Untersuchungen zur automatischen XML-Schema-Ableitung

Diplomarbeit

Christian Romberg
Universität Rostock
Fachbereich Informatik, LS DBIS
Albert-Einstein-Straße 21
D–18051 Rostock

Organisatorisches

Autor:	Christian Romberg
Geburtstag:	10.09.1976
Geburtsort:	Rostock
Matrikelnummer:	096202039
Betreuer:	Dr.-Ing. Holger Meyer
Erstgutachter:	Prof. Dr. Andreas Heuer
Zweitgutachter:	Prof. Dr. Clemens Cap
Bearbeitungszeitraum:	15. Mai – 30. September 2001
Abgabe:	01. Oktober 2001

Zusammenfassung

Diese Diplomarbeit untersucht die vielfältigen Probleme die auftreten, wenn wir aus einer gegebenen Menge an XML-Dokumenten ein XML-Schema (bzw. eine DTD) ableiten wollen. Das umfaßt eine Einführung in die theoretischen Grundlagen und eine intensive Analyse existierender Ansätze eine DTD zu generieren und eine Untersuchung, ob und wie die Ergebnisse genutzt werden können, um ein XML-Schema zu erzeugen. Weiterhin werden zusätzliche Komponenten von XML-Schema diskutiert, z.B. Integritätsbedingungen und Datentypen.

Abstract

This master thesis discusses the various problems that arise if we try to infer an XML schema (or a DTD respectively) from a given collection of XML documents. This includes an introduction to the theoretical background and an intensive analysis of existing approaches for generating a DTD, an investigation if and how the results can be transferred to produce an XML schema. Furthermore additional components of an XML schema are discussed, e.g. integrity constraints and data types.

CR-Klassifikation:

F.4.3 Formal Languages (D.3.1)
- Algebraic language theory
- Classes defined by grammars or automata
- Operations on languages

H.2.1 Logical Design
- Data models

H.2.3 Languages (D.3.2)
- Date description languages (DDL)

H.3.1 Content Analysis and Indexing
- Abstracting methods

I.7.2 Document Preparation
- Hypertext/hypermedia

Key Words:

XML, DTD, XML schema, grammar inference, finite state automaton, ambiguity, determinism, minimum descriptor length principle, language identification in the limits, 1-unambiguity, data types, key constraints

Inhalt

Kapitel 1

Einleitung

1.1 Motivation und Hintergrund

XML ist das zukünftige universelle Datenaustauschformat für das WWW und andere Einsatzgebiete. Ein konkretes XML-Dokument kann speziellen Restriktionen an Struktur und Inhalt unterliegen, wenn ihm ein Document Type Descriptor (DTD) oder ein XML-Schema zugeordnet ist.

DTDs oder XML-Schemata werden benötigt, um die Dokumente zu validieren (d.h. ihre Korrektheit sicherzustellen). Für die effiziente Speicherung, Anfragesprachen oder auch die Abbildung von XML-Daten in eine Datenbank benötigen wir ebenso konkrete Strukturinformationen (vgl. [22]).

Zu einer Menge von gegebenen XML-Dokumenten hat man aber nicht zwingend eines der beiden gegeben, sei es, daß die XML-Dokumente aus einer Quelle stammen, die nicht gleichzeitig eine DTD (o.ä.) zur Verfügung stellte und/oder es wurde vielleicht noch überhaupt keine DTD oder kein XML-Schema definiert. Ebenso können die XML-Dateien das Ergebnis einer Migration einer anderen Darstellung (aus einer Datenbank oder – wie bei Ahonen und Manila – aus einem konkreten Wörterbuch) sein, so daß hier evt. ebenfalls keine Metainformationen vorliegen.

Denkbar sind auch Migrationen aus der HTML-Darstellung von Daten, wobei dann zusätzliche Tags eingeführt werden oder die Migration von XML-Dateien, die aus verschiedenen Quellen (mit ähnlicher Struktur) stammen, auch hier kann der Fall eintreten, daß eine DTD (bzw. XML-Schema) fehlt.

Das Erzeugen einer DTD oder eines XML-Schemas ist eine Semantikerschließung aus der Dokumentmenge, daher kann es sogar dann eingesetzt werden, wenn wir schon eine DTD bzw. ein XML-Schema haben, um festzustellen, ob die vorhandene Strukturbeschreibung auch wirklich adäquat ist, also einerseits keine Aspekte festlegt, die konkret nie benutzt werden und andererseits auch nicht zu allgemein ist, so daß die konkreten XML-Daten mehr Informationen implizieren, als in dieser Strukturbeschreibung festgelegt sind.

1.2 Konkrete Problemstellung

Diese Diplomarbeit untersucht die Möglichkeiten, zu einer gegebenen Menge an *wohlgeformten* XML-Dokumenten Strukturinformationen zu erschließen, konkretes Ziel ist dabei die Erstellung einer DTD oder eines XML-Schemas.
Die gegebene Menge an XML-Dokumenten sei mit I bezeichnet, es wird dabei nicht vorausgesetzt, daß I vollständig ist, also weitere Dokumente sollen prinzipiell zu I hinzufügbar sein und die aus I abgeleiteten Strukturinformationen sollen auch für diese gelten. I wird deshalb auch als *Beispiel(dokument)menge* bezeichnet.

Ein Szenario wäre z.B. die Gewinnung einer DTD zu 100 XML-Dokumenten, welche die bibliographischen Daten von Büchern enthalten, generell soll das erzeugte Schema die Eigenschaft haben, weder zu konkret, als noch zu unkonkret zu sein, d.h. einserseits auch Dokumente zu validieren, die ähnlich zu den in I enthaltenen sind, andererseits aber nicht jedes wohl-geformte XML-Dokument zu validieren.

Weiterhin wird vorausgesetzt, daß eine DTD bzw. ein XML-Schema existiert, so daß I eine Menge von Beispielen der durch diese DTD bzw. dieses XML-Schema definierten Klassen von XML-Dokumente ist. Dieses ursprüngliche Schema ist natürlich unbekannt[1], Ziel ist es, diesem ursprünglichen Schema so nahe wie möglich zu kommen, wofür wir natürlich voraussetzen müssen, daß die Menge I auf gewisse Art und Weise *repräsentativ*, d.h. *charakteristisch* für das Schema ist. Die Annahme dient dazu, eine (unbekannte) Semantik der Dokumente vorauszusetzen, welche wir erschließen wollen. In der Anwendung können und wollen wir die Verfahren natürlich auch ohne Prüfung dieser Annahme benutzen.

Diese Annahme dient deshalb dazu sicherzustellen, daß die Verfahren ein *sinnvolles* Ergebnis liefern. Wenden wir das Verfahren ohne Prüfung an und ist das Ergebnis sinnvoll, so muß die Annahme gelten, denn das Ergebnis wäre ja genau das geforderte Schema. Die Prüfung der Güte eines Schemas wird für den Fall der Anwendung ohne Voraussetzung nicht weiter diskutiert, wir wollen annehmen, daß ein Mensch, der sich Ergebnis und Dokumente ansieht, die Güte bzgl. dieses Punktes unmittelbar erkennt.

Gilt die Annahme nämlich nicht – das wäre z.B. der Fall, wenn jedes Dokument aus der Menge eine völlig andere Struktur und Bedeutung hat – so können wir natürlich kein sinnvolles Ziel der Strukturerschließung festlegen, das Verfahren wird hier mit großer Wahrscheinlichkeit eine Übergeneralisierung durchführen, die wir als Mensch (oder mit Heuristiken) rasch erkennen können, da die Schachtelungstiefe dann erwartungsgemäß nicht tief sein wird, ebenso wird der *-Operator häufige Anwendung finden.

Womit sich diese Arbeit nicht beschäftigt, ist die vollständige Diskussion der Migration von ähnlichen Daten aus verschiedenen Datenquellen. Bei der Migration müssen auch Aspekte diskutiert werden wie die Ableitung eine Schemainformation aus den evt. vorhandenen Schemainformationen der zu migrierenden Datenbestände, weiterhin ist oftmals eine Bearbeitung der Daten (z.B. Elementumbenennung) sinnvoll und notwendig.

Folgendes soll erreicht werden:

- Jedes XML-Dokument aus I ist gültig gemäß XML- bzw. XML-Schema-Spezifikation in Bezug auf die erstellte DTD bzw. das erstellte XML-Schema.

- Die erfolgte Generalisierung ist sinnvoll, d.h. die Menge der gültigen Dokumente gemäß des erzeugten Schemas umfaßt einerseits nicht alle wohl-geformten XML-Dokumente – da wir dann ja praktisch keine nähere Semantik abgeleitet haben – aber andererseits beschränkt sich die Menge nicht nur auf eine minimale Obermenge von I, so daß wir praktisch kaum ein Dokument haben, was einerseits den Dokumenten aus I strukturell und

[1] es ist wohlgemerkt kein Widerspruch, einem *unbekannten* Schema so nahe wie möglich zu kommen

semantisch entspricht, aber nicht von dem erzeugten Schema validiert wird.
Die konkreten Verfahren und Vorgehensweisen besitzen implizite oder explizite Metriken bzw. Kriterien für die Gewährleistung dieses Punktes, welche dann an der jeweiligen Stelle vorgestellt und motiviert werden.

- Das erzeugte Schema soll nach Möglichkeit für einen Menschen gut lesbar sein, einige Verfahren behandeln diesen Aspekt schon unter obigem Punkt.
- Das Verfahren soll ein Schema zu einer *Menge* von XML-Dokumenten finden, wobei die Dokumente physisch natürlich eine gewisse Reihenfolge besitzen. Offenbar soll die Semantik nichts mit dieser konkreten Reihenfolge zu tun haben, die Verfahren sollen also reihenfolgeunabhängig arbeiten und weiterhin auch schon bei einer geringen Größe von I sinnvolle Ergebnisse liefern.

Unter Umständen können noch speziellere Forderungen gestellt werden, welche z.B. die Abbildung in ein Datenbanksystem erleichtern. Solche spezielleren Forderungen werden aber in dieser Arbeit nicht diskutiert.

1.3 Gliederung

Diese Arbeit teilt sich auf wie folgt: Kapitel 2.1 führt die formalen Grundlagen zu XML ein und behandelt auch andere wichtige benötigte Aspekte wie z.B. endliche Automaten.

In Kapitel 3 werden verschiedene vorhandene Verfahren vorgestellt – und vor allem diskutiert – die zur Generierung einer DTD dienen, am Ende dieses Kapitel wird dann untersucht, wie und ob sich diese Herangehensweisen auch auf XML-Schema übertragen lassen.
Die abzuleitenden Schemainformationen können wir auftrennen in (reine) Strukturinformationen, Datentypen für Attribute bzw. Elemente die nur Stringinhalt haben, sowie Integritätsbedingungen. In Kapitel 3 wurde nur der erste Aspekt behandelt, weshalb sich in den Kapiteln 4 und 5 eine Betrachtung der anderen beiden Aspekte anschließt.
In Kapitel 6 wird beispielhaft eine Vorgehensweise implementiert.
Das Kapitel 7 bringt dann eine Zusammenfassung und abschließende Diskussion.

Kapitel 2

Theoretische Grundlagen

2.1 Einführung

XML ist das vorgesehene universelle Datenaustauschformat für das WWW und soll potentiell HTML ablösen können.

An dieser Stelle wird eine Vertrautheit des Lesers mit dem Aufbau von HTML-Dateien vorausgesetzt, also daß HTML-Dokumente aus Elementen bestehen, die in der Regel ein Start-Tag (z.B. `<html>`) und ein End-Tag besitzen (z.B. `</html>`), daß diese Elemente zwischen ihren Start- und End-Tags Inhalt besitzen, der wiederum aus Elementen oder Text besteht, sowie daß Elemente Attribute besitzen können, welche Werte haben. Wie ein korrektes HTML-Dokument exakt auszusehen hat, entnehme man der HTML-Spezifikation [28].

HTML-Dateien werden auf Nutzerseite hauptsächlich mit Browsern bearbeitet, wobei bearbeitet hier etwas hochgegriffen klingt. Vielmehr stellen die Browser nur HTML-Dateien dar, es ist also so, daß die Semantik der in den HTML-Dokumenten enthaltenen Elemente konkret festliegt, vgl. HTML-Spezifikation [28].

XML kann als Verallgemeinerung von HTML in dem Sinne verstanden werden, daß es möglich ist, eigene Elemente und deren Attribute zu definieren.

Im Gegensatz zu HTML, welches eine SGML-Anwendung ist, soll XML als Teilmenge von SGML verstanden werden. Das hier verbal genutzte Teilmengenkonzept wurde aber nach meinem Kenntnisstand nirgendwo formal definiert, die Aussage bezieht sich wahrscheinlich darauf, daß jede XML-Anwendung auch eine SGML-Anwendung ist. Sowohl SGML als auch XML sind aus sprachtheoretischer Sicht keine formalen Sprachen, vielmehr geben beide die Möglichkeiten vor, gewisse eigene formale Sprachen zu definieren, deshalb werden SGML und XML i.allg. und auch im folgenden Metasprachen genannt.

Definition:
Eine konkrete SGML- bzw. XML-Sprache wird auch als **SGML-** bzw. **XML-Anwendung** bezeichnet.

Beispiel:
HTML ist eine SGML-Anwendung[1]

Definition:
Ein Wort einer XML-Anwendung (bzw. SGML-Anwendung) (ob nun in Form einer Datei, als

[1] ein XML-konformes HTML wurde mit XHTML definiert; XHTML ist eine XML-Anwendung

Datenbankinhalt, als Stream, ...) wird als **XML-Dokument** (bzw. SGML-Dokument) bezeichnet.

Beispiel:

```
<html>
  <head>
    <title>a pretty useless html-file</title>
  </head>
  <body>
    Hello World!
  </body>
</html>
```

ist ein HTML (und sogar XHTML-)Dokument

Bemerkung:
„Wort“ ist in diesem Kontext eine Zeichensequenz über einem gewissen Alphabet, später wird eine isomorphe Graphendarstellung (XML-Dokument als Baum) eingeführt.

XML enthält zwei wesentliche Konzepte, nämlich das Konzept der **Wohlgeformtheit** und das Konzept der **Gültigkeit**.

Definition:
Ein XML-Dokument ist **wohlgeformt** (engl. well-formed) wenn es alle Wohlgeformtheitskriterien der XML-Spezifikation [11] erfüllt.

Diese Kriterien sollen an dieser Stelle nicht wiederholt werden, u.a. enthalten sie Bedingungen wie, daß zu jedem Start-Tag auch ein End-Tag gehört und daß die Schachtelung der Elemente korrekt ist.

Definition:
Jede Teilmenge aller wohlgeformten XML-Dokumente ist eine XML-Sprache.

Bemerkung:
Wie schon bei einer formalen Sprache (als beliebige Teilmenge über Σ^*) existiert nicht zwingend eine Beschreibungsmöglichkeit für eine konkrete XML-Sprache.

Eine **Document Type Definition** (DTD) ist das durch die XML-Spezifikation gegebene Konzept, eine XML-Anwendung zu definieren. Im wesentlichen umfaßt eine konkrete DTD Restriktionen an die Schachtelung der Elemente, also welche Elemente können in welcher Reihenfolge innerhalb (bzw. unterhalb in der Baumsichtweise) eines gegebenen Elementes auftreten, Restriktionen an die Attribute von Elementen, also welches Element kann bzw. muß welche Attribute haben, welche Typen haben die Attribute usw.

Das `<!ELEMENT` a X`>`-Konstrukt dient zur Definition eines Elementes vom Typ und Namen a. In einem konkreten XML-Dokument welches ein a-Element enthält, tritt dieses in der Form `<a ...> ... </a>` oder in der Form `<a ... />` auf. X ist eine Modellgruppe (oder auch Inhaltsmodell; engl. model group bzw. content model) über anderen in dieser DTD definierten Elementen und der Zeichenkette „#PCDATA“.

Eine Modellgruppe ist im wesentlichen isomorph zu einem regulären Ausdruck (vgl. [20]) über dem Alphabet der Elementnamen. Die XML-Spezifikation fordert wörtlich, daß diese Modellgruppe deterministisch ist. Die konkrete Bedeutung dieser Forderung wird weiter unten noch genauer betrachtet. Nachfolgend einige einfache Beispiele für Modellgruppen (`a,b,c,d` seien Elementnamen) die anstelle von X stehen könnten:

`(b,c,d)`	Sequenz, d.h. die Elemente müssen in dieser Reihenfolge genau einmal innerhalb des `a`-Elementes auftreten.
`(b|c|d)`	Auswahl, d.h. genau eines dieser Elemente muß innerhalb des `a`-Elementes auftreten.
`(b*)`	Wiederholung(*), d.h. es dürfen 0 bis beliebig viele `b`-Elemente innerhalb des `a`-Elementes auftreten.
`(b+)`	Wiederholung(+); siehe `b*`, aber mindestens 1 `b`-Element
`(b?)`	Optional, d.h. es dürfen 0 oder 1 `b`-Element innerhalb des `a`-Elementes auftreten

Diese Operatoren können auch auf vielfältige Art und Weise miteinander kombiniert werden, die genaue Syntax von X entnehme man der XML-Spezifikation bzw. Abschnitt 3.3.

Im Vergleich zu herkömmlichen regulären Ausdrücken entspricht die Sequenz der Konkatenation, die Auswahl der Vereinigung und die Wiederholung(*) dem Kleene-Stern. Die Wiederholung(+) und die Option kann man als abkürzende Schreibweisen auffassen, d.h. $X+$ steht für $XX*$ und $X?$ steht für $(\epsilon|X)$.

Offenbar definiert jede Modellgruppe eine reguläre Sprache über den vorhandenen Elementnamen, welche Eigenschaften diese Sprache hat, wird später näher untersucht. Bisher wurde nur der Fall betrachtet, daß ein Element ausschließlich andere Elemente enthält. Es ist auch möglich (gemischter Inhalt; engl. mixed content type), daß ein Element Text gemischt mit Elementen enthält, wobei es hier in der DTD nur möglich ist anzugeben, welche Elemente auftreten dürfen, nicht aber die Reihenfolge oder Häufigkeit einzuschränken, d.h. die genannten Elemente dürfen in beliebiger Reihenfolge beliebig oft auftreten.

Zusätzlich sind für X noch die Zeichenfolgen `ANY` und `EMPTY` erlaubt, `ANY` erlaubt es, gemischten Inhalt mit beliebigen Elementen, deren Typ definiert ist, zu beinhalten, `EMPTY` hingegen steht für Elemente, die keinen Inhalt haben können.

Bemerkung:
Wenn X nicht die Form `EMPTY` hat, kann es trotzdem auftreten, daß ein a-Element in einem Dokument keinen Inhalt hat, insbesondere dann, wenn das leere Wort ϵ in der durch X definierten regulären Sprache enthalten ist.

Über das `<!ATTLIST` a Y`>`-Konstrukt, wobei es zu einem a mehrere geben kann, werden Attribute für das a-Element definiert. Attribute sind reihenfolgeunabhängig, d.h. wenn ein Element mehrere Attribute besitzt, so können diese im XML-Dokument (bezogen auf die textuelle Repräsentation) in beliebiger Reihenfolge innerhalb des Start-Tags des a-Elementes auftauchen. Attribute dürfen auch nur maximal einmal auftreten. Attribute haben Zeichenketten als Werte, wobei sich für diese Zeichenketten aber ein Typ angeben läßt. Ebenso kann angegeben werden, ob ein Attribut optional oder obligatorisch ist, es einen Default-Wert besitzt oder sein Wert sogar konstant ist (`#FIXED`). Natürlich nicht in allen denkbaren, sondern nur den sinnvollen Kombinationen, vgl. XML-Spezifikation [11].

Definition:
Ein wohlgeformtes XML-Dokument ist **gültig** (engl. valid) bezüglich einer DTD, wenn es alle durch diese DTD definierten Restriktionen erfüllt.

Die formale Validierung eines wohlgeformten XML-Dokumentes in Hinsicht auf eine DTD ist ebenfalls in der XML-Spezifikation exakt definiert. Zusammengefaßt wird im wesentlichen geprüft (Wohlgeformtheit vorausgesetzt), ob der Inhalt eines Elementes durch die in der DTD definierte reguläre Sprache darstellbar ist und ob Attribute vorhanden sind, wenn sie es laut DTD sein müssen, ggf. ob sie die richtigen Werte haben und auch, ob der Wert dem definierten Typ entspricht.

Eine DTD definiert eine XML-Anwendung (also eine konkrete XML-Sprache), welche eine (i.allg. unendliche) Teilmenge der Menge der wohlgeformten XML-Dokumente darstellt.

Desweiteren können verschiedene Arten von Entities (`<!ENTITY ...>`-Konstrukt) in der DTD definiert werden, welche in einem XML-Dokument oder wiederum in der DTD (Parameter Entities) referenziert werden können. Ebenso können in einer DTD Notationen (`<!NOTATION ...>`-Konstrukt) definiert werden, welche für die Arbeit mit Nicht-XML-Bestandteilen (also z.B. `JPEG, WAV`) dienen.
Für diese Möglichkeiten der DTD sei an dieser Stelle auf die XML-Spezifikation verwiesen, da sie im Rahmen dieser Diplomarbeit nur in untergeordnetem Maße Bedeutung haben.

Im folgenden werden XML-Sprachen, die durch eine DTD definiert sind, mit DTD-XML-Sprachen bezeichnet, analog wird der Begriff Schema-XML-Sprachen für XML-Sprachen verwandt, welche mit XML-Schema definiert sind.

2.2 Reguläre Sprachen

Zur Wiederholung nun einige wichtige Aussagen zu regulären Sprachen, da sie eine wesentliche Rolle bei der Definition der Inhaltsmodelle von Elementen spielen. Zur exakten Definition von linkslinearen Grammatiken, regulären Ausdrücken und Varianten von endlichen Automaten sei auf die entsprechende Literatur verwiesen (Schöning [29] und Hopcroft-Ullmann [20]).

