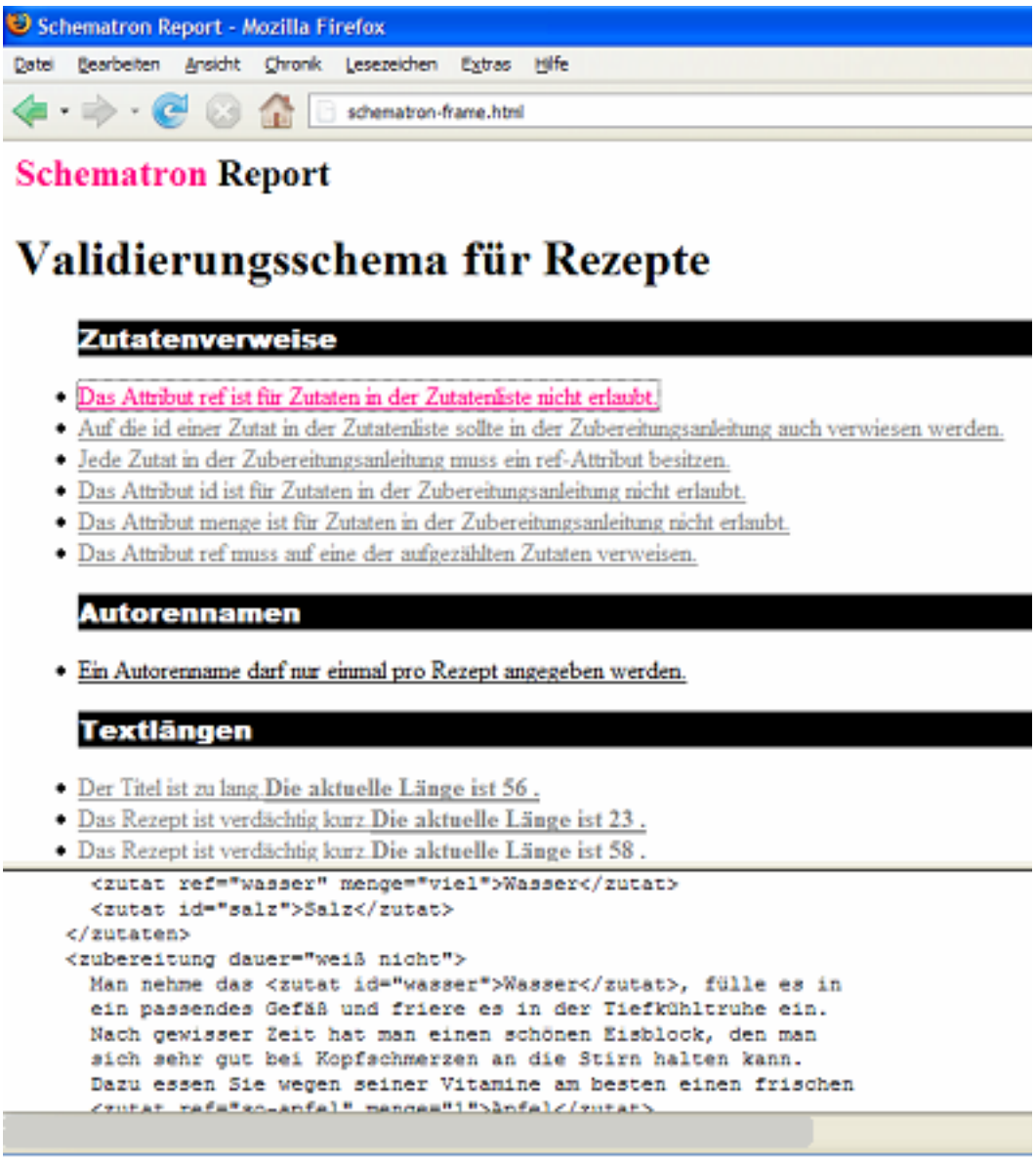


Abbildung 3.5: Schematron Report in HTML-Frames
[BECK02]

```
Legende:
? ... optionales Element (0-1 mal)
+ ... Element tritt mindestens 1 mal bis beliebig oft auf
* ... Element tritt 0 mal bis beliebig oft auf
| ... Alternativen
```

Das folgende Schematron-Schema zeigt zudem ein kompaktes Beispiel, in dem die Schematronbeschreibungselemente aufgezeigt werden sollen:

```
<schema xmlns="http://www.ascc.net/xml/schematron" >
    <title>Simple Example</title>
    <phase id="print all">
        <active pattern="print positive result">
        <active pattern="print negative result">
    </phase>
    <phase id="print only positiv results">
        <active pattern="print positive result">
    </phase>
    <pattern  name="positive results" id="print positive results">
```

```
        <rule context="AAA">
            <report test="BBB">BBB element is present.</report>
            <report test="@name">AAA contains attribute name.</report>
            <report test="string-length(@name) >= 2" diagnostics="namelength">
             Attribute name has a length of minimum 2.
            </report>
        </rule>
    </pattern>
    <pattern name="negative results" id="print negative results">
        <rule context="AAA">
            <assert test="BBB">BBB element is missing.</assert>
            <assert test="@name">AAA misses attribute name.</assert>
            <assert test="string-length(@name) >= 2" diagnostics="namelength">
                Attribute name has NOT a length of minimum 2.
            </assert>
        </rule>
    </pattern>
    <diagnostics>
        <diagnostic id="namelength">
            The current length is <value-of select="string-length(@name)"/>.
        </diagnostic>
    </diagnostics>
</schema>
```

Das 'schema'-Element ist, wie aus der obigen Strukturbeschreibung ersichtlich, das Wurzelelement eines jeden Schematron-Schema. Optional kann das Element ein 'schemaVersion'-Attribute besitzen, das die Version des Schemas angibt. Ein weiteres optionales Attribut ist das 'defaultPhase'-Attribut. Im 'defaultPhase'-Attribut kann man spezifizieren, welche Phase aktiv wird, wenn keine Benutzerangabe hierzu gemacht wird. Auf das 'schema'-Element kann optional ein 'title'-Element folgen, das wie der Name schon sagt, den Titel des Schemas definiert. Des Weiteren können danach mehrere Namensraumangaben über das 'ns'-Element erfolgen.

In den 'phase'-Elementen kann man mehrere 'active'-Elemente angeben, die die Pattern angeben, die überprüft werden sollen. Es lassen sich mehrere Phasen spezifizieren, die so verschiedene Variationen von aktiven Pattern definieren. Das bedeutet, dass man im Endeffekt bestimmte Pattern und damit Regeln nach Bedarf aktivieren bzw. deaktivieren kann. Die Definition der Pattern die aktiviert werden sollen erfolgt im 'active'-Element über das geforderte Attribut 'pattern'. Als Wert wird die ID des Pattern angegeben, das aktiviert werden soll.

Darauf folgt die Definition der Pattern, von denen mindestens ein Element auftreten muss. Die wichtigsten Attribute des 'pattern'-Elementes sind 'name' und 'id'. Während 'name' den im Fehlerreport sichtbaren Namen des Pattern beschreibt, definiert 'id' eine im Schema eindeutige Bezeichnung des Pattern. Diese wird z.B. für die Aktivierung von Pattern in 'active'-Elementen verwendet. Pattern enthalten mindestens eine Regel ('rule'). Regeln enthalten positive und negative Gültigkeitsmuster ('report' und 'assert'). Das wichtigste Attribut von 'rule' ist 'context', mit dem über XPath-Ausdrücke angezeigt wird, welche Elemente oder Elementmengen bzw. Attribute oder Attributmengen getestet werden sollen. Reports beschreiben Fehlerzustände. Wenn also der im 'report'-Element über das Attribut 'test' spezifizierte XPath-Ausdruck wahr wird, wird eine Fehlermeldung im Fehlerreport erzeugt. Diese Fehlermeldung ist in natürlicher Sprache frei definierbar und wird zwischen Anfang- und Endetag des 'report'-Elementes angegeben. Über das Attribut 'diagnostics' in 'report' lässt sich zu umfangreicheren Fehlerbeschreibungen bzw. -analysen verweisen. Dazu müssen als Werte des 'diagnostics'-Attributes die durch Leerzeichen getrennten IDs der entsprechenden 'diagnostic'-Elemente des 'diagnostics'-Teiles angegeben werden. Analog zum 'report'-Element lässt sich das 'assert'-Element beschreiben. Es hat denselben Aufbau und dieselbe Funktion. Der Unterschied zum 'report'-Element liegt darin, dass 'assert' nicht Fehlerzustände beschreibt, sondern Zusicherungen mittels XPath-Ausdrücken formuliert. Werden diese Zusicherungen verletzt, wird ein Fehlereintrag im Fehlerreport erzeugt.

Der optionale Diagnoseteil wird eingeleitet durch das Element 'diagnostics'. Das 'diagnostics'-Element kann mehrere 'diagnostic'-Kindelemente besitzen. In diesem wiederum erfolgt eine Analyse bzw. eine ausführliche Beschreibung eines Fehlers. Das notwendige Attribut 'id' des 'diagnostic'-Elementes identifiziert die Beschreibung des Fehlers. Auf diese IDs verweisen die entsprechenden Reports

oder Assertions, die eine genauere Analyse des Fehlers unterstützten. Im obigen Beispiel wird in einer Zusicherung (Assertion) getestet ob das Attribut 'name' des Elementes 'AAA' mindestens eine Länge von 2 Zeichen hat. Ist dies nicht der Fall wird die spezifizierte Fehlermeldung „Attribute name has NOT a length of minimum 2." im Fehlerreport erzeugt. Des Weiteren aber wird im Attribut 'diagnostics' der Zusicherung auf das Diagnoseelement mit der ID 'namelength' verwiesen. Dieses erzeugt die zusätzliche Information über die aktuelle Länge des Attributwertes von 'name'. Über das Attribut 'xml:lang' von 'diagnostic' lassen sich auch speziell für mehrere Sprachen verschiedene Fehleranalysen definieren. Dazu ein Beispiel aus dem Anhang der ISO-Schematronspezifikation [STRON04]:

```
<sch:schema  xmlns:sch="http://purl.oclc.org/dsdl/schematron" xml:lang="en" >
    <sch:title>Example of Multi-Lingual Schema</sch:title>
    <sch:pattern>
        <sch:rule context="dog">
            <sch:assert test="bone" diagnostics="d1 d2">
                A dog should have a bone.
            </sch:assert>
        </sch:rule>
    </sch:pattern>

    <sch:diagnostics>
        <sch:diagnostic id="d1" xml:lang="en">
            A dog should have a bone.
        </sch:diagnostic>
        <sch:diagnostic id="d2" xml:lang="de">
            Das Hund muss ein Bein haben.
        </sch:diagnostic>
    </sch:diagnostics>
</sch:schema>
```

Eine komplette Übersicht über die Schematron-Elemente und deren Attribute findet man im Kapitel 5 von [STRON04] und in der Quick-Referenz-DTD Beschreibung [STRONq04].

Kapitel 4

Vergleich der Schemasprachen

Im folgenden Kapitel soll ein Vergleich der drei zuvor vorgestellten Schemasprachen vorgenommen werden. Dieses Kapitel orientiert sich hierbei stark an [DLWC00]. Damit sollen die Gemeinsamkeiten und Unterschiede dieser Sprachen festgestellt werden. Dabei lassen sich die Sprachen nach mehreren Gesichtspunkten vergleichen. Dazu gehören Vergleichsmöglichkeiten nach unterstützten Features, Erlernbarkeit & Benutzbarkeit, Spracheigenschaften, Datenbankeigenschaften und Ausdruckstärke.

4.1 Erlernbarkeit & Benutzbarkeit

Aus Sicht der Handhabbarkeit ist die DTD wohl die beste Sprache. Sie ist sehr kompakt, beschränkt sich auf wenige Beschreibungselemente und unterstützt nur wenige Datentypen. Durch die eingeschränkten und übersichtlichen Beschreibungsmöglichkeiten der DTD, ist sie relativ leicht zu erlernen und gut menschenlesbar. Der einzige Nachteil auf diesem Gebiet ist, dass DTDs nicht in XML-Syntax notiert werden. Schematron beschränkt sich ebenfalls auf wenige Beschreibungselemente, setzt jedoch mit XPath eine weitere Sprache als bekannt voraus, was durchaus nicht immer der Fall ist. Wegen der vielen Beschreibungsmöglichkeiten und zahlreichen Datentypen ist XML-Schema schwieriger erlernbar und handhabbar. Selbst XML-Schemata für kleine Dokumentbeschreibungen können mitunter schnell sehr umfangreich werden. Daher ist XML-Schema auch weniger gut menschenlesbar.

4.2 Spracheigenschaften

Aus Sprachsicht kann man die Schemasprachen anhand einiger Sprachfaktoren unterscheiden. Während XML-Schema und DTD strukturorientiert, grammatikbasiert und definitionsorientiert sind, ist Schematron dagegen regelorientiert, musterbasiert, und validationsorientiert. Abbildung 4.1 zeigt die Einordnung der Schemasprachen in die drei Kategorien strukturorientiert vs. regelorientiert, grammatikbasiert vs. musterbasiert und definitionsorientiert vs. gebrauchsorientiert. Die grammatikbasierten Schemasprachen haben durch die in ihnen enthaltenen Strukturinformation Vorteile bei der Optimierung von XML-Anfragen. Schematron hingegen ist hinsichtlich der Ausdruckskraft der Beschränkungen umfangreicher.

4.3 Unterstützung von Datenbankeigenschaften

Weil eine Menge von XML-Dokumenten aus Datenbankinhalten generiert werden, ist es notwendig, dass Schemasprachen nach Möglichkeit auch datenzentrierte Eigenschaften wie in Datenbankdefinitionssprachen unterstützen. Diese Eigenschaften sind im Wesentlichen:

- Relationen und Attribute
- Definitionsbereiche, Wertevorräte
- Integritätsbedingungen
- Indizes
- Datensicherheitsbestimmungen

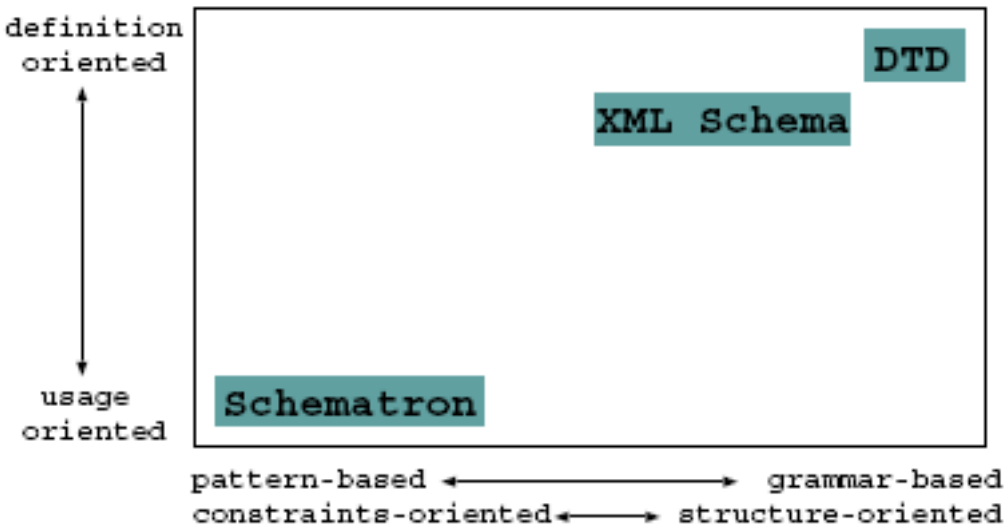


Abbildung 4.1: Klassifikation von XML-Schemasprachen
[DLWC00]

Im Vergleich der Schema-Sprachen zu den SQL-DDLs fällt auf, dass keine Sprache alle Features dieser SQL-DDLs zur Verfügung stellen kann. Während DTD schon Namensräume (Definitionsbereiche), Datentypen und Integritätsbedingungen nicht vollständig darstellen kann, unterstützt XML-Schema zumindest eine Reihe von Built-In-Datentypen und Integritätsbedingung wie das Eindeutigkeits- bzw. Schlüsselkonzept. Was sich aber nicht darstellen lässt, sind z.B. `CHECK`-Klauseln. Mit Schematron lassen sich zwar auch diese Integritätsbedingungen darstellen, aber es lassen sich keine physischen Indizes zur Performanzsteigerung unterstützen.

4.4 Ausdrucksstärke der XML-Schemasprachen

Die Abbildung 4.2 zeigt die Klassifikation der Ausdrucksstärke von einer Auswahl von XML-Schemasprachen, wobei uns hier insbesondere die drei bisher betrachteten Schemasprachen interessieren.

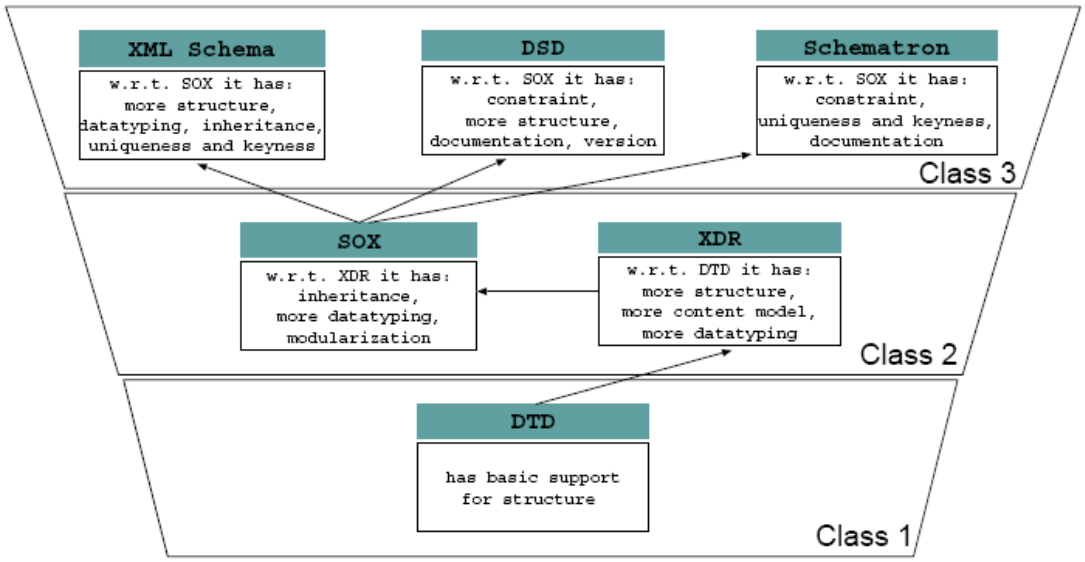


Abbildung 4.2: Klassifikation Ausdrucksstärke der XML-Schemasprachen
[DLWC00]

In der Klasse 1 der Schemasprachen finden wir die DTD wieder. Sie hat im Vergleich die schwächste Ausdrucksstärke der Schemasprachen und unterstützt nur grundlegende Strukturbeschreibungen und wenige

Datentypen und Beschränkungselemente. Wesentlich ausdrucksstärker in der 3. Klasse ist XML-Schema. Es hat im Vergleich zu DTD mehr Strukturelemente und Inhaltsmodelle. Es unterstützt des Weiteren das Datentypsystem komplett und erlaubt sogar Vererbung und Modularisierung. Darüber hinaus bietet XML-Schema auch ein ausgereiftes Eindeutigkeits- und Schlüsselkonzept. Ebenfalls in der 3. Klasse der obigen Klassifikation ist Schematron. Schematrons Ausdruckskraft geht noch über die von XML-Schema hinaus. Schematron unterstützt nämlich eine sehr flexible Mustererkennungssprache, die auch detaillierte Beschreibung von der Semantik eines Schemas zulässt. Dies ist mit XML-Schema bzw. DTD nicht möglich. Es sei noch mal auf das Beispiel der 'XYZ-AG' im Kapitel 3.3 verwiesen. Eine solche Definition, dass die Summe der Anteile gleich 100 sein müssen, kann man weder mit DTD noch mit XML-Schema beschreiben. Auf der anderen Seite ist Schematron weniger für Strukturbeschreibungen geeignet und auch nicht dafür konzipiert worden. Diese sind theoretisch zwar darstellbar, müssen aber umständlich simuliert werden.

4.5 Zusammenfassung

Tabelle 4.1 zeigt zusammenfassend eine vergleichende Tabelle der Eigenschaften der drei beschriebenen Schemasprachen. Im ersten Abschnitt der Tabelle werden Eigenschaften bzgl. des Schemas verglichen. Während XML-Schema sowohl selbst in XML-Syntax notiert wird, Namensräume, Imports sowie Includes unterstützt, unterstützen Schematron und XML-DTD diese nur teilweise bzw. gar nicht. In den folgenden drei Teilen geht es um die Bereitstellung verschiedener Datentypen, Attribut- und Elementinhaltskonzepte, wie z.B. Built-In-Typen, Domänenbeschränkungen und die verschiedenen Inhaltsmodelle. In diesen Bereichen stellen sich XML-Schema und Schematron als die am meisten geeigneten Schemasprachen dar. Das Vererbungskonzept wird, wie in Teil 5 der Tabelle ersichtlich, in unterschiedlichen Ausprägungen nur von XML-Schema bereitgestellt. Auch bei der Unterstützung des Eindeutigkeits- und Schlüsselkonzepts tun sich insbesondere XML-Schema aber auch Schematron hervor, während DTD nur grundlegende Unterstützung bzgl. des Eindeutigkeitskonzept ermöglicht. Im letzten Teil der Tabelle wird noch auf verschiedene Besonderheiten eingegangen. So unterstützt keine der hier vorgestellten Schemasprachen eine Versionierung, wie sie z.B. die Document Structure Description DSD 1.0 ermöglicht. Dokumentation, d.h. in diesem Fall textuelle Beschreibungen der Schemafragmente, eingebettete Dokumentation für Anwendungsprogramme und Fehlermeldungen bzw. -hinweise für die Schemavalidierung, haben wiederum nur XML-Schema und Schematron.

Eigenschaften	DTD 1.0	XML Schema 1.0	Schematron 1.4
Schema			
Syntax in XML	Nein	Ja	Ja
Namensräume	Nein	Ja	Ja
Include	Nein	Ja	Nein
Import	Nein	Ja	Nein
Datentypen			
Built-In Typen	10	37	0
Benutzerdefinierte Typen	Nein	Ja	Nein
Domänenbeschränkung	Nein	Ja	Ja
Explizites „null“	Nein	Ja	Nein
Attribute			
Defaultwerte	Ja	Ja	Nein
Alternativen	Nein	Nein	Ja
optional vs. gefordert	Ja	Ja	Ja
Domänenbeschränkung	Teilweise	Ja	Ja
bedingte Definition	Nein	Nein	Ja
Elemente			
Defaultwerte	Nein	Teilweise	Nein
Inhaltsmodell	Ja	Ja	Ja
ungeordnete Reihenfolgen	Ja	Ja	Ja
geordnete Reihenfolgen	Nein	Ja	Ja
Alternativen	Ja	Ja	Ja
min. & max Auftreten	Teilweise	Ja	Ja
offenes Modell	Nein	Nein	Ja
bedingte Definition	Nein	Nein	Ja
Vererbung			
einfache Typen per Erweiterung	Nein	Nein	Nein
einfache Typen per Beschränkung	Nein	Ja	Nein
komplexe Typen per Erweiterung	Nein	Ja	Nein
komplexe Typen per Beschränkung	Nein	Ja	Nein
Eindeutigkeit & Schlüsseleigenschaften			
Eindeutigkeit für Attribute	Ja	Ja	Ja
Eindeutigkeit für Nicht-Attribute	Nein	Ja	Ja
Schlüssel für Attribute	Nein	Ja	Ja
Schlüssel für Nicht-Attribute	Nein	Ja	Ja
Fremdschlüssel für Attribute	Teilweise	Ja	Ja
Fremdschlüssel für Nicht-Attribute	Nein	Ja	Nein
Sonstiges			
dynamische Beschränkungen	Nein	Nein	Ja
Versionierung	Nein	Nein	Nein
Dokumentation	Nein	Ja	Ja
Einbettung von HTML	Nein	Ja	Teilweise
selbstbeschreibend	Nein	Teilweise	Teilweise

Tabelle 4.1: Zusammenfassender Eigenschaftsvergleich der XML-Schemasprachen

Kapitel 5

Model Driven Architecture

Die `Model Driven Architecture` MDA (Modellgetriebene Architektur) bzw. das `Model Driven Engineering` MDE (Modellgetriebene Entwicklung) ist eine Idee aus der Softwaretechnik. Die MDA stellt ein Framework für den Softwareentwicklungsprozess dar. Die MDA ist eine Entwicklung der OMG, die Modelle als zentrales Element des Softwareentwicklungsprozesses einsetzt. Zur Modellierung wird auf Empfehlung der OMG vornehmlich die UML (Unified Modeling Language) zusammen mit OCL (Object Constraint Language) verwendet. In diesem Abschnitt soll die MDA kurz vorgestellt werden, da sie nachfolgend die grundlegende Idee für den modellgetriebenen Entwurf von XML-Schemata ist.

