Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik
Lehrstuhl Datenbank- und Informationssysteme

Diplomarbeit

Entwicklung und Implementierung einer Methode zum konzeptuellen Entwurf von XML-Schemas

Robert Stephan
geb. am 19. Mai 1979
in Prenzlau

Abgabetermin: 01. Februar 2006

Hochschullehrer:
Prof. Andreas Heuer
Prof. Peter Forbrig

Betreuer:
Dr. Meike Klettke

Zusammenfassung

Der Entwurf und die Bearbeitung von XML-Schemas kann durch die Verwendung eines konzeptuellen Modells wesentlich vereinfacht werden. Im Rahmen dieser Diplomarbeit wurde daher ein konzeptuelles Modell für die wesentlichen Aspekte des W3C XML-Schema-Standards entwickelt und implementiert. Zusätzliche wurden Algorithmen beschrieben, die aus dem Modell ein XML-Schema ableiten (Export) und aus einem gegebenen XML-Schema das Modell generieren (Import).

Weiterhin wurde ein graphischer Editor implementiert, der den Anwender beim Erstellen und Editieren des Modells unterstützt. Auch die Implementierungen der Im- und Exportmethoden sind Bestandteil des Editors.

Abstract

Designing and editing XML Schemas can be simplified by the use of a conceptual model. Therefore a conceptual model, that can handle the fundamental parts of the W3C XML Schema standard has been created and implemented. In addition algorithms have been developed, that describe the derivation of an XML schema from the model (export) and the generation of a model out of an existing XML Schema (import).

Furthermore a graphical editor has been developed, that supports users creating and editing the model. XML schema import and export functionalities have been integrated as well.

CR-Klassifikation

- I.6.5 [SIMULATION AND MODELING]:
  Model Development
- I.7.2 [DOCUMENTS AND TEXTPROCESSING]:
  Document preparation - Markup languages, XML

Key Words

XML, XML Schema, conceptual model, model editor

Abkürzungsverzeichnis

CVS	Concurrent Versions System (Programm zur Versionsverwaltung von Dateien)
DSD	Document Structure Definition (Schemasprache für XML)
DT4DTD	DataTypes for DTDs (Schemasprache für XML)
DTD	Document Structure Declaration (Schemasprache für XML)
EER	Extend Entity-Relationship Model (erweitertes ER-Modell)
EMX	Entity Model for XML-Schema (konzeptuelles Modell dieser Diplomarbeit)
ENF	Element Normal Form (Spezialform eines XML-Dokumentes)
ER	Entity Relationship Model (konzeptuelles Modell für Datenbankentwurf)
EReX	Entity Relationship Model extended for XML (konzeptuelles Modell für XML)
IDE	Integrated Development Environment (Anwendungsprogramm zur Entwicklung von Software)
ISO	International Standards Organization (Standardisierungsgremium)
OASIS	Organization for the Advancement of Structured Information Standards (Standardisierungsgremium)
OMG	Object Management Group (Standardisierungsgremium)
RELAX NG	Regular Language description for XML Next Generation (Schemasprache für XML)
RHG	Regular Hedge Grammar (Grammatik, basierend auf Hedges)
RSL	Regular Schema Language (Schemasprache für XML)
SGML	Standard Generalized Markup Language (Dokumentbeschreibungssprache)
SOX	Schema for Object-Oriented XML (Schemasprache für XML)
UML	Unified Modeling Language (Modellierungssprache aus der Softwaretechnik)
W3C	World Wide Web Consortium (Standardisierungsgremium)

X-Entity	XML-Entity (konzeptuelles Modell für XML)
XER	eXtensible Entity Relationship Model (konzeptuelles Modell für XML)
XMI	XML Metadata Interchange (XML-Sprache zum Datenaustausch u.a. im UML-Umfeld)
XML	eXtensible Markup Language (Daten- und Dokumentformat)
XSD	XML Schema Definition (Schemasprache für XML)
XSLT	XML Stylesheet Language Transformation (XML-Sprache zur Umwandlung von XML-Dokumenten)

Inhaltsverzeichnis

Kapitel 1

Einführung

Ziel dieser Diplomarbeit ist es, ein konzeptuelles Modell für XML-Schema zu entwickeln. Dafür werden zunächst Anforderungen untersucht und im Anschluss daran das Modell sowie seine formale und graphische Repräsentation beschrieben. Es werden Algorithmen entwickelt, die beschreiben, wie ein XML-Schema und ein konzeptuelles Modell ineinander umgewandelt werden können (XML-Schema-Import und -Export).

Das im Rahmen dieser Diplomarbeit implementierte Programm ist ein graphischer Editor für das beschriebene Modell, der auch den XML-Schema Import und Export unterstützt.

Einleitend wird zunächst der Modellbegriff erklärt. Anschließend wird kurz die allgemeine Bedeutung von XML und XML-Schema beschrieben und danach erläutert, warum konzeptuelle Modellierung für XML-Schema überhaupt notwendig ist. Abgeschlossen wird dieses Kapitel mit einem kurzen Überblick über die Struktur dieser Diplomarbeit.

1.1 Was ist ein Modell?

Das Wort *Modell* (von ital. *modello*) entstand im Zeitalter der Renaissance. Aber schon seit der Antike denkt man in „Modellen“, auch wenn der Begriff noch nicht explizit verwendet wurde.

Nach der klassischen Definition ist ein *Modell* ein vereinfachtes Abbild eines vorhandenen oder Vorbild für ein zu schaffendes Gebilde. Das Abbild oder Vorbild wird *Original* genannt.

Modelle benötigt man zum Verstehen eines Gebildes, als Diskussionsgrundlage für ein Gebilde, als gedankliches Hilfsmittel zum Gestalten und Bewerten eines geplanten Gebildes, zur Spezifikation von Anforderungen an ein Gebilde, zum Aufstellen/Prüfen von Hypothesen über das Gebilde oder zur Durchführung von Experimenten, die am Gebilde nicht durchgeführt werden können, sollen oder dürfen.

Modelle treten in verschiedensten Wissenschaften auf, zum Beispiel:

Kunst. In der Kunst dient ein Modell als Vorbild für den Bildhauer oder Maler.

Architektur. Das 3D-Modell eines Gebäudes hilft dem Architekt und seinem Auftraggeber bei der Diskussion über das Projekt. Aspekte wie die äußere Gestaltung, Integration ins Stadtbild oder Raumaufteilung lassen sich am Modell schon vor Baubeginn erörtert. Auch Bauzeichnungen und statische Berechnungen sind Modelle, die dazu dienen die Sicherheit zu prüfen oder den Ablauf des Baus zu planen.

Ingenieurwissenschaften. In diesem Bereich versteht man unter Modellen die Nachbildung eines technischen Erzeugnisses im verkleinerten Maßstab. An Versuchsmodellen werden Messungen vorgenommen und die Ergebnisse auf das Originalobjekt umgerechnet. Zum Einsatz kommen Modelle zum Beispiel im Wind- oder Strömungskanal.

Informatik. In der Informatik dienen Modelle als Hilfsmittel für die Konstruktion und die Einsatzplanung von Informatiksystemen. Sie können Abbilder der Vorstellungen des Auftraggebers oder Vorbilder für zu konstruierende Informatiksysteme sein. Informatikmodelle sind meist *abstrakt*. - „Software kann man nicht anfassen.“ [KKB05].

Herbert Stachowiak hat 1973 eine *Allgemeine Modelltheorie* [Sta73] verfasst. Der darin entwickelte Modellbegriff ist interdisziplinär, also allgemein, anwendbar. Nach Stachowiak ist ein Modell durch die folgenden drei Merkmale charakterisiert:

Abbildung

- Jedes Modell ist ein Abbild oder Vorbild eines vorhandenen oder zu erschaffenden Originals.
- Zu jedem Modell gehört eine Abbildungsvorschrift, welche die Bestandteile und Eigenschaften des Originals auf diejenigen des Modells abbildet.
- Das Original kann selbst wieder ein Modell sein.
- Für ein Original kann es verschiedene Modelle geben.

Verkürzung

- Jedes Modell *abstrahiert*, das heißt, es werden nicht alle Bestandteile und Attribute des Originals dargestellt.
- Es wird nur modelliert, was aus Sicht des Modellierenden wichtig / wesentlich / nützlich erscheint.

- Das Modell kann auch Bestandteile und Attribute besitzen, die keine Entsprechung im Original haben.

Pragmatismus

- Jedes Modell wird für einen spezifischen Zeitraum und Verwendungszweck erstellt und bearbeitet.
- Das Modell wird interpretiert.
- Es dürfen nur Modelleigenschaften ausgewertet werden, die eine Entsprechung im Original haben.

1.2 Bedeutung von XML und XML-Schema

XML, die *eXtensible Markup Language* (erweiterbare Auszeichnungssprache), ist eine Metasprache zur Definition weiterer Auszeichungssprachen. Mittels XML lassen sich Sprachen zur Beschreibung der Struktur von Textdokumenten, Grafiken, mathematischen Formeln, Datenbanken oder anderen Arten von strukturierten Daten definieren.

Entwickelt und standardisiert wurde XML beim World-Wide-Web-Consortium (W3C). Es basiert auf der Metasprache SGML (Standard Generalized Markup Language), welche ihren Ursprung schon Ende der 60er-Jahre des vorigen Jahrhunderts hatte.

Der Vorteil von XML ist, dass die XML-Spezifikation [XML00] nur wenige formale Regeln für den Aufbau von XML-Dokumenten enthält. Dokumente, die diesen Regeln genügen, nennt man *wohlgeformt*.

Darüber hinaus besteht die Möglichkeit, XML-Dokumentenformate zu definieren und Dokumente von diesen Definitionen abhängig zu machen. Dokumenten, die einer bestimmten Definition genügen, nennt man *gültig*.

Die klassische Definitionssprache ist die Document Type Declaration (DTD). Sie ist bereits aus SGML bekannt und ebenfalls Bestandteil des XML-Standards. Sie wurde jedoch vereinfacht und in ihrem Umfang starkt eingeschränkt. Als leistungsfähigere Alternative führte das W3C im Jahre 2001 XML-Schema ein. DTDs, XML-Schema und RELAX NG (eine weitere Schemabeschreibungssprache) werden in Kapitel 2 noch detailliert vorgestellt.

1.3 Vorteile der konzeptuellen Modellierung für XML-Schemas

Der XML-Schema-Standard ist relativ komplex. Deshalb wird ein konzeptuelles Modell benötigt, welches auf einfache, graphische Art und Weise, die Zusammenhänge des Schemas darstellen kann.

Der XML-Schema-Laie kann mit einem speziellen Editor das konzeptuelle Modell erzeugen und bearbeiten. Dabei kann er von dem Programm geführt werden, indem er beispielsweise auf Fehler hingewiesen wird. Aus dem erstellten Modell kann anschließend ein XML-Schema in korrekter Syntax generiert werden.

Aber auch für den XML-Schema-Experten bietet die Arbeit an einem Modell Vorteile. Der Quelltext eines XML-Schemas kann sehr umfangreich sein und sich sogar über mehrere Dateien erstrecken. Die Struktur des Schemas lässt sich aus der graphischen Notation leichter erschließen, da zum Beispiel Beziehungen zwischen den Komponenten dargestellt werden oder bestimmte Aspekte temporär ausgeblendet werden können, um die Übersichtlichkeit zu erhöhen.

Der Editor muss deshalb in der Lage sein, das Modell aus einer bestehenden XML-Schema-Datei zu generieren.

1.4 Struktur der Arbeit

In Kapitel 2 werden zunächst die wichtigsten Schemabeschreibungssprachen vorgestellt. Kapitel 3 verschafft einen Überblick über allgemeine Anforderungen, die an Modelle für XML gestellt werden können. Kapitel 4 stellt dann unterschiedliche existierende Ansätze für die Modellierung von XML-Daten vor. Die verschiedenen Modelle lassen sich in physische, formale und konzeptuelle Modelle kategorisieren. In Kapitel 5 wird das im Rahmen dieser Diplomarbeit entwickelte konzeptuelle Modell vorgestellt. Im nächsten Kapitel werden die Operationen, die auf dem Modell ausgeführt werden können, beschrieben. In Kapitel 7 werden die Tools und Bibliotheken, die zur Erstellung des Modell-Editors verwendet wurden, vorgestellt.

Kapitel 2

Schemasprachen

Schemasprachen wurden entwickelt, um den Inhalt und die Struktur von Dokumenten beschreiben und validieren zu können. DTDs (Document Type Declarations) entstanden zusammen mit SGML (Standard Generalized Markup Language) [Int86] zur Beschreibung des Aufbaus von SGML-Dokumenten. Sie waren fester Bestandteil in jedem SGML-Dokument, da es noch kein Konzept für wohlgeformtes (schema-loses) SGML gab. DTDs wurden in vereinfachter Form in den XML-Standard [XML00] übernommen. Sie sind heute ein weit verbreitetes Mittel zur Beschreibung der Struktur von XML-Daten. Neben DTDs haben sich weitere Schemasprachen für XML entwickelt. Sie werden in diesem Abschnitt vorgestellt und miteinander verglichen.

2.1 DTDs

Eine vollständige Übersicht sowie die konkrete Syntax der einzelnen Konstrukte kann im XML-Standard [XML00] nachgelesen werden. Die Angabe der DTD kann innerhalb eines XML-Dokumentes oder durch eine Referenz auf eine externe Datei erfolgen.

Die wichtigsten Bestandteile einer DTD sind:

Elementdeklaration. Elemente können keinen Inhalt, nur Textdaten, Subelemente oder gemischten Inhalt (Subelemente und Text) enthalten. Die Auftretenshäufigkeit der Subelemente kann durch die Operatoren `?` (null- oder einmalig), `+` (ein- oder mehrmalig) und `*` (null- oder mehrmalig) definiert werden. Subelemente können als Sequenz (`,`) oder Alternative (`|`) auftreten.

Attributdeklaration. Jedem Element kann eine Liste von Attributen zugeordnet werden. Für die Attribute gibt es eine vordefinierte Menge von Typen (`CDATA` [Textinhalt], `..|..|..` [Aufzählung vordefinierter Werte], `ID` [eindeutiger Identifikator], `IDREF, IDREFS` [Referenz oder Referenzliste auf IDs anderer Elemente], `NMTOKEN` [gültiger XML-Name], `NMTOKENS` [Liste gültiger XML-Namen], `ENTITY` [Entität], `ENTITIES` [Liste von Entitäten] und `NOTATION` [Notation = Definition des Formates externer Nicht-XML-Dateien]).
Jedes Attribut kann durch Angabe eines der folgenden Konstrukte noch genauer definiert werden: `value` [der Defaultwert des Attributes], `#REQUIRED` [Das Attribut muss im Element angegeben werden.], `#IMPLIED` [Das Attribut ist optional.] und `#FIXED value` [Das Attribut hat einen unveränderlichen Wert.]

Die größten Nachteile der DTDs sind der geringe Umfang integrierter Datentypen, fehlende Unterstützung für Namespaces sowie eingeschränkte Ausdrucksmöglichkeiten bei der Definition eines komplexen Inhaltsmodells.

2.2 XML-Schema

Auf Grund der offensichtlichen Schwächen von DTDs wurden viele Erweiterungen vorgeschlagen, angefangen von Unterstützung für Namensräume oder Datentypen (DT4DTD [BGP00]) bis hin zu komplexen objektorientierten Modellen (SOX [DFH99]).

2001 verabschiedete das W3C den XML-Schema-Standard (XSD) [XS001] [XS101] [XS201]. Er bildet die Grundlage für viele weitere Standards, die das W3C seitdem verabschiedet hat und wird deshalb auch in der nächsten Zeit noch eine wichtige Rolle in der XML-Welt spielen.

Ein Vorteil von XML-Schema ist, das es eine XML-Syntax besitzt. Der XSD-Standard ist aber ziemlich komplex und daher ist es für den Einsteiger schwierig XML-Schemas zu implementieren und zu verstehen. Eine Studie von BEX ET ALL [BMNS05] zeigt, dass eine Vielzahl von den im Internet auffindbaren XML-Schemas falsch sind und eine Vielzahl der korrekten XML-Schemas sich auf die Verwendung von Konstrukten beschränken, die auch von DTDs zu Verfügung gestellt wurden. Daher könnte man vermuten, dass es sich um XML-Schemas handelt, welche durch Tools automatisch aus DTDs abgeleitet wurden.

An dieser Stelle soll kurz in die wichtigsten Bestandteile von XML-Schemas eingeführt werden. Details entnehme man dem Standard [XS001] [XS101] [XS201].

Elementdefinition. Mit der Elementdefinition werden Elemente eingeführt, die in den XML-Instanzdokumenten auftreten können. Den Elementen wird ein Name und eine Typdefinition zugeordnet. Die Typdefinition kann anonym sein, das heißt, sie hat keinen eigenen Namen und wird in die Elementdefinition integriert.

Definition einfacher Typen. In den XML-Schema-Standard [XS101] wurden bereits eine Reihe einfacher Datentypen integriert. Abbildung 2.2 zeigt sie in einer Übersicht. Diese integrierten Typen bilden die Grundlage für die Definition weiterer einfacher Datentypen, in dem beispielsweise der Wertebereich durch Fassetten (*engl.* facets) beschränkt wird oder Listen und Vereinigungen von einfachen Datentypen definiert werden.

Definition komplexer Typen. Komplexe Typen beschreiben Inhaltsmodelle, die aus Subelementen und Attributen bestehen. Sie werden durch Gruppierung von Subelementen aufgebaut. Gruppen können auch geschachtelt werden.

Als mögliche Gruppenoperatoren stehen zur Verfügung:

Sequence. Die Subelemente der Gruppe treten in der angegebenen Reihenfolge auf.

Choice. Eines der Subelemente aus der Gruppe kann eingesetzt werden.

All. Jedes der angegebenen Subelemente kann maximal einmal auftreten. Die Reihenfolge ist beliebig.

Komplexe Typen können aber auch aus einfachen Datentypen abgeleitet werden, indem zum Beispiel ein einfacher Datentyp durch Attribute ergänzt wird.

Durch das Attribut `mixed` in der Typdefinition wird angezeigt, dass der Inhalt neben den Subelementen noch weiteren Text enthalten kann.

Häufigkeitsbeschränkungen. Die Auftretenshäufigkeit von Elementen innerhalb einer Gruppe oder des XML-Dokumentes kann durch die Attribute `minOccurs`, `maxOccurs`, `fixed` und `default` beschränkt werden. In der Attributdeklaration können der Parameter `use` mit dem Wertebereich {`required, optional, prohibited`} sowie die Parameter `fixed` und `default` zur Häufigkeitsbeschränkung und zur Angabe von Default-Werten verwendet werden.

Namensräume. Durch Namensräume (*engl.* namespaces) lassen sich Typ- und Elementdefinitionen kategorisieren.

Verteilung auf mehrere Instanzen. XML-Schema bietet Operatoren, durch die ein Schema aus mehreren Schemadateien zusammengesetzt werden kann.

Typableitung. Zur Definition neuer Typen ist es möglich, bestehende Typdefinitionen wiederzuverwenden und zu erweitern bzw. einzuschränken. Die Definition abstrakter Typen ist möglich. Das bedeutet, dass im XML-Dokument an dieser Stelle ein Element mit einem der abgeleiteten Typen eingesetzt werden muss.

Eindeutigkeit, Schlüssel-Fremdschlüssel-Beziehung. In XML-Schema stehen Konstrukte zur Verfügung, die zusichern, dass sich alle Elemente in einer XML-Instanz bezüglich einer Bedingung (zum Beispiel dem Wert eines Attributes) unterscheiden. Dadurch sind die Elemente eindeutig identifizierbar und es lassen sich Beziehungen zwischen ihnen festlegen.

2.3 RELAX NG

Mit dem Ziel eine kleinere und „saubere“ Sprache als XML-Schema zu entwickeln entstand RELAX NG [REL01]. RELAX NG wird, im Gegensatz zu vielen anderen Schemasprachen, die nur als Prototypen in Forschung und Lehre existieren, auch in der „realen“ Welt verwendet. Der RELAX NG-Standard wird von der OASIS betreut und ist von der ISO als Standard (ISO/IEC 19757-2) akzeptiert. Die Autoren des Standards [REL01] charakterisieren RELAX NG mit folgenden Eigenschaften:

- einfach
- leicht zu lernen
- hat eine XML-Syntax und eine kompaktere Nicht-XML-Syntax
- unterstützt Namespaces
- behandelt soweit möglich Attribute und Elemente gleich
- uneingeschränkte Unterstützung für nicht-geordneten Inhalt (*engl.* unordered content)
- uneingeschränkte Unterstützung für gemischten Inhalt (*engl.* mixed content)
- hat eine solide theoretische Basis
- kann Datentyp-Definitionen beliebiger Datentyp-Sprachen (z.B. die XML-Schema-Datentypen) integrieren.

RELAX NG ist genauso wie DTDs und XML-Schema grammatikbasiert. Ein Schema ist also eine Menge von Regeln, aus denen gültige Instanzen abgeleitet werden können.

2.4 Bewertung der verschiedenen Sprachen

	SGML DTD	XML DTD	XSD	RELAX NG
attribute order				
simple datatypes	(✓)	(✓)	✓	(✓)
all/interleave	✓		(✓)	✓
exceptions (inclusions/exclusions)	✓			
non-deterministic content models				✓
local element declarations			✓	✓
element/attribute choices				✓
generalized mixed content	✓			✓
ANY/wildcards	(✓)	(✓)	✓	✓

Abbildung 2.1: Eigenschaften von Schemasprachen aus [Wil05]

Einfachheit und Ausdrucksstärke einer Sprache sind oft gegensätzliche Ansprüche. DTDs sind einfach zu lesen, aber relativ ausdrucksschwach. XML-Schemas benötigen wegen der umfangreichen Spezifikation eine längere Einarbeitungsszeit. Wilde [Wil05] sieht in RELAX NG einen Versuch, die Balance dazwischen zu finden, welcher aber nur von einer kleinen Gruppe von Anwendern genutzt wird.

Abbildung 2.1 stellt grundlegende Eigenschaften für die Modellierung von Dokumentenklassen mit den verschiedenen Schemasprachen gegenüber.

Obwohl SGML-DTD keine XML-Schemasprache darstellt, wurde sie mit aufgenommen, um Unterschiede zu XML-DTDs zeigen zu können.

Bislang gibt es in keiner Schemasprache Konstrukte, mit welchen die Reihenfolge von Attribute beschrieben werden kann. Deshalb wird allgemein akzeptiert, dass unter Attributen keine Ordnung definierbar ist.

In vielen Anwendungsfällen werden einfache Datentypen, wie `integer` oder `date` benötigt. SGML-DTDs und DTDs lassen jedoch nur eine sehr begrenzte Anzahl von Datentypen auf Attributen zu. In XML-Schema wurde eine umfangreiche Typbibliothek integriert. RELAX NG hat keine eigene Typbibliothek, kann aber auf andere Typbibliotheken (z.B. von XML-Schema) zurückgreifen.

Das `all`-Inhaltsmodell aus SGML-DTDs wurde in XML-DTDs zu Gunsten der einfacheren Implementierung, nicht übernommen. In XML-Schema wurde es mit Einschränkungen (z.B. darf nicht mit anderen Modellgruppen zusammenstehen) wieder eingeführt. RELAX NG lässt es ohne Einschränkungen zu.

SGML-DTDs und XML-DTDs gestatten keine nichtdeterministischen Inhaltsmodelle. XML-Schema gestattet diese aus Gründen der Abwärtskompatibilität ebenfalls nicht. Nur mit RELAX NG lässt sich nichtdeterministischer Inhalt darstellen.

SGML-DTDs und XML-DTD erlauben nur globale Elementtypdeklarationen. Somit gibt es für jedes Element genau eine Typdefinition. In XML-Schema ist es möglich, den Elementtyp lokal zu definieren. Das bedeutet, dass zwei Elemente, mit der Beschränkung, dass sie in unterschiedlichem Kontext auftreten, unterschiedliche Typen haben können. In RELAX NG fällt auch diese Restriktion weg.

SGML-DTDs, XML-DTDs und XML-Schema behandeln Elemente und Attribute verschieden. Das bedeutet, dass zwischen Element- und Attributinhalten keinerlei Abhängigkeiten definiert werden können. In der Praxis können solche Abhängigkeiten durchaus auftreten. Mit RELAX NG kann die Wahlmöglichkeit, ob Inhalte als Attribut oder Element dargestellt werden, auch ausgedrückt werden.

SGML-DTDs enthalten umfangreiche Möglichkeiten, gemischten Inhalt zu spezifizieren. Diese Möglichkeiten wurden in XML-DTDs und XML-Schema stark eingeschränkt. In XML-Schema wird gemischter Inhalt innerhalb eines Element durch eine boolesche Variable erlaubt oder verboten. RELAX NG setzt das Konzept wie SGML um.

SGML-DTDs und XML-Schemas enthalten das `ANY`-Inhaltsmodell, welches jede Art von Inhalt unterhalb eines Elementes gestattet. Dieses Konzept ist aber oft zu allgemein und lässt sich nur unzureichend kontrollieren. Mit der Einführung von Namespaces in XML-Schema und RELAX NG ist es möglich zuzusichern, dass die Kindelemente nur aus einem bestimmten Namespace stammen.

Diese kurze Übersicht zeigt, dass XML-DTDs und XML-Schema einige Funktionen vermissen lassen, die in bestimmten Anwendungsfällen durchaus sinnvoll erscheinen. In RELAX NG gibt es viele dieser Einschränkungen nicht. Deshalb ist es verwunderlich, dass RELAX NG einen so geringen Verbreitungsgrad gegenüber XML-Schema und DTDs hat.

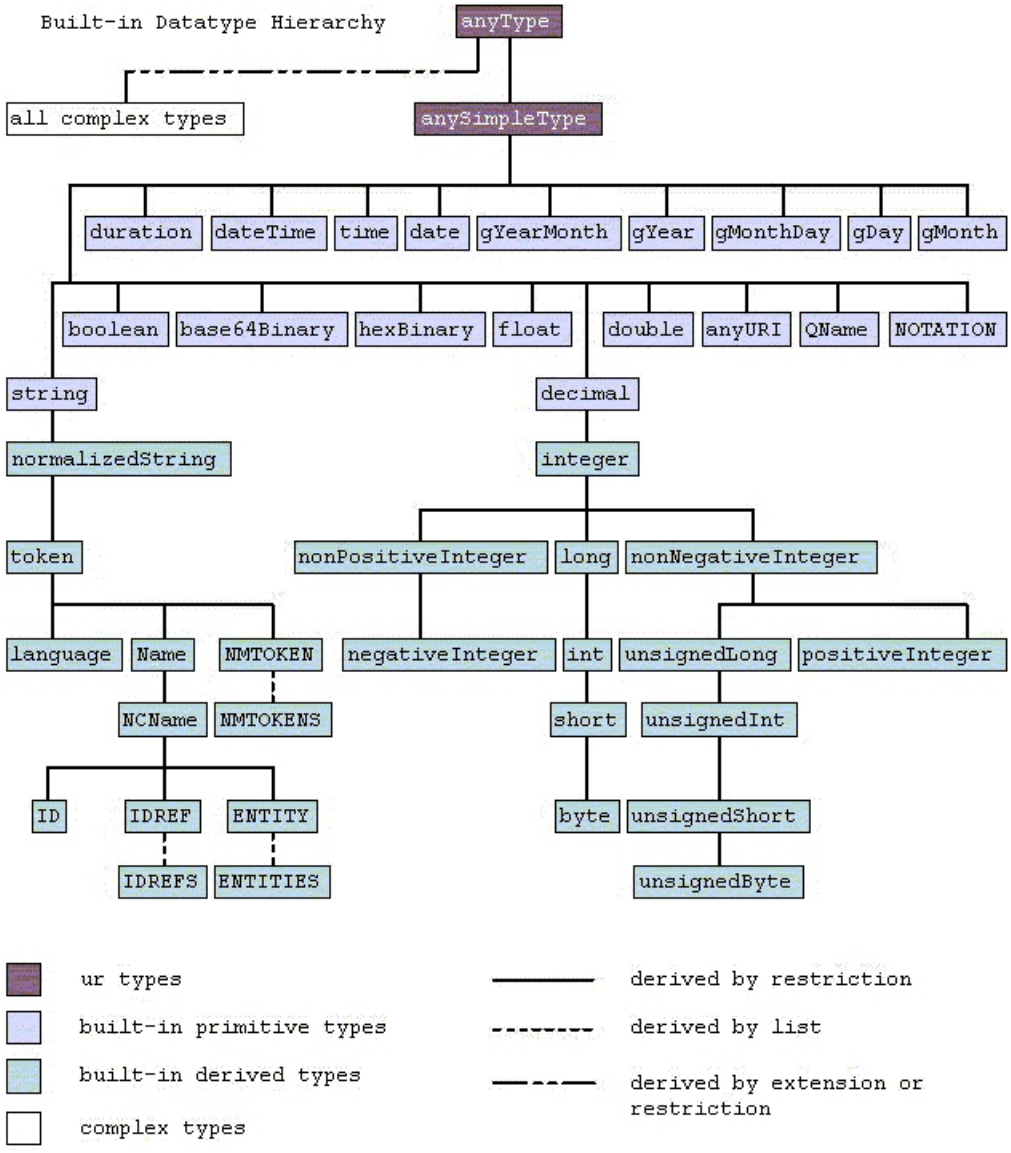


Abbildung 2.2: „Built-in Datatypes“ aus [XS201]

Kapitel 3

Anforderungen an Modelle

3.1 Gütekriterien für konzeptuelle Modelle

Wilde stellt in [Wil05] eine Liste von allgemeinen Anforderungen an konzeptuelle Modelle vor. Wenn das Modell für eine bestimmte Schema-Sprache entwickelt werden soll, lassen sich nicht alle Punkte vollständig abbilden, da die Zielsprache in der Regel Einschränkungen vorgibt. Außerdem können abhängig vom Anwendungsfall einzelne Punkte von größerer oder geringerer Bedeutung sein.

Die einzelnen Kriterien werden an dieser Stelle vorgestellt. Der Beschreibung folgt jeweils eine kurze Anmerkung (in serifenloser Schrift) darüber, wie das Konzept im EMX-Modell (dem im Rahmen dieser Diplomarbeit entwickelten konzeptuellen Modell) umgesetzt wird oder welche Einschränkungen es gibt.

Formale Grundlage

Das Modell braucht eine solide formale Grundlage, die es ermöglicht Eigenschaften des Modells herzuleiten und zu beweisen. Basierend auf der formalen Grundlage, ist es möglich, Modelle zu vergleichen und zu testen, ob ein Modell Teil eines anderen Modells ist. Spezielle Eigenschaften (z.B. Nichtdeterminismus) können überprüft werden. Bestimmte Operationen (z.B. Erkennung von Redundanzen) lassen sich ausführen.

Auf eine formale Grundlage kann nicht verzichtet werden, auch das EMX-Modell braucht eine solide Basis.

Graphische Notation

Eines der wichtigsten Ziele der konzeptuellen Modellierung ist das Erzeugen eines Modells, welches einfach zu lesen und frei von konkreten Implementationsdetails ist. Verschiedene Anwender sollen gemeinsam über das Modell kommunizieren können. Dazu ist eine intuitive graphische Notation notwendig, die auch von Nicht-XML-Experten einfach

zu verstehen sein muss. In der graphischen Notation sollte es möglich sein, bestimmte Teile ein- und auszublenden.

Bei der Entwicklung des EMX-Modells wurde dieser Aspekt besonders berücksichtigt.

Hierarchische und Referenzielle Strukturen

In XML gibt es zwei verschiedene Arten von Beziehungen - die *hierarchische Beziehung*, welche sich aus der Baumstruktur ergibt sowie *Referenzen* (Schlüssel-Fremdschlüssel-Konzept), welche häufig durch Attribute implementiert werden. Ein konzeptuelles Modell sollte beide Arten darstellen können.

Beide Konzepte haben ihre Berechtigung. In der prototypischen Umsetzung werden jedoch nur hierarchische Beziehungen umgesetzt.

Mapping auf Schema-Sprachen

Die Modellierung sollte unabhängig von einer bestimmten Schema-Sprache erfolgen. Danach muss das Modell in eine Schema-Sprache überführt werden. Die bekanntesten Schemasprachen (DTD, XSD, RELAX NG) sollten unterstützt werden. Da der Ableitungsprozess nicht vollständig automatisch erfolgen kann, sollte dies ein geführter Prozess sein.

Die Ableitung eines XML-Schemas aus dem EMX-Modell erfolgt so wie hier ausgeführt. Weitere Schema-Sprachen werden nicht unterstützt. Das Modell und der Modell-Editor könnten erweitert werden, um auch andere Zielsprachen zu unterstützen.

Nichtdeterministischer Inhalt

Nichtdeterministischer Inhalt ist ein mächtiges Konzept. Es kann dabei allerdings zu Mehrdeutigkeiten kommen. Ein konzeptuelles Modell sollte nichtdeterministischen Inhalt unterstützen, allerdings sollte der Nutzer auf mögliche Probleme hingewiesen werden.

Da das EMX-Modell mit dem Ziel erstellt wird, daraus eine XSD abzuleiten, wird Mehrdeutigkeit nicht unterstützt.

Konsistente Behandlung von XML-Knoten

Elemente und Attribute sollten durch die gleichen Modellkonstrukte dargestellt werden. Attribute haben einige zusätzliche Einschränkungen (keine Ordnung, keine Wiederholung, kein komplexer Inhalt); aber abgesehen davon sollte es keine unterschiedliche Behandlung zu Elementen geben.

Dieser Punkt wurde beim Erarbeiten des EMX-Modells untersucht. Er erscheint durchaus sinnvoll. Vor allem zu Gunsten der Übersichtlichkeit wurde jedoch entschieden, Attribute anders als Elemente darzustellen. Das

EMX-Modell könnte aber einfach erweitert werden, sodass es Elemente und Attribute auf gleiche Weise anzeigt.

Modellgruppen

Durch Modellgruppen können verschiedene Knoten zu komplexen Inhaltsmodellen kombiniert werden. Die aus XSD bekannten Modellgruppen (`sequence, choice, all`) stellen eine gute und allgemein bekannte Grundlage dar.

Die Definition komplexer Inhaltsmodelle durch Modellgruppen wird durch das EMX-Modell vollständig unterstützt.

Wiederverwendung

Elemente und Attribute sollten auf allen Ebenen wiederverwendet werden können. Die Unterstützung von *Vererbung* (Übernahme von Eigenschaften und Hinzufügen neuer Eigenschaften) wäre wünschenswert.

Wiederverwendung lässt sich im EMX-Modell über die Typdefinitionen realisieren. Vererbung ist ein mächtiges Konzept. Es sollte im Falle einer zukünftigen Erweiterung des EMX-Modells unbedingt umgesetzt werden.

Verallgemeinerter gemischter Inhalt

Textknoten sollten genauso wie Element- und Attributknoten behandelt werden. Dies gestattet größere Modellierungsfreiheit als die einfache Umsetzung von *Mixed Content* in DTDs und XML-Schema.

Im EMX-Modell wird das Konzept nur in der Form eingesetzt, wie es im XML-Schema-Standard definiert ist.

Offener Inhalt

Während die Modellierung so genau wie möglich erfolgen sollte, sollte sie auch so flexibel wie nötig sein. In bestimmten Anwendungen ist es notwendig, offenen Inhalt zu modellieren. Der offene Inhalt sollte sich z.B. durch Namespaces beschränken lassen - sodass nur Inhalt von einem spezifizierten Namespace erlaubt ist.

Die Möglichkeit, nicht konkret definierte Inhalte zuzulassen, wäre ein nettes Feature und sollte in einer zukünftigen Version des EMX-Modells unterstützt werden. XML-Schema stellt dafür das `any`-Konstrukt bereit.

Inter- und Intra-Dokument Beziehungen

Beziehungen durch Referenzen sind unbedingt notwendig, um Beschränkungen innerhalb des Datenmodells zu definieren. Allerdings sollte das Konzept nicht auf ein Dokument beschränkt sein; auch Beziehungen zwischen Elemente verschiedener Dokumente (sogar unterschiedlicher Dokumentklassen) sollten modellierbar sein.

Auch dieses Konzept sollte, soweit es sich in der XSD umsetzen lässt, zukünftig in das EMX-Modell integriert werden.