Satz 1 *Eine (formale) Sprache $L \subseteq \Sigma^*$ ist regulär* [2]

gdw. es existiert eine rechtslineare Grammatik Gr mit $L = L(Gr)$

gdw. es existiert ein regulärer Ausdruck r mit $L = L(r)$

gdw. es existiert ein nicht-deterministischer endlicher Automat M mit $L = L(M)$

gdw. es existiert ein deterministischer endlicher Automat M' mit $L = L(M')$

An dieser Stelle wird die Definition von endlichen Automaten wiederholt, da endliche Automaten noch eine größere Rolle spielen werden:

Definition (aus [29]):
Ein **endlicher Automat** (EA) M wird spezifiziert durch ein 5-Tupel

$M = (Z, \Sigma, \delta, z_0, E)$

wobei:

- Z die (endliche) Zustandsmenge,
- Σ das endliche Eingabealphabet mit $Z \cap \Sigma = \emptyset$,
- z_0 der Startzustand mit $z_0 \in Z$,
- E die Endzustandsmenge mit $E \subseteq Z$ und
- δ die Zustandsüberführungsrelation mit $\delta \subseteq Z \times \Sigma \times Z$

ist.

M heißt auch nicht nichtdeterministischer endlicher Automat (NEA). Ist δ anstelle obiger Definition als (partielle) Funktion $\delta : Z \times \Sigma \to Z$ definiert, so wird M als deterministischer endlicher Automat (DEA) bezeichnet.

[2] einer der vier Punkte ist als Definition zu wählen

Offenbar ist jeder DEA ein NEA, denn ist δ als Funktion wie oben angegeben definiert, dann ist jede konkrete Instanz von δ auch eine Relation des oben angegebenen Typs.

Bemerkungen:
Alternativ kann δ bei einem NEA auch als (partielle) Funktion $\delta : Z \times \Sigma \to \mathcal{P}(Z)$ definiert werden, diese Darstellung ist isomorph zur oben angegebenen.

Es sei generell vereinbart, daß das leere Wort ϵ immer den Zustand z_i nach z_i überführen kann, das entspricht der Anschauung, daß das Nicht-Abarbeiten eines Eingabezeichens auch zu keiner Zustandsänderung führt. Alternativ kann man ϵ-Übergange einführen, die auch einen Zustand in einen anderen Zustand überführen dürfen, die Mächtigkeit der Automaten ändert sich hierdurch nicht. Weiterhin kann man auch vereinbaren, daß statt einem Startzustand auch eine Menge von Startzuständen erlaubt ist.

Arbeitsweise eines endlichen Automaten

Ein EA befindet sich immer in genau einem Zustand z_i, zu Beginn ist dies der Startzustand z_0. Sukzessive wird das Eingabewort $w = a_1 \ldots a_i \ldots a_n$ Zeichen für Zeichen gelesen und ausgehend vom aktuellen Zustand und aktuellem Zeichen ein Zustandsübergang (auch Transition genannt) ausgeführt, d.h. der Automat ist danach i.allg. in einem anderen Zustand z_j. Nach Abarbeitung des vollständigen Eingabewortes befindet sich der Automat in einem bestimmten Zustand, hier kurz z_e genannt. Ist $z_e \in E$, dann **akzeptiert** der Automat das Wort w.

Der Zustandsübergang wird mit der Überführungsrelation bzw. -funktion δ ausgeführt, d.h. ist $(z_i, a_i, z_j) \in \delta$, wobei der Automat im Zustand z_i und das aktuelle Eingabezeichen a_i sei, dann darf ein Zustandsübergang in den Zustand z_j stattfinden, anderenfalls nicht.

Ist M ein DEA, so ist δ auf (z_i, a_i) entweder definiert oder nicht, im ersten Fall ist z_j eindeutig bestimmt (da δ eine Funktion ist) im anderen Fall sei vereinbart, daß wenn sich der Automat im Zustand z_i befindet und das aktuelle Eingabesymbol a_i ist, M das Eingabewort w nicht akzeptiert. Ist M ein NEA aber kein DEA, so akzeptiert M das Wort w gdw. es eine Folge von Zustandsübergängen mit w von z_0 nach z_e mit $z_e \in E$ gibt. Führt eine Folge von Zustandsübergängen in einen Zustand z_i und ist das aktuelle Eingabezeichen a_i und es gibt kein Tripel in Delta, welches mit (z_i, a_i) beginnt, so beendet der Automat seine Arbeit so, als ob das Wort komplett abgearbeitet worden wäre und wir einen Zustand erreicht hätten, der nicht Element von E ist.

Im Falle des DEA erweitern wir δ auf Σ^* zu $\hat{\delta} : Z \times \Sigma^* \to Z$, nennen die erweiterte Funktion später aber ebenfalls δ, wenn die Unterscheidung nicht wichtig ist. $\hat{\delta}$ sei wie folgt definiert:

$$\begin{array}{rcll} \hat{\delta}(z_i, \epsilon) & := & z_i & \text{für alle } z_i \in Z \\ \hat{\delta}(z_i, aw) & := & \hat{\delta}(\delta(z_i, a), w) & \text{für alle } a \in \Sigma \text{ und } w \in \Sigma^* \end{array}$$

Ist $M = (Z, \Sigma, \delta, z_0, E)$ ein DEA so bezeichnet $L(M)$ die vom Automaten erkannte Sprache, das sind informal die Worte aus Σ^*, die M ausgehend von z_0 in einen Endzustand überführen.

Formal:

$$L(M) \quad := \quad \{w \in \Sigma^* \mid \hat{\delta}(z_0, w) \in E\}$$

Analog können wir δ für einen NEA erweitern, allerdings definieren wir dann $\hat{\delta}$ als Funktion $\hat{\delta} : \mathcal{P}(Z) \times \Sigma^* \to \mathcal{P}(Z)$ wie folgt:

$$\begin{array}{rcll} \hat{\delta}(Z', \epsilon) & := & Z' & \text{für } Z' \subseteq Z \\ \hat{\delta}(Z', aw) & := & \hat{\delta}(\bigcup_{z_i \in Z'} \{z_j \mid (z_i, a, z_j) \in \delta\}, w) & \text{für } Z' \subseteq Z, a \in \Sigma,\ w \in \Sigma^* \end{array}$$

Mit der Vereinbarung, daß die Vereinigung ($\bigcup$) von 0 Mengen die leere Menge ($\emptyset$) ergibt, gilt dann auch:

$$\hat{\delta}(\emptyset, w) \quad = \quad \emptyset \quad \text{für alle } w \in \Sigma^*$$

Ist $M = (Z, \Sigma, \delta, z_0, E)$ ein NEA, so bezeichnet $L(M)$ die vom Automaten erkannte Sprache, das sind informal die Worte aus Σ^*, für die es irgendeine Folge von Zustandsübergängen mit w ausgehend von z_0 zu einem Endzustand gibt.

Formal:

$L(M) := \{w \in \Sigma^* \mid \hat{\delta}(\{z_0\}, w) \cap E \neq \emptyset\}$

Im folgenden findet die Darstellung als rechtslineare Grammatik keine besondere Betrachtung, es wird ausschließlich mit regulären Ausdrücken (in der Form von Inhaltsmodellen) und endlichen Automaten gearbeitet.

Die Sprache eines Inhaltsmodells[3] E ist analog den regulären Ausdrücken wie folgt rekursiv definiert:

$E = a,\ a \in \Sigma$	$L(E) = \{a\}$
$E = (E_1, E_2, \ldots, E_n)$	$L(E) = \{w_1, \ldots, w_n \mid \forall i = 1 \ldots n : w_i \in L(E_i)\}$
$E = (E_1 \vert E_2 \vert \ldots \vert E_n)$	$L(E) = L(E_1) \cup L(E_2) \cup \ldots \cup L(E_n)$
$E = (E_1)?$	$L(E) = L(E_1) \cup \{\epsilon\}$
$E = (E_1)+$	$L(E) = \{w_1, \ldots, w_n \mid n \geq 1\, und\, \forall i = 1 \ldots n : w_i \in L(E_i)\}$
$E = (E_1)*$	$L(E) = L((E_1)+) \cup \{\epsilon\}$

Definition:
Ein regulärer Ausdruck r **subsummiert** bzw. **überdeckt** einen regulären Ausdruck r' :gdw. $L(r) \supseteq L(r')$. Analog gilt das für Inhaltsmodelle.

2.3 Nicht-1-Mehrdeutigkeit bei regulären Sprachen

Jede Grammatik hat die Eigenschaft, entweder *eindeutig* (auch: nicht-mehrdeutig; engl. non-ambiguous) oder *mehrdeutig* (engl. ambiguous) zu sein. Eine Grammatik Gr wird als mehrdeutig bezeichnet, wenn es für ein Wort $w \in L(Gr)$ zwei Ableitungsbäume gibt.

Diese Eigenschaft bezieht sich wohlgemerkt auf Grammatiken, nicht auf Sprachen, denn i.allg. gilt, es existieren Grammatiken Gr_1 und Gr_2 mit $Gr_1 \neq Gr_2$ aber $L(Gr_1) = L(Gr_2)$, wobei es sein kann, daß Gr_1 eindeutig und Gr_2 mehrdeutig ist. Eine Sprache, für die überhaupt eine Grammatik existiert, wobei aber jede Grammatik, die diese Sprache beschreibt mehrdeutig ist, heißt *inhärent mehrdeutige* Sprache.

Die Begrifflichkeit der Ein- bzw. Mehrdeutigkeit wird analog auf reguläre Ausdrücke (und damit auch Inhaltsmodelle) erweitert. Hierfür werden sogenannte markierte Inhaltsmodelle eingeführt, informal ist ein markiertes Inhaltsmodell E' zu einem Inhaltsmodell E syntaktisch identisch zu E aufgebaut, nur das alle Symbole mit einem Index versehen sind, so daß alle Symbole in E' paarweise verschieden sind. E' ist offenbar nicht eindeutig bestimmt.

Beispiel:
$E = a(a|b)* \rightarrow E' = a_1(a_2|b_3)*$

Definition:
E' ist ein **markiertes** Inhaltsmodell :gdw.
E' ist ein Inhaltsmodell über dem Alphabet $\Pi = \{v_i \mid v \in \Sigma\ und\ i \in \mathbb{N}\}$ wobei alle Symbole in E' paarweise verschieden sind.
Sei $um : \Pi \rightarrow \Sigma$ wie folgt definiert: $um(v_i) := v$ und erweitert auf Inhaltsmodelle und Π^* in naheliegender Weise. Dann korrespondiert zu dem markierten Inhaltsmodell E' das nicht markierte Inhaltsmodell $E := um(E')$.

Offenbar gilt: $L(um(E')) = um(L(E'))$

[3] Hierbei wurden redundante Klammern und Whitespaces vernachlässigt, sowie die Tatsache, daß die XML-Spezifikation auf höchster Ebene eine Klammer erfordert, also Inhaltsmodelle wie z.B. `element1|element2` sind nicht erlaubt, sondern nur `(element1|element2)`.

Ist $w \in L(E)$ dann ergeben sich analog den Ableitungen bei Grammatiken einige Zusammenhänge: Ist der oberste Operator bei E die Sequenz (z.B. $E = E_1, E_2$), so läßt sich w zerlegen in Teilworte (z.B. $w = uv$) wobei dann die Teilworte aus entsprechenden Sequenzmitgliedern abgeleitet sind (z.B. u aus E_1 und v aus E_2). Bei gegebenen Wort w stellt sich hier dann die Frage, welches Teilwort aus welchem Sequenzmitglied abgeleitet wurde. Analog ist die Problematik beim Auswahl-Operator.

Informal ist ein Inhaltsmodell E (bzw. ein regulärer Ausdruck) genau dann mehrdeutig, wenn es ein Wort $w \in L(E)$ gibt, das mehrere Ableitungen aus dem Inhaltsmodell E besitzt.

Beispiel:
$a?(a|b)*$ ist mehrdeutig, denn das Wort a kann aus dem ersten oder dem zweiten Sequenzmitglied abgeleitet werden, wobei aus dem jeweiligen anderen Sequenzmitglied dann ϵ abgeleitet wird.

Definition:
Ein Inhaltsmodell E ist **nicht-mehrdeutig** (engl. unambiguous) :gdw.
$\forall uxv, uyv \in L(E') : um(uxv) = um(uyv) \Rightarrow x = y$, wobei $x, y \in \Pi$ und $u, v \in \Pi^*$.

Definition:
Ein Inhaltsmodell E ist **nicht-1-mehrdeutig** (engl. 1-unambiguous) :gdw.
$\forall uxv, uyw \in L(E') : um(ux) = um(uy) \Rightarrow x = y$, wobei $x, y \in \Pi$ und $u, v, w \in \Pi^*$.

Beispiel:
Das Inhaltsmodell $E = a^*b?a^*$ ist mehrdeutig, denn mit $E' = a_1^*b_1?a_2^*$ gilt: $a_1, a_2 \in L(E')$ und $um(a_1) = um(a_2) = a$.

Das Inhaltsmodell $E = a^*b?a$ ist nicht-mehrdeutig aber 1-mehrdeutig, denn mit $E' = a_1^*b_1?a_2$ gilt: $a_1a_2, a_2 \in L(E')$ und $um(a_1) = um(a_2) = a$.

Bemerkung:
Sprachen, für die es einen nicht-1-mehrdeutigen Ausdruck gibt, werden auch als nicht-1-mehrdeutige Sprachen bezeichnet. Weiterhin folgt aus der Nicht-1-Mehrdeutigkeit natürlich die Nichtmehrdeutigkeit.

Brüggemann-Klein und Wood geben in [12] auch eine unmittelbar einsichtige Charakterisierung für reguläre Ausdrücke an:

Sei E ein regulärer Ausdruck und $a \in \Sigma$.

$Sym(E)$ ist die Menge der auftretenden Symbole in E.

Beispiel:
$E = abc?a \rightarrow Sym(E) = \{a, b, c\}$

Wir definieren:

$$\begin{aligned}
First(E) &:= \{b \mid b \in \Sigma \, und \, \exists w \in \Sigma^* : bw \in L(E)\} \\
Last(E) &:= \{b \mid b \in \Sigma \, und \, \exists w \in \Sigma^* : wb \in L(E)\} \\
Follow(E, a) &:= \{b \mid b \in \Sigma \, und \, \exists v, w \in \Sigma^* : vabw \in L(E)\}
\end{aligned}$$

Lemma 1 *(Brüggemann-Klein und Wood [12]):*
Der reguläre Ausdruck E ist nicht-1-mehrdeutig gdw.
$(\forall x, y \in First(E') : x \neq y \Rightarrow um(x) \neq um(y))$
und
$(\forall z \in Sym(E') \, und \, x, y \in Follow(E', z) : x \neq y \Rightarrow um(x) \neq um(y))$

Begründung:
Ist E' markiert, so folgt aus $u_1zv_1, u_2zv_2 \in L(E')$ auch $u_1zv_2 \in L(E')$, was erkennbar wird, wenn wir den Vorgang der Ableitung eines Wortes aus einem regulären Ausdruck betrachten: **z**

ist genau eine Position im regulären Ausdruck und es gibt keine Möglichkeit, Abhängigkeiten von Teilworten die zu den regulären Teilausdrücken vor und nach **z** gehören auszudrücken.
$\Rightarrow$:
Sei $x \neq y$.
Angenommen, die erste Bedingung ist verletzt, dann gibt es zwei Worte in $L(E')$ die mit x bzw. y beginnen und $um(x) = um(y)$, was direkt im Widerspruch zur Nicht-1-Mehrdeutigkeit von E steht.
Angenommen, die zweite Bedingung ist verletzt, für alle Worte $uzxv$ und $uzyv$ gilt wegen der Nicht-1-Mehrdeutigkeit von E $um(x) \neq um(y)$.
x sei Element von $Follow(E', z)$ wegen $u_1zxv_1 \in L(E')$, y entsprechend wegen $u_2zyv_2 \in L(E')$. Wegen obiger Bemerkung gilt: $u_1zyv_1 \in L(E')$. Also ist $um(x) \neq um(y)$.
$\Leftarrow$:
Angenommen, E ist nicht nicht-1-mehrdeutig,
d.h. es existieren Worte $uxv, uyw \in L(E')$ mit $x \neq y$ und $um(x) = um(y)$.
Ist $u = \epsilon$, so sind $x, y \in First(E')$, also $um(x) \neq um(y)$ im Widerspruch zur Annahme.
Ist $u = u'z$ für $z \in \Pi$, $u' \in \Pi^*$, so $x, y \in Follow(E', z)$ und somit ebenfalls $um(x) \neq um(y)$ im Widerspruch zur Annahme.
Also ist E nicht-1-mehrdeutig.
q.e.d.

Folgende Aussagen für reguläre Sprachen und reguläre Ausdrücke bzw. Inhaltsmodelle sind bekannt:

- Für jede reguläre Sprache L gibt es einen nicht-mehrdeutigen regulären Ausdruck r mit $L = L(r)$. (vgl. [8], Proposition 8)
- Reguläre Sprachen sind abgeschlossen unter Vereinigung, Schnitt, Komplement, Konkatenation und Kleene-Stern. (vgl. [29], S.48).
- Folgende Probleme sind entscheidbar:
  Wortproblem, Leerheitsproblem, Endlichkeitsproblem, Äquivalenzproblem.
- Die Menge der nicht-1-mehrdeutigen Sprachen ist eine echte Teilmenge der regulären Sprachen, sie ist auch nicht abgeschlossen unter Vereinigung, Konkatenation und Kleene-Stern. (vgl. [12])
- Es ist effizient entscheidbar, ob ein Inhaltsmodell E nicht-1-mehrdeutig ist (vgl. [12]).
- Es ist effizient entscheidbar, ob eine die Sprache eines gegebenen deterministischen endlichen Automaten nicht-1-mehrdeutig ist (vgl. [12]). (Konstruktion ist aber exponential)

Die SGML und XML-Spezifikationen fordern, daß das Inhaltsmodell „deterministisch“ sei, was dort auch mit „unambiguous“ bezeichnet wird, nach den hier getroffenen Definitionen aber nicht-1-mehrdeutig (engl. 1-unambiguous) bedeutet.

Die theoretischen Grundlagen für die Nicht-1-Mehrdeutigkeit

Die Grundlage hierfür liefert die Arbeit von Brüggemann-Klein und Wood [12], in der sich intensiv mit den nicht-1-mehrdeutigen Sprachen auseinandergesetzt wird. Brüggemann-Klein und Wood liefern unter anderem einen Entscheidungsalgorithmus, welcher später auch die Grundlage für den Algorithmus von Ahonen, der einen DEA in einen nicht-1-mehrdeutigen DEA transformiert, bildet. An dieser Stelle wird nur das eine wesentliche Theorem F aus [12] wiederholt, welches als Grundlage für den Algorithmus von Ahonen zur Erzeugung der Nicht-1-Mehrdeutigkeit dient.

Hierzu zunächst die benötigten Definitionen:

Sei $M = (Z, \Sigma, \delta, z_0, E)$ ein DEA:

Für ein $z_i \in Z$ sei der **Orbit** von z_i (kurz: $O(z_i)$) wie folgt definiert:
$O(z_i) := \{z_j \mid \exists v, w \in \Sigma^* \; mit \; \delta(z_i, v) = z_j \; und \; \delta(z_j, w) = z_i\}$

Informal ist der Orbit eines Zustandes also die Menge aller Zustände, welche von erstem erreichbar sind und von denen aus der erste ebenfalls wieder erreichbar ist, natürlich gilt $z \in O(z)$, da $\delta(z, \epsilon) = z$ ist.

Erreichbarkeit ist transitiv, die Definition der Orbits selbst symmetrisch, so daß die Zerlegung von Z in Orbits eine Äquivalenzklasseneinteilung ist, weiterhin können wir eine teilweise Ordnung über den Orbits über die Erreichbarkeit definieren, da es nicht möglich ist, daß von Orbit A der Orbit B erreichbar ist und umgekehrt, denn dann wären alle Zustände der Orbits A und B im selben Orbit.

Gilt $O(z) = \{z\}$ und gibt es kein $(z, a, z) \in \delta$ für irgendein $a \in \Sigma$, so nennen wir $O(z)$ **trivial**.

Ein Zustand z_i heißt **Tor** eines Orbits, gdw. $z_i \in E$ gilt oder es existiert $(z_i, a, z_j) \in \delta$ für irgendein $a \in \Sigma$ und $z_j \notin O(z_i)$.

M hat die **Orbit-Eigenschaft** :gdw.
für alle Orbits K von M: für alle Tore $z_i, z_j \in K$ gilt:
$z_i \in E$ gdw. $z_j \in E$
und für $z_k \notin K$: $(z_i, a, z_k) \in \delta$ gdw. $(z_j, a, z_k) \in \delta$

Ein Symbol $a \in \Sigma$ ist **M-konsistent**, wenn es einen Zustand $f(a) \in Z$ gibt, wobei für alle $z \in E$ gilt: $(z, a, f(a)) \in \delta$.

Eine Menge S von Symbolen ist M-konsistent gdw. jedes Element von S ist M-konsistent.

Für M und eine Menge von Symbolen S ist der **S-Schnitt** von M, bezeichnet mit M_S, der Automat, der aus M hervorgeht, wenn alle Transitionen der Form (z, a, z') weggelassen werden, für alle $z \in E$ und $a \in S$.

Für M und ein $z \in Z$ ist der DEA M_z der **Orbit-Automat für z** der aus M wie folgt hervorgeht:

$$\begin{aligned} M_z &:= (O(z), \Sigma, \delta', z, \{z_i \,|\, z_i \; ist \; Tor \; von \; O(z) \; in \; M\}) \\ \text{mit } \delta' &:= \{(z_i, a, z_j) \,|\, (z_i, a, z_j) \in \delta \; und \; z_i, z_j \in O(z)\} \end{aligned}$$

Das bedeutet informal, daß M auf die Zustände von $O(z)$ eingeschränkt wird, z der Startzustand ist und die Menge der Tore von $O(z)$ in M die Menge der Endzustände ist. Die Sprache $L(M_z)$ wird entsprechend als **Orbit-Sprache für z** bezeichnet.