5.1 Einführung MDA

In der Softwaretechnik ist die Entwicklung von guter Software zu einer zunehmend großen Herausforderung geworden. Insbesondere bei hoher Komplexität der Produkte wird es schwierig Probleme zu händeln. Um Quellcode beherrschen zu können, sind ab einer bestimmten Projektgröße höhere Dokumentationsformen unumgänglich, um die Kommunikation mit und zwischen Menschen zu gewährleisten. Als Problemlösung werden hier verschiedene Ansätze, wie Abstraktion oder Separation von Teilproblemen, angestrebt. Der modellgetriebene Ansatz zur Softwareentwicklung benutzt diese Prinzipien, um die Qualität und Produktivität der Softwareentwicklung zu erhöhen. Dieses wird dadurch erreicht, dass die wichtigsten Aspekte der zu entwickelnden Software in einer gut menschenlesbaren Form beschrieben werden [SPBG]. Dabei soll die Beschreibung zugleich aber auch möglichst formal sein. Dazu bieten Modelle eine optimale Möglichkeit. Sie sind in der Regel relativ leicht verständlich und schnell zu erfassen, bieten aber bereits zu Beginn der Softwareentwicklung einen hohen Formalisierungsgrad. Abschließend seien die Hauptziele (Potentiale) der MDA noch einmal zusammengefasst [SCHN04]:

- Entwicklungs-Performance: Automation durch Formalisierung (Metapher: Produktionsstraßen im Automobilbau)
- Software-Qualität
- Wiederverwendbarkeit
- Handhabbarkeit von Komplexität durch Abstraktion
- Wartbarkeit durch Trennung von Verantwortlichkeiten (Separation of Concerns)
- Handhabbarkeit von Technologiewandel
- Interoperabilität durch Standardisierung

5.2 Architektur

MDA ist ausgerichtet auf die Trennung der Spezifikation der Funktionalität der zu entwickelnden Software und der Implementierung dieser Spezifikation auf einer bestimmten Plattform. Der Fokus der MDA liegt auf der konzeptionellen Ebene. Dazu sollen Entwurfsentscheidungen für ein System getroffen werden. Auf Basis dieser Entscheidungen können dann Codeschablonen bzw. -rahmen generiert werden. Hierfür hat die OMG eine Architekturvorschlag unterbreitet, der sprachen-, kunden- und middlewareneutral ist (Abb. 5.1) [OMG07]. Im inneren Kreis stehen die wichtigsten und präferierten Technologien für die Modellierung von Softwaresystemen mittels MDA. Diese sind die ebenfalls von der OMG entwickelten

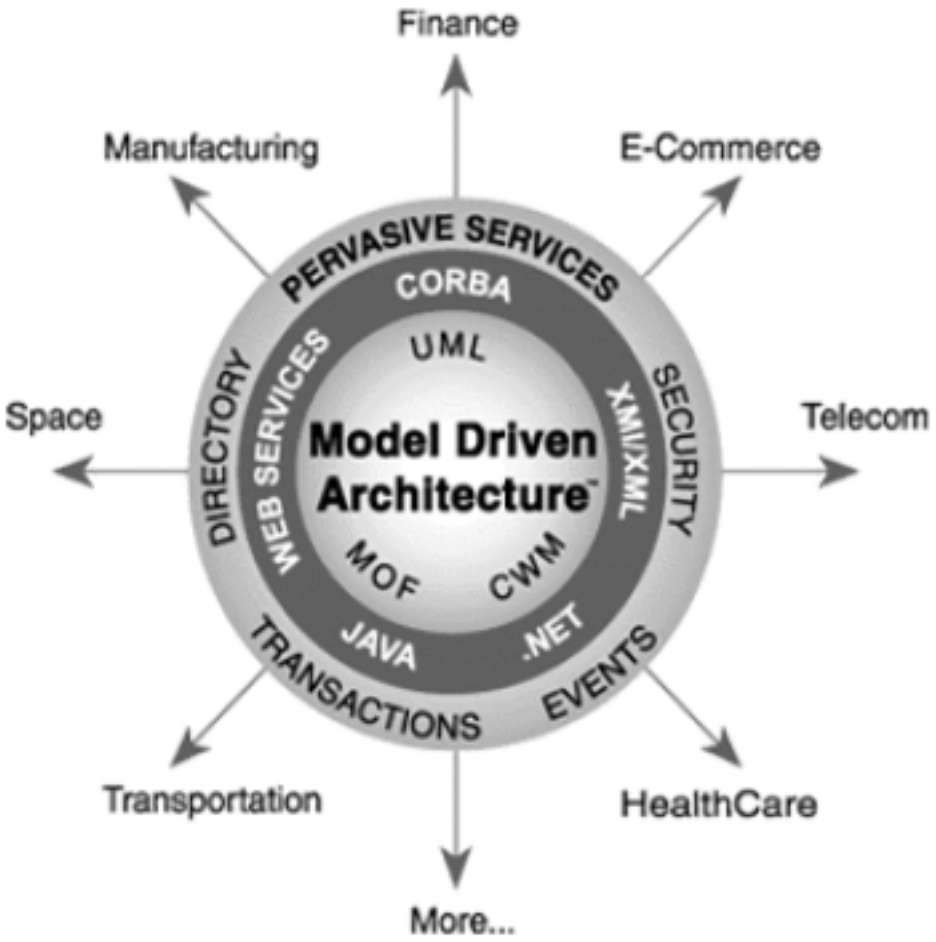


Abbildung 5.1: OMG's Model Driven Architecture
[OMG07]

Spezifikationen der `Unified Modeling Language` (UML) [UML], der `Meta Object Facility` (MOF) [MOF] und der `Common Warehouse Model` (CWM) [CWM]. UML ist die weit verbreitete und wohl bekannte Modellierungssprache zur Darstellung von Struktur und Verhalten von Systemen. Sie bietet eine Reihe verschiedener Diagrammarten wie z.B. Klassen-, Komponenten-, Aktivitäts-, Anwendungsfall- und Sequenzdiagramm. Durch die Fähigkeit, gut von bestimmten Systemen abstrahieren zu können, wird sie als Modellierungssprache für MDA insbesondere für PIMs empfohlen. Die MOF wurde entwickelt um Metamodelle beschreiben und in einander überführen zu können. Dazu wird mit dem `4-Schichtenmodell der OMG` (Abb. 5.2) eine Modellhierarchie beschrieben, die schrittweise die verschiedenen Modellebenen definiert. Im Regelfall modelliert der Anwender weitestgehend auf der Modell-Schicht M1, während auf Ebene M0 die Instanzen, also konkrete Objekte zur Laufzeit, beschrieben werden. Die in M1 definierten Modelle und ihre Modellelemente werden vom UML-Metamodell auf M2 beschrieben. Auf Ebene M3 hingegen ist die Basis für alle Metamodelle angesiedelt. Es ist also ein Metametamodell. Durch die MOF werden auf dieser Schicht die Sprachelemente zur Spezifikation von Metamodellen spezifiziert (z. B. ModelElement, Namespace, Classifier, etc.). Die UML lässt sich mittels eines MOF-konformen Metamodells vollständig beschreiben. Das bringt die Möglichkeit mit sich UML-Modelle mit Hilfe von MOF-QVT in andere MOF-kompatible Modelle zu überführen. CWM ist ein Metamodell der OMG für den Bereich Data Warehouse. Eine weitere wichtige Komponente der MDA ist die textuelle Beschreibung von Bedingungen (Constraints) an Objekte mittels der `Object Constraint Language` OCL. Dieses ist notwendig, da bestimmte Spezifikationen bzw. Einschränkungen nicht oder nur unzureichend graphisch dargestellt werden können.

Im mittleren Ring von Abbildung 5.1 werden die am häufigsten verwendeten Zielplattformen „Java“, „CORBA“, „.NET“ usw. aufgeführt, wobei auch durchaus andere möglich sind. Diese in Ringen getrennte Darstellung widerspiegelt die Idee der MDA mit der Trennung der Spezifikation von einer bestimmten Plattform.

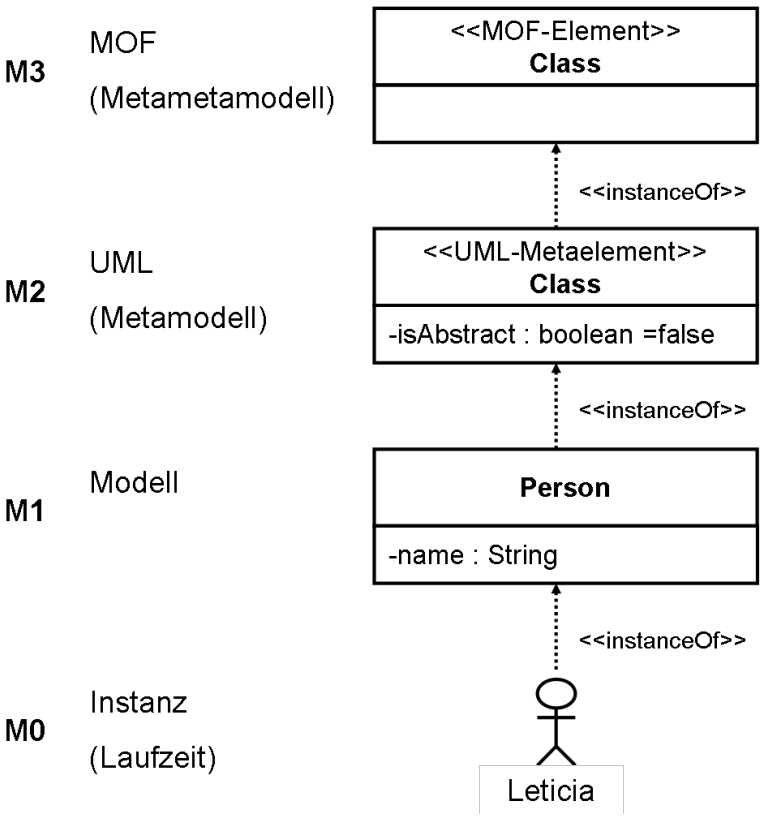


Abbildung 5.2: (Meta-) Modellebenen am Bsp. [PEME06]

Im äußeren Ring sind die so genannten „Pervasive Services“ abgebildet. Diese stellen grundlegende Dienste wie z.B. Sicherheit und Transaktionen dar, die bei vielen Plattformen benötigt werden. Die Vektoren am äußeren Ring zeigen dann die vielfältig möglichen Einsatzgebiete für MDA an. Für diese gibt es schon gewisse in OMG IDL definierte Standardschnittstellen.

5.3 MDA-Konzepte

In diesem Abschnitt sollen kurz die wichtigsten Komponenten der MDA vorgestellt werden. Zu diesen grundlegenden Konzepten gehören [SCHN04]:

- Modelle
- PIM, PSM, CIM
- Plattform
- UML-Profile
- Metamodelle (MOF)
- Transformationen

Wie bereits in Abschnitt 5.1 erwähnt, sind `Modelle` das grundlegende Konzept der MDA. Ein Modell stellt eine abstrakte Repräsentation von Aufbau, Struktur, Funktion und Verhalten eines Systems dar. Dabei wird in der MDA besonderes Augenmerk darauf gelegt, dass diese Modelle möglichst formal sind. Der hohe Grad an Formalisierung wird deshalb gefordert, damit die Modelle eindeutig beschrieben werden können. Das wiederum gewährleistet eine automatische Weiterverarbeitungsmöglichkeit mittels entsprechender Tools. Des Weiteren lassen sich so gewisse Modelle auch besser wiederverwenden. Als Modellierungssprache wird, wie bereits erwähnt, die UML wegen der bereits geschilderten Vorteile vorgeschlagen. Alternativ könnten aber auch andere Modellierungssprachen, Programmcode verschiedenster Programmiersprachen oder normaler Text eingesetzt werden. Auf die konkreten Modellausprägungen wie PIM, PSM, CIM wird

im Abschnitt 5.4 genauer eingegangen.

`Plattformen` sind in der MDA nicht gleichzusetzen mit der Gesamtarchitektur eines Systems. Sie stellen vielmehr einen Teil der Architektur dar. Der Plattformbegriff ist nicht absolut sondern relativ [PEME06, Seite 97]. Dabei können Plattformen verschiedene Abstraktionsgrade besitzen und aufeinander aufbauen. So können regelrechte Plattformstapel (Platform Stack) entstehen. Im folgenden Abschnitt 5.4 wird noch einmal etwas konkreter auf die Plattform eingegangen.

`UML-Profile` sind eine Möglichkeit der leichtgewichtigen Erweiterung des UML-Metamodells. Leichtgewichtig bedeutet hier, dass es mit UML-Profilen möglich ist, spezifische Sprachkonstrukte wie z.B. J2EE/EJB in UML einzubinden und dabei die Standardisierung und Interoperabilität von UML sicherzustellen [OMGp07]. Mit den UML-Sprachprofilen können zweck-, projekt-, unternehmens-, architektur-, domänen- und vorgehensspezifische Spezialisierungen bzw. Erweiterungen geschaffen werden. Diese Erweiterungen erfolgen über so genannte Stereotypen. Weiterhin gibt es einige von der OMG in der UML 2.0 Superstructure Specification vordefinierte UML-Profile für z.B. Geschäftsprozessmodellierung, Echtzeitsysteme und Multimediaanwendungen. Profile sind aber nicht nur ein Mechanismus des UML-Metamodells. Der Erweiterungsansatz befindet sich bereits auf der Metametaebene, was zur Folge hat das alle MOF-Modelle (vgl. 5.2) ebenfalls via Profile erweiterbar sind [PEME06, Seite 74 f.]. Auf die Modellhierarchie mit Metamodellen und MOF wurde im vorherigen Abschnitt bereits eingegangen

Die Modellierung auf Basis von klar definierten Metamodellen bringt den Vorteil mit sich, dass die so entstandenen Modelle grundsätzlich transformierbar sind. Dabei kann die `Transformation` von einem oder auch von mehreren in ein anderes bzw. mehrere andere Modelle auch automatisiert erfolgen. „Bei der Transformation werden Aspekte der Zieldomäne, also des Fachgebietes des Zielmodells, dem ursprünglichen Modell hinzugefügt und für die Zieldomäne irrelevante Aspekte herausgefiltert“ [PEME06, Seite 95 f.]. Die Transformation von Modellen erfolgt also grundsätzlich auf der Basis von Metamodellen, damit so der Transformierungsprozess (teil-)automatisiert werden kann. Meist erfolgt die Transformation dabei in zwei Schritten. Zuerst werden alle aus dem Quellmodell automatisch transformierbaren Informationen ins Zielmodell überführt. Anschließend erfolgt eine manuelle Ergänzung bzw. Erweiterung des Zielmodells um weitere Informationen, die nicht im Quellmodell vorhanden sind. Abbildung 5.3 zeigt das prinzipielle Transformationsschema der MDA. In der MDA wird zwischen verschiedenen Arten von Trans-

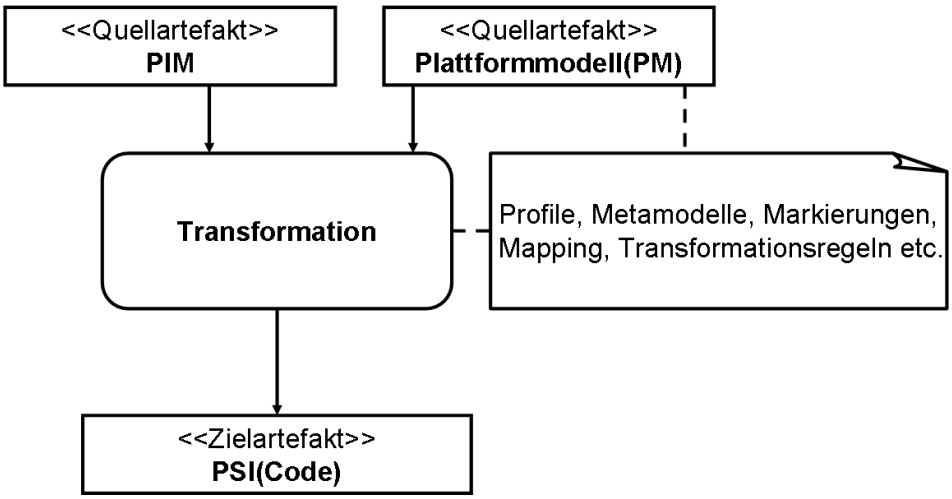


Abbildung 5.3: Prinzipielles MDA-Transformationsschema
[PEME06]

formationen unterscheiden. Zum einen gibt es die `Model-to-Model-Transformationen` bzw. Modell-zu-Modell-Transformationen. Da es verschiedene Modellarten (PIM, PSM) gibt, unterteilt man bei Modell-zu-Modell-Transformationen weiter in PIM-PIM-Transformationen, PIM-PSM-Transformationen, PSM-PSM-Transformationen und PSM-PIM-Transformationen. Während PIM-zu-PIM und PSM-zu-PSM dazu genutzt wird Modelle auf ihrer jeweiligen Ebene zu verfeinern bzw. zu ergänzen, nutzt man PIM-zu-PSM-Transformationen um Modelle auf bestimmte Plattformen zu transferieren. Natürlich kann auch in dieser Transformation eine Verfeinerung des Modells erfolgen. Einen Sonderfall stellen die PSM-PIM-Transformationen dar. Sie werden meist dazu eingesetzt um aus bestehenden Systemen ein Modell abzuleiten. Hier wird also eine Art „Reverse Engineering“ betrieben. Dieser Transformationsprozess ist allerdings

relativ schwierig, da natürlich keine Informationen über Metamodelle für die Transformationen vorliegen. Zum anderen gibt es noch die `Model-to-Code-Transformationen` bzw. Modell-zu-Code-Transformationen, bei denen aus PSMs Code erzeugt wird.

Außerdem unterscheidet man in der MDA verschiedene `Transformationsansätze`. Diese sind:

- Template-orientierter Ansatz
- Pattern-orientierter Ansatz
- Metamodellorientierter Ansatz

Im `Template-Ansatz` gibt es eine Vorlage, bei der bestimmte Bereiche als Platzhalter ausgezeichnet sind. Diese Bereiche werden bei der Transformation durch entsprechende Werte ersetzt. Dieser Ansatz eignet sich aufgrund der Code-Orientierung eher für Model-to-Code Transformationen und weniger für mehrstufige Model-to-Model-Transformationen.

Beim `Patternansatz` werden Pattern (Entwurfsmuster), wie sie in der Softwaretechnik häufig genutzt werden, für den Gebrauch in der MDA verwendet. Dazu muss das entsprechende Entwurfs- bzw. Analysemuster mit einem formalen Transformationsmechanismus verbunden werden. Dazu eignen sich vor allem Entwurfsmuster wie „abstrakte Fabrik“ oder „Singelton“ aber auch das „Visitor“-Pattern. Beim „Visitor“-Pattern z.B. wird die interne Struktur des Modells ausgenutzt um die einzelnen Elemente zu „besuchen“ und dabei den Code zusammenzusetzen. Im Patternansatz verläuft der Transformationsprozess zumeist CASE-werkzeuggestüzt und ebenfalls eher code-orientiert, was zu Folge hat, dass auch hier der Einsatz eher bei Model-to-Code Transformationen als bei Model-to-Model-Transformationen liegt.

Der `Metamodellansatz` basiert, wie der Name schon vermuten lässt, auf Metamodellen. Diese Metamodelle bilden die Basis für die Transformationen. Dazu werden die Modellelemente als Instanzen ihrer Metamodellklassen „angesprochen“. So ist es möglich, per Skriptsprachen auf die Ausgangsobjekte zuzugreifen und diese in die Zielobjekte zu überführen [PEME06, Seite 133]. Abbildung 5.4 zeigt den Mechanismus des Metamodellansatzes mit Ausgangs-, Zielmodell, den entsprechenden Metamodellen und dem Mapping der Metamodellelemente.

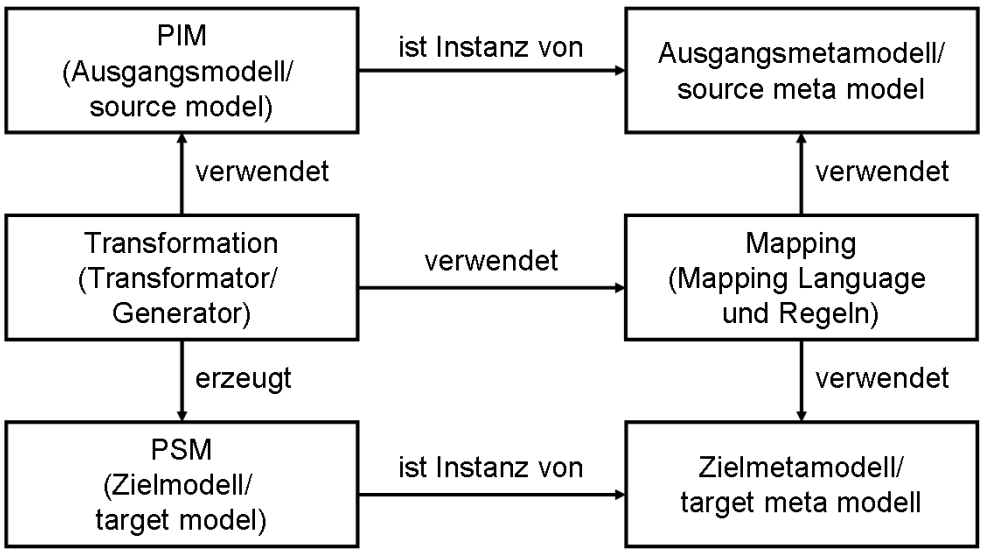


Abbildung 5.4: Mapping und Metamodelle für das Ausgangs- und das Zielmodell [PEME06]

Eine Kombination der gezeigten Ansätze und auch weiterer Ansätze wie z.B. Graphentransformation ist durchaus denkbar und oftmals auch Realität. So kann beispielsweise eine Model-to-Model Transformation von einem plattformunabhängigen Fachklassenmodell zu einem plattformabhängigen Entwurfsmodell auf Grundlage zweier Metamodelle gemacht werden. Anschließend kann dann eine Model-to-Code Transformation mit Hilfe von Code-Templates erfolgen.

5.4 MDA-Entwicklungsprozess

Der `MDA Entwicklungsprozess` hat im Großen und Ganzen drei grobe Schritte (Abb. 5.5). Der erste Schritt ist die Modellierung auf höherem Abstraktionsniveau unabhängig von der Implementierungstechnologie. Dieses erste Modell bildet die fachliche Spezifikation und wird `Platform Independent Model` (PIM) genannt. Im zweiten Schritt erfolgt eine Modell-zu-Modell(en)-Transformation, des im ersten Schritt entstandenen Modells in ein bzw. mehrere neue Modelle. Diese sind nun auf bestimmte Plattformen ausgerichtet und werden deshalb `Platform Specific Models` (PSM) genannt. Im letzten Schritt wird aus diesen Modellen mittels einer Modell-zu-Code-Transformation Quelltext für die jeweilige Plattform generiert. Dabei kann es auch mehrere Zwischenschritte geben. Gerade bei sehr umfangreichen Projekten sind mehrere zum Teil aufeinander aufbauende Plattformen üblich. Es kann dabei auch zu Modell-zu-Modell-Transformation von PSMs in andere PSMs oder auch von PIMs in andere PIMs zu kommen. So kann es sein das ein Modell A abhängig bzgl. der Programmiersprache Java ist aber unabhängig von der darunterliegenden Plattform z.B. des Betriebssystems. Die Betriebssystemplattform könnte z.B. Linux oder Windows sein. Nun kann das erste PSM 'Modell A' in ein weiteres PSM, nennen wir es 'Modell B', überführt werden. Dieses 'Modell B' kann nun auch abhängig bzgl. des Betriebssystems sein.

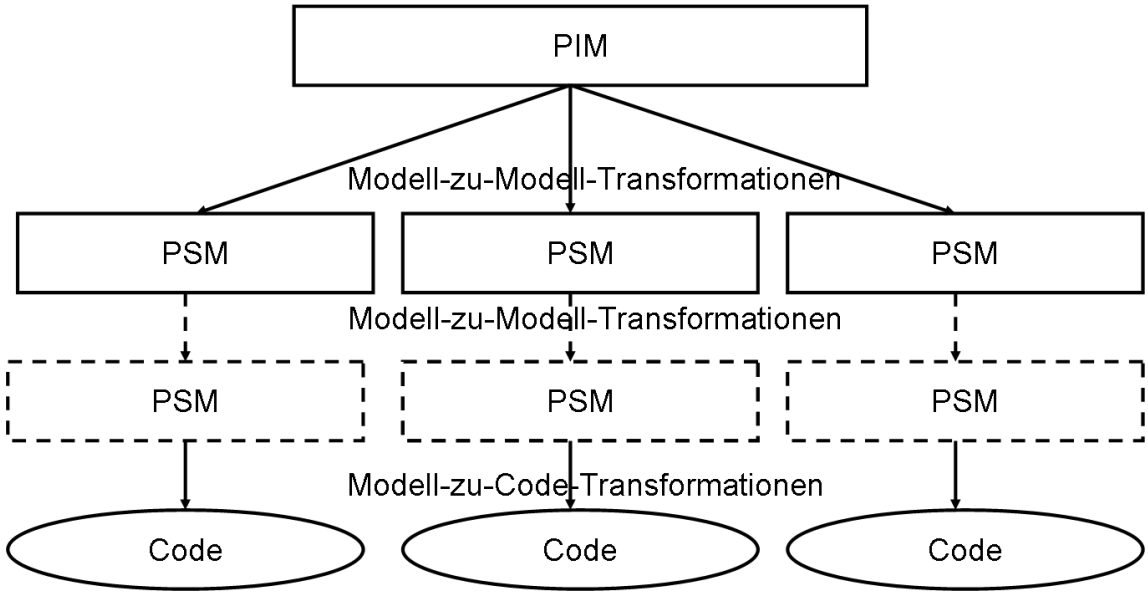


Abbildung 5.5: PIM-PSM-Code-Transformationen

Des Weiteren kann man auch ein hard- und softwareunabhängiges Modell dem Entwicklungsprozess voranstellen. Diese Modell wird auch `Computation Independent Model` CIM bzw. „Domain Model“ oder „Buisness Model“ genannt. Hier steht die Benutzung des Systems bzw. die Betriebsumgebung im Mittelpunkt. Das CIM soll eine Beschreibung der Anforderungen an das System auf einem sehr hohen Abstraktionsniveau sein. Ein kleines Bsp. zeigt Abbildung 5.6, in der ein 'Buch' mit 'Seiten' modelliert und entsprechend zweier verschiedener Plattformen verfeinert wird.

Die MDA schreibt keine genauen Vorgehensweisen vor, sondern akzeptiert verschiedene Methoden. Eine mögliche Vorgehensmethode sei hier kurz vorgestellt. Diese Methode orientiert sich an MDA-Pattern und dessen Konkretisierung nach dem Model-to-Code-Ansatz [PEME06] (Abb. 5.7).