Wilde führt aus, dass für ein konkretes konzeptuelles Modell eine Wichtung der einzelnen Eigenschaften notwendig ist, da sich nicht alle Eigenschaften gleichzeitig umsetzen lassen.
Der Schwerpunkt bei der Entwicklung des EMX-Modells liegt auf der graphischen Notation. Die anderen Punkte finden aber auch ihre Berücksichtigung. Viele Einschränkungen ergeben sich aus der Vorbedingung - ein konzeptuelles Modell für XML-Schema (XSD) zu erstellen. Das EMX-Modell ist leicht erweiterbar, sodass noch fehlende Aspekte einfach nachgerüstet werden können.

3.2 Verschiedenes

KLARLUND, SCHWENTICK und SUCIU stellen in [KSS03] einige allgemeine Aussagen und Definitionen vor, die auch im Rahmen dieser Diplomarbeit an unterschiedlichen Stellen berücksichtigt werden müssen. Sie können nicht eindeutig einem bestimmten Thema zugeordnet werden. Deshalb werden sie an dieser Stelle vorgestellt.

Innere Ordnung des XML-Dokuments Wenn ein XML-Dokument aus einer Datei eingelesen wird, entspricht die Ordnung $<$ der Reihenfolge der Knoten innerhalb der Datei. Wird aber das XML-Dokument auf andere Art generiert, muss darauf geachtet werden, dass eine solche Ordnung definiert ist. Dies gilt beispielsweise für Ergebnisse von XQuery oder XPath-Anfragen. Die Zusicherung der innere Ordnung spielt auch in den verschiedenen Modellen eine Rolle.

Sequenzen von Elementen Normalerweise besitzt jedes XML-Dokument genau ein Root-Element. In einigen Fällen kann es jedoch sinnvoll sein, mit Sequenzen von Bäumen zu arbeiten. Ein mögliches Beispiel sind Log-Dateien im XML-Format. Hier wird für jeden neuen Eintrag ein XML-Element an das Ende einer Datei angehängt. Ein weiteres Beispiel sind Anfragen in XQuery, die eine Liste von Elementen zurückliefern können. In der XML-Syntaxbeschreibung [XML00] ist dafür bereits das Konzept der „well-formed external parsed entities“ enthalten.

Element-Normal-Form (ENF) ENF ist eine Spezialform eines XML-Dokumentes. Ein XML-Dokument liegt in *ENF* vor, wenn es keine Attribute enthält. Für jedes XML-Dokument kann ein XML-Dokument in ENF erzeugt werden, indem man jedes Attribut durch ein Element ersetzt. Der Namen des neuen Elementes wird idealerweise aus einem Suffix (z. B. „`@`“) und dem Attributnamen gebildet. Dieses Vorgehen stellt sicher, dass auch eine Rückübersetzung möglich ist, da man am Elementnamen erkennt, ob es sich um ein „ehemaliges“ Attribut handelt.

Kapitel 4

Modelle

Ausgangspunkt für die Recherche über existierende Modelle für XML waren die Artikel [SM03] und [Wil05]. Für die Vorstellung einzelner Modelle wurden auch die Originalquellen herangezogen. Sengupta und Mohan führen in [SM03], ausgehend vom Abstraktionslevel, eine dreistufige Klassifikation für Modellen ein:

1. Physische Modelle
2. Formale Modelle
3. Konzeptuelle Modelle

Auf der untersten Abstraktionsstufe stehen *physischen Modelle*. Sie bilden die Grundlage für die Implementierung und enthalten Details über die Datenstruktur sowie Methoden und Mechanismen um auf den Daten zu navigieren.

Formale Modelle bilden die zweite Kategorie. Ein formales Modell ist eine notwendige Voraussetzung für die Entwicklung guter Sprachen und Verarbeitungsmethoden für die Daten sowie für die Erstellung effizienter Optimierungsstrategien für den Zugriff auf die physische Datenstruktur.

Konzeptuelle Modelle dienen vor allem zur Veranschaulichung und dem Verständnis der Daten. Aus ihnen lassen sich formale und physische Modelle ableiten.

Diese Einteilung ist nicht strikt. In der Praxis gibt es auch Modelle, die sich nicht eindeutig kategorisieren lassen.

Diese Klassifikation soll auch in dieser Arbeit übernommen werden. Sie bildet die Grundlage für die weitere Gliederung dieses Kapitels. Es werden zu jedem Punkt Beispielmodelle vorgestellt. Der Schwerpunkt liegt natürlich auf der Betrachtung der konzeptuellen Modelle.

4.1 Physische Modelle

Physische Modelle beschreiben, wie die XML-Daten gespeichert werden und wie Anfragen und Updates auf der Speicherstruktur realisiert werden können.

4.1.1 Strukturierte Daten in Dateien

ABITEBOUL, CLUET und MILO zeigen in [ACM98], dass sich aus dem Datenbankbereich bekannte Techniken auch auf, in Dateien gespeicherte, strukturierte Daten anwenden lassen. Sie zeigen, wie Anfragen und Updates durch deklarative Datenbanksprachen auf diesen Dateien umgesetzt werden können. Dazu führen sie ein strukturierendes Schema (*engl.* structuring schema) ein, welches auf einer mit Datenbankprogrammen annotierten Grammatik basiert. Ausgehende von diesem Schema können Anfragen und Updates wie auf einer Datenbankstruktur umgesetzt werden. [ACM98] führt eine formale Definition für das strukturierende Schema ein und zeigt detailiert, wie sich Anfrageoptimierung und Updates durchführen lassen.

Die Erkenntisse dieses Artikel lassen sich auch auf Dateien im XML-Format anwenden. Abbildung 4.1 zeigt ein einfaches Beispiel für eine XML-Datei.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--einfaches Beispiel-->
<Hotel sterne="5">
 <Name>Hotel NEPTUN</Name>
 <Adresse>
   <Straße>Seestraße 19</Straße>
   <PLZ>18119</PLZ>
   <Ort>Warnemünde</Ort>
 </Adresse>
</Hotel>
```

Abbildung 4.1: Beispielhafte XML-Datei

Die XML-Syntax, basierend auf dem Standard des W3C [XML00], wird als bekannt vorausgesetzt und soll hier nicht weiter erläutert werden.

4.1.2 DOM

Das Document Object Model (DOM) [DOM00] ist eine vom W3C entwickelte plattform- und programmiersprachenneutrale Schnittstelle (API) zur Verarbeitung von XML-Dokumenten. DOM definiert lediglich die logische Struktur. Inhalt, der auf physischer Ebene auf mehrere XML-Dokumente verteilt sein kann, wird zu einem *logischen* Dokument zusammengefasst.

Mit DOM kann man Dokumente erstellen, durch die Dokumentenstruktur navigieren sowie Elemente und Inhalte erstellen, ändern oder löschen.

Im DOM haben die Dokumente meist eine baum-ähnliche logische Struktur; genauer betrachtet ähnelt sie eher einen „Wald“, da sie auch aus mehren Bäumen bestehen kann. Die logische Struktur ist jedoch nicht explizit vorgeschrieben; das interne logische Modell kann auf jede geeignete Weise implementiert werden.

Für die unterschiedlichen Bestandteile eines XML-Dokumentes werden verschiedene Klassen von Knoten (`Element`, `Attr`, `Text`, ...) eingeführt.

Als Objektmodell im klassischen objektorientierten Sinne, spezifiziert DOM:

- Schnittstellen und Objekte, um ein Dokument darzustellen und zu manipulieren.
- die Semantik (Eigenschaften und Verhalten) dieser Schnittstellen und Objekte
- Beziehungen und das Zusammenwirken zwischen diesen Schnittstellen und Objekten.

Abbildung 4.2 zeigt eine mögliche DOM-Struktur für die Beispieldatei aus Abbildung 4.1. (Es gibt keine normierte graphische Notation.)

4.1.3 XML Infoset und XML Data Model

XML Infoset

Das XML Information Set [XIS04], [Sal99] (auch Infoset) stellt eine konsistente Menge von Definitionen zur Verfügung, die von weiteren Spezifikationen (des W3C und anderer) genutzt werden können, um sich auf die Bestandteile eines XML-Dokuments zu beziehen. Für jedes wohlgeformte XML-Dokument, das die Namespace-Bedingungen aus dem Infoset-Standard [XIS04] erfüllt, existiert ein Infoset.

Das Infoset ist eine Sammlung von *Information Items*. Ein *Information Item* ist die abstrakte Beschreibung eines Teilstücks des XML-Dokuments. Jedes Information Item besitzt eine Menge von Eigenschaften. Es gibt elf verschiedene Arten von Information Items (*Document, Element, Attribute, Processing Instruction, Unexpanded Entity Reference, Character Information, Comment Information, Document Type Declaration, Unparsed Entity,*

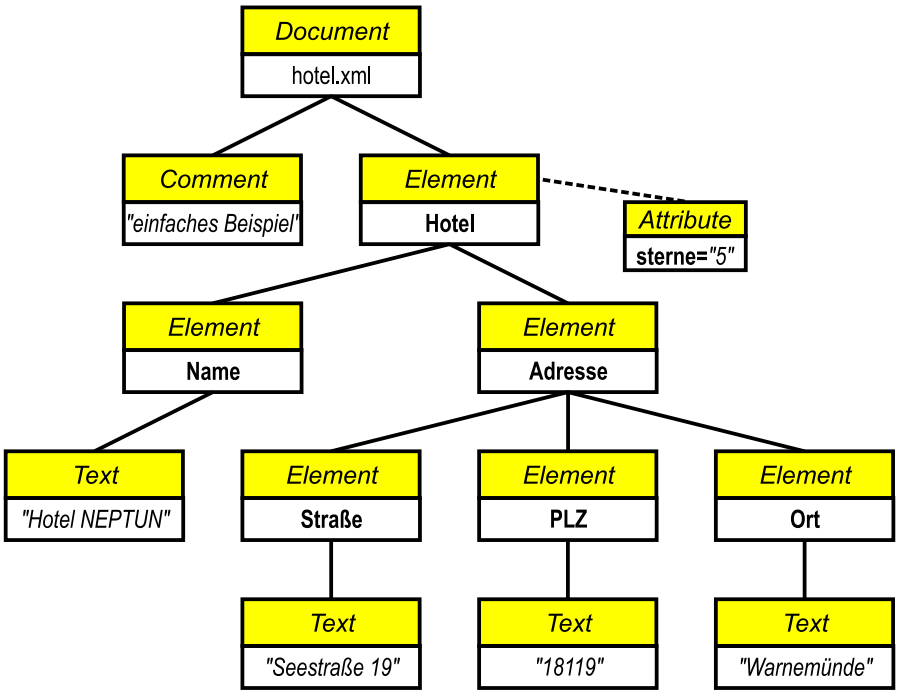


Abbildung 4.2: DOM-Struktur für Beispiel aus Abb. 4.1

Notation und Namespace). Jedes wohlgeformte XML-Dokument besitzt zumindest ein *Document Information Item*.

Die XML-Syntax selbst kann als eine mögliche Serialisierung der vom Infoset definierten logischen Struktur betrachtet werden.

XML Data Model

Bislang hatten die verschiedenen W3C-Spezifikationen XSLT, XPath und XQuery jeweils ein eigenes Datenmodell. Mit der W3C Candidate Recommendation *XQuery und XPath 2.0 Data Model (XDM)*, [XDM05] wurde ein einheitliches Datenmodell entwickelt, welches als Basis für zukünftige Versionen der W3C-Spezifikationen dienen soll.

Mit dem Datenmodell werden zwei Ziele verfolgt: Erstens definiert es die Struktur der Eingabe für einen XSLT- oder XQuery-Prozessor und zweitens definiert es alle zulässigen Werte und Ausdrücke in XSLT, XQuery und XPath.

XSLT 2.0, XQuery 1.0 und XPath sind bezüglich des Datenmodells *abgeschlossen*. Das bedeutet, das Eingabe und Ausgabe einer Anfrage sowie alle Werte und Ausdrücke, die in Berechnungsschritten auftreten können, durch das Modell definiert sind.

Das Datenmodell basiert auf dem Infoset-Modell und erweitert es um:

Unterstützung von XML-Schema-Typen.
Einfache und Komplexe Datentypen der XML-Schemaspezifikation [XS101] [XS201] erweitern das XML Infoset um exakte Typinformationen.

Darstellung für Dokumentkollektionen und Komplexe Werte.
Das sind Anforderung von XQuery.

Unterstützung für getypte atomare Werte.

Einführung geordneter, heterogener Sequenzen.
Das sind Sequenzen, die sowohl Elementknoten als auch atomare Werten enthalten können (Beispiel: `(1, <two />, 3)`). Sie können als Ergebnis bei der Anfragebearbeitung (z.B. in XQuery) auftreten.

Da das XML Data Model auch als Datenmodell für XML-Anfragesprachen verwendet wird, musste zugelassen werden, das es auch nicht-wohlgeformtes XML enthalten kann.

4.1.4 Speicherung von XML in Datenbanken

Prinzipiell lassen sich XML-Dokumente auch in Datenbanken speichern. Dazu gibt es verschiedene Konzepte, wie die XML-Strukturen auf Datenbankstrukturen abgebildet werden. Klettke und Meyer stellen in [KM03] die unterschiedlichen Ansätze vor. Sie unterscheiden drei Kategorien:

Speicherung als Ganzes.
In diesem Fall wird das XML-Dokument, so wie es als Datei vorlag, in die Datenbank übertragen. Das Modell ist somit analog zum physischen Modell der strukturierten Datei aus Abschnitt 4.1.1.

Generische Speicherung der Dokumentenstruktur.
Hier wird ein allgemein gültiges Datenbankschema verwendet, welches jedes beliebige wohlgeformte XML-Dokument speichern kann. Eine einfache Vorgehensweise wäre die Abbildung der Baumstruktur auf Tabellen eines relationalen Datenbanksystems. Tabelle 4.1 zeigt eine mögliche Speicherstruktur für das Beispieldokument aus Abbildung 4.1. Die Tabelle speichert XML-Objekte (Attribute und Elemente) mit eindeutiger ID, Art, Namen und Wert. Zusätzlich wird eine Referenz auf den Vorgängerknoten gespeichert. Die Ordnung sichert zu, dass bei der Wiederherstellung des XML-Dokumentes auch die Reihenfolge der Elemente wiederhergestellt werden kann.

DocID	Art	Name	ID	Vorg.	Ordnung	Wert
h001	elem	Hotel	101		1	
h001	attr	sterne	102	101	0	5
h001	elem	Adresse	103	101	1	
h001	elem	Straße	104	103	1	Seestraße 19
h001	elem	PLZ	105	103	2	18119
h001	elem	Ort	106	103	3	Warnemünde

Tabelle 4.1: Generische Speicherung für XML am Beispiel aus Abbildung 4.1)

Alternativ kann auch das Document Object Model (DOM) als Grundlage für die Entwicklung eines generischen Datenbankschemas verwendet werden.

Abbildung des Schemas auf die Datenbankstruktur.
Bei diesem Ansatz wird die interne Struktur der XML-Dokumente auf Datenbankstrukturen abgebildet. Die Tabellen werden ausgehend von einer Schemabeschreibung (z.B. XML-Schema, DTD) erstellt. Element- und Attributnamen treten also auch im Datenbankschema als Tabellen- oder Spaltennamen wieder auf. Das Mapping kann automatisch erfolgen. Andere Ansätze überlassen es dem Anwender, eine Mappingvorschrift, z.B. durch Annotieren des Schemas, explizit zu definieren. Da objektorientierte und objekt-relationale Datenbanksysteme hierarchische Strukturen speichern können, sind sie besser geeignet als die klassisch relationalen.

Bei allen Verfahren sind zusätzliche Maßnahmen zu ergreifen, um die innere Ordnung der Elemente zu sichern. Dies ist für die korrekte Wiederherstellung der XML-Dokumente unbedingt notwendig.

4.2 Formale Modelle

Formale Modelle für die Inhaltsbeschreibung von XML Dokumenten basieren auf unterschiedlichen Konzepten. In der Literatur findet man beispielsweise folgende Ansätze:

Das relationale Datenbankmodell kann auch als formales Modell für XML verwendet werden. Wie XML-Strukturen auf Datenbankstrukturen abgebildet werden können, wurde in Abschnitt 4.1.4 gezeigt. Deshalb kann das Relationenmodell als Basis für formale XML-Modelle verwendet werden. Die Erweiterung des Relationmodells durch komplexe Objekte oder geschachtelte Relationen vereinfacht die Erstellung formaler XML-Modelle, weil sich dadurch Hierarchien besser abbilden lassen.

Baumgrammatiken [CDG+97] lassen sich durch Erweiterungen ebenfalls zur Darstellung formaler Modelle für XML verwenden. Als Beispiele seien hier XGrammar [Man04, Abschnitt 3] und Hedge Grammars [Mur99] aufgeführt.

Graphen stellen eine weitere Modellierungsmöglichkeit dar. Als Vertreter dieser Kategorie sei das Datenmodell von SAL (Semi-Structured Algebra) [BT99] , einer algebraischen Anfragesprache, erwähnt.

In diesem Abschnitt sollen zwei Beispiele für formale Modelle vorgestellt werden.

4.2.1 XML-Baum

Auf Grund der hierarchischen Struktur von XML-Dokumenten sind Bäume eine geeignete Wahl als Grundlage für ein formales Modell. Die folgende Beschreibung eines *XML-Baums* basiert auf den Ausführungen in [KSS03]. Klarlund, Schwentick und Suciu beschreiben ein einfaches, generelles Modell, welches als Grundlage für die Graphenmodelle in verschiedenen Anwendungen wie zum Beispiel DOM [DOM00] dienen kann.

Die Definition des XML-Baumes basiert auf zwei Mengen: einer endlichen Menge von Bezeichnern Σ und einer unendliche Menge von Werten D (*engl.* data).

Bezeichner modellieren Namen für XML-Tags und Attribute. Der XML-Standard [XML00] schreibt vor, welche Zeichensätze verwendet werden können (z.B. Unicode) und welche Buchstabenkombinationen für die Namensbildung erlaubt sind. Die Autoren definieren Σ als endliche Menge, um das Modell an späterer Stelle für die Definition von Formalismen über XML-Anfragesprachen verwenden zu können. In der Praxis stellt dies keine Einschränkung dar, da in der Regel Tag- und Attributnamen durch ein Schema vorgegeben werden.

Im Gegensatz dazu ist die Menge D unendlich und kann *Werte* von Typ String, Integer, Date, u.s.w. enthalten.

Unter diesen Vorbedingungen besteht ein *XML-Baum*
$t = (N, E, <, \lambda, \nu)$ aus:

- einem gerichteten Baum (N, E) mit einer Menge von *Knoten* N und einer Menge von *Kanten* $E \subseteq N \times N$,
- einer *totalen Ordnung* $<$ über N, welche *depth-first* ist; das heißt: für jeden Vorgänger x von y gilt $x < y$.
- einer *partiellen Funktion* $\lambda : \ N \rightarrow \Sigma$, die einem Knoten einen Bezeichner zuweist, und
- einer *partiellen Funktion* $\nu : \ N \rightarrow D$, die einem Knoten einen Wert zuweist.

Die Menge N kann als Domäne von t aufgefasst werden: $N = dom(t)$.

Diese Definition stellt nur ein vereinfachtes Modell dar. Jedes wohlgeformte XML-Dokument lässt sich in einen solchen XML-Baum überführen. Allerdings lassen sich abstrakte XML-Bäume konstruieren, die keine wohlgeformten XML-Dokumente darstellen. Deshalb sind folgende weitere Regeln notwendig:

- N bestehe aus vier disjunkten Mengen:
  $N = \{d\} \cup Elmt \cup Attr \cup Txt$ mit:
  - d ist die Wurzel von t und wird *Dokumentknoten* genannt.
  - $Elmt$ ist die Menge der *Elementknoten*
  - $Attr$ ist die Menge der *Attributknoten*
  - Txt ist die Menge der *Textknoten*
- $Attr \cup Txt$ besteht nur aus Blattknoten, also gilt:
  wenn $(n, n') \in E$ **dann ist** $n \in \{d\} \cup Elmt$.
  Attribut-Child-Knoten stehen in der Ordnungrelation vor allen anderen Arten von Child-Knoten:
  wenn $(n, n_1), (n, n_2) \in E, \ n_1 \in Attr, \ n_2 \in Elmt \cup Txt$
  dann ist $n_1 < n_2$
- Die Bezeichnerfunktion $\lambda : \ N \rightarrow \Sigma$ ist definiert über $Elmt \cup Attr$ und undefiniert über allen anderen Knotenarten.
- Die Wertefunktion $\nu : \ N \rightarrow D$ ist definiert über $Attr \cup Txt$ und undefiniert über allen anderen Knotenarten.

- Child-Knoten des Dokumentknotens müssen Elementknoten sein:
  wenn $(d, n) \in E$ **dann ist** $n \in Elmt$

Etwas komplizierter ist der Fakt darzustellen, dass benachbarte Attributknoten ungeordnet sind. Das bedeutet, wenn ein Knoten mehrere Child-Knoten der Typen Text, Attribut und Element besitzt müssen durch die Ordnungsfunktion $<$ alle Attributknoten vor Element- und Textknoten sortiert werden. Unter den Attributknoten selbst lässt sich keine Ordnung definieren.

Ein anderes Problem ist die Terminologie des XML-Standards: Als Child-Knoten sind nur Element- und Textknoten definiert. Das bedeutet: Attributknoten sind keine Child-Knoten, obwohl sie Parent-Knoten besitzen!

4.2.2 Hedge-Grammatiken

Hedge-Grammatiken wurden in der XML-Gemeinschaft als einfaches, aber mächtiges Modell für XML-Schemasprachen erkannt. Die beiden Vorgänger von XML-Schema: RSL (Regular Schema Language) und DSD (Document Structure Description) basieren auf dieser Theorie. Die folgenden formalen Ausführungen basieren auf [Mur99]:

Hedges
Hedges (Hecken) lassen sich „nicht-formal§ als eine geordnete Reihe von Bäumen und Teilbäumen beschreiben. In der XML-Welt ist eine Hedge eine Sequenz von Elementen, die möglicherweise durch Text unterbrochen ist. Ein XML-Dokument ist eine Hedge. Formal wird eine Hedge wie folgt definiert:

Eine *Hedge* übere einer endlichen Menge Σ (von Symbolen) und einer endlichen Menge X (von Variablen) ist:

1. ϵ, die „Null“-Hedge,
2. x, als eine Variable aus X,
3. $a\langle u\rangle$, mit a als einem Symbol aus Σ und u als Hedge
   (das Hinzufügen eines Symbols als Wurzelknoten), oder
4. uv, mit u und v als Hedges (Die Verkettung zweier Hedges).

Abbildung 4.3 zeigt drei Beispiel-Hedges: $a\langle\epsilon\rangle$, $a\langle x\rangle$ und $a\langle x\rangle b\langle c\langle\epsilon\rangle x\rangle$. Man beachte, das Elemente von Σ (z. B. a und b) die Nicht-Blattknoten und Elemente von X (z. B. x) die Blattknoten bilden.

In Abbildung 4.4 wird eine Hedge für ein Beispiel-XML-Fragment dargestellt.

Abbildung 4.3: Drei Beispiele für Hedges: $a\langle\epsilon\rangle$, $a\langle x\rangle$ und $a\langle x\rangle b\langle c\langle\epsilon\rangle x\rangle$

XML-Element

```
<adresse>
  <straße> #PCDATA </straße>
  <plz> #PCDATA </plz>
  <ort> #PCDATA </ort>
</adresse>
```

Hedge

```
adresse<
  straße<#PCDATA>
  plz<#PCDATA>
  ort<#PCDATA>
>
```

Abbildung 4.4: Beispiel-Hedge für XML-Fragment

Reguläre Hedge-Grammatik
In diesem Abschnitt wird eine *Reguläre Hedge-Grammatik* (RHG) zur Generierung von Hedges eingeführt. Mit anderen Worten: eine *RHG* beschreibt eine Menge von Hedges.

Eine *Reguläre Hedge Grammatik* (RHG) ist ein 5-Tupel $\langle\Sigma, X, N, P, r_f\rangle$ aus:

1. Σ, einer endlichen Menge von Symbolen
2. X, einer endlichen Menge von Variablen
3. N, einer endlichen Menge von Nicht-Terminalen
4. P, einer endlichen Menge von *Produktionsregeln* einer der folgenden Formen
   (a) $n \to x$ mit $n \in N$ und $x \in X$
   (b) $n \to a\langle r\rangle$ mit $n \in N$ und $a \in \Sigma$ und r einem regulären Ausdruck aus Nicht-Terminalen.
5. r_f, einem regulären Ausdruck aus Nicht-Terminalen

Jetzt sollen die *Ableitungsregeln* betrachtet werden. Einfach ausgedrückt, werden in einer gegebenen Folge von Nichtterminalen wiederholt Nichtterminale durch Hedges von rechten Seiten anwendbarer Produktionsregeln ersetzt.

Hedge v ist direkt von Hedge u abgeleitet, wenn gilt:

1. Bei einer Produktionsregel $n \to x$ erzeugt man eine Hedge v durch Ersetzen eines Vorkommens von n in u durch x oder
2. Bei einer Produktionsregel $n \to a\langle r \rangle$ wird Hedge v erzeugt, indem ein Auftreten von n in u durch ein $a\langle w \rangle$ ersetzt wird, so dass w eine Folge von Nichtterminalen ist, die r entspricht.

Die Sprache $L(G)$, die durch G erzeugt wird, ist eine Menge von Hedges, welche durch Anwendung der Produktionsregeln auf r_f erzeugt werden.

Sei zum Beispiel folgende RGH gegeben:
$G = \langle \{a\}, \{x\}, \{n_1, n_2\}, P, n_1? \}$
mit $P = \{n_1 \to a\{n_2^+\},\ n_2 \to x\rangle$.
Dann ist $L(G) = \{\epsilon, a\langle x \rangle, a\langle xx \rangle, a\langle xxx \rangle, \ldots\}$ die erzeugte Sprache.

DTD