Satz 2 (Nicht-1-Mehrdeutigkeit) *(Brüggemann-Klein und Wood in [12])*
Ist M ein minimaler DEA und S eine Menge M-konsistenter Symbole. Dann ist $L(M)$ nicht-1-mehrdeutig gdw.

M_S hat die Orbit-Eigenschaft
und
Jede Orbit-Sprache von M_S ist nicht-1-mehrdeutig

Weiterhin gilt: Besteht M nur aus einem einzigen nicht-trivialen Orbit, dann existiert ein M-konsistentes Symbol.

Beweis: in [12]

Bemerkung:
Die Orbit-Sprachen von trivialen Orbits sind nicht-1-mehrdeutig, denn sie haben die Form $\{\epsilon\}$.

Aus dem Satz 2 leiten Brüggemann-Klein und Wood direkt einen möglichen Algorithmus für den Test eines (minimalen) DEAs auf Nicht-1-Mehrdeutigkeit ab:

Algorithmus Entscheide_Nicht-1-Mehrdeutigkeit:

Eingabe: Ein minimaler DEA M
Rückgabe: ja/nein

(1) Falls M nur aus einem trivialen Orbit besteht, dann Rückgabe ja
(2) Berechne Menge S der M-konsistenten Symbole
(3) Falls M nur aus genau einem nichttrivialen Orbit besteht und $S = \emptyset$
(4) dann Rückgabe $nein$
(5) Berechne M_S
(6) Falls M_S nicht die Orbit-Eigenschaft hat, dann Rückgabe $nein$
(7) Für alle K Orbit von M_S, für ein $x \in K$:
(8) *Falls Entscheide_Nicht-1-Mehrdeutigkeit*($(M_S)_x$)=$nein$
(9) dann Rückgabe $nein$
(10) Rückgabe ja

Lemma 2 *(aus [12])*
Ist M ein minimaler DEA, der die Orbit-Eigenschaft hat, dann ist für jedes $x \in Z$ der Orbit-Automat M_x minimal.

Beweis: in [12]
Grundidee ist die folgende: In M ist jedes Paar von Zuständen *unterscheidbar*, d.h. für alle Zustände z_i, z_j mit $i \neq j$ gilt: es existiert $w \in \Sigma^*$ mit $\delta(z_i, w) \in E$ und $\delta(qj, w) \notin E$ bzw. umgekehrt.

Der Minimierungsalgorithmus (vgl. [20]) stellt diese Eigenschaft beim minimierten Automaten sicher und wenn es keine überflüssigen Zustände gibt – das sind Zustände, die vom Startzustand nicht erreichbar sind oder von denen kein Endzustand erreichbar ist – ist diese Eigenschaft sogar hinreichend.

Weiterhin ist bekanntermaßen jeder minimale DEA M' mit $L(M') = L(M)$ isomorph zu M, d.h. bis auf das Umbenennen von Zuständen identisch. Da M_x nur aus genau einem Orbit besteht, gibt es in M_x keine überflüssigen Zustände, denn jeder Zustand ist von jedem anderen Zustand erreichbar.

Annahme: M_x ist nicht minimal, das ist gdw. es existieren $z_i, z_j \in Z(M_x)$ mit $i \neq j$ und z_i, z_j sind nicht-unterscheidbar. O.B.d.A. hat das Wort $w \in \Sigma^*$ aber z_i, z_j in M unterschieden.

1.Fall: w wird innerhalb von $O(z_i) = O(z_j)$ abgearbeitet, d.h. eine Erkennung von w in M endet in einem Endzustand, die andere nicht. Der Endzustand ist ein Tor, der andere Zustand aber nicht, also ist der Endzustand von M ein Endzustand in M_x, der andere Zustand ist wegen der Orbit-Eigenschaft kein Endzustand in M_x, also unterscheidet w auch in M_x die Zustände z_i und z_j.

2.Fall: Die Abarbeitung von w verläßt in mindestens einem Fall $O(z_i) = O(z_j)$. Bei der Erkennung von w muß $O(z_i)$ an einer Stelle, d.h. einem Tor, verlassen worden sein. Durch den Determinismus sind alle Zustandsübergänge eindeutig bestimmt. Wenn $w = a_1...a_i...a_n$ ist und beide Erkennungsvorgänge $O(z_i)$ mit a_i verlassen, folgt aus der Orbit-Eigenschaft, daß $\delta(z_i, a_1...a_i) = \delta(z_j, a_1...a_i)$ gilt, womit w natürlich nicht z_i und z_j unterscheiden könnte. Verläßt ein Erkennungsvorgang $O(z_i)$ mit a_i, so kann also der andere Erkennungsvorgang vor der Abarbeitung von a_i nicht in einem Tor sein. Tore sind aber genau die Endzustände in M_x, so daß also im Widerspruch zur Annahme das Wort $a_1...a_{i-1}$ die Zustände z_i und z_j in M_x unterscheidet.
q.e.d.

Dieses Lemma ermöglicht es, Zeile (8) so zu definieren, wie es oben geschah, anstelle der aufwendigeren Version:

(8') Falls *Entscheide_Nicht-1-Mehrdeutigkeit*(minimiere($(M_S)_x$))=$nein$

Lemma 3 *Der Algorithmus Entscheide_Nicht-1-Mehrdeutigkeit terminiert.*

Beweis:
Da die Anzahl der Zustände der Automaten mit denen der rekursive Aufruf erfolgt monoton – aber nicht streng monoton – abnimmt, und mit der Bildung des S-Schnittes das gleiche für die Anzahl der Zustandsübergänge gilt, wobei die Ausgangsanzahlen endlich waren, wäre eine Endlosschleife theoretisch nur für den Fall möglich, wenn $M = (M_S)_x$ gilt. Das ist wiederum nur möglich, wenn M nur einen einzigen Orbit besitzt und $S = \emptyset$ gilt. Dann aber terminiert der Algorithmus entweder in Zeile (1) oder in Zeile (4).

Lemma 4 Lemma (ähnlich in [12][4]):
Sei $M = (Z, \Sigma, \delta, z_0, E)$ ein minimaler DEA der nur aus einem Orbit besteht, es sei $x \in Z$ und $|O(x)| \geq 2$ und M habe die Orbit-Eigenschaft.
Dann gilt:
Für alle $y \in O(x)$: $L(M_x)$ ist nicht-1-mehrdeutig gdw. $L(M_y)$ ist nicht-1-mehrdeutig.

Beweis:
Brüggemann-Klein und Wood zeigen die Aussage mit Hilfe der Linksquotienten (Definition in Abschnitt 3.2) von Sprachen, und vielen Aussagen zu diesen, hier nun ein Beweis, der mit den bisher eingeführten Definitionen und Sätzen auskommt.

Ersetzen wir Zeile (7) durch Zeile (7') so erhalten wir einen Algorithmus A', dessen Korrektheit sofort aus dem Satz 2 folgt.

(7') Für alle K Orbit von M_S, $\forall x \in K$

Zum Beweis betrachten wir den Ablauf des Algorithmus A', wobei wir den Algorithmus auf alle Automaten der Form $M = (Z, \Sigma, \delta, z_i, E)$, $z_i \in Z$ beliebig anwenden. Es ist offenbar hinreichend zu zeigen, daß der Algorithmus auf jedem dieser Automaten zum gleichen Ergebnis kommt. Hierfür betrachten wir o.B.d.A. für $x, y \in Z$ die Automaten $M_1 = (Z, \Sigma, \delta, x, E)$ und $M_2 = (Z, \Sigma, \delta, y, E)$.

Nach Lemma 2 sind M_1 und M_2 minimal, denn M hat die Orbit-Eigenschaft und es gilt $M_1 = M_x$ und $M_2 = M_y$ da M nur einen einzigen Orbit besitzt. A' berechnet die Menge S der M-konsistenten Symbole und prüft die Orbit-Eigenschaft, beides ist unabhängig vom konkreten Startzustand, ebensowenig wird in Zeile (7') Bezug auf den Startzustand genommen. Somit ist klar, daß A' auf beiden Automaten das gleiche Ergebnis liefert.
q.e.d.

Lemma 5 *Der Algorithmus Entscheide_Nicht-1-Mehrdeutigkeit ist korrekt.*

Begründung:
Die partielle Korrektheit folgt sofort aus dem Satz 2, dem Lemma 4 mit $M := M_S$ und Lemma 2. Mit Lemma 3 folgt sofort, daß dann auch die totale Korrektheit gilt.

Bemerkung:
Brüggemann-Klein und Wood zeigen, daß der Algorithmus eine quadratische Zeitkomplexität in der Größe von M hat.

[4] Brüggemann-Klein und Wood erlauben konkret mehr als einen Orbit

2.4 XML-Dokumente als Baum

Bisher waren wir in den Betrachtungen von einer sequentiellen Zeichendarstellung der XML-Dokumente ausgegangen. Die durch die Schachtelung der Elemente untereinander gegebene Hierarchie legt natürlich die Darstellung von XML-Dokumenten als geordneten Wurzelbaum nahe. XML-Prozessoren verwendet oftmals intern ebenfalls eine Baumdarstellung der Dokumente, z.B. das Document Object Model (DOM), wobei wir bei diesem genau genommen einen Wald haben.

Für die Baumdarstellung gibt es mehrere Vorschläge, so etwa der xtree von Behrens [8] und der Vorschlag von Fan und Libkin [14] sowie natürlich das DOM-Modell selbst.

Die Unterschiede liegen hauptsächlich in der Attributbehandlung, entweder werden sie ignoriert (xtree) den Knoten zugeordnet (DOM) oder sind Knoten im Graph.

Grundsätzlich benötigen wir einen zusammenhängenden, gerichteten, azyklischen Graphen. Aus der Forderung nach Zusammenhang und Azyklizität ergibt sich automatisch die Baumeigenschaft. Weiterhin müssen wir fordern, daß es einen Wurzelknoten gibt, also einen Knoten, von welchem es gerichtete Wege zu allen anderen Knoten gibt. Desweiteren muß für jeden Knoten eine Ordnung über allen ausgehenden Bögen dieses Knotens definiert sein.

Jedem Knoten wird eine Menge bestehend aus (Attributname,Attributwert)-Paaren funktional zugeordnet, wobei ein Attribut höchstens einmal pro Knoten auftreten darf. Die Knoten entsprechen Elementen, daher handelt es sich um einen beschrifteten (engl. labeled) Graphen, wobeo hier die Knoten und nicht die Kanten beschriftet sind.

Definition:

Ein **XMLtree** T ist eine Struktur $T = (V, E, name, children, attr, value)$ für die gilt:
(V, E) ist ein (endlicher) gerichteter Graph für den gilt:

1. $V \neq \emptyset$
2. G ist azyklisch [5]
3. $|E| = |V| - 1$
4. $\forall v \in V$ existiert höchstens ein $e = (v', v) \in E$

Der einzige Knoten ohne einlaufende Kanten sei Wurzelknoten genannt, bezeichnet mit $root(T)$, Knoten ohne ausgehende Kanten seien bezeichnet als Blattknoten.
Ist $(v', v) \in E$, dann heiße v Kindknoten von v' und v' heiße Vaterknoten von v.

$name$ ist eine Funktion $V \rightarrow Bezeichner \cup \{„\#PCDATA“\}$, wobei Bezeichner wie $Name$ in der XML-Spezifikation definiert sei[6], es gelte weiterhin: $name(v) = „\#PCDATA“ \Rightarrow v$ ist Blattknoten.

$children$ ist eine Funktion $V \rightarrow V^*$, wobei $children(v)$ ein n-Tupel über Knoten aus V ist, gdw. wenn v n Kindknoten hat und $v' \in children(v)$ gdw. v' ist Kindknoten von v.

$attr$ ist eine Funktion $V \rightarrow (Bezeichner \rightarrow_{part} String)$ wobei $String$ die Menge der laut XML Spezifikation erlaubten Attributwerte darstellt und $attr(v)$ eine endliche partielle Funktion der Menge der Bezeichner (vgl. $name$) nach String ist. Weiterhin gelte: $name(v) = „\#PCDATA“ \Rightarrow attr(v) = \emptyset$

$value$ ist eine Funktion $V|_{\{v|name(v)=„\#PCDATA“\}} \rightarrow String$

[5] Aus (1)-(4) zusammen folgt auch, daß der zugehörige ungerichtete Graph azyklisch ist.
[6] siehe hierzu auch Kapitel 4

Bemerkung:
Durch die Darstellung als Baum erfüllt ein XMLtree automatisch viele Wohlgeformtheitsbedingungen, insbesondere die der korrekten Tag-Schachtelung. Setzt man die oben getroffenen Vereinfachungen und korrekte Attributwerte voraus, so ist solch ein XMLtree wohlgeformt, aber natürlich i.Allg. nicht gültig zu einer gewissen DTD d. Wohlgeformt setzt genau genommen eine textuelle Darstellung des XML-Dokumentes voraus, ein XMLtree T sei wohlgeformt :gdw. $serialize(T)$ ist wohlgeformt, $serialize()$ wird weiter unten definiert.

Beachte weiterhin, daß ein xmltree keine Typen für die Attribute festlegt. Die Darstellung eines XML-Dokumentes als Baum hat in dieser Form auch den Vorteil, daß `CDATA`-Abschnitte im Prinzip nicht nötig sind. Für die Bijektion auf die textuelle Darstellung ist es aber vorteilhaft, sie beizubehalten, da sonst gegebenenfalls `CDATA`-Abschnitte bei der Transformation erzeugt werden müßten.

Sei $tree(v)$ für $v \in V$ definiert als der Teilbaum mit v als Wurzel, d.h. $root(tree(v)) = v$ und sei $tree()$ erweitert auf $\{v_1, \ldots, v_n\} \subseteq V$ wie folgt:
$tree(\{v_1, \ldots, v_n\}) := \{tree(v_1), ..., tree(v_n)\}$

Satz 3 *Jedem XMLtree T läßt sich bijektiv ein (wohlgeformtes) XML-Dokument d zuordnen, wenn die Reihenfolge der Attribute vernachlässigt wird. Die Baumdarstellung ist also isomorph zur Textrepräsentation.*

Beweis:

Sei $d := serialize(T)$ mit
$\mathbf{serialize(T_1, \ldots, T_n)} := serialize(T_1) \ldots serialize(T_n)$
$\mathbf{serialize(T)} :=$

<$name(root(T))$ $serialize(attr(root(T)))$>
$serialize(tree(children(root(T)))$
</$name(root(T))$>

für $name(root(T)) \neq$ „$\#PCDATA$“
$\mathbf{serialize(T)} := value(root(T))$ für $name(root(T)) =$ „$\#PCDATA$“
sowie sei $serialize()$ für die Attribute die Konkatenation der Zeichenfolgen „*bez=‚attr(v)(bez)‘*“ für jeden Bezeichner bez für den $attr(v)$ definiert ist, wobei falls $attr(v)(bez)$ das Zeichen (‘) enthält, statt dem oben benutzen (‘) das Zeichen (“) zu benutzen ist.

Kursive Schrift kennzeichnet Funktionsaufrufe, deren Ergebnis sich anstelle der Aufrufe zu denken ist.
q.e.d.

Haben wir eine Grammatik Gr gegeben deren Sprache $L(Gr)$ XML-Dokumente in textueller Darstellung sind, so können wir mit

$$\{T : T\ xmltree\ und\ \exists l \in L(Gr)\ mit\ serialize(T) = l\}$$

zur zugehörigen Baummenge übergehen. Andererseits ist es bei der Arbeit mit Bäumen wesentlich einfacher, eine Baumgrammatik zu benutzen, der Unterschied zwischen einer Grammatik und einer Baumgrammatik ist im wesentlichen, daß mit einer Grammatik Zeichenketten abgeleitet werden und mit einer Baumgrammatik Bäume.

Auch bei Baumgrammatiken gibt es die Klassifikation in reguläre Baumgrammatiken, kontextfreie Baumgrammatiken usw. Im folgenden wird nicht die Definition einer allgemeinen Baumgrammatik gegeben, sondern im Hinblick auf die weiteren Betrachtungen gleich die Definition einer regulären Baumgrammatik, die [23] entnommen ist:

Definition:
Eine **reguläre Baumgrammatik** ist ein 4-Tupel $G = (N, \Sigma, P, S)$ wobei

- N die Menge der Nichtterminale ist
- Σ die Menge der Terminale ist
- P eine Menge von Produktionen der Art $X- > a\,Ausdruck$ ist, wobei $X \in N$, $a \in \Sigma$ gilt und Ausdruck eine Modellgruppe über N ist
- S die Menge der Startsymbole mit $S \subseteq N$ ist

Ableitungen mit einer Baumgrammatik funktionieren analog einer „normalen" Grammatik, d.h. beginnend mit einem Startsymbol $s \in S$ werden sukzessive Regeln aus P angewandt, um Nichtterminale zu ersetzen, wobei das Terminalsymbol am Beginn jeder Regel dem Wurzelknoten des (Teil-)Baumes zugeordnet ist und aus der Modellgruppe der restliche Baum abgeleitet wird.

Beispiel (aus [23]):

Der (internen) DTD

```
<!DOCTYPE book [
  <!ELEMENT book (author+, publisher)>
  <!ELEMENT author (#PCDATA)>
  <!ELEMENT publisher EMPTY>
  <!ATTLIST publisher Name CDATA #IMPLIED>
]>
```

kann auf naheliegende Art und Weise folgende äquivalente Baumgrammatik zugeordnet werden:

N = {Book, Author, Publisher, Pcdata}
Σ = {book, author, publisher, #PCDATA}
S = {Book}
P:
Book $\rightarrow$ book(Author+, Publisher)
Author $\rightarrow$ author(Pcdata)
Publisher $\rightarrow$ publisher(ϵ)
Pcdata $\rightarrow$ #PCDATA(ϵ)

Beachte, daß dies nur eine Möglichkeit ist, denn einerseits können wir die Menge der Nichtterminale fast beliebig umbenennen und andererseits können schon zwei verschiedene DTDs zueinander äquivalent sein, wenn die Inhaltsmodelle der gleichen Elemente paarweise äquivalent sind.

Der Einfachheit halber nehmen wir an, daß wir die Funktion $name()$ für den XMLtree vorübergehend über $N \cup \Sigma$ definieren und daß $N \cap \Sigma = \emptyset$ gilt. Beim Ableiten starten wir mit einem XMLtree, der nur aus einem Knoten besteht, für den $name()$ definiert ist als s für ein $s \in S$.

Solange der XMLtree Blätter v_i besitzt mit $name(v_i) \in N$ und es eine Regel in P gibt mit der Form: $name(v_i) \rightarrow a\,CM$, wählen wir ein Wort $w = w_1 \dots w_n \in L(CM) \subseteq N^*$, ordnen v_i dann n neue Kindknoten $v_{w_1}, \dots, v_{w_n}$ zu und definieren $name(v_i) := a$ und für alle $i = 1 \dots n : name(v_{w_i}) := w_i$.

Die Ableitung ist beendet, wenn für alle Knoten v_i des XMLtrees gilt: $name(v_i) \in \Sigma$.

Zu einer regulären Baumgrammatik G ist $TL(G, s)$ die zugehörige *reguläre Baumsprache*, das ist die Menge der Bäume, die sich auf oben angegebene Art und Weise ableiten läßt, wobei als Startsymbol $s \in S$ dient.

Zu jeder regulären Baumgrammatik gibt es eine äquivalente reguläre Baumgrammatik in Normalform. Bevor wir diese einführen, sei noch angemerkt, daß wir o.B.d.A. voraussetzen können, daß es in P für jedes Nichtterminal nur eine einzige Regel gibt, bei der dieses Nichtterminal auf der linken Seite steht, andernfalls führen wir einfache Umbenennungen und Ersetzungen aus [23]:

$A \to a\,X;\ A \to b\,Y$ werden ersetzt durch $A \to a\,X;\ A1 \to b\,Y$ und jedes Auftreten von A auf der rechten Seite einer Regel wird ersetzt durch $(A|A1)$
Regeln der Form $A \to a\,X;\ A \to a\,Y$ werden ersetzt durch $A \to a\,(X|Y)$.

Bei der Normalform trennen wir die Ableitung des Wurzelknotens von der Ableitung des Inhaltsmodells (also der Kindknoten).

Definition:
Eine reguläre Baumgrammatik G in Normalform ist definiert als:

$G = (N_1, N_2, \Sigma, P_1, P_2, S)$

- N_1 ist die Menge von Nichtterminale, die für die Ableitungsbäume benötigt wird
- N_2 ist die Menge der Nichtterminale für die Angabe des Inhaltsmodells
- Σ ist die Menge von Terminalsymbolen
- P_1 sind Regeln der Form $A \to aX$, wobei $A \in N_1, a \in \Sigma$ und $X \in N_2$; für jedes $A \in N_1$ gibt es auch höchstens eine Regel dieser Form in P_1
- P_2 sind Regeln der Form $X \to Ausdruck$, wobei $X \in N_2$ und $Ausdruck$ ist ein Inhaltsmodell über N_1, für jedes $X \in N_2$ gibt es auch höchstens eine Regel dieser Form in P_2
- S ist die Menge der Startsymbole mit $S \subseteq N_1$

Mit oben angegebenen Bemerkungen wird unmittelbar klar, daß wir zu jeder regulären Baumgrammatik die äquivalente Normalform bilden können. Bezogen auf XML- und SGML-Dokumente bedeuten die Regeln aus P_1 die Angabe der *Elementnamen* und die Regeln aus P_2 die Angabe der *Elementtypen*.

Beispiel (aus [23]):

$G = (N_1, N_2, \Sigma, P_1, P_2, S)$

N_1 = {Book, Author, Publisher, Pcdata}
N_2 = {BOOK, AUTHOR, PUBLISHER, PCDATA}
Σ = {book, author, publisher, #PCDATA}
S = {Book}
P_1:
Book → book BOOK
Author → author AUTHOR
Publisher → publisher PUBLISHER
Pcdata → #PCDATA PCDATA
P_2:
BOOK → (Author+, Publisher)
AUTHOR → Pcdata
PUBLISHER → ϵ
PCDATA → ϵ

Weitere Restriktionen an reguläre Baumgrammatiken führen zu Teilklassen von regulären Baumsprachen.