Die Vorgehensweise unterteilt sich in mehrere Phasen. In der Phase der OOA (Objekt Orientierte Analyse) wird zunächst ein CIM erstellt. Es beinhaltet die Systemidee, Fachklassen, Anwendungsfälle und Geschäftsprozesse. Danach wird daraus ein PIM abgeleitet, in dem die Klassen, Anwendungsfälle und Geschäftsprozesse verfeinert bzw. ergänzt werden. Zuletzt folgt in dieser Phase noch einmal die Überprüfung des entwickelten PIMs. In der Plattformmodellierungsphase (PMP) werden die Zielarchitektur beschrieben, UML-Profile erstellt und entsprechende Metamodelle entwickelt. Es werden weiterhin in einer Folgephase der PMP der Generator, die Konfiguration und das Projektsetup erstellt. Aus dem Plattformmodell (PM) und dem PIM wird in der OO-Designphase (OOD) ein PSM erstellt bzw. Code generiert. Dazu werden die PIMs mit plattformspezifischen Elementen markiert und dann transformiert. In diesem Teil des Transformationsprozesses werden zwei MDA-Techniken eingesetzt, auf die noch mal

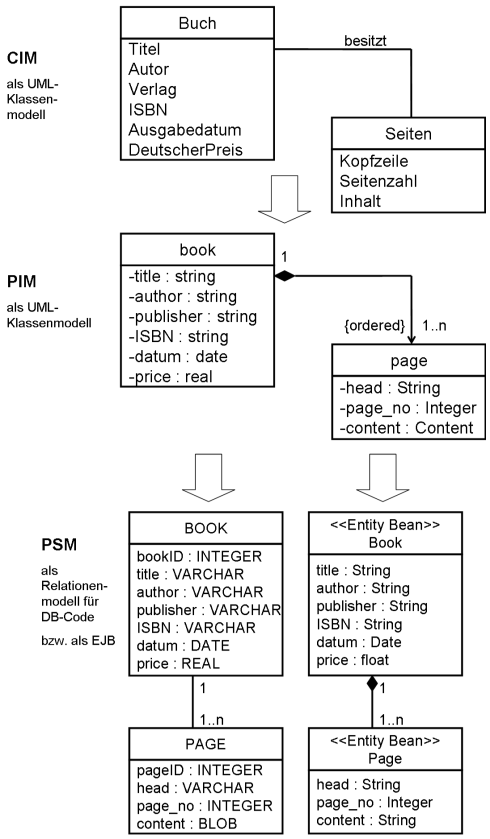


Abbildung 5.6: Beispiel CIM-PIM-PSM-Transformationsprozess

kurz eingegangen werden soll. Beim `Marking` wird das PIM mit „Marks“ (Markierungen) behaftet, die durch das Plattformmodell definiert wurden. Dabei gibt es zu jeder Markierung einen Satz von Regeln, der spezifiziert wie das markierte PIM-Element in das entsprechende PSM-Element abgebildet wird. Diese Elementabbildungen, auch `Mappings` genannt, sind also das Herzstück des Transformators (vgl. Abb. 5.4 und Abb. 5.8) und ermöglichen eine zielgerichtete Transformation der Modelle. Man unterscheidet dabei eine ganze Reihe von Mappingarten:

- refining - verfeinerndes Mapping
- abstracting - abstrahierendes Mapping
- representing - repräsentierendes Mapping
- migrating - abwandelndes Mapping
- merging - mischendes Mapping

So ist z.B. das XMI-Mapping der OMG zum Austausch von UML-Modellen ein repräsentierendes Mapping, weil es UML-Modelle nicht verändert, also weder abstrahiert, noch abwandelt oder verfeinert. Es wird nämlich nur die Notationsform zum besseren Datenaustausch informationsverlustfrei verändert. Auf die OOD-Phase folgt die OO-Programmierungs- bzw. Implementierungsphase (OOP). Hier können manuell bearbeitete Code-Bereiche ergänzt werden. Diese müssen mit den automatisch generierten Anteilen abgeglichen werden. In der letzen Phase der OOT wird dann die fertige Applikation erzeugt und auf dem Laufzeitsystem installiert („Built & Deploy“). Anschließend sollte die Applikation getestet werden. Insbesondere für kleinere Projekte empfiehlt sich eine iterative Vorgehensweise, indem die Phasen mehrmals durchlaufen werden (Abb. 5.9).

5.5 Vorteile und Probleme der MDA

Der folgende Abschnitt über die Vor- und Nachteile der MDA basiert weitestgehend auf [MIMU01] und [WAKE03].

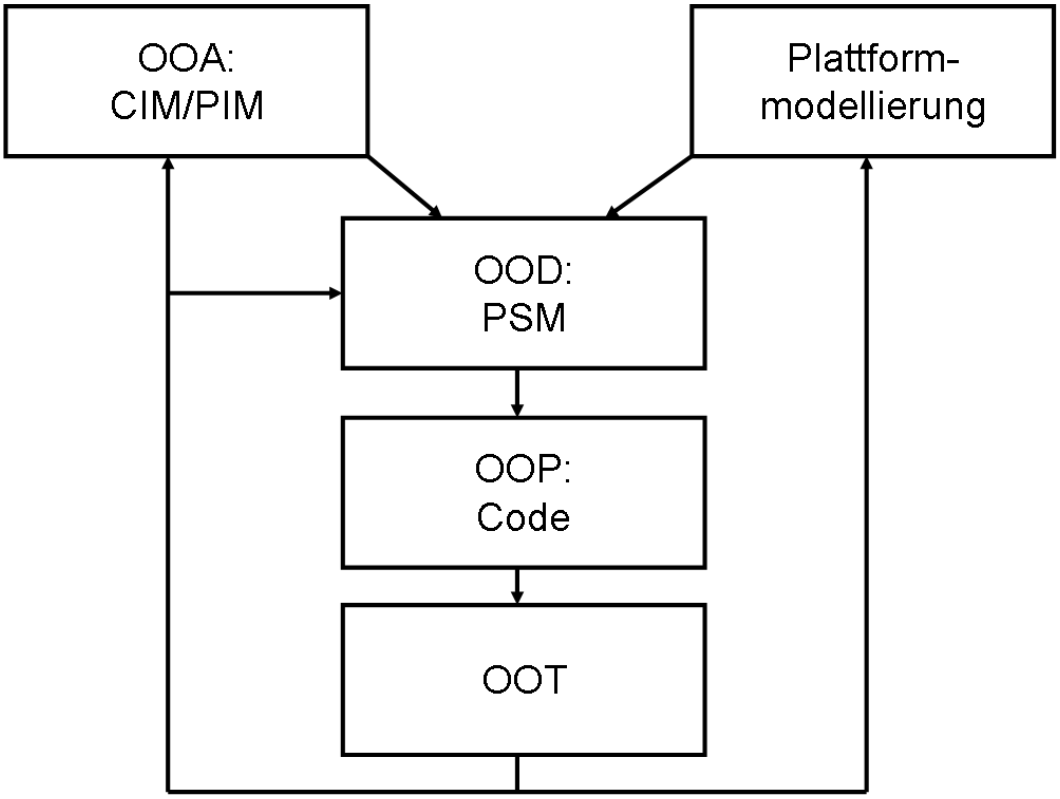


Abbildung 5.7: Beispiel Vorgehensweise MDA
[PEME06]

Einer der wichtigsten Vorteile der MDA liegt in der kompakten Darstellung komplexer Systeme auf mehreren Abstraktionsebenen. Ein PIM z.B. hilft dabei, die komplizierten fachlichen Abläufe eines unter Umständen sehr komplexen Softwaresystems zu verstehen und zu organisieren, ohne auf technische Details eingehen zu müssen. Positiv zu bewerten ist sicherlich auch die Flexibilität des Ansatzes, mit der die Möglichkeit eröffnet wird, sich neuen Entwicklungen in der (Software-) Technik anzupassen. So kann ein „veraltetes Softwaresystem“ auf eine neue, modernere Plattform angepasst werden, indem einfach ein neues Plattformmodell entwickelt wird. Mit Hilfe dieses Plattformmodells kann dann aus dem wieder verwendbaren „alten“ PIM ein entsprechend neues PSM abgeleitet werden. Außerdem lassen die Standardisierung der empfohlenen Modellierungs- und Metamodellierungssprachen eine Integration von Werkzeugen auch unterschiedlicher Hersteller zu. Die zumindest zum Teil automatische Transformation der Modelle unter einander und der Modelle zu Quellcode beschleunigen den Softwareentwicklungsprozess erheblich. Dabei werden gerade die oft „lästigen“ monotonen Programmieraufgaben wie z.B. Klassenerzeugung, Methodendefinitionen an den Generator abgegeben. Dies vermindert bei korrekter Implementierung der Generatoren natürlich auch die Wahrscheinlichkeit von Programmierfehlern in diesen Bereichen. Schließlich ist mit dem CIM und evtl. auch noch mit dem PIM eine effiziente Kommunikation mit dem Kunden über das zu entwickelnde Softwaresystem möglich.

Neben den bereits vielfältig aufgezeigten Vorteilen der MDA, gibt es aber auch einige kritische Schwierigkeiten. So ist es meist nicht möglich, ein Softwaresystem komplett aus den Modellen automatisch zu generieren, was einige Probleme mit sich bringt. Denn es müssen so Änderungen bzw. Ergänzungen per Hand gemacht werden. Erstens muss sich auf das nicht selbst geschriebene Programm eingestellt werden um es zu editieren. Das automatisch generierte Programm sollte also hinsichtlich der Qualität gewisse Standards wie (Menschen-) Lesbarkeit, Strukturierung und Änderbarkeit erfüllen. Außerdem ist es relativ schwierig manuelle Änderungen bzw. Erweiterungen z.B. auf Codeebene in den darüber liegenden Modellen konsistent zu halten. Auch umgekehrt besteht das Problem der wiederholten Codeerzeugung. Denn wenn Änderungen auf Codeebene stattgefunden haben und entsprechende Änderungen an oder in den Modellen nicht möglich sind, muss darauf geachtet werden, dass bei erneuter automatischer Codeerzeugung aus den Modellen diese Änderungen nicht überschrieben werden. Ein weiteres Problem stellt die eventuell schlechte Performanz des generierten Codes dar. Da manuelle Optimierungsmöglichkeiten am Code durch das Verfahren der (halb-)automatischen Generierung eingeschränkt sind, könnte eine schlechte Codeperformanz kaum oder schwierig zu beheben sein. Es gibt weiterhin Schwierigkeiten mit

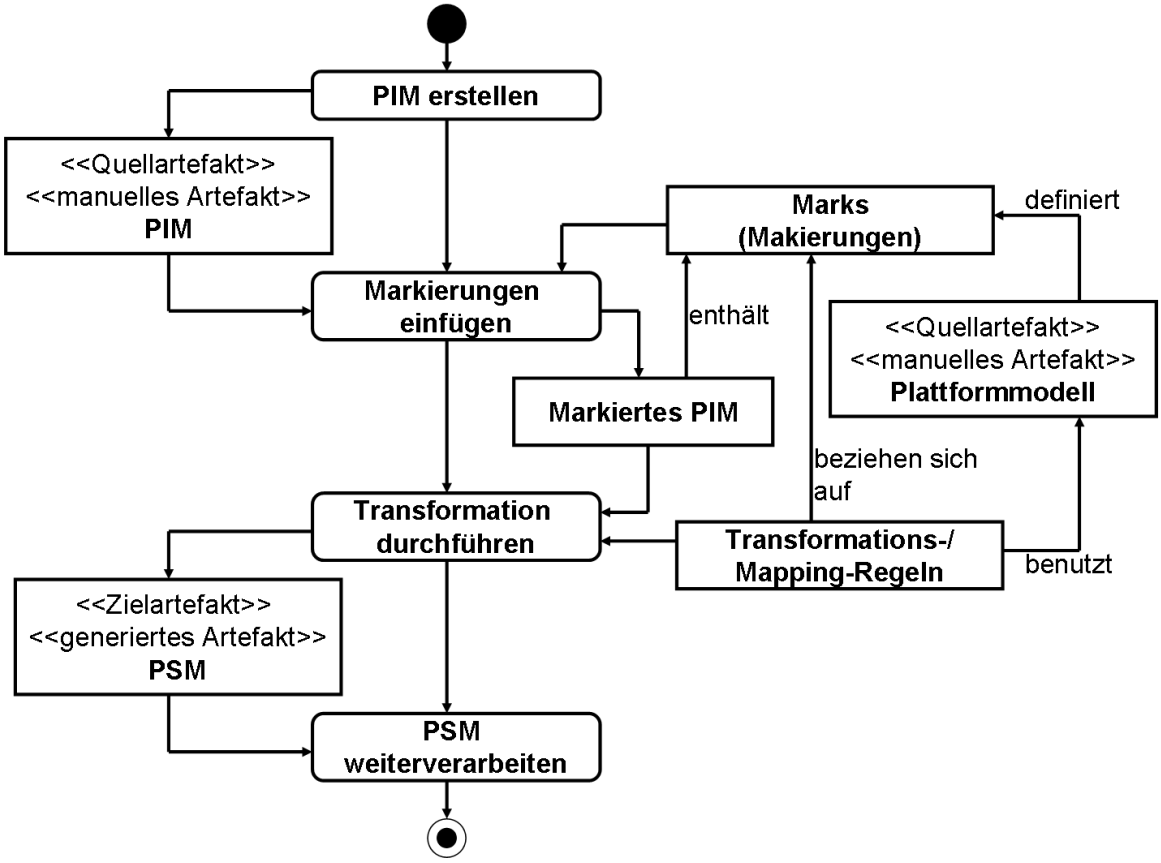


Abbildung 5.8: PIM-zu-PSM-Transformation mit Mapping und Marking [PEME06]

der Interoperabilität von verschiedenen MDA-Werkzeugen und -Tools, sowie der Modellierung des dynamischen Verhaltens für Softwaresysteme. Für eine Unterstützung der automatischen Erzeugung der Anwendungslogik der Software wäre dies aber notwendig. UML 2.0 hält dafür zwar einige Möglichkeiten bereit, diese sind aber allesamt von einem sehr geringen Abstraktionsniveau. Das heißt, sie entsprechen eher programmiersprachenartigen Konstrukten, so dass das Modellieren letztendlich zum Programmieren verkommt. Es artet in eine Art „graphischer Programmierung“ aus, die am Ende nicht einmal übersichtlicher als die herkömmliche, textuelle Programmierung ist.

Neben den angedeuteten Problemen stellt sich natürlich auch die Frage, ab wann oder ab welcher Größe bzw. Komplexität der Einsatz der MDA sinnvoll ist. Für kleinere Softwareprojekte ist eine Modellentwicklung sicherlich zu zeitaufwendig und damit wenig sinnvoll. Zusammenfassend lässt sich aber feststellen das die MDA gerade bei heutzutage immer komplexer werdenden Softwareprojekten eine sehr interessante Möglichkeit ist, die Softwareentwicklung zu formalisieren, und damit schneller und weniger fehleranfällig zu machen.

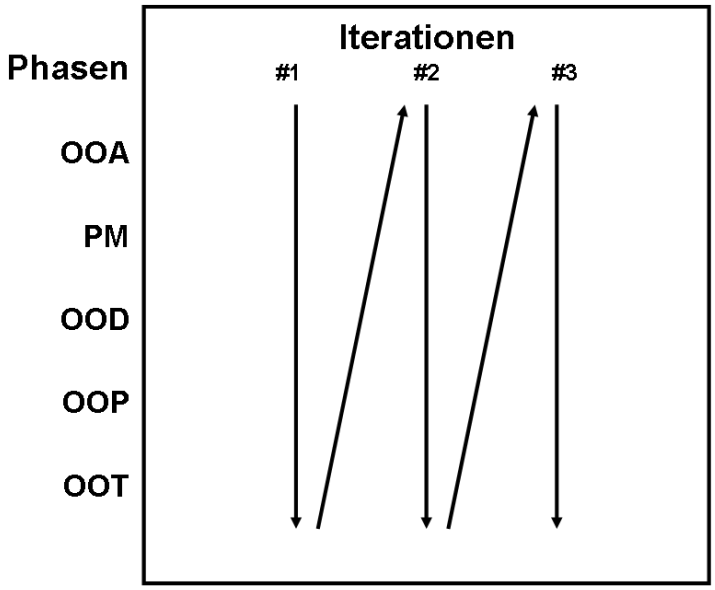


Abbildung 5.9: Phasen und Iterationen in der MDA [PEME06]

Kapitel 6

Adaption der MDA auf Entwurf für XML-Schemata

In diesem Kapitel soll darauf eingegangen werden, wie die zuvor vorgestellte Idee der MDA auf die Entwicklung von XML-Schemata übertragen werden kann. Die Kernidee dabei ist ein „Obermodell“ zu finden, das die Bestandteile und Konzepte enthält, die sich in allen drei Schemasprachen darstellen lassen. Aus diesem Modell sollen sich dann die Grundgerüste der verschiedenen schemasprachspezifische Modelle leicht erzeugen lassen können. An dieser Aussage lässt sich auch schon der Bezug zur MDA entdecken. Das „Obermodell“ entspricht also etwa einem PIM oder evtl. auch einem CIM, während die schemasprachspezifischen Modelle den PSMs entsprechen. Die in Kapitel 5 vorgestellten Verfahren und Vorgehensweisen sollen also mit ihren Vorteilen auf die XML-Schemaentwicklung adaptiert werden.

6.1 Vorgehensweise

Im ersten Schritt wurden die Schemasprachen detailliert untersucht, ihre Eigenschaften, Ausrichtung, Besonderheiten, Vor- und Nachteile heraus gearbeitet (vgl. Kapitel 3). Im zweiten Schritt wurden die Schemasprachen auf ihre Gemeinsamkeiten und Unterschiede hin untersucht und miteinander verglichen (vgl. Kapitel 4). Anhand dieser Ergebnisse sollen nun die folgenden Schritte eingeleitet werden:

- Entwicklung eines geeigneten plattformunabhängigen „Ober“-Modells aus den Gemeinsamkeiten der unterschiedlichen Schemasprachen
- aus dem plattformunabhängigem Modell entsprechende plattformabhängige Modelle für die XML-Schemasprachen entwickeln
- Mappingregeln für automatischen Transformationsprozess (PIM–>PSM–>Code) finden

6.2 Plattformunabhängiges Modell für XML-Schemata

Anhand der Gemeinsamkeiten der Schemasprachen, die in Kapitel 4 besprochen wurden, lassen sich die im plattformunabhängigen Modell zu unterstützenden Features ableiten. Für das plattformunabhängige Modell soll es also minimal die folgenden `Beschreibungsmöglichkeiten` geben:

- Elemente mit den Inhaltsmodellen:
  - leeres Element
  - reine Textelemente
  - Elemente, die nur Subelemente enthalten
  - Gemischter Inhalt (Mixed Content)
  - Alternativen
  - Sequenzen
  - 'Any'-Elemente
  - min und max Beschränkung der Auftretenshäufigkeit

- Attribute:
  - optional vs. required vs. fixed und Default- bzw. Fixwerte
  - Eindeutigkeitseigenschaft
  - Schlüsseleigenschaft (teilweise)
- Kommentare & Dokumentation (teilweise)

Als plattformunabhängiges Modell sollen hier 2 verschiedene Ansätze vorgestellt werden. Der erste beruht auf dem EMX-Modell, das in der Diplomarbeit von Robert Stephan [ST06] vorgestellt wurde und inzwischen unter dem Namen „Conceptual Design and Evolution of XML schemas“ kurz CoDEX genutzt wird. Der zweite Ansatz ist ein selbst entwickeltes Modell.

6.2.1 Vereinfachtes Entity Model for XML-Schema

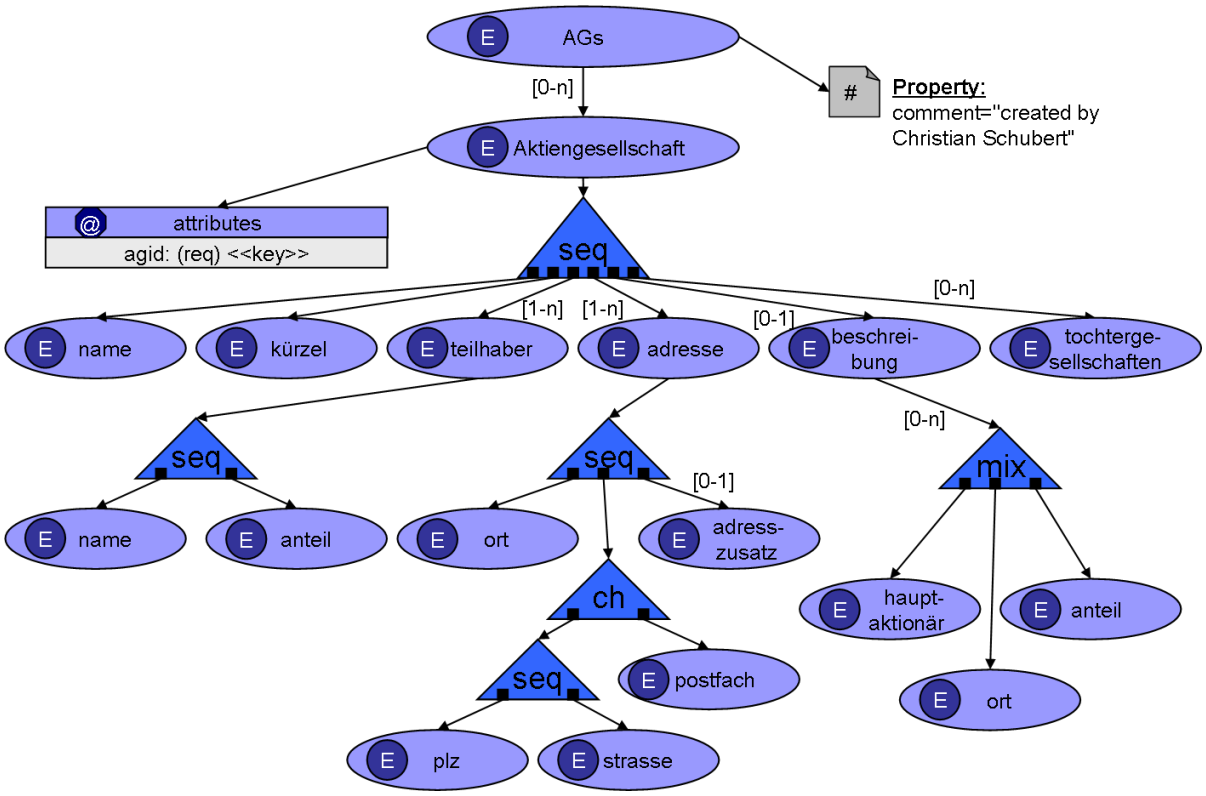


Abbildung 6.1: vereinfachtes CoDEX-Beispiel

Das hier vorgestellte Modell ist eine Vereinfachung des graphenbasierten `CoDEX`. Die Grundlagen des Modells werden in [ST06, Kapitel 5] beschrieben. Zum Zwecke des Einsatzes als PIM wird das dort vorgestellte Modell leicht abgewandelt und vereinfacht verwendet. Dazu werden alle Typbeschreibungselemente des CoDEX weggelassen, alle Verbindungen (Kanten) gerichtet verwendet und Mixed-Content-Elemente sowie Auftretenshäufigkeiten etwas anders dargestellt. So entsteht ein relativ einfaches und kompaktes Modell zur Beschreibung der der Grundelemente der XML-Schemasprachen. Das vereinfachte CoDEX besteht aus folgenden Komponenten:

- Entity: `Element`
- Entity: `Gruppenoperator`
- Entity: `Attributbox`
- Entity: `Anmerkung`
- `Verbindungen` (gerichtete Kanten) zwischen Entities

- `Eigenschaften` von Entities und Verbindungen:
  - Angabe ob Attribut optional, required, fixed beim jeweiligen Attribut der Attributboxentity
  - Fix- bzw. Defaultwerte angegeben hinter den Attributgebrauchswerten
  - Eindeutigkeitseigenschaft ('«unique»') angegeben in Element- bzw. Attributboxentity
  - Schlüsseleigenschaft ('«key»') angegeben in Element- bzw. Attributboxentity
  - Fremdschlüsseleigenschaft ('«keyref»') angegeben in Element- bzw. Attributboxentity
  - leere Elemente ('«empty»') angegeben in Elemententity
  - beliebige Subelemente ('«any»') angegeben in Elemententity
  - min und max Beschränkung der Auftretenshäufigkeit angegeben an den Kanten

Bei diesem Modell zu beachten ist, das bei nicht expliziter Angabe der Auftretenshäufigkeit an den Kanten, die Häufigkeit gleich eins angenommen wird. Weiterhin hingewiesen sei auf die sogenannten Andockpunkte für Kanten bei Gruppenoperatoren, die die Reihenfolge der Elemente in einem Gruppenoperator sicherstellen.

Abbildung 6.2 zeigt die fürs vereinfachte CoDEX benutzen Modellelemente mit ihren graphischen Repräsentation und Abbildung 6.1 präsentiert ein kleines Beispiel Schema in vereinfachter CoDEX-Notation.