```
<!ELEMENT adresse
  ((straße|postfach),plz, ort)>
<!ELEMENT straße (#PCDATA)>
<!ELEMENT postfach (#PCDATA)>
<!ELEMENT plz (#PCDATA)>
<!ELEMENT ort (#PCDATA)>
```

Hedge-Grammatik

$G=(\Sigma, X, N, P, n_{adr})$

Σ={adresse, straße, postfach, plz, ort}

X={#PCDATA}

$N=\{n_{adr}, n_{str}, n_{pf}, n_{plz}, n_{ort}, n_{\blacklozenge}, n_1\}$

$P=\{$
$n_{adr} \to$ adresse$(n_1\ n_{plz}\ n_{ort})$
$n_1 \to n_{str} \mid n_{pf}$
$n_{str} \to$ straße$(n_{\blacklozenge})$
$n_{pf} \to$ postfach$(n_{\blacklozenge})$
$n_{plz} \to$ plz$(n_{\blacklozenge})$
$n_{ort} \to$ ort$(n_{\blacklozenge})$
$n_{\blacklozenge} \to$ #PCDATA $\}$

Abbildung 4.5: Hedge-Grammatik für Beispiel-DTD

In Abbildung 4.5 wird zu einer DTD die ihr entsprechende Hedge-Grammatik dargestellt.

4.3 Konzeptuelle Modelle

Bis zum jetzigen Zeitpunkt hat sich noch keine Methode zur konzeptuellen Modellierung von XML-Schemabeschreibungen durchsetzen können. Existierende Ansätze für den Entwurf von XML-Inhaltsmodellen basieren auf bekannten Modellierungskonzepten. Sie sind häufig Erweiterungen von:

- Entity-Relationship-Modell
- UML

In diesem Kapitel werden diese Modellierungskonzepte sowie entsprechende Beispiel für ihre Anwendung als Basis für die konzeptuelle Modellierung von XML-Inhaltsmodellen vorgestellt.

4.3.1 Das Entity-Relationship-Modell (ER-Modell)

Das Entity-Relationsship-Modell ist wohl das bekannteste Modell für den konzeptuellen Datenbankentwurf. CHEN hat es 1976 in [Che76] erstmals vorgestellt. Im Laufe der Zeit wurde das Modell genauer erforscht und an einigen Stellen erweitert. Heute ist es Bestandteil eines jeden Lehrbuches über den Datenbankentwurf. Als problematisch erweist sich, dass es keinen einheitlichen Standard für die Notation gibt. So werden in verschiedenen Artikeln und Büchern die Konzepte unterschiedlich dargestellt, beziehungsweise Details wie Kardinalitäten oder Datentypen unterschiedlich behandelt.

Im Folgenden sollen die wesentlichen Merkmale kurz vorgestellt werden. Der Schwerpunkt liegt dabei auf den Konzepten, die sich auch für die Modellierung von XML-Inhaltsmodellen eignen. Details zum formalen Hintergrund entnehme man der Literatur (z.B. [Che76] [HS00]).

Die folgenden Definitionen basieren auf den Ausführungen von HEUER und SAAKE [HS00]), die bereits die unterschiedlichen Ausprägungen und Erweiterungen für das ER-Modell vorstellen. An einigen Stellen wird auch auf das Originalmodell von CHEN [Che76] zurückgegriffen. Der wesentliche Unterschied zwischen dem Originalartikel [Che76] und der Beschreibung von HEUER und SAAKE besteht darin, dass CHEN gleichartige Entitäten und Relationen zu Mengen zusammenfasst, während HEUER und SAAKE dazu Typen für Entitäten und Beziehungen einführen. Der Vorteil von Typen liegt darin, dass sie eine Objektmenge und Funktionen, die auf den Objekten definiert sind, miteinander kombinieren.

Die drei wesentlichen Bestandteile des ER-Modells sind Entität, Beziehung und Attribut.

Entität. Eine Entität (*engl.* entity) repräsentiert ein Objekt der realen Welt, welches eindeutig identifizierbar ist. Eine bestimmte Person oder eine Firma können Beispiele für Entitäten sein.

Beziehung. Eine Beziehung (*engl.* relationsship) beschreibt einen Zusammenhang zwischen Entitäten. Zum Beispiel `ARBEITSVERTRAG` ist eine Beziehung zwischen einer `PERSON`-Entität und einer `FIRMA`-Entität. Es sind auch Beziehungen zwischen mehr als zwei Entitäten möglich.

Attribut. Attribute (*engl.* attributes) enthalten Informationen über Entitäten oder Beziehungen. Sie können entweder vom Anwender definiert oder berechnet werden.

Im Gegensatz zu Attributen sind Entitäten nicht direkt durch Werte darstellbar. Sie sind nur durch ihre Eigenschaften repräsentierbar. Die Entscheidung, ob ein Faktum als Entität oder Beziehung definiert wird, ist abhängig vom Standpunkt des Entwicklers. Er sollte die Repräsentation wählen, die ihm am besten geeignet für den Sachverhalt scheint.

Weiter notwendige Konzepte sind:

Werte. Werte sind einfache Datenelemente, die direkt darstellbar sind.

Datentypen. Datentypen charakterisieren Werte. Sie bestehen aus einer Wertemenge und Basisoperationen auf diesen Werten. Zunächst sind nur einfache Datentypen wie `INTEGER` oder `STRING` zugelassen; das Konzept wird später erweitert.

ER-Diagramm

Chen führt in [Che76] auch eine graphische Repräsentation für das ER-Modell ein - das ER-Diagramm. Auch zum *ER-Diagramm* gibt es eine Reihe von Erweiterungen, die ebenfalls von Heuer und Saake zusammengetragen wurden. Es soll hier wieder vor allem auf die Konzepte eingegangen werden, die für die Modellierung von XML-Inhaltsmodellen von Interesse sein könnten.

Folgende Konzepte können in einem ER-Diagramm dargestellt werden:

Entitäts-Typen. Entitäts-Typen werden durch Rechtecke repräsentiert.

Beziehungs-Typen. Beziehungs-Typen fassen Beziehungen mit gleichen Eigenschaften zusammen. Sie werden durch Rauten dargestellt.

Rollen. Rollen dienen der Vermeidung von Mehrdeutigkeiten. Diese können entstehen, wenn Entitäten vom gleichen Typ in eine Beziehung eingehen. Das klassische Beispiel ist die Beziehung `VERHEIRATET` über zwei Entitäten vom Typ `PERSON`. Hierbei müssen die Rollen `MANN` und `FRAU` zugewiesen werden. In der graphischen Notation wird der Rollenname einfach an die Verbindungslinie geschrieben.

Attribute. Attribute werden durch Rechtecke mit abgerundeten Ecken dargestellt (siehe Abbildung 4.7). Andere Ansätze verwenden Ellipsen oder kleine Kreise mit Bezeichner als Notation. Zusätzlich zum Namen kann man auch den Typ (getrennt durch „:“) in den Bezeichner aufnehmen.

Kardinalitäten geben an, ob nur eine Entität (`1`) eines Types oder mehrere Entitäten (`N` oder `M`) in die Beziehung eingehen können.

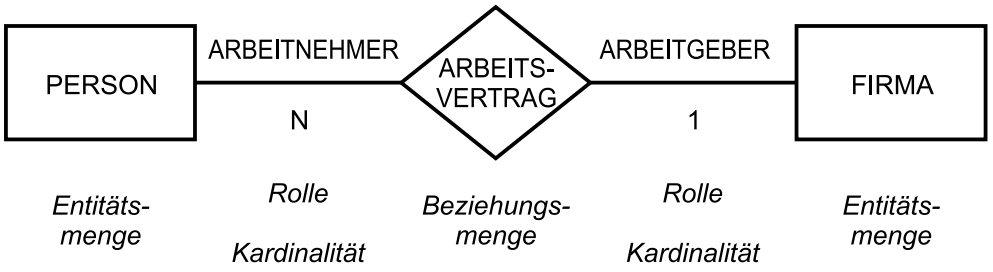


Abbildung 4.6: Ein einfaches ER-Diagramm

Abbildung 4.6 zeigt ein einfaches Beispieldiagramm. Der Beziehungstyp `ARBEITSVERTRAG` ist definiert zwischen den Entitätstypen `PERSON` und `FIRMA`. Personen gehen mehrfach mit der Rolle `ARBEITNEHMER` und Firmen einfach mit der Rolle `ARBEITGEBER` in die Beziehung ein.

Zusätzlich zu den bisher beschriebenen Basiskonzepten gibt es Konstrukte um das ER-Modell zu erweitern; diese sollen hier kurz vorgestellt werden:

Funktionale Beziehung. Die Funktionale Beziehung ist ein spezieller Beziehungstyp. Es liegt eine zweistellige Beziehung vor, die eindeutig eine Entität zu einer anderen Entität zuordnet. In der graphischen Notation

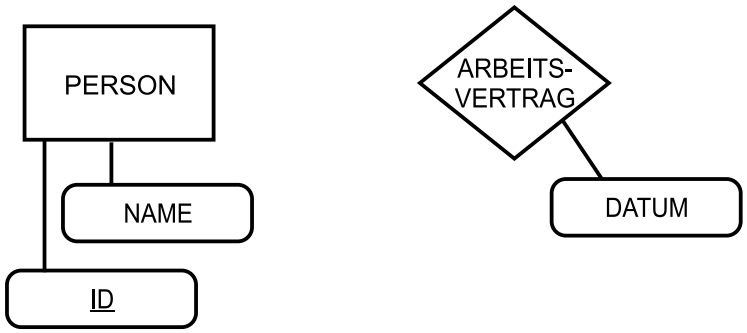


Abbildung 4.7: Darstellung von Attributen im ER-Diagramm

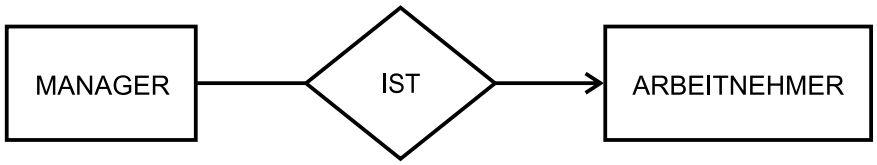


Abbildung 4.8: IST-Beziehung im ER-Diagramm

wird dieser Sachverhalt durch eine Pfeilspitze an der Verbindungslinie, die in die zugeordnete Entität eingeht, dargestellt.

Schlüssel. Ein Schlüssel ist ein Attribut oder eine Attributmenge, die eine Entität innerhalb eines Entitätstypes eindeutig identifiziert. In der graphischen Notation werden Schlüsselattribute unterstrichen oder die zugehörenden Verbindungslinien speziell gekennzeichnet.

IST-Beziehung. Die IST-Beziehung stellt eine Spezialisierungs-/Generalisierungsbeziehung zwischen Entitäten dar.

Abbildung 4.8 zeigt ein Beispiel für eine IST-Beziehung. Jeder `MANAGER`-Instanz wird eine `ARBEITNEHMER`-Instanz zugeordnet. Jeder Manager ist auch ein Arbeitnehmer, aber umgekehrt gilt nicht, dass jeder Arbeitnehmer ein Manager ist. Die Attribute (sowohl Deklaration als auch Werte) der `ARBEITGEBER`-Instanz gelten auch für die `MANAGER`-Instanz. Sie werden „vererbt“.

Kardinalitäten werden erweitert. Es ist jetzt möglich, für jeden Entitätstyp ein Intervall `[MIN, MAX]` zu definieren, das beschreibt, wie oft Entitäten des Typs in die Beziehung eingehen können. Wird kein Intervall definiert, wird automatisch `[0,*]` angenommen.

Optionalität von Attributen. Durch eine spezielle Markierung an der Verbindungslinie können Attribute als „optional“ deklariert werden.

Erweiterung bei Attributen. Während das klassische ER-Modell nur Standarddatentypen für Attribute verwendet, erweitern es neuere Ansätze um zusammengesetzte Datentypen und mengenwertige Attribute. Einige Erweiterungen lassen generell abstrakte Datentypen als Wertebereich für Attribute zu.

Bereits Chen [Che76] führte abgeleitete Attribute ein, deren Werte nicht gespeichert, sondern berechnet werden können. Als graphische Notation werden gestrichelte Verbindungs- und Begrenzungslinien vorgeschlagen.

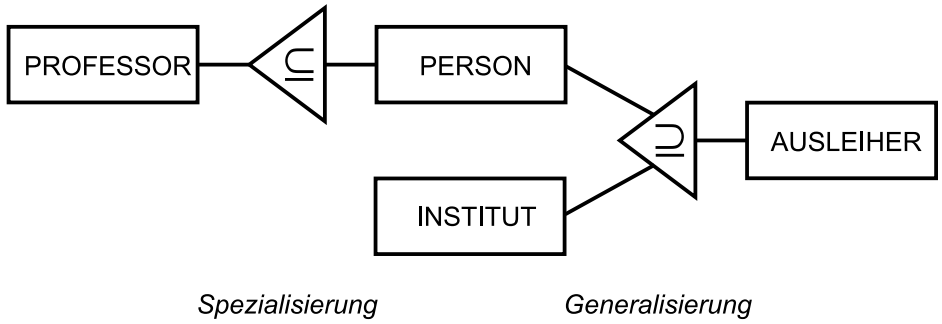


Abbildung 4.9: Darstellung von Spezialisierung und Generalisierung im ER-Diagramm

Spezialisierung und Generalisierung. *Spezialisierung* liegt vor, wenn es sich um eine Teilmengenbeziehung handelt und die Eigenschaften des allgemeineren Entitätstyps auf den speziellen Entitätstyp vererbt werden. Ein PROFESSOR ist zum Beispiel eine Spezialisierung einer PERSON, die Attribute wie NAME und AUSWEIS-ID übernimmt. *Generalisierung* wird benötigt, wenn Entitäten verallgemeinert werden sollen. So können beispielsweise PERSONEN und INSTITUTE als AUSLEIHER in einer Bibliothek auftreten. Hier ist zu beachten, dass die Attribute der verallgemeinerten Entität auch in den Ausgangsentitäten vorhanden sein müssen.

HEUER und SAAKE weisen explizit darauf hin, dass die Begriffe in der Literatur auch vertauscht oder synonym gebraucht werden. Eine mögliche graphische Notation zeigt Abbildung 4.9.

4.3.2 Algorithmus nach Kleiner-Lipeck

KLEINERT und LIPECK stellen in [KL02] einen Algorithmus vor, der die Strukturinformationen eines konzeptuellen Schemas in eine DTD überträgt. Es wird zunächst dargestellt, wie die Modellierungskonstrukte eines „klassischen“ ER-Modells in eine DTD übertragen werden. Anschließend wird gezeigt, wie mit Konstrukten eines erweiterten ER-Modells (EER-Modell) zu verfahren ist.

Der Algorithmus

Die folgende Beschreibung des Algorithmus im Pseudocode entstammt [KL02].

1. Verwende `<SchemaName>` als Wurzelelement
2. Subelemente der Kardinalität `*` des Wurzelelements `<SchemaName>` sind:
   (a) Elemente für alle Entitäten, die nicht auf der n-Seite einer 1:n-Beziehung auftreten
   (b) Elemente für alle binären n:m-Beziehungen und alle k-stelligen Beziehungen mit $k = 2$
   (c) Elemente für alle Entitäten, die nur auf n-Seiten und nur partiell in Beziehungen vorkommen
3. Solange noch nicht alle bisher verwendeten Elemente zu Entitäten in der DTD definiert sind, definiere das nächste Element wie folgt (wähle dabei zunächst Entitäten mit Beziehungen zu schwachen Entitäten):
   (a) Führe ein Unterelement für jedes zusammengesetzte Attribut ein
   (b) Führe ein Unterelement mit der Kardinalität `*` oder `+` für jedes mehrwertige Attribut ein
   (c) Definiere ein Unterelement für jede Beziehung zu einem schwachen Entitätstyp (Kardinalität wie im ER-Schema)
   (d) Führe ein Unterelement mit der Kardinalität `*` oder `+` für jede 1:n-Beziehung ein, bei der das aktuelle Element auf der 1-Seite steht
   (e) Definiere XML-Attribute für alle Attribute der aktuellen Entität mit einfachem Datentypen (Schlüsselattribute als `ID`, andere als `CDATA`)
   (f) Definiere XML-Attribute für alle Attribute von 1:1-Beziehungen, an denen die aktuelle Entität total und die andere Seite partiell teilnimmt; definiere ein `IDREF`-Attribut für die andere Seite der Beziehung. Verfahre analog für 1:1 Beziehungen, an denen auch die andere Seite total teilnimmt, wenn die Entität bereits vorher übersetzt wurde
   (g) Füge alle 1:1-Beziehungen mit totaler Beteiligung auf beiden Seiten, bei denen die Entität auf der anderen Seite noch nicht übersetzt ist, alle Definitionen für die Entität auf der anderen Seite sowie für alle Beziehungsatttribute zu der aktuellen Definition hinzu

(h) Für jedes Beziehungselement aus (c) und (d):

i. Definiere ein Subelement für die andere Seite der Beziehung, falls diese Entität noch nicht übersetzt wurde

ii. Definiere ein `IDREF`-Attribut für die andere Seite der Beziehung, wenn diese Entität vorher übersetzt wurde

iii. Definiere Attribute für alle Beziehungsattribute wie oben für Attribute von Entitäten

4. Für alle Beziehungselemente von n:m- oder k-stelligen Beziehungen mit $k > 2$:

(a) Führe ein `IDREF`-Attribut für alle beteiligten Entitäten ein

(b) Definiere Attribute bzw. Subelemente für alle Beziehungsattribute wie zuvor bei Attributen von Entitäten

Behandlung von Konstrukten eines Erweiterten ER-Modells

An dieser Stelle geht es um die Unterstützung von Generalisierung und Spezialisierung sowie Kategorien und Vereinigungstypen. Die Autoren unterscheiden nicht zwischen Generalisierung und Spezialisierung und stellen folgende drei Formen vor:

Disjunkte Spezialisierung mit totaler Beteiligung. Für diese wohl verbreitetste Form schlagen die Autoren vor, XML-Elemente sowohl für Ober- als auch Unterklassen zu verwenden. Oberklassen werden „normal“ durch den Algorithmus behandelt. Für die Elemente der Oberklasse wird folgende innere Struktur festgelegt:
`<!ELEMENT O (S1|...|SN)>`
Dadurch wird gleichzeitig die Abstraktheit der Oberklasse erzwungen.

Disjunkte, partielle Spezialisierung. Diese Form wird auf ähnliche Weise definiert, nur dass die Subelemente optional verwendet werden können; das bedeutet, dass auch die Oberklasse instanziert werden kann. Der entsprechende DTD-Ausdruck ergibt sich wie folgt:
`<!ELEMENT O (S1|...|SN)?>`.

Überlappende Spezialisierungen. Diese Form wird im totalen und partiellen Fall gleich behandelt. Die generelle Herangehensweise ist wieder dieselbe. Der zu erzeugende DTD-Ausdruck für die Oberklasse lautet nun: `<!ELEMENT O (S1?|...|SN?)>`. Dadurch wird erreicht, dass zu einer Instanz der Oberklasse mehrere Instanzen von Unterklassen möglich sind.

Vollständige Kategorie. Diese Form wird ähnlich wie eine disjunkte, vollständige Spezialisierung behandelt. Für alle Oberklassen, die durch

eine Unterklasse kategorisiert werden, werden XML-Elemente erzeugt. Sie enthalten ein obligatorisches Element für die Unterklasse.

Partielle Kategorie. In diesem Fall ist das Subelement in allen Oberklassen-Elementen optional.

Es sei angemerkt, dass alle Umformungen umkehrbar sind.

Bewertung

Kleiner und Lipeck analysieren den Algorithmus auch bezüglich Qualität und Umkehrbarkeit. Indem sie versuchen aus der DTD wieder das Originalmodell zu erzeugen, ermitteln sie den Informationsverlust. Teile, die nicht rekonstruierbar sind, stellen verlorene Informationen dar. Sie kommen zu dem Ergebnis, dass die meisten Informationen sich problemlos wiederherstellen lassen. Lediglich in den folgenden Fällen gelingt dies nicht:

- Beteiligte Entitäten, die durch IDREF abgebildet worden sind, lassen sich nicht vollständig wiederherstellen. Es lässt sich nicht mehr ermitteln, welche Entitäten in die Beziehung eingegangen sind.
- Attribute von 1:1-Beziehungen, die dem Element auf der Seite der totalen Beteiligung hinzugefügt wurden, lassen sich nicht mehr von Attributen der Entität selbst unterscheiden.
- Schwache (abhängige) Entitätstypen lassen sich nur als gewöhnliche Entitätstypen wiederherstellen.
- Rollennamen im ER-Modell wurden nicht in die DTD übernommen und lassen sich so nicht wiederherstellen.
- Zusammengesetzte Schlüssel (z.B. Attributkombinationen) lassen sich nicht mehr trennen, da sie als ein Attribut umgesetzt wurden.

Alle diese Problemfälle lassen sich durch Annotationen im XML-Dokument umgehen, falls eine vollständige Rekonstruktion des ER-Modells notwendig sein sollte, und stellen somit keine größeren Probleme dar.

4.3.3 X-Entity-Modell

Auch das X-Entity-Modell [LSRGa03] von Lòscio, Salgado und do Rêgo Galvão ist eine Erweiterung des ER-Modells. Sie stellen zusätzlich zur graphischen Repräsentation auch eine textuelle Repäsentation vor. In einem weiteren Abschnitt beschreiben sie den Konvertierungsprozess von einem XML-Schema in ein X-Entity-Schema.

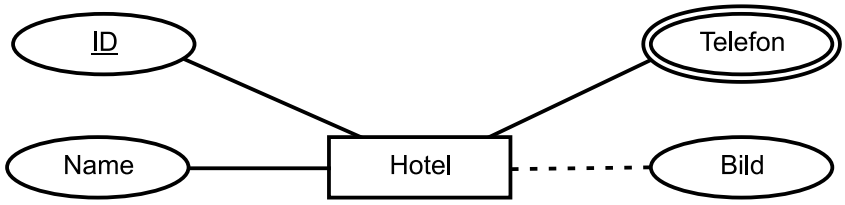


Abbildung 4.10: Darstellung von Entitätstypen in X-Entity

Grundlegende Konzepte

Entitätstypen. Ein Entitätstyp `E` wird durch $\texttt{E}(\{\texttt{A}_1,\ldots,\texttt{A}_n\},\{\texttt{R}_1,\ldots,\texttt{R}_m\},\{\texttt{D}_1,\ldots,\texttt{D}_k\})$ repräsentiert und besteht aus einem Entitätsnamen `E`, einer Attributmenge $\{\texttt{A}_1,\ldots,\texttt{A}_n\}$, einer Relationenmenge $\{\texttt{R}_1,\ldots,\texttt{R}_m\}$ und einer Menge von Disjunktionsbeschränkungen $\{\texttt{D}_1,\ldots,\texttt{D}_m\}$.

Ein Entitätstyp repräsentiert eine Menge von Elemente mit einer komplexen Struktur. Jeder Entitätstyp besteht aus Attributen, die ihn beschreiben. Attribute im X-Entity-Modell repräsentieren sowohl XML-Attribute als auch Subelemente mit einfachem Datentyp. Jedem Attribut $\texttt{A}_\texttt{i}$ wird eine Domäne $\texttt{Dom}(\texttt{A}_\texttt{i})$ zugeordnet, die seinen Wertebereich darstellt. Zusätzlich wird $\texttt{A}_\texttt{i}$ eine Kardinalität $\texttt{Card}(\texttt{A}_\texttt{i}) = (\texttt{min},\texttt{max})$ zugeordnet, die die minimale und maximale Anzahl von Instanzen von $\texttt{A}_\texttt{i}$ angibt, welche einer Instanz von `E` zugeordnet werden können. Das Schlüsselkonzept aus dem ER-Modell wurde ebenfalls integriert.

Abbildung 4.10 zeigt die graphische Repräsentation eines Entitätstyps `Hotel`. Wie im ER-Modell üblich, werden Entitäten als Rechtecke und Attribute als Ellipsen dargestellt. Schlüsselattribute (z.B. `ID`) werden unterstrichen. Optionale Attribute (z.B. `Bild`) werden durch eine gestrichelte Verbindungslinie dargestellt. Attribute, die mehrfach auftreten können (z.B. `Telefon`) werden doppelt umrahmt.

Enthaltensein-Beziehung. Eine Enthaltensein-Beziehung (*engl.* containment relationship) zwischen zwei Entitätstypen sagt aus, dass jede Instanz von `E` Instanzen von $\texttt{E}_1$ enthält. Die textuelle Repräsentation lautet $\texttt{R}(\texttt{E},\texttt{E}_1,(\texttt{min},\texttt{max}))$ mit dem Namen `R` und den minimalen und maximalen Kardinalitäten $(\texttt{min},\texttt{max})$. Jedem Entitätstyp können mehrere Enthaltensein-Beziehungen zugeordnet werden.

Abbildung 4.11: Darstellung von Enthaltensein-Beziehungen in X-Entity

Die graphische Repräsentation (Abbildung 4.11) ist in Anlehnung an das ER-Modell eine Raute mit dem Text `contains`. Der enthaltene Entitätstyp wird durch die Pfeilspitze an der Verbindungslinie gekennzeichnet. Kardinalitäten werden an der Verbindungslinie notiert. Das Beispiel beschreibt, dass ein `Katalog` mehrere `Hotels` enthält.

Disjunktionsbeschränkung. Disjunktionsbeschränkungen (*engl.* disjunction constraints) werden als $\texttt{D}(\texttt{d}_1, \ldots, \texttt{d}_n)$ dargestellt. $\texttt{d}_i$ kann ein Attribut, eine Enthaltensein-Beziehung, eine Attributmenge oder Menge von Enthaltensein-Beziehungen eines Entitätstyps `E` sein. Eine Disjunktion `D` legt fest, dass eine Instanz von `E` nur eines der genannten Konstrukte enthalten kann.

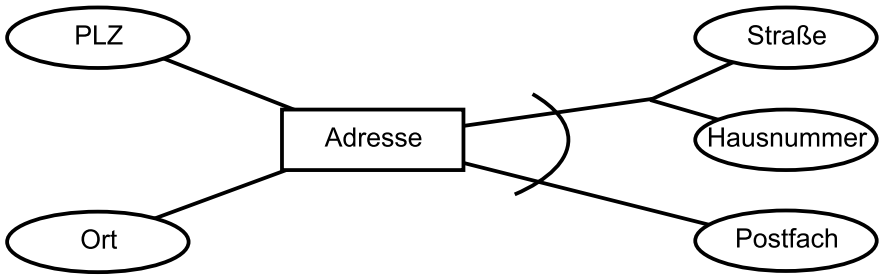


Abbildung 4.12: Darstellung von Disjunktheit in X-Entity

Graphisch wird die Beschränkung mittels eines Bogens über die zugehörenden Verbindungslinien dargestellt. (siehe Abbildung 4.12). Eine `Adresse` kann entweder `Straße` und `Hausnummer` oder ein `Postfach` enthalten. Gleichzeitig wird in Abbildung 4.12 gezeigt, wie Attribute (`Straße` und `Hausnummer`) gruppiert werden können.

Die Autoren beschreiben anschließend den Konvertierungsprozess eines XML Schemas in ein X-Entity-Schema. Dieser besteht aus zwei Schritten.

Schritt 1: Vorbereitung. In dieser Phase wird das Schema modifiziert, um den Ableitungsprozess zu vereinfachen. Im Wesentlichen werden die folgenden Schritte ausgeführt:

1. Auflösen benannter (*engl.* named) Element- und Attributgruppen durch Einsetzen der entsprechenden Definition.
2. Auflösen anonymer Typdefinitionen
3. Beseitigung von Elementdeklarationen, die nur als Hilfsmitel bei der Definition von Inhaltsmodellen anderer Elemente dienen und alleine keine Bedeutung haben.

4. Umbenennung von Elementen mit gleichem Namen und verschiedenem Datentyp. XML-Schema erlaubt deren Definition, solange sie in unterschiedlichem Kontext stehen. Diese lokale Sicht auf Elemente lässt sich mit X-Entity nicht modellieren.

Schritt2: Umwandlung. Zur Beschreibung des Umwandlungsprozesses führen die Autoren eine textuelle Repräsentation für XML-Schema ein, die der textuellen Repräsentation des X-Entity-Modells ähnelt. Anschließend geben sie eine Reihe von Regeln an, die den Ableitungsprozess beschreiben. Diese sind im einzelnen:

- Umwandlung von Elementen mit komplexem Typ
- Umwandlung von Elementen mit Basistyp
- Umwandlung von Attributen
- Erzeugen von Enthaltensein-Beziehungen

Die genaue Beschreibung dieser Regeln entnehme man [LSRGa03].

Bewertung

Die vorgestellten Erweiterungen lösen einige der Probleme, die sich bei der Übertragung von XML-Inhaltsmodellen in das ER-Modell ergeben. Die eingeführte Enthaltensein-Beziehung ermöglicht die Darstellung komplexer Typdefinitionen. Mittels der Disjunktionsbeschränkung ist es möglich, Teile der Typdefinition optional zu gestalten.

Allerdings scheint die graphische Realisierung der Disjunktionsbeschränkung nur bedingt geeignet, um komplexe Inhaltsmodelle darzustellen. So müssen sich die Objekte, die verbunden werden sollen, in direkter Nachbarschaft befinden. Das macht die Arbeit mit einem graphischen Editor problematisch. Es lassen sich Fälle konstruieren, wo diese Bedingung nicht erfüllt ist. Diese Fälle würden dann sehr unübersichtlich oder gar falsch dargestellt (wenn der zu zeichnende Bogen auch nicht betroffene Verbindungslinien überspannt).

Leider ist die Vorstellung des Modells nicht vollständig. Wichtige Eigenschaften des X-Entity-Modells werden nur am Ende erwähnt und nicht genauer vorgestellt. So wird aus dem Artikel nicht ersichtlich, wie die Autoren die Reihenfolge der Subelemente definieren oder beim Reverse-Engineering erhalten wollen. Sie verweisen auf einen Sequenz-Operator (*engl.* sequence composer). Da die Reihenfolge von Subelementen ein wichtiges Details des XML-Inhaltsmodells ist, wäre eine etwas detailliertere Beschreibung dieses Konstrukts hilfreich gewesen. Auch Ausführungen, ob und wie das Modell Elemente mit gemischtem Inhalt (*engl.* mixed content) unterstützt, wären für eine abschließende Bewertung des Ansatzes notwendig gewesen.

4.3.4 XER - Extensible Entity Relationship Model

Auch XER, vorgestellt von Sengupta, Mohan und Doshi in [ASD03] ist eine Erweiterung des ER-Modells. Zur Vereinfachung der Modellierung setzen die Autoren Element Normal Form (ENF) voraus. Wie bereits in Abschnitt 3.2 beschrieben wurde, ist diese einfach herzustellen.

Schwerpunkte der Modellierung von XML

Die Autoren sehen folgende Faktoren, die die XML-Modellierung mittels ER-Diagrammen verkomplizieren:

- XML-Objekte haben eine innere Ordnung, sowohl zwischen verschiedenen Elementen als auch verschiedenen Instanzen desselben Elements.
- n:m-Beziehungen werden nicht direkt unterstützt, da die Struktur von XML-Dokumenten hierarchisch ist.
- Es können heterogene Typen auftreten - das bedeutet das verschiedene Instanzen eines Elementes, abhängig von ihrer Position im XML-Dokument verschiedene Strukturen haben können.
- Elemente können durch eine komplexe Struktur aufgebaut sein, die strukturierte Gruppen, optionale oder mehrwertige Elemente enthalten kann.
- Elemente können gemischten Inhalt haben (*engl.* mixed content).
- Es gibt eine Menge weiterer Konzepte (z.B. Namensräume (*engl.* namespaces), die die Erstellung eines Modells verkomplizieren.

XER-Konstrukte

Das XER-Modell verwendet alle Basisbestandteile des ER-Modells und einige eigene neue Konstrukte. Sie wurden auf Grund der bereits beschriebenen Erkenntnisse über die Modellierung von XML eingeführt und werden im Folgenden beschrieben.

XER-Entität (*engl.* XER Entity) ist das grundlegende konzeptuelle Objekt in XER. Es wird durch ein Rechteck dargestellt. In einem Titelfeld steht der Namen und im Hauptteil stehen Attribute. Attribute sind Eigenschaften der Entität. Sie sind zumeist atomar und können auch optional oder mehrwertig sein. Sie sind standardmäßig geordnet, gemäß der Anordnung im Diagramm von oben nach unten. Bei mehrwertigen Attributen wird die Mehrwertigkeit durch Angabe des Wertes in Klammern angegeben. Abhängig vom Typ des zu modellierenden XML Elements werden geringe Unterschiede in der graphischen Repräsentation eingeführt. Es können die folgenden Typen auftreten:

Geordnete Entitäten sind die Standardannahme in XER-Diagrammen. In einer geordneten Entität treten die Attribute in der Ordnung auf, wie sie im Diagramm aufgeführt werden. XML-Attribute (umgewandelt durch die Umformung in ENF) sind durch das Präfix „@“ gekennzeichnet. Schlüsselattribute werden unterstrichen. Abbildung 4.13 zeigt eine geordnete Entität und den dazugehörenden XML-Schema-Code.

```
<xs:element name="hotel">
  <xs:attribute name="ID" type="xs:ID"/>
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Name"/>
      <xs:element name="Telefon"
                  minOccurs="0"
                  maxOccurs="unbounded/>
      <xs:element name="Bild"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

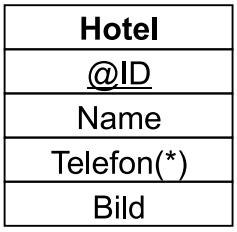


Abbildung 4.13: Geordnete XER Entität und zugehörendes XML Schema

Ungeordnete Entitäten bedeuten, dass alle Attribute zwar auftreten müssen, ihre Reihenfolge aber beliebig ist. Dieses Konzept entspricht dem `<all>`-Konzept in XML-Schema. In der graphischen Notation wird dem Titel ein Fragezeichen (`?`) vorangestellt (siehe Abbildung 4.14).

```
<xs:element name="Ausstattung">
  <xs:complexType>
    <xs:all>
      <xs:element name="Zimmerzahl"/>
      <xs:element name="Bar"/>
      <xs:element name="Schwimmbad"/>
      <xs:element name="Sauna"/>
    </xs:all>
  </xs:complexType>
```

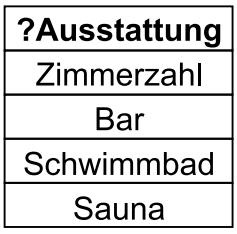


Abbildung 4.14: Ungeordnete XER Entität und zugehörendes XML Schema

Gemischte Entitäten lassen sowohl Text als auch andere Elemente als Inhalt zu. Graphisch werden sie durch abgerundete Ecken gekennzeichnet (siehe Abbildung 4.15).

XER-Beziehungen können zwischen zwei oder mehr Entitäten definiert werden. Sie werden verwendet, wenn eine Entität ein kom-

```
<xs:element name="Beschreibung"
            mixed="true">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Lage"/>
      <xs:element name="Ausstattung"/>
      <xs:element name="Aktivitäten"/>
    </xs:sequence>
  </xs:complexType>
```

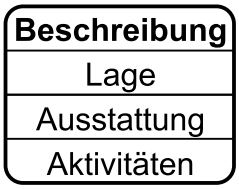


Abbildung 4.15: XER Entität mit gemischtem Inhalt u. XML-Schema

plexes Unterelement enthalten soll. Es werden `1:1`, `1:n` und `n:m`-Beziehungen unterstützt. Die Kardinalität der Relation entspricht `minOccurs` und `maxOccurs` in XML Schema. Im Diagramm wird eine Beziehung durch eine Raute dargestellt (siehe Abbildung 4.16). Beziehungen können benannt und unbenannt sein.

```
<xs:element name="hotel">
  <xs:attribute name="ID" type="xs:ID"/>
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Name"/>
      <xs:element name="Adresse">
        <xs:compleType>
          <xs:sequence>
            <xs:element name="Straße"/>
            <xs:element name="PLZ" />
            <xs:element name="Ort" />
          </sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

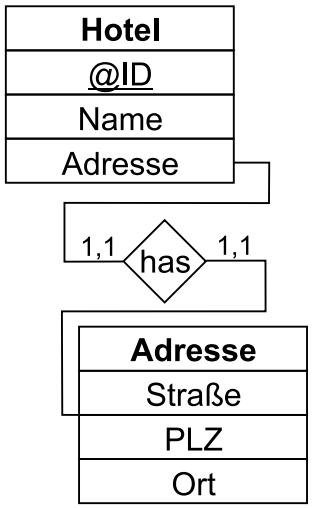


Abbildung 4.16: XER Beziehung und zugehörendes XML Schema

XER Generalisierungen treten auf, wenn eine Entität verschiedene Unter-Entitäten haben kann. Das entspricht der IST-Beziehung des ER-Modells. Dargestellt werden sie, indem die verschiedenen Unter-Entitäten innerhalb eines Rechtecks der Entität gezeichnet werden (siehe Abbildung 4.17).

```
<xs:element name="adresse">
  <xs:complexType>
    <xs:choice>
      <xs:element name="Hausanschrift"/>
        <xs:compleType>
          <xs:sequence>
            <xs:element name="Straße"/>
            <xs:element name="Nummer"/>
            <xs:element name="PLZ" />
            <xs:element name="Ort" />
          </sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Post"/>
        <xs:compleType>
          <xs:sequence>
            <xs:element name="Postfach"/>
            <xs:element name="PLZ" />
            <xs:element name="Ort" />
          </sequence>
        </xs:complexType>
      </xs:element>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

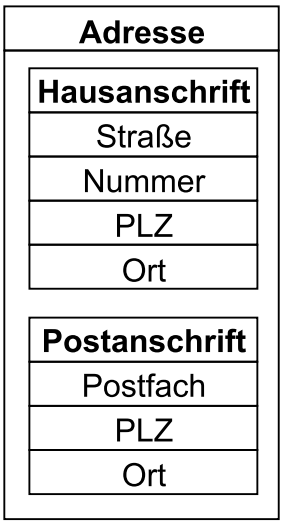


Abbildung 4.17: XER Generalisierung und zugehörendes XML Schema

Weitere Konzepte - wie *schwache Entitäten*, *mehrwertige Relationen* und *Aggregationen* werden durch die Autoren zwar benannt, aber nicht genauer erläutert.

Übersetzung

Die Autoren beschreiben in [ASD03] kurz folgende Übersetzungsprozesse:

- DTD → XER-Modell
- XML-Schema → XER-Modell
- XER-Modell → XML-Schema

Sie geben an, dass ein Prototyp in VBA (Visual Basic for Applications) als Erweiterung für Microsoft Visio existiert, der die Übersetzung von XML-Schema in ein XER-Modell und die Rückübersetzung vornimmt. Die Übersetzungsprozesse starten jeweils beim Wurzelelement (*engl.* root element) und gehen dann mittels Tiefensuche (*engl.* depth-first) vor, um die weiteren Elemente bzw. Entitäten zu erzeugen.

Bewertung

Das hier vorgestellte Modell orientiert sich an den Möglichkeiten von XML-Schema. Es ist leicht zu verstehen, wie bestimmte XML-Schema-Konstrukte in das XER-Modell übertragen werden können. Allerdings könnte die Abbildung komplexer Inhaltsstrukturen (geschachtelte `choice` und `sequence`-Konstrukte) schnell unübersichtlich werden, da alle Unter-Entitäten betitelt sein müssen. Es wäre zu untersuchen, ob die Einführung anonymer Entitäten zu mehr Übersichtlichkeit führen kann. Problematisch scheint mir auch die unterschiedliche Behandlung komplexer Unter-Entitäten. Während sie in geordneten Entitäten über Beziehungen „von außen“ eingebunden werden müssen, lassen sie sich bei der Generalisierung direkt innerhalb der Hauptentität darstellen. Da sich die beiden Konzepte auf die syntaktisch sehr ähnlichen XML-Schema-Konstrukte `choice` und `sequence` abbilden lassen, würde eine einheitlichere Behandlung zu mehr Übersichtlichkeit führen. Weiterhin werden zwar Schlüsselattribute explizit eingeführt; aus dem Artikel wurde aber nicht ersichtlich, ob sie in einem `ID-IDREF-` bzw XML-Schema-`key`-Konzept auch angewendet werden können.

4.3.5 EReX

Das EReX-Modell (ER extended for XML) von Mani wird in [Man04] eingeführt. Die Erweiterungen des EReX-Modells gegenüber dem klassischen ER-Modell werden im Folgenden vorgestellt:

Kategorien. (*engl.* kategories) sind ein spezieller Beziehungstyp, welcher dem IST-Beziehungstyp ähnelt. Die Entitätstypen $E_1, \ldots, E_n$ sind Kategorien des Entitätstyps E, wenn für jede Datenbankinstanz I gilt $I(E_i) \subseteq I(E)$, für alle $i = 1 \ldots n$. Dieser Sachverhalt wird im EReX Schema durch Pfeile von den E_i's zu den Es dargestellt. Das Konzept unterscheidet sich von der IST-Beziehung, bei welcher ein Schlüssel für E definiert werden muss.

Überdeckungsbeschränkungen. (*engl.* coverage constraints) können über Entitätstypen oder über Rollen spezifiziert werden. Man unterscheidet *totale* und *exklusive* Überdeckung. Die Beschränkungen gelten für die Unter-Entitätstypen in Kategoriebeziehungen. *Totale Überdeckung bedeutet*, dass der Obertyp die Vereinigung aller Untertypen ist. *Exklusive Überdeckung* heißt, dass die Mengen aller Untertypen disjunkt sind.

Ordnungsbeschränkungen. (*engl.* order constraints) werden für Entitätstypen spezifiziert, die in einen Beziehungstyp eingehen. Sie besagen, dass die Elemente der Menge des Entitätstyps gemäß ihrer inneren Ordnung in die Beziehung eingehen. Graphisch wird der Sachverhalt durch eine dicke Verbindungslinie verdeutlicht.

Übersetzung

Im Gegensatz zu den bisher vorgestellten Modellen ist die Zielsprache weder DTD noch XML-Schema sondern XGrammar. XGrammar ist eine verbreitete Notationsform in der Theorie Formaler Sprachen. Sie basiert auf einer regulären Baumgrammatik, die um die Fähigkeit zur Darstellung von Attributen und von Datentypen erweitert wurde. XGrammar wird in [MLM01] formal eingeführt.

Der Übersetzungsalgorithmus wird detailiert an einem Beispiel in [Man04] vorgestellt. Der Schwerpunkt liegt dabei auf der sinnvollen Anwendung der Überdeckungsbeschränkungen.

Bewertung

Der vorgestellte Algorithmus überführt das EReX-Modell in die sehr formale X-Grammar. Das EReX-Modell ist gut geeignet, um die Struktur der XML Dokumente zu beschreiben. Es werden das ID-IDREF Konzept, Vereinigung und Rekursion direkt unterstützt. Es fehlen allerdings Konzepte für die Spezifikation einfacher Datentypen oder für die exakte Angabe der Kardinalitäten (mittels `minOccurs` und `maxOccurs`), wie sie von XML-Schema zur Verfügung gestellt werden. Weiterhin funktioniert der Algorithmus nur in eine Richtung. Die Rückübersetzung von X-Grammar in EReX scheint nach Aussagen des Autors noch nicht gelöst zu sein.

4.3.6 Zusammenfassende Bewertung für ER-Modell-basierte Ansätze

Der große Vorteil der Verwendung des ER-Modells für die Erstellung von XML-Inhaltsmodellen ist seine große Verbreitung. Besonders im Datenbankenbereich dürfte das ER-Modell den meisten Anwendern bekannt sein. Das ER-Diagramm benutzt einfache Konstrukte, wie Rechtecke und Rauten, um Datenobjekte und Beziehungen zwischen ihnen zu beschreiben. Der Modellierung von Entitäten und Beziehungen steht die hierarchische Struktur von XML-Dokumenten teilweise entgegen. SENGUPTA, MOHAN und DOSHI geben in [ASD03] einen kurzen Überblick über weitere Probleme:

- XML-Objekte besitzen eine natürliche Ordnung. Sie ist sowohl zwischen verschiedenen Elementen als auch mehreren Instanzen desselben Elements definiert.
- Auf Grund der hierarchischen Struktur von XML gibt es keine Möglichkeit n:m-Beziehungen direkt umzusetzen.
- Typen sind heterogen. Das heißt, dass Elemente mit gleichem Namen verschiedene Typen haben können, abhängig von ihrer Position im XML-Dokument.

- Die Elementstruktur kann komplex (mit strukturierten Gruppen von Elementen), optional, notwendig (*engl.* required) oder mehrwertig (*engl.* multi-valued) sein.
- XML Elemente können gemischten Inhalt (*engl.* mixed content) enthalten.
- Es gibt eine Vielzahl weiterer Konzepte (wie z.B. Namensräume), die die Entwicklung eines konzeptuellen Modells erschweren.

Die vorgestellten Ansätze versuchen, einige der oben genannten Probleme durch verschiedene Erweiterungen zu beheben. Dabei gelingt es jedoch keinem Ansatz alle Punkte ausreichend abzudecken.

Ein wesentlicher Bestandteil von XML Schema ist die Definition benutzerdefinierter Typen. Ausgehend von einer umfangreichen Bibliothek von Basistypen lassen sich neue Typen durch Angaben von Beschränkungen oder Vereinigungen definieren. Dieser Aspekt wird durch die vorgestellten Ansätze nicht näher betrachtet. Das liegt in der Tatsache begründet, dass sie meist nur DTDs als Zielsprache verwenden.

Spezialisierung und Generalisierung werden in XML-Schema direkt unterstützt und ließen sich so leichter übersetzen.

Die meisten Ansätzen führen bei der Überführung einer DTD / XML-Schemas in das konzeptuelle Modell eine Vorübersetzung durch, in der die DTD / das XML-Schema vereinfacht wird. So werden zum Beispiel XML-Entitäten aufgelöst oder durch die Erzeugung der ENF (Element Normal Form) XML-Attribute in Elemente umgewandelt. Diese Vorgehensweise führt zu einer Vereinfachung des Imports der DTD / des XML-Schemas.

4.3.7 Einführung in UML

Die Modellierungssprache UML (Unified Modeling Language) findet ihre Hauptanwendung in der Softwaretechnik. Sie unterstützt die Spezifizierung, Visualisierung und Dokumentation von Softwaresystemen. UML eignet sich auch zur Erstellung von Geschäftsmodellen (*engl.* business modeling) und zur Modellierung von Systemen, die nicht auf Software basieren. Die Spezifikationen in UML sind unabhängig von Hardware, Netzwerksystemen, Betriebssystemen oder Programmiersprachen. Da UML auf objektorientierten Konzepten, wie Klassen und Operationen aufbaut, eignet sie sich besonders gut für die Modellierung von objektorientierten Systemen. Aber auch nicht-objektorientierte Systeme lassen sich mittels UML modellieren. Der UML-Standard [UML05] wird von der OMG (Object Management Group) betreut und weiterentwickelt. Der Vorteil der Standardisierung liegt vor allem in der Möglichkeit, die Modelle zwischen verschiedenen Anwendungen austauschen zu können.

Wie lässt sich mit UML modellieren?