Definition:
Die Funktion **ContentModel()** sei über N_1 wie folgt definiert:
Sei $A \in N_1, B \in N_2, A \to a\,B \in P_1, B \to Ausdruck \in P_2$, dann:
$ContentModel(A) := Ausdruck$.

Beachte, daß es nach Voraussetzung für jedes Element aus N_1 bzw. N_2 nur maximal eine Regel in P_1 bzw. P_2 gibt, wo dieses Element auf der linken Seite steht. Setzen wir weiterhin voraus, daß die Grammatik keine überflüssigen Produktionen enthält – das sind Produktionen, mit denen sich keine „terminalen" Worte bzw. Bäume ableiten lassen – so ist $ContentModel()$ über ganz N_1 wohl-definiert.

Beispiel:

$ContentModel(Book) = (Author+, Publisher)$

Definition:
Eine reguläre Baumgrammatik G ist **baum-lokal** (engl. local tree) :gdw. für jedes Symbol $a \in \Sigma$ existiert höchstens eine Regel der Form $A \to a\,X$.

Entsprechend heißt eine reguläre Baumsprache TL baum-lokal gdw. es existiert eine baum-lokale Baumgrammatik $G = (N_1, N_2, \Sigma, P_1, P_2, \{s\})$ mit $TL = TL(G, s)$

Setzen wir einmal voraus, daß es in P_1 und P_2 keine überflüssigen Produktionen gibt, so heißt das, daß es eine Abbildung von den Elementnamen (hier: a) auf Elementtypen (hier: der $Ausdruck$ der Regel $X \to Ausdruck \in P_2$) gibt. Der Elementname legt also eindeutig das Inhaltsmodell fest, genau das ist auch die Mächtigkeit bei DTDs, wo Elementname und Elementtyp zusammenfallen und es für jeden Elementnamen maximal ein Inhaltsmodell gibt.

Definition:
Eine reguläre Baumgrammatik G ist **ein-typig** (engl. single-type) :gdw.
$\forall A, B, C \in N_1$ mit $A \neq B$ und $A, B \in Sym(ContentModel(C))$ und $A \to a\,Ausdruck_1, B \to b\,Ausdruck_2$ gilt: $a \neq b$

Entsprechend heißt eine reguläre Baumsprache TL ein-typig gdw. es existiert eine ein-typige Baumgrammatik $G = (N_1, N_2, \Sigma, P_1, P_2, \{s\})$ mit $TL = TL(G, s)$

Klarerweise impliziert die Baum-Lokalität die Ein-Typigkeit; Lee, Mani und Murata geben auch ein Beispiel an, so daß klar wird, daß die Teilmengenbeziehung eine echte ist.

Definition:
Eine reguläre Baumgrammatik G ist vom Typ **TDLL(1)** :gdw.
$\forall C \in N_1$: $ContentModel(C)$ ist nicht-1-mehrdeutig.

Dies ist genau die Forderung, die die XML Spezifikation ([11]; Anhang E) und auch die XML-Schema Spezifikation (siehe hierzu Diskussion in Abschnitt 2.7) an eine DTD bzw. ein XML-Schema stellen.

2.5 DTD-XML-Sprachen

Jede konkrete DTD-XML-Sprache L ist eine formale Sprache über einem gewissen Alphabet Σ. L ist natürlich keine kontextfreie Sprache durch die vielen Gültigkeitskriterien, die z.B. fordern, daß innerhalb eines XML-Dokumentes die Werte der Attribute vom Typ ID eindeutig seien müssen. Vielmehr ist es möglich, attributierte kontextfreie Grammatiken für die Darstellung von L zu benutzen, wobei dann natürlich eine kontextfreie Grammatik zu Grunde liegt.

Trotzdem werden oftmals erweiterte kontextfreie Grammatiken (ekfG) benutzt, um DTDs darzustellen. Erweiterte kontextfreie Grammatiken (engl. extended context free grammars) unterscheiden sich von kontextfreien Grammatiken dadurch, daß sie auf der rechten Seite der Grammatikregeln reguläre Ausdrücke über Terminalen und Nichtterminalen erlauben.

Bezüglich der Mächtigkeit unterscheiden sie sich nicht von kontextfreien Grammatiken, nur entsprechen die Regeln der DTD solchen erlaubten erweiterten Produktionen (Isomorphie der Modellgruppe zu einem regulären Ausdruck) und die Ableitungsbäume unterscheiden sich wesentlich von denen normaler kontextfreier Grammatiken, da die Anzahl der Kindknoten nicht immer festgelegt ist (* und $^+$-Operator)

Im folgenden geht es aber nicht um die Grammatik der DTD-XML-Sprachen bzw. Schema-XML-Sprachen als ganzes sondern vielmehr um die regulären Ausdrücke, die einem Element zugeordnet sind und deren Inhaltsmodelle beschreiben.

Die Grammatikregeln für eine reguläre Baumgrammatik $G = (N_1, N_2, \Sigma, P_1, P_2, S)$ ergeben sich recht einfach und naheliegend aus der DTD d:

Σ:= Menge aller in d definierten Elementnamen.
Σ besteht aus Worten über einem Alphabet Σ', sei ‚#'ein Zeichen mit ‚#'$\notin \Sigma'$.
$N_1 := \Sigma.\{\#\}$ [7]
$N_2 := \Sigma.\{\#\#\}$
Sei `<!ELEMENT A Ausdruck>` in d enthalten, wobei *Ausdruck* das Inhaltsmodell über in d definierten Elementen ist. Dann wird die Regel $A\# \rightarrow A\,A\#\#$ zu P_1 und die Regel $A\#\# \rightarrow$ *Ausdruck'* zu P_2 hinzugefügt, wobei *Ausdruck'* so aus `Ausdruck` hervorgeht, daß jedes Symbol B in Ausdruck durch $B\#$ ersetzt wurde.

Ein wesentlicher Punkt ist, betrachtet man die Grammatikregeln der Baumgrammatik, die sich entsprechend der DTD ergeben, daß die Terminale gleich den Nichtterminalen (bis auf #) sind, also Elementnamen genau ein Elementtyp zuordbar ist. Wie oben bereits erwähnt, sind demnach die DTD-XML-Sprachen baum-lokale TDLL(1)-Sprachen.

Beachte, daß bei einer externen DTD – so wie wir es hier voraussetzen – die Information über das Wurzelelement nicht gegeben ist, das geschieht es durch das `DOCTYPE`-Konstrukt im XML-Dokument selbst oder könnte natürlich auch andersweitig festgelegt werden.

Abgeschlossenheit unter der Vereinigung

Die Abgeschlossenheitseigenschaft unter der Vereinigung ist wichtig für die Integration verschiedener DTDs aber auch für die DTD-Generierung selbst.

Eine Idee zur Generierung einer DTD besteht darin, jedes XML-Dokument der Kollektion als DTD-XML-Sprache zu betrachten und dann die Vereinigung aller dieser einzelnen DTD-XML-Sprachen zu betrachten. Wenn diese Vereinigung eine DTD-XML-Sprache ist, dann hat man schon eine DTD, die die Kollektion exakt beschreibt und kann von dieser ausgehend bei Bedarf generalisieren. Offensichtlich ist auch jede XML-Sprache, die nur ein XML-Dokument umfaßt, eine DTD-XML-Sprache bei Vernachlässigung der Attributwerte und Werte der `#PCDATA`-Bereiche.

Bemerkung:
Viele Autoren vernachlässigen in ihrer Betrachtung Attributwerte und fassen Zeicheninhalt von Elementen einfach als `#PCDATA`-Knoten auf, es wird also nur die grobe Baumstruktur analysiert. In dieser Arbeit werden Attributwerte und auch Werte der `#PCDATA`-Bereiche berücksichtigt, insbesondere auch schon im Hinblick auf die vielfältigen Möglichkeiten bei XML-Schema Typen zu definieren.

Wenn wir also einem einzelnen XML-Dokument eine DTD zuordnen, so wird also schon in der Tat generalisiert, denn aus dieser DTD lassen sich XMLtrees ableiten, die einen gleichen Graphen als Gerüst haben, aber verschiedene Werte für Attribute und `#PCDATA`-Bereiche – natürlich noch mit Restriktionen an die Textinhalte und Attributwerte, wobei die Typableitung hierfür später noch diskutiert wird. Diese Generalisierung ist aber in jedem Fall sinnvoll, korrekte

[7] d.h. $abc\# \in N_1$ gdw. $abc \in \Sigma$

Typen (inklusive ggf. der `#FIXED` oder `#DEFAULT`-Angabe) vorausgesetzt, denn ansonsten gibt es i.allg. keine DTD, die genau die DTD-XML-Sprache definiert, welche nur aus diesem einzelnen XML-Dokument besteht.

Sind DTD-XML-Sprachen (in Bezug auf die Baumstruktur) unter der Vereinigung abgeschlossen? D.h. seien die DTD-XML-Sprachen L_1, L_2 gegeben, ist $L_1 \cup L_2$ eine DTD-XML-Sprache?

Es ist naheliegend, bei der Vereinigung einfach eine neue DTD d so zu generieren, daß wenn das Inhaltsmodells eines Elementes A in d_1 (der DTD zu L_1) gleich X ist und das Inhaltsmodell des Elementes A in d_2 (der DTD zu L_2) gleich Y ist, das Inhaltsmodell des Elementes A in d gleich $(X|Y)$ gesetzt wird, natürlich mit evt. noch zu korrigierender Klammerung.

$(X|Y)$ ist nicht notwendig nicht-mehrdeutig (engl. unambiguous), es gibt aber zu jedem regulären Ausdruck – die Inhaltsmodelle sind wie gesagt isomorph zu solchen – einen nicht-mehrdeutigen, der die selbe Sprache beschreibt (vgl. oben). Leider jedoch wird die Nicht-1-Mehrdeutigkeit des Inhaltsmodells gefordert, aber die nicht-1-mehrdeutigen Sprachen sind bekanntermaßen nicht unter der Vereinigung (also dem Auswahl-Operator **|** , vgl. Def. $L(E)$) abgeschlossen.

Auch wenn man zusätzlich noch annimmt, das resultierende Inhaltsmodell sei nicht-1-mehrdeutig, dann ist in diesem eingeschränkten Fall die Vereinigung nicht abgeschlossen, was folgendes Gegenbeispiel veranschaulicht:

eins.xml *zwei.xml* *drei.xml*

```
<A>                  <A>                  <A>
  <B>                  <B>                  <B>
    <C/>                 <F/>                 <C/>
  </B>                 </B>                 </B>
  <D>                  <D>                  <D>
    <E/>                 <G/>                 <G/>
  </D>                 </D>                 </D>
<A>                  <A>                  <A>
```

Betrachten wir jeweils die DTDs zu den Sprachen, welche jeweils nur das Dokument *eins.xml* bzw. *zwei.xml* enthalten und verfahren wie oben vorgeschlagen bzgl. der Vereinigung, so ist auch das Dokument *drei.xml* in dieser vereinigten DTD-XML-Sprache enthalten. Man kann sich leicht überlegen, daß es auch keine andere Möglichkeit gibt, eine DTD zu definieren, die nur Dokument *eins.xml* und *zwei.xml* beschreibt, einfach deshalb, da es nicht möglich ist, einen Bezug der Inhaltsmodelle der Elemente B und D untereinander darzustellen.

Mit der oben genannten Vorgehensweise führen wir also offenbar eine Generalisierung durch, wobei zu untersuchen ist, ob diese sinnvoll ist und ggf. auch noch zu einer weiteren Generalisierung übergegangen werden muß, um die Forderung nach Nicht-1-Mehrdeutigkeit zu erfüllen.

Daraus folgt dann auch unmittelbar, daß es i.allg. nicht möglich ist, zu einer gegebenen Teilmenge der wohlgeformten XML-Dokumente (also einer XML-Sprache) eine DTD zu finden. Die Menge der DTD-XML-Sprachen ist also eine echte Teilmenge aller XML-Sprachen.

2.6 Schema-XML-Sprachen

Ein XML-Schema[8] definiert ein abstraktes Datenmodell, die Darstellung des XML-Schemas in XML-Form ist nur eine mögliche Präsentation. Generell gilt: Element-Deklarationen legen den Element*namen* fest, Element*typen* werden zugeordnet, so daß aus einem Elementnamen nicht zwingend der Elementtyp folgt. Es gilt aber die Restriktion, daß in einer konkreten Instanz zwei

[8]Es gibt viele Schwierigkeiten, die Spezifikation zu interpretieren, vgl. dazu Anhang C.

Kindelemente mit gleichem Namen unterhalb eines bestimmten Elementes den gleichen (exakt: denselben)[9] Typ haben müssen.

Die XML-Schema-Spezifikation ist sehr länglich, nachfolgend deshalb nur ein kurzer Überblick, es ist für das Verständnis unabdingbar, zumindest den Primer [13] zu lesen.
An dieser Stelle werden jetzt nicht die einfachen Datentypen und Integritätsbedingungen eingeführt, daß geschieht in Kapiteln 4 und 5. Wir benutzen an dieser Stelle auch ausschließlich die XML-Darstellung eines Schemas, generell gilt, daß das Wurzelelement ein `schema`-Element ist, welches unter anderem den zu populierenden Namensraum festlegt. Generell bezeichnen wir Elemente direkt unter dem `schema`-Element als Top-Level-Deklaration bzw. -Definition.
An vielen Stellen, z.B. den `element`-, `choice`- und `sequence`-Elementen ist die Angabe von Kardinalitäten mittels den `minOccurs`- und `maxOccurs`-Attributen möglich, mehr dazu in 5.6.
Nachfolgend wird nur ein kurzer Überblick gegeben, die angegebenen Beispiele sind auch bei weitem nicht ausschöpfend.

Elementdeklarationen

Elementdeklarationen haben die Form:

```
<element name="a" type="b"/>
<element ref="c"/>
<element name="d">
  ...
</element>
```

Im ersten Fall legen wir den Namen fest und weisen dem Element einen Typ zu. a kann ein Top-Level-Element sein oder auch im Inhaltsmodell eines anderen Elementes (bzw. dessen Typ) auftreten. Im zweiten Fall führen wir eine Referenz auf eine Top-Level-Elementdeklaration durch, was semantisch einem Einsetzen gleich kommt, im dritten Fall definieren wir den Typ des Elementes direkt innerhalb des Elementes.

Typdefinitionen

Typdefinitionen haben die Form:

```
<complexType name="b">
  ...
</complexType>
<simpleType name="f">
  ...
</simpleType>
```

Das Namensattribut entfällt u.U., komplexe Typen (zu `complexType`) sind ausschließlich für Elemente da, sie umfassen das Inhaltsmodell und die Attributdeklarationen. Einfache Typen (zu `simpleType`) sind Restriktionen an Zeichenketten und können in Attributen und für Elementdeklarationen ohne Attribute und Kindelemente verwandt werden.
Die Typdefinitionen enthalten oftmals `restriction`-, `union`, `list`- oder `extension`-Elemente, um die in XML-Schema mögliche Strukturvererbung zu realisieren.

[9] Es wird gefordert, daß im Schema beide Elementdeklarationen auf den selben Typ mit einem Namen verweisen.

Attributdeklarationen

Innerhalb der Typfestlegung für ein Element können mit dem `attribute`-Element Attribute festgelegt werden.

Inhaltsmodelle

Die `choice`-, `sequence`- und `all`-Elemente sowie die nicht-Top-Level `element`-Elemente dienen der Festlegung von Inhaltsmodellen analog wie bei XML und SGML. `choice`-Elemente entsprechen der Auswahl bei regulären Ausdrücken, `sequence`-Elemente entsprechen der Konkatenation, die Klammerung einzelner Bestandteile der regulären Ausdrücke findet sich in der Schachtelung dieser Elemente untereinander wieder. Auf tiefster Ebene stehen letztendlich `element`-Elemente.

Das `all`-Element ist eine vereinfachte Version des SGML &-Operators. Es darf nur auf höchster Ebene[10] innerhalb eines Inhaltsmodells für ein Element stehen und nur `element`-Elemente enthalten, die entweder genau einmal oder maximal einmal auftreten dürfen. Schon der &-Operator verändert aber die Mächtigkeit der möglichen Ausdrücke nicht, wir wir können das `all`-Element äquivalent ersetzen:

```
<all>
  <element name="a" type="a_typ"/>
  <element name="b" type="b_typ"/>
</all>
```

wird ersetzt durch:

```
<choice>
  <sequence>
    <element name="a" type="a_typ"/>
    <element name="b" type="b_typ"/>
  </sequence>
  <sequence>
    <element name="b" type="b_typ"/>
    <element name="a" type="a_typ"/>
  </sequence>
</choice>
```

Beachte, daß wenn n Elemente im `all`-Element enthalten sind, wir zu $n!$ `sequence`-Elementen im `choice`-Element übergehen.

Zusätzlich existiert das `group`-Element, mit welchem wir ein Inhaltsmodell auch außerhalb von Typdefinitionen festlegen können. Sie übernehmen damit auch die Funktion von Parameter Entities bei DTDs, wobei aber innerhalb einer Gruppe eine andere benutzt werden kann, zyklische Abhängigkeiten sind verboten, so daß es auf einen reinen Ersetzungsmechanismus hinausläuft, wir also nicht an Ausdrucksfähigkeit hinzugewinnen.

Spezialitäten von XML-Schema

Was wir bisher gesehen haben, geht nur in dem Punkt über die Mächtigkeit von DTDs hinaus, daß Elementnamen nicht notwendig nur genau ein Typ zugeordnet ist.

[10] siehe dazu auch Anhang C: Interpretation der XML-Schema Spezifikation

Weiterhin können wir die Strukturvererbung vielfältig mit `restriction`- und `extension`-Elementen steuern, wobei (wenn wir es nicht verboten haben) abgeleitete Typen in einer Instanz anstelle des ursprünglichen Types benutzt werden können, was aber über das spezielle `xsi:type`-Attribut angezeigt werden muß.

XML-Schema bietet die Möglichkeit der Ersetzungsgruppen (engl. substitution groups), wobei ein ausgezeichnetes (Top-Level) Element in der Instanz durch Mitglieder *seiner* Ersetzungsgruppe ersetzt werden kann.

Analog `ANY` bei den DTDs bietet XML-Schema die Konstrukte `any` und `anyAttribute` die auch noch mehr Möglichkeiten bieten.
Elementdeklarationen können zusätzlich das Attribut `nillable` besitzen, wobei dann entsprechende Instanzen durch das Setzen des Attributes `xsi:nil="true"` einen Nullwert signalisieren und in diesem Fall sogar keinen Inhalt haben müssen (und dürfen), wenn das Inhaltsmodell ϵ eigentlich nicht in seiner Sprache enthält.

Die Umformung in Regeln einer regulären Baumgrammatik entnehme man [23].

2.7 Nicht-1-Mehrdeutigkeit bei XML-Schema

In [31] Abschnitt 3.8.6 *Constraints on Model Group Schema Components*; Punkt *Schema Component Constraint: Unique Particle Attribution*, wird analog SGML und XML der Determinismus gefordert:

> A content model must be formed such that during validation of an element information item sequence, the particle contained directly, indirectly or implicitly[11] therein with which to attempt to validate each item in the sequence in turn can be uniquely determined without examining the content or attributes of that item, and without any information about the items in the remainder of the sequence.

Bemerkung:
Daraus folgt unter anderem, daß in einer `all`-Gruppe kein Element zweimal enthalten sein darf.

Bei SGML und XML entspricht diese Forderung der Nicht-1-Mehrdeutigkeit des regulären Inhaltsmodells. Die Inhaltsmodelle beziehen sich auf Elemente, die Verwendung eines abgeleiteten Elementtypes anstelle des Originaltyps stellt kein Problem dar, den die Kennzeichnung erfolgt über das `xsi:type`-Attribut.

Schwieriger (da nicht gekennzeichnet) wird die Betrachtung bei Ersetzungsgruppen. Die Top-Level-Elementdeklarationen bilden bezüglich der Mitgliedschaft in Ersetzungsgruppen einen gerichteten Wurzelwald (also eine Menge gerichteter Wurzelbäume). Es ist erlaubt, daß in einem Inhaltsmodell Elemente derselben Ersetzungsgruppe verwandt werden.

Seien A und B Elementdeklarationen, wobei B in der Ersetzungsgruppe von A sei. Das Inhaltsmodell möge die Form haben: $A?B^*$, welches ohne Berücksichtigung von Ersetzungsgruppen nicht-1-mehrdeutig ist. Ist das aktuelle Element aber vom Typ B, dann wäre es auch möglich, dieses für $A?$ zu validieren. Die Nicht-1-Mehrdeutigkeit des Inhaltsmodelles reicht hier also nicht aus, ersetzen wir in $A?B^*$ das Auftreten von A durch jedes Mitglied in der Ersetzungsgruppe von A, dann erhalten wir zusätzlich das Inhaltsmodell $B?B^*$, welches nicht nicht-1-mehrdeutig ist.

Wir sehen also, es ist hinreichend zu fordern, jedes Element der Menge $subGrp(CM)$ ist nicht-1-mehrdeutig:

[11] das bezieht sich auf Ersetzungsgruppen, vgl. [31]

$subGrp(CM)$ enthält alle Inhaltsmodelle, die strukturell identisch zu CM aufgebaut sind und für jedes (einzelne) Auftreten einer Top-Level-Elementdeklaration A in CM kann eine beliebige Deklaration B der Ersetzungsgruppe eingesetzt werden, an jeder betreffenden Stelle unabhängig voneinander.

Beispiel:
$CM = ABCD$, wobei
$substitutionGroup(A) = \{A, B, C\}$,
$substitutionGroup(B) = \{B, C\}$,
$substitutionGroup(C) = \{C\}$ und
$substitutionGroup(D) = \emptyset$ [12] sei.