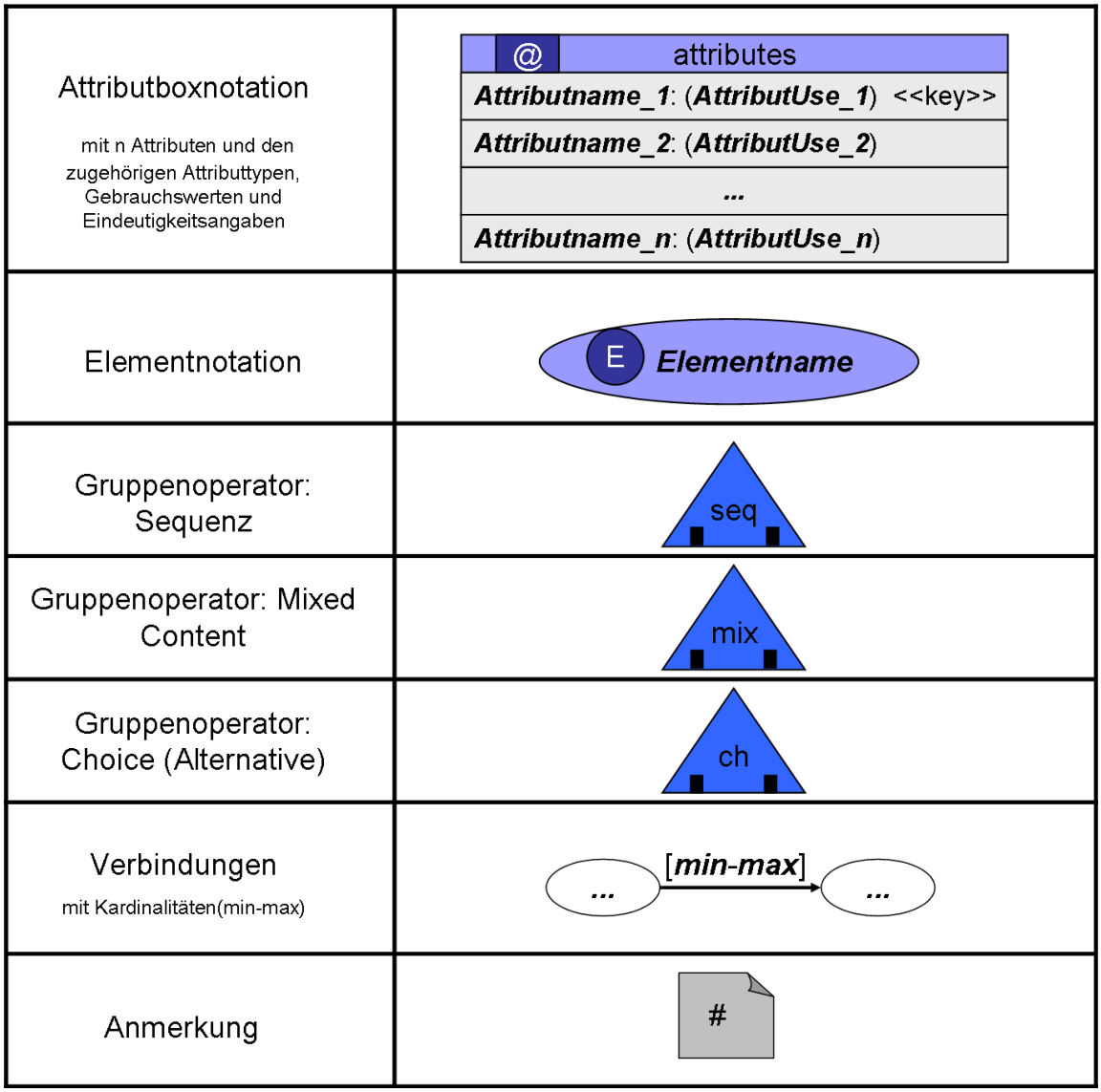


Abbildung 6.2: Notation im vereinfachten CoDEX-Modell (kursiv + fett dargestellt Platzhalter)

6.2.2 Eigenes Modell

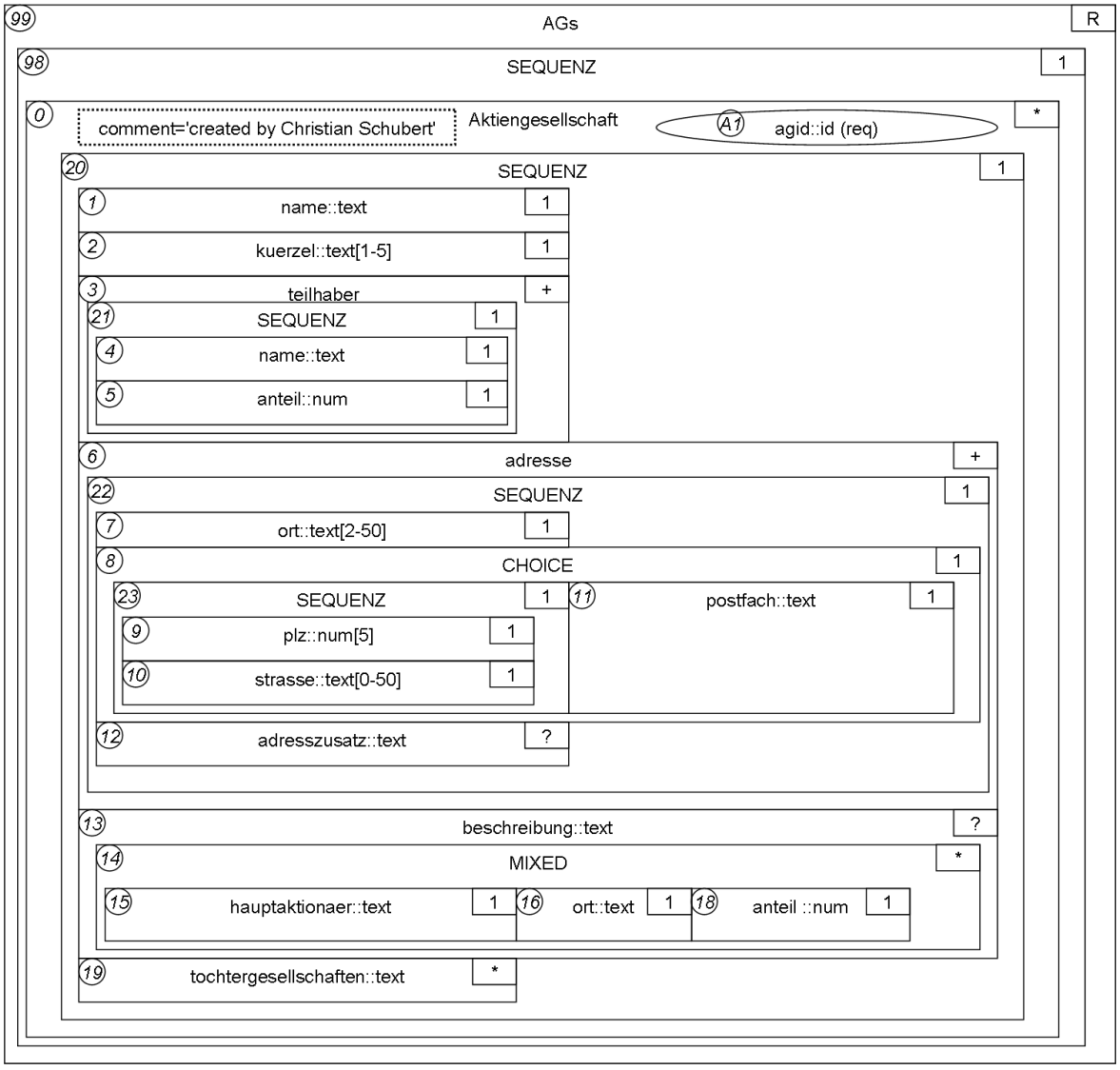


Abbildung 6.3: Eigenes Modell am Beispiel

Als eine weitere Möglichkeit für ein plattformunabhängiges Modell soll hier noch ein weiteres selbst konzipiertes Modell vorgestellt werden. Es ist hierarchisch und komponentenhaft aufgebaut, d.h., dass einem Modellelement untergeordnete Objekte in dem jeweiligen Modellelement dargestellt werden. Die dazu benötigten Grundelemente sind:

- `Elementbox`: In der Elementbox, dargestellt durch ein Rechteck, werden der Name des Elementes, ein einfacher Datentyp, die zugehörigen Attribute, untergeordnete Elemente und Anmerkungen notiert. Außerdem besitzt jede Elementbox links oben eine in einem Kreis angegebene ID und einen rechts oben „eingekasteltenˮ Quantifizierer. Die ID soll es in den Folgemodellen erleichtern sich auf bestimmte Elemente zu beziehen bzw. diese zu referenzieren. Der Quantifizierer gibt an in welcher Anzahl die Elemente minimal bzw. maximal auftreten. Hier wird auf die in DTDs übliche '?', '+', '*' Notierung zurückgegriffen und '1' für geforderte Elemente genutzt. Für das Wurzelelement wird statt des Quantors 'R' für „Rootelementˮ, also dem Wurzelelement, angegeben. Als einfache Datentypen werden zur groben Unterscheidung nur 'num' für rein numerische Werte und 'text' für Zeichenwerte vorgeschlagen. Hinter der Angabe des Datentypen kann in eckigen Klammern noch eine Längenbeschränkung folgen. Der Ausdruck 'zahl::num[5]' bedeutet also, dass das Element 'zahl' einen nummerischen Datentyp mit der Länge von genau 5 Zeichen besitzt, während 'ort::text[3-50]' einen String-Datentyp mit einer Länge von mindestens 3 bis maximal 50 Zeichen beschreibt. Des Weiteren ist es möglich, als Elementtypen die Schlüsselwörter 'EMPTY' für leere Elemente oder 'ANY' für ein Element mit beliebigen Unterelementen anzugeben. Ist kein Datentyp für ein Element

angegeben, wird davon ausgegangen, dass um ein Element handelt, was nur Subelemente aber keine eigenen Zeichendaten enthält.

- `Attributellipse`: Die zum Element gehörigen Attribute werden in einer Ellipse in der zum Element gehörigen Box angegeben und besitzen einen Attributnamen, einen Attributtypen, ein Attributgebrauchswert und eventuell noch ein Default- bzw. Fixwert. Der Attributgebrauchswert gibt an, ob das entsprechende Attribut einen fixen ('fixed') Wert hat, es optional ('optional') ist oder gefordert ('required') wird. Ist der Attributgebrauchswert gleich 'fixed', so ist in jedem Falle eine Fixwertangabe nötig, sonst ist statt dessen auch eine Defaultwertangabe möglich. Auch Attributellipsen bekommen IDs zugeordnet. Als in diesem plattformunabhängigen Modell erlaubte Attributtypen, werden hier als Grundtypen die Attributtypen von DTDs, mit der Ausnahme der Aufzählung erlaubter Werte, unterstützt.

- `Anmerkung`: Anmerkungen bzw. Kommentare werden in gepunkteten Boxen textuell notiert. Man bemerke den Unterschied von „gepunktet“ zu „gestrichelt“, weil letzteres zu Darstellung anderer Konzepte später noch verwendet wird.

- `Elementsequenzen`: Elementsequenzen werden in einer separaten Elementbox ohne Bezeichner (Elementnamen und Typ), dafür mit Schlüsselwort 'SEQUENZ', angegeben. Sequenzen werden durch die direkte, d.h. Unterkante der ersten an Oberkante der zweiten Elementbox, vertikale Aneinanderreihung von Elementboxen ausgedrückt, wobei die Reihenfolge der Sequenz von oben nach unten definiert ist.

- `Elementalternativen`: Elementalternativen werden analog aber mit Schlüsselwort 'CHOICE' und als horizontale Aneinanderreihung von Elementboxen dargestellt. Hier heißt das also rechte Kante von Box eins an linker Kante von Box zwei.

- `Mixed Content`: Mixed Content wird analog zu Elementalternativen mit dem Schlüsselwort 'MIXED' dargestellt. Zu beachten ist, dass als Quantor für Mixed Content '*' angegeben werden muss.

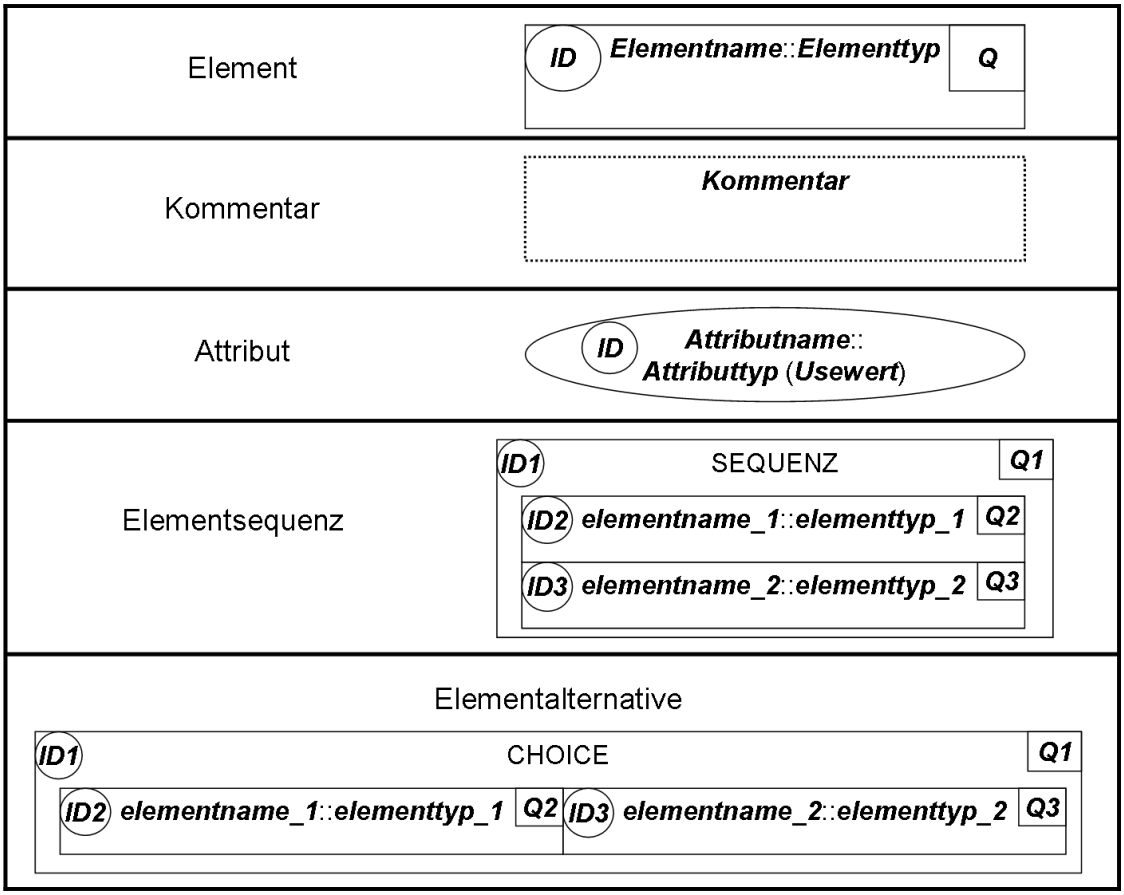


Abbildung 6.4: Notation im eigenen Modell (kursiv + fett dargestellt Platzhalter)

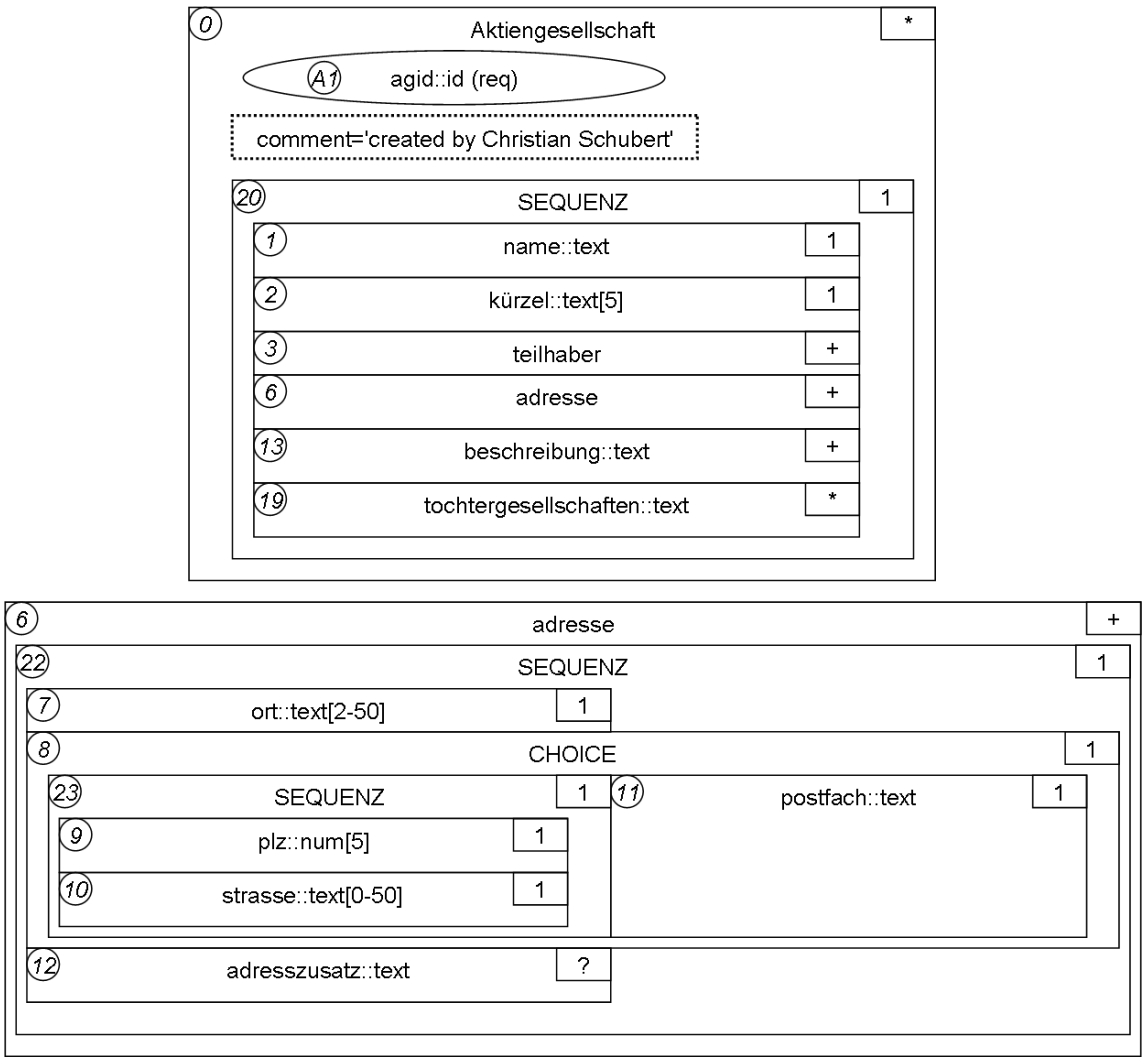


Abbildung 6.5: Komponentensicht für eigenes Modell (Beispiel nicht vollständig)

Abbildung 6.4 zeigt beispielhaft die graphische Notation der einzelnen Elemente, während Abbildung 6.3 das gleiche Beispiel, wie es beim vereinfachten CoDEX-Beispiel benutzt wurde, für die Notation im eigenen Modell zeigt.

Als Grundideen für dieses Modell dienten unter anderem Komponentendiagramme, wie sie z.B. in UML eingesetzt werden, oder auch Struktogramme, wie sie z.B. beim Testen für Anweisungsüberdeckung und Ähnliches benutzt werden. Wie schon bereits an dem kleinen Beispielmodell (siehe Abb. 6.3) zu erkennen, werden die Modelle relativ voluminös und unübersichtlich. Deshalb empfiehlt es sich, wie bei Komponentendiagrammen üblich, zu Übersichtszwecken Unterkomponenten oder in diesem Falle Unterelemente ein- („Whitebox“) bzw. ausblenden („Blackbox“) zu können. Abbildung 6.5 zeigt dazu ein kurzes Beispiel in `Komponentensicht`. Durch die Nutzung der ID für die Elemente wird auch sichergestellt, dass verschiedene Elemente mit gleichem Namen unterschieden werden können.

6.3 Plattformabhängige Modelle für XML-Schemata

Als plattformabhängige Modelle werden die in Abschnitt 6.2 vorgestellten Modelle als Grundlage genommen. Es werden plattformspezifische Modellelemente für die verschiedenen XML-Schemasprachen ergänzt. Dazu werden die zwei vorgestellten Modelle wieder getrennt betrachtet und jeweils für jede Schemasprache ein „eigenes“ Modell konzipiert.

6.3.1 Plattformabhängige Modelle für das vereinfachte CoDEX-Modell

Aus dem plattformunabhängigen vereinfachten CoDEX-Modell werden in diesem Abschnitt nacheinander die drei Konzepte für DTD, XML-Schema- und schematronspezifische Modelle abgeleitet.

Modell für DTD

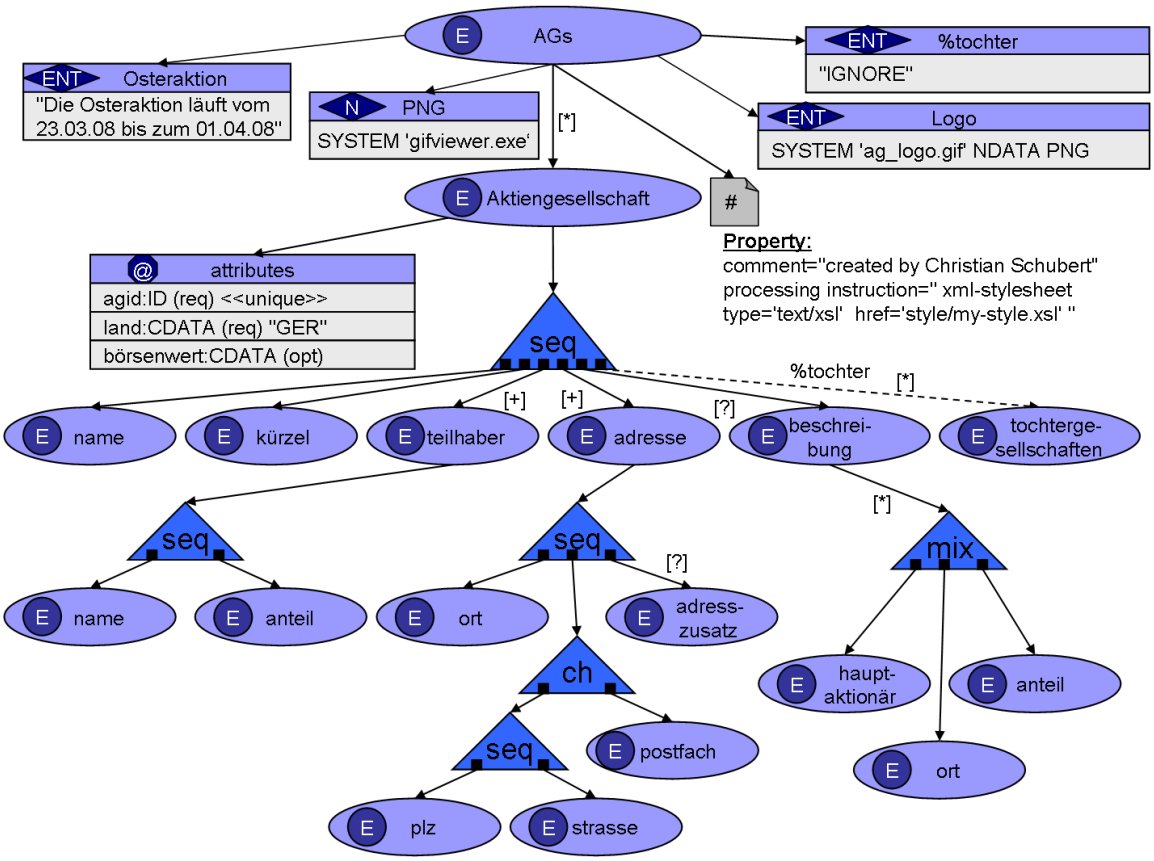


Abbildung 6.6: erweitertes Beispiel für DTDabhängiges Modell in CoDEX-Notation

Für das von der Plattform DTD abhängige Modell werden nur wenige Erweiterungen benötigt, da bereits das plattformunabhängige Modelle die meisten Informationen zur Erstellung einer DTD enthält. Das vereinfachte CoDEX-Modell muss also nur um folgende Konzepte erweitert werden:

- Attributtypen: Attributtypen werden einfach hinter dem Namen des jeweiligen Attributes mit Doppelpunkt abgetrennt angegeben.

- Entity `Entities`: Entities werden mit einem speziellen 'ENT'-Symbol (Raute) gekennzeichnet in einer Box mit Name und Wert notiert. Dabei wird der Name und das 'ENT'-Symbol in der Kopfzeile und der Wert in der Inhaltszeile der Box angegeben.

- Entity `Notations`: Notationen werden ebenfalls in einer Box angegeben, die durch eine Raute, die das Zeichen 'N' für Notation enthält, im Kopf der Box makiert ist. Im Kopf der Box wird zusätzlich noch der Name und im Inhaltsteil der Box der Wert der Notation notiert.

- `Processing Instructions`: Processing Instructions werden den schon im plattformunabhängigen Modell vorgestellten Anmerkungen untergeordnet. Statt nur der Eigenschaft (Property) 'comment' wird nun auch die Eigenschaft 'processing instruction' unterstützt.

- `Bedingte Abschnitte`: Bedingte Abschnitte werden über die Kanten angegeben. Bei einem bedingten Abschnitt wird die entsprechende Kante gestrichelt dargestellt und der entsprechende Wert ('INCLUDE', 'IGNORE' oder eine Parameterentity) an der Kante notiert.