UML 2.0 definiert 13 verschiedene Diagrammtypen, die sich in drei verschiedene Kategorien aufteilen lassen:

Strukturdiagramme mit Klassendiagramm, Objektdiagramm, Komponentendiagramm, Kompositionsstrukturdiagramm, Verteilungsdiagramm, Paketdiagramm

Verhaltensdiagramme mit Anwendungsfalldiagramm, Aktivitätsdiagramm, und Zustandsdiagramm

Interaktionsdiagramme (abgeleitet von den Verhaltensdiagrammen) mit Kommunikations-, Sequenz-, Zeitdiagramm und Interaktionsübersicht.

Eine Reihe von Tools unterstützen den Anwender bei der Erstellung dieser Diagramme. Neben Mitteln zur graphischen Erzeugung enthalten viele Tools Funktionen, die es ermöglichen aus den Diagrammen Quellcode abzuleiten (Codeerzeugung) oder aus vorhandenem Quellcode die entsprechenden Diagramme zu erzeugen (Reverse Engineering). Die Kombination der beiden Verfahren ermöglicht es dem Programmierer, sowohl am Modell als auch am Quellcode zu arbeiten. Modell und Quellcode bleiben dabei synchron. Man spricht hier auch vom Round-Trip-Engineering.

Für XML-Inhaltsmodelle wird in der Regel nur ein bestimmter Diagrammtyp - das *Klassendiagramm* - verwendet. Konzepte für Anpassungen oder Erweiterungen sind in den UML-Standard bereits integriert.

Das UML-Klassendiagramm

Im Folgenden sollen die Konzepte des Klassendiagramms vorgestellt werden. Die Ausführungen basieren auf [Alh98]. Der Schwerpunkt liegt dabei auf Konzepten, die in den anschließend vorgestellten Konzepten zur Modellierung von XML-Inhaltsmodellen angewendet werden. Für eine umfassende Beschreibung sei auf die zahlreich verfügbare Literatur (z.B. [Alh98]) verwiesen.

Klassendiagramme stellen die statische Struktur eines Systems dar. Sie modellieren Klassen und Beziehungen (*engl.* associations).

Klassen

- werden als umrandete Rechtecke mit Abschnitten (*engl.* compartments) dargestellt.
- enthalten einen Namensabschnitt, der den Klassennamen enthält. Klassennamen sind zentriert und fettgedruckt und beginnen mit großem Buchstaben. Bei abstrakten Klassen wird der Name kursiv dargestellt.
- können einen Abschnitt mit Attributen enthalten.
- können einen Abschnitt mit Methoden enthalten.
- können noch andere Abschnitte enthalten, die für die Modellierung weiterer Eigenschaften verwendet werden können.
- können Schnittstellen (*engl.* interfaces) enthalten, die zusichern, dass eine Klasse die definierten Methoden enthält.
- können mit anderen Klassen in Generalisierungsbeziehung stehen.

Attribute

- werden als Texte im Attributabschnitt einer Klasse dargestellt.
- werden meist linksbündig ausgerichtet, haben normale Schrift und beginnen mit kleinem Buchstaben.
- müssen einen (innerhalb der Klasse) eindeutigen Namen besitzen.
- können einen Multiplizitätsausdruck in eckigen Klammern enthalten. Er drückt aus, wieviele Instanzen von dem Attribut innerhalb einer Klasseninstanz zugelassen sind. Attribute ohne Multiplizitätsausdruck kommen genau einmal vor.
- können einen Typausdruck enthalten, der (durch Doppelpunkt separiert) den Typ für die Werte des Attributs festlegt.

- können, durch Gleichheitszeichen angegeben, einen Defaultwert enthalten.

Datentypen

- werden als Zeichenketten dargestellt.
- haben eindeutige Werte ohne Identität. Das bedeutet, dass verschiedene Repräsentationen desselben Wertes sich nicht unterscheiden lassen. Zum Beispiel ist der Wert `1` einer Variable `x` nicht vom Wert `1` einer Variable `y` zu unterscheiden.
- können auch Operationen besitzen.

Methoden

- werden als Texte im Methodenteil dargestellt.
- werden meist linksbündig ausgerichtet, haben normale Schrift und beginnen mit kleinem Buchstaben. Abstrakte Methoden werden kursiv dargestellt.
- müssen einen Namen besitzen.
- können (in Klammern angegeben) eine komma-getrennte Liste von Parametern enthalten.
- können einen Rückgabetyp spezifizieren, der nach einem Doppelpunkt steht.
- Die Definition des Verhalten (Implementierung) der Methode kann innerhalb eines Kommentars als Text oder durch Verhaltensdiagramme, die der Methode zugeordnet werden, erfolgen.

Beziehungen (*engl.* associations)

- werden als Linien oder Pfade zwischen zwei oder mehr Klassen gezeichnet.
- können reflexiv (oder rekursiv) sein. Das heißt, dass eine Klasse mit sich selbst verbunden werden kann.
- können einen Bezeichner haben. Dem Namen kann ein Dreieckspfeil angehängt werden, welcher die Richtung anzeigt, in die der Name gelten soll.
- können einen Rollennamen enthalten.

- können einen Multiplizitätsausdruck enthalten, der anzeigt wie viele Instanzen der Klasse in die Verbindung eingehen.
- können einen Pfeil enthalten, der die Richtung der Navigierbarkeit anzeigt.

Aggregationen

- sind eine spezielle Art von Beziehung.
- enthalten einen Aggregationsindikator (Symbol: Raute).
- Schwache Aggregation (Symbol: ungefüllte Raute) liegt vor, wenn die zugehörenden Klassenobjekte selbständig existieren können oder mehrere Besitzer haben und diese auch wechseln können.
- Starke Aggregation, auch Komposition genannt, (Symbol: gefüllte Raute) bedeutet, dass die Klassenobjekte nur einen Besitzer haben und nicht selbstständig existieren können.

Generalisierungen

- sind Beziehungen zwischen allgemeineren und spezielleren Klassen.
- Die spezielleren Klassen sind zu den allgemeineren kompatibel. Sie erben deren Eigenschaften (Attribute, Methoden und Beziehungen) und können diese erweitern. Spezielle Klassen können geerbte Methoden überschreiben und sind durch die allgemeineren Klassen ersetzbar.
- werden mit voller Linie und großem hohlen Dreieck gezeichnet, welches auf das allgemeinere Element zeigt.
- Generalisierungsbeziehungen dürfen nicht zirkulär sein.

Kompositionen (*engl.* compositions)

- werden verwendet, um Modellelemente darzustellen, die konzeptuell fest mit dem ihnen zugeordneten Besitzer (Container) verbunden sind.
- Kompositionsklassen werden innerhalb eines Abschnittes des Containers dargestellt.
- Kompositionsbeziehungen können auch zwischen Klassen innerhalb eines Containers bestehen.
- Beziehungen zu Modellelementen außerhalb des Containers gelten als selbstständig und werden nicht mehr dem Container zugeordnet.

Packages

- gehören zu den allgemeinen Modellierungskonzepten von UML; das heißt, sie lassen sich in mehreren Diagrammtypen anwenden.
- sind ein Mechanismus, um semantisch verwandte Modellelemente zu gruppieren.
- werden durch zwei Rechtecke (Folder-Symbol) dargestellt. Das kleinere Rechteck wird gewöhnlich oben links an das große Rechteck angehängt.
- können neben Modellelementen auch weitere Packages enthalten.
- sind Eigentümer ihrer Komponenten. Jedes Modellelement kann nur zu einem Package gehören.
- können ihren Inhalt visuell verbergen.
- können selbst mit anderen Packages in Beziehung stehen. Dadurch wird angezeigt, dass zwischen einigen Elementen der beiden Packages Beziehungen bestehen. Generalisierungsbeziehungen sind möglich.

Stereotypen

- sind einer der Erweiterungsmechanismen von UML.
- dienen der Klassifizierung oder Kennzeichnung von Modellelementen und führen neue Typen von Modellelementen ein.
- werden als Zeichenketten in doppelten spitzen Klammern (`<<...>>`) dem Elementnamen vorangestellt.
- werden verwendet, um neue Modellelemente einzuführen. Die neuen Modellelemente haben folgende Eigenschaften:
  - sind Unterklassen existierender Modellelemente.
  - müssen dieselbe Form und Eigenschaften (Attribute und Operationen) wie ihre Basis-Modellelemente haben. Das heißt, sie erben die Eigenschaften ihrer Basis-Modellelemente durch Generalisierung.
  - müssen einen anderen Zweck oder Semantik wie ihre Basis-Modellelemente haben.
  - können die Basis-Modellelemente um neue Eigenschaften erweitern.
  - können ein eigenes Icon oder graphische Markierung besitzen.
  - lassen sich in einer Hierarchie beschreiben, die die Metamodell-Hierarchie erweitert.

Properties

- charakterisieren ein Element.
- sind Strings, bestehend aus einer komma-separierten Liste von Substrings, die in geschweiften Klammern ({...}) eingeschlossen sind.
- Die Listenbestandteile können Attribute, Assoziationen, Beschränkungen oder strukturierte Werte sein.
- werden einem Modellelement zugeordnet, indem sie
  - dem Elementnamen folgen
  - in einer Notiz stehen, die (durch eine gestrichelte Linie) mit einem anderen Symbol verbunden wurde. Wird die Notiz mit dem Stereotyp (`<<properties>>`) gekennzeichnet, könnnen die geschweiften Klammern in den Substrings entfallen.
- lassen sich auch mehreren Modellelementen zuordnen.

Constraints (Beschränkungen)

- definieren semantische Regeln oder Bedingungen, die ein bestimmtes Element oder Modell erfüllen muss.
- sind Bestandteile der Property-Liste.
- werden in einer bestimmten Sprache (Programmiersprache, natürliche Sprache, OCL = Object Constraint Language) beschrieben, die auf konzeptueller Ebene interpretiert oder ausgeführt werden kann.

4.3.8 UML-Modell für DTDs nach Conrad, Scheffner und Freytag

Conrad, Scheffner und Freytag beschreiben in [CSF00] wie man ein DTD-basiertes XML-Modell, mittels UML-Konstrukten darstellen kann. Ihr Ansatz soll an dieser Stelle vorgestellt werden, da das erzeugte UML-Diagramm relativ einfach zu verstehen ist.

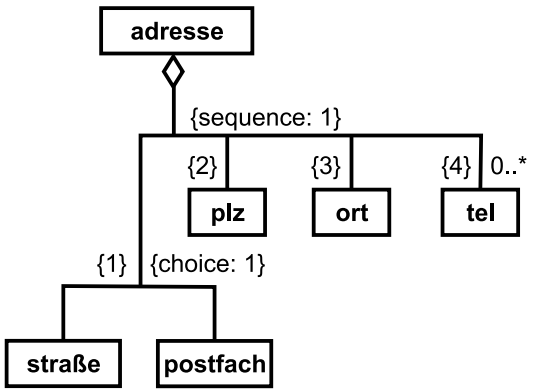
Die Autoren verwenden das UML-Klassendiagramm, welches durch Stereotypen erweitert wird. Das Klassensymbol repräsentiert Elemente. Der Klassenname entspricht dem Namen des Elementes. Für die Darstellung der DTD-Inhaltsmodelle werden zwei Notationen vorgeschlagen:

1. **Attribute.** Werden Attribute verwendet, entsprechen die UML-Attributnamen den XML-Elementnamen. Die Reihenfolge der Attribute ist in UML nicht explizit festgelegt. Zur Erzeugung von XML-Inhaltsmodellen ist sie jedoch zwingend notwendig. Sie wird einfach als „Top-Down“

festgelegt. UML-Datentypen sowie Einschränkungen der Gültigkeit (z.B. `public` oder `private` werden nicht verwendet. Der Typ des XML-Elements wird aus dem Namen abgeleitet, d.h. es existiert eine Klasse mit dem gleichen Namen, die den Typ beschreibt. Fehlt eine solche Klasse wird als Elementtyp automatisch `#PCDATA` angenommen. Die DTD-Multiplizitäten (`?`, `*`, `+`) werden durch Kardinalitäten ([`0..1`], [0..∗], [`1`..∗]) abgebildet. (siehe Abbildung 4.20)

2. Aggregation. Die Aggregationsbeziehung bietet sich als Mittel zur Darstellung von Elementinhalten an. Es wird explizit festgelegt, dass die Reihenfolge der UML-Subklassen in der graphischen Darstellung von links nach rechts, der Reihenfolge der Sub-Elemente im XML-Inhaltsmodell entspricht. Zusätzlich kann die Ordnung auch durch Kommentare definiert werden Dazu werden die Subelemente mit {`1`}, {`2`}, ... gekennzeichnet. Kardinalitäten können an den Verbindungslinien notiert werden. Sequenzen und Alternativen werden durch die Kommentare {`sequence`} und {`choice`} gekennzeichnet. Zusätzlich kann in dem Kommentar noch eine Kardinalität notiert werden, wie z.B. {`sequence:1`}.

Die Schachtelung von Alternativen und Sequenzen wird durch „Aufspalten“ der Verbindungslinien modelliert (siehe Abbildung 4.18). Dieses Darstellung ist aber nicht UML-konform. Zur Herstellung von UML-Konformität könnten zusätzliche „Klassen“-Objekte mit einem neuen Stereotyp (z.B. `contentmodel`) für Sequenzen und Alternativen eingeführt werden.



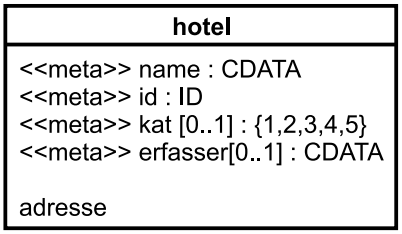

```
<!ELEMENT adresse
  ((straße|postfach),
    plz, ort, tel*)>
<!ELEMENT straße (#PCDATA)>
<!ELEMENT postfach (#PCDATA)>
<!ELEMENT plz (#PCDATA)>
<!ELEMENT ort (#PCDATA)>
<!ELEMENT tel (#PCDATA)>
```

Abbildung 4.18: Beispiel für Inhaltsmodell über Aggregationen

In DTDs wird *Mixed Content* durch ein zusätzliches `#PCDATA`-Element innerhalb einer Choice-Gruppe mit der Kardinalität `*` modelliert. Im UML-

Diagramm wird eine Klasse `#PCDATA` mit dem Stereotyp `<<content>>` eingeführt, welche analog zum DTD-Modell ins Diagramm eingebunden wird.

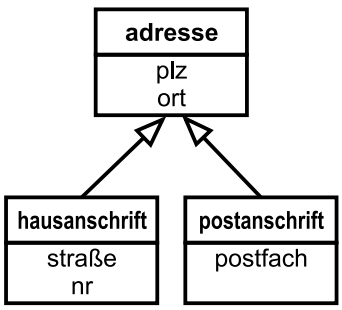
XML-Attribute werden durch UML-Metaattribute dargestellt. Metaattribute innerhalb eines UML-Klassendiagramms erkennt man am Stereotyp `<<meta>>`. Metaattribute erhalten einen Typ, der einem der möglichen DTD Typen (`CDATA`, `ID`, ...) entspricht. Auch Aufzählungen (*engl.* enumerations) können als Typ angegeben werden. Optionalität der Attribute wird durch die Kardinalität [`0..1`] angegeben. Fehlt die Kardinalität, wird angenommen, dass der Attributwert notwendig (`#REQUIRED`) ist. Default-Werte werden sowohl von DTDs als auch von UML unterstützt, so kann dieses Konstrukt 1:1 abgebildet werden. Abbildung 4.19 zeigt ein Beispiel-UML-Diagramm und die dazu gehörende DTD.




```
<!ELEMENT hotel(adresse)>
<!ELEMENT adresse (#PCDATA)>
<!ATTLIST hotel
    name CDATA #REQUIRED
    id   ID    #REQUIRED
    kat (1|2|3|4|5)
    erfasser CDATA #IMPLIED>
```

Abbildung 4.19: Beispiel für Attribute und Inhaltsmodell mit UML-Attributen

UML-Generalisierungsbeziehungen sollen auch als Generalisierung in XML gedeutet werden. Da DTDs dieses Konzept nicht unterstützen, wird das Konstrukt durch automatisch eingeführte ID- und IDREF-Attribute simuliert (siehe Abbildung 4.20).




```
<!ELEMENT adresse (plz, ort)>
<!ATTLIST adresse
          id ID #REQUIRED>

<!ELEMENT hausanschrift
     (straße, nr)
<!ATTLIST hausanschrift
          idref IDREF #REQUIRED>

<!ELEMENT postanschrift
     (postfach)
<!ATTLIST postanschrift
          idref IDREF #REQUIRED>
```

Abbildung 4.20: Beispiel für Generalisierung

UML-Beziehungen lassen sich zur Modellierung von Links (siehe XLink [XLI01]) zwischen Elementen verwenden. Mittels UML-Packages lassen sich komplexe XML-Entities darstellen. Komplexe Entities enthalten immer komplette Markup-Beschreibungen, die ihrerseits wieder Entities enthalten können. Sie werden in einzelne DTD-Dateien aufgelöst.

Die Elementinhalte `ANY` und `#PCDATA` werden als einfache Klassensymbole mit dem Stereotyp `<<content>>` umgesetzt. `EMPTY` braucht nicht explizit dargestellt werden, es steht dem Nutzer jedoch frei, einen UML-Kommentar zu verwenden.

Einfache Entitäten werden als Klassensymbole mit dem Stereotyp `<<entity>>` dargestellt. Lokale Entitäten werden durch ein Präfix (`%`) gekennzeichnet, um sie von globalen Entitäten zu unterscheiden.

DTD-Notations lassen sich durch Klassen mit dem Stereotyp `<<notation>>` umsetzen.

4.3.9 UML-Profil für XML Schema

Der Ansatz [BKK04b] von BERNAUER, KAPPEL und KRAMLER wurde ausgewählt, da die Autoren den Anspruch erheben, eine vollständige bidirektionale Abbildung von XML-Schemas auf UML zu definieren. Dadurch soll es möglich sein XML-Schemas in UML-Diagramme und wieder zurück umzuwandeln, ohne dabei Informationen zu verlieren. Dabei verfolgen die Autoren die folgenden drei Ziele:

1. Jedes beliebige XML-Schema muss in UML repräsentierbar sein. Daraus folgt, dass für alle wichtigen XML-Schema Konzepte entsprechende Notationen in UML gefunden werden müssen.

2. Die Repräsentation des XML-Schemas muss, auch wenn man die im Profil definierten Stereotypen weglässt, lesbar sein. Dadurch soll gewährleistet werden, dass Anwender mit geringer Kenntnis von XML-Schema die Modelle lesen können, und Anwendungen, die keine Profile auswerten können, die Modelle trotzdem verarbeiten können. Diese Anforderung ist konform mit der Eigenschaft der UML-Stereotypen, die zwar UML Konzepte erweitern aber nicht grundsätzlich ändern können.

3. Die Anzahl der benötigten UML Konzepte sollte möglichst gering sein, um hohe Lesbarkeit zu gewährleisten.

Im Folgenden werden die UML-Notation der wichtigsten XML-Schema Konzepte vorgestellt. Die graphischen Darstellungen setzen Teile des Beispielschemas der XML-Schema-Spezifikation [XS001] um.

Schemadokument

Schemadokumente und ihre Abhängigkeiten (mittels `import`, `include` und `redefine`) werden durch die folgenden Regeln abgebildet:

`SC:` Für jedes Schemadokument wird ein Package mit dem Stereotyp `<<xml-schema>>` eingeführt. Die Eigenschaften des Stereotyps enthalten den Target-Namespace, Location und Version (siehe Abbildung 4.21). Schemaabhängigkeiten werden als UML-Abhängigkeitsrelationen mit den Stereotypen `<<xs-import>>` `<<xs-include>>` und `<<xs-redefine>>` dargestellt. Für UML 2.0 wird vorgeschlagen die vordefinierten Abhängigkeiten `Permission` und `PackageImport` verwenden.

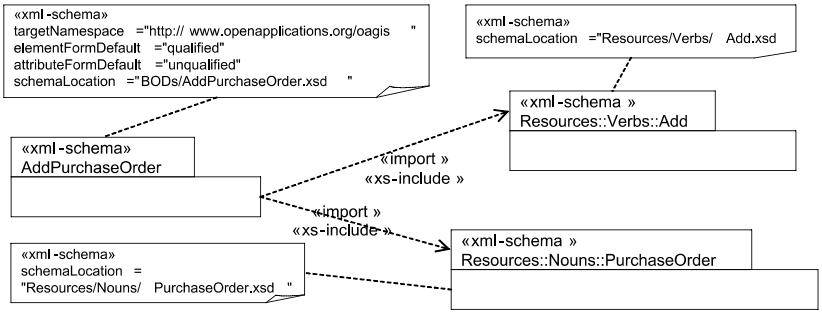


Abbildung 4.21: Darstellung des `AddPurchaseOrder`-Schemas und seiner Abhängigkeiten (aus [BKK04b])

Definition eines komplexen Datentyps

`CT1:` Jede globale Definition komplexer Datentypen wird als Klasse mit dem Namen des Typs und dem Stereotyp `<<complexType>>` umgesetzt (siehe Abbildung 4.22).

`CT2:` Jede lokale Definition eines komplexer Datentyps wird als Klasse mit dem Stereotyp `<<complexType>>` definiert. Der Name der Klasse entspricht dem Namen des Elementes, dem die lokale Typdefinition zugeordnet wird. Die UML-Klasse wird innerhalb der UML-Klasse des enthaltenden Elements geschachtelt. Ein alternativer Ansatz verzichtet auf Schachtelung, in dem die Hierarchie über den Klassennamen kodiert wird. Die diversen Eigenschaften komplexer Datentypdefinition werden wie folgt abgebildet:

- Das Inhaltsmodell wird entweder als einfacher Inhalt oder durch Modellgruppen dargestellt. Beide Konzepte werden an anderer Stelle noch genauer vorgestellt. Gemischter Inhalt (*engl.* mixed content) wird durch

ein gleichlautende Eigenschaft (*engl.* property) mit dem Stereotyp `<<complexType>>` dargestellt.

- Ein abstrakter komplexer Datentyp wird auf eine abstrakte komplexe Klasse abgebildet.
- Ableitung durch Erweiterung (*engl.* extension) und Beschränkung (*engl.* restriction) werden als UML-Generalisierungen mit den Stereotypen `<<extension>>` und `<<restriction>>` dargestellt.
- Randbedingungen für Ableitungen und Ersetzungen werden durch UML-Constraints angegeben.
- Einfacher Inhalt (*engl.* simple content) wird durch ein Attribut mit dem Stereotyp `<<content>>` abgebildet. Es erhält den Namen `value` und den entsprechenden Datentyp.

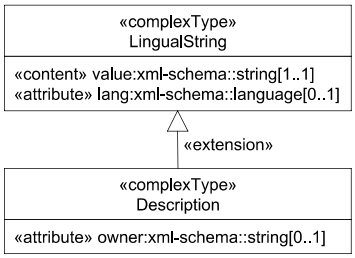


Abbildung 4.22: Darstellung eines komplexen Typs (aus [BKK04b])

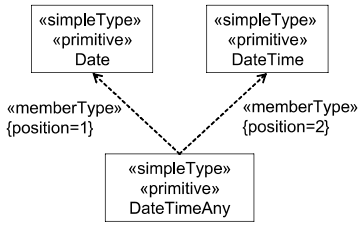


Abbildung 4.23: Bildung eines einfache Datentyps mittels `union` (aus [BKK04b])

Definition eines einfachen Datentyps

Generell können die einfachen Datentypen des XML-Schema Standards als UML-Datentypen umgesetzt werden. Da UML zwischen Aufzählungen und anderen Arten von Datentypen unterscheidet, werden die folgenden Regeln eingeführt:

`ST1:` Jeder einfache Datentyp, der eine Aufzählungsbeschränkung (*engl.* enumeration facet) enthält, wird als UML-Enumeration mit dem Stereotyp `<<simpleType>>` umgesetzt (siehe Abbildung 4.24).

`ST2:` Jeder einfache Datentyp ohne Aufzählungsbeschränkung wird als UML-Datentyp mit dem Stereotyp `<<simpleType>>` umgesetzt. Andere Ansätze verwenden an dieser Stelle UML-Klassen mit Stereotypen.

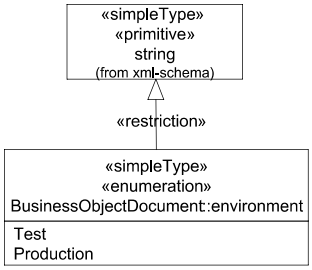


Abbildung 4.24: Darstellung eines lokalen einfachen Typs (aus [BKK04b])

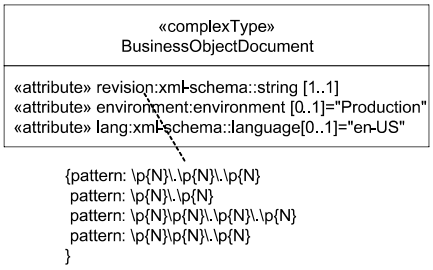


Abbildung 4.25: Darstellung einer Restriction eines einfachen Typs `union` (aus [BKK04b])

Lokale einfache Datentypen werden wie lokale komplexe Datentypen innerhalb der sie enthaltenden Klasse geschachtelt. Der Name des Types wird aus dem Namen des ihn enthaltenden Elements abgeleitet.

Alle weiteren Beschränkungen (*engl.* XML facets) werden als UML-Beschränkungen (in geschweiften Klammern) angegeben. Da XML-Schema nicht zwischen den verschiedenen Facets unterscheidet, stellt sich hier die Frage, ob es nicht besser wäre, zu Gunsten der Übersichtlichkeit auf das Konzept der UML-Enumeration zu verzichten und Aufzählungsbeschränkungen ebenfalls als „gewöhnliche“ UML-Constraints umzusetzen. Es ist auch möglich, Beschränkungen als Eigenschaften der Stereotypen umzusetzen.

Ableitung durch Beschränkung einfacher Datentypen wird durch eine UML-Generalisierung mit dem Stereotyp `restriction` umgesetzt. Bei einer Ableitung von UML-Enumerations lassen sich lediglich Elemente hinzufügen. Das steht im Widerspruch zu XML-Schema. Um UML-Konformität zu erreichen, müsste eine Abhängigkeit mit Stereotyp verwendet werden; dadurch wäre die Lesbarkeit eingeschränkt.

Listen (*engl.* list) und Vereinigungen (*engl.* union) werden als Abhängigkeiten mit den Stereotypen `<<itemType>>` und `<<memberType>>` dargestellt (siehe Abbildung 4.23).

An einigen Stellen ist es nicht notwendig, lokale einfache Datentypen als UML-Datentyp zu modellieren; so kann beispielsweise der Typ eines Attributes direkt als Typ des UML-Attribut angegeben werden (siehe Abbildung 4.25).

Für eine genaue Beschreibung der Regeln `ST3` und `ST4`, die sich mit dieser und weiterer Vereinfachungsmöglichkeiten befassen, sei auf [BKK04b] verwiesen.

Definition von Elementen

In XML-Schema lassen sich Definition und Verwendung eines Elementes trennen. Elemente werden global definiert und lassen sich mittels `ref` referenzieren. Dieses Konzept wird durch UML nicht unterstützt. Deshalb werden lokale und globale Elemente unterschiedlich behandelt. Lokale Elemente (Definition und gleichzeitige Verwendung) werden wie folgt abgebildet:

`EL1:` Lokale Elemente mit komplexem Typ werden als Rolle einer Assoziation mit dem Stereotyp `<<element>>` dargestellt. Die Assoziation besteht zwischen der Klasse, die das Element enthaltende Modellgruppe repräsentiert, und der Klasse, die den Typ des Elements darstellt. Zusätzlich wird noch die entsprechende Multiplizität angegeben(siehe `DataArea` in Abbildung 4.26).

`EL2:` Lokale Elemente mit einfachem Typ werden als Attribute mit dem stereotyp `<<element>>` in der Klasse, die die enthaltende Modellgruppe repräsentiert, dargestellt. Elementtyp und die Multiplizität können angegeben werden. Zusätzlich lässt sich bei dieser Art der Repräsentation auch ein Default-Wert angeben.

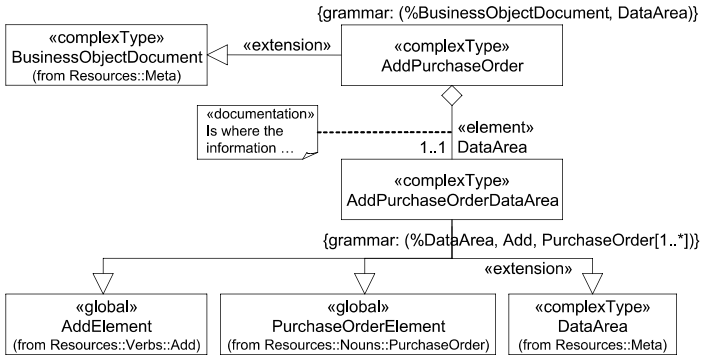


Abbildung 4.26: Deklaration komplexer Datentypen (aus [BKK04b])

`EG:` Globale Elemente werden genauso wie lokale Elemente dargestellt - als Rolle mit Stereotyp oder als Attribut. Da globale Elemente außerhalb einer Modellgruppe definiert werden, muss eine Klasse mit dem Stereotyp `<<global>>` eingeführt werden. Die Klasse erhält den Namen des Elements mit dem Suffix `Element`. Die Verwendung des Elementes wird durch eine Generalisierung-Beziehung abgebildet. Eventuell müssen die Attribute in der benutzenden Klasse angepasst bzw. die geerbten Assoziationen spezialisiert werden, um geänderte Multiplizitäten spezifizieren zu können (siehe Abbildung 4.27).

`EW:` Zur Darstellung von Elementen mit beliebigem Inhalt (Any-Element) wird eine Beschränkung {`multiple-classification`} eingeführt. Diese sagt aus, dass die Instanz der Klasse durch eine Instanz einer beliebigen anderen Klasse, die ein globales Element darstellt, ersetzt werden kann (siehe Abbildung 4.28).

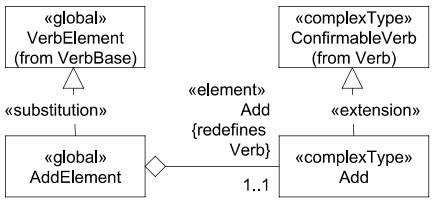


Abbildung 4.27: Deklaration des globalen Elements `Add` (aus [BKK04b])

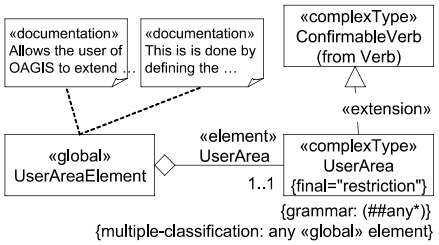


Abbildung 4.28: Deklaration einer globalen Elements und komplexen Typs (aus [BKK04b])

Definition von Attributen

Auch bei Attributen wird zwischen lokalen und globalen Attributen unterschieden. Lokale Attribute lassen sich unter Berücksichtigung einiger Besonderheiten 1:1 auf UML-Attribute abbilden.

`AL:` Jedes lokale Attribut wird als UML-Attribut mit dem Stereotyp `<<attribute>>` in der UML-Klasse, welche den komplexen Typ oder die Gruppe beschreibt, hinzugefügt. Die Angabe der Multiplizität als `[0..1]` oder `[1..1]` und eines Standard-Wertes ist möglich. Attribute mit festem (*engl.* fixed) Wert lassen sich durch die entsprechende UML-Constraint {`read only`} darstellen.

`AG:` Globale Attribute werden wie lokale Attribute dargestellt. Sie werden einer neuen Klasse mit dem Stereotyp `<<global>>` mit der Multiplizität `0..1` hinzugefügt. Der Name der Klasse wird aus dem Atttributnamen und dem Suffix `Attribute` gebildet. Die Verwendung der Attribute erfolgt über die Generalisierungsbeziehung. In der benutzenden Klasse kann das Attribut neu definiert werden, um eventuell veränderte Multiplizität oder einen Standardwert zu modellieren.

`AW:` Attribute mit beliebigem Inhalt werden wie Elemente mit beliebigem Inhalt behandelt (siehe `EW`).

Inhaltsmodelle

In UML gibt es kein Konzept für Modellgruppen (`sequence`, `choice`, `all`). Deshalb werden die folgenden zwei Konzepte vorgeschlagen:

`MG1:` Jede Modellgruppe wird durch eine Klasse mit dem entsprechenden Stereotyp (`<<sequence>>`, `<<choice>>`, `<<all>>`) dargestellt. Der Namen der Klasse wird gebildet aus dem Präfix „`mg`“ und einer hierarchischen Nummer. Geschachtelte Modellgruppen lassen sich durch Kompositions-Assoziationen und entsprechende Multiplizität darstellen. Da Modellgruppen privat sind, werden sie in der Klasse, die den komplexen Datentyp repräsentiert, geschachtelt. Als Vereinfachung kann die „äußerste“ Modellgruppe auch durch die Klasse repräsentiert werden, die den dazugehörenden komplexen Datentyp darstellt (siehe Abbildung 4.29).

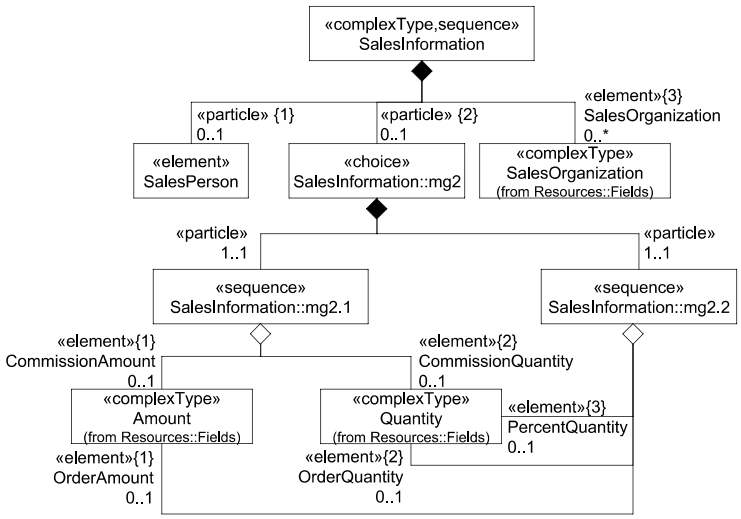


Abbildung 4.29: Komplexes Inhaltsmodell nach `MG1` (aus [BKK04b])

Da es häufig vorkommt, dass eine Modellgruppe keine eigenständige semantische Bedeutung hat, weil sie meist Teil der Definition eines komplexen Datentyps ist oder als Teil komplexeren Struktur vorkommt, wird noch ein alternativer Ansatz vorgeschlagen:

`MG2:` Die Grammatik, die durch die Modellgruppe ausgedrückt wird, wird durch eine Beschränkung festgehalten. Das geschieht als textuelle Notation der hierarchischen Struktur und Reihenfolge. Vorgeschlagen wird, eine um benutzerdefinierte Auftretenshäufigkeiten, Referenzen und `all`-Operator erweiterte DTD-Notation zu verwenden. Durch Assoziationen werden alle Teile der Modellgruppe mit dieser verbunden. Die Autoren schlagen vor, möglichst diesen Ansatz zu verwenden (siehe Abbildung 4.30).

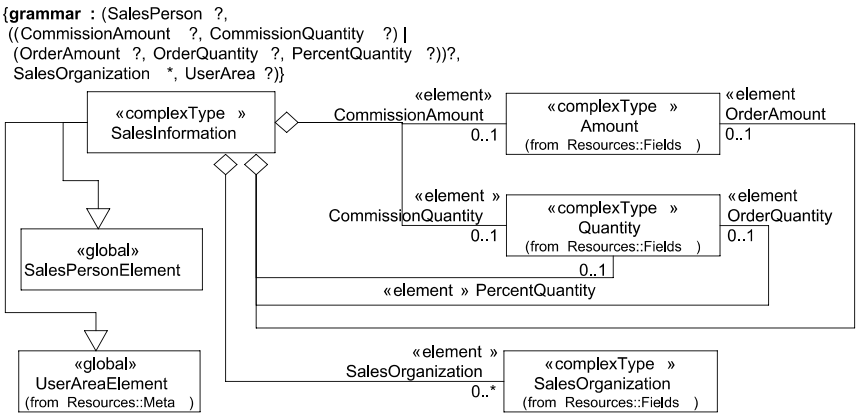


Abbildung 4.30: Komplexes Inhaltsmodell nach `MG2` (aus [BKK04b])

Identitätsbeschränkungen

`KY1:` Schlüsselbedingungen werden als Beschränkung in der Form {`key` *name*`:for_all selector1, sel2, ... : field1, field2, ...` } definiert. Diese Beschränkung wird der Klasse zugeordnet, die das Element, für welches der Schlüssel gelten soll, definiert. Für die Definition von `selector` wird eine vereinfachte OCL-Notation (OCL = object constraint language, Teil des UML-Standards) vorgeschlagen, die dann später in XPath-Ausdrücke übersetzt werden muss (siehe Abbildung 4.31).