Es gilt dann:
$subGrp(CM) = \{ABCD, ACCD, BBCD, BCCD, CBCD, CCCD\}$
Jedes dieser Inhaltsmodelle muß nicht-1-mehrdeutig sein.

Somit haben wir hier auch die fehlende formale Definition des „Determinismus" gem. der XML-Schema-Spezifikation nachgeliefert, welche selbst diese in „H Analysis of the Unique Particle Attribution Constraint (non-normative)" nicht formal ausführt. Der einzige Punkt, der noch fehlt, sind die `any`-Elemente, diese sind in $subGrp(CM)$ so aufzunehmen, daß für jedes A', welches entsprechend `any` validiert würde, dieses mit einzusetzen ist. Dadurch kann $subGrp(CM)$ unter Umständen unendlich werden, der Test kann aber so aufgebaut werden, daß wenn wir $First$- und $Follow$-Menge von CM testen, immer separat geprüft wird, ob die zu `any` gehörige Menge Probleme bereitet. Der Namensraumpräfix wird an dieser Stelle immer als integraler Namensbestandteil betrachtet, also auch als vorhanden angenommen, wenn das jeweilige Element in einem Namensraum ist.

[12] da D keine Top-Level-Elementdeklaration sei.

Kapitel 3

Ansätze für das Inhaltsmodell

In diesem Kapitel geht es um die Ableitung des Inhaltsmodelles, wofür verschiedene existierende Ansätze untersucht werden. Zunächst werden noch einige Einschränkungen an die XML-Dokumente getroffen, die von ihrer Natur her die Komplexität des Problems aber nicht ändern.

O.B.d.A. enthalte das XML-Dokument

- keine interne DTD, vielmehr ist alles in einer externen DTD definiert,
- keinerlei Processing Instructions und Kommentare,
- an seinem Anfang ausschließlich den Prolog `<?xml version="1.0" ?>`, d.h. Probleme der Zeichenkodierung (etwa UTF-8, UTF-16) werden hier nicht diskutiert,
- keine `DOCTYPE`-Deklaration, die Zuordnung eines Dokumentes zu einer DTD wird als bekannt vorausgesetzt,
- keine noch zu bearbeitenden Entity-Referenzen

Generell gilt, daß innerhalb einer DTD oder auch eines XML-Schemas das Wurzelelement nicht festgelegt wird, d.h. bei einer DTD können alle definierten Elemente und bei XML-Schema alle Top-Level-Elemente als Wurzelelement auftreten.

Wenn es um die Erzeugung einer DTD geht, so haben wir bei Elementen, die auch `#PCDATA`-Inhalt besitzen können, nur zwei Möglichkeiten bei der Konstruktion des Inhaltsmodells (siehe auch Anhang D), konkret kann der entsprechende DTD-Teil entweder nur die Form `<!ELEMENT e (#PCDATA)>` oder `<!ELEMENT e (#PCDATA |e1|...|en)*>` annehmen, wobei im Falle des Auftretens von Kindelementen immer die zweite Form zu wählen ist, wobei dann mindestens die Elementnamen der auftretenden Kindelemente aufzuführen sind.

Dieser Fall ist verhältnismäßig einfach, da wir nur alle Inhalte auf vorhandene Kindelemente untersuchen müssen, aus diesem Grund deshalb wollen wir ihn aus der folgenden Diskussion ausklammern, d.h. die betrachteten Elemente weisen keinen gemischten Inhalt (engl. mixed content type) auf.

Bei XML-Schema ist die Situation etwas anders, gemischte Inhaltstypen haben gleiche Inhaltsmodelle wie normale Inhaltstypen und sind nur besonders gekennzeichnet. Somit müssen wir für Elemente, die gemischten Inhalt haben, im entsprechenden Typ die Kennzeichnung `mixed="true"` aufnehmen und ansonsten können wir die Textinhalte vernachlässigen.

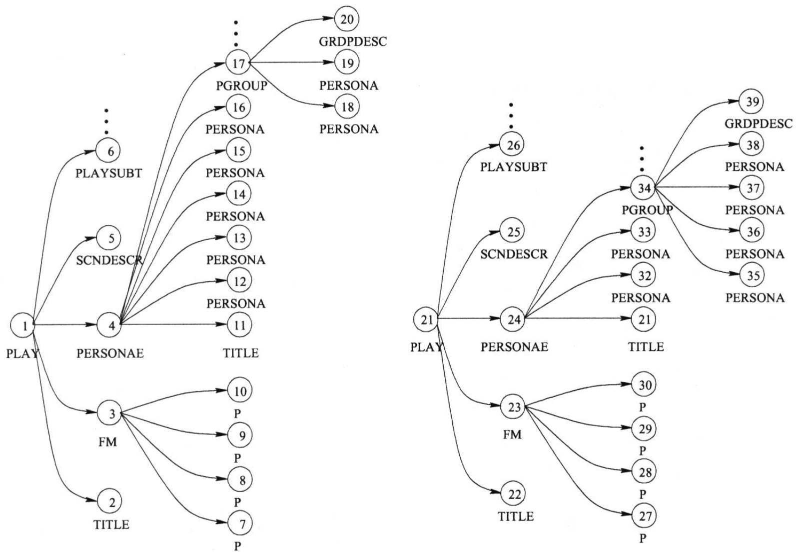


Abbildung 3.1: Beispielbäume für XML-Dokumente (Abb. aus [25])

Hat ein Elementtyp allerdings niemals Kindelemente, so müssen wir bei XML-Schema einen einfachen Typ (engl. simple type) annehmen und diesen entsprechend ableiten, was in Kapitel 4 diskutiert wird.

3.1 Einfache Ansätze

Nachfolgend werden zunächst zwei einfache Ansätze vorgestellt, einfach in dem Sinne, daß die theoretische Grundlage dieser Ansätze bei weitem nicht die Komplexität der theoretischen Grundlagen der später vorgestellten Ansätze besitzt.

3.1.1 DTD-Miner

Moh, Lim und Ng stellen mit ihrem Tool DTD-Miner die Möglichkeit zu Verfügung, WWW-basiert sich zu seinen XML-Dokumenten eine DTD generieren zu lassen. An dieser Stelle soll ausschließlich der theoretische Hintergrund für die DTD-Generierung betrachtet werden, die Benutzerschnittstelle und konkrete Implementierungsdetails werden an dieser Stelle vernachlässigt.

Als Ausgangspunkt dienen bei DTD-Miner Dokumentbäume, die in [25] nicht identisch zu unseren XMLtrees charakterisiert werden, aber von ihrer Mächtigkeit nicht mehr bieten, weshalb wir an dieser Stelle o.B.d.A. die Darstellung als XMLtrees benutzen.

Die gegebene Menge an XML-Dokumenten sei $I = \{d_1, d_2, ..., d_n\}$, wobei die Knotenmenge zu jedem XMLtree o.B.d.A. $V(d_i) = \{v_{i_1}, v_{i_2}, ..., v_{i_m}\}$ sei.

Beispiel :
Aus [25] stammen die Abbildungen 3.1 und 3.2, dabei zeigt Abbildung 3.1 zwei XML-Bäume und Abbildung 3.2 den zugehörigen Gesamtgraphen.

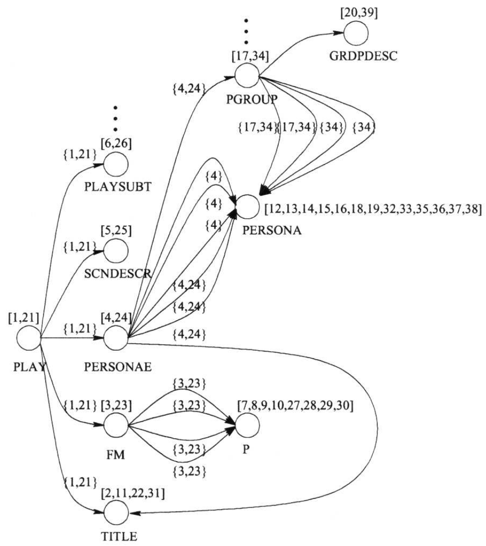


Abbildung 3.2: Gesamtgraph für XML-Dokumente (Abb. aus [25])

Ausgehend von I wird ein neuer geordneter, gerichteter, azyklischer Graph $G = (V, E)$ generiert, der aber nach unserer Definition kein XMLtree ist, da der zugehörige ungerichtete Graph nicht azyklisch ist und doppelte Kanten erlaubt sind. In [25] wird dieser Graph der „spannende Graph“ (engl. spanning tree), genannt, damit das nicht zu Missverständnissen führt, wird er im folgenden als **Gesamtgraph** bezeichnet.

Eine Kante $e \in E$ kann im Gegensatz zu den vorhergehenden Betrachtungen nicht als Paar von Knoten charakterisiert werden, da doppelte Kanten erlaubt sind. Vielmehr ist E eine abstrakte endliche Menge, deren Elementen über die Funktionen **from**: $E \rightarrow V$ und **to**: $E \rightarrow V$ entsprechende Knoten aus V zugeordnet werden, $from(e) = to(e)$ ist ebenfalls zulässig. Die Ordnung bezieht sich beim Gesamtgraphen auch nicht nur auf V sondern – vermutlich[1] – auch auf E. Weiterhin sei zu jedem Knoten v des Gesamtgraphen ein Attribut $pcdata(v)$ definiert, das den Wert $wahr$ oder $falsch$ annehmen kann.

Für diesen Gesamtgraphen wird analog den XMLtrees eine Funktion $name()$ definiert, weiterhin eine Funktion **quelle**: $E \cup V \rightarrow \mathcal{P}_{fin}{}^{2}(\bigcup_{i=1...n} V(d_i))$.
Die Funktion $quelle()$ kennzeichnet die Menge der Knoten der Ausgangsdokumente, die zu einem Knoten im Gesamtgraphen „zusammengefaßt“ wurden ($quelle(v)$; in Abb. 3.2 mit [...]) bzw. welche als Vaterknoten für andere Knoten fungierten ($quelle(e)$; in Abb. 3.2 mit {...}).

Beispiel aus [25]:
Abb. 3.2 zeigt einen Ausschnitt des Gesamtgraphen zu den XML-Dokumenten aus Abb. 3.1

Die konkrete Definition des Gesamtgraphen G und der Funktionen $name()$ und $quelle()$ erfolgt konstruktiv, wobei die Idee ist, einen neuen Knoten nur dann einzufügen, wenn es noch keinen Knoten mit dem aktuellen Elementnamen gibt und neue Kanten nur dann einzufügen, wenn allen anderen Kanten mit gleichem Start- und Zielknoten mittels $quelle()$ bereits der aktuelle Elementname zugeordnet wurde, ansonsten wird die Menge $quelle(e)$ für die Kante e erweitert, welcher der Elementname noch nicht zugeordnet wurde und im Falle mehrerer Möglichkeiten wird die zuerst eingefügte gewählt.

[1] d.h. bei Betrachtung des Beipieles in [25]
[2] $\mathcal{P}_{fin}$ ist die Menge aller *endlichen* Teilmengen.

Bei den Attributen wurde nur die einfache Vorgehensweise realisiert, einfach die Menge aller Attributnamen zu einem Elementtyp einzusammeln, eine Typerkennung findet nicht statt, die Autoren haben sich diesen Punkt für spätere Arbeiten aufgehoben.

Aus der Konstruktion folgt, daß die Funktion $name()$ für alle Knoten im Gesamtgraphen paarweise verschiedene Werte liefert.

Aus dem Gesamtgraphen wird nun die DTD generiert.

Leider sind die Autoren von [25] bei der Beschreibung nicht besonders exakt und vollständig vorgegangen, nicht einmal die Beispielabbildungen sind vollständig korrekt. Die Definition der Ordnung bezogen auf die Knoten erfolgt nicht, es wird zwar gesagt:

> The left-right ordering of sibling edges denotes the left-right ordering of subelements of a parent document element. For example, if the node coordinates ist the left sibling of the node area in the Spanning Graph, then in every XML document spanned by the Spanning Graph and having the two tag pairs as sibling nodes, the tag pairs <coordinates></coordinates> must appear before <area></area>.

Diese Aussage ist meistens falsch, als Gegenbeispiel wollen wir das Inhaltsmodell `((coordinates,area)|(area,coordinates))` und jeweils ein Beispieldokument für jede der beiden Möglichkeiten der Auswahl betrachten. Die obige Skizze des Algorithmus ergab sich aus den von den Autoren gemachten Aussagen und Beipielen, wobei die eben zitierte Aussage vernachlässigt wurde. Da ein Tool existiert, muß in der Implementation eine Vorgehensweise für Probleme, die aufgrund obiger Ausführungen nicht geklärt wurden, festgelegt worden sein – diese ist aber nicht bekannt.

Anzunehmen ist, daß das Attribut $pcdata(v)$ zu jedem Knoten v genau dann *wahr* ist, wenn es in irgendeinem Dokument irgendeinen Knoten w mit $name(w) = name(v)$ gibt, der unter seinen Kindknoten einen Knoten w' mit $name(w') =$ „$\#PCDATA$“ gibt.

Zur DTD-Generierung bemerken die Autoren nur das Anwenden von drei Heuristiken:

Define Optionality bestimmt, ob ein Element optional ist oder nicht (Anwendung von ?).

Merge Repeat identifiziert benachbarte Kindelemente mit gleichem Namen (hier natürlich auch gleichem Typ) für das Anwenden von * oder + auf ein einzelnes Element.

Define Group sucht nach Anwendungen für * oder + auf Gruppen von Elementen. Bis auf die Erwähnung, daß $DefineGroup$ nur dann zwei Gruppen von Elementen zusammenfassen kann, wenn sie identische Mitglieder haben, ist absolut nichts über die Funktionsweise der Heuristiken ausgeführt.

Der Nutzer kann die erhaltene DTD weiter generalisieren lassen, indem er einen Parameter vorgibt, der angibt, wie oft ein Element maximal in einem Inhaltsmodell auftreten darf, der DTD-Miner generalisiert dann – durch nicht näher charakterisierte Zusammenfassungen – weiter, bis die DTD dieses Restriktion erfüllt.

Diskussion:

Die Autoren führen die Arbeit von Ahonen [5] unter „Verwandte Arbeiten“ an, ohne sich jedoch dem Vergleich zu stellen, nur mit der Bemerkung, daß Ahonen nicht darauf eingeht, wie der Nutzer mit einem quantitativen Maß eine generierte DTD vereinfachen kann, wobei die Autoren es vernachlässigt haben, einfach einmal der Selbstreferenz von Ahonen auf [2] nachzugehen, wobei gerade die oben beschriebene Kontextualitätseigenschaft parametrisiert ist und somit auf eine gewisse Art und Weise den Grad der Generalisierung steuert.

Die Steuerung über einen einzigen Wiederholungsparameter möchte ich an dieser Stelle als zu simpel charakterisieren, da er erstens außer acht läßt, daß der Nutzer nicht alle Produktionen weiter generalisieren möchte, und zweitens, die Klasse der so darstellbaren Inhaltsmodelle ziemlich willkürlich gewählt ist.

Über die konkrete Vorgehensweise zur Generierung einer DTD ausgehend vom Gesamtgraphen kann ich an dieser Stelle nur spekulieren. Aus dem Gesamtgraphen ergibt sich natürlich, welche Elemente überhaupt im Inhaltsmodell eines Elementtypes auftreten können und auch die Gesamthäufigkeit bezogen auf die Beispiele. Die Reihenfolge der Kindelemente kann nicht mehr rekonstruiert werden, da die Ordnung des (geordneten) Gesamtgraphen nicht definiert wurde, wie die Implementierung vorgeht ist mir nicht bekannt.

An dieser Stelle wird schon klar, daß mit diesem Ansatz nicht viel zu erreichen ist, denn der Gesamtgraph ist für oben genanntes Problem nicht definiert und folgendes Beispiel zeigt die Probleme der Arbeit:

Eingabe (in dieser Reihenfolge):

eins.xml:

```
<?xml version="1.0" ?>
<content>
  <a>inhalt</a>
  <b>inhalt</b>
</content>
```

zwei.xml:

```
<?xml version="1.0" ?>
<content>
  <b>inhalt</b>
  <a>inhalt</a>
</content>
```

Ergebnis:

```
<!-- DTD-Miner Version 1.5 ( 6 Dec 99 ) -->
<!-- CAIS, NTU, Singapore -->
<!-- Copyright 1999 Moh Chuang Hue -->
<!-- Parameters specified:  -->
<!DOCTYPE content [
    <!ELEMENT content ( b?, a, b? ) >
    <!ELEMENT b ( #PCDATA ) >
    <!ELEMENT a ( #PCDATA ) >
]>
```

Wenn die Reihenfolge der Eingabe vertauscht wird, erhalten wir folgendes Ergebnis:

```
<!-- DTD-Miner Version 1.5 ( 6 Dec 99 ) -->
<!-- CAIS, NTU, Singapore -->
<!-- Copyright 1999 Moh Chuang Hue -->
<!-- Parameters specified:  -->
<!DOCTYPE content [
    <!ELEMENT content ( a?, b, a? ) >
    <!ELEMENT a ( #PCDATA ) >
    <!ELEMENT b ( #PCDATA ) >
]>
```

Der DTD-Miner arbeitet also nicht reihenfolgeunabhängig, was bemerkenswert ist hinsichtlich der Problemstellung, zu einer gegebenen *Menge* eine DTD zu generieren – was selbstverständlich nicht reihenfolgeabhängig sein darf. Interessanterweise setzen sich die Autoren mit den eigentlichen Problemen – u.a. das Finden des Inhaltsmodelles und der Attributtypzuordnung – nicht auseinander, bis auf den einen Artikel von Ahonen [5] scheinen sie sich relevante Arbeiten die es bis zum Publizierungszeitpunkt des Artikels auch schon gab, nicht angesehen zu haben.

Für den folgenden Fall wird sogar eine ungültige DTD erzeugt:

drei.xml:

```
<?xml version="1.0" ?>
<content>
  <a>inhalt</a>
  <a>inhalt</a>
  <b>inhalt</b>
  <a>inhalt</a>
  <a>inhalt</a>
</content>
```

vier.xml:

```
<?xml version="1.0" ?>
<content>
  <a>inhalt</a>
  <a>inhalt</a>
  <a>inhalt</a>
</content>
```

Ergebnis:

```
<!-- DTD-Miner Version 1.5 ( 6 Dec 99 ) -->
<!-- CAIS, NTU, Singapore -->
<!-- Copyright 1999 Moh Chuang Hue -->
<!-- Parameters specified:  -->
<!DOCTYPE content [
    <!ELEMENT content ( a+, b?, a+ ) >
    <!ELEMENT a ( #PCDATA ) >
    <!ELEMENT b ( #PCDATA ) >
]>
```

Man beachte hierbei nämlich, daß das Inhaltsmodell für das Element `content` nicht deterministisch im Sinne der XML-Spezifikation, d.h. nicht-1-mehrdeutig ist.

Der Ansatz vom DTD-Miner sieht insgesamt nicht sehr vielversprechend aus, man muß aber positiv hervorheben, daß sehr viel Arbeit in die Erstellung der Oberfläche und das Verfügbarmachen im WWW gesteckt wurde.

3.1.2 GB-Engine

Einer der ersten Ansätze für die Erstellung von DTD kam 1995 von Schafer et al. und ist in [30] beschrieben. Im Rahmen des *SGML Document Grammar Builder Projektes* wurde dort die GB-Engine und zugehörige Tools bzw. Oberflächen entworfen.

Die GB-Engine arbeitet auf einer internen Baumdarstellung der XML-Dokumente, auftretende Attribute werden gesammelt, inwiefern eine Typerkennung stattfindet, ist in [30] nicht beschrieben.

Intern arbeitet die GB-Engine mit Strukturregeln der Form `Element -> Definition`, z.B. `article -> title author+ body`. Diese Regeln werden dann manipuliert und am Ende in korrekter DTD-Syntax ausgegeben.

Im ersten Schritt werden zu allen XML-Dokumenten und allen enthaltenen Elementen Strukturregeln der genannten Art erzeugt, die Operatoren ?, +, * treten dabei zunächst nicht auf. Wie bereits erwähnt können wir das Problem der Ableitung einer DTD reduzieren auf die Einzelprobleme der Ableitung der Inhaltsmodelle für einzelne Elemente. Auch die GB-Engine verfährt so, indem alle gesammelten Strukturregeln mit gleicher linker Seite zu einer einzigen Regel wie folgt kombiniert werden:

`A -> B ; A->C; A->DE` werden kombiniert zu `A -> B|C|DE`

Danach werden diese Regeln mit geeigneten Heuristken generalisiert und reduziert, d.h. insbesondere Operatoren wie +, *, ? werden eingeführt und auch einige Regelteile weggelassen, die durch andere subsummiert werden.

Dem Nutzer der GB-Engine stehen folgende Reduktionen zur Verfügung, welche er entweder durchführen lassen kann oder nicht, z.Zt. wohl aber nur in der genannten Reihenfolge. Die Beispiele stammen direkt aus [30]:

- **Single and Empty**

  Leere Teilregeln und unnötige Klammern werden entfernt.

```
A () (B)                           -> A B
((A?)+) () ((B|(C))) (B)?          -> A* (B|C) B?
```

- **Collapse ANDs**

  Faßt Teilregeln bzgl. der Sequenz zusammen, wenn diese Teilregeln die Kardinalität „genau einmal“ – also nicht ?, +, * – besitzen.

```
(A B) C                            -> A B C
A B (C D) F (G H)*                 -> A B C D F (G H)*
```

  Es ist an dieser Stelle leider nicht klar, wie die Regeln vor dieser Transformation überhaupt entstehen können und warum diese redundanten Klammern nicht schon von *Single and Empty* entfernt wurden.