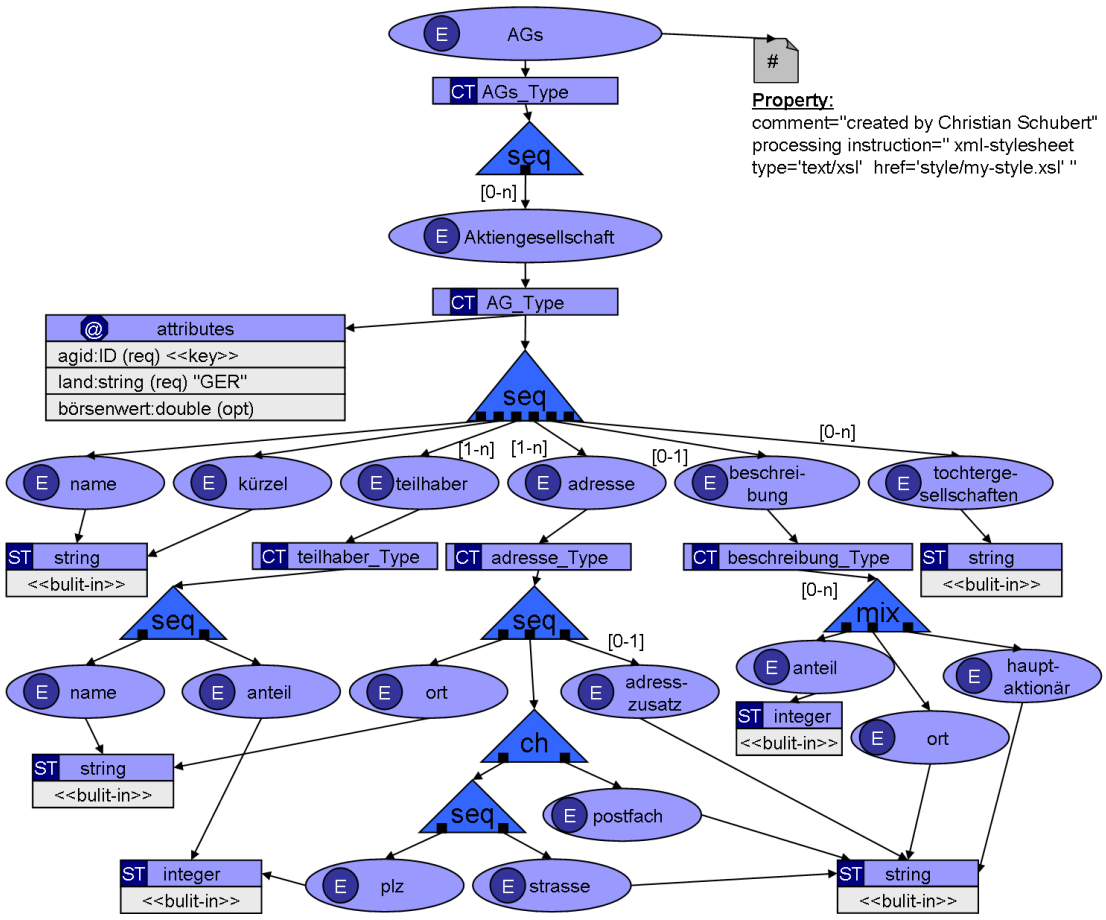


Abbildung 6.7: erweitertes Beispiel für das von XML-Schema abhängige Modell in CoDEX-Modell-Notation

Modell für XML-Schema

Für die „Plattform“ XML-Schema bietet sich hier natürlich das vollständige „Conceptual Design and Evolution of XML schemas“-Modell (CoDEX) an, da es ja speziell für diese Sprache konzipiert worden war. Es kommen also im wesentlichen Datentypendefinitions-Modellelemente hinzu. Die Beschreibung des Modells kann in [ST06] nachgelesen werden. Die im vereinfachten CoDEX veränderte 'Mixed Content'-Notation wird hier beibehalten. Es soll aber darüber hinaus auch das `Schlüssel-` und `Fremdschlüsselkonzept` unterstützt werden. Dazu werden Schlüsselelemente oder -attribute wie bereits im vereinfachten CoDEX mit dem Signalwort `<<key>>` in der entsprechenden Modellelementdefinition notiert. Der Schlüssel kann wie folgt um einen eindeutigen Schlüsselnamen ergänzt werden: `<<key:schlüsselname>>`. Das wird spätestens notwendig, wenn mehrere Modellelemente einen Schlüssel bilden. Ist kein Schlüsselname angegeben, wird der Name des Modellelementes mit der Endung '_key' als Schlüsselname verwendet. Fremdschlüssel werden durch das Signalwort `<<keyref>>` gekennzeichnet. Im `<<keyref>>` muss natürlich der zu referenzierende Schlüssel angegeben werden. Dies geschieht über die Angabe des Schlüsselnamens. `<<keyref:agid_key>>` referenziert also den 'agid'-Schlüssel (siehe Beispiel). Da in XML-Schema ein Name für einen Fremdschlüssel Bedingung ist, wird, wenn nicht explizit angegeben, automatisch ein Name nach dem Muster *Modellelementname* + '_keyref_' + *Schlüsselname* angelegt. Alternativ könnte man auch mit entsprechend beschrifteten Pfeilen zwischen den Modellelementen arbeiten. Dies hätte den Vorteil von weniger textueller Notation, aber auch den Nachteil von gesteigerter Unübersichtlichkeit wegen des zusätzlichen „Pfeilewirrwarrs“.

Des Weiteren soll das CoDEX-Modell auch um das `Attributgruppenkonzept` ergänzt werden. Dazu soll es nun auch möglich sein, Attributboxen statt mit `attributes` auch `attgroup` zu bezeichnen und dahinter in Anführungszeichen den Namen der Attributgruppe anzugeben.

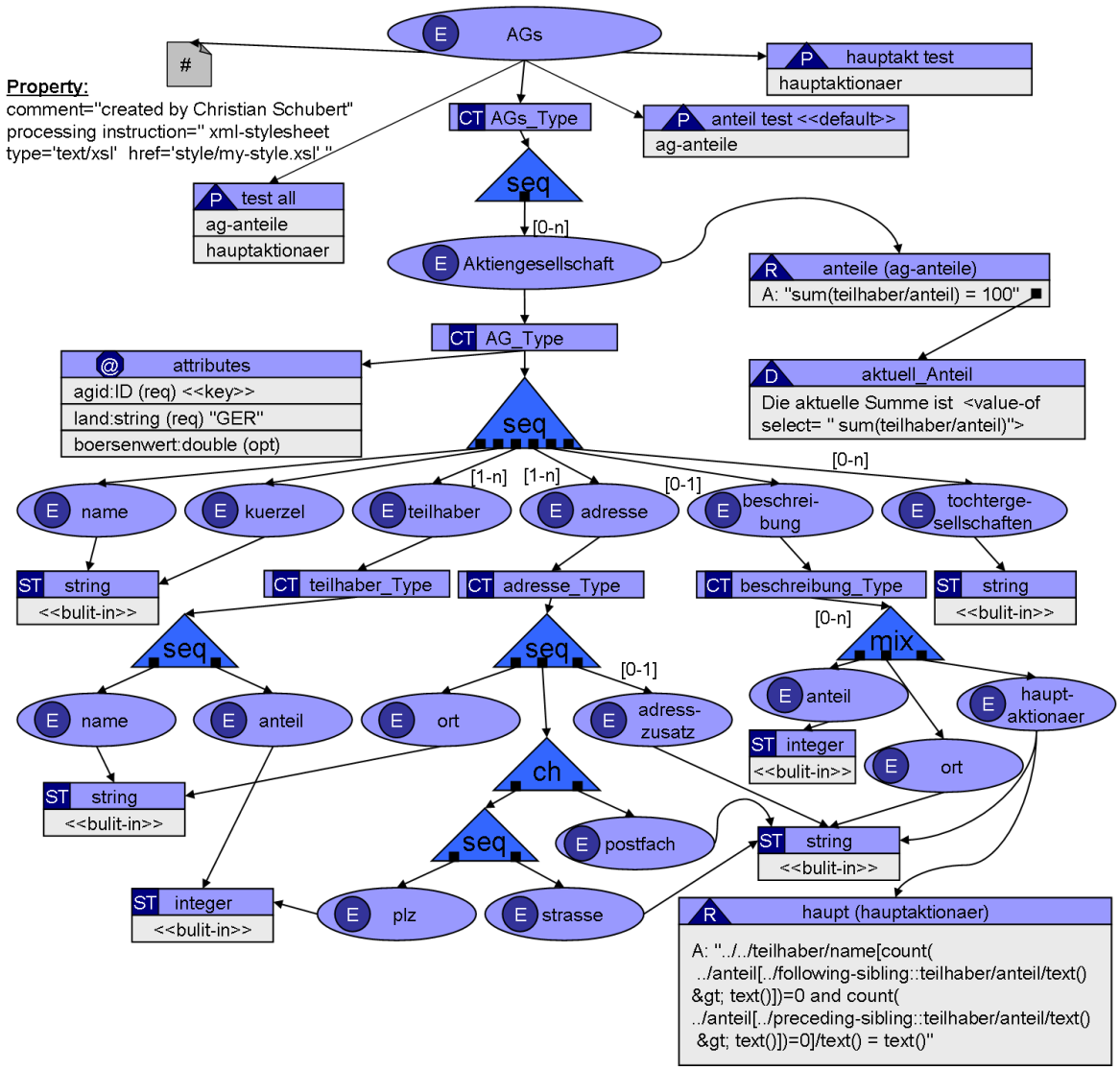


Abbildung 6.8: erweitertes Beispiel für das von Schematron abhängige Modell in CoDEX-Notation

Modell für Schematron

Schematron ist nicht auf die strukturelle Beschreibung von XML-Dokumenten ausgerichtet, und wird zumeist im Zusammenspiel mit anderen auf XML-Syntaxbeschreibung ausgerichteten Schemasprachen eingesetzt. Es ist zwar möglich, mit Schematronregeln fast alle Strukturbeschreibungen zu simulieren, doch ist dieses nicht besonders sinnvoll und relativ umständlich, weil selbst für kleine Schemata schnell ein nicht unerheblicher Satz an Regeln nötig wäre. Deshalb soll auch hier als Grundgerüst für die Strukturbeschreibung in „Schematron“ eine andere Schemasprache und zwar XML-Schema verwendet werden. Zusätzlich sollen aber weitere Modellelemente benutzt werden, um mehr semantische Regeln („Constraints“) zu beschreiben. Dazu wird erst einmal das komplette im vorigen Abschnitt beschriebene plattformabhängige Modell für XML-Schema übernommen und um folgende Konzepte erweitert:

- Entity `Rule`: Ein neues „Regel“- oder „Rule“-Element wird eingeführt. Es wird ähnlich wie die Attribute in einer Box notiert, die jetzt aber mit einem mit „R“ für Rule gekennzeichnetem Dreieck beschrieben wird. Im Kopf der „Rule-Box“ muss ein Name angegeben werden, der im Schemaquelltext dann dem Wert des 'id'-Attribut des 'rule'-Elementes entspricht. Des Weiteren wird direkt dahinter in Klammern die ID des Pattern angegeben, dem die Regel zugeordnet werden soll. Das heißt also auch, dass mit dieser Angabe indirekt auch die entsprechenden Pattern erzeugt werden, wenn dies nicht schon vorher durch ein andere „Rule-Boxmit gleicher ID-Pattern-Notation geschehen ist. Anzumerken ist auch noch, dass man so mehrere Regeln (rules) zu einem Pattern zuordnen kann. Die Regel wird genauso wie andere Modellelemente per gerichteter Kante einem bestimmten Element oder Attribut zugeordnet. Dies entspricht in Schematron der Auswahl eines Schemaelementes mittels dem '`context`'-Attribut des '`rule`'-Elementes. In den Inhaltzeilen der „Rule-Box“

können dann textuell die XPath-Test-Ausdrücke von Asserts und Reports notiert werden. Dabei werden Assertions mit vorangestelltem 'A:' und Reports mit 'R:' gekennzeichnet. Soll das entsprechende '`assert`'- oder '`report`'-Element einem oder mehreren '`diagnostics`'-Elementen zugeordnet werden, so muss eine gerichtete Kante von einem „Andockpunkt" zum jeweiligen Diagnoseelement angelegt werden. Die jeweiligen Fehlermeldungen der Assertions und Reports sollten meinem Erachten nach erst im Quelltext eingefügt werden, weil diese nur textuell notiert werden könnten. Das Modell würde somit nur voluminöser und damit unübersichtlicher werden. Andere in Schematron erlaubte „Rule"-Attribute wie z.B. 'abstract' oder 'role' werden im „Properties View" (vgl. [ST06, Kapitel 5.2, Seite 76]), also nicht direkt in der graphischen Notation des Modellelementes angegeben.

- Entity `Phase`: In einem neuen „Phasen"-Element können alle für diese Phase aktiven „Pattern" definiert werden. Die Notation erfolgt ähnlich zum „Rule"-Element nur mit 'P' als Kennzeichen im Dreieck und den zu aktivierenden „Pattern" in den Inhaltszeilen. Die „Phasen"-Elemente werden dem Modellwurzelelement untergeordnet. Die in Schematron spezifizierbare Default-Phase kann durch Angabe von `<<default>>` hinter dem Phasennamen angegeben werden.

- Entity `Diagnostic`: Auch hier wird ein neues „Diagnose" oder „Diagnostic"-Modellelement eingeführt, das jetzt mit 'D' im Dreieck gekennzeichnet wird. Im Kopfteil des „Diagnostic"-Modellelementes steht die ID des '`diagnostics`'-Elementes. Im Inhaltsteil wird textuell die Diagnosemeldung notiert. Das „Diagnostic"-Modellelement ist über eine gerichtetete Kante einem '`assert`' oder '`report`' zugeordnet.

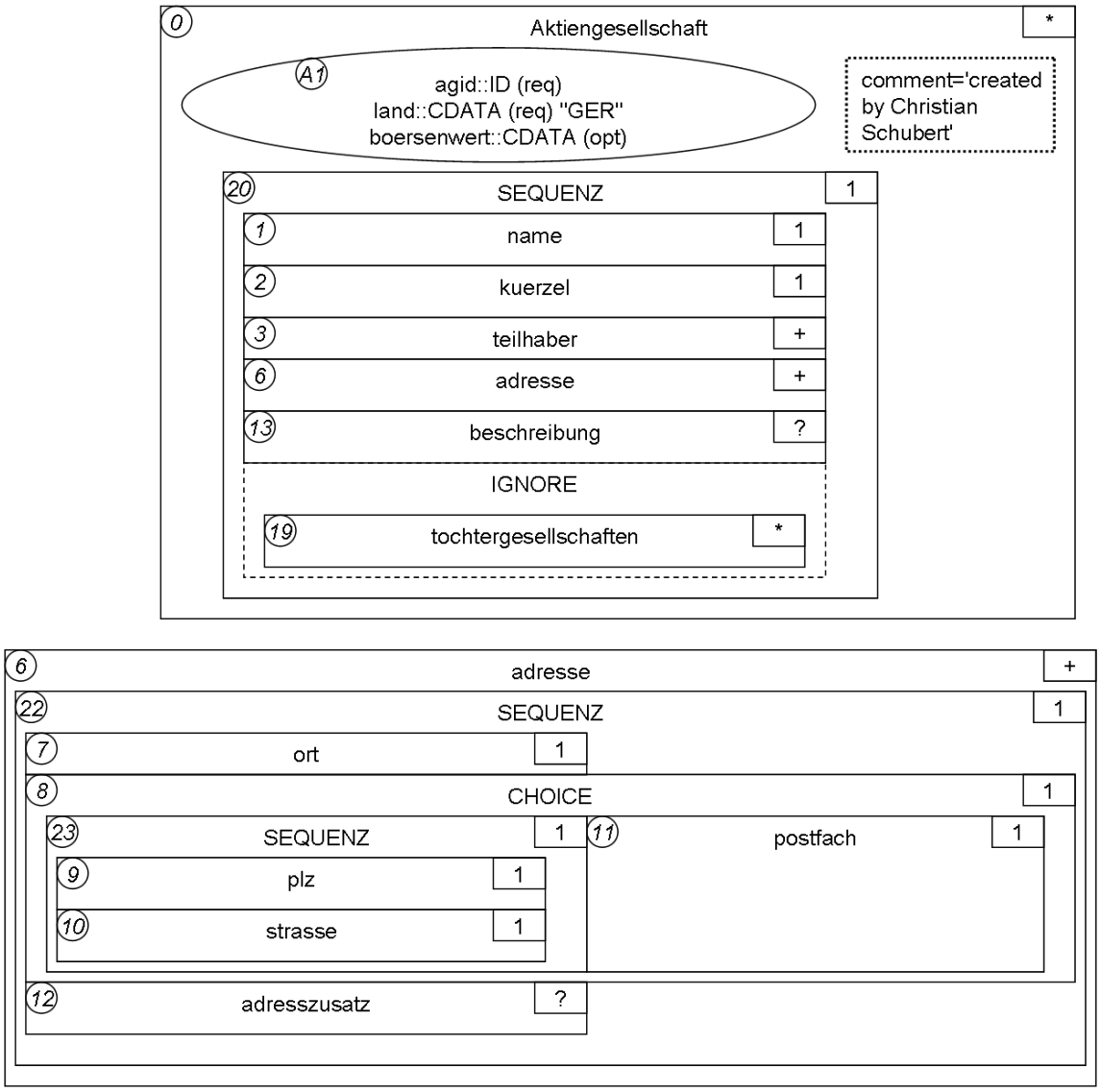


Abbildung 6.9: Beispiel für eigenes DTD-Modell (wird fortgesetzt...)

6.3.2 Plattformabhängige Modelle für das eigene Modell

In den folgenden Abschnitten sollen nun, alternativ zu den CoDEX-Modellen, für das in Kapitel 6.2.2 vorgestellte eigene Modell, plattformabhängige Modelle bzw. Modellmodifikationen für die verschiedenen Schemasprachen konzipiert werden.

Modell für DTD

Für ein bezüglich der Schemasprache DTD abhängiges Modell kann das eigene Modell wie folgt modifiziert werden:

- Da es in DTD kaum eine Unterscheidung der Datentypen für Elemente gibt, werden alle Typangaben für Elemente weggelassen bzw. entfernt. Ausgenommen hiervon sind die Typen 'ANY' und 'EMPTY'.
- Die Längenbeschränkungen der Datentypen können ebenfalls vernachlässigt werden.
- Im Übrigen kann das eigene Modell komplett übernommen werden.
- Zusätzlich kommt nur das Konzept der „Bedingten Abschnitte“ hinzu, die durch Angabe einer gestrichelten Box darstellt werden.

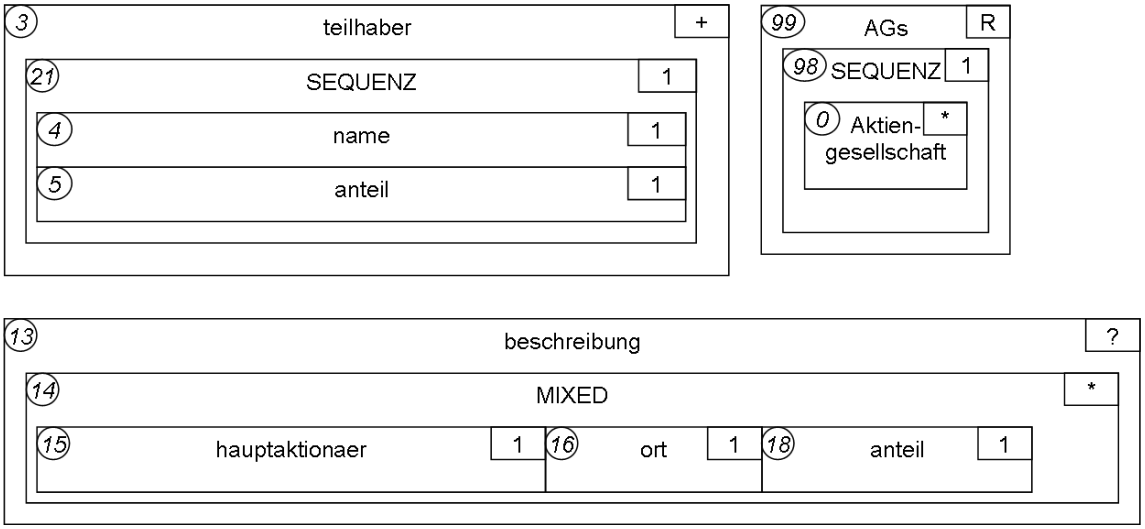


Abbildung 6.10: Beispiel für eigenes DTD-Modell (Fortsetzung)

Zusammenfassend lässt sich also sagen, dass fast keine Modellelemente hinzugekommen sind, sondern im Gegenteil sogar einige entfernt werden, weil sich entsprechende Informationen nicht mit DTD darstellen lassen. Auf die Unterstützung anderer DTD-Konzepte wie „Notations“ oder „Entities“ wurde hier verzichtet, weil sich diese graphisch nicht gut repräsentieren lassen und größten Teils nur textuell notiert werden könnten. Abbildungen 6.9 und 6.10 zeigen ein von DTD abhängiges Modell in Komponentennotation, das sich auf das Beispiel des plattformunabhängigen Modells aus Abbildung 6.3 bezieht. Es wurde nur zusätzlich um einen „Bedingten Abschnitt“ und zwei Attribute erweitert.

Modell für XML-Schema

Das von Schematron abhängige eigene Modell, wird in sofern erweitert, dass alle Elemente einen Typ erhalten. Das heißt auch Elemente, die nur Subelemente und keine Zeichendaten enthalten, müssen getypt werden. Erlaubt sind nun alle in XML-Schema erlaubten Datentypen, also vordefinierte und selbst definierte Typen. Komplexe Typen werden durch die Notierung ihres Inhaltmodells in ihrer Elementbox als komplex gekennzeichnet. Einfache Typen werden dadurch ausgewiesen, dass sie keine Submodellelemente besitzen. Auch die Attributtypen dürfen in diesem Modell alle in XML-Schema und selbst definierten Typen annehmen. Außerdem ist beim XML-Schema-Modell auch die Angabe von konkreten Werten für die Auftretenshäufigkeit in den Quantorenkästchen erlaubt. Für minimal 1 und maximal 3 Auftreten wird '1-3' im entsprechenden Kästchen notiert. Hinzu kommt des Weiteren die Angabe von Schlüsseln und Fremdschlüsseln. Diese Schlüsseldefinitionen können über die Markierungen '«key»' bzw. '«keyref»' im jeweiligen Modellelement vorgenommen werden. Hinter dem Signalwörtern 'key' bzw. 'keyref' wird, mit

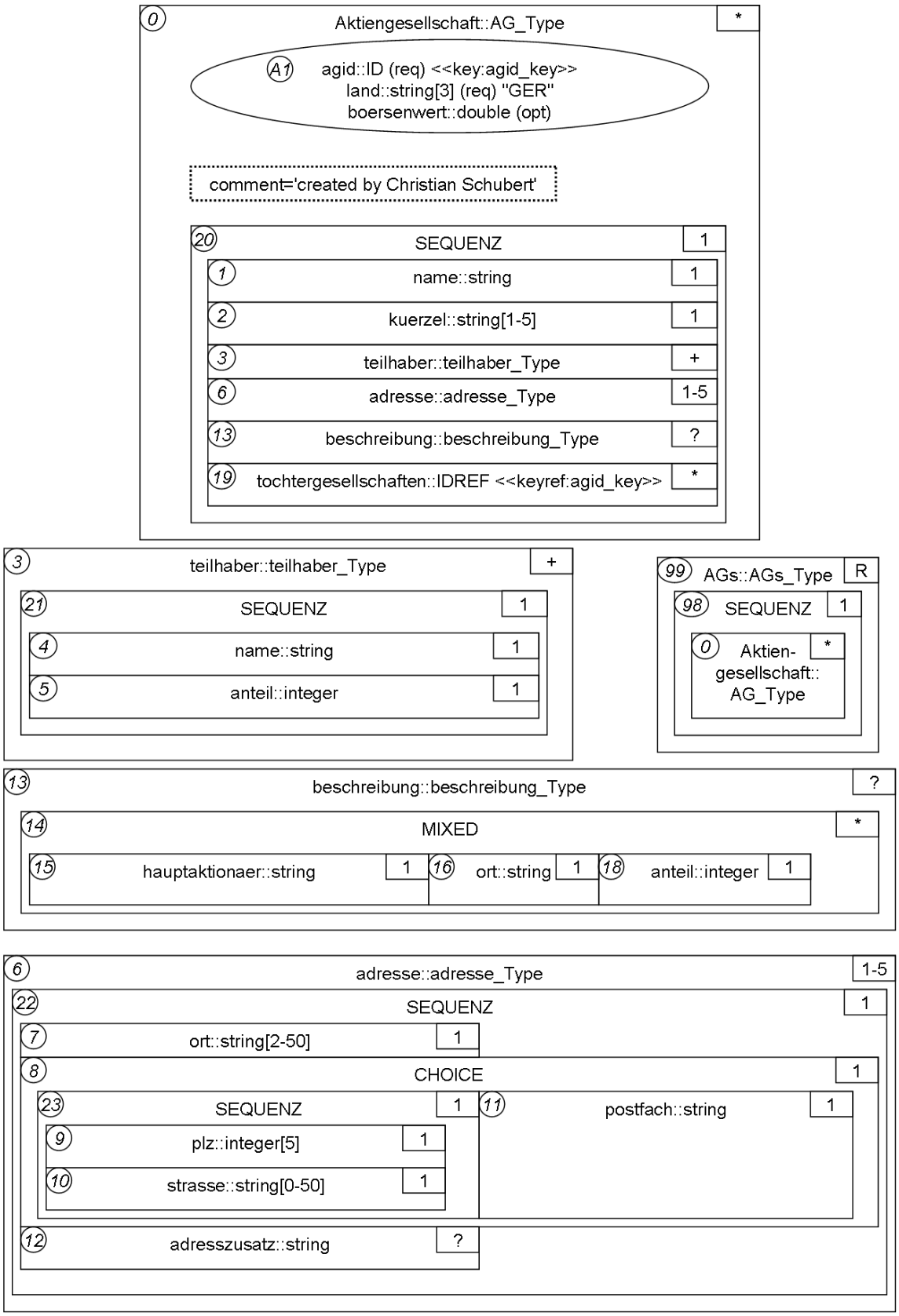


Abbildung 6.11: Beispiel für eigenes XML-Schema-Modell

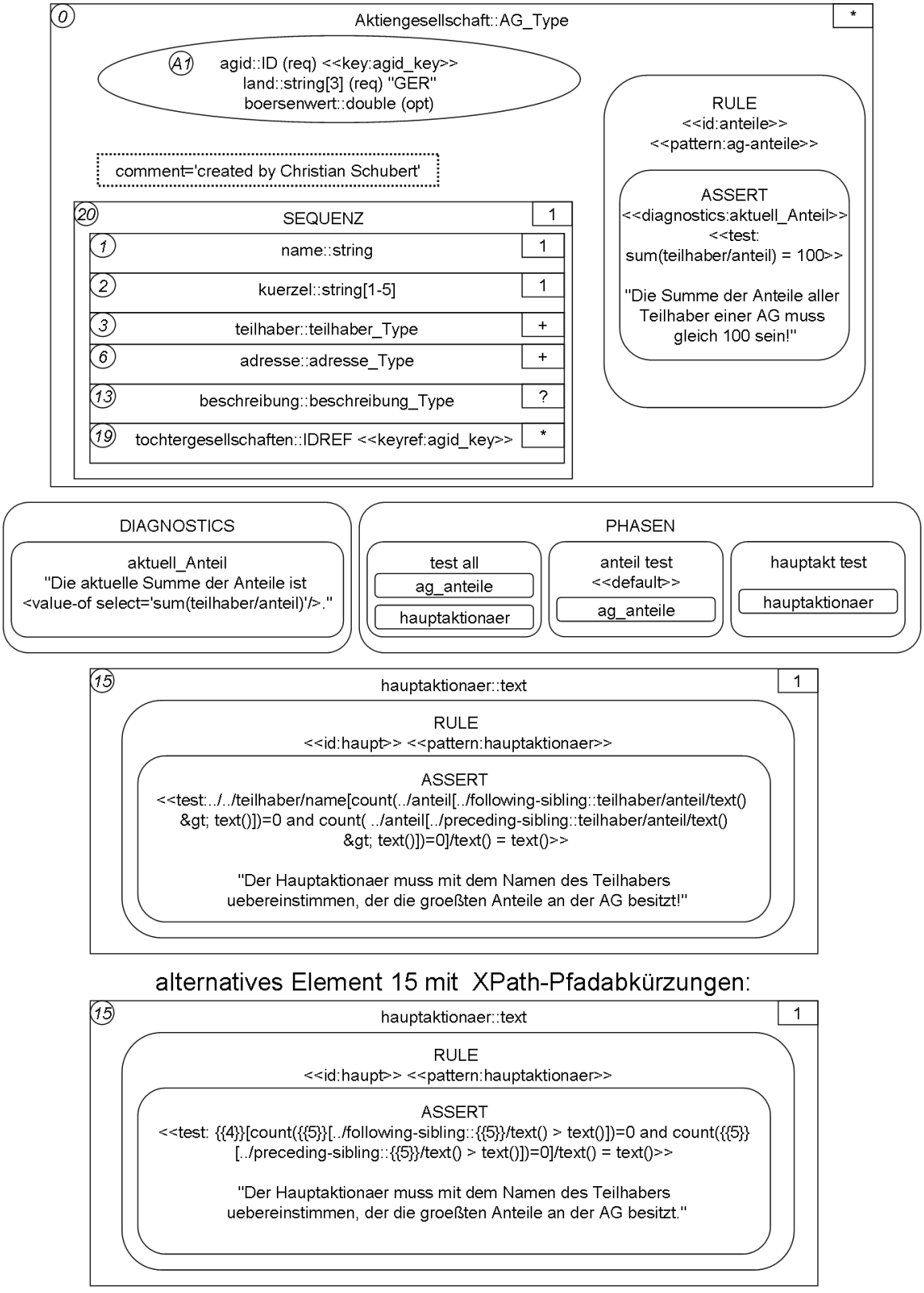


Abbildung 6.12: Beispiel für eigenes Schematron-Modell

Doppelpunkt abgetrennt, der Schlüsselname bzw. der Name, des Schlüssels auf den referenziert werden soll, angegeben.