Eindeutigkeitsbedingungen und Fremdschlüsselbedingungen (in XML-Schema: `unique` und `keyref`) lassen sich auf ähnliche Weise darstellen. Für Details sei auf [BKK04b] verwiesen.

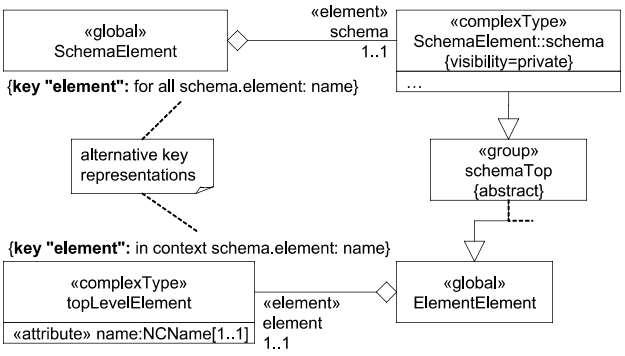


Abbildung 4.31: `Key`-Beschränkung (aus dem Schema für XML-Schema) (aus [BKK04b])

Gruppendefinition und Referenzen

`GA:` Benannte Attributgruppen werden als abstrakte Klassen mit dem stereotyp `<<group>>` dargestellt. Die enthaltenen Attribute werden analog zu Regel `AL` dargestellt. Referenzen zur Attributgruppe werden als Generalisierung abgebildet.

`GE1:` Elementgruppen werden als Klasse mit dem Stereotyp `<<group>>` dargestellt. Jede Referenz zu dieser Gruppe wird als Kompositions-Beziehung umgesetzt, wie in `MG1`. Die Klasse wird nach der Gruppe benannt und nicht geschachtelt.

`GE2:` Diese Regel wird angewendet, wenn die Elementgruppe in Modellgruppen nach `MG2` verwendet werden. Die Elementgruppe wird in diesem Fall als abstrakte Klasse mit dem Gruppennamen dargestellt. Referenzen zu der Gruppe werden als Generalisierung umgesetzt.

Kommentare

`AN:` Kommentare (in XML-Schema: `annotations`) werden durch Kommentare in UML dargestellt. Um XML-Schema-konform zu sein, werden die Stereotypen `<<appInfo>>` und `<<documentation>>` eingeführt, um zwischen Anwendungs- und Benutzerinformationen unterscheiden zu können (siehe Abbildung 4.28).

Notation

Notationen in XML definieren gewöhnlich eine benannte Referenz zu einer Anwendung.

`NO:` Jede Notation wird als Literal mit dem Stereotyp `<<notation>>` in einer Aufzählung mit dem Namen `Notations` dargestellt. Diese Aufzählung enthält alle Notationen eines Schemas und durch Generalisierung auch die Notationen aller eingeschlossenen Schemata.

4.3.10 Weitere Arbeiten auf der Basis von UML

Die beiden hier beschriebenen Ansätzen stehen stellvertretend für eine Vielzahl weiterer Ansätze, die sich mit der Darstellung von XML-Inhaltsmodellen in UML und die daran anschließende Übersetzung in eine XML-Schema-Beschreibung befassen. Alle Ansätze basieren auf UML-Klassendiagrammen und ähneln sich daher stark. Unterschiede treten auf, jenachdem ob eine DTD oder ein XML-Schema erzeugt werden soll. Das liegt natürlich darin begründet, dass der XML-Schema-Standard wesentlich umfangreicher ist

und eine Vielzahl weiterer Konzepte gegenüber DTDs enthält, die im Ableitungsprozess berücksichtigt werden müssen.

Als Einstiegspunkt in das Themengebiet Modellierung von XML-Schema mittels UML eignet sich der Artikel "Representing XML Schema in UML - A Comparison of Approaches'" [BKK04a] von Bernauer, Kappel und Kramler. Die Autoren vergleichen darin fünf verschiedene Ansätze bezüglich der Möglichkeiten der Modellierung von XML-Schema mit UML. Darin kommen sie zu dem Ergebnis, dass nur ihr Ansatz, welcher auch in dieser Diplomarbeit näher beschrieben ist, den XML-Schema-Standard vollständig abbildet. Es muss allerdings gesagt werden, dass die anderen Ansätze in der Regel allgemeinere konzeptuelle Modelle für XML entwickeln, die anschließend in XML-Schema übersetzt werden. Deshalb ist es nicht erforderlich, alle Konzepte von XML-Schema zu unterstützen.

Einen interessanten Weg gehen auch Routledge und Goodchild in [RBG02]. Sie stellen einen zweistufigen Ableitungsprozess vor. Die konzeptuelle Ebene wird durch ein UML-Klassendiagramm mit zusätzlichen konzeptuellen Beschränkungen beschrieben. Dieses Modell wird zunächst in ein anderes UML-Klassendiagramm exportiert. Es basiert auf einem XML-Schema Profil, das heißt die verschiedenen XML-Schema-Konstrukte lassen sich damit darstellen. Es repräsentiert die logische Ebene. Im nächsten Schritt wird aus diesem Diagramm das eigentliche XML-Schema (physische Ebene) abgeleitet.

Durch die Einführung der drei Ebenen wird vermieden, dass frühe XML-Schema-spezifische Design-Entscheidungen den Modellierungsprozess auf der konzeptuellen Ebene ungünstig beeinflussen.

4.3.11 Zusammenfassende Bewertung UML-basierter Ansätze

Generell ist UML eine gute Wahl zur Modellierung konzeptueller Modelle für XML. Sie wird in der Softwaretechnik vielfältig angewendet und besitzt dadurch einen hohen Bekanntheitsgrad. Im Gegensatz zu ER ist bei UML auch die graphische Notation standardisiert. Durch Stereotypen, als ein im Standard enthaltenen Erweitungsmechnismus, lässt sich UML einfach an die geänderte Bedürfnisse der XML-Modellierung anpassen. Zum Erzeugen der graphischen Modelle existieren eine Vielzahl kommerzieller und freier Tools.

Mit dem XMI-Standard (XML Metadata Interchange) existiert ein XML-basiertes Austauschformat für UML. So ist es möglich UML-Diagramme mit verschiedenen Tools zu erstellen und anschließend in das XMI-Format zu exportieren. Dadurch erhält man eine programmunabhängige, konforme Ausgangsbasis für den Ableitungsprozess des XML-Schemas / der DTD.

Weil XMI ein XML-Format ist, kann der Ableitungsprozess auch mit XSLT-Regeln (eXtensible Stylesheet Language) beschrieben werden. KUDRASS und KRUMBEIN beschreiben die Verwendung von XSLT auf XMI-Dateien in [KK03].

CONRAD, SCHEFFNER und FREYTAG beschreiben die Umwandlung eines UML-Modells in eine DTD. Dadurch fehlen einige XML-Schema-relevante Konzepte, wie zum Beispiel die Ableitung und Erzeugung neuer Datentypen. Ihre Art der Modellierung komplexer Inhaltsmodelle durch Aufspalten der Verbindungslinien ist allerdings nicht UML-konform.

Die Autoren BERNAUER, KAPPEL und KRAMLER bieten in [BKK04b] Lösungsvorschläge für alle XML-Schema-Konzepte. Damit sollte es möglich sein ein beliebiges Schema in UML-Notation darzustellen.

An ein paar Stellen werden an sich ähnliche Konzepte auf verschiedene Art umgesetzt. Außerdem sehe ich kleinere Probleme bei den Regeln für globale Elemente und Attribute (`GA` und `GE1/2`). Die Autoren verwenden hier Namenssuffixe. Vielleicht könnte man stattdessen Stereotypen `<<globalElement>>` und `<<globalAttribut>>` verwenden, um diesen Umstand zu vermeiden. Ein weiteres Problem sehe ich in Regel `MG2`. Die textuelle Notation der Grammatik erhöht zwar die Übersichtlichkeit, müsste allerdings auch auf Korrektheit überprüft werden können. Da gerade die Definition solcher Strukturen ein Hauptbestandteil der XML-Schema-Definition ist, sollte der Benutzer dabei auch graphisch unterstützt werden.

Da mit diesem Ansatz das Ziel verfolgt wurde, XML-Schemas direkt abbilden zu können, ist er sehr detailliert. Das Ergebnis ist aber mehr eine graphische Notation für XML-Schema als ein konzeptuelles Modell. Ein konzeptuelles Modell sollte wesentlich einfacher aufgebaut sein und braucht zu Gunsten der Einfachheit nicht alle Fähigkeiten von XML-Schema 1:1 unterstützen. Die Modellierung sollte auf abstrakterer Ebene erfolgen. Dadurch ließen sich einzelne Details einfacher umsetzen und der Einarbeitungsaufwand für den Anwender würde verkürzt.

Die weiteren Ansätze ähneln sich, da auch sie alle auf UML-Klassendiagrammen basieren. Lediglich in Detailfragen gibt es Unterschiede. Zum Beispiel werden Inhaltsmodelle mal durch UML-Attribute und an anderer Stelle durch Beziehungen realisiert. Häufig bieten Autoren auch Alternativen für die Umsetzung desselben Konzeptes an. Bei einer Implementierung müsste der Programmierer sich für die Variante, die am besten ins „Gesamtbild“ passt, entscheiden. Einige Autoren machen Erweiterungen, die nicht UML-konform sind. Allerdings sollten sich auf Grund der Flexibilität von UML immer konforme Lösungen finden lassen.

Als problematisch bei der Modellierung von XML-Inhaltsmodellen mittels UML erweist sich auch hier wieder die Ordnung der Elemente (z.B. in Sequenzen), welche in XML explizit gegeben ist. UML-Klassen-Diagramme benötigen dieses Konzept nicht und müssen an dieser Stelle erweitert werden. In der Literatur finden sich verschiedene Ansätze:

- **Ablesen aus graphischer Notation:** Die Ordnung soll aus der Abbildung abgelesen werden (links → rechts, oben → unten). Dieser Vorschlag ist jedoch nicht praktikabel, da diese Informationen spätestens beim XMI-Export verloren gehen. Außerdem wird der Benutzer bei der Modellierung eingeschränkt. Es lassen sich sicherlich Fälle konstruieren, wo sich komplexe Inhaltsmodelle nicht in einem UML-Diagramm darstellen lassen.
- **Einführung von Bezeichnern:** Die Beschreibung der Ordnung durch Kennzeichnung der Verbindungslinien oder Subelementen mit Bezeichnern oder in zusätzlichen Kommentaren ist die bessere Lösung. Es kann eine fortlaufende oder hierarchische Nummerierung verwendet werden. Allerdings wird dadurch der Aufwand für den Anwender geringfügig erhöht.

Kapitel 5

Das EMX-Modell

In diesem Kapitel wird ein neues konzeptuelles Modell für XML-Schemas eingeführt - das *EMX-Modell*.
EMX ist die Abkürzung für *Entity Model for XML Schema*.

5.1 Grundlagen

Das EMX-Modell besteht aus folgenden Basiskonstrukten:

Entities (Synonyme: Objekt, Einheit, Bestandteil) beinhalten die wichtigsten Informationen eines Entwurfs. Die Verwendung des Begriffs ist analog zu *Entities* im Entity-Relationship-Modell. Für die Modellobjekte des EMX-Modells werden verschiedene Entity-Typen eingeführt.

Verbindungen treten zwischen zwei Entities auf. Sie können *gerichtet* sein. In disem Fall entspricht die Richtung der Verbindung semantisch einer Enthaltensein-Beziehung zwischen den Entities. Während des Entwurfsprozesses sind auch *ungerichtete* Verbindungen zulässig, wenn dem Entwerfer die Richtung zum Zeitpunkt der Erstellung noch nicht klar ist. Vor der Ableitung des XML-Schemas müssen jedoch sämtliche ungerichtete Verbindungen durch gerichtete ersetzt werden. Die Anzahl der eingehenden und ausgehenden Verbindungen eines Entities ist abhängig vom Entity-Typ.

Eigenschaften Entities und Verbindungen können Eigenschaften zugewiesen werden, die das Objekt genauer beschreiben. Eigenschaften sind Schlüssel-Wert-Paare. Auf die Bezeichnung „Attribute“ wurde verzichtet, um Namenskonflikte mit XML-Attributen zu vermeiden.

Da die Definition neuer Datentypen ein Schwerpunkt des XML-Schema-Standards ist, wurde darauf geachtet, dass die Konstrukte, die XML-Schema dazu bietet, so genau wie möglich umgesetzt werden.

Bei der Beschreibung von XML-Inhaltsmodellen mittels XML-Schema gibt es verschiedene Darstellungsmöglichkeiten (*XML-Schema-Designs*). Mit unterschiedlicher Syntax lässt sich eine identische Semantik beschreiben. Die veschiedenen XML-Schema-Designs werden im Abschnitt 6.4.1 vorgestellt. Durch die Festlegung auf ein Schema-Design werden Mehrdeutigkeiten bei der Ableitung eines XML-Schemas vermieden, da das Schema-Design gewisse Regeln für die Struktur vorschreibt. Weiterhin kann der Anwender in der Designphase durch eine automatische Überprüfung des EMX-Modells besser unterstützt werden. Für das EMX-Modell wurde das „Venetian Blind"-Schema-Design als Basis ausgewählt. Das hat verschiedene Auswirkungen. So impliziert die Verwendung von „Venetian Blind" verschiedene Einschränkungen bei den Möglichkeiten der Verbindung unterschiedlicher Modellentities. Außerdem gibt es keine anonymen (unbenannten) Typdefinitionen. Grundsätzlich ist das Modell jedoch fähig, auch andere Schema-Designs zu unterstützen. Dafür müsste ein Konzept zur Darstellung anonymer Typdefinitionen eingeführt werden und einige Programmanpassungen bei der automatischen Modellüberprüfung und im Übersetzungsprozess vorgenommen werden.

In dieser ersten prototypischen Implementierung werden Namespaces nicht unterstützt. Die Funktionalität lässt sich nachimplementieren, indem für jedes Modellentity für eine Element- oder Typdefinition eine zusätzliche Namespace-Eigenschaft eingeführt wird.

5.2 Modellobjekte und ihre graph. Repräsentation

In diesem Abschnitt werden die verschiedenen Modellobjekte mit ihren Eigenschaften vorgestellt. Alle Modellobjekte sind spezielle Entities. Nur die wichtigesten Eigenschaften werden in der Graphik dargestellt. Weniger wichtige Eigenschaften werden in einer Tabelle (Properties View) außerhalb der graphischen Notation angezeigt. Das stellt die einfache Lesbarkeit des Modells sicher. Innerhalb des Properties View lassen sich alle Eigenschaften (auch die, die in der graphischen Notation enthalten sind) editieren.

5.2.1 Entity: Element

Elemente werden in der graphischen Notation durch eine Ellipse dargestellt. Sie besitzen einen Namen und die Eigenschaften `minOccurs` und `maxOccurs`. Diese Eigenschaften zeigen an, wie oft die Elemente innerhalb einer Gruppendefinition auftreten können. Sie stehen in der graphischen Notation an den Verbindungen zwischen dem Gruppenoperator und dem darin auftretenden Element. Auf die Eigenschaft `root` kann bei Verwendung des „Ventian Blind"-Schema-Designs verzichtet werden, da es vorschreibt, dass alle Elementdeklarationen lokal sind. Somit sind alle Elemente ohne eingehende Kanten automatisch als *Root-Elemente* identifzierbar. Das Root-Merkmal

wird in der graphischen Notation durch ein Symbol am Element angezeigt, welches automatisch gesetzt oder entfernt wird.

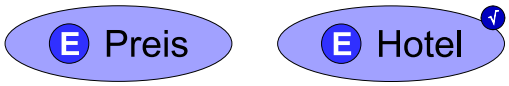


Abbildung 5.1: Notationen für „einfaches“ Element und Root-Element

5.2.2 Entity: Einfacher Datentyp

Da sich das EMX-Modell im Bezug auf die Datentypdefinition möglichst dicht am XML-Schema-Standard orientieren soll, gibt es vier verschiedene Arten von Einfachen Datentypen. Die graphische Notation besteht aus einem Rechteck, welches den Namen des Typs und einen zusätzlichen Hinweis über seine Art enhält.

Es werden die folgenden vier Arten eingeführt:

built-in. Mit dieser Art werden die Datentypen modelliert, die im XML-Schema-Standard [XS001] definiert sind. Dem Nutzer wird eine Liste von Typen vorgegeben, aus welcher er sich den entsprechenden Standard-Datentyp auswählen kann. Der Inhalt dieser Liste kann so voreingestellt werden, dass zu Gunsten der Übersichtlichkeit nur die gebräuchlichsten Typen angezeigt werden.

user-defined. Benutzerdefinierte Datentypen haben einen Namen und basieren auf einem Standard-Datentyp. Dieser wird im XML-Schema-Standard durch zusätzliche Eigenschaften (*engl.* facets) erweitert. Der Name des Typs wird in der graphischen Notation angezeigt. Seinen Basistyp und die Facets findet man im Properties View. Ausgehend vom Properties View lässt sich ein zusätzlicher Dialog öffnen, der das Hinzufügen, Editieren und Löschen einzelner Facets ermöglicht.

list. Der XML-Schema-Standard bietet die Möglichkeit, Listen über einem einfachen Datentyp zu modellieren. Im EMX-Modell wird das durch ein weiteres Modellobjekt für einfache Datentypen dargestellt. Die graphische Notation enthält lediglich den Namen und einen Hinweis, dass es sich um einen Listentyp handelt. Zur Beschreibung des Typs der Listenelemente existiert eine Verbindung zu einem Modellobjekt, welches den Datentyp der Listenelemente beschreibt.

union. Der XML-Schema-Standard bietet die Möglichkeit, einen neuen Datentyp aus verschieden bestehenden Datentypen zu bilden, indem deren Wertebereiche vereint werden. Die graphische Darstellung erfolgt

analog zum Listentyp, nur dass hier Verbindungen zu mehreren Modellelementen für einfache Datentypen möglich sind.

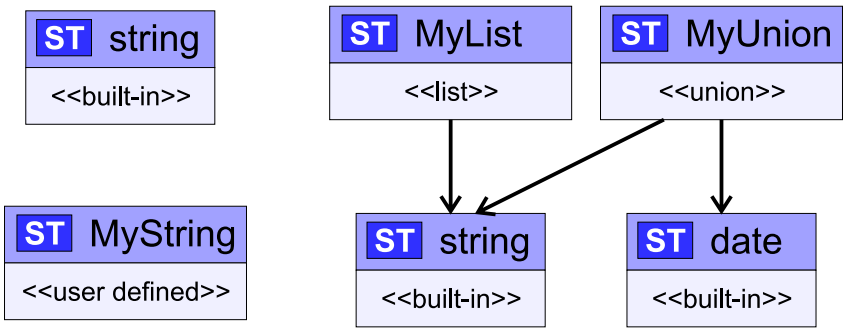


Abbildung 5.2: Beispielnotationen für Einfache Datentypen

5.2.3 Entity: Komplexer Datentyp

Das Modellobjekt für einen komplexen Datentyp enhält den Namen des Types und die Eigenschaft `mixed`, welche anzeigt, ob der Datentyp zusätzlich zu den Subelementen noch weiteren Text zulässt (*engl.* mixed content). Das Inhaltsmodell wird durch weitere Entities (Attributbox, Gruppenoperator, SimpleType) gebildet, die mit diesem Entity verbunden sind. Wenn das Inhaltsmodell ein Einfacher Datentyp ist, entspricht das Entity dem XML-Schema-Konstrukt `SimpleContentComplexType`, welches einen einfachen Datentyp um Attribute erweitert.

Beispiele für die graphische Notation findet man in den Abbildungen 5.4 und 5.3).

5.2.4 Entity: Gruppenoperator

Der Gruppenoperator ist das Basiskonstrukt zur Bildung komplexer Inhaltsmodelle. Er hat mehrere Ausgänge, an die jeweils eine Verbindung angedockt werden kann. Die Anzahl der Ausgänge kann der Nutzer festlegen und ändern. Es ist nicht notwendig, alle Andockpunkte zu belegen. Leere Andockpunkte werden im Ableitungsprozess ignoriert. Die Einführung der Andockpunkte ist notwendig, da nur so die Reihenfolge für die Subelemente definiert werden kann. Bei „normalen“ Entitäten und auch im formalen Graphenmodell kann man die Reihenfolge der Subelemente / Kindknoten nicht explizit festlegen. Theoretisch könnte man die Reihenfolge aus der graphischen Notation (z.B. durch *Lesen von links nach rechts*) ablesen. Das ist jedoch nicht praktikabel, da das Verschieben von Subelementen (z.B. während des Layoutens) Einfluss auf ihre Reihenfolge im Modell hätte. Durch die Andockpunkte wird die Reihenfolge gesichert.

Der Gruppenoperator kann folgende Eigenschaften haben:

`minOccurs`, `maxOccurs`
: Sie legen fest, wie oft der Operator in einen übergeordneten Gruppenoperator eingeht.

`mode`
: Sie legt den Modus entsprechend der XML-Schema-Definition (`sequence`, `choice`, `all`) fest.

`numDockingPoints`
: Sie legt die Anzahl der Andockpunkte für Subelemente fest.

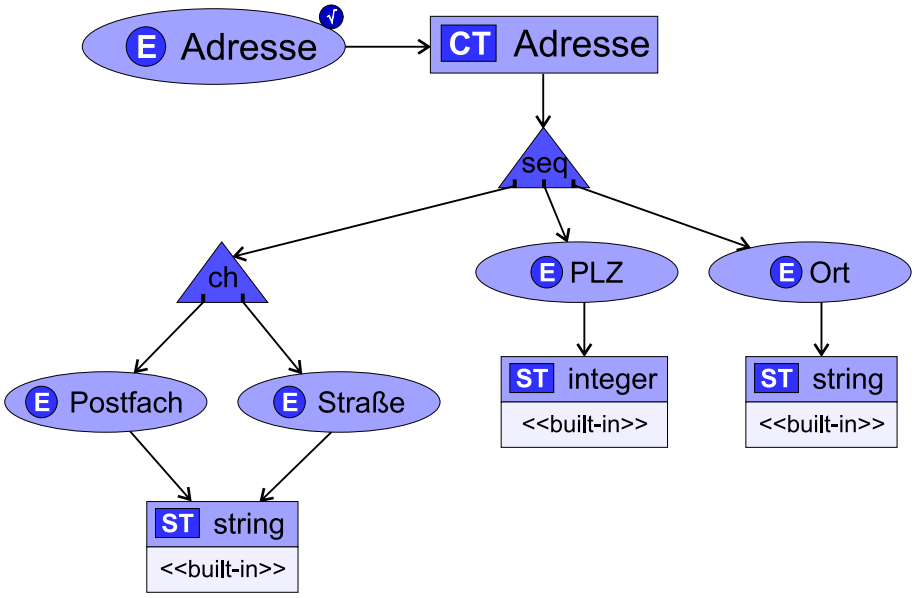


Abbildung 5.3: Beispielnotation für Komplexen Datentyp und Gruppenoperatoren

5.2.5 Entity: Attributbox

Für die Modellierung von Attributen standen folgende Konzepte zur Wahl:

Modellierung als einzelne Entities.
: Mit dieser Variante würden Attribute und Elemente auf ähnliche Weise behandelt werden. Wilde schlägt so ein Vorgehen in [Wil05] vor. Die Konsequenzen wurden in Abschnitt 3.1 bereits erörtert.

Modellierung als Eigenschaften eins Komplexen Datentyps
: Diese Variante ist am dichtesten an der Umsetzung im XML-Schema-Standard, wo Attribute auch Bestandteil einer `ComplexType`-Definition sind.

Ausgliederung in einem eigenen Entity: Attributbox
Das bedeutet, das alle Attributdefinitionen für einen komplexen Typ in einem weiteren Entity zusammengefasst werden.

Die letzte Variante wurde implementiert, da sie die Wiederverwendung von Attributdefinitionen ermöglicht. Inspiriert wurde sie durch das XML-Schema-Konstrukt `attributeGroup`. Es gestattet die Definition von Attributen außerhalb der Typdefinitionen. Mehrere Typdefinitionen können dadurch ein und dieselbe Attributdefinition verwenden.

In der Implementation wurde das `attributeGroup`-Konzept nicht vollständig umgesetzt. Bei der Übersetzung in ein XML-Schema werden die Attribute wieder in die Definition des komplexen Typs integriert. Falls nötig lässt sich diese Variante jedoch einfach erweitern, um vollständig zur XML-Schema-`attributeGroup` konform zu sein. Dazu müsste die Benennung der Attributbox ermöglicht werden.

In der graphischen Notation werden nur einige der Eigenschaften der Attribute angezeigt. Details und Möglichkeiten zum Editieren der einzelnen Attribute befinden sich wieder im Properties View. Die Definition der einzelnen Attribute wurde entsprechend dem XML-Schema-Standard umgesetzt.

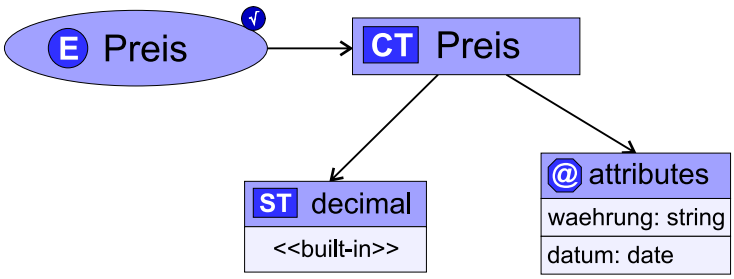


Abbildung 5.4: Beispielnotation für komplexen Datentyp mit einfachem Datentyp als Inhalt und Attributen

5.2.6 Entity: Modul

Module werden verwendet, um externe EMX-Modelle darstellen zu können. Sie können geöffnet dargestellt werden; in diesem Fall zeigen sie das EMX-Modell des ihnen zugeordneten XML-Schemas an.

In der prototypischen Implementation dieser Arbeit werden jedoch lediglich die Typdefinitionen aufgelistet, auf welche von außerhalb zugegriffen werden kann. Ist der Anwender am vollständigen EMX-Modell interessiert, muss er die entsprechende Datei im Editor öffnen. Im Programm lassen sich dazu mehre Instanzen des Editors parallel öffnen. Für eine zukünftige Version des Programms könnte man darüber nachdenken, auch den Modul-

inhalt als vollständiges EMX-Modell-Graphen anzeigen lassen. Dabei ist zu überprüfen, inwieweit das die Übersichtlichkeit negativ beeinflussen würde.

Alle wiederverwendbaren Typdefinitionen werden als **External Entities** modelliert. Sie können nur innerhalb eines Modul-Entities existieren.

Das Modul dient somit als Container für External Entities. Verbindungen bestehen zwischen Elementen und den External Entities innerhalb des Moduls. Im geschlossenen Zustand des Moduls werden die External Entities verborgen. Verbindungen zu den External Entities im Modul werden so dargestellt, als ob sie zum Modul führen würden. Im Modell bestehen sie jedoch noch immer zwischen Elementen und External Entities. Das bedeutet, dass auf Modellebene keine Verbindungen zum Modul als Entity gestattet sind.

Als Eigenschaften werden dem Modul die Adresse des externen Schemas sowie der Zustand (geöffnet / geschlossen) zugeordnet.

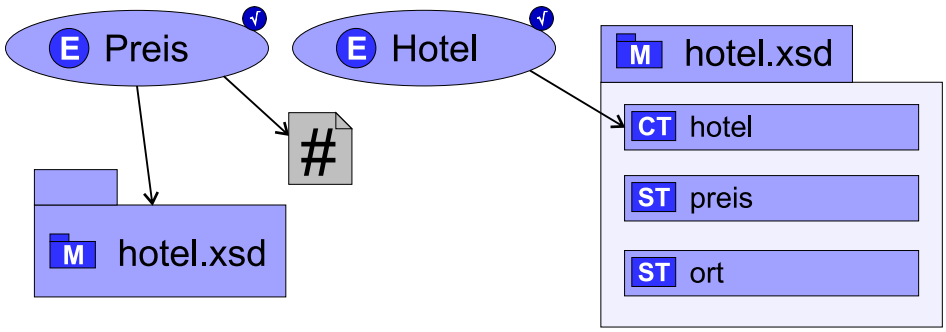


Abbildung 5.5: Beispielnotation für geschlossenes Modul mit Anmerkung und geöffnetes Modul

5.2.7 Entity: Anmerkung

Anmerkungen sind ebenfalls spezielle Entitys. Sie können anderen Entities zugeordnet werden. Ihre Eigenschaften orientieren sich am XML-Schema `annotation`-Konstrukt.

Deshalb besitzen sie die zwei Eigenschaften:
`documentation` und `appInfo`.
Die `documentation`-Eigenschaft enthält Hinweise für den Nutzer und die `appInfo`-Eigenschaft Anweisungen für Programme, die die dem Schema zugehörenden XML-Dateien bearbeiten. Die graphische Notation für die Anmerkung kann man in Abbildung 5.5 sehen.

5.3 Formale Grundlagen

Das formale Modell für EMX basiert auf einem gemischten Graph (*engl.* mixed graph).

Es handelt sich also um ein 3-Tupel: $G = (V, E, A)$, mit

- V, einer Menge von Knoten
- E, einer Menge ungerichteter Kanten. Eine ungerichtete Kante $e \in E$ ist ein ungeordnetes Knotenpaar: $e = (v_1, v_2)$ mit $v_1, v_2 \in V$
- A, einer Menge gerichteter Kanten. Eine gerichtete Kante $a \in A$ ist ein geordnetes Knotenpaar: $a = (v_1, v_2)$ mit $v_1, v_2 \in V$

Für die unterschiedlichen Entity-Typen werden *disjunkte* Teilmengen von V eingeführt. Es gilt:

$$V = El \cup CT \cup ST_{blt-in} \cup ST_{udef} \cup ST_{list} \cup ST_{union} \cup Grp \cup AttrBox \cup Ann \cup EE_{ST} \cup EE_{CT} \cup EE_{El} \cup M$$

mit
El = Menge der Element-Entities,
CT = Menge der ComplexType-Entities,
ST_{blt-in} = Menge der SimpleType-Entities, (für Datentypen der XML-Schema-Spezifikation [XS101]),
ST_{udef} = Menge der benutzerdefinierten SimpleType-Entities,
ST_{list} = Menge der List-SimpleType-Entities,
ST_{union} = Menge der Union-SimpleType-Entities,
Grp = Menge der Group-Entities,
$AttrBox$ = Menge der AttributeBox-Enties,
Ann = Menge der Annotation-Entities,
EE_{ST} = Menge der SimpleType-ExternalEntities,
EE_{CT} = Menge der ComplexType-ExternalEntities,
EE_{El} = Menge der Element-ExternalEntities,
M = Menge der Module.

Um das formale Modell zu vervollständigen, werden noch zwei Menge von Regeln R_a und R_e eingeführt. Die Regeln haben die Form:

$$r_a : \quad (X \subset V) \times (Y \subset V) \rightarrow \{true, false\} \in R_a$$
$$r_e : \quad (X \subset V) \times (Y \subset V) \rightarrow \{true, false\} \in R_e$$

Für jede mögliche Kombination der oben definierten Teilmengen wird jeweils eine Regel r_a und eine Regel r_e aufgenommen. Diese Regeln zeigen an, ob gerichtete oder ungerichtete Kanten zwischen den Entity-Typen gestattet sind. Sie können aus den Abbildungen 6.2 und 6.3 abgelesen werden.

Kapitel 6

Entwurf und Bearbeitung eines EMX-Modells, Import/Export

6.1 Überblick

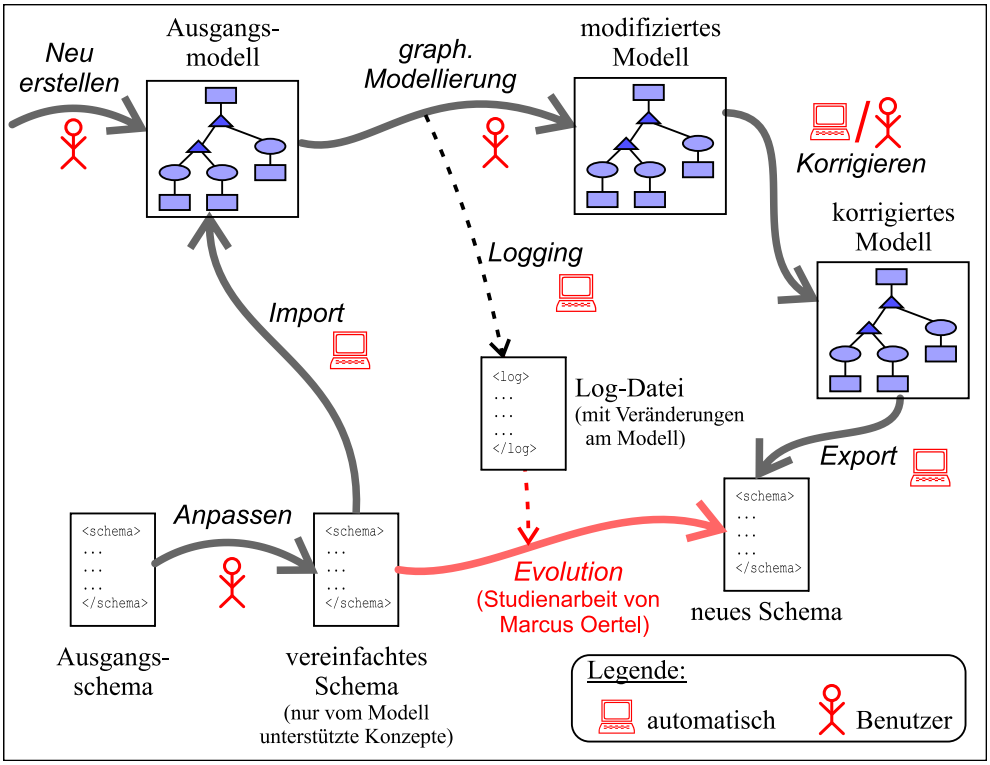


Abbildung 6.1: Operationen auf dem EMX-Modell

Um ein EMX-Modell erstellen und bearbeiten zu können, wurde im Rahmen dieser Diplomarbeit ein Editor entwickelt. Abbildung 6.1 gibt einen Überblick über die Operationen am EMX-Modell, die der Editor unterstützen soll.

Es gibt zwei Einstiegspunkte. Der Nutzer kann entweder ein *leeres Modell* erzeugen oder ein Modell (mittels *Importfunktion*) aus einem bestehenden XML-Schema generieren.

Um das Modell aus einem bestehenden Schema zu importieren, sind zwei Schritte notwendig: Zuerst muss das Schema angepasst werden. Konstrukte, die der Editor nicht darstellen kann, werden in einem Vorübersetzungsschritt umgewandelt oder entfernt. Die Modifikationen haben keinen Einfluss auf die durch das XML-Schema akzeptierte Menge von XML-Dokumenten. Das angepasste Schema kann anschließend in ein EMX-Modell umgewandelt werden. Im Rahmen dieser Diplomarbeit wurde nur der zweite Schritt implementiert. Die Anpassung des Schemas muss manuell erfolgen.

Für die Arbeit am Modell stellt der Editor eine Vielzahl von Funktionen zur Verfügung. Sie umfassen u. a. das Erstellen, Modifizieren und Löschen von Entities und Verbindungen sowie das Modifizieren von Eigenschaften der Entities und Verbindungen.

Um aus dem Modell wieder ein XML-Schema zu erzeugen, ist ebenfalls ein zweistufiger Ableitungsprozess nötig. In der ersten Phase wird das Modell auf eventuelle Fehler untersucht. Diese werden entweder automatisch korrigiert oder dem Nutzer angezeigt, damit er sie manuell korrigieren kannn. Anschließend wird aus dem „berichtigten“ Modell das XML-Schema abgeleitet.

Aufbauend auf dieser Diplomarbeit, beschäftigt sich Marcus Oertel in seiner Studienarbeit mit der Schema-Evolution. Die Änderungen am Modell werden in einer Log-Datei protokolliert. Im Rahmen der Studienarbeit wird untersucht, ob die Änderungsschritte auch direkt auf die XML-Schema-Datei angewendet werden können. Das direkt modifizierte Schema sollte semantisch identisch mit dem Schema sein, welches aus dem bearbeiteten Modell abgeleitet wird.

In den nächsten Abschnitten sollen die einzelnen Funktionen im Detail vorgestellt werden.