- **Collapse ORs**

  Faßt Teilregeln bzgl. | zusammen, wenn die entstehende Regel die alte Regel subsummiert, konkret heißt das Ausnutzung der Asoziativität von | und die Vermeidung von Redundanzen.

```
(A|B)|C                            -> A|B|C
(A|(B|C)|(D|E)+)*                  -> (A|B|C|D|E)*
```

- **Repeating Atoms**

  Wenn zwei oder mehr identische Elemente nebeneinander in einer Regel auftreten, wird der Operator + eingesetzt. Den Beispielen zufolge scheinen hier auch redundante Teilregeln der Art `(B|C|B+)` zu der (äquivalenten) Teilregel `(C|B+)` reduziert zu werden.

```
AAA                                -> A+
A A? (B|C|B+)+ (B|C|B+) D          -> A+ (B+|C)+ D
```

- **PCDATA to ORs**

  In DTDs erzwingt das Auftreten von Zeicheninhalt ganz bestimmte Regelkonstruktionen der Art `(#PCDATA| element1 |...|elementn)*` bzw. `(#PCDATA)`.

```
A #PCDATA                          -> (A|#PCDATA)+
A B | #PCDATA B                    -> (A|B|#PCDATA)+
```

  Beachte, daß in den genannten Beipielen die Ergebnisse der Reduktion noch nicht exakt die benötigte Form für DTDs haben, sicher aber leicht umformen lassen.

- **Identical Bases**

  Teilregeln, welche gleiche Elemente unabhängig von Qualifikatoren wie *,+,? haben, werden zusammengefaßt, wobei die neue Teilregeln – zumindest nach den Beipielen – die alten umfaßt, allerdings nach Anmerkung von Schafer hier auch leicht die Gefahr der Übergeneralisierung entsteht.

```
(A B)* | A? B?                     -> (A? B?)*
(A B)* | (A? B?) | A B             -> (A? B?)*
(A B? | A* B)                      -> A* B?
(A B)* | (A? B?)                   -> (A? B?)*
```

- **Off by One**

  Aus den Beispielen folgen sofort Indizien für „ein Element tritt genau einmal auf“ und „ein Element tritt mindestens einmal auf“. Schwieriger wird das Finden von Auftreten von ? und * für die Regeln, hierfür werden bei dieser Heuristik Teilregeln entsprechend zusammengefaßt, wenn sie sich nur an einer Stelle unterscheiden.

  ```
  A+ B | B                        -> A* B
  A B C | A C | A B               -> A B? C | A B C?
  ```

  Im zweiten Beispiel wird absichtlich die Semantik gewahrt, daß `A` nicht alleine auftritt und deshalb wird nicht zu `A B? C?` reduziert.

- **Redundant**

  Hier werden Teilregeln entfernt, welche durch andere Teilregeln subsummiert werden, allerdings werden aus Effizienzgründen Vergleichsverfahren eingesetzt, die nicht notwendig alle Subsummationen finden, beachte hier, daß im allgemeinen auch zwei Teilregeln eine dritte Subsummieren können und das Finden der „besten“ Subsummationen aufwendig ist.

  ```
  A B | A B*                      -> A B*
  A B | A B* | A B | C | C        -> A B* | C
  ```

Genaueres über die Funktionsweise der genannten Reduktionen ist in [30] nicht beschrieben, so daß man teilweise für die Semantik auf die Beispiele angewiesen ist.

Beachte aber, daß der Nutzer dafür zuständig ist, ggf. einige Reduktionsregeln auszuschließen und alle Regeln dazu führen, die Schachtelungstiefe von Operatoren zu verringern oder zumindest nicht zu erhöhen. Daraus folgt automatisch, daß es nicht möglich ist, Inhaltsmodelle korrekt abzuleiten, die eine bestimmte Schachtelungstiefe besitzen und es keine äquivalenten Inhaltsmodell mit geringerer Schachtelungstiefe gibt.

Es ist also zu erwarten, daß das Verfahren von Schafer nur für gewisse einfachere XML-Dokumente eine „gute“ DTD liefert, es ist aber auch explizit gesagt, daß das Ziel der GB-Engine keine perfekte DTD ist, sondern vielmehr eine Grundlage für die (manuelle) Weiterarbeit darstellen soll.

Shafer ist sich des Problems der Nicht-1-Mehrdeutigkeit bewußt, auch wenn dieses nicht ganz richtig dargestellt wird, da ja auch nicht formal eingeführt wird, was eine Teilregel überhaupt ist. Für die Fälle, daß man Nicht-1-Mehrdeutigkeit erreichen will, wird auf Arbeiten von Exoterica (1991) verwiesen, welche leider nicht beziehbar waren. Ob aber sein Ansatz überhaupt eindeutige Grammatiken produziert, wird nicht näher diskutiert, aber wir können ja zu jedem mehrdeutigen regulären Ausdruck einen eindeutigen finden.

3.2 Theoretisch fundierte Ansätze

3.2.1 Grundlagen

Nachfolgend werden einige Verfahren und Ansätze vorgestellt, die eine komplexere theoretische Grundlage haben, als die bisher vorgestellten Ansätze, welche eine Anzahl von Regeln bemühen, die sich aus verschiedenen Heuristiken ergeben. Das soll aber nicht bedeuten, daß die folgenden Verfahren ohne Heuristiken auskommen, der Abstraktionsgrad ist nur um einiges höher.

Auch hier gilt natürlich, daß das Problem die Ableitung eines Inhaltsmodells für *einen* Elementtyp ist, die Inhaltsmodelle verschiedener Elementtyp können wir ja offensichtlich getrennt handhaben. Wie immer sei I die Menge unser Beispiel-XML-Dokumente.

Die theoretischen Grundlagen der nachfolgenden Verfahren stammen aus der Lerntheorie, zunächst wird die Grundlage der Verfahren von Muggleton, Ahonen/Manila und Fernau vorgestellt, die theoretischen Aspekte zu dem XTRACT-System werden dann unmittelbar vor der Beschreibung selbigens dargestellt.

Bei den erstgenannten dreien geht es um die Erlernung einer regulären Sprache ausgehend von I. Hierbei wird angenommen, daß es genau eine Sprache L mit $I \subseteq L$ gibt, welche wir suchen. In etwa kann man sich das gedanklich so vorstellen, daß wir L kennen und dem Algorithmus Beispielworte aus L präsentieren und erwarten, daß die Sprache, die der Algorithmus liefert, gegen L konvergiert.

Das Grundproblem bei der Grammatik-Ableitung ist es also, ausgehend von einer Familie von Sprachen C genau eine Sprache $L \in C$ auszuwählen, welche von der Beispielmenge E charakterisiert wird. Für C sei ein Hypothesenraum vorhanden, mit dem die jeweiligen Sprachen beschrieben werden, z.B. kann der Hypothesenraum die Menge an endlichen Automaten oder an regulären Ausdrücken sein, wenn C die Klasse der regulären Sprachen ist.

E enthält positive und negative Beispiele, d.h. Paare von Worten über dem entsprechenden Alphabet und einer wahrheitswertigen Variable, in unserem Fall ist $E := I$ und E enthält ausschließlich positive Beispiele. Wichtig hierfür ist eine grundlegende Definition von Gold (1967) hier wiedergegeben aus [26]:

Definition:
Ein Ableitungs-Algorithmus Alg **identifiziert eine Sprache in den Grenzen**, genau dann, wenn nach einer gewissen Anzahl an Beispielen, die Alg übergeben wurden, Alg sich für das korrekte $L \in C$ entscheidet und seine Wahl nicht mehr ändert, wenn weitere Beispiele übergeben werden. Eine Sprachfamilie C heißt dann dementsprechend **identifizierbar in den Grenzen**, wenn es einen Algorithmus für C mit der genannten Eigenschaft gibt.

Gold zeigt weiterhin, daß eine Sprachfamilie, die alle endlichen und mindestens eine unendliche Sprache enthalten nicht identifizierbar in den Grenzen ist, wenn E ausschließlich positive Beispiel enthält.

Bei uns ist C zunächst die Menge der regulären Sprachen, die Diskussion der Nicht-1-Mehrdeutigkeit wird separat geführt. Beachte, daß alle endlichen Sprachen regulär sind und es auch unendliche reguläre Sprachen gibt, da $E = I$ gilt und I ausschließlich positive Beipiele enthält, folgt aus Golds Aussage unmittelbar, daß es in unserem Fall keinen Algorithmus gibt, der uns unsere gesuchte Sprache in den Grenzen identifiziert.

Es gibt jedoch viele Teilfamilien der Familie der regulären Sprachen, die identifizierbar in den Grenzen sind. Das heißt, daß es einen Algorithmus gibt, der für $w_1, w_2, w_3, ...$ als eine beliebige Darstellung von L, Ausgaben $A_1, A_2, A_3, ...$ liefert, die gegen A_L konvergieren. Konkret könnten A_i deterministische endliche Automaten und A_L der Minimalautomat für L sein.

Die Definition von Gold geht von einer Aufzählung der gesamten Sprache aus, für unseren konkreten Anwendungsfall haben wir aber nur eine endliche Teilmenge der gesuchten Sprache. Angluin zeigt in [6] (Theorem 1), daß eine indizierte Klasse von nicht-leeren, rekursiven Sprachen[3] genau dann identifizierbar in den Grenzen ist, wenn es eine berechenbare Funktion gibt, die für i die *endliche* Menge T_i liefert, für die gilt: $T_i \subseteq L_i$ und $\forall j : T_i \subseteq L_j \Rightarrow L_j \not\subset L_i$.

D.h. es reicht für einen Algorithmus aus zu zeigen, daß er (bei festgelegter identifizierbarer Sprachklasse) für jede übergebene Beispielmenge – welche nicht gleich einem T_i sein muß, aber eines als Teilmenge enthält – ein Ergebnis aus dem Hypothesenraum liefert, welches die

[3]d.h. es ist entscheidbar, ob $w \in \Sigma^*$ in L_i enthalten ist

genannten Bedingungen erfüllt. In diesem Fall identifiziert *dieser* Algorithmus diese Sprachklasse.

Angluin führt die Klasse der k-umkehrbaren Sprachen (engl. k-reversible languages) ein und zeigt, daß diese Klasse identifizierbar in den Grenzen ist.

Definition (Angluin, hier wiedergegeben aus [24]):
Eine Sprache L heißt k-umkehrbar ($k \geq 0$) :gdw.
$u_1vw, u_2vw \in L$ mit $|v| = k \Rightarrow T_L(u_1v) = T_L(u_2v)$.

Hierbei ist $T_L(w) := \{v \mid wv \in L\}$ der Linksquotient von L.

Bekanntermaßen ist ein endlicher Automat M deterministisch genau dann, wenn es keinen Zustand $z \in Z$ und kein Zeichen $a \in \Sigma$ gibt, und $(z, a, z'), (z, a, z'') \in \delta$ mit $z' \neq z''$ existieren.

Anschaulich heißt das, wenn wir in einem Zustand sind und das Zeichen a abarbeiten wollen, nur maximal ein neuer Zustand nach der Transition möglich ist, es kann allerdings auch sein, daß überhaupt keine a-Transition für den aktuellen Zustand definiert ist.

Wir können den Determinismusbegriff verallgemeinern:

M ist **deterministisch mit der Vorausschau k** :gdw. es gibt kein $w \in \Sigma^*$ mit $|w| = k$ und es gibt keine Zustände $z_1, z_2 \in Z$ mit $z_1 \neq z_2$ und (z_1, z_2 sind beide Startzustände oder $(z_3, a, z_1), (z_3, a, z_2) \in \delta$ für ein $z_3 \in Z$) und $\delta(z_1, w)$ sowie $\delta(z_2, w)$ sind beide definiert und verschieden. D.h. nach der Abarbeitung von w (beginnend in allen Startzuständen) bzw. aw (beginnend in einem beliebigen Zustand) gibt es höchstens einen Zustand, in den wir gelangen können. Setzen wir $k := 0$ so folgt daraus, daß es nur einen Startzustand gibt und der Automat deterministisch ist, da $|w| = 0$ impliziert, daß $w = \epsilon$ gilt.

Ist M ein endlicher Automat, so ist M^r der Umkehrautomat zu M, wenn – informal ausgedrückt – M^r sich aus M so ergibt, daß die Menge der Start- und Endzustände vertauscht werden und alle Transitionen umgekehrt werden.

Satz 4 *(Angluin, hier wiedergegeben aus [24]):*
Ein endlicher Automat M ist k-umkehrbar gdw.
M ist deterministisch und M^r ist deterministisch mit Vorausschau k.

Für M bedeutet das, daß M zunächst natürlich deterministisch ist und es keine Zustände z_1', z_2' gibt, so daß es ein Wort w mit $|w| = k$ und $z_1 = \delta(z_1', w)$ und $z_2 = \delta(z_2', w)$ gibt mit $z_1 \neq z_2$ und z_1, z_2 sind beide Endzustände oder es gibt für ein Symbol $a \in \Sigma$ von z_1 und z_2 jeweils eine a-Transition in denselben Zustand z_3 hinein.

3.2.2 Standardalgorithmus

Für k-umkehrbare und k-kontextuelle Sprachen gibt Muggleton in [26] den Standardalgorithmus für die Erkennung der Sprachen an:

Ausgehend von I wird ein Präfix-Baum-Automat (engl. prefixtree-automaton) konstruiert, der Name impliziert schon, daß die zugrunde liegende Graphenstruktur ein Baum ist und daß bei der Erkennung eines Wortes $w = uv$, der Zustand des Automaten nach der Abarbeitung des Präfixes u von w eindeutig bestimmt ist, w muß dabei nicht notwendig akzeptiert werden.

O.B.d.A. seien alle Zustände des Automaten mit irgendeinem Eingabewort erreichbar, was natürlich nicht heißt, daß dieses Eingabewort den Automat in einen akzeptierenden Zustand überführt bzw. aus diesem Zustand hinaus überhaupt ein Endzustand erreichbar ist.

Hieraus kann man natürlich sofort ersehen, daß M ein deterministischer endlicher Automat ist, denn gäbe es o.B.d.A. ausgehend von einem Zustand q_i einen Zustandsübergang auf der Eingabe

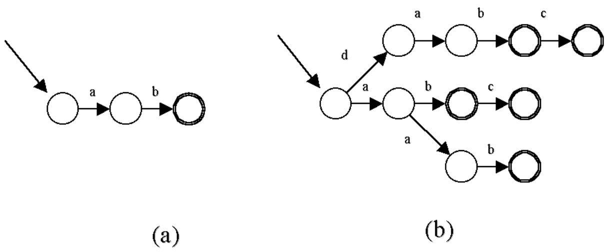


Abbildung 3.3: Präfixbaumautomat

a sowohl nach q_k als auch nach q_l mit $l \neq k$, und ist q_i mit dem Präfix v erreichbar, dann verletzt das Wort (bzw. der Präfix) va die obige Präfixbaumeigenschaft.:

Der endliche Präfixbaum-Automat $M = (Z, \Sigma, \delta, z_0, E)$ für den Elementtyp e wird wie folgt iterativ definiert, wobei I ein *characteristic sample* für eine Sprache aus der jeweiligen Sprachklasse sei:

$Z := \{z_0\}$	d.h. Z enthält zu Beginn nur den Startzustand
$\delta := \emptyset$	d.h. zu Beginn noch keine Zustandsübergänge definiert

Ist $\epsilon \in I$ dann $E := \{z_0\}$, sonst $E := \emptyset$
Für jedes $w \in I$ mit $w \neq \epsilon$ führe durch:

Wiederhole
  Sei $w = uav$ mit $u, v \in \Sigma^*$ und $a \in \Sigma$
  Ist $Z = \{z_0, ..., z_i\}$ und ist $\delta(z_0, u)$ definiert und $\delta(z_0, ua)$ nicht definiert, dann:
    $Z := Z \cup \{z_{i+1}\}$, wobei $z_{i+1} \notin Z$ war.
    $\delta(\delta(z_0, u), a) := z_{i+1}$
Bis $\delta(z_0, w)$ definiert ist
$E := E \cup \{\delta(z_0, w)\}$

Beispiel:
Sei $I = \{ab, abc, dab, dabc, abab\}$, Abb. 3.3 (a) zeigt den Automat nachdem ab integriert wurde, (b) nachdem ganz I enthalten ist.

Bemerkung:
Der Algorithmus liefert einen bis auf Isomorphie (d.h. Umbenennung der Zustände) eindeutigen Automaten. Weiterhin ist zu beachten, daß in dem letzten Schritt nicht notwendig ein „neuer“ Zustand zur Endzustandsmenge hinzugefügt wird.

Lemma 6 $L(M) = I$

Begründung:
Aufgrund der Konstruktion ist klar, daß M jedes Wort aus I erkennt, somit gilt offensichtlich $I \subseteq L(M)$.

Betrachten wir ein beliebiges Wort $w \in L(M)$, das M also in einen akzeptierenden Zustand überführt.

$\delta(z_0, u)$ ist für jeden Präfix u von w definiert, mit vollständiger Induktion über die Länge von u läßt sich zeigen, daß es für jede Länge von u ein Wort in I gibt, daß ebenfalls den Präfix u besitzt. Da w Präfix von w ist, gibt es demnach auch ein Wort in I welches w als Präfix hat.

Angenommen $w \notin I$, der Zustand $\delta(z_0, w)$ wird nur genau bei der Erkennung von Wörtern passiert, die den Präfix w besitzen, da die Eindeutigkeit dieses Zustandes aus der

Präfixbaumautomaten-Eigenschaft folgt. $\delta(z_0, w)$ wird aber nur genau dann in die Menge E aufgenommen, wenn ein Wort aus I existiert, bei dessen Erkennung M in diesen Zustand überführt wird. Dieses Wort muß also den Präfix w besitzen und darf auch nicht länger sein als w, somit ist also $w \in I$, Widerspruch zur Behauptung, d.h. $w \in I$.
$\Rightarrow L(M) \subseteq I$

Der nachfolgende Standardalgorithmus verschmilzt Zustände, solange ein gewisses Kriterium erfüllt ist:

Standardalgorithmus

Sei $M = (Z, \Sigma, \delta, z_0, E)$
Wiederhole $\forall q_i, q_j \in Z$ mit $q_i \neq q_j$:
  Falls q_i, q_j die Verschmelzungsbedingung erfüllen:
    Verschmelze q_i, q_j in M und erhalte M'
    $M := M'$
Solange bis es keine $q_i, q_j \in Z, q_i \neq q_j$, gibt mit:
  (q_i und q_j erfüllen die Verschmelzungsbedingung)

„Verschmelzen" von zwei Zuständen q_i, q_j mit $q_i \neq q_j$ bedeutet informal, daß diese beiden Zustände aus dem Automaten entfernt werden und ein neuer Zustand q_k (der vorher noch nicht im Automaten war) eingeführt und δ so modifiziert wird, daß alle Zustandsübergänge von beliebigen Zuständen (also auch q_i, q_j selbst) nach q_i bzw. q_j jetzt nach q_k überführen und alle Zustandsübergänge ausgehend von q_i bzw. q_j zu beliebigen Zuständen (also auch q_i, q_j selbst) jetzt ihren Ursprung in q_k haben.

Der neue Zustand q_k befindet sich in der Endzustandsmenge E gdw. mindestens einer der Knoten q_i, q_j war vorher in der Endzustandsmenge E, falls q_i oder q_j der Startzustand waren, wird im konstruierten generelleren Automaten q_k der neue Startzustand. Die formale Definition befindet sich in Anhang E.

Bemerkung:
Beachte, daß diese formale Definition des Verschmelzens von Zuständen sicherstellt, daß $L(M') \supseteq L(M)$ ist. Führen wir demnach auf dem Ergebnis wieder eine Zustandsverschmelzung aus und verfahren so iterativ weiter bis zu einem Automaten M'', so gilt auch $L(M'') \supseteq L(M)$. Weiterhin ist klar, daß die Mächtigkeit der Zustandsmenge bei jedem Verschmelzungsschritt um eins reduziert wird.

Beachte auch, daß wenn M ein DEA ist, M' i.Allg. kein DEA ist, was damit zusammenhängt, daß es einen mit a markierten Zustandübergang von q_i nach q_s und einen ebenfalls mit a markierten Zustandübergang von q_j nach q_t mit $q_s \neq q_t$ in M gegeben haben kann, welche in zwei neuen Zustandsübergängen in M' resultieren, die ihren Ursprung in q_k haben, nach q_s bzw. q_t übergehen und beide mit a markiert sind.

Das Verschmelzungskriterium für k-umkehrbare Sprachen ergibt sich aus der Definition für k-Umkehrbarkeit wie folgt:

Verschmelze q_i, q_j falls
$\exists q' \in Z, a \in \Sigma : (q', a, q_i), (q', a, q_j) \in \delta$
oder
$\exists q'_i, q'_j \in Z, w \in \Sigma^* \; mit \; |w| = k \; und \; q_i \in \delta(q'_i, w), q_j \in \delta(q'_j, w) :$
  $(q_i, q_j \in E \; oder \; \delta(q_i, a) \cap \delta(q_j, a) \neq \emptyset)$

Also einerseits wird solange verschmolzen, bis der Automat deterministisch ist und andererseits solange, bis es keine zwei Pfade mit w mehr gibt, die entweder in *verschiedenen* Enzuständen enden oder aber mit a fortgesetzt (also wa) im selben Zustand ankommen. Dadurch wird der Determinismus von M und der Determinismus mit Vorausschau k von M^r sichergestellt.

Der Standardalgorithmus verschmilzt ausgehend von dem Präfixbaumautomaten für I solange Zustände, wie die genannten Bedingungen verletzt sind. Angluin zeigt dann, daß der Algorithmus einen endlichen Automaten liefert, der die kleinste k-umkehrbare Sprache liefert, welche I enthält und weiterhin, daß die erzeugten Automaten nach M_i nach i Verschmelzungsschritten und der bisherigen Abarbeitung von $w_1, \ldots, w_i$ als Anfang einer beliebigen Darstellungen für L gegen den Minimalautomaten für L konvergieren.

Andere Kriterien werden dann in den Abschnitten 3.2.3 und 3.2.4 vorgestellt.