Ein Beispiel für ein XML-Schema nach dem eigenen Modell zeigt die Abbildung 6.11.

Modell für Schematron

Für das schematronspezifische Modell, wurde ein neuer Modellelementtyp definiert. Dieser wird ebenfalls als ein Rechteck, also eine „Box“ aber mit abgerundeten Ecken, notiert. So wird es relativ einfach, die schematronspezifischen Modellelemente auch visuell heraus zu filtern. Diese Boxen können die folgenden Konzepte beinhalten:

- `Phasen`: Der Bezeichner dieser Box ist 'PHASEN', und, wie der Name schon sagt, werden hier alle Schematronphasen in separaten Unterboxen definiert. Diese Unterboxen besitzen alle einen Bezeichner. Weiterhin kann eine der Unterboxen auch die Markierung '«default»' zur Markierung einer Defaultphase beinhalten. In diesen Unterboxen werden durch weitere Unterboxen also Unterunterboxen die in der jeweiligen Phase zu aktivierenden Pattern notiert. Das 'PHASEN'-Element wird keinem speziellen Modellelement untergeordnet, sondern außerhalb der Strukturdefinition notiert.
- `Diagnostics`: Diese Boxen werden mit 'DIAGNOSTICS' betitelt und enthalten Unterboxen, in denen alle Diagnoseelemente eines Dokumentes spezifiziert werden. Dieses erfolgt in der Unterbox durch Angabe eines Bezeichners, der keine Leerzeichen enthalten darf, und der textuell in Anführungszeichen notierten Diagnosemeldung. Auch das 'DIAGNOSTICS'-Element wird keinem speziellen Element zugeordnet.
- `Rule`: Im Gegensatz dazu werden 'RULE'-Elemente entsprechenden Modellelementen zugeordnet, damit später der Kontext ('`context`-Attribut') im Schematron-Schema abgeleitet werden kann. Diese Modellelemente werden also mit 'RULE' bezeichnet. Über die Markierungen '«id»' und '«pattern»' lässt sich die ID des 'RULE'-Elementes festlegen und ein Pattern angegeben, dem dieses 'RULE'-Element untergeordnet ist. Des Weiteren kann die „Rulebox“ weitere Unterboxen enthalten. Diese können mit 'ASSERT' bzw. 'REPORT' betitelt werden. Diese Unterboxen haben auf jeden Fall die Markierung '«test»' und eventuell auch noch optional '«diagnostics»'. In der '«test»'-Markierung wird, durch Doppelpunkt abgetrennt, der XPath-Ausdruck angegeben, der den Regeltest spezifiziert. In '«diagnostics»' können, getrennt durch Leerzeichen, mehrere Diagnoseelemente referenziert werden. Dieses erfolgt über die Angabe der entsprechenden Diagnoseelementbezeichner. Die Fehlermeldung, die bei Verletzung der Report- und Asserttestausdücke ausgegeben werden soll, wird in der entsprechenden „Assert-“ bzw. „Report-Box“ in Anführungszeichen notiert.

Bei langen XPath-Ausdrücken z.B. bei der Markierung '«test»' wird vorgeschlagen, zur Abkürzung der Ausdrücke die IDs der Modellelemente zu verwenden. Dabei werden die IDs in spitzen Doppelklammern '{{ID}}' angegeben. Die IDs werden dann später im Quelltext wieder automatisch durch den direkten (kürzesten) Pfad vom Ausgangs- zum Zielpunkt ersetzt. Abbildung 6.12 zeigt das „AGs“-Beispiel für Schematron. Es werden hier nur die gegenüber XML-Schema abweichenden oder hinzu gekommenen Modellelemente gezeigt, da die anderen Modellbestandteile identisch mit denen von XML-Schema sind. Außerdem wird zusätzlich die Abkürzung der Notation durch '{{ID}}'-Konzept am Element mit der ID 15 gezeigt.

6.4 Entwurfsmethode & Transformationsprozess für CoDEX-Modelle

Im vereinfachten CoDEX lässt sich relativ leicht und komfortabel mit wenigen Modellelementen ein Grundmodell für ein XML-Schema erstellen. Diese Arbeit muss natürlich manuell vorgenommen werden, könnte aber durch ein entsprechendes Tool unterstützt werden. Ist das plattformunabhängige Modell „fertig“ editiert, kann mit der automatischen Transformation in die verschiedenen schemasprachenspezifischen Modelle begonnen werden.

6.4.1 Entwurfsmethode & Transformationsprozess für die Plattform DTD

Bei der `automatischen Überführung` vom plattformunabhängigen in ein DTD-spezifisches Modell wird wie folgt vorgegangen:

- Zuerst wird das komplette plattformunabhängige Modell übernommen.

- Es werden automatisch erzeugte Attributtypen zu allen Attributen einer jeden Attributbox hinzugefügt:
  - ist das Attribut als `<<unique>>` oder `<<key>>` gekennzeichnet ist Attributtyp `ID`
  - ist das Attribut als `<<keyref>>` gekennzeichnet ist Attributtyp `IDREF`
  - sonst ist Attributtyp erstmal `CDATA`
- Die Kardinalitätsangaben an den Kanten werden entsprechend DTD-Definition modifiziert:
  - [m-n] $\rightarrow$ [+] für $m > 0$ und $n \geq m$ und $n, m \in \mathbb{N}$
  - [0-1] $\rightarrow$ [?]
  - [0-n] $\rightarrow$ [*] für $n > 0$ und $n \in \mathbb{N}$
- Wurde ein Modellelement mittels `<<key>>` als Schlüssel gekennzeichnet, wird dieses zu `<<unique>>` korrigiert, da es in DTD keine „echten“ Schlüssel gibt. Ist das markierte Modellelement kein Attribut werden die Markierungen ganz verworfen.

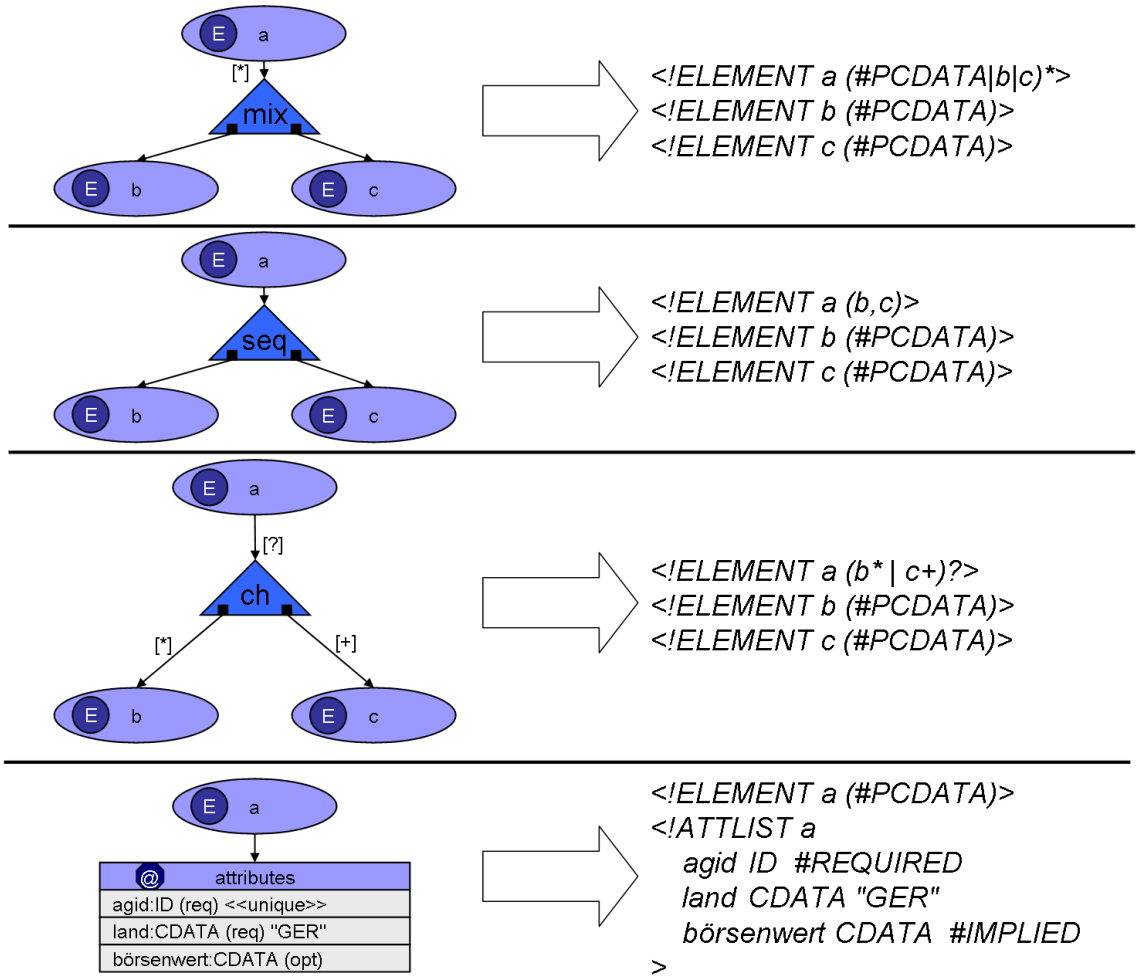


Abbildung 6.13: Transformation (-sschablonen) vom abhängigen CoDEX-DTD-Modell zu Code

Nun kann in dem gerade entstandenen und bezüglich DTD abhängigen Modell weiter modelliert werden. Möglich sind nun auch das Angeben von Notationen (Notations), Entities, bedingten Abschnitten und in Anmerkungen das Formulieren von Ausführungsanweisungen (Processing Instructions). Außerdem können die Attributtypen auch zu den in DTD erlaubten Typen (IDREF(S), Werteaufzählung usw.) geändert werden. Des Weiteren können natürlich auch neue Elemente, Attribute usw. hinzugefügt bzw. die bestehen Modellelemente abgeändert werden.

Ist man mit dem Ergebnis des Modellierens zufrieden und ist das Modell hinsichtlich der Definition für das DTD-abhängige Metamodell valide, kann die `Codeerzeugung` aus dem plattformabhängigen Modell

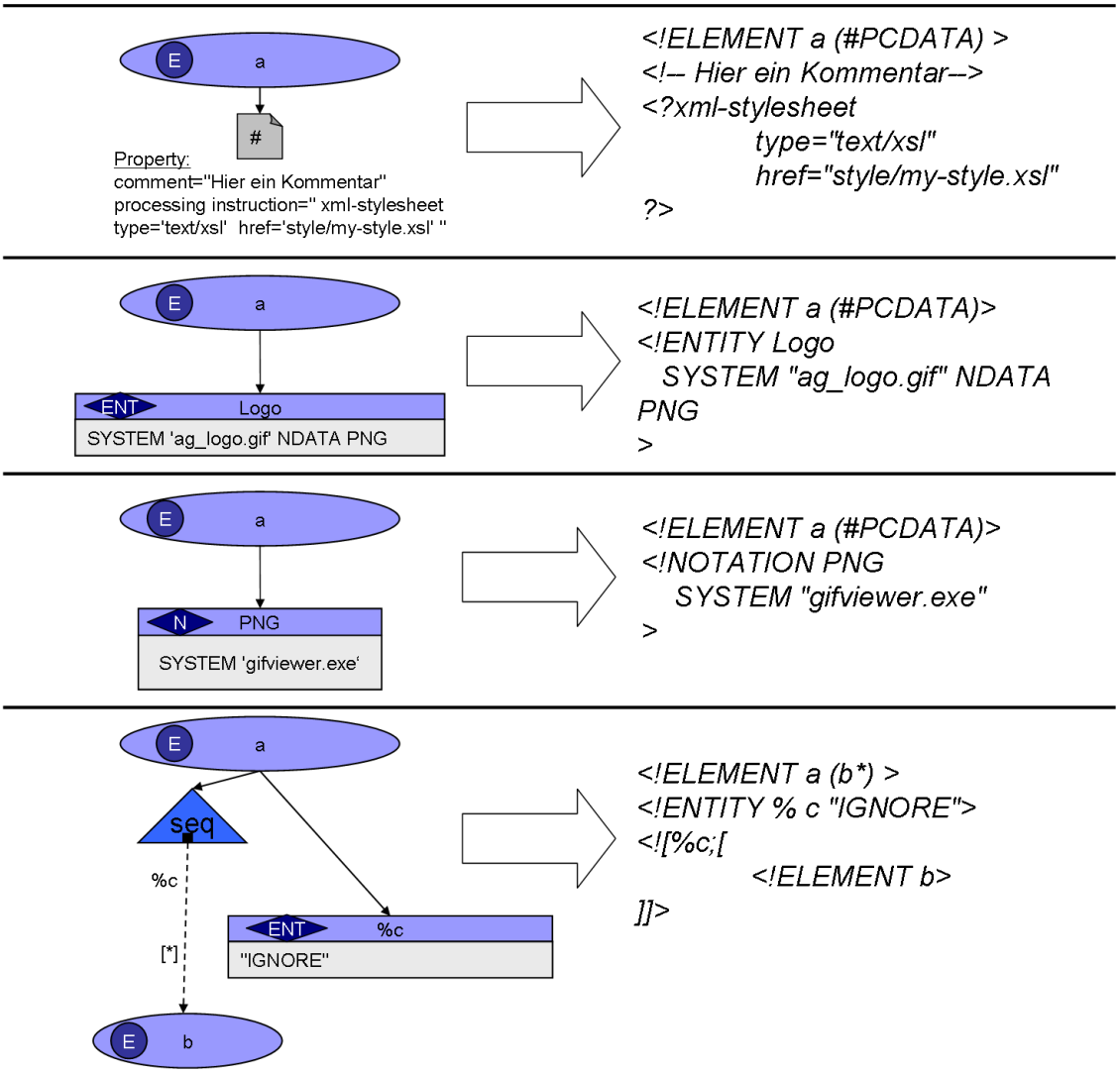


Abbildung 6.14: Transformation (-sschablonen) vom abhängigen CoDEX-DTD-Modell zu Code (Fortsetzung)

beginnen. Diese wird nun schablonenartig („Pattern-Matching“) vorgenommen werden. Dazu wird zu aller erst das Wurzelelement bestimmt, welche das ist, das keine eingehenden Kanten besitzt. Das Wurzelelement des Modells wird das Wurzelelement des DTD-Dokuments. Das beinhaltet also, dass der Name und das Inhaltsmodell des Wurzelelements des Modells auf das Wurzelelement des DTD-Dokuments abgebildet werden.

Die Abbildungen 6.13 und 6.14 verdeutlichen das Mapping der Modellelemente zum entsprechenden Quelltext. Dabei werden also ausgehend vom Wurzelelement „top down“ für jedes Modellelement die dazu gehörigen Werte und das zugehörige Inhaltsmodell bestimmt und in Quelltext übertragen. Dabei ist darauf zu achten, dass an mehren Stellen benutzte Modellelemente (vgl. Abb. 6.6 das Element ’name’) nicht mehrfach in den Code eingebaut werden.

Für unser CoDEX-DTD-Beispielmodell (vgl. Abb. 6.6) ergibt sich nach Anwenden der Codeschablonen also folgender Quellcode:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT AGs (Aktiengesellschaft*)>
<!ENTITY Osteraktion  "Die Osteraktion läuft vom 23.03.08 bis zum 01.04.08">
<!NOTATION PNG SYSTEM "gifviewer.exe" >
<!-- created by Christian Schubert -->
<?xml-stylesheet type="text/xsl"  href="style/my-style.xsl" ?>
<!ENTITY Logo  SYSTEM "ag_logo.gif" NDATA PNG>
```

```
<!ENTITY % tochter  "INCLUDE">

<!ELEMENT Aktiengesellschaft (name, kürzel, teilhaber+, adresse+,
        beschreibung?, tochtergesellschaften*)>
<!ATTLIST Aktiengesellschaft
        agid  ID  #REQUIRED
        land  CDATA "GER"
        börsenwert CDATA  #IMPLIED
>

<!ELEMENT name (#PCDATA)>
<!ELEMENT kürzel (#PCDATA)>
<!ELEMENT teilhaber (name, anteil)>
<!ELEMENT adresse (ort, ((plz, strasse) | postfach), adresszusatz?)>
<!ELEMENT beschreibung (#PCDATA | hauptaktionär | ort | anteil)*>
<![%tochter;[
        <!ELEMENT tochtergesellschaften (#PCDATA)>
]]>

<!ELEMENT anteil (#PCDATA)>

<!ELEMENT ort (#PCDATA)>
<!ELEMENT plz (#PCDATA)>
<!ELEMENT strasse (#PCDATA)>
<!ELEMENT postfach (#PCDATA)>
<!ELEMENT adresszusatz (#PCDATA)>

<!ELEMENT hauptaktionär (#PCDATA)>
```

6.4.2 Entwurfsmethode & Transformationsprozess für die Plattform XML-Schema

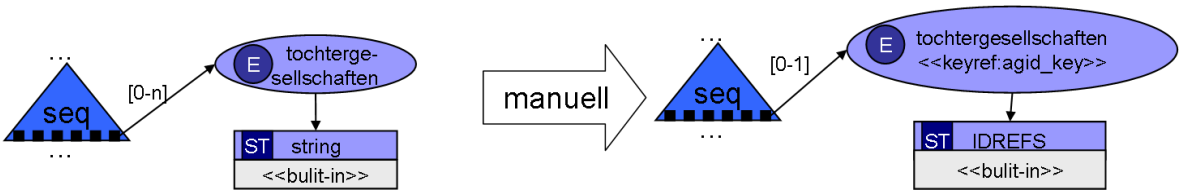


Abbildung 6.15: Modifikationen am CoDEX-XML-Schema-Modell

Als Grundlage für ein von der Plattform XML-Schema abhängiges Modell dient auch hier das plattformunabhängige Modell. Ähnlich wie beim CoDEX-Modell für die DTD wird beim CoDEX-Modell für XML-Schema das gesamte plattformunabhängige Modell erst einmal übernommen. Bei dieser `Transformation` werden außerdem noch folgende Schritte unternommen:

- Jedem Modellelement 'Element' wird ein Typ zugeordnet. Dieser wird als zusätzliches Modellelement zwischen Element-Knoten und dessen Subbaum eingefügt. Eventuell an Elemente angehängte Attributboxen werden nun, statt direkt dem Element-Knoten, dem entsprechenden Datentyp-Knoten zugeordnet. Ist bzw. sind einem Element Subelemente oder eine Attributbox zugeordnet, so wird der Datentyp als `CT` also `ComplexType` festgelegt. Als Bezeichner für das diese Komplextypen dienen der Modellelementname plus die Endung '_Type'. Hat ein Element dagegen keine Subelemente oder Attributboxen, so wird ein Modellelement `ST` also `SimpleType` mit dem Built-In-Typ `string` erzeugt und dem Element zugeordnet. Ausnahme bilden als `<<unique>>`, `<<key>>` oder `<<keyref>>` gekennzeichnete Elemente. Sie erhalten die Attributtypen `ID` für `<<unique>>` oder `<<key>>` bzw. `IDREF` für `<<keyref>>`.

- Jedem Attribut des Modellelementes 'Attributbox' wird ein automatisch erzeugter Attributtyp zugeordnet. Diese geschieht fast analog zu dem Vorgehen bei der Transformation vom plattformunabhängigen Modell zum DTD-abhängigen Modell:

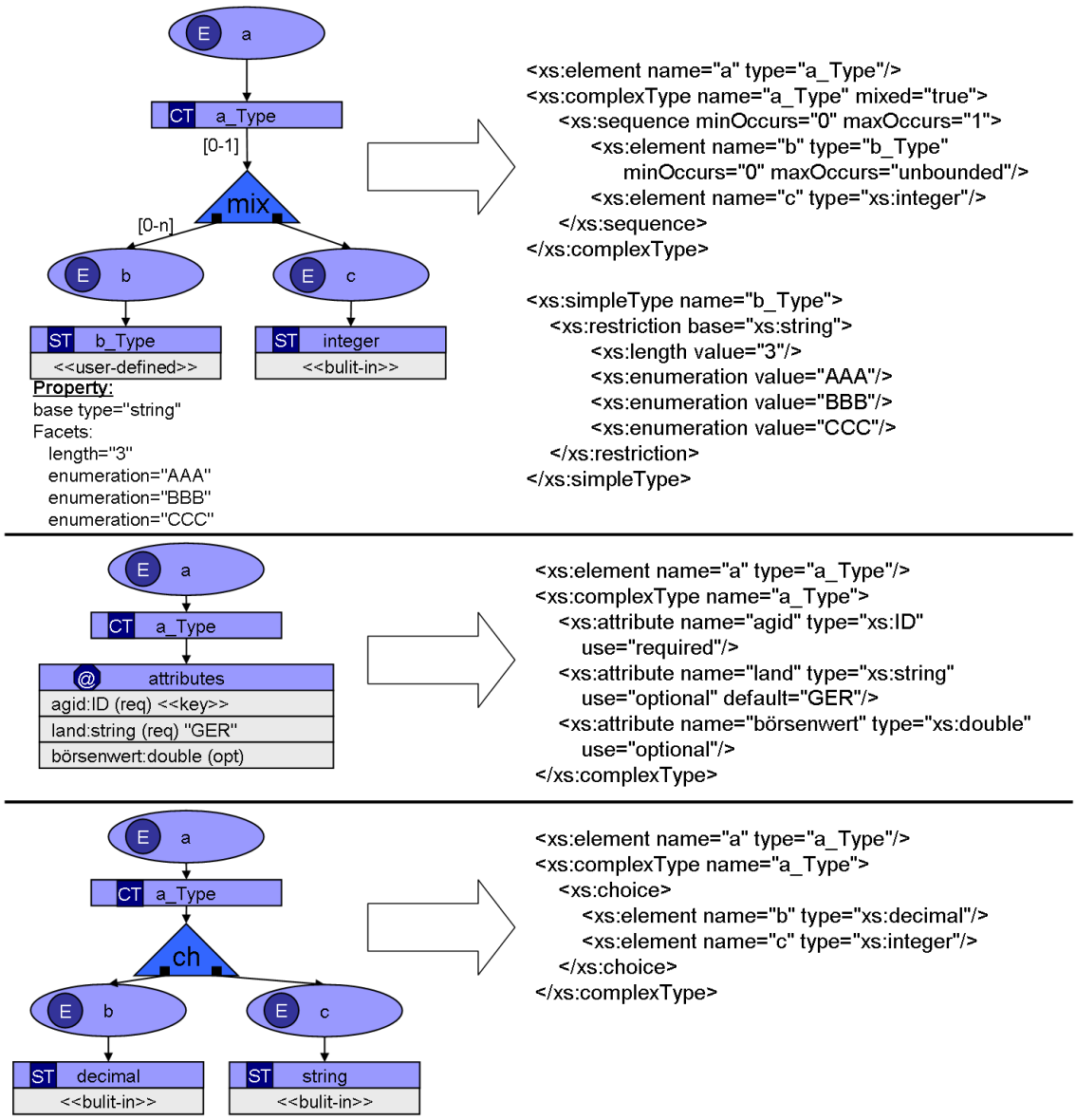


Abbildung 6.16: Transformation (-sschablonen) vom abhängigen CoDEX-XML-Schema-Modell zu Code

- ist das Attribut als `<<unique>>` oder `<<key>>` gekennzeichnet ist Attributtyp `ID`
- ist das Attribut als `<<keyref>>` gekennzeichnet ist Attributtyp `IDREF`
- sonst ist der Attributtyp `string`

Nach Überführung (Transformation) in ein von XML-Schema abhängiges Modell besteht die Möglichkeit der Bearbeitung des generierten Modells. Dabei ist auf jeden Fall eine Verfeinerung der Datentypen für Elemente und Attribute zu empfehlen. Erlaubt sind alle Built-In-Typen sowie selbst definierten Datentypen. Es können nun also auch neue selbstdefinierbare Datentypen ins Modell eingefügt werden. Das genaue Vorgehen entspricht dem in [ST06, insbesondere Kapitel 5.2] beschriebenen Ablauf.

Im Beispielmodell (vgl. Abb. 6.7) wurden z.B. die Datentypen vom Attribut 'börsenwert' sowie von den Elementen 'anteil' und 'plz' vom automatisch angelegten `string` auf `double` bzw. `integer` geändert. Außerdem wurde der automatisch generierte Typbezeichner vom Element 'Aktiengesellschaft' von 'Aktiengesellschaft_Type' auf 'AG_Type' gekürzt. Des Weiteren soll in Abbildung 6.15 noch eine kurze Modifikation aufgezeigt werden, in der das Element 'tochtergesellschaften' zu einem „`<<keyref>>`-Element" als Listentyp mit Datentyp `IDREFS` migriert. Dabei wird das Element Tochtergesellschaften als Fremdschlüssel vom Attribut 'agid' des Elementes 'Aktiengesellschaft' deklariert.

Die `Quellcodeerzeugung` erfolgt auch hier mittels „Pattern-Matching“, d.h. bestimmte Modell-Muster werden in entsprechenden Quelltext umgewandelt. Zuerst allerdings wird automatisch ein Coderahmen erzeugt, in dem ein Defaultprolog mit Angabe der XML-Version und dem `xs:schema`-Element angegeben wird. In dem `xs:schema`-Element wird der aus dem Modell gewonnene Quelltext eingebettet. Abbildung 6.16 zeigt beispielhaft die grundlegenden Mappings. Es sei noch mal auf die unterschiedliche Umformung von Built-In-Typen und selbstdefinierten Typen hingewiesen werden soll. Built-In-Typen brauchen, wie der Name schon sagt keine eigene Typdefinition und werden nur im Element mit dem entsprechenden Namensraumpräfix notiert. Selbstdefinierte Typen hingegen bekommen eine eigene Typdefinition „außerhalb“ des Schemaelementes. Eine Besonderheit stellen die `<<key>>`- bzw. `<<keyref>>`-Markierungen. Sie werden ins Wurzelelement des XML-Schemas umgesetzt. Dort werden also alle Schlüssel und Fremdschlüsselbeziehungen des Modells angegeben. Dazu müssen die entsprechenden Pfade aus dem Modell ermittelt und in die `xpath`-Attribute der jeweiligen `selector`- bzw. `field`-Schlüsselsubelemente übertragen werden. Ein Beispiel wird im folgenden Quellcode, das sich auf das Beispielmodell (vgl. Abb. 6.7) mit manueller Änderung (vgl. Abb. 6.15) bezieht, gezeigt:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">

  <xs:element name="AGs" type="AGs_Type">
    <xs:key name="agid_key">
      <xs:selector xpath="AGs/AG"/>
      <xs:field xpath="@agid"/>
    </xs:key>
    <xs:keyref name="tochtergesellschaften_keyref_agid_key" refer="agid_key">
      <xs:selector xpath="AGs/AG"/>
      <xs:field xpath="tochtergesellschaften"/>
    </xs:keyref>
  </xs:element>

  <!--created by Christian Schubert-->
  <?xml-stylesheet type="text/xsl"  href="style/my-style.xsl"?>

  <xs:complexType name="AGs_Type">
    <xs:sequence>
      <xs:element name="Aktiengesellschaft" type="AG_Type" minOccurs="0"
          maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="AG_Type">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="kürzel" type="xs:string"/>
      <xs:element name="teilhaber" type="teilhaber_Type" maxOccurs="unbounded"/>
      <xs:element name="adresse" type="adresse_Type" maxOccurs="unbounded"/>
      <xs:element name="beschreibung" type="beschreibung_Type" minOccurs="0"/>
      <xs:element name="tochtergesellschaften" type="xs:IDREFS" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="agid" type="xs:ID" use="required"/>
    <xs:attribute name="land" type="xs:string" use="optional" default="GER"/>
    <xs:attribute name="börsenwert" type="xs:double" use="optional"/>
  </xs:complexType>

  <xs:complexType name="teilhaber_Type">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="anteil" type="xs:integer"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="adresse_Type">
```

```
    <xs:sequence>
      <xs:element name="ort" type="xs:string"/>
      <xs:choice>
        <xs:sequence>
          <xs:element name="plz" type="xs:integer"/>
          <xs:element name="strasse" type="xs:string"/>
        </xs:sequence>
        <xs:element name="postfach" type="xs:string"/>
      </xs:choice>
      <xs:element name="adresszusatz" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="beschreibung_Type" mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="anteil" type="xs:integer"/>
      <xs:element name="ort" type="xs:string"/>
      <xs:element name="hauptaktionär" type="xs:string"/>
    </xs:choice>
  </xs:complexType>

</xs:schema>
```