6.2 Arbeit am Modell

Mit den folgenden Funktionen lässt sich das EMX-Modell modifizieren:

Anlegen von Entities
: Entities sind die wichtigsten Bestandteile des Modells. Der Editor kann die unterschiedlichen Entity-Typen erzeugen.

Löschen von Entities

Der Editor ist ebenfalls in der Lage, Entities wieder aus dem Modell zu entfernen. Beim Löschen ist darauf zu achten, dass auch alle Verbindungen zu und von diesem Entity gelöscht werden.

Hinzufügen, Ändern, Löschen von Entity-Eigenschaften

Die Eigenschaften der Entities werden durch den Editor erzeugt, geändert oder gelöscht. Das betrifft sowohl Eigenschaften, die direkt in der graphischen Repräsentation sichtbar sind, als auch Eigenschaften, die als Zusatzinformationen außerhalb der graphischen Repräsentation existieren. Zu den Änderungsoperationen zählt auch das Umbenennen von Entities.

Erzeugen von Verbindungen

Verbindungen zwischen Entities sind ein weiterer Bestandteil des Modells. Sie repräsentieren Beziehungen zwischen Entities. Der Editor kann sowohl gerichtete als auch ungerichtete Verbindungen anlegen. Es ist Voraussetzung, dass die Entities in einem früheren Schritt bereits erzeugt wurden.

Löschen von Verbindungen

Nicht mehr benötigte Verbindungen zwischen Entities können im Editor natürlich auch wieder entfernen werden. Das Löschen von Verbindungen erfolgt automatisch, wenn eines der durch sie in Beziehung stehenden Entities gelöscht wird.

Festlegen der Richtung von Verbindungen

Verbindungen sollten in der Regel gerichtet sein. Ihre Richtung bestimmt die Enthaltensein-Beziehung zwischen den beteiligten Entities. Ungerichtete Verbindungen sind zulässig, damit der Anwender eine Beziehung zwischen Entities ausdrücken kann, deren Richtung ihm zum Zeitpunkt des Erzeugens noch unbekannt ist. Allerdings ist solche Verbindungen vor dem Start des Ableitungsprozesses in ein XML-Schema eine Richtung zuzuweisen oder sie müssen wieder entfernt werden. Dies kann in eindeutigen Fällen automatisch geschehen. (Ein Beispiel für einen eindeutigen Fall ist die Beziehung zwischen komplexer Typdefinition und Gruppenoperator. Der Gruppenoperator ist in jedem Fall Teil des komplexen Typs.) Bei nicht eindeutigen Fällen ist der Nutzer vor dem Start des Ableitungsprozesses auf solche „Problem“-Verbindungen hinzuweisen. Er muss diese dann manuell beseitigen. Das Ändern der Richtung kann trivialer Weise durch Löschen und neues Erzeugen der Verbindung erfolgen. Wünschenswert wäre allerdings die Unterstützung durch den Editor.

Export
Aus dem EMX-Modell wird während des Exports eine XML-Schema-Definition erzeugt. Der Export-Prozess wird im Abschnitt 6.3 genauer beschrieben.

Import
Mit dem Editor kann ein EMX-Modell aus einer bestehenden XML-Schema-Definition generiert werden. Der Import-Prozess wird im Abschnitt 6.4 genauer beschrieben.

6.3 Export: EMX-Modell → XML-Schema

Der Export-Prozess gliedert sich in zwei Stufen:

Korrektur-Phase
Während des Entwurfs wird dem Nutzer die größtmögliche Freiheit gelassen. Es können dadurch zeitweilig unvollständige und auch falsche Modellinstanzen enstehen. So können zum Beispiel temporär Entities auftreten, die eigentlich nicht ohne Verbindung zu anderen Entities existieren können. Die bestehenden Verbindungen müssen dahingehend überprüft werden, ob die durch sie verbundenen Entities überhaupt in Beziehung stehen dürfen. Ungerichtete Verbindungen müssen beseitigt werden. Dem Nutzer werden Problemfälle angezeigt, damit er sie korrigieren kann.

Ableitungsphase
In dieser Phase wird aus dem „korrigierten“ Modell das XML-Schema generiert. Da alle Problemfälle im ersten Schritt beseitigt worden sind, kann dieser Prozess völlig automatisch durchgeführt werden.

6.3.1 Korrekturphase

Auflösen ungerichteter Verbindungen

Während des Entwurfes sind ungerichtete Verbindungen zulässig. Ihnen muss, bevor der Ableitungsprozess gestartet werden kann, eine Richtung zugewiesen werden. Diese Richtung kann in vielen Fällen automatisch bestimmt werden.

Abbildung 6.2 enthält eine Tabelle, die anzeigt, welche Richtung einer ungerichteten Verbindung zugewiesen werden kann und in welchen Fällen der Nutzer gefragt werden muss. Es kann sein, dass temporär unzulässige Verbindungen erzeugt werden, die in einem späteren Schritt berichtigt werden müssen. In diesem Schritt soll es lediglich um die Festlegung einer Richtung gehen.

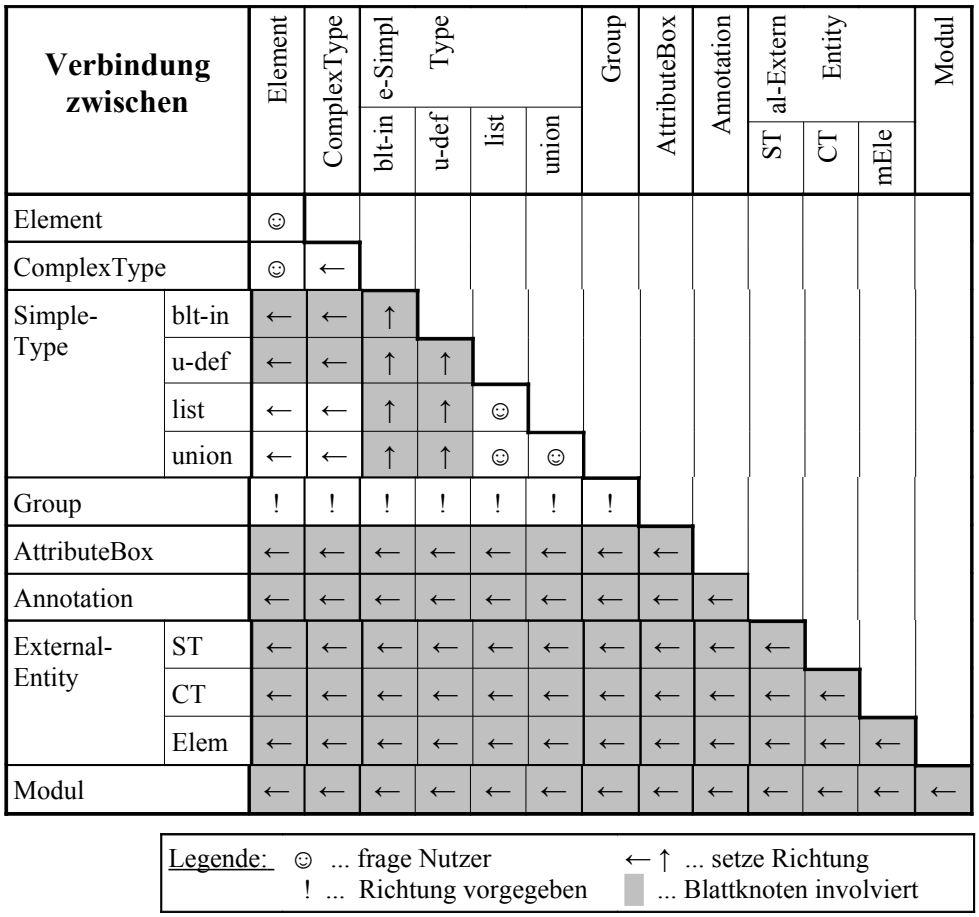


Abbildung 6.2: Festlegung einer Richtung für ungerichtete Verbindungen

Die meisten Fälle können automatisch gelöst werden, da bestimmte Entities nur als Blattknoten auftreten können oder zwischen zwei Entities eindeutig eine Parent-Child-Beziehung existiert.

Am Gruppenoperator ergibt sich die Richtung der eingehenden und ausgehenden Verbindungen auf Grund der Andockposition. Ausgehende Verbindungen beginnen immer an einem festen Andockpunkt. In einigen wenigen Fällen lässt sich die Richtung der Kanten jedoch nicht eindeutig bestimmen. Diese Kanten sind dem Nutzer anzuzeigen, damit er die Richtung manuell setzen kann.

In einer zukünftigen Version könnte man durch die Anwendung graphentheoretischer Ansätze weiteren Verbindungen automatisch eine Richtung zuweisen und dadurch die dem Nutzer angezeigten Problemfälle minimieren.

Erzeugen fehlender Entities

Beim Bearbeiten des Modells im EMX-Editor wird dem Anwender die größtmögliche Freiheit gelassen. In diesem Schritt werden offensichtlich fehlende Entities ermittelt und ergänzt. Die folgende Liste von Modifikationen ist sicherlich nicht vollständig. In einer zukünftigen Version des Editors könnten weitere „Korrekturen“ integriert werden:

- Ist ein Element-Entity Blattknoten im Graph (Es fehlt die Typdefinition.) wird automatisch ein Default-Typ (z.B. `String`) angenommen und das entsprechende SimpleType-Entity sowie die zusätzliche Verbindung erzeugt.
- Besteht eine direkte Verbindung zwischen einem Element und einem Gruppenoperator wird ein ComplexType-Entity dazwischen eingefügt. Der Name des ComplexType-Entities kann aus dem Elementnamen abgeleitet werden. Dabei ist darauf zu achten, dass der Name nicht bereits einem anderen Type-Entity im Modell zugeordnet wurde.
- Besitzt ein Element mehrere Subelemente als direkte Nachfolger, wird automatisch eine ComplexType-Entity und ein Gruppenoperator-Entity vom Typ `sequence` eingfügt. Die Reihenfolge der Subelemente könnte man anhand ihrer graphischen Position festlegen.

Kontrolle der Verbindungen

Es gibt eine Vielzahl von Entity-Typen, die nicht miteinander in Verbindung stehen können. Die Tabelle in Abbildung 6.3 zeigt, welche Entities miteinander in Verbindung stehen dürfen. Sie basiert auf den Annahmen zum Modell aus Kapitel 5. Fehler werden dem Nutzer zur Berichtigung angezeigt.

Root-Entities

Nicht alle Entities können ohne Parent-Entity existieren. Root-Entities lassen sich automatisch ermitteln werden; es muss lediglich geprüft werden, ob sie eingehende Verbindungen besitzen. Wurde ein Root-Entity gefunden, wird überprüft, ob sein Typ einem der gültigen Typen für Root-Entities (Element, Typdefinition, (External Entity), Modul) entspricht. Fehler werden dem Nutzer angezeigt.

Namensvalidierung

Es ist sicherzustellen, dass alle Typdefinitionen einen eindeutigen Namen besitzen. Dabei müssen auch die Namen der Typdefinitionen in den Modulen berücksichtigt werden. Die Namen aller Root-Elemente werden ebenfalls auf Eindeutigkeit überprüft.

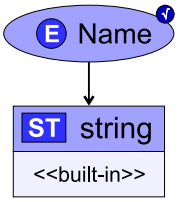
Verbindung zu: / von:		Element	ComplexType	Simple-Type				Group	AttributeBox	Annotation	External-Entity			Modul
				blt-in	u-def	list	union				ST	CT	Elem	
Element		E*	1	1	1	1	1	E1	E1	1	1	1	X	X
ComplexType		E*	X	1	1	1	1	1	1	1	1	X	X	X
Simple-Type	blt-in	X	X	X	X	↔	↔	X	X	X	X	X	X	X
	u-def	X	X	X	X	↔	↔	X	X	1	X	X	X	X
	list	X	X	1	1	1	1	X	X	1	1	X	X	X
	union	X	X	*	*	*	*	X	X	1	*	X	X	X
Group		*	X	X	X	X	X	*	X	X	X	X	X	X
AttributeBox		↔	↔	↔	↔	↔	↔	X	X	X	X	X	X	X
Annotation		↔	↔	↔	↔	↔	↔	X	X	X	X	X	X	X
External-Entity	ST	↔	↔	↔	↔	↔	↔	X	X	X	X	X	X	X
	CT	↔	X	X	X	X	X	X	X	X	X	X	X	X
	Elem	X	X	X	X	X	X	X	X	X	X	X	X	X
Modul		X	X	X	X	X	X	X	X	X	X	X	X	X

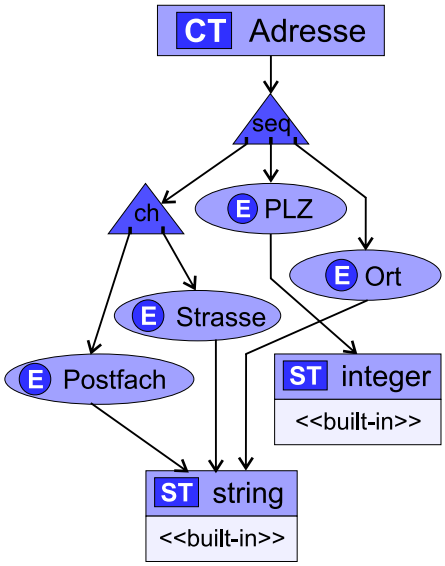
Legende: X ... keine V. 1 ... genau eine V. E ... V. im Entwurf zulässig
↔ ... Umkehren * ... mehrere V.

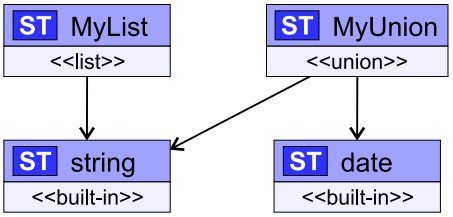
Abbildung 6.3: Erlaubte Verbindungen zwischen Entities

6.3.2 Ableitungsphase

Die Ableitung des XML-Schemas aus dem „korrigierten“ EMX-Modell kann vollständig automatisiert erfolgen. Für jedes Entity und seine Beziehungen zu benachbarten Entities existiert eine entsprechende XML-Schema-Notation. Tabelle 6.1 zeigt beispielhaft Ausschnitte aus dem EMX-Modell und die ihnen entsprechende XML-Notation.

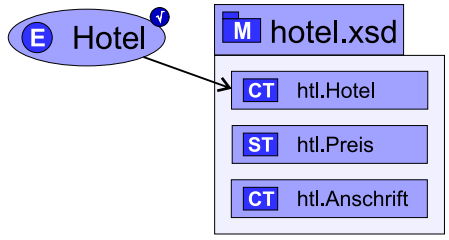
Emx-Entity	XML-Schema
	`<xs:element name="Name"` `type="xs:string"/>`

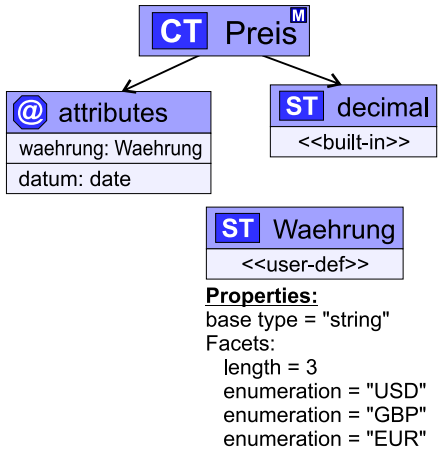
	`<xs:complexType name="Adresse">`
	`<xs:sequence>`
	`<xs:choice>`
	`<xs:element name="Postfach"`
	`type="xs:string"/>`
	`<xs:element name="Strasse"`
	`type="xs:string"/>`
	`</xs:choice>`
	`<xs:element name="PLZ"`
	`type="xs:integer"/>`
	`<xs:element name="Ort"`
	`type="xs:string"/>`
	`</xs:sequence>`
	`</xs:complexType>`

	`<xs:simpleType name="MyList">`
	`<xs:list itemType="xs:int"/>`
	`</xs:simpleType>`
	`<xs:simpleType name="MyUnion">`
	`<xs:union memberTypes`
	`="xs:int xs:date"/>`
	`</xs:simpleType>`

... wird fortgesetzt ...	

Emx-Entity	XML-Schema

	`<xs:include`
	`schemaLocation="htl.xsd"/>`
	`<xs:element name="hotel"`
	`type="htl.Hotel"/>`

	`<xs:simpleType name="Waehrung">`
	`<xs:restriction base="xs:string">`
	`<xs:length value="3"/>`
	`<xs:enumeration value="USD"/>`
	`<xs:enumeration value="GBP"/>`
	`<xs:enumeration value="EUR"/>`
	`</xs:restriction>`
	`</xs:simpleType>`
	`<xs:complexType name="Preis"`
	`mixed="true">`
	`<xs:simpleContent>`
	`<xs:extension base="xs:decimal">`
	`<xs:attribute name="waehrung"`
	`type="Waehrung"/>`
	`<xs:attribute name="datum"`
	`type="xs:date"/>`
	`</xs:extension>`
	`</xs:simpleContent>`
	`</xs:complexType>`

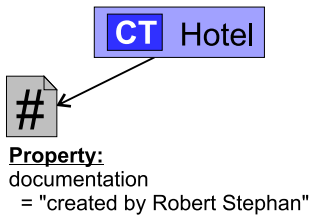
	`<xs:simpleType name="Ausstattung">`
	`<xs:annotation>`
	`<xs:documentation>`
	`created by Robert Stephan`
	`</xs:documentation>`
	`</xs:annotation>`
	`<xs:restriction base="xs:string"/>`
	`</xs:simpleType>`

Tabelle 6.1: EMX-Entities und XML-Schema-Syntax

Bei der automatischen Ableitung ist folgende Reihenfolge einzuhalten:

1. **Anlegen und Initialisieren eines neuen Schemas**
   Zunächst wird eine XML-Schema-Datei erzeugt und initialisiert. Zur Initialisierung gehört unter anderem die Definition des Namespace-Attributs für den XML-Schema-Namespace.

2. **Umwandeln der Module**
   Für jedes Modul wird die entsprechende `include`-Anweisung erzeugt. Das externe XML-Schema wird eingelesen, damit die externen Typdefinitionen im weiteren Ableitungsprozess zur Verfügung stehen.

3. **Anlegen der Typdefinitionen**
   Für alle Typdefinitionen im Modell (einfache und komplexe Datentypen, außer XML-Schema-Standard-Datentypen) im Modell werden die entsprechenden XML-Schema-Typdefinitionen erzeugt. In diesem Schritt wird ihnen zunächst nur der Name zugewiesen. Somit wird sichergestellt, dass im nächsten Schritt Referenzen auf alle (auch noch nicht vollständig definierten) Typdefinitionen möglich sind.

4. **Formulieren der Typdefinitionen**
   In diesem Schritt wird der Inhalt aller Typdefinitionen aus dem Modell ausgelesen und die Syntax für die bereits angelegten Typdefinitionen im XML-Schema vervollständigt.

5. **Anlegen der Root-Elemente**
   Für jedes Root-Element wird die entsprechende XML-Schema-Syntax erzeugt. Sie besteht aus dem Namen und einer Referenz auf eine Typdefinition.

6.4 Import: XML-Schema → EMX-Modell

Auch der Import ist ein zweistufiger Prozess. Er besteht aus den folgenden Phasen:

- Vereinfachungsphase
- Übersetzungsphase

6.4.1 Vereinfachungsphase

Auf Grund des Umfanges des XML-Schema-Standards werden bestimmte Bedingungen an die XML-Schema-Dateien gestellt, damit sie sich importieren lassen. Es ist möglich beliebige XML-Schema-Dateien so umzuwandeln, dass sie die geforderten Bedingungen erfüllen. Für die Implementation wird vorausgesetzt, dass eine entsprechend angepasste XML-Schema-Datei vorliegt.

Korrektur des Schema-Designs

Die verschiedenen Schema-Designs unterscheiden sich vor allem bezüglich der Sichtbarkeit und Wiederverwendbarkeit der Element- und Typdefinitionen. In XML-Schema [XS001] [XS101] [XS201] können Typ- und Elementdefinitionen sowohl global als auch lokal definiert werden. *Global* bedeutet hier, dass die Definition direkter Nachfolger des `<schema>`-Knotens ist. *Lokale* Komponenten sind innerhalb anderer Komponenten (wie z.B. `<complexType>`) geschachtelt. Die Entscheidung über lokale oder globale Definition hat insbesondere Einfluss auf die Wiederverwendungsmöglichkeiten. In abgeleiteten Schemas kann nur auf globale Komponenten zugegriffen werden.

Die vier Schema-Designs *Russian Doll*, *Salami Slice*, *Venetian Blind* und *Garden of Eden* werden von Costello in [Cos05, Abschnitt: „Global versus Local“] und Maler in [Mal02] ausführlich vorgestellt. Sie beschreiben verschiedene Konzepte, wie XML-Schemas aus lokalen und globalen Komponentendefinitionen aufgebaut werden können und die damit verbundenen Auswirkungen auf die Eigenschaften des XML-Schemas. Die Änderung des Schema-Designs hat keinen Einfluss auf die Menge der akzeptierten Dokumente.

Das EMX-Modell basiert auf dem *Venetian-Blind* Design. Das bedeutet, dass alle Typdefinitionen global definiert sind. Alle Elementdeklarationen, bis auf die Root-Elemente, sind lokal. *Venetian Blind* lässt also die Wiederverwendung aller Typdefinitionen zu.

Bevor das XML-Schema in ein EMX-Modell überführt werden kann, muss das Schema-Design überprüft werden. Liegt das Schema nicht in *Venetian Blind* vor, muss es angepasst werden. Das bedeutet im Wesentlichen die Auflösung anonymer Typdefinitionen (durch Erzeugen eines neuen globalen Typs und Anlegen einer Referenz darauf). Außerdem müssen Element-Element-Beziehungen (`ref`-Konstrukt) in entsprechende Element-Typdefinition-Beziehungen umgewandelt werden.

Weitere Modifikationen

Der XML-Schema-Standard enthält weitere Konstrukte (z.B. Schlüssel-Fremdschlüssel-Beziehungen), die im Rahmen der prototypischen Umsetzung des Modell-Editors nicht behandelt werden. Diese müssen ebenfalls falls möglich umgewandelt oder entfernt werden.

Namespaces könnte man durch Integration des Namespace-Attributs in den Namen der Typ- oder Elementdeklaration auflösen. Allerdings werden dadurch auch Anpassungen an den, durch das Schema beschriebenen, XML-Dokumenten notwendig.

Weiterhin kann XML-Schema neue Typen aus vorhandenen Typen durch Erweiterung oder Restriktion ableiten. Dieses Konzept lässt sich auflösen, indem die Typen eine neue, vollständige Typdefinition erhalten, in welcher die Typbeschreibung des Parent-Typs wiederholt wird.

6.4.2 Übersetzung

Wenn das gegebene XML-Schema alle genannten Bedingungen erfüllt, lassen sich sämtliche XML-Schema-Konstrukte durch EMX-Entities und ihre Verbindungen darstellen. Dazu wird das vereinfachte Schema eingelesen und die entsprechenden Modellobjekte erzeugt. Zur Anwendung kommt die Umkehrung des Mappings aus dem EMX-Modell-Export. Für die Details der Übersetzung einzelner XML-Schema-Konstrukte in das EMX-Modell sei deshalb wieder auf Tabelle 6.1 verwiesen. Die Reihenfolge der Übersetzung der einzelnen Konstrukte entspricht auch der im Abschnitt 6.3.2 beschriebenen Reihenfolge.

Kapitel 7

Programmbeschreibung

7.1 EMX und Eclipse

7.1.1 Was ist Eclipse?

Eclipse ist eine IDE (Integrated Development Environment). Eine *IDE* fasst verschiedene Werkzeuge und Tools zur Entwicklung von Software unter einer einheitlichen Oberfläche zusammen. Ziel von Eclipse ist die Bereitstellung *eines* Werkzeuges, mit dem der Entwickler möglichst viele seiner täglichen Arbeiten durchführen kann.

Zu den Grundfunktionen und Features einer IDE zählen u.a.:

- Projektverwaltung
  (Zusammenfassen unterschiedlicher Source-Dateien zu einem Projekt)
- Syntax-Hervorhebung und -Überprüfung
- Code-Vervollständigung und Code-Vorlagen
- (kontextsensitive) Hilfe
- Anbindung an verschiedene Source-Code-Repositories (CVS, ...)
- Unterstützung beim Zugriff auf Datenbanken
- ...

Die Grundfunktionen einer IDE sind oft ähnlich. Lediglich in den sprachspezifischen Ausprägungen gibt es Unterschiede.

Eclipse ist eine Meta-IDE. Es bietet ein Framework zur Implementierung sprachunabhängiger Werkzeuge. Besonderer Fokus liegt auf der Erweiterbarkeit sämtlicher Funktionalität. Eclipse ist eine „offene Plattform zur Integration von Werkzeugen“. Es wurde als Java-Anwendung entwickelt und steht somit unter verschiedenen Betriebssystemem zur Verfügung.

Die Benutzeroberfläche basiert auf *SWT* - einem im gleichen Zeitraum wie Eclipse entstandenen Open-Source-Projekt, mit dem Ziel die Java GUI-Bibliotheken AWT und Swing zu ersetzen.

Die Eclipse-Architektur besteht aus einem kleinen sprachunabhängigen Kern, der durch Plugins erweitert wird. Auch die den meisten Anwendern bekannte Eclipse-Java-Entwicklungsumgebung ist „nur“ ein Serie von Plugins.

Eclipse ist OpenSource. Eclipse sowie die meisten Plugins sind als Open-Source unter der Common Public Licence (CPL) veröffentlicht. Somit ist die Entwicklung und Weitergabe sowohl als Opensource- als auch als kommerzielle Software erlaubt. Die Idee für Eclipse entstand bei IBM; im Jahre 2001 stellte IBM den Eclipse-Source-Code frei. Weitere Unternehmen (z.B. Borland, Nokia, Intel, Sybase, BEA, ...) beteiligen sich an der Weiterentwicklung und nutzen Eclipse als Grundlage für eigene kommerzielle Projekte.

7.1.2 Was sind Plugins?

Plugins sind der Erweiterungsmechanismus von Eclipse. Bis auf einen kleinen Kern wird sämtliche Funktionalität in Eclipse durch Plugins bereitgestellt.

Für Eclipse wurde eine Reihe von Erweiterungspunkten (*engl.* extension points) definiert. Neue Plugins können an ihnen die Eclipse-Umgebung an wohldefinierten Stellen erweitern, indem sie u. a. neue Menüeinträge erzeugen oder neue Editoren registrieren. Für die Registrierung stellt jedes Plugin eine Datei `plugin.xml` bereit. Anhand dieser Datei kann Eclipse die Funktionalität des Plugins erkennen und integrieren. Die Installation eines neuen Plugins ist einfach. In einem neuen Verzeichnis unter `eclipse\plugins` wird der ausführbare Code, zusätzlich benötigte Dateien (z.B. Icons) sowie die Datei `plugin.xml` abgelegt. Beim Start von Eclipse wird das Plugin dann automatisch erkannt und integriert.

7.1.3 Der EMX-Editor als Eclipse-Plugin

Der EMX-Editor passt zu Eclipse. Die implementierten Funktionen lassen sich vom Thema her gut in Eclipse integrieren. Das Erstellen und Bearbeiten von XML-Schemas ist eine Aufgabe, die durch IDEs unterstützt wird. So gibt es unter Eclipse schon einen XML-Schema-Editor im Eclipse-Webtools-Project (`http://www.eclipse.org/webtools/`). Auch XMLSpy (`http://www.altova.com/products_ide.html`), der bekannte XML-Editor, lässt sich in Eclipse integrieren.

Nutzen der Eclipse-Funktionalität. Mit Eclipse bekommt der Programmierer ein umfangreiches, stabiles und gut getestetes Fundament. Mit Eclipse als Basis lassen sich viele der Aufgaben schnell und elegant lösen. Es existieren Plugins für das Erstellen modellbasierter graphischer Editoren und eine API zum Auslesen und Generieren von XML-Schemas. Mit dem in Eclipse integrierten XML-Editor lässt sich ein aus dem EMX-Modell exportiertes XML-Schema gleich (mit Syntax-Highlighting) anzeigen. Um die Korrektheit des Schemas zu überprüfen, stellt IBM die Testversion eines Schema-Quality-Checkers im Web unter `http://www.alphaworks.ibm.com/tech/xmlsqc` bereit. Das Program lässt sich ebenfalls als Plugin in Eclipse integrieren und kann zur Validierung des generierten Schemas verwendet werden.

Auch als Standalone-Version nutzbar. Möchte der Anwender ausschließlich mit dem EMX-Editor arbeiten, installiert er lediglich eine Minimal-Version von Eclipse und integriert darin das EMX-Plugin sowie einige weitere (vom EMX-Plugin benötigte) externe Plugins. So erhält er ein schlankes Programm, welches auf die zusätzlichen Funktionalität der Eclipse-IDE verzichtet.

7.2 Externe Bibliotheken (APIs)

In diesem Abschnitt werden die für das EMX-Plugin benötigten externen Bibliotheken und Plugins kurz vorgestellt.

7.2.1 GEF

Es lag nahe, das Graphical Editing Framework (GEF), welches im Rahmen des Eclipse-Projektes verfügbar ist, zu verwenden. Die Komponente bietet alle Features, die für die Entwicklung eines graphischen Editors benötigt werden. Sie ist als Open-Source frei verfügbar und es existiert eine Vielzahl von Artikeln, die die zugrunde liegenden Konzepte beschreiben. Falls nötig kann man via Newsgroups auch direkt mit den Entwicklern in Kontakt treten, um Probleme zu klären.

7.2.2 XSD

Das Programm muss an den verschiedensten Stellen XML-Schema-Dateien einlesen oder generieren. Dies könnte trivialer Weise auf der Basis des Document Object Modell (DOM) der XML-Schema-Dateien erfolgen. Dazu würde das XML-Schema in ein DOM-Modell eingelesen werden. Das erzeugte Modell könnte modifiziert und anschließend wieder in eine XML-Schema-Datei serialisiert werden. Allerdings würde man mit der Verwendung des DOM lediglich eine korrekte XML-Syntax sicherstellen können. Zur Behandlung spezifischer Eigenschaften von XML-Schema ist das Modell nicht geeignet.

Mit dem **XML Schema Infoset Modell (XSD)** steht als Teil des Eclipse-Projektes ein Plugin zur Verfügung, das eine API für das Erstellen und Manipulieren von XML-Schemas und XML-Schema-Fragmenten bereitstellt. Die Open-Source-Bibliothek enthält auch Methoden zum Einlesen und Schreiben von XML-Schema-Dateien. Die Korrektheit von Typdefinitionen (z.B. korrekte Verwendung von Facets) kann ebenfalls überprüft werden. Das Ziel des XSD-Projektes ist es, den XML-Schema-Standard [XS001] vollständig zu unterstützen.

Die URL der Homepage des Projektes lautet: `http://www.eclipse.org/xsd/`. Dort finden man die benötigten Dateien zum Download. Allerdings ist die Dokumentation nicht sehr umfangreich. Die Funktionalität muss zum größten Teil aus der JavaDoc erschlossen werden. Die JavaDoc für die Klasse `org.eclipse.xsd.util.XSDPrototypicalSchema` sei als Lektüre zum Einstieg empfohlen. Darin enthalten sind Code-Beispiele für das Erzeugen, Einlesen und Ausgeben eines Schemas. Weiterhin wird beschrieben, wie das in der XML-Schema-Beschreibung [XS001] eingeführte `PurchaseOrderSchema` mit Hilfe der API erstellt werden kann. Somit verfügt man über ausreichend Code-Beispiele, um eigene Schemas erstellen zu können.

7.3 Ausgewählte Konzepte

In diesem Abschnitt werden einige Konzepte kurz vorgestellt, um den Einstieg für nachfolgende Projekte zu erleichtern.

7.3.1 Anwendung des Graphical Editor Frameworks

GEF basiert auf der Model-View-Controller Architektur. Das bedeutet, dass es ein Datenmodell und eine Sicht gibt, die streng voneinander getrennt sind. Der Controller vermittelt zwischen beiden. Er steuert den Abbildungsprozess vom Modell zur Sicht und muss Aktionen auf der Sicht-Ebene in Änderungsanweisungen für das Modell übersetzen. Zum Erlernen der grundlegenden Konzepte sei der Artikel „Creating an Eclipse-based application using the Graphical Editing Framework - How to get startet with GEF“ von der IBM Developerworks Website empfohlen (`http://www-128.ibm.com/developerworks/opensource/library/os-gef/`).

Der EMX-Editor basiert auf dem Shapes Diagram Editor, der in dem Artikel: „A Shapes Diagram Editor“ (`http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html`) beschrieben wird. Quellcode und lauffähige Version als Eclipse-Plugin befinden sich in den GEF-Beispielen, die unter `http://download.eclipse.org/tools/gef/downloads/` in der aktuellen Version heruntergeladen werden können. Der Quellcode des Shapes Diagram Editors diente als Ausgangsbasis und wurde schrittweise in den EMX-Editor umgewandelt und erweitert.

7.3.2 Hinzufügen eines neuen Modellobjektes

An dieser Stelle wird die Vorgehensweise dargestellt, den Editor um ein neues Objekt zu erweitern. Kenntnisse über die Konzepte des GEFs werden hier vorausgesetzt. An dieser Stelle kann nur eine grobe Einführung gegeben werden. Es empfiehlt sich immer, einen Blick auf die bereits implementierten Klassen zu werfen.

Icon Zuerst wird das Icon, eine GIF-Datei, mit einem Grafikprogramm erzeugt. Es muss in zwei Größe (16x16 und 24x24 Pixel) erstellt werden. Eclipse verwendet abhängig vom Betriebssystem mal die eine und mal die andere Größe. Das Icon wird sowohl in der Werkzeug-Palette des Editors als auch als Symbol im Header der graphischen Notation angezeigt. Die Dateien werden im Projekt unter `src/emx/icons` abgelegt.

Modellobjekt Als nächstes wird eine Klasse erstellt, die das Datenmodell des Objektes darstellt. Sie muss von der Klasse `emx.model.EmxEntity` abgeleitet werden und sollte auch im gleichen Package abgelegt werden. Die Modellklasse besteht hauptsächlich aus Properties und den dazugehörenden Setter- und Getter-Methoden.

View Es wird eine weitere Klasse erstellt, die von `org.eclipse.draw2d.Figure` abgeleitet wird. Sie enthält u.a. eine Methode `paintFigure()`, die das Zeichen des Elementes übernimmt. In der Klasse wird beschrieben, wie bestimmte Eigenschaften des Elementes graphisch dargestellt werden. Außerdem enthält sie Methoden, die zur Aktualisierung der gezeichneten Eigenschaften dienen. Die Klasse wird im Package `emx.view.figures` abgelegt.

PropertySources Eigenschaften, die nicht direkt in der Grafik angezeigt werden, lassen sich in einem separatem Fenster, dem Properties View, darstellen und modifizieren. Die Aufbereitung der Anzeige der Daten und die Aktualisierung des Modells nach Änderung von Werten im PropertiesView übernimmt eine Klasse, die von `emx.view.propertysources.EmxModelObjectPropertySource` abgeleitet werden muss. Sie sollte im selben Package gespeichert werden. Die Einträge im Properties View werden durch Eclipse automatisch nach dem Alphabet sortiert.

Controller Die Verwaltung und Aktualisierung von Modell- und View-Klasse übernimmt der Controller. Die Klasse wird von `org.eclipse.gef.editparts.AbstractGraphicalEditPart` abgeleitet und im Package `emx.view.editparts` abgelegt. Um Informationen über Änderungen des Modell zu erhalten, ist er über das `Listener`-Konzept mit dem Modell verbunden.

Zusammenwirken Damit Modell, View und Controller zusammenarbeiten können, sind noch einige weitere Klassen zu ergänzen. In der Klasse `emx.view.editparts.EntityEditPartFactory` werden die EditParts (Controller) erzeugt. Die Methode `getPartForElement(...)` muss um einen neuen Eintrag mit Modellklasse und dazugehörigem EditPart erweitert werden.

In `emx.view.editparts.ModelGraphEditPart` muss das Modell noch in den Methoden `getCreateCommand()` und `createChangeConstraintCommand()` angemeldet werden. In der Modellklasse (Package: `emx.model`) muss in der Methode `getAdapter()` die entsprechende PropertySource-Klasse zurückgegeben werden.

In der EditPart-Klasse muss in der Methode `createFigure()` die entsprechende View-Klasse zurückgegeben werden.