3.2.3 Ahonen und Manila

Ahonen et al. arbeiten schon seit vielen Jahren unter anderem an dem Problem der Erstellung eines Inhaltsmodelles für SGML. Das Problem für Inhaltsmodelle für XML ist etwas einfacher, da viele Schwierigkeiten entfallen, die mit dem &-Operator zu tun haben. Die Arbeiten [2] und [5] haben aber nicht viel mit dem &-Operator zu tun, so daß sich die Ansätze eins-zu-eins auf das Erstellen von Inhaltsmodellen für XML-Dokumente übertragen lassen.

Dieser Ansatz ist einer der zwei vielversprechenden Möglichkeiten, sozusagen das zielgerichtete Automatenorientierte (also Graphenorientierte) Top-Down-Vorgehen, im Gegensatz zu dem Ansatz von XTRACT, den man als Bottom-Up-Methode bezeichnen könnte, der rein auf Zeichensequenzen arbeitet und einen Suchraum aufspannt, also in gewisser Hinsicht nicht so „zielorientiert“ wie der Ansatz von Ahonen arbeitet.

Andere Ansätze (wie DTD-Miner) sind hierzu im Vergleich etwas naiv, was bedeuten soll, daß sie nur auf bestimmten Klassen von XML-Dokumente vernünftige Ergebnisse liefern.

Die Arbeiten [2], [4] und [3] werden als Gesamtheit nochmals in [1] wiedergegeben, zwar inhaltlich erweitert, aber ohne die nachfolgend angesprochenen Probleme der Einzelartikel zu lösen.

In [2] sind leider ziemlich viele Sprünge und auch einige kleine Ungenauigkeiten. Einige Aspekte wurden auch nur angesprochen und nicht endgültig ausgeführt. Um hier eine geschlossene Darstellung zu geben, sind alle wesentlichen Zusatzinformationen, die vom Autor dieser Diplomarbeit stammen, als Lemmata präsentiert. Falls diese Aussagen schon in den Arbeiten von Ahonen et al. enthalten sind, dann ist das entsprechend gekennzeichnet.

Der Grundalgorithmus besteht aus vier Schritten:

- für jeden Elementtyp wird ein endlicher Automat konstruiert,
- der erhaltene endliche Automat wird generalisiert entsprechend einer Generalisierungsbedingung,
- es wird sichergestellt, daß der Automat nicht-1-mehrdeutig ist,
- der Ergebnisautomat wird umgeformt in einen regulären Ausdruck (bzw. in ein Inhaltsmodell)

Die Konstruktion des endlichen Automaten

Der endliche Automat für das Element e soll zunächst die Eigenschaft haben, genau die Sprache zu erkennen, die die Beispielmenge bildet, d.h. die konkreten Sequenzen von Elementen die als Inhalt von e auftauchen. Hierfür wird gem. dem Standardalgorithmus in 3.2.2 verfahren, wobei I die Menge der Sequenzen von Elementen aus den XML-Dokumenten ist, die als Inhalt von e an irgendeiner Stelle auftreten und Σ ist die Menge aller auftretender Elementtypen in den XML-Dokumenten.

Nach Lemma 6 erkennt der Präfixbaum-Automat M nun genau I, der nächste Schritt besteht nun darin, $L(M)$ zu generalisieren, d.h. M dahingehend zu transformieren, daß $L(M) \supseteq I$

ist, unter Einhaltung gewisser Kriterien, da ja wie bereits diskutiert nicht jede Generalisierung sinnvoll ist.

An dieser Stelle sei bemerkt, daß wir ausgehend von dem Präfixbaum-Automaten M einen regulären Ausdruck r (bzw. Inhaltsmodell) in $O(|E|)$ erhalten können, mit $L(r) = L(M)$ und r ist nicht-1-mehrdeutig, was aus der besonderen Struktur des Präfixbaum-Automaten folgt, denn wir können den Präfixbaum-Automaten von vorne nach hinten „ablesen" und bei Verzweigungen den |-Operator in den Ausdruck einfügen.

Ein generellerer Automat M' zu M, d.h. $L(M') \supseteq L(M)$, hat nun offenbar die Eigenschaft, daß es für jedes $w \in \Sigma^*$, für welches wir in M ausgehend von z_0 eine Folge von Zustandübergängen haben, die in einem Endzustand endet, eine ähnliche Folge von Zuständsübergängen in M' gibt, die dort ausgehend von z_0 ebenfalls in einem Endzustand endet.

Eine Möglichkeit zu einem generellerem Automaten konstruktiv überzugehen wäre, den ursprünglichen Automaten M als „Gerüst" bestehen zu lassen und mit neuen Zuständen und Zustandsübergängen zu erweitern, eine andere Möglichkeit wäre das „Verschmelzen" von Zuständen.

Einerseits soll die Verschmelzungsbedingung eine sinnvolle Generalisierung gewährleisten, andererseits wäre es schön, wenn der nach Ablauf des oben genannten Algorithmus konstruierte Automat gewisse Eigenschaften hat, wie z.B. daß der Automat deterministisch ist.

Für die Generalisierung legen Ahonen und Mannila folgende Heuristik zu Grunde: Wenn eine hinreichend lange Sequenz von Nichtterminalen (d.h. hier Elementen) an zwei verschiedenen Stellen innerhalb von Sequenzen von I auftaucht, so ist das, was in der Sprache hinter diesen Sequenzen folgen kann unabhängig von der Position, an der diese Sequenz auftritt. Diese Eigenschaft wird im Folgenden k-Kontextualität genannt, wobei k als Parameter für die Länge der angesprochenen „hinreichend" langen Sequenz steht.

Diese Sequenz legt also praktisch die Position im ursprünglichen Inhaltsmodell (bzw. den Zustand im endlichen Automaten) fest, denn ein bestimmtes Element tritt im Inhaltsmodell i.allg. mehrfach auf.

Beim Präfixbaum-Automaten beobachten wir, daß jeder Zustand (außer dem Startzustand) nur eine einlaufende Kante hat, wir können also den Zustand mit dem Symbol der einlaufenden Kante identifizieren. Später werden wir sehen, daß diese Eigenschaft auch bei einigen Verschmelzungskriterien erhalten bleibt.

Im Unterschied zum Gesamtgraphen des DTD-Miner Ansatzes (vgl. Abschnitt 3.1.1) werden aber i.d.R. *nicht* alle Zustände, denen das gleiche Symbol zugeordnet ist, zu *einem einzigen* Zustand verschmolzen, denn wir werden auch sehen, daß verschiedene Zustände später auch zu verschiedenen Positionen in abgeleiteten Inhaltsmodell korrespondieren, wobei im Gesamtgraphen die Informationen schon zu stark „zusammengeworfen" wurden.

Hier offenbaren sich nun die Schwierigkeiten:

- Welches Kriterium verschmilzt die „richtigen" Knoten miteinander?
- Wie erhalten wir dann das Inhaltsmodell selbst?
- Ist das Inhaltsmodell gültig, d.h. ist es nicht-1-mehrdeutig?

Steht a_i im Inhaltsmodell CM', so werden in daraus abgeleiteten Worten oftmals die selben Teilworte (bestimmter Länge) a_i voranstehen, aus diesem Grunde erscheint die k-Kontextualität als Verschmelzungskriterium sinnvoll, wobei k dann die Länge dieser Teilworte bestimmt.

Definition:
Eine reguläre Sprache L heißt **k-kontextuell**, wenn es einen k-kontextuellen Automaten M mit $L(M) = L$ gibt.

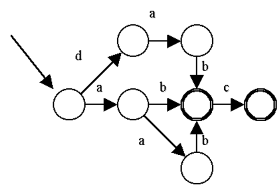


Abbildung 3.4: 2-kontextueller Automat

Definition:
Ein deterministischer endlicher Automat $M = (Z, \Sigma, \delta, z_0, E)$ heißt **k-kontextuell**, wenn für alle Worte $w \in \Sigma^*$ mit $|w| = k$ und alle Zustände $q_i, q_j \in Z$ gilt: $\delta(q_i, w) = \delta(q_j, w)$, falls beide definiert.

Wenn wir aus unserem Präfixbaumautomaten einen k-kontextuellen Automaten erzeugen wollen, können wir den oben angegebenen Verschmelzungsalgorithmus mit folgendem Verschmelzungskriterium benutzen:

Verschmelzungskriterium für k-Kontextualität:
q_i und q_j erfüllen das Verschmelzungskriterium :gdw.
$\exists w \in \Sigma^*, \exists q_0, p_0 \in Z$ mit $\delta(q_0, w) = q_i$ und $\delta(p_0, w) = q_j$

Offenbar ist der Automat, der nach der Anwendung des Standardalgorithmus mit dem oben genannten Kriterium für die k-Kontextualität erzeugte Automat k-kontextuell.

Beispiel:
Abbildung 3.4 zeigt das Ergebnis des Standardalgorithmus angewandt auf den Automaten aus Abb. 3.3 (b) mit $k = 2$.

Lemma 7 *Der Standardalgorithmus mit dem Verschmelzungskriterium für k-Kontextualität liefert ein eindeutig bestimmtes Ergebnis bis auf Zustandsumbenennung.*

Beweis:
Seien o.B.d.A. alle Zustände $Z = \{z_0, ..., z_m\}$ zu Beginn mit den Mengen $\{0\}, \{1\}, ..., \{m\}$ bezeichnet und wenn zwei Zustände verschmolzen werden, die mit $U_1, U_2 \subseteq \{0, ..., m\}$ bezeichnet sind, so wird das Ergebnis mit $U_1 \cup U_2$ bezeichnet. Somit trägt jeder Zustand die Kennzeichnung, welche ursprünglichen Zustände in ihm aufgegangen sind.

Überführt ein Wort w zu einem beliebigen Zeitpunkt im Ablauf des Algorithmus den Zustand U_1 in den Zustand U_2, so gilt nach beliebigen weiteren Schritten (also Verschmelzungen), daß w den Zustand U_1' in den Zustand U_2' überführt mit $U_1 \subseteq U_1'$ und $U_2 \subseteq U_2'$. Das folgt direkt aus der Definition der Verschmelzung.

Man kann sich demnach leicht überlegen, daß wenn der Algorithmus zu einem Zeitpunkt die Möglichkeit hat, entweder die Verschmelzung V_1 (von U_1 und U_2) oder die Verschmelzung V_2 (von U_3 und U_4) auszuführen und o.B.d.A. V_1 ausführt, daß er dann hinterher immer noch die Möglichkeit hat, V_2 auszuführen. Es kann dabei natürlich sein, daß V_2 ($U_1 \cup U_2$) mit U_3 bzw. U_4 verschmilzt, wenn $U_1 \cup U_2$ nicht disjunkt zu $U_3 \cup U_4$ ist.

Es ist also für das Ergebnis egal, welche Verschmelzung ausgeführt wird, wenn es mehrere Möglichkeiten gibt, da der Algorithmus ja solange arbeitet, bis keine Verschmelzungen mehr möglich sind.
q.e.d.

Für weitergehende Betrachtungen ist die Beobachtung wichtig, daß der erzeugte k-kontextuelle Automat M' deterministisch ist.

Lemma 8 *Sei M ein Präfixbaum-Automat entsprechend Abschnitt 3.2.2, sei $M' = (Z', \Sigma, \delta', z_0', E')$ der mit dem Standardalgorithmus erzeugte k-kontextuelle Automat. Dann gilt: M' ist deterministisch*

Beweis:
Angenommen M' ist nicht-deterministisch, dann existieren $z_k, z_i, z_j \in Z'$ mit $z_i \neq z_j$ und es existiert ein $a \in \Sigma$, so daß gilt: $(z_k, a, z_i) \in \delta'$ und $(z_k, a, z_j) \in \delta'$.
M war natürlich nach der Konstruktion deterministisch.
z_k entstand durch die Verschmelzung von mindestens zwei Zuständen aus M, hier o.B.d.A. z_1, z_2 genannt. Wäre das nicht der Fall, d.h. z_k wurde unverändert aus M übernommen, dann wären auch alle Tripel aus δ welche mit z_k beginnen unverändert übernommen worden und somit wäre das Auftreten der oben genannten beiden Tripel in δ' gar nicht möglich, da M ja deterministisch war.

Die Verschmelzung setzt voraus, daß für z_1 und z_2 das Verschmelzungskriterien gültig war, d.h. es existieren zum Zeitpunkt der Verschmelzung $q_0, p_0 \in Z$ mit $p_0 \neq q_0$ und $\delta(p_0, w) = z_1$ und $\delta(q_0, w) = z_2$ für ein Wort $w \in \Sigma^*$ mit $|w| = k$.

Nach Ablauf des kompletten Algorithmus seien p_0' und q_0' die Zustände, welche letzten Endes p_0 und q_0 ersetzt haben. Beachte, daß nach jedem Verschmelzungsschritt (z_a und z_b zu z_c) jede Zustandsübergangsfolge (d.h. entsprechend gelabelter Pfad) von z_m nach z_p mit dem Wort $v \in \Sigma^*$ immer noch vorhanden ist, nur sind alle Auftreten von z_a und z_b in dieser Folge durch den sie ersetzenden Zustand z_c zu ersetzen.

Also gibt es auch in M' noch die Zustandsübergangsfolge von p_0' nach z_k und q_0' nach z_k mit Hilfe des Wortes w.

Sei $w = a_1 a_2 ... a_k$ und es existieren p_1' und q_1' mit $(p_0', a_1, p_1'), (q_0', a_1, q_1') \in \delta'$.
Offenbar ist $u := a_2 ... a_k a$ ein Wort der Länge k, für das gilt: $z_i \in \hat{\delta}'(\{p_1'\}, u)$ und $z_j \in \hat{\delta}'(\{q_1'\}, u)$, also ist nach Beendigung des Algorithmus das Verschmelzungskriterium für z_i und z_j erfüllt $\Rightarrow$ Widerspruch zur Abbruchbedingung des Algorithmus
Die Annahme war also falsch, demnach ist M' deterministisch.
q.e.d.

Definition:
Sei I eine Sprache.
Eine k-kontextuelle Sprache L heißt **minimale k-kontextuelle Sprache, die I enthält**, gdw. $L \supseteq I$ und $\forall L' \supseteq I, L'$ k-kontextuell gilt: $L' \supseteq L$

Ahonen und Manila schreiben, daß man zeigen kann, daß eine eindeutige minimale k-kontextuelle Sprache die I enthält existiert und der genannte Algorithmus mit dem Verschmelzungskriterium für k-Kontextualität auch einen Automaten mit dieser minimalen Sprache erzeugt.

Der Beweis folgt jetzt an dieser Stelle:

Lemma 9 *L ist k-kontextuell[4] gdw.*
L ist regulär und für alle $u, v, w, x, y \in \Sigma^* : |v| \geq k$ und $uvw, xvy \in L \Rightarrow uvy, xvw \in L$

Beweis:
$\Rightarrow$: Sei der DEA M k-kontextuell mit $L(M) = L$, M existiert nach der Definition der k-Kontextualität. uvw und xvy überführen M vom Startzustand in jeweils einen Endzustand, sei p der Zustand von M nach dem Erkennen von uv und q der Zustand von M nach Erkennen von xv, offenbar gilt $p = q$ denn M ist k-kontextuell und v besitzt einen Suffix der Länge k. Mit

[4] Das ist äquivalent zur Definition von Muggleton [26]: L ist k-kontextuell :gdw. L ist regulär und ($uvw, xvy \in L$ mit $|v| = k \Rightarrow T_L(uv) = T_L(xv)$)

$p = q$ kann also nach dem Erkennen von uv auch y bzw. nach dem Erkennen von xv auch w erkannt werden, somit sind uvy und xvw in L.

$\Leftarrow$: Sei $M = (Z, \Sigma, \delta, z_0, E)$ der minimale DEA der L erkennt. Offenbar ist M genau dann k-kontextuell, wenn der Standardalgorithmus mit dem Verschmelzungskriterium für k-Kontextualität M nicht ändert, denn jede Zustandsverschmelzung auf einem minimalen DEA hat eine Generalisierung der Sprache zur Folge, da alle minimalen DEAs die dieselbe Sprache erkennen, isomorph zueinander sind.

Angenommen, der Algorithmus führt eine Zustandverschmelzung aus, o.B.d.A. werden $p, q \in Z$ mit $p \neq q$ miteinander verschmolzen, da ein Wort $v \in \Sigma^*$ mit $|v| = k$ irgendwelche Zustände p' und q' nach p bzw. q überführte.

Da M minimal ist, müssen p und q aber unterscheidbar sein, d.h. es existiert o.B.d.A. ein Wort $w \in \Sigma^*$ mit $\delta(p, w) \in E$ und $\delta(q, w) \notin E$.

Wegen Ms Minimalität sind p' und q' auch von z_0 aus erreichbar, seien die entsprechenden Worte u bzw. x. Ebenso muß es ein Wort y geben mit $\delta(q, y) \in E$, da q sonst überflüssig wäre und M wegen seiner Minimalität keine überflüssigen Zustände enthält.

Zusammengefaßt überführen die Worte uvw und xvy M in einen Endzustand, das Wort xvw aber nicht, also gilt $uvw, xvy \in L$ und $xvw \notin L$, da M deterministisch ist. Nach der Voraussetzung und der Tatsache, daß $|v| = k$ ist, muß aber $xvw \in L$ sein, was ein Widerspruch ist, die Annahme ist also falsch, d.h. der Algorithmus führt keine Zustandsverschmelzungen aus, M und somit L sind also k-kontextuell.
q.e.d.

Lemma 10 *k-kontextuelle Sprachen sind unter Durchschnittsbildung abgeschlossen*

Beweis:
Seien L_1, L_2 k-kontextuell, sei $L_3 := L_1 \cap L_2$.
L_3 ist bekanntermaßen regulär, seien für $u, v, w, x, y \in \Sigma^* : |v| \geq k$ und $uvw, xvy \in L_3$
Somit gilt auch $uvw, xvy \in L_1$ und $uvw, xvy \in L_2$, aus der k-Kontextualität von L_1 bzw. L_2 folgt auch $uvy, xvw \in L_1$ und $uvy, xvw \in L_2$.
Somit gilt auch $uvy, xvw \in L_3$; L_3 ist also k-kontextuell.

Lemma 11 *Sei der DEA M k-kontextuell und M' der minimale DEA mit $L(M) = L(M')$. Dann ist M' k-kontextuell.*

Beweis:
Angenommen in M' existiert ein $w \in \Sigma^*$ mit $|w| = k$, so daß $\delta'([p_0], w) = [p_k]$ und $\delta'([q_0], w) = [q_k]$ mit $[q_k] \neq [p_k]$.
w überführt in M' $[p_0]$ nach $[p_1]$, $[p_1]$ dann nach $[p_2]$, usw. In M überführt w p_0 nach p'_1, p'_1 nach p'_2, usw. wobei gemäß der Funktionsweise des Minimierungsalgorithmus gelten muß: $p'_1 \in [p_1]$ und per Induktion dann auch $p'_i \in [p_i]$.
Es gilt also $p'_k \in [p_k]$ und analog $q'_k \in [q_k]$ wobei durch die k-Kontextualität von M zwingend gilt: $p'_k = q'_k$ und durch die Eigenschaft von Äquivalenzrelationen damit auch $[p_k] = [q_k]$ im Widerspruch zur Annahme, also ist M' k-kontextuell.

Lemma 12 *Jeder k-kontextuelle endlicher Automat M ist auch k-umkehrbar.*

Beweis:
Per Definition ist M zunächst einmal deterministisch, was auch Voraussetzung für die k-Umkehrbarkeit ist. Betrachten wir die Eigenschaften eines k-umkehrbaren Automaten genauer, so ist es unter gewissen Randbedingungen verboten, daß sich zwei Zustände z'_1 und z'_2 mit einem

Wort w mit $|w| = k$ in $z_1 = \delta(z_1', w)$ und $z_2 = \delta(z_2', w)$ überführen lassen mit $z_1 \neq z_2$.
k-Kontextualität bedeutet aber eben selbes Verbot *ohne* Randbedingungen, klarerweise ist also jeder k-kontextuelle Automat ein k-umkehrbarer Automat.
Damit ist dann auch jede k-kontextuelle Sprache eine k-umkehrbare Sprache.

Definition:
Die von I implizierte k-kontextuelle Sprache L_I wird wie folgt definiert:
$L_I := \lim_{n\to\infty} L_{I_i}$
mit
$L_{I_0} := I$ und für $i > 0$: $L_{I_i} := L_{I_{i-1}} \cup \{uvy, xvw \mid uvw, xvy \in L_{I_{i-1}}, |v| \geq k\}$
Es ist an dieser Stelle noch nicht klar, daß L_I regulär ist.
Es ist aber klar, daß gelten muß: (L k-kontextuell und $I \subseteq L$) $\Rightarrow L_I \subseteq L$

Lemma 13 *Sei I die Eingabemenge wie oben beschrieben. Es existiert genau eine Sprache L mit: L ist k-kontextuell, $L \supseteq I$ und für alle L' mit $L' \neq L$: $I \subseteq L' \subseteq L \Rightarrow L'$ nicht k-kontextuell.*
Diese Sprache ist L_I und wird vom Standardalgorithmus geliefert.
Muggleton zeigt diese Aussage in [26] auch, benutzt dazu aber unzählige andere Sätze und Lemmata.

Beweis:
Angenommen, es gäbe zwei verschiedene Sprachen L_1 und L_2 die die Bedingung erfüllen.
Nach Lemma 10 ist aber auch $L_1 \cap L_2$ k-kontextuell und es gilt $I \subseteq L_1 \cap L_2$. Daraus folgt unmittelbar $L_1 = L_2$, denn es gilt allgemein: $L_1 \cap L_2 \subseteq L_i$ für $i \in \{1, 2\}$.
Hiernach wissen wir, daß es höchstens eine Sprache mit der geforderten Eigenschaft gibt und der Standardalgorithmus liefert zunächst einmal überhaupt eine Sprache L für die gilt: L ist k-kontextuell und $L \supseteq I$.