6.4.3 Entwurfsmethode & Transformationsprozess für die Plattform Schematron

Da wie schon zuvor erwähnt der strukturbezogene Teil der Modellbeschreibung fürs CoDEX-Schematron-Modell, aus besagten Gründen (vgl. Abschnitt 6.3.1), in XML-Schema übersetzt wird, verläuft die Transformation dieser Modellteile analog zu der im vorherigen Abschnitt beschriebenen Vorgehensweise. Deshalb wird in diesem Abschnitt der Fokus auf die schematronspezifischen Modellbestandteile gelegt. Für das Schematronmodell werden also parallel zwei Quellcodeartefakte erzeugt, eins mit den vorwiegend syntaktischen Schemabestandteilen in XML-Schema und eins mit den eher semantischen Anteilen in Schematron. Auf die Beschreibung der Transformation vom plattformunabhängigen Modell zum von Schematron abhängigen Modell kann also an dieser Stelle verzichtet werden, weil keine semantischen bzw. schematronspezifischen Informationen im plattformunabhängigen Modell vorhanden sind. Das heißt, dass diese Informationen auch erst auf der schematronspezifischen Ebene modelliert werden. Die Schematronmodellelemente werden also ins XML-Schema-Modell an den entsprechenden Stellen eingefügt. Möglich ist hier das Hinzufügen der im Abschnitt 6.3.1 beschriebenen Entities, die einen Großteil der Möglichkeiten von Schematron „abdecken“.

Im `automatischen Transformationsprozess` vom schematronabhängigen Modell in den entsprechenden Code wird dann wie folgt vorgegangen:

- Erzeugung des Codegrundgerüst: Das `Codegrundgerüst` besteht aus einer Defaultangabe zur XML-Version und zur Zeichencodierung und dem Startelement '`<schema></schema>`', in das, der aus den entsprechenden Modellelementen zu erzeugende Quelltext, eingebettet wird. Des Weiteren wird als erstes Element im Schema ein '`<title>`'-Attribut mit einem Defaultnamen angelegt. Dieser wird nach dem Muster Name des Wurzelelementes des Modell + '_Schematronschema' erzeugt.

- Modell nach schematronspezifischen Modellelementen durchsuchen. Diese sind die drei Entities „Rule“, „Diagnostics“ und „Phase“ (vgl. 6.3.1).

- Aus diesen Modellelementen Quellcode erzeugen: Am aufwendigsten ist das Mapping der `Rule-Modellelemente`. Dazu müssen aus dem Modell vorher die Pfade der Elemente und Attribute ermittelt werden, denen schematronspezifische Rule-Modellelemente über gerichtete Kanten zugeordnet wurden. Dabei werden nur xpath-relevante Modellelemente des Modells berücksichtigt. Diese Pfade werden also in die Xpath-Pfadangaben im '`context`'-Attribut des '`rule`'-Elementes umgesetzt. Der im Kopf des „Rulebox“ angegebene Name ('ID') wird in das '`id`'-Attribut des '`rule`'-Elementes übertragen. Der in runden Klammern dahinter angegebene Bezeichner, entspricht der ID eines Pattern und damit der Zuordnung der Regel (Rule) zu einem Pattern. Existiert das so angegebene Pattern noch nicht, muss es natürlich zuerst erzeugt werden. Dabei wird der zwingend

anzugebende Name des Pattern erst einmal gleich der ID gesetzt. Im Inhalt „Rulebox“ können mindestens eine aber auch mehrere Assert- und Reportregeln definiert worden sein. Die Werte der Assert- und Report-'`test`'-Attribute des Schematron-Schemas werden direkt aus den in der „Rulebox“ textuell angegebenen XPath-Ausdrücken übernommen. Als Inhalt der '`assert`'- bzw. '`report`'-Elemente, wird eine Default-Fehlermeldung im Quellcode erzeugt, z.B.: “*RuleID*: !!!BITTE HIER FEHLERMELDUNG SPEZIFIZIEREN!!!“ Außerdem können von Asserts und Reports `Diagnose-Modellelemente` über gerichtete Kanten referenzieren. Ist dies der Fall, so wird im '`assert`'- bzw. '`report`'-Element ein '`diagnostics`'-Attribut generiert, in dem über die Angabe der ID auf das entsprechende '`diagnostics`'-Element verwiesen wird. Ist dieses '`diagnostics`'-Element noch nicht erzeugt worden, wird es entsprechend seiner Definition im Modell erzeugt. Dabei wird der in der Kopfzeile des '`diagnostics`'-Elementes angegebene Bezeichner als '`id`'-Attribut umgesetzt und der Inhalt des Modellelementes als Inhalt des Schemaelementes abgebildet. Die `Phasen-Modellelemente` sind, wie bereits erwähnt, direkt mit dem Wurzelelement des Modells über gerichtete Kanten verbunden. Ein Phasen-Modellelement wird, wie sich schon vermuten lässt, in ein '`phase`'-Element des Schemas überführt. Der Bezeichner des Phasen-Modellelementes wird in das '`id`'-Attribut des '`phase`'-Element gemappt. Für jede Inhaltszeile des Phasen-Modellelementes wird ein '`active`'-Element generiert und der Inhalt der Inhaltszeile wird als Wert dem '`pattern`'-Attribut des jeweiligen '`active`'-Elementes zugeordnet. Eines der Phasenelemente kann im Modell als Defaultphase markiert werden. Diese Markierung erzeugt im automatisch generierten Wurzelelement '`schema`' des Schematron-Schemas ein zusätzliches Attribut '`defaultphase`', welches als Wert die ID des so markierten Phasenelementes erhält.

Der folgende Quelltext zeigt, den aus dem Beispielmodell (vgl. Abb. 6.8) automatisch generierten Quellcode:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema defaultphase="anteil test">
  <title>AGs_Schematronschema</title>
  <phase id="test all">
    <active pattern="ag-anteile"/>
    <active pattern="hauptaktionaer"/>
  </phase>
  <phase id="haupakt test">
    <active pattern="hauptaktionaer"/>
  </phase>
  <phase id="anteil test">
    <active pattern="ag-anteile"/>
  </phase>
  <pattern name="ag-anteile" id="ag-anteile">
    <rule context="AGs/Aktiengesellschaft" id="anteile">
      <assert test="sum(teilhaber/anteil) = 100" diagnostics="aktuell_Anteil">
        anteile: !!!BITTE HIER FEHLERMELDUNG SPEZIFIZIEREN!!!
      </assert>
    </rule>
  </pattern>
  <pattern name="hauptaktionaer" id="hauptaktionaer">
    <rule context="AGs/Aktiengesellschaft/beschreibung/hauptaktionaer" id="haupt">
      <assert test="
        ../../teilhaber/name[
          count(
            ../anteil[../preceding-sibling::teilhaber/anteil/text() &gt; text()]
          )=0 and count(
            ../anteil[../following-sibling::teilhaber/anteil/text() &gt; text()]
          )=0
        ]/text() = text()">
          haupt: !!!BITTE HIER FEHLERMELDUNG SPEZIFIZIEREN!!!
      </assert>
    </rule>
  </pattern>
  <diagnostics>
    <diagnostic id="aktuell_Anteil">
```

```
      Die aktuelle Summe der Anteile ist <value-of select="sum(teilhaber/anteil)"/>.
    </diagnostic>
  </diagnostics>
</schema>
```

6.5 Entwurfsmethode & Transformationsprozess für eigene Modelle

Im Folgenden sollen noch die Entwurfsmethode und der Transformationsprozess, ähnlich wie zuvor schon für die CoDEX-Modelle, für das eigene Modell erklärt werden. Dabei wird anhand des bekannten Beispiels der ganze Entwicklungsprozess, vom plattformunabhängigem Modell über die plattformspezifischen Modelle hin zu den Quelltexten der unterschiedlichen Schemasprachen, anschaulich beschrieben.

6.5.1 Entwurfsmethode & Transformationsprozess für die Plattform DTD

Im ersten Schritt der Entwurfsmethode muss vom Anwender manuell ein plattformunabhängiges Modell für seine Anwendung erstellen. Aus diesem lässt sich dann automatisch ein `plattformspezifisches Grundmodell` erzeugen. Wie schon in Kapitel 6.3.2 beschrieben, werden dabei die Typangaben für Elemente entfernt, sofern diese nicht 'ANY' oder 'EMPTY' sind, da es nur den Datentyp `PCDATA` in DTD gibt. Gleiches gilt für die Längenbeschränkungen aller Datentypen.

An diesem Grundmodell kann dann weiter modelliert werden, d.h., es können weitere Elemente und Attribute hinzugefügt oder geändert werden usw.. Auch die Definition von „Bedingten Abschnitten“ ist nun wie in Kapitel 6.3.2 beschrieben möglich. Im Beispielmodell (vgl. Abb. 6.9 und 6.10) wurden so z.B. zwei Attribute ergänzt und ein „Bedingter Abschnitt“ definiert.

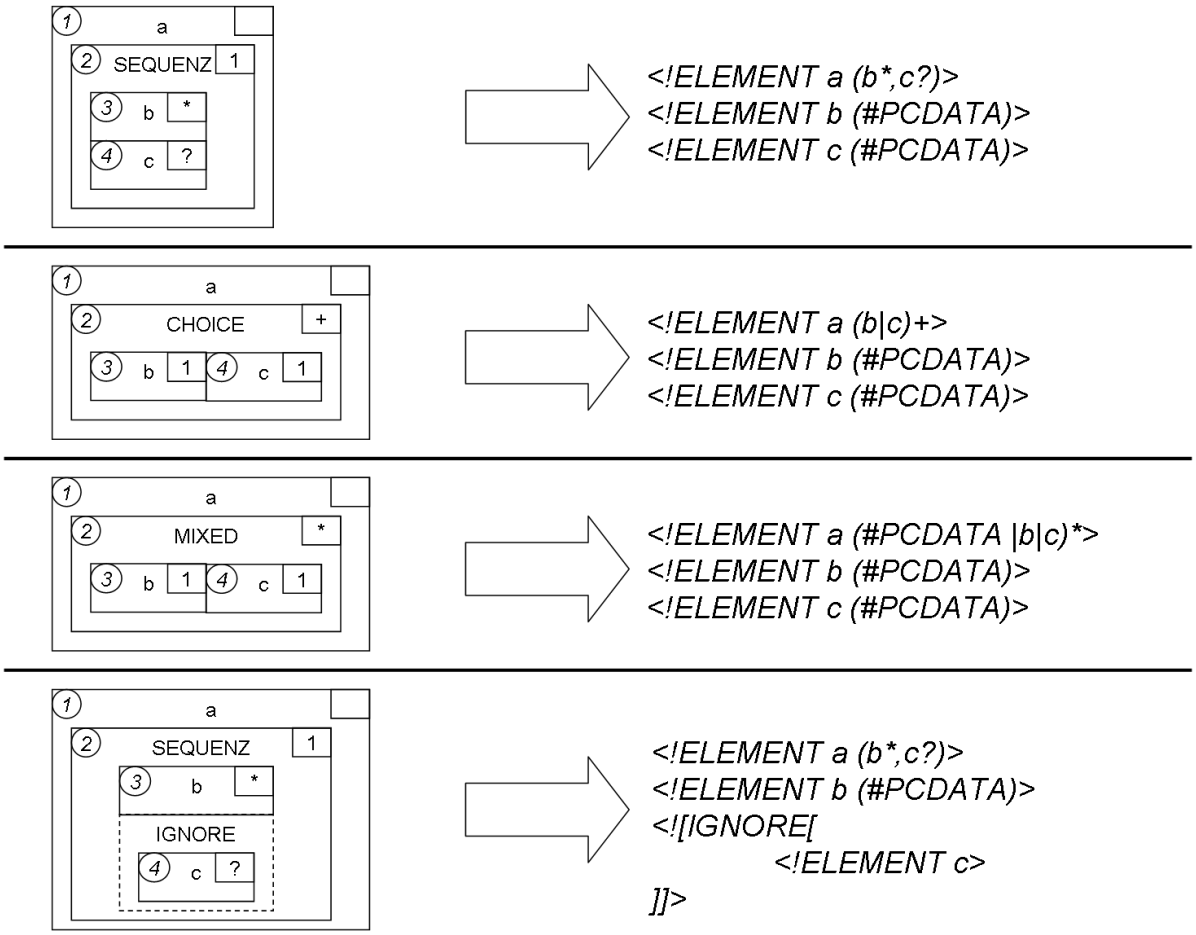


Abbildung 6.17: Transformation (-sschablonen) für eigenes DTD-Modell

Wenn man das plattformabhängige DTD-Modell „fertig“ modelliert hat, kann also der `Transformationsprozess hin zum Quelltext` gestartet werden. Dieser Prozess ist relativ leicht zu beschreiben, da der Quellcode schon fast 1:1 aus dem Modell ablesbar ist, insbesondere wenn die Komponentensicht vorliegt. Somit kann der Quelltext leicht über Codeschablonen (6.17) erzeugt werden. Dabei werden ausgehend vom Wurzelelement „top down“ für jedes Modellelement die dazu gehörigen Markierungen, Werte und das zugehörige Inhaltsmodell bestimmt und in Quelltext übertragen. Es ist darauf zu achten, dass an mehren Stellen benutzte Modellelemente nicht mehrfach in den Code eingebaut werden. Als Codegrundgerüst werden als erste Zeile des Quellcodes Angaben zur XML-Version und der Zeichencodierung ergänzt.

Aus dem Beispielmodell (vgl. Abb. 6.9 und 6.10) lässt sich also folgender Quelltext automatisch generieren:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT AGs (Aktiengesellschaft*)>

<!ELEMENT Aktiengesellschaft (name, kuerzel, teilhaber+, adresse+,
        beschreibung?, tochtergesellschaften*)>
<!-- created by Christian Schubert -->
<!ATTLIST Aktiengesellschaft
        agid  ID  #REQUIRED
        land  CDATA "GER"
        boersenwert CDATA  #IMPLIED
>

<!ELEMENT name (#PCDATA)>
<!ELEMENT kuerzel (#PCDATA)>
<!ELEMENT teilhaber (name, anteil)>
<!ELEMENT adresse (ort, ((plz, strasse) | postfach), adresszusatz?)>
<!ELEMENT beschreibung (#PCDATA | hauptaktionaer | ort | anteil)*>
<![IGNORE[
        <!ELEMENT tochtergesellschaften (#PCDATA)>
]]>

<!ELEMENT anteil (#PCDATA)>

<!ELEMENT ort (#PCDATA)>
<!ELEMENT plz (#PCDATA)>
<!ELEMENT strasse (#PCDATA)>
<!ELEMENT postfach (#PCDATA)>
<!ELEMENT adresszusatz (#PCDATA)>

<!ELEMENT hauptaktionaer (#PCDATA)>
```

6.5.2 Entwurfsmethode & Transformationsprozess für die Plattform XML-Schema

Ausgangspunkt des Entwurfsprozesses ist auch hier ein manuell erstelltes plattformunabhängiges Modell. Aus diesem Modell wird automatisch ein bezüglich der Plattform XML-Schema abhängiges `Grundmodell` erzeugt. Dabei werden alle Modellelemente vom plattfomunabhängigen Modell übernommen und folgende Schritte ausgeführt:

- Alle Elemente, die bisher auf ’text’ getypt waren, werden zu ’string’ gewandelt.
- Alle Elemente, die bisher den Typ ’num’ besaßen, erhalten den Typ ’integer’.
- Alle Elemente, die bisher keinen Typ sondern nur Unterelemente bzw. Attribute besaßen oder vom Typ ’EMPTY’ sind, erhalten einen Komplexen Typen mit dem Namen *eigenerElementname* + ’_Type’.
- Für alle Attribute der Typen ’ID’,’IDREF’,’IDREFS’, ’NMTOKEN’, ’NMTOKENS’, ’ENTITY’, ’ENTITIES’ und ’NOTAION’ bleiben die Typen erhalten. Nur Attribut vom Typ ’CDATA’ werden in Attribute vom Typ ’string’ gewandelt.

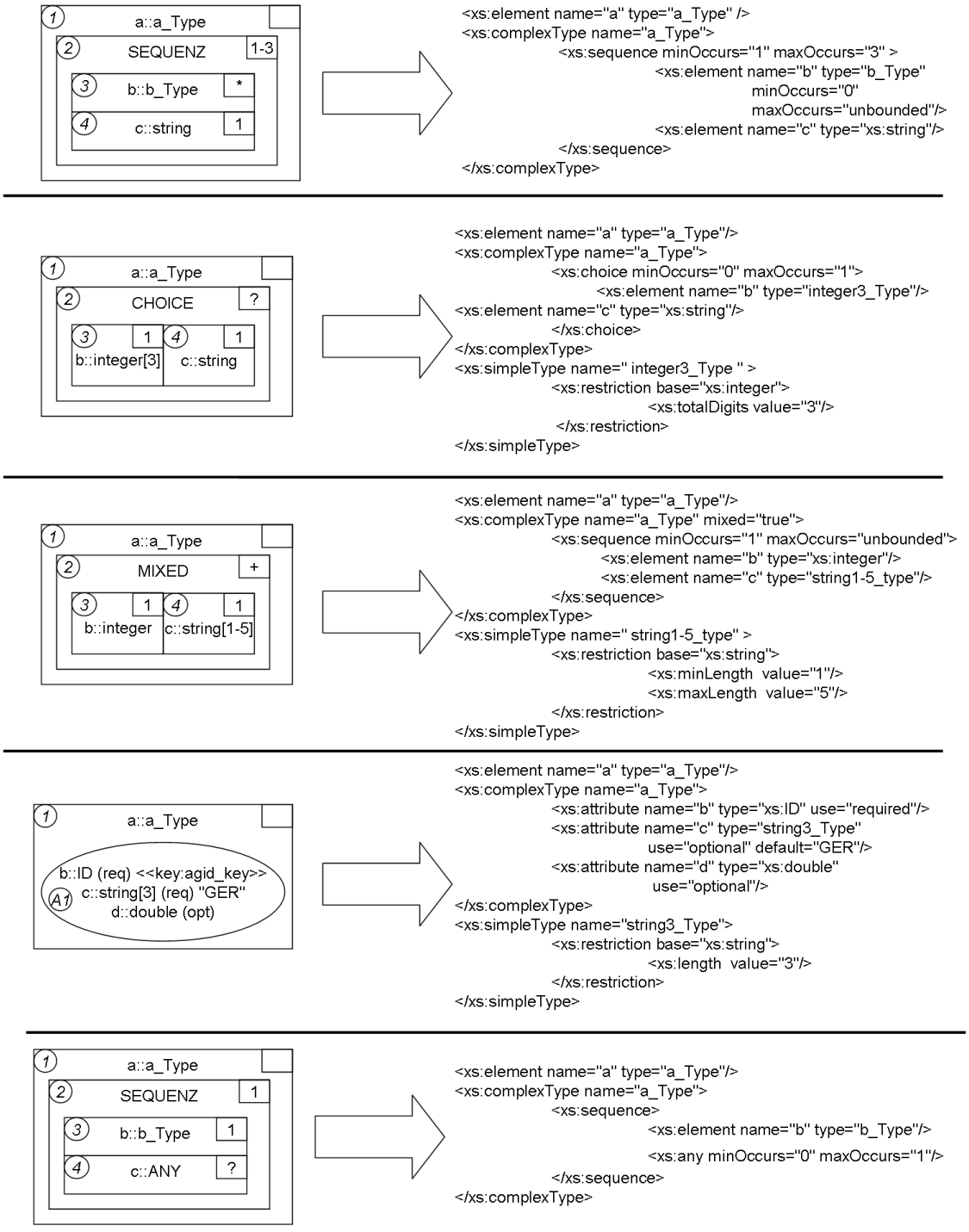


Abbildung 6.18: Transformation (-sschablonen) für eigenes XML-Schema-Modell

Nach der automatischen Transformation kann auch hier am Modell weiter modelliert werden. Es bieten sich insbesondere Typverfeinerungen an, da von XML-Schema jede Menge von Standarddatentypen unterstützt werden und ein umfassendes Typendefinitionskonzept zur Verfügung steht. Auch die Verfeinerung der Kardinalitäten (Quantoren) und Schlüssel bzw. Fremdschlüsseldefinitionen können hier nun entsprechend der Definition aus Kapitel 6.3.2 modelliert werden. Im Beispielmodell(6.11) wurden gegenüber dem plattformunabhägigem Modell noch zwei weitere Attribute ergänzt und jeweils ein Schlüssel bzw. Fremdschlüssel definiert. Außerdem wurde die Auftretenshäufigkeit vom Element 'adresse' auf minimal ein und maximal fünf Vorkommen verfeinert.

Die `Transformation` vom plattformspezifischen `Modell zum Code` erfolgt wieder mittels spezieller Codeschablonen (siehe 6.18). Zuerst allerdings wird automatisch ein Coderahmen erzeugt, in dem ein Defaultprolog mit Angabe der XML-Version und dem `xs:schema`-Element angegeben wird. In dem `xs:schema`-Element wird der aus dem Modell gewonnene Quelltext eingebettet. Hierbei ist zu beachten, dass die Umformung von Built-In-Typen und selbstdefinierten Typen unterschiedlich vorgenommen wird. Dabei gelten auch schon per eckigen Klammern eingeschränkte Built-In-Typen, als selbstdefinierte einfache Typen ('simpleType') und werden in entsprechende selbstdefinierte Typen überführt. Eine weitere Besonderheit stellen wieder die `<<key>>`- bzw. `<<keyref>>`-Markierungen. Sie werden ins Wurzelelement des XML-Schemas umgesetzt. Dort werden also alle Schlüssel und Fremdschlüsselbeziehungen des Modells angegeben. Dazu müssen die entsprechenden Pfade aus dem Modell ermittelt und in die `xpath`-Attribute der jeweiligen `selector`- bzw. `field`-Schlüsselsubelemente übertragen werden.