Palette Damit das Element später auch aus der Palette des Editors ausgewählt werden kann, muss ein entsprechender Eintrag in der Methode `createEntityDrawer` Klasse `emx.EmxModelEditorPaletteFactory` erzeugt werden.

Logging Damit auch das Logging funktioniert, muss in der Klasse `emx.logging.EmxModelLog` die Methode `entityCreated()` erweitert werden. Beim Erzeugen eines neuen Objektes soll ein entsprechender Log-Eintrag angelegt werden. Außerdem muss das XML-Schema für die Log-Datei angepasst werden.

7.3.3 Die Datei `plugin.xml`

Die XML-Datei `plugin.xml` bildet die Basis für die Erweiterung der Eclipse IDE. Durch sie wird das Plugin in der Eclipse IDE registriert.

Sie enthält eine Übersicht über die Plugins, die zusätzlich installiert sein müssen, damit das EMX-Plugin gestartet werden kann (Plugin-Abhängigkeiten). Alle Erweiterungen der IDE, wie zum Beispiel neue Menü-Einträge für Kontextmenüs, zusätzliche Wizards (mehrseitige Dialoge zu Steuerung einfacher Aufgaben), Erweiterungen der Eclipse-Einstellungen oder neue Editoren werden hier in sogenannten *extension points* registriert.

```
<extension point="org.eclipse.ui.editors">
   <editor
      name="Emx Model"
      extensions="emx"
      icon="icons/emx.gif"
      default="true"
      class="emx.EmxModelEditor"
      contributorClass
         ="emx.EmxModelEditorActionBarContributor"
      id="emx.EmxModelEditor">
   </editor>
</extension>
```

Abbildung 7.1: Ausschnitt aus `plugin.xml` zur Registrierung des EMX-Editors

Abbildung 7.1 zeigt einen Ausschnitt aus der Datei. Es werden die Einstellungen für den EMX-Editor dargestellt. Neben dem Namen und einer ID wird angegeben, welche Dateitypen sich mit dem Editor öffnen lassen, welches Icon angezeigt wird und in welcher Klasse sich der Code für den Editor befindet.

Für den Einstieg in die Arbeit mit Plugins eignet sich der Artikel **„Your First Plug-in“** (`http://www.eclipse.org/articles/Article-Your%20First%20Plug-in/YourFirstPlugin.html`)

7.3.4 Marker

Eclipse enthält einen zentralen Mechanismus *„Marker“*, um mit dem Anwender zu kommunizieren. Durch Marker wird er auf Probleme oder noch auszuführende Aufgaben hingewiesen. Marker können aber auch nur als „Lesezeichen“ für eine bestimmte Stelle in einem Programm gesetzt werden. Durch Anklicken des Markes wird ein Editor mit der betroffenen Datei geöffnet und die entsprechende Stelle markiert.

Das EMX-Plugin nutzt Marker vor allem, um Fehler im Modell, wie zum Beispiel nicht-gestattete Verbindungen, dem Nutzer anzuzeigen. Die vom EMX-Plugin verwendeten Marker werden als Eintrag im *Problems View* (einem von Eclipse bereitgestellten Fenster) angezeigt.

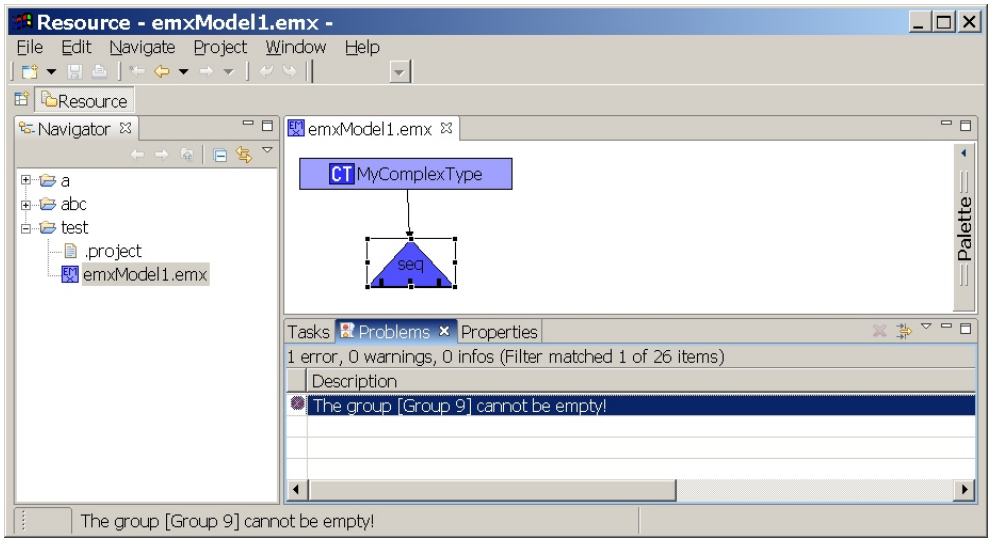


Abbildung 7.2: Screenshot: Anzeige eines Fehlers im Problems View

Abbildung 7.2 zeigt einen fehlerhaften Gruppenoperator.

Mit dem Code in Abbildung 7.3, lässt sich so ein Marker einfach erzeugen. Es muss lediglich der Typ des Markers, die anzuzeigende Nachricht, die Datei, auf die sich der Marker bezieht, und das Objekt, welches innerhalb der Datei markiert werden soll, angegeben werden.

Alle weiteren Aufgaben (Anzeige im Problems View, Verhalten beim Anklicken, ...) werden durch Eclipse übernommen.

Das Marker-Konzept wird in dem Artikel **„Mark My Words“** (`http://www.eclipse.org/articles/Article-Mark%20My%20Words/mark-my-words.html`) vorgestellt.

```
private static void createMarkerForBadRootEntity(EmxEntity e) {
   //the EMX-File
   IResource resource = e.getModel().getFile();
   try {
      IMarker marker = resource
            .createMarker("org.eclipse.core.resources.problemmarker");
      marker.setAttribute(IMarker.SEVERITY, IMarker.SEVERITY_ERROR);
      marker.setAttribute(IMarker.MESSAGE, "[" + e.getDescriptiveID()
            + "] cannot be a root entity!");
      marker.setAttribute("userEditable", false);
      marker.setAttribute(EmxModelEditor.MARKERATTRIBUTE_OBJECTID,
            e.getDescriptiveID());
   } catch (CoreException ce) {
      ce.printStackTrace();
   }
}
```

Abbildung 7.3: Codebeispiel: Erzeugen eines Markers

7.3.5 Wizards

Wizards sind ein häufig verwendetes Konzept in Eclipse. Wizards werden benutzt, um neue Ressourcen, wie Dateien und Ordner zu erzeugen. Sie bestehen in der Regel aus mehrseitigen Dialogen, in welchen zunächst Informationen gesammelt werden. Mit diesen Informationen werden danach die neuen Objekte erzeugt.

Für das EMX-Plugin wurde ein Wizard für das Erzeugen einer neuen Modelldatei implementiert. Aufgerufen wird der Wizard über das `File->New->Other...`-Menü. Es öffnet sich ein Dialog, der alle durch Eclipse erzeugbaren Objekte anzeigt. Nach Auswahl von `EMX Model` wird der Wizard gestartet. Die für das Modell notwendigen Informationen (Dateiname und Ordner) werden gesammelt. Im Anschluss wird die Datei erzeugt und der Editor geöffnet.

Weitere Wizards existieren für den Import und Export von XML-Schema-Dateien. Sie werden über das Kontextmenü einer `*.xsd`- oder `*.emx`-Datei gestartet. Sämtliche Wizards werden über *extension points* in der Datei `plugin.xml` registriert.

Einen umfangreicheren Überblick gibt der Artikel **„Creating JFace Wizards“**(`http://www.eclipse.org/articles Article-JFace%20Wizards/wizardArticle.html`).

7.3.6 Properties

Für allgemeine Einstellungen kann der *Properties Dialog* von Eclipse durch zusätzliche Seiten erweiterter werden. Diese Seiten werden auch in der Datei `plugin.xml` registriert. Das EMX-Plugin nutzt die Properties zur Einstellung von Farben der Modellobjekte und zur Verwaltung der Liste der an-

gezeigten XML-Schema-Datentypen. Für die Verwaltung der Einstellungen (Laden, Speichern, Default-Werte) stellt Eclipse einen sogenannten *Preference Store* für jedes Plugin zur Verfügung.

Weitere Details entnehme man dem Artikel **„Preferences in the Eclipse Workbench UI“**(`http://www.eclipse.org/articles/Article-Preferences/preferences.htm`).

7.4 UML-Diagramm der Modellobjekte

In diesem Abschnitt soll die Klassenhierarchie und die Beziehungen zwischen den Objekten des EMX-Modells dargestellt werden. Die Klassen, die die Modellobjekte repräsentieren, befinden sich im Package `emx.model`.

Klassenhierarchie
Abbildung 7.4 zeigt die Klassenhierarchie der Modellobjekte. Basisklasse ist `EmxModelObject`. Davon abgeleitet werden die Klassen `EmxModelGraph`, `EmxConnection` und `EmxEntity`. `EmxEntity` bildet die Basis für alle Entity-Klassen.

Zusätzlich zur Vererbungshierarchie kann man der Abbildung die für jede Klasse wichtigsten Properties entnehmen. Weiterhin werden die Inhalte der Aufzählungstypen (*engl.* enumerations) dargestellt.

Beziehungen
Die Beziehungen zwischen den Modellobjekten zeigt Abbildung 7.5.

Die Klasse `EmxModel` repräsentiert das Modell. Es enthält eine Liste von Verbindungen (Klasse: `EmxConnection`) und eine Referenz auf einen Graphen (Klasse: `EmxModelGraph`).

Im Gegensatz zu den aus der theoretischen Informatik bekannten *Graphen*, enthält dieser Graph nur eine Liste von Knoten (Unterklassen von `EmxEntity`). Bei einer Erweiterung könnte es notwendig sein, dass ein Modell mehrere Graphen enthält. Dadurch könnte man den Inhalt von Modulen ebenfalls als Graph (anstelle einer Liste von Typdefinitionen) darstellen. Diese Darstellungsweise führt zu Verbindungen zwischen Knoten verschiedener Graphen, und deshalb werden die Verbindungen nicht einem einzelnen Graphen, sondern direkt dem Modell zugeordnet.

Auch zwischen den Entity-Klassen existieren Beziehungen. Die Attributbox (Klasse `EmxAttributeBox`) verwaltet eine Liste von Attributen (Klasse: `EmxAttribute`). Jedem Modul (Klasse: `EmxModule`) werden die darin enthaltenen Element- und Typdefinitionen als `EmxExternalEntity` zugeordnet. Weiterhin speichert jeder Gruppenoperator (Klasse: `EmxGroup`) eine Liste ausgehender Verbindungen, um daraus die Reihenfolge der Subelemente und Subgruppen ableiten zu können.

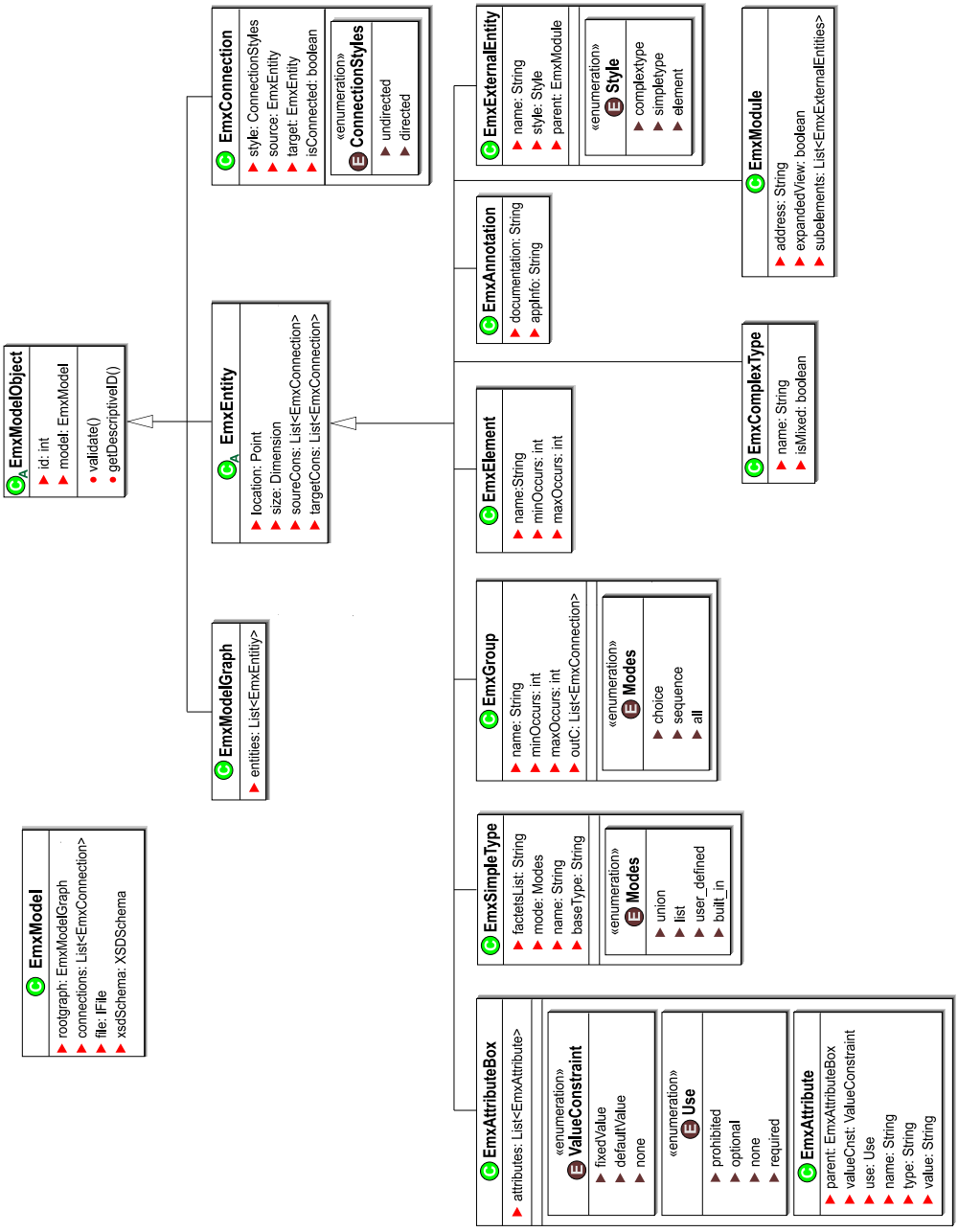


Abbildung 7.4: Vererbungsdiagramm für Modellobjekte

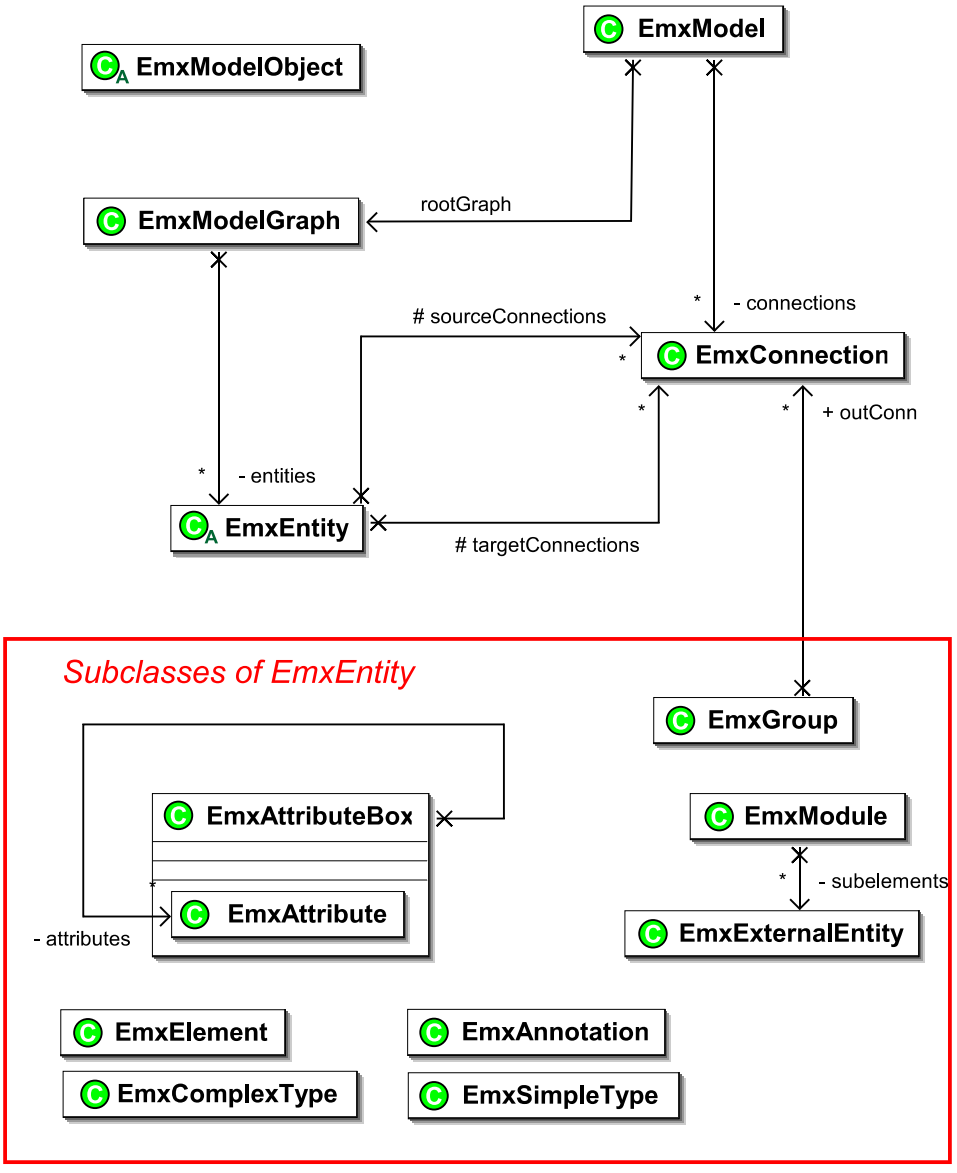


Abbildung 7.5: Beziehungsdiagramm für Modellobjekte

Kapitel 8

Fazit und Ausblick

Im Rahmen dieser Diplomarbeit wurde ein konzeptuelles Modell für XML-Schema entwickelt - das EMX-Modell.

Zuerst wurden verschiedene Schemasprachen (DTD, XML-Schema, RELAX NG) mit dem Ziel untersucht, eine geeignete Zielsprache für das EMX-Modell zu finden. Auf Grund des größeren Verbreitungsgrades gegenüber RELAX NG und der Mächtigkeit gegenüber DTDs wurde XML-Schema ausgewählt.

In einem weiteren Abschnitt wurden verschieden Anforderungen an konzeptuelle Modelle herausgearbeitet und untersucht, inwieweit sie auf das EMX-Modell angewendet werden können. Es erwies sich dabei als unmöglich, alle Anforderungen gleich gut zu unterstützen. Deshalb wurden Schwerpunkte für das EMX-Modell gesetzt.

Weiterhin wurden in der Arbeit verschiedene existierende Modelle für XML vorgestellt. Dazu wurden die Kategorien physische, formale und konzeptuelle Modelle eingeführt und für jede Kategorie Beispiele vorgestellt. Viele der existierende Ansätze zur Entwicklung konzeptueller Modelle basieren auf dem ER-Modell oder auf UML. Es wurden Modelle gefunden, die wichtige Konzepte des XML-Schema-Standards nicht umsetzten. Andere Modelle waren wiederum zu detailiert und damit für Laien zunächst unverständlich.

Mit dem EMX-Modell wurde ein einfaches, aber dennoch mächtiges konzeptuelles Modell für XML-Schema entwickelt. Es basiert auf wenigen Basiskonstrukten (Entities, Verbindungen und Eigenschaften). Für einzelne XML-Schema-Konstrukte wurden spezielle Entities abgeleitet. Weiterhin wurde untersucht, wie die Übersetzungsprozesse EMX-Modell → XML-Schema und XML-Schema → EMX-Modell aussehen. Dabei erwies es sich als hilfreich, die Übersetzung in zwei Schritten vorzunehmen. Beim Import eines XML-Schemas werden in einem Vorbereitungsschritt zuerst die nicht abbildbaren Konstrukte durch semantisch äquivalente Konstrukte ersetzt, bevor der eigentliche Import-Prozess gestartet wird. Beim Export in ein XML-

Schema wird das EMX-Modell im ersten Schritt auf Fehler überprüft, die durch den Nutzer oder automatisch berichtigt werden. Dadurch lassen sich in der eigentlichen Import- bzw. Exportphase die XML-Schema-Konstrukte eineindeutig in EMX-Modell-Konstrukte umwandeln.

Um den Nutzer bei der Arbeit am EMX-Modell zu unterstützen wurde ein graphischer Editor implementiert. Dazu musste zunächst eine geeignete Java-Repräsentation für das Modell gefunden werden. Der Editor beherrscht den visuellen Entwurf des Modells und führt den Nutzer durch den Import- und Exportprozess. Der EMX-Editor wurde in die Eclipse-IDE integriert und macht umfangreichen Gebrauch von den durch Eclipse bereitgestellten Funktionen. Davor stand jedoch eine umfangreiche Einarbeitung in die verschiedenen APIs.

Die zusätzlichen unter Eclipse verfügbaren Anwendungen (XML-Editor, XML-Schema-Editor, IBM's Schema Quality Checker) arbeiten nahtlos mit dem EMX-Editor zusammen. Vor allem lassen sich die im- und exportierten XML-Schemas sofort ansehen und bearbeiten. Der Anwender kann so umfassend bei der Erstellung von XMl-Applikationen unterstützt werden.

Ideen für die Weiterführung des Projektes

- Auf Grund der Komplexität des XML-Schema-Standards wurden bei der prototypischen Implementierung einige Aspekte noch nicht umgesetzt. Vor allem die Fähigkeit XML-Schemas Typdefinionen durch Vererbung abzuleiten sollte durch das Modell und den Editor unterstützt werden. Die Umsetzungen von Eindeutigkeitsbeschränkung und Schlüssel-Fremdschlüsselkonzept wären ebenfalls sinnvolle Erweiterungen des EMX-Modells.

- Der Vorübersetzungsprozess (Auflösung unbekannter XML-Schema-Konstrukte und Transformation in das „Venetian Blind"-Schemadesign) sollte automatisiert werden, damit der Import eines beliebigen Schemas ohne manuelle Bearbeitung möglich ist.

- Es könnte ebenfalls, darüber nachgedacht werden, ob alternativ eine graphische Notation in UML entwickelt wird. Mit den Erweiterungen, die UML bietet, sollte es möglich sein, für die einzelnen Bestandteile des EMX-Modells entsprechende UML-Darstellungen zu entwickeln. Dabei sollte vor allem auf Einfachheit geachtet werden, denn die existierenden UML-Modelle für XML-Schema sind oft viel zu komplex und schwer zu lesen.

- Man könnte untersuchen, inwieweit externe Bibliotheken mit XML-Schema-Definitionen zur Wiederverwendung integriert werden können. Dafür ist das bereits implementierte Modulkonzept nur geringfügig zu erweitern.

- Es sollte untersucht werden, ob weitere Schemasprachen integriert werden können. Die Ableitung einer DTD (mit Informationsverlust) ist sicherlich problemlos realisierbar. Die Implementation des Import und Export von RELAX NG ist komplexer, da es keine Unterstützung durch eine API (siehe XSD-API für XML-Schema) gibt. Außerdem müsste zur Definition einfacher Datentypen auf XML-Schema zurückgegriffen werden, da RELAX NG keine eigene Typbibliothek besitzt.
- Die Schemaevolutionskomponente aus der Studienarbeit von Marcus Oertel, die mit den erstellten Log-Dateien arbeitet, könnte ebenfalls in die Eclipse-Umgebung integriert werden. Dadurch ließen sich das durch Schemaevolution veränderte XML-Schema und das XML-Schema, welches aus dem modifizierten EMX-Modell erzeugt wurde, einfach vergleichen.

Literaturverzeichnis

[ACM98] Abiteboul, Serge ; Cluet, Sophie ; Milo, Tova: A Logical View of Structured Files. In: *VLDB J.* 7 (1998), Nr. 2, 96-114. http://www.springerlink.com/openurl.asp?genre=article&issn=1066-8888&volume=7&issue=2&spage=96

[Alh98] Alhir, Sinan S.: *UML in a Nutshell: a desktop quick reference*. O'Reilly & Associates, Inc. http://www.oreilly.com/catalog/umlnut/. – ISBN 1–56592–448–7

[ASD03] Arijit Sengupta, Sriram M. ; Doshi, Rahul: XER - Extensible Entity Relationship Modeling. In: Tolksdorf, Robert (Hrsg.) ; Eckstein, Rainer (Hrsg.): *XML Conference & Exposition 2003*

[BGP00] Buck, Lee (Hrsg.) ; Goldfarb, Charles F. (Hrsg.) ; Prescod, Paul (Hrsg.): *Datatypes for DTDs (DT4DTD) 1.0 W3C Note*. http://www.w3.org/TR/2000/NOTE-dt4dtd-20000113. Version: 13 January 2000

[BKK04a] Bernauer, Martin ; Kappel, Gerti ; Kramler, Gerhard: Representing XML Schema in UML - A Comparison of Approaches. In: Koch, Nora (Hrsg.) ; Fraternali, Piero (Hrsg.) ; Wirsing, Martin (Hrsg.): *ICWE* Bd. 3140, Springer. – ISBN 3–540–22511–0, 440-444

[BKK04b] Bernauer, Martin ; Kappel, Gerti ; Kramler, Gerhard: Representing XML Schema in UML - An UML Profile for XML Schema. (2004). http://www.big.tuwien.ac.at/research/publications/2003/1303.pdf

[BMNS05] Bex, Geert J. ; Martens, Wim ; Neven, Frank ; Schwentick, Thomas: Expressiveness of XSDs: from practice to theory, there and back again. In: Ellis, Allan (Hrsg.) ; Hagino, Tatsuya (Hrsg.): *WWW*, ACM. – ISBN 1–59593–046–9, 712-721

[BT99] Beeri, Catriel ; Tzaban, Yariv: SAL: An Algebra for Semistructured Data and XML. In: *WebDB (Informal Proceedings)*, 37-42

[CDG+97] Comon, H. ; Dauchet, M. ; Gilleron, R. ; Jacquemard, F. ; Lugiez, D. ; Tison, S. ; Tommasi, M.: *Tree Automata Techniques and Applications*. Available on: http://www.grappa.univ-lille3.fr/tata, 1997. – release October, 1rst 2002

[Che76] Chen, Peter P.: The Entity-Relationship Model - Toward a Unified View of Data. In: *ACM Trans. Database Syst.* 1 (1976), Nr. 1, 9-36. http://bit.csc.lsu.edu/~chen/pdf/erd.pdf

[Cos05] Costello, Roger L.: XML Schemas: Best Practices. (2005). http://www.xfront.com/BestPracticesHomepage.html

[CSF00] Conrad, Rainer ; Scheffner, Dieter ; Freytag, Johann C.: XML Conceptual Modeling Using UML. In: *ER*, 558-571

[DFH99] Davidson, Andrew (Hrsg.) ; Fuchs, Matthew (Hrsg.) ; Hedin, Mette (Hrsg.) ; Jain, Mudita (Hrsg.) ; Koistinen, Jari (Hrsg.) ; Lloyd, Chris (Hrsg.) ; Maloney, Murray (Hrsg.) ; Schwarzhof, Kelly (Hrsg.): *Schema for Object-Oriented XML 2.0 W3C Note*. http://www.w3.org/1999/07/NOTE-SOX-19990730. Version: 30 July 1999

[DOM00] Apparao, Vidur (Hrsg.) ; Byrne, Steve (Hrsg.) ; Champion, Mike (Hrsg.) ; Isaacs, Scott (Hrsg.) ; Jacobs, Ian (Hrsg.) ; Hors, Arnaud L. (Hrsg.) ; Nicol, Gavin (Hrsg.) ; Robie, Jonathan (Hrsg.) ; Sutor, Robert (Hrsg.) ; Wilson, Chris (Hrsg.) ; Wood, Lauren (Hrsg.): *Document Object Model (DOM) Level 1 Specification Version 1.0 W3C Recommendation*. http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001. Version: 1 October 2000

[HS00] Heuer, Andreas ; Saake, Gunter: *Datenbanken: Konzepte und Sprachen*. 2. Auflage. MITP-Verlag, 2000. – ISBN 3–8266–0619–1

[Int86] International Organization for Standardization: *ISO 8879:1986: Information processing — Text and office systems — Standard Generalized Markup Language (SGML)*. Geneva, Switzerland : International Organization for Standardization, 1986. – 155 S

[KK03] Kudrass, Thomas ; Krumbein, Tobias: Rule-Based Generation of XML DTDs from UML Class Diagrams. In: Kalinichenko, Leonid A. (Hrsg.) ; Manthey, Rainer (Hrsg.) ; Thalheim, Bernhard (Hrsg.) ; Wloka, Uwe (Hrsg.): *ADBIS* Bd. 2798, Springer. – ISBN 3–540–20047–9, 339-354

[KKB05] Kastens, Uwe ; Kleine-Büning, Hans: *Modellierung Grundlagen und formale Methoden*. Hanser, 2005. – 256 S. – ISBN 3–44640–460–0

[KL02] Kleiner, Carsten ; Lipeck, Udo W.: Automatische Erzeugung von XML-DTDs aus konzeptionellen Datenbankschemata. In: *Datenbank-Spektrum* 2 (2002), Nr. 2, 14–22. http://www.datenbank-spektrum.de/v2/archiv/beitrag.html?key=dbspektrum/KlettkeM03&nummer=5

[KM03] Klettke, Meike ; Meyer, Holger: Speicherung von XML-Dokumenten - eine Klassifikation. In: *Datenbank-Spektrum* 5 (2003), 40-50. http://www.datenbank-spektrum.de/v2/archiv/beitrag.html?key=dbspektrum/KlettkeM03&nummer=5

[KSS03] Klarlund, Nils ; Schwentick, Thomas ; Suciu, Dan: XML: Model, Schemas, Types, Logics, and Queries. In: Chomicki, Jan (Hrsg.) ; Meyden, Ron van d. (Hrsg.) ; Saake, Gunter (Hrsg.): *Logics for Emerging Applications of Databases*, Springer. – ISBN 3–540–00705–9, 1-41

[LSRGa03] Lósio, Bernadette F. ; Salgado, Ana C. ; Rêgo Galvão, Luciano do: Conceptual modeling of XML schemas. In: *WIDM '03: Proceedings of the 5th ACM international workshop on Web information and data management*. ACM Press. – ISBN 1–58113–725–7, 102–105

[Mal02] Maler, Eve: Representing XML Schema in UML - An UML Profile for XML Schema. (2002). http://www.idealliance.org/papers/xml02/dx_xml02/papers/05-01-02/05-01-02.pdf

[Man04] Mani, Murali: EReX: A Conceptual Model for XML. In: Bellahsene, Zohra (Hrsg.) ; Milo, Tova (Hrsg.) ; Rys, Michael (Hrsg.) ; Suciu, Dan (Hrsg.) ; Unland, Rainer (Hrsg.): *XSym* Bd. 3186, Springer. – ISBN 3–540–22969–8, 128-142

[MLM01] Mani, Murali ; Lee, Dongwon ; Muntz, Richard R.: Semantic Data Modeling Using XML Schemas. In: Kunii, Hideko S. (Hrsg.) ; Jajodia, Sushil (Hrsg.) ; Sølvberg, Arne (Hrsg.): *ER* Bd. 2224, Springer, 2001. – ISBN 3–540–42866–6, S. 149–163

[Mur99] Murata, Makoto: Hedge automata: a formal model for XML schemata. (1999). http://www.geocities.com/murata_makoto/hedge_nice.ps

[RBG02] Routledge, Nicholas ; Bird, Linda ; Goodchild, Andrew: UML and XML Schema. In: Zhou, Xiaofang (Hrsg.): *Australasian Database Conference* Bd. 5, Australian Computer Society. – ISBN 0–909–92583–6

[REL01] Clark, James (Hrsg.) ; Makoto, MURATA (Hrsg.): *RELAX NG Specification Committee Specification*. `http://www.relaxng.org/spec-20011203.html`. Version: 3 December 2001

[Sal99] Sall, Kenneth B.: Exploring the XML Infoset. (1999). `http://www.informit.com/articles/article.asp?p=31934`

[SM03] Sengupta, Arijit ; Mohan, Sriram: *Formal and conceptual models for XML structures - the past, present, and future*. `http://www.indiana.edu/~isdept/research/papers/tr137-1.pdf`. Version: 2003

[Sta73] Stachowiak, Herbert: *Allgemeine Modelltheorie*. Springer, Wien : O'Reilly & Associates, Inc., 1973. – 494 S. – ISBN 3–21181–106–0

[UML05] *Unified Modeling Language: Superstructure, Version 2.0*. `http://www.omg.org/docs/formal/05-07-04.pdf`. Version: August 2005

[Wil05] Wilde, Erik: Towards Conceptual Modeling for XML. In: Tolksdorf, Robert (Hrsg.) ; Eckstein, Rainer (Hrsg.): *Berliner XML Tage*, XML-Clearinghouse. – ISBN 3–9810105–2–3, 213-224

[XDM05] Fernández, Mary (Hrsg.) ; Malhotra, Ashok (Hrsg.) ; Marsh, Jonathan (Hrsg.) ; Nagy, Marton (Hrsg.) ; Walsh, Norman (Hrsg.): *XQuery 1.0 and XPath 2.0 Data Model (XDM) W3C Candidate Recommendation*. `http://www.w3.org/TR/2005/CR-xpath-datamodel-20051103/`. Version: 3 November 2005

[XIS04] Cowan, John (Hrsg.) ; Tobin, Richard (Hrsg.): *XML Information Set (Second Edition) W3C Recommendation*. `http://www.w3.org/TR/2004/REC-xml-infoset-20040204`. Version: 4 February 2004

[XLI01] DeRose, Steve (Hrsg.) ; Maler, Eve (Hrsg.) ; Orchard, David (Hrsg.): *XML Linking Language (XLink) Version 1.0 W3C Recommendation*. `http://www.w3.org/TR/2000/REC-xlink-20010627/`. Version: 27 June 2001

[XML00] Bray, Tim (Hrsg.) ; Paoli, Jean (Hrsg.) ; Sperberg-McQueen, C. M. (Hrsg.) ; Maler, Eve (Hrsg.): *Extensible Markup Language (XML) 1.0 (Second Edition) W3C Recommendation*. http://www.w3.org/TR/2000/REC-xml-20001006/. Version: 6 October 2000

[XS001] Fallside, David C. (Hrsg.): *XML Schema Part 0: Primer - W3C Recommendation*. http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/. Version: 2 May 2001

[XS101] Thompson, Henry S. (Hrsg.) ; Beech, David (Hrsg.) ; Maloney, Murray (Hrsg.) ; Mendelsohn, Noah (Hrsg.): *XML Schema Part 1: Structures - W3C Recommendation*. http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/. Version: 2 May 2001

[XS201] Biron, Paul V. (Hrsg.) ; Malhotra, Ashok (Hrsg.): *XML Schema Part 2: Datatypes - W3C Recommendation*. http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/. Version: 2 May 2001

Anhang A

Kurzanleitung für den EMX-Editor

A.1 Installation

Der EMX-Editor ist ein Eclipse-Plugin. Das bedeutet, er lässt sich in die Eclipse-IDE (`http://www.eclipse.org`) integrieren. Er kann aber auch als separate Anwendung (basierend auf einer minimalen Eclipse-Umgebung) gestartet werden.

A.1.1 ... als Einzelanwendung

Java 5.0 installieren
: Das Programm basiert auf Java in der Version 5.0. Deshalb ist zunächst zu prüfen, ob eine Java Runtime Environment (JRE) dieser Version auf dem Rechner zur Verfügung steht. Falls sie nicht vorhanden ist, sollte geprüft werden, ob der Download (von `http://java.sun.com`) und eine Installation lohnen. Für den Start des EMX-Editors reicht es aber auch, eine der Java-Versionen (abhängig vom Betriebssystem) von der CD auf die Festplatte zu kopieren.

EMX-Editor kopieren
: Die CD enthält das Programm in vier Versionen (für Windows, Linux (GTK + Motif), Solaris (GTK)). Die gewünschte Version ist auf die Festplatte zu übertragen.

start-Skript überprüfen

Für jedes Version wurde eine Startdatei (`run*.*`) erstellt. Die Datei muss editiert werden, um den Pfad an die lokale Java-Installation anzupassen.