Ein Beispiel wird im folgenden Quellcode, das sich auf das Beispielmodell (vgl. Abb. 6.11) bezieht, gezeigt:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">

  <xs:element name="AGs" type="AGs_Type">
    <xs:key name="agid_key">
      <xs:selector xpath="AGs/AG"/>
      <xs:field xpath="@agid"/>
    </xs:key>
    <xs:keyref name="tochtergesellschaften_keyref_agid_key" refer="agid_key">
      <xs:selector xpath="AGs/AG"/>
      <xs:field xpath="tochtergesellschaften"/>
    </xs:keyref>
  </xs:element>

  <xs:complexType name="AGs_Type">
    <xs:sequence>
      <xs:element name="Aktiengesellschaft" type="AG_Type" minOccurs="0"
          maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="AG_Type">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="kuerzel" type="string1-5_Type"/>
      <xs:element name="teilhaber" type="teilhaber_Type" maxOccurs="unbounded"/>
      <xs:element name="adresse" type="adresse_Type" maxOccurs="unbounded"/>
      <xs:element name="beschreibung" type="beschreibung_Type" minOccurs="0"/>
      <xs:element name="tochtergesellschaften" type="xs:IDREFS" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="agid" type="xs:ID" use="required"/>
    <xs:attribute name="land" type="string3_Type" use="optional" default="GER"/>
    <xs:attribute name="boersenwert" type="xs:double" use="optional"/>
  </xs:complexType>

  <!--created by Christian Schubert-->

  <xs:simpleType name="string1-5_Type">
```

```
  <xs:restriction base="xs:string">
    <xs:minLength  value="1"/>
    <xs:maxLength  value="5"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="string3_Type">
  <xs:restriction base="xs:string">
    <xs:length  value="3"/>
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="teilhaber_Type">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="anteil" type="xs:integer"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="adresse_Type">
  <xs:sequence>
    <xs:element name="ort" type="string2-50_Type"/>
    <xs:choice>
      <xs:sequence>
        <xs:element name="plz" type="integer5_Type"/>
        <xs:element name="strasse" type="string0-50_Type"/>
      </xs:sequence>
      <xs:element name="postfach" type="xs:string"/>
    </xs:choice>
    <xs:element name="adresszusatz" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:simpleType name="string2-50_Type">
  <xs:restriction base="xs:string">
    <xs:minLength  value="2"/>
    <xs:maxLength  value="50"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="integer5_Type">
  <xs:restriction base="xs:integer">
    <xs:totalDigits value="5"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="string0-50_Type">
  <xs:restriction base="xs:string">
    <xs:minLength  value="0"/>
    <xs:maxLength  value="50"/>
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="beschreibung_Type" mixed="true">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="anteil" type="xs:integer"/>
    <xs:element name="ort" type="xs:string"/>
    <xs:element name="hauptaktionaer" type="xs:string"/>
  </xs:choice>
</xs:complexType>
```

```
</xs:schema>
```

6.5.3 Entwurfsmethode & Transformationsprozess für die Plattform Schematron

Für das eigene Modell der Plattform Schematron wird, wie auch schon beim CoDEX-Modell, der strukturbezogene Teil der Modellbeschreibung aus den bekannten Gründen (vgl. Abschnitt 6.3.1) in XML-Schema übersetzt. Die Transformation dieser Modellteile verläuft also analog zu der im vorherigen Abschnitt beschriebenen Vorgehensweise für XML-Schema. In diesem Abschnitt wird also nur auf die schematronspezifischen Modellbestandteile eingegangen. Für das Schematronmodell werden also parallel zwei Quellcodeartefakte erzeugt, eins mit den vorwiegend syntaktischen Schemabestandteilen in XML-Schema und eins mit den eher semantischen Anteilen in Schematron. Auf die Beschreibung des Transformationsprozesses vom plattformunabhängigen Modell zum von Schematron abhängigen Modell kann also an dieser Stelle verzichtet werden, da keine schematronspezifischen Informationen im plattformunabhängigen Modell vorhanden sind. Somit verläuft der erste Transformationsschritt analog zu dem bei XML-Schema. Das bedeutet aber auch, dass die schematronabhängigen Informationen auch erst auf der schematronspezifischen Modellebene modelliert werden können. Die Schematronmodellelemente werden also, ins bisher nur mit XML-Schema-Modellelementen bestücktem Modell, an den entsprechenden Stellen eingefügt. Dazu werden die im Abschnitt 6.3.2 erklärten Modellelemente verwendet.

Nach der Modellierung der Schematronmodellelemente folgt der `automatisches Transformationsprozess` vom schematronabhängigem Modell in den entsprechenden Quelltext. Auch hier wird zuerst ein `Codegrundgerüst` angelegt. Es besteht aus einer Defaultangabe zur XML-Version und zur Zeichencodierung und dem Startelement '`<schema></schema>`', in das, der aus den schematronspezifischen Modellelementen erzeugte Quelltext, eingebettet wird. Des Weiteren wird als nächstes Element im Schema ein '`<title>`'-Attribut mit einem Defaultnamen angelegt. Dieser wird nach dem Muster *Name des Wurzelelementes des Modell* + '_Schematronschema' erzeugt. Danach muss das Modell nach schematronspezifischen Modellelementen durchsucht werden, aus denen dann der entsprechende Quellcode generiert werden kann.

Dazu werden zuerst die `RULE-Modellelemente` aus dem Modell gefiltert. Dann müssen die Pfade der XML-Schemaelemente bestimmt werden, in denen die jeweiligen „Rule"-Modellelemente enthalten sind. Pfade bedeutet hier, die Verschachtelungsreihenfolge der „Elementboxen" und „Attributellipsen". Diese Pfade werden also in die XPath-Pfadangaben im '`context`'-Attribut des '`rule`'-Elementes umgesetzt. Der Wert der Markierung '«id»' der „Rulebox" wird in das '`id`'-Attribut des '`rule`'-Elementes übertragen. Der Wert der Markierung '«pattern»' entspricht dagegen der ID eines Pattern und damit der Zuordnung der Regel (Rule) zu einem Pattern. Wurde das so angegebene Pattern noch nicht generiert, muss es zuerst erzeugt werden. Dabei wird der zwingend anzugebende Name des Pattern erst einmal gleich der ID gesetzt. Im Inhalt der „Rulebox" können mindestens eine aber auch mehrere Assert- und Reportregeln über entsprechende „Boxen" definiert worden sein. Die Werte der Markierungen '«test»' und '«diagnostics»' der „Assert"- und „Reportboxen" des Modells spezifizieren die Werte der Attribute '`test`' und '`diagnostics`' der '`assert`'- bzw. '`report`'-Elemente des Schematron-Schemas. Als Inhalt der '`assert`'- bzw. '`report`'-Elemente wird die in der „Assertbox" bzw. „Reportbox" in Anführungszeichen textuell angegebene Fehlermeldung in den Quellcode übernommen.

Das `PHASEN-Modellelement` enthält alle definierten Phasen in Unterboxen notiert. Die einzelnen Phasen werden in entsprechende '`phase`'-Elemente des Schemas überführt. Dabei wird der Bezeichner der „Phasenunterboxen" in das '`id`'-Attribut des '`phase`'-Element gemappt. Für jede Unterbox einer einzelnen Phase wird ein '`active`'-Element generiert und der Inhalt der Unterbox wird als Wert dem '`pattern`'-Attribut des jeweiligen '`active`'-Elementes zugeordnet. Eine der Phasen kann im Modell als Defaultphase markiert worden sein. Diese Markierung erzeugt im automatisch generierten Wurzelelement '`schema`' des Schematron-Schemas ein zusätzliches Attribut '`defaultphase`', welches als Wert die ID des so markierten Phasenelementes erhält.

Das `DIAGNOSTICS-Modellelement` kann optional auch auftreten. Ist dies der Fall, so wird im Schematron-Schema ein entsprechendes '`diagnostics`'-Element angelegt. In diesem '`diagnostics`'-Element wird dann für jede Unterbox des „DIAGNOSTICS"-Modellelementes ein '`diagnostic`'-Element eingefügt. Diese '`diagnostic`'-Elemente erhalten ihre Werte und den Inhalt aus den Unterboxen des „DIAGNOSTICS"-Modellelementes. Die Bezeichner der Unterboxen werden auf das '`id`'-Attribut umgesetzt und der in den

Unterboxen in Anführungszeichen angegebene Diagnosetext wird der Inhalt des '`diagnostic`'-Elementes. Der folgende Quelltext zeigt den aus dem Beispielmodell (vgl. Abb. 6.12) automatisch generierten Quellcode, der dann noch weiter bearbeit oder ergänzt werden kann:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema defaultphase="anteil test">
  <title>AGs_Schematronschema</title>
  <phase id="test all">
    <active pattern="ag-anteile"/>
    <active pattern="hauptaktionaer"/>
  </phase>
  <phase id="haupakt test">
    <active pattern="hauptaktionaer"/>
  </phase>
  <phase id="anteil test">
    <active pattern="ag-anteile"/>
  </phase>
  <pattern name="ag-anteile" id="ag-anteile">
    <rule context="AGs/Aktiengesellschaft" id="anteile">
      <assert test="sum(teilhaber/anteil) = 100" diagnostics="aktuell_Anteil">
        Die Summe der Anteile aller Teilhaber einer AG muss gleich 100 sein!
      </assert>
    </rule>
  </pattern>
  <pattern name="hauptaktionaer" id="hauptaktionaer">
    <rule context="AGs/Aktiengesellschaft/beschreibung/hauptaktionaer" id="haupt">
      <assert test="
        ../../teilhaber/name[
          count(
            ../anteil[../preceding-sibling::teilhaber/anteil/text() &gt; text()]
          )=0 and count(
            ../anteil[../following-sibling::teilhaber/anteil/text() &gt; text()]
          )=0
        ]/text() = text()">
          Der Hauptaktionaer muss mit dem Namen des Teilhabers
          uebereinstimmen, der die groeßten Anteile an der AG besitzt!
      </assert>
    </rule>
  </pattern>
  <diagnostics>
    <diagnostic id="aktuell_Anteil">
      Die aktuelle Summe der Anteile ist <value-of select="sum(teilhaber/anteil)"/>.
    </diagnostic>
  </diagnostics>
</schema>
```

Kapitel 7

Evaluation und Ausblick

Im nun folgenden abschließenden Kapitel sollen die hier zuvor vorgestellten Modelle und Konzepte noch einmal kritisch betrachtet und ein Ausblick auf möglich Verbesserungen und Weiterentwicklungen gegeben werden.

7.1 Bewertung der Modelle und Entwurfsmethode

Sowohl das CoDEX-Modell als auch das eigene Modell und deren in dieser Arbeit eingeführten Modifikationen für die Schemasprachen DTD, XML-Schema und Schematron sind eine relativ kompakte und durch die wenigen Modellelemente leicht erlernbare Beschreibungsmöglichkeit für XML-Schemata. Abstriche dagegen mussten hinsichtlich der Vollständigkeit der Beschreibungsmöglichkeiten gemacht werden. So können aus Übersichtlichkeitsgründen eine Reihe von möglichen aber eher selten genutzten Eigenschaften nicht direkt im Modell dargestellt werden. Dies betrifft in XML-Schema vor allem Facetten zum Einschränken von Typen, die nur zum Teil oder außerhalb des eigentlichen graphischen Modells (CoDEX-Property-View) darstellbar sind, und einge Attribute von Schemaelementen. In Schematron betrifft das vorwiegend einige Attribute, wie z.B. das `'role'`-Attribut, von Schemaelementen.

Des Weiteren als etwas kritisch zu betrachten ist das „Pfeilewirrwarr“, was durchaus relativ schnell zu einer gewissen Unübersichtlichkeit in den CoDEX-Modellen führt. Auch die fürs CoDEX-Schematron-Modell notwendigen XPath-Ausdrücke sind bei längeren bzw. komplizierteren Ausdrücken störend. Deswegen sollten vielleicht alle Modellelemente eines Modells durchnummeriert werden, ähnlich wie dies im eigenen Modell gemacht wurde. Die dadurch entstandenen IDs könnten dann im Modellelement mit angezeigt werden. Mit Hilfe dieser IDs ließen sich dann z.B. Pfad- oder Referenzausdrücke verkürzen. Das eigene Modell dagegen wird selbst für kleine Schemata sehr schnell relativ voluminös, was sich negativ auf die Übersichtlichkeit auswirkt.

Problematisch ist auch die Konsistenzhaltung der verschiedenen Modelle und des Codes zu einander, wenn manuelle Änderungen an einem Modell oder am Quelltext vorgenommen werden. Wird zum Beispiel im plattformabhängigen Modell ein Detail verändert, das auch das plattformunabhängige Modell betrifft, dann muss dafür gesorgt werden, dass diese Änderung auch ins plattformunabhängige Modell übernommen wird. Sonst besteht die Gefahr das bei erneuter Erzeugung des plattformabhängigen Modell aus dem unabhängigen Modell gemachte Änderungen verloren gehen, d.h. überschrieben werden. Dieses Problem ist auch schon aus der MDA bekannt und es gibt dafür verschiedene Lösungsansätze.

Da in der MDA die UML als Modellierungssprache vorgeschalgen wird, wäre auch eine UML-Konformität der Modelle wünschenwert. Diese Konformität wurde aber noch nicht vollständig hergestellt, auch wenn die Notation im eigenen Modell zum Teil schon sehr UML-nah ist.

7.2 Fazit & Ausblick

Die hier vorgestellten Modelle bieten eine durchaus interessante und komfortable Möglichkeit der Modellierung von XML-Schemata, wobei auch noch Weiterentwicklungspotential vorhanden ist. So könnte man z.B. das CoDEX-Modell für XML-Schema noch in soweit verfeinern, dass bei Typdefinitionen neben Angaben wie `<<user-def>>` usw. auch explizite Angaben zur Vererbung wie z.B. `<<restricts>>: xyz` oder `<<extends>>: xyz` möglich sind. Diese Angaben sollen dann bedeuten, dass vom Typ 'xyz' geerbt

wurde und der definierte Typ nun eine Erweiterung bzw. eine Einschränkung des Typs 'xyz' ist.

Ein empfehlenswerter nächster Schritt wäre die Implementation eines Tools, mit dem sich die Modelle komfortabel erzeugen und editieren lassen und die den Transformatiosprozess automatisieren bzw. den Anwender bei Transformationsprozess unterstützen. Denkbar wären dann auch eine Benutzerabfrage bei Typmigrationen bzw. bei Typverfeinerungen. Dazu bietet sich eine Weiterentwicklung des CoDEX-Editors an, der bisher nur auf XML-Schema ausgerichtet ist.

Ein weiteres Thema könnte eine konkretere Betrachtung der Möglichkeit der Rücktransformation, d.h. von Code zum Modell oder vom „Untermodell" zum „Obermodell", sein. Horizontale Transformationen, also Transformationen von einem plattformabhängigem Modell in ein anderes plattformabhängiges Modell einer konkurierenden Plattform, wären ebenfalls untersuchenswert.

Die Herstellung einer UML-Konformität für die Modelle wäre, wie schon in vorherigen Abschnitt erwähnt, auch eine denkbare und interessante Erweiterungsmöglichkeit für diese Arbeit.

Es würde sich evetuell auch noch anbieten, andere XML-Schemasprachen zu unterstützen. Auch sehr interessant dürfte die Möglichkeit der Erweiterung bzw. Übertragung der hier beschriebenen Konzepte in andere Bereiche sein. So ist z.B. in der Datenaustauschsprache Java Script Object Notation (JSON) [JSON], gesprochen wie der Name „Jason", eine ähnliche Entwicklung bezüglich Schemata zu beobachten. Es wäre durchaus auch vorstellbar, aus einem plattformunabhängigen Modell ein JSON-spezifisches Modell und auch JSON-Code zu erzeugen.

Literaturverzeichnis

[BECK02] Oliver Becker, „XML-Halbkurs: Schematron“, Sommersemester 2002, http://www2.informatik.hu-berlin.de/~obecker/Lehre/SS2002/XML/09-schematron.html, Abrufdatum: 30.01.2008

[CWM] „Data Warehousing, CWM and MOF Resource Page“, http://www.omg.org/cwm/, Abrufdatum: 24.01.2008

[DLWC00] Dongwon Lee, Wesley W. Chu, „Comparative Analysis of Six XML Schema Languages“, Juni 2000, http://www.cobase.cs.ucla.edu/tech-docs/dongwon/ucla-200008.html, Abrufdatum: 30.01.2008

[HACK] Rick Jelliffe, Simon St.Laurent „Schematron: An Interview with Rick Jelliffe“, 15. November 1999, http://xmlhack.com/read.php?item=121, Abrufdatum: 26.01.2008

[JSON] http://www.json.org/, Abrufdatum: 24.03.2008

[KLME02] Meike Klettke, Holger Meyer, „XML und Datenbanken - Konzepte, Sprachen und Systeme“, Dpunkt Verlag, Auflage: 1 (Dezember 2002), ISBN 3898641481, http://www.xml-und-datenbanken.de/folien_buch.html

[MIMU01] J. Miller, J. Mukerji, „Model Driven Architecture (MDA)“, http://www.omg.org/mda/mda_files/MDA_Briefing_Mukerji.pdf, Abrufdatum: 12.02.2008

[MOF] „OMG's MetaObject Facility“, http://www.omg.org/mof/, Abrufdatum: 24.01.2008

[NOE] Tobias Nöbel, „Vortrag zu: The Schematron Assertion Language“, http://www2.informatik.hu-berlin.de/~noebel/Schematron.htm, Abrufdatum: 24.10.2007

[OMG07] „OMG Model Driven Architecture“, Last updated on 16. November 2007, http://www.omg.org/mda/, Abrufdatum: 24.01.2008

[OMGp07] Fred Waskiewicz(Editor), „Catalog of UML Profile Specifications“, http://www.omg.org/technology/documents/profile_catalog.htm, 05.11.2007, Abrufdatum: 15.02.2008

[OMG03] Joaquin Miller, Jishnu Mukerji, „MDA Guide Version 1.0.1“, 12.06.2003, http://www.omg.org/cgi-bin/apps/doc?@omg/03-06-01.pdf, Abrufdatum: 24.10.2007

[PEME06] Roland Petrasch, Oliver Meimberg, „Model-Driven Architecture - Eine praxisorientierte Einführung in die MDA“, Dpunkt Verlag, Auflage: 1 (Juni 2006), ISBN 3898643433, http://www.mdabuch.de/

[SCHN04] Peter Schnell, Vortrag:„Java-Softwareentwicklung mit dem MDA-Ansatz“, 01. Juli 2004, http://dbis.informatik.uni-rostock.de/Forschung/Projekte/Kooperationen/Forschung/Kooperationen/idg-data/MDA-Vortrag_Uni_Rostock.pdf, Abrufdatum: 05.02.2008

[SPBG] Shane Sendall, Gilles Perrouin, Nicolas Guelfi, Olivier Biberstein, „Supporting Model-to-Model Transformations: The VMT Approach“, http://ic2.epfl.ch/publications/documents/IC_TECH_REPORT_200351.pdf, Abrufdatum: 02.02.2008

[ST06] Robert Stephan, „Entwicklung und Implementierung einer Methode zum konzeptuellen Entwurf von XML-Schemas“, 01. Februar 2006, http://dbis.informatik.uni-rostock.de/uploads/tx_userdbis/thesis_50_da-stephan.pdf, Abrufdatum: 02.02.2008

[STRON04] James Clark, Ken Holman, Martin Bryan, „ISO Schematron“, Final Committee Draft, Oktober 2004, http://www.schematron.com/iso/Schematron.html, Abrufdatum: 25.01.2008

[STRONe04] „Schematron's Six Basic Elements“, http://www.schematron.com/elements.html, Abrufdatum: 26.01.2008

[STRONq04] Rick Jelliffe, „The Schematron 1.5 Syntax(Quick Reference)“, http://xml.ascc.net/resource/schematron/quick1-5.html, Abrufdatum: 28.01.2008

[UML] „UML - Unified Modeling Language“, http://www.uml.org, Abrufdatum: 24.01.2008

[W3CDTD06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, Francois Yergeau, John Cowan, „Extensible Markup Language (XML) 1.1 (Second Edition)“, Kapitel 2.8 - 3.2, W3C Recommendation 16 August 2006, edited in place 29 September 2006, (http://www.w3.org/TR/xml11/)

[W3CXML06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, Francois Yergeau, John Cowan, „Extensible Markup Language (XML) 1.1 (Second Edition)“, W3C Recommendation 16 August 2006, edited in place 29 September 2006, (http://www.w3.org/TR/xml11/)

[WAKE03] Jos Warmer, Anneke Kleppe, „MDA Explained - The Model Driven Architecture“, Addison-Wesley Longman (28. Mai 2003), ISBN 032119442X, http://www.klasse.nl/books/mdaexplained.html

[XMLA03] Helmut Erlenkötter, „XML. Extensible Markup Language von Anfang an“, Rowohlt Tb., Auflage: 1 (September 2003), ISBN 3499612097

[XMLC02] Stefan Mintert (Hrsg.), „XML & Co - Die W3C-Spezifikationen für Dokumenten- und Datenarchitektur“, Addison-Wesley, München, ISBN 3827318440

[XMLD02] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, Übersetzer: Stefan Mintert, „Extensible Markup Language (XML) 1.0 (Zweite Auflage) Deutsche Übersetzung“, (http://www.edition-w3c.de/TR/2000/REC-xml-20001006/#Notations)

[XMLidP] Henning Behme, Stefan Mintert, „XML in der Praxis Extensible Markup Language für Profis“ Kapitel 1.3, http://www.linkwerk.com/pub/xmlidp/2000/buch.html, Abrufdatum: 16.11.2007

[XMLS04] „XML Schema 1.1“, http://www.w3.org/XML/Schema.html, Abrufdatum: 25.01.2008

[XMLSa04] David C. Fallside, Priscilla Walmsley, „XML Schema Part 0: Primer Second Edition“, W3C Recommendation 28 October 2004, http://www.w3.org/TR/xmlschema-0/

[XMLSb04] Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn, „XML Schema Part 1: Structures Second Edition“, W3C Recommendation 28 October 2004, http://www.w3.org/TR/xmlschema-1/

[XMLSc04] Paul V. Biron, Kaiser Permanente, Ashok MalhotraThis, „XML Schema Part 2: Datatypes Second Edition“, W3C Recommendation 28 October 2004, http://www.w3.org/TR/xmlschema-2/

[ZEN07] Barbara Zengler, http://www.barbara-zengler.de/vorlesung/dtd.html, Abrufdatum: 05.12.2007

[ZEN07a] Barbara Zengler, http://www.barbara-zengler.de/pics/valid-documents.png, Abrufdatum: 05.12.2007

[ZEN07b] Barbara Zengler, http://www.barbara-zengler.de/pics/schema-valid-documents.png, Abrufdatum: 05.12.2007

Thesen

- Die Schemasprachen XML-DTD und XML-Schema sind vorrangig auf Strukturbeschreibung ausgerichtete Schemasprachen. Sie stellen kontextfreie Grammatiken dar, mit denen sich XML-Dokumente definieren lassen.
- Schematron ist dagegen eine eher auf Inhaltsprüfung ausgerichtete Schemasprache, die XML-Dokumente nicht komplett definiert, sondern „nur“ auf bestimmte Muster überprüft.
- Mit Schematron lassen sich Regeln definieren, die mit XML-DTD und XML-Schema nicht darstellbar sind.
- Die MDA ist ein sehr geeignetes Konzept zur Softwareentwicklung mit Hilfe von Modellen.
- Die Erzeugung von Quellcode aus Modellen erfolgt in der MDA zumeist nicht vollständig automatisch. Es werden nur sinnvolle Anteile, wie insbesondere Strukturbeschreibungen, automatisch erzeugt. Die automatisch erzeugten Anteile können manuell bearbeitet, d.h. geändert, und manuell ergänzt werden.
- Das MDA-Konzept ist auf die Modellierung von XML-Schemata übertragbar.
- Es lassen sich vor allem Strukturbeschreibungen von XML-Dokumenten sehr gut und übersichtlich mittels Modellen beschreiben.
- Das vereinfachte CoDEX-Modell ist eine geeignete Grundlage für ein, bezüglich einer konkreten Schemasprache, plattformunabhängiges Modell.
- Das CoDEX-Modell und seine Modifikationen für die verschiedenen Schemasprachen DTD, XML-Schema und Schematron sind einfache, kompakte und leicht verständliche Modelle. Mit ihnen lassen sich die wesentlichen Aspekte der jeweiligen Schemasprachen darstellen. Sie sind daher als plattformspezifische Modelle für die verschiedenen Schemasprachen geeignet.
- Das eigene Modell ist als plattformunabhängiges Modell für XML-Schemasprachen geeignet.
- Die verschiedenen Variationen des eigenen Modelles für die unterschiedlichen XML-Schemasprachen sind eine einfache, leicht erweiterbare und übersichtliche Darstellungsmöglichkeit für XML-Schemata.
- Mit den Variationen des eigenen Modelles lassen sich die wesentlichen Beschreibungsmöglichkeiten der Schemasprachen DTD, XML-Schema und Schematron darstellen. Sie eignen sich somit als plattformabhängige Modelle.
- Die jeweiligen plattformunabhängigen Modelle für das CoDEX- und das eigene Modell lassen sich automatisch in die entsprechenden plattformabhängigen Modelle und diese in entsprechenden Quellcode überführen.
- Bei der Transformation vom plattformunabhängigem CoDEX-Modell ins DTD-abhängige CoDEX-Modell kommt es zu Informationsverlusten bezüglich der Schlüsseleigenschaften und Auftretenshäufigkeiten von Modellelementen.
- Bei der Transformation vom eigenem plattformunabhängigem Modell ins DTD-abhängige Modell kommt es zu Informationsverlusten bezüglich der Datentypen von Elementen.
- Der Quelltext eines XML-Schemas ist, bis zu einem gewissen Anteil, sinnvoll aus den Modellen für die XML-Schemasprachen automatisch generierbar. Die Erweiterung und Änderbarkeit der automatisch generierten Modelle, auf den verschiedenen Modellierungsebenen, bzw. des Codes ist durch die Konzepte der MDA möglich.

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, den 29. April 2008