A.1.2 ... als Eclipse-Plugin

Der EMX-Editor wurde für Eclipse 3.1.0 entwickelt. Er sollte zu zukünftigen Eclipse-Versionen kompatibel sein. Um das Plugin zu installieren, muss es lediglich aus dem `plugin`-Verzeichnis der CD in das `eclipse/plugins`-Verzeichnis kopiert werden. Die weiteren Plugins (im Verzeichnis auf der CD) werden zusätzlich benötigt und müssen, wenn noch nicht vorhanden, ebenfalls kopiert werden.

Bei der Verwendung einer höheren Eclipse-Version kann es passieren, dass aktuelle Versionen der externen Plugins von der Eclipse-Homepage (`www.eclipse.org`) geladen und installiert werden müssen.

!Wichtig bei Updates! *Für den Editor war es nötig, im GEF-Plugin zwei Dateien zu modifizieren. Im JAR-File des EMX-Plugins befinden sich die Dateien:* `org/eclipse/gef/ui/properties/SetValueCommand.class` und `.../SetValueCommandClassWrapper.class`.
Diese Dateien müssen an dieselbe Position in das JAR-File des GEF-Plugins hinein kopiert werden; dabei wird `SetValueCommand` *ersetzt und* `SetValueCommandClassWrapper` *neu eingefügt.*

A.2 Die Bedienung im Überblick

A.2.1 Programmstart

Aufruf von Eclipse und Initialisierung

Das Programm wird durch Aufrufen des Start-Skriptes bzw. der Eclipse-IDE gestartet. Es öffnet sich ein Dialog, in dem zuerst ein *Workspace*-Verzeichnis ausgewählt werden muss.

Mit *Workspace* wird das Arbeitsverzeichnis bezeichnet. In diesem Verzeichnis wird Eclipse alle projektspezifischen Dateien ablegen. Es empfiehlt sich, für jedes Projekt einen eigenen Workspace zu verwenden. Mit dem Kommandozeilenparameter: `eclipse.exe -data "workspaceDir"` lässt sich der Dialog umgehen und Eclipse verwendet automatisch das angegebene Verzeichnis.

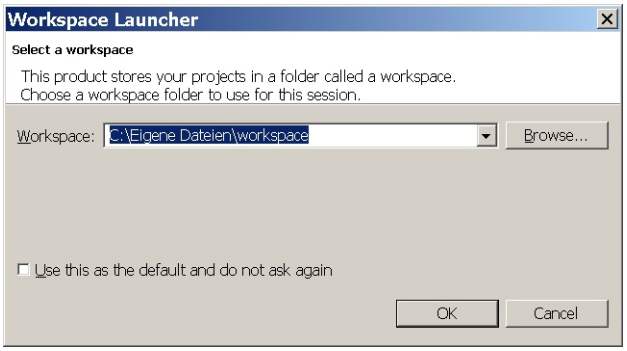


Abbildung A.1: Festlegen des Workspace-Verzeichnisses

Nach dem Start wird die leere Eclipse-Oberfläche (Abb. A.2) angezeigt:

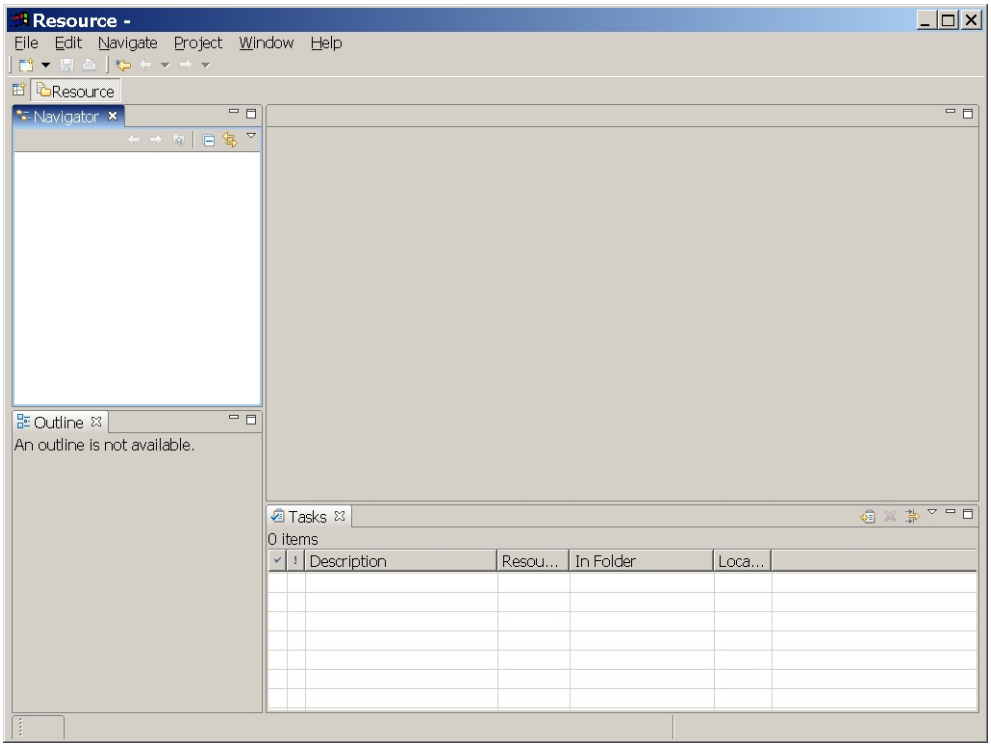


Abbildung A.2: Leere Eclipse-Oberfläche

Öffnen weiterer Views

Als nächstes sollten der *Problems View* und der *Properties View* geöffnet werden. Im *Problems View* wird der EMX-Editor Fehler am Modell anzeigen und im *Properties View* lassen sich die Eigenschaften der Modell-Entities anzeigen und editieren.

Menü: **Window** → **Show View** → **Problems** und
Menü: **Window** → **Show View** → **Properties**.

Einstellungen

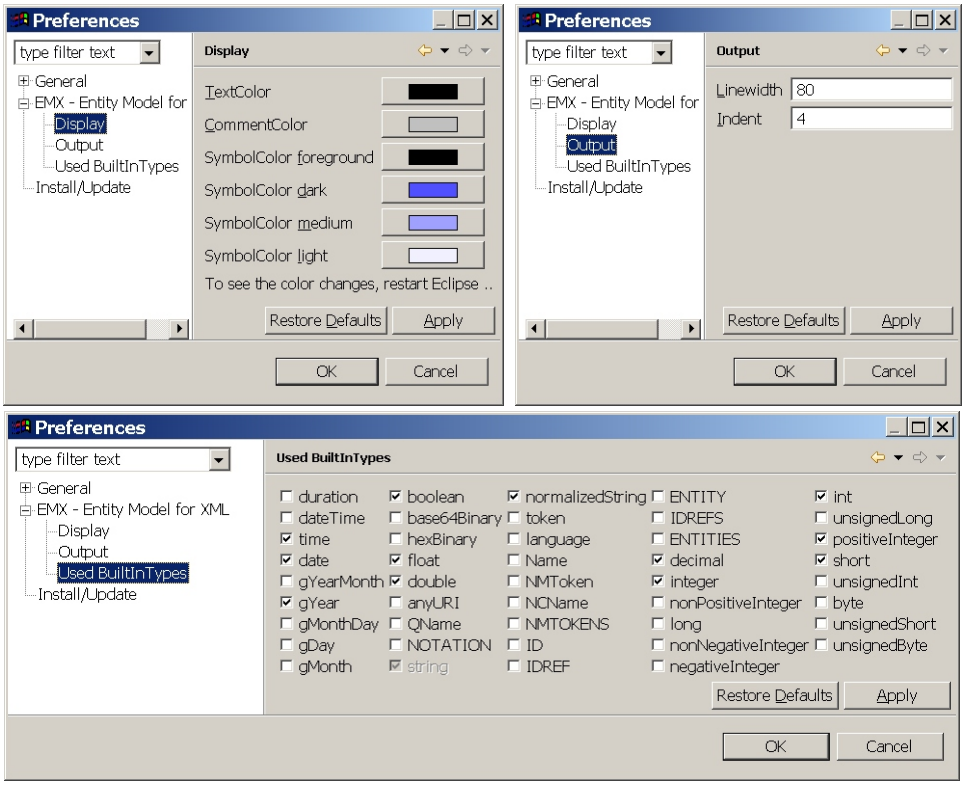


Abbildung A.3: Einstellungen für EMX-Editor

Allgemeine Einstellungen, die für alle EMX-Modelle gelten, werden im Eclipse-Einstellungsdialog vorgenommen. Dazu gehören die Farbeinstellungen für den Editor, Festlegungen für das Ausgabeformat der XSD-Dateien sowie die Auswahl der anwendbaren XML Schema-Datentypen.

Menü: **Window** → **Preferences...**

Anlegen eines neuen Projekts

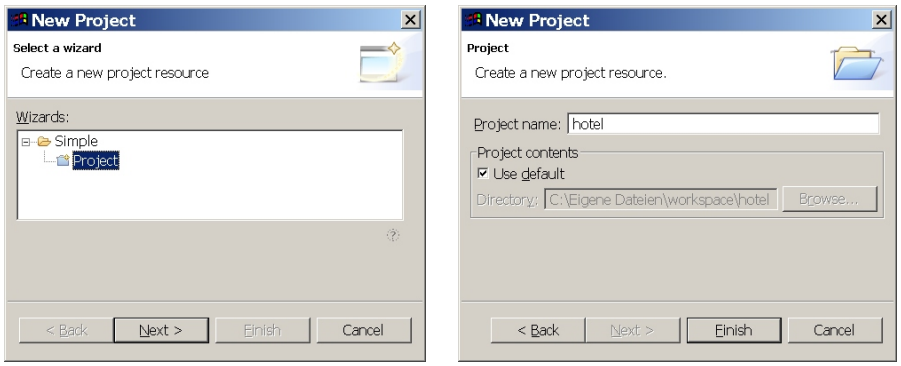


Abbildung A.4: Neues Projekt anlegen

Bevor es losgehen kann, muss noch ein neues *Projekt* erzeugt werden. Zum Anlegen des Projektes wird der *New Project Wizard* gestartet.

Menü: **File** → **New** → **Project**
Auswahl: **Simple** → **Project**
Button: **Next**
Eingabe: **Project name: ...**
Button: **Finish**

Das neue Projekt wird erstellt und im *Navigator View* angezeigt.

Neues EMX-Modell erzeugen

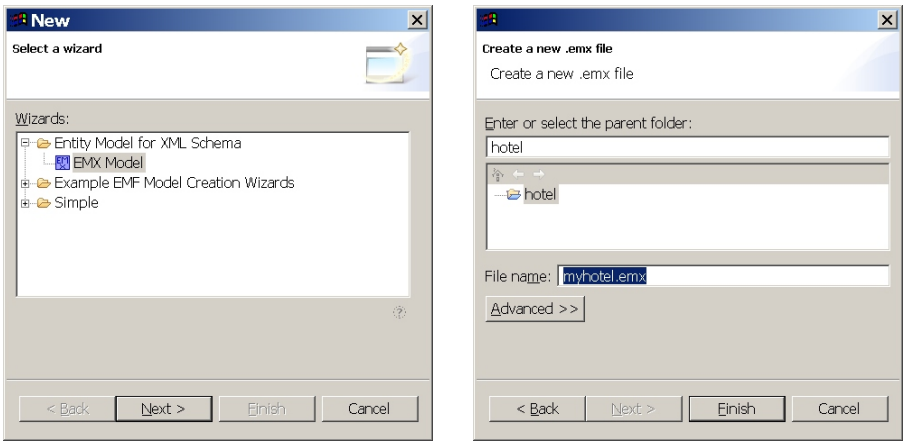


Abbildung A.5: Neues EMX-Modell anlegen

Zum Erstellen eines neuen EMX-Modells wird der *New Wizard* aufgerufen:

Menü: **File → New → Other**
Auswahl: **Entity Model for XML Schema → EMX-Model**
Button: **Next**
Eingabe: **Parent folder: ...** und **File name: ... .emx**
Button: **Finish**

Es wird ein leeres EMX-Modell erzeugt, das jetzt mit dem EMX-Editor bearbeitet werden kann.

EMX-Modell aus vorhandenem XML-Schema importieren

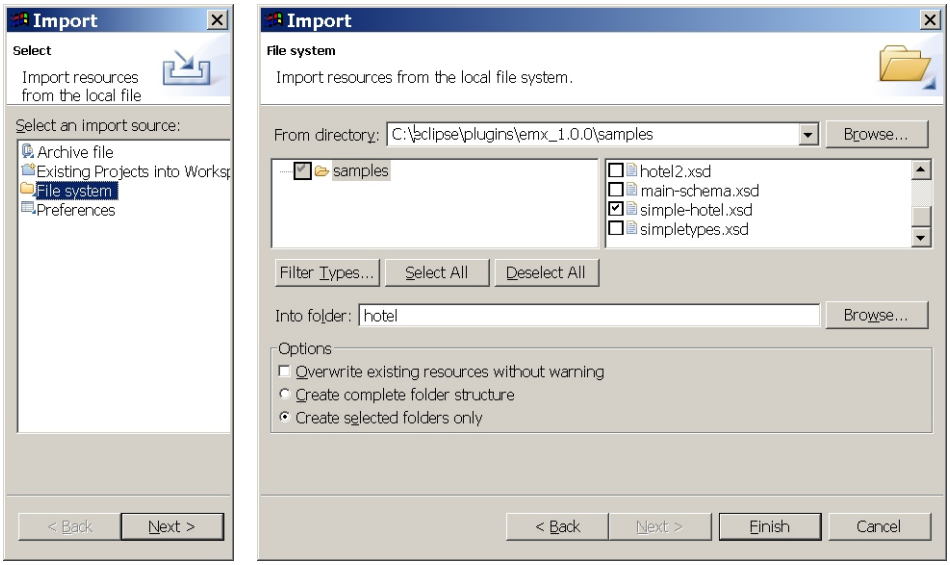


Abbildung A.6: XML-Schema importieren

Der EMX-Editor kann aus einem gegebenen XML-Schema ein EMX-Modell generieren. Dazu muss das Schema bestimmte Voraussetzungen erfüllen (z.B. „Venetian Blind"–Schema-Design, Imports nur über `include`, keine Erweiterung komplexer Typen, ...). Bevor ein EMX-Modell erzeugt werden kann, muss das XML-Schema zunächst in den *Workspace* importiert werden. Dazu verfügt Eclipse über einen *Import-Wizard*. In einem Unterverzeichnis des EMX-Plugins (`eclipse/plugins/emx_1.0.0/samples`) befinden sich einige XML-Schema-Beispieldateien zum Testen.

Menü: **File → Import...**
Auswahl: **File system**
Button: **Next**
Eingabe: **From directory**
Auswahl: **Dateien** (z.B. simple-hotel.xsd)
Eingabe: **Into folder**
Button: **Finish**

Die importierten Dateien werden im *Navigator View* angezeigt.

Der Wizard für die Übersetzung der XSD-Datei in ein EMX-Modell wird über das Kontextmenü (rechte Maustaste) gestartet.

Auswahl: **Datei** (z.B. simple-hotel.xsd) im Navigator View
Kontextmenü: **Transform into EMX Model...**
Auswahl: **Parent folder**
Eingabe: **File name**
Button: **Finish**

Die vom Wizard erstellte EMX-Datei wird im Editor geöffnet.

EMX-Modell bearbeiten

Für das Hinzufügen neuer Entities und Verbindungen kann vom rechten Rand des EMX-Editors eine Palette aufgeklappt werden. Aus der Palette kann ein Entity oder eine Verbindung ausgewählt und anschließend im Editor gezeichnet werden. Die Eigenschaften der Entities lassen sich im *Properties View* anzeigen und editieren. Für spezielle Eigenschaften (Facets und Attribute) einzelner Entities enthält der *Properties View* Buttons, über welche sich spezielle Dialoge für detailiertere Eingaben öffnen lassen.

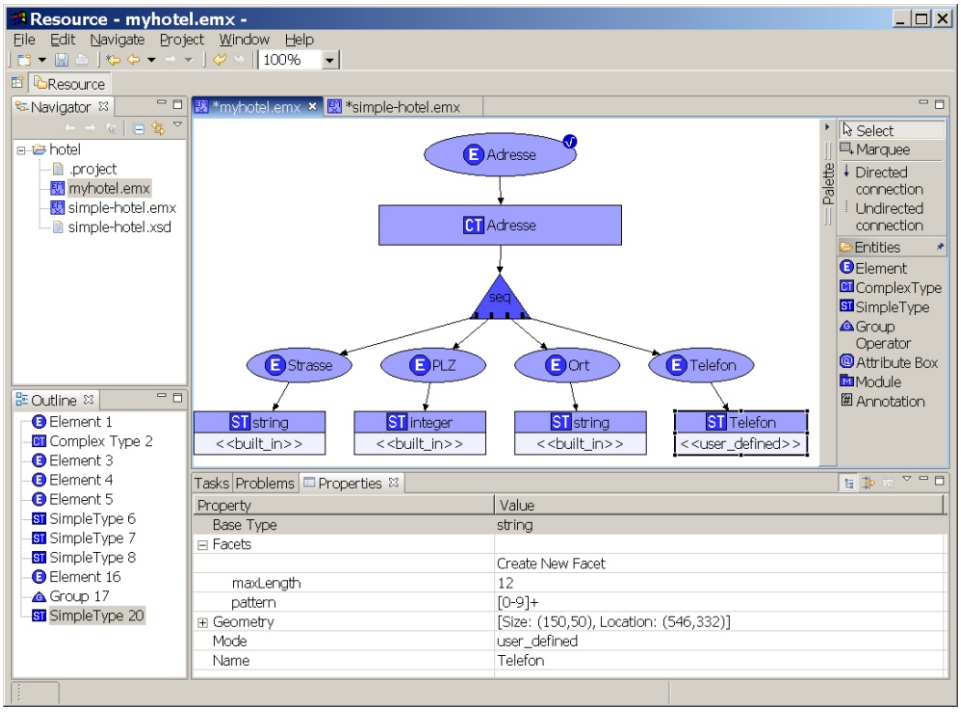


Abbildung A.7: EMX-Modell bearbeiten

Tipps und Tricks

- Die Symbolleiste (oben) enthält, eine ComboBox, über die der Zoom-Faktor für den EMX-Editor eingestellt werden kann. Außerdem enthält die Symbolleiste die Buttons für **Undo** und **Redo**
- Der *Outline View* enthält eine Liste aller Entities. Er zeigt ihre Namen so an, wie sie auch in den Fehlermeldungen verwendet werden.
- Gespeichert wird das Modell über das **File**-Menü oder **Ctrl-S**.
- Das EMX-Modell wird bei jedem Speichervorgang auf Korrektheit überprüft und Inkonsistenzen werden als Einträge im *Problems View* angezeigt. Durch Anklicken eines Eintrags wird das fehlerhafte Entity markiert.
- Das Kontextmenü für Verbindungen wurde um die Befehle **Switch Style** (gerichtet ↔ ungerichtet) und **Reverse Direction** (Umkehren der Richtung der Verbindung) erweitert. Die Einträge sind jedoch nur sichtbar, wenn die Aktion für die selektierte Verbindung zulässig ist.
- Das Kontextmenü einer EMX-Datei hat folgende Einträge:

  Make Picture Das Modell wird als Bilddatei (JPEG, PNG, BMP) exportiert.

  Layout Der Editor versucht, das Modell zu layouten. Dafür darf der Graph des EMX-Modells keine Zyklen auf der Root-Ebene aufweisen. Mit Zyklen in einem Teilgraph kann der Layout-Algorithmus umgehen.

  Fix Model Es werden Heuristiken angewandt, um Fehler im EMX-Modell zu finden und zu beheben und um ungerichteten Kanten eine Richtung zuzuweisen.

  Create XML Schema Das EMX-Modell wird in ein XML-Schema umgewandelt. Dafür werden zunächst Name und Speicherort der XSD-Datei durch einen Wizard abgefragt. Der Befehl ist nur aktiviert, wenn das Modell vorher gespeichert wurde und keine Fehler im *Problems View* mehr angezeigt werden.

- Das Kontextmenü einer XML-Schema-Datei wurde um folgende Einträge erweitert:

  Validate with SQC . Es wird der *XML Schema Quality Checker* gestartet, den IBM auf seiner *alphaworks-Website* `http://www.alphaworks.ibm.com/tech/xmlsqc` als Testversion zum Download bereitstellt. Der SQC erzeugt ein Unterverzeichnis **SchemaAnalysis** im Projektverzeichnis, in welchem er XML-Dateien speichert, die die Ergebnisse des Tests enthalten.

Open With → Sample XML Schema Editor Zu Testzwecken wurde ein einfacher graphischer Editor für XML-Schemas integriert. Er stammt aus dem XSD-Projekt von Eclipse (`http://www.eclipse.org/xsd/`).

- Für das Erstellen korrekter EMX-Modelle ist ein Blick in die Diplomarbeit unerlässlich. Sie enthält sowohl Beispiele als auch eine Verbindungsmatrix, die anzeigt, wie Entities miteinander in Beziehung stehen dürfen.

- Eclipse-spezifische Hinweise

  - Durch einen **Doppelklick** auf die Überschrift des EMX-Editors wird die Ansicht maximiert. Ein weiterer Doppelklick verkleinert sie wieder.
  - Der Menüeintrag **Window → Reset Perspective** stellt die Ausgangsansicht wieder her. Dies kann hilfreich sein, wenn die Anordnung der verschiedenen Views durcheinander geraten ist. Im Anschluss daran müssen *Problems View* und *Properties View* wieder manuell geöffnet werden.
  - Das Editieren der Einträge im *Properties View* sollte immer mit ENTER abgeschlossen werden.
  - Vor dem Aufruf des Kontextmenüs (rechte Maustaste) muss die entsprechende Datei explizit markiert werden (linke Maustaste).
  - Laufzeitfehler-Meldungen (Exceptions) speichert Eclipse in einer Log-Datei. Sie heißt `.log` und befindet sich im Unterverzeichnis `.metadata` des Workspaces.

Anhang B

Hinweise für den zukünftigen Programmierer

B.1 Java 5.0

Beim Programmieren des EMX-Editors sollten auch die neuen Features von Java 5.0 ausprobiert werden. Einen guten Überblick über die Neuerungen bietet der Artikel „J2SE 5.0 in a Nutshell“ (`http://java.sun.com/developer/technicalArticles/releases/j2se15/`) und die Bakkalaureatsarbeit „Neuerungen in Java 5.0“ von Petra Brosch (`http://www.big.tuwien.ac.at/teaching/practicals-diplomatheses/bachelor/doc/neuerungen_in_java5.pdf`).

Vor allem von den *Typesafe Collections* wurde intensiver Gebrauch gemacht. Dadurch ist es möglich, den Datentyp der Inhaltselemente von Collections (Vektoren, Hashtables, Arraylists, ...) explizit festzulegen. Die Korrektheit der Typen kann so bereits zur Compilezeit überprüft werden. `get`-Methoden liefern nun gleich den konkreten Typ für die Elemente anstelle des allgemeinen Typs `Object`, welcher zur Laufzeit (ohne Prüfung des Compilers) gecastet werden musste.

Vorteilhaft erwies sich auch die erweitere Syntax für `for`-Schleifen. Durch sie kann mit sehr geringem Kodieraufwand über den Inhalt von Collection-Objekten iteriert werden.

B.2 Kompilieren der Anwendung

Für den Fall, dass das Projekt: *„EMX-Editor“* fortgeführt wird, wurde der Inhalt des Eclipse-Workspaces auch auf die CD gebrannt. Er enthält sämtliche Source-Dateien und externe Plugins.

Für das Kompilieren und Zusammenstellen der Standalone-Versionen für die verschiedenen Betriebssysteme wurde ein Buildskript `build/build.xml` für die Build-Umgebung *ANT* (`http://ant.apache.org`) geschrieben.

ANT ist auch Bestandteil der Eclipse-IDE. Für den Start des Build-Prozesses existiert eine Batchdatei. In dieser müssen die Pfade an das aktuelle System angepasst werden.

B.3 Minimal benötigte Eclipse-Plugins

Die Minimalversion von Eclipse, die benötigt wird, um den EMX-Editor zu starten, wird aus folgenden Dateien und Verzeichnissen (Quellen für Plugins) erstellt:

eclipse-RCP-3.1-*OS*.zip
: Die Datei enthält die Platform-abhängigen Bestandteile von Eclipse. Je nach Platform wird eine der folgenden Dateien benötigt:
`eclipse-RCP-3.1-linux-gtk.tar.gz`,
`eclipse-RCP-3.1-linux-motif.tar.gz`,
`eclipse-RCP-3.1-solaris-gtk.zip` oder
`eclipse-RCP-3.1-win32.zip`.
Die Datei muss zuerst vollständig extrahiert werden. Dabei wird ein `eclipse`-Verzeichnis erstellt.

emx-Verzeichnis
: Das Verzeichnis enthält den im Rahmen der Diplomarbeit erstellten EMX-Editor als Eclipse-Plugin. Er wird in das `eclipse/plugins`-Verzeichnis kopiert.

eclipse-platform-3.1-win32.zip
: Aus dieser Datei werden einige Eclipse-Plugins benötigt. Diese Komponenten sind für alle Betriebssysteme gleich; deshalb kann diese Datei auch für Linux- und Solaris-Version verwendet werden. Es sollten nur die benötigten Plugins in das `eclipse/plugins`-Verzeichnis extrahiert werden.

emf-sdo-runtime-2.1.0.zip
: Diese Datei enthält Plugins, die gemeinsam von der XSD-API und der GEF-API verwendet werden. Es sollten nur die benötigten Plugins in das `eclipse/plugins`-Verzeichnis extrahiert werden.

GEF-runtime-3.1.zip
: Diese Datei enthält die Plugins des Graphical Editing Frameworks (GEF). Es werden nicht alle Plugins benötigt.

xsd-runtime-2.1.0.zip
: Diese Datei enthält die Plugins der XSD-API zum Generieren und Modifizieren von XML-Schemas. Es werden nicht alle Plugins benötigt.

SQC2.2.1.jar

Diese Datei ist eine Testversion des *XML Schema Quality-Checkers* (SQC) von der IBM Alphaworks Website (`http://www.alphaworks.ibm.com/tech/xmlsqc`). Sie wird einfach in das `eclipse/plugins`-Verzeichnis kopiert. Mit dem SQC lassen sich die erstellten XML-Schema-Dateien auf Korrektheit überprüfen.

Die benötigten Plugins und ihre Herkunft werden in Tabelle B.1 dargestellt. Zur Installation der Plugins reicht es, sie in das `eclipse/plugins`-Verzeichnis zu extrahieren.

Provider	**Plugin Name + ID**	**Ver.**	**Quelle**
Eclipse.org	Ant Build Tool Core `org.eclipse.ant.core`	3.1.0	`eclipse-platform` `-3.1-win32.zip`
Eclipse.org	Cheat Sheets `org.eclipse.ui.cheatsheets`	3.1.0	`eclipse-platform` `-3.1-win32.zip`
Eclipse.org	Command `org.eclipse.core.commands`	3.1.0	`eclipse-RCP` `-3.1-OS.zip`
Eclipse.org	Core Resource Management `org.eclipse.core.resources`	3.1.0	`eclipse-platform` `-3.1-win32.zip`
Eclipse.org	Core Runtime `org.eclipse.core.runtime`	3.1.0	`eclipse-RCP` `-3.1-OS.zip`
Eclipse.org	Core.R. Plug-in Compatibility `... .core.runtime.compatibility`	3.1.0	`eclipse-platform` `-3.1-win32.zip`
Eclipse.org	Core Variables `org.eclipse.core.variables`	3.1.0	`eclipse-platform` `-3.1-win32.zip`
Eclipse.org	Default Text Editor `org.eclipse.ui.editors`	3.1.0	`eclipse-platform` `-3.1-win32.zip`
Eclipse.org	Draw2d `org.eclipse.draw2d`	3.1.0	`GEF-runtime` `-3.1.zip`
Eclipse.org	Eclipse Forms `org.eclipse.ui.forms`	3.1.0	`eclipse-platform` `-3.1-win32.zip`
Eclipse.org	Eclipse IDE UI `org.eclipse.ui.ide`	3.1.0	`eclipse-platform` `-3.1-win32.zip`
Eclipse.org	Eclipse UI `org.eclipse.ui`	3.1.0	`eclipse-platform` `-3.1-win32.zip`
Eclipse.org	EMF Common `org.eclipse.emf.common`	2.1.0	`emf-sdo-runtime` `-2.1.0.zip`
Eclipse.org	EMF Common UI `or.eclipse.emf.common.ui`	2.1.0	`emf-sdo-runtime` `-2.1.0.zip`
Eclipse.org	EMF Ecore `org.eclipse.emf.ecore`	2.1.0	`emf-sdo-runtime` `-2.1.0.zip`
... wird fortgesetzt ...			

Provider	Plugin Name + ID	Ver.	Quelle
Eclipse.org	EMF Edit `org.eclipse.emf.edit`	2.1.0	`emf-sdo-runtime -2.1.0.zip`
Eclipse.org	EMF Edit UI `org.eclipse.edit.ui`	2.1.0	`emf-sdo-runtime -2.1.0.zip`
Eclipse.org	EMF XMI `org.eclipse.ecore.xmi`	2.1.0	`emf-sdo-runtime -2.1.0.zip`
Eclipse.org	Expression Language `org.eclipse.core.expressions`	3.1.0	`eclipse-RCP -3.1-OS.zip`
Eclipse.org	File Buffers `org.eclipse.core.filebuffers`	3.1.0	`eclipse-platform -3.1-win32.zip`
Eclipse.org	Graphical Editing Framework `org.eclipse.gef`	3.1.0	`GEF-runtime -3.1.zip`
Eclipse.org	Help System Core `org.eclipse.help`	3.1.0	`eclipse-RCP -3.1-OS.zip`
Eclipse.org	Install/Update Configurator `org.eclipse.update.configurator`	3.1.0	`eclipse-RCP -3.1-OS.zip`
Eclipse.org	Install/Update Core `org.eclipse.update.core`	3.1.0	`eclipse-platform -3.1-win32.zip`
Eclipse.org	Install/Update UI `org.eclipse.update.ui`	3.1.0	`eclipse-platform -3.1-win32.zip`
Eclipse.org	JFace `org.eclipse.jface`	3.1.0	`eclipse-RCP -3.1-OS.zip`
Eclipse.org	JFace Text `org.eclipse.jface.text`	3.1.0	`eclipse-platform -3.1-win32.zip`
Eclipse.org	OSGi System Bundle `system.bundle`	3.1.0	`eclipse-RCP -3.1-OS.zip`
Eclipse.org	Standard Widget Toolkit `org.eclipse.swt`	3.1.0	`eclipse-RCP -3.1-OS.zip`
Eclipse.org	Standard Widget Toolkit for Windows *(Linux, ...)* `... .swt.win32.win32.x86`	3.1.0	`eclipse-RCP -3.1-OS.zip`
Eclipse.org	Text `org.eclipse.text`	3.1.0	`eclipse-platform -3.1-win32.zip`
Eclipse.org	Text Editor Framework `... .ui.workbench.texteditor`	3.1.0	`eclipse-platform -3.1-win32.zip`
Eclipse.org	Views `org.eclipse.ui.views`	3.1.0	`eclipse-platform -3.1-win32.zip`
Eclipse.org	Workbench `org.eclipse.ui.workbench`	3.1.0	`eclipse-RCP -3.1-OS.zip`
... wird fortgesetzt ...			

Provider	Plugin Name + ID	Ver.	Quelle
Eclipse.org	XML Schema Edit Framework `org.eclipse.xsd.edit`	2.1.0	`xsd-runtime-2.1.0.zip`
Eclipse.org	XML Schema Editor `org.eclipse.xsd.editor`	2.1.0	`xsd-runtime-2.1.0.zip`
Eclipse.org	XML Schema Infoset Model (XSD) `org.eclipse.xsd`	2.1.0	`xsd-runtime-2.1.0.zip`
IBM Reseach via alphaworks	SQC Plugin `com.ibm.etools.sketch.sqc`	2.2	`SQC2.2.1.jar`
Robert Stephan	Emx Plug-in `emx`	1.0.0	`emx_1.0.0-Verzeichnis`

Tabelle B.1: Minimal benötigte Plugins

B.4 Artikel und Tutorials

An dieser Stelle sind Tutorials und Artikel aufgelistet, die für das Verständis einzelner Konzepte hilfreich seien können.

Java 5.0

- „J2SE 5.0 in a Nutshell“
  `http://java.sun.com/developer/technicalArticles/releases/j2se15/`
- Bakkalaureatsarbeit: „Neuerungen in Java 5.0“ von Petra Brosch
  `http://www.big.tuwien.ac.at/teaching/practicals-diplomatheses/bachelor/doc/neuerungen_in_java5.pdf`

Eclipse Basics

- „Your First Plug-in“
  `http://www.eclipse.org/articles/Article-Your%20First%20Plug-in/YourFirstPlugin.html`
- „PDE Does Plug-ins“
  `http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html`
- „Creating Eclipse plug-ins“
  `http://devresource.hp.com/drc/technical_articles/ePlugIn/index.jsp`
- Eclipse-Help (offline in Eclipse IDE oder online)
  `http://help.eclipse.org/help31/index.jsp`

SWT und JFace

(GUI-Programmierung in Eclipse)

- Eclipse SWT Project Homepage
  `http://www.eclipse.org/swt/`
  (besonders die Unterseiten: „Widgets“ und „Snippets“)
- „Understanding Layouts in SWT“
  `http://www.eclipse.org/articles/Understanding%20Layouts/Understanding%20Layouts.htm`
- „Creating JFace Wizards“
  `http://www.eclipse.org/articles/Article-JFace%20Wizards/wizardArticle.html`
- „Wizards in Eclipse“
  `http://devresource.hp.com/drc/technical_articles/wizards/index.jsp`

Eclipse - Spezialitäten

- ”Contributing Actions to the Eclipse Workbench“
  `http://www.eclipse.org/articles/Article-action-contribution/Contributing%20Actions%20to%20the%20Eclipse%20Workbench.html`
- „Preferences in the Eclipse Workbench UI“
  `http://www.eclipse.org/articles/Article-Preferences/preferences.htm`
- „Simplifying Preference Pages with Field Editors“
  `http://www.eclipse.org/articles/Article-Field-Editors/field_editors.html`
- „Mark My Words - Using markers to tell users about problems and tasks“
  `http://www.eclipse.org/articles/Article-Mark%20My%20Words/mark-my-words.html`

Draw2D und GEF (Graphical Editing Framework)

- „Display a UML Diagram using Draw2D“
  `http://www.eclipse.org/articles/Article-GEF-Draw2d/GEF-Draw2d.html`
- IBM Redbook:
  „Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework“
  `http://www.redbooks.ibm.com/redbooks/pdfs/sg246302.pdf`
- „A Shape Diagram Editor“
  `http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html`
- „Create an Eclipse-based application using the Graphical Editing Framework“
  `http://www-128.ibm.com/developerworks/opensource/library/os-gef/`

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit mit dem Titel

„Entwicklung und Implementierung einer Methode zum konzeptuellen Entwurf von XML-Schemas“

selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Dargun, den _____ _____

Unterschrift

Thesen

- Konzeptuelle Modelle unterstützen den Entwurf komplexer XML-Anwendungen.
- Das EMX-Modell ist ein konzeptuelles Modell für XML-Schema.
- Das EMX-Modell besteht aus wenigen Basiskonstrukten (Entities, Verbindungen, Eigenschaften).
- Die graphische Darstellung des EMX-Modells ist leicht zu lesen.
- Mit dem EMX-Modell lassen sich die wesentlichen Aspekte des XML-Schema-Standards darstellen. - Es ist einfach zu erweitern.
- Eine formale Grundlage ist wichtig für ein konzeptuelles Modell. - Das EMX-Modell basiert auf gemischten (*engl.* mixed) Graphen.
- Das EMX-Modell kann automatisch aus einem XML-Schema erzeugt werden. Aus einem EMX-Modell kann automatisch das XML-Schema generiert werden.
- Der Import-Prozess wird durch die Einführung eines Vorübersetzungsprozesses vereinfacht. In diesem Schritt werden vom Modell nicht darstellbare XML-Schema-Konstrukte in semantisch äquivalente Konstrukte, die durch das Modell unterstützt werden, umgewandelt.
- Mit dem EMX-Editor steht ein Tool zur Verfügung, dass den Benutzer beim Erstellen und Bearbeiten von EMX-Modellen gut unterstützt und anleitet.