Beim Verschmelzen von Zuständen könnte es geschehen, daß Worte ableitbar sind, die nicht in L_I sind. Möglich ist dies ausschließlich dann, wenn wir z_i und z_j zu z_k verschmelzen, weil es Pfade mit Suffix v, $|v| = k$ zu z_i und z_j gab und es o.B.d.A. zu z_i einen Pfad u von z_0 gab, der nicht mit v endete. Dadurch könnten nach Abarbeitung dieses Pfades u alle Pfade die z_j in einen Endzustand überführten ebenfalls abgearbeitet werden, diese neuen Worte sind aber nicht notwendig in der von I implizierten Sprache L_I.

Wir zeigen jetzt, daß es für jeden Zustand der vom Standardalgorithmus gelieferten Automaten M' entweder nur genau einen Pfad der Länge $< k$ gibt oder eine Menge von Pfaden der Länge $\geq k$, wobei alle Pfade denselben Suffix v mit $|v| = k$ haben. Damit kann dann die geschilderte Situation überhaupt nicht auftreten.

Betrachten wir den Ablauf des Standardalgorithmus genauer, so werden wir feststellen, daß kein Zustand zu dem es von z_0 nur einen einzigen Pfad gibt, der kürzer k ist, jemals an einer Verschmelzung teilnimmt. Zu Beginn gibt es ja zu jedem Zustand nur genau einen Pfad, was aus der Präfixbaumeigenschaft folgt.

Sei Z_0 die Menge dieser Zustände und Z_1 die Menge der Zustände, zu denen es im Präfixbaumautomaten nur einen Pfad der Länge $\geq k$ gibt. D.h. Verschmelzungen finden nur von Zuständen aus Z_1 statt oder den Zuständen, die hieraus hervorgehen. Eine Induktion über die Anzahl der Verschmelzungen zeigt, daß es keine Pfade mit einer Länge $< k$ vom Startzustand zu verschmolzenen Zuständen gibt. Vor der ersten Verschmelzung ist das klar. Für beliebige spätere Verschmelzungen von z_i und z_j zu z_k gilt:
Sei J die Menge der Worte, die z_0' nach z_k überführt: $J := \{w \mid \delta'(z_0', w) = z_k\}$

Klarerweise gilt:
$J = J_1 \cup J_2$; wobei:

$J_1 := \{w \mid \delta(z_0, w) = z_i \text{ } oder \text{ } \delta(z_0, w) = z_j\}$ sowie
$J_2 := \{u \mid \delta(z_0, u) = z_i \text{ } oder \text{ } \delta(z_0, u) = z_j\}\{v \mid \delta(z_t, v) = z_l \text{ } mit \text{ } \{t, l\} \subseteq \{i, j\}\}^*$

Beachte, daß $J_1 \subseteq J_2$ und alle Wörter aus J_2 mit einem Wort aus J_1 beginnen, also kommen keine kürzeren Pfade hinzu, als schon in J_1 sind, welche aber nach Voraussetzung alle mindestens k Zeichen lang sind.

Genauso können wir mit einer vollständigen Induktion über die Verschmelzungen zeigen, daß diese Pfade in J alle mit demselben Suffix w enden. Die Pfade in J_1 tun dies nach Induktionsvoraussetzung, alle Pfade von z_t nach z_l mit $\{t, l\} \subseteq \{i, j\}$, welche eine Länge $\geq k$ aufweisen, tun dies ebenfalls nach Induktionsvoraussetzung, so daß wir nur die Pfade in J_3 mit $J_3 := \{v \mid |v| < k \text{ } und \text{ } \delta(z_t, v) = z_l \text{ } mit \text{ } \{t, l\} \subseteq \{i, j\}\}$ näher betrachten müssen.

Sei $v_i \in J_3$ beliebig, J_3 ist natürlich endlich durch die Längenbeschränkung. Jedes v_i liefert Informationen über den Aufbau von w, denn ist v_i o.B.d.A. ein Pfad von z_i nach z_j, so überlappen sich die k-Suffixe (also Suffixe der Länge k) des Pfades von z_0 nach z_i, der mit w endet und des Pfades von z_0 nach z_i und der dann mit v fortgesetzt wird – und ebenfalls mit w endet – um n_i Zeichen, mit $n_i > 0$, da $|v| < k$. (Siehe auch Abbildung 3.5 (a))

Ist $n_i \leq \frac{k}{2}$, so muß $w = aba$ sein, mit $a, b \in \Sigma^*$ und $|a| = n_i$. (Abbildung 3.5 (b)). Je länger ein Wort aus J_3 ist, desto kleiner ist die Überlappung. Wir betrachten o.B.d.A. die Elemente aus J_3 geordnet nach der Länge und die kürzesten als erstes, wobei es keine zwei gleichlangen Elemente in J_3 geben kann, da jedes Wort aus J_3 ein Suffix von w ist; es gilt dann: $n_1 > n_2 > n_3 > \dots$

Gilt wie oben schon erwähnt $n_1 \leq \frac{k}{2}$, so impliziert das $w = aba$, $|a| = n_1$.
Gilt dann $n_2 \leq \frac{n_1}{2}$, so folgt zunächst $w = cdc$ mit $|c| = n_2$, a fängt demnach mit c an und endet auch mit c, b ist in der Mitte und kürzer als d, also gilt hiernach $w = cecbcec$, a wurde verfeinert in cec.

Letztenendes haben wir dann w als symmetrischen Binärbaum zerlegt, wobei jeder Knoten eine Zeichenkette trägt (möglicherweise ϵ) und sich w ergibt aus der Konkatenation der Zeichenketten in Infixorder. (Abbildung 3.5 (c))

Bisher haben wir die Fälle nicht diskutiert, daß $n_1 > \frac{k}{2}$ oder auch $n_{i+1} > \frac{n_i}{2}$. Überlappt sich ein Wort selbst aber um mehr als die Hälfte, so folgt offensichtlich (Abbildung 3.5 (d)), daß $w = aa...a$ mit einer gewissen Anzahl (mindestens 3) a's ist. a kann wiederum verfeinert werden in cec oder mit gleicher Überlegung in $bb...bb$ für mindestens 3 b's.

Haben wir alle solche Strukturinformationen für w aus J_3 abgeleitet, so ist w also als symmetrischer Baum darstellbar, wobei jeder innere Knoten mindestens 2 Kindknoten hat und die Inhalte aller Kindbäume gleich sind. Die Inhalte des Baumes (bzw. von Teilbäumen) ergeben sich durch Konkatenation der Zeichenkette in Infixorder (binärer Fall) bzw. von links nach rechts (n-ärer Fall, $n \geq 3$).

Ob ein Knoten *zwei* oder n Kindknoten hat, kann in jeder Baumebene wechseln. Letztendlich ist jedes v_i ein Suffix von w und wenn wir versuchen, v_i als Baum analog zu w darzustellen, beobachten wir zwei Tatsachen:

- Der v_i-Baum ist ein Teilbaum des w-Baumes, denn es fand ja eine Zerlegung in kleinste gemeinsame Zeichenkettenteile statt.
- Dem Wurzelknoten des v_i-Baumes – der *immer* gleich dem Wurzelknoten des w-Baumes ist – fehlen höchstens Kindknoten an „der linken Seite“, diese Aussage gilt auch absteigend für alle Knoten des v_i-Baumes.

Das Konkatenieren zweier Wörter v_1, v_2 aus J_3 entspricht dem Hintereinanderfügen der entsprechenden Bäume für v_1, v_2. Ist $n_1 \leq \frac{k}{2}$ so enthält jeder dieser Bäume den Wurzelknoten und den vollständigen rechten Teilbaum des Baumes für w, anderenfalls ($w = aa...a$) gewisse a-Teilbäume. Evt. ist der linke Kindknoten k_{links} des Wurzelknotens im binären Fall enthalten, falls ja, dann auch der *vollständige* rechte Teilbaum unterhalb von k_{links}. Evt. ist auch der linke Kindknoten von k_{links} enthalten usw.

Falls auf einer Ebene eine Verfeinerung $a = bb...b$ stattfand, dann nicht der vollständige rechte Teilbaum, sondern eine gewisse Anzahl von b-Teilbäumen und ebenso gilt dann die Argumentation nicht für den linken Kindknoten sondern für den am weitesten links stehenden Kindknoten.

Alle k-Suffixe von allen Pfaden nach z_i enden ja mit w, so daß sich dieses Pfadende der Länge k auch als Baum darstellen läßt, also der vollständige Baum für w (siehe Abbildung 3.5 (f), wobei die gestrichelte Linie den Baum für w „ergibt“).

Ist $n_1 > \frac{k}{2}$, so legen wir entsprechend der Konkatenation von $v_1 v_2 ... v_n$ Bäume beginnend mit dem Baum für w nebeneinander. Betrachten wir die Ebene unter den Wurzelknoten ausgehend von rechts, so liegen hier nur a-Knoten und die Wurzelknoten enthalten ϵ, also endet $wv_1 v_2 ... v_n$ mit w (siehe Abbildung 3.5 (e); sei hier z.B. $w = aaaa$).

Ist $n_1 \leq \frac{k}{2}$, so fehlt v_n nur ein gewisser linker Teilbaum (oder eine Anzahl gleicher b-Teilbäume) im Vergleich zu w, egal, ob der Pfad mit $...wv_n$ oder $...v_{n-1}v_n$ endet. Durch die absolute Symmetrie sind in w bzw. v_{n-1} aber genau diese fehlenden Teile (auf der rechten Seite) enthalten. Legen wir die Bäume von w (bzw. v_{n-1}) und v_n nebeneinander, so sehen wir auch hier, daß der Pfad mit w endet.
q.e.d.

Bemerkung:
Aus diesem Satz folgt natürlich auch (wenn k > 0), daß für alle $z_i \in Z$, ein Symbol $a \in \Sigma$ existiert, so daß alle Zustandsübergänge aus δ, die nach z_i überführen, dies mit dem Symbol a tun.

Beachte aber, daß die Aussage von Lemma 13 *nicht* für einen beliebigen k-kontextuellen Automaten gilt!

Entsprechend der Diskussion in Abschnitt 3.2.1 identifiziert der Standardalgorithmus eine k-kontextuelle Sprache also in den Grenzen.

Im Beispiel in [2] wird die Zustandsverschmelzung fehlerhaft – vor allem bezüglich der Endzustände – ausgeführt, was dann tatsächlich dazu führt, daß der erzeugte k-kontextuelle Automat M' zu dem ursprünglichen Präfixbaum-Automaten M nicht mehr dessen Sprache vollständig enthält, also nicht $L(M') \supseteq L(M)$ gilt!

Der Automat enthält auch nur genau einen Endzustand, was aber aus dem Algorithmus nicht ableitbar ist und auch nicht motiviert wird, auch wenn Muggleton in [26] solche speziellen Automaten noch genauer diskutiert.

Offensichtlich muß die k-Kontextualität des Automaten M ihn für Ahonen und Manila noch nicht genug generalisiert haben, denn sie führen folgende Spezialisierung für die k-Kontextualität ein, wobei $0 \leq h \leq k$ gelte:

Definition:
Eine reguläre Sprache L heißt $(\mathbf{k}, \mathbf{h})$**-kontextuell**, wenn es einen (k, h)-kontextuellen Automaten M mit $L(M) = L$ gibt.

Definition:
Ein endlicher Automat $M = (Z, \Sigma, \delta, z_0, E)$ heißt $(\mathbf{k}, \mathbf{h})$**-kontextuell**, wenn für alle Worte $w = a_1 \dots a_k \in \Sigma^*$ mit $|w| = k$ und alle Zustände $z_i, z_j \in Z$ und $\forall i$ mit $0 \leq h \leq i \leq k$ gilt: $\delta(z_i, a_1 \dots a_i) = \delta(z_j, a_1 \dots a_i)$, falls $\delta(z_i, w)$ und $\delta(z_j, w)$ definiert sind[5].

Bemerkung:
(k, h)-Kontextualität impliziert nicht h-Kontextualität, was Abbildung 3.6 zeigt, in (a) ist der Präfix-Baum-Automat für $I = \{abab, ca\}$ dargestellt und in (b) der vom Standardalgorithmus

[5] Beachte, daß M per Definition durchaus nicht-deterministisch sein kann

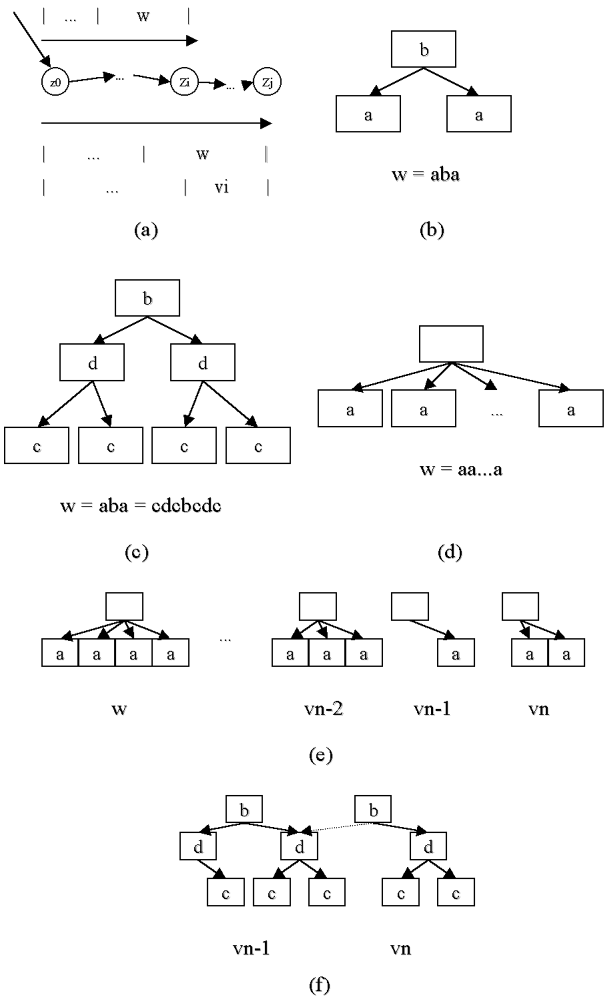


Abbildung 3.5: zum Beweis von Lemma 13

erzeugte (2,1)-kontextuelle Automat, der nicht 1-kontextuell ist. (k, h)-Kontextualität impliziert aber natürlich die k-Kontextualität, hierzu beachte man die Situation, wenn $i = k$ gilt.

Anschaulich bedeutet (k, h)-Kontextualität, daß wenn wir in der Sprache an verschiedenen Stellen das Auftreten einer Sequenz der Länge k haben, nicht nur die folgenden Elemente im XML-Dokument unabhängig von der konkreten Position der genannten Sequenz sind, was im Automaten durch den Übergang in denselben Zustand realisiert wird, sondern vielmehr die Erkennung schon vorher „in der selben Bahn“ läuft. Der Grund von der k-Kontextualität zum Zwecke der weiteren Generalisierung zur (k, h)-Kontextualität anstatt zur $(k-1)$-Kontextualität überzugehen, liegt vermutlich darin, daß im letzteren Fall der Automat übergeneralisiert wäre, beachte hier auch, daß Ahonen und Manila mit Werten wie $k = 2$ arbeiten.

Aus dem Beweis für Lemma 13 folgt, daß wir bei einem entsprechenden k-kontextuellen Automaten für jeden Zustand z_i nur genau einen Pfad kürzer k Zeichen haben, der ausgehend vom Startzustand z_0 nach z_i geht, oder aber, daß alle Pfade nach z_i eine Länge größer oder gleich k haben und mit demselben k-Suffix enden und weiterhin aufgrund der Arbeitsweise des Standardalgorithmus für $z_i, z_j \in Z$ mit $i \neq j$ auch gilt, daß die entsprechenden k-Suffixe für z_i und z_j verschieden sind. Also haben wir für jeden Zustand z_i *genau* ein Wort der Länge k, welches ihn

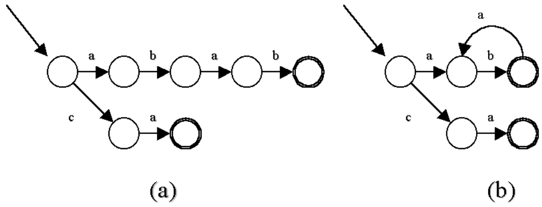


Abbildung 3.6: (2,1)-kontextueller Automat

identifiziert.

Hieraus folgt sofort, daß ein k-kontextueller Automat M eine Zustandszahl in $O(|\Sigma|^k)$ hat und dementsprechend ein $(k-1)$-kontextueller Automat der aus M hervorgeht im Durchschnitt – d.h. wenn wir die Gesamtmenge aller k-kontextuellen Automaten über Σ betrachten – nur den $\frac{1}{|\Sigma|}$-Teil der Zustände von M hat, was von der Granularität einfach nicht fein genug ist.

Beim Übergang zu (k,h)-Kontextualität gilt weiterhin, daß wir zur Zustandsidentifikation immer noch ein Wort der Länge k brauchen, also heuristisch gesehen viele Zustände verschmelzen aber trotzdem i.d.R. nicht soviele, daß wir die Zustandszahl eines $(k-1)$-kontextuellen Automaten erreichen.

Betrachten wir die Ausführungen zur (k,h)-Kontextualität von Ahonen in [1] genauer, so fallen zwei Tatsachen auf:

- Die Definition der (k,h)-Kontextualität ist inkonsistent.

  Anders als oben ist die primäre Definition der (k,h)-kontextuellen Sprachen in [1] auf lexikalischer Ebene, die Äquivalenz zu der Definition mit Automaten versucht Ahonen in Lemma 5.10 zu zeigen.

  Definition (lexikalische Variante von Ahonen):
  Sei $0 \leq h \leq k$. Eine reguläre Sprache L ist **(k,h)-kontextuell** :gdw.
  $\forall u_1, u_2, v_1, v_2, w_1, w_2 \in \Sigma^*, a_1, \ldots, a_k \in \Sigma$:
  $u_1 a_1 \ldots a_k w_1,\ u_2 a_1 \ldots a_k w_2,\ v_1 a_i \ldots a_k w_1,\ v_2 a_i \ldots a_k w_2 \in L$
  und $0 \leq h \leq i \leq k \Rightarrow T_L(v_1 a_i) = T_L(v_2 a_i)$

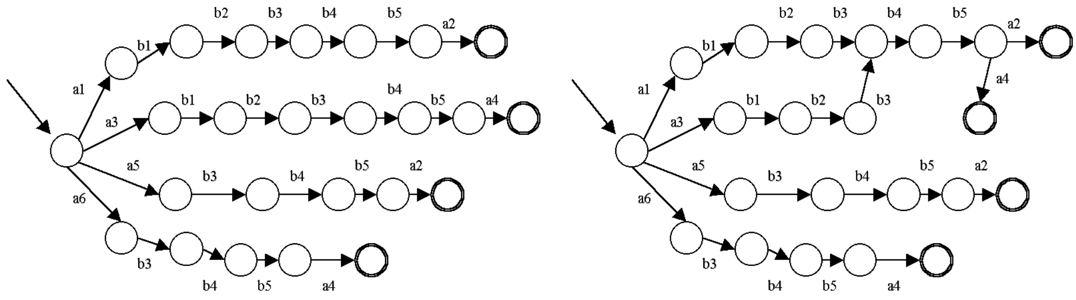
  Wenn wir das als Definition nehmen und die Äquivalenz zu den Automateneigenschaften aus der anderen Definition zeigen wollen, so folgt die Richtung *lexikalische Definition* $\Rightarrow$ *Automatendefinition*, was Ahonen auch korrekt zeigt, es gilt aber: *Automatendefinition* $\not\Rightarrow$ *lexikalische Definition*, denn:

  Sei $\Sigma := \{a_1, a_2, a_3, a_4, a_5, a_6, b_1, b_2, b_3, b_4, b_5\}$ [6] und $L = \{a_1 b_1 b_2 b_3 b_4 b_5 a_2,\ a_3 b_1 b_2 b_3 b_4 b_5 a_4,\ a_5 b_3 b_4 b_5 a_2,\ a_6 b_3 b_4 b_5 a_4\}$, sowie $(k,h) = (5,3)$. Es gibt immer einen Automaten M, für den $L(M) \supseteq L$ gilt und der (5,3)-kontextuell nach *Automatendefinition* ist, konkret betrachten wir den vom Standardalgorithmus mit dem Verschmelzungskriterium für (5,3)-Kontextualität erzeugten Automaten M.

  Abbildung 3.7 (a) zeigt den Präfix-Baum-Automat für L. Offenbar gab es nur ein einziges Teilwort (nämlich $b_1 b_2 b_3 b_4 b_5$), daß die Verschmelzung von drei Zuständen auslöste, wonach der Automat dann (5,3)-kontextuell nach *Automatendefinition* war, siehe Abb. 3.7 (b). Die Pfade für $a_5 b_3 b_4 b_5 a_2$ und $a_6 b_3 b_4 b_5 a_4$ sind unverändert geblieben.

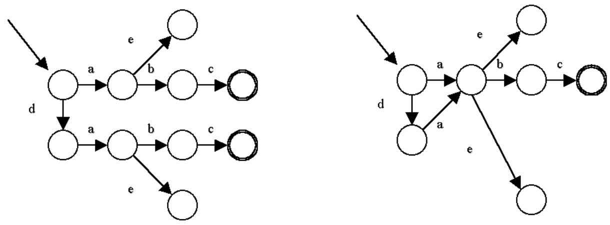
  Die *lexikalische Definition* fordert u.a. $T_L(a_5 b_3) = T_L(a_6 b_3)$, es gilt aber $T_L(a_5 b_3) = \{b_4 b_5 a_2\}$ und $T_L(a_6 b_3) = \{b_4 b_5 a_4\}$. Also ist $L(M)$ (5,3)-kontextuell nach *Automatendefinition* und *nicht* (5,3)-kontextuell nach *lexikalischer Definition*.

[6] Es gilt $|\Sigma| = 11$.



(a) (b)

Abbildung 3.7: Inkonsistenz bei der Definition von Ahonen



(a) (b)

Abbildung 3.8: Nicht-Determinismus bei (k, h)-Kontextualität

- Benutzen wir Standardalgorithmus mit dem Verschmelzungskriterium für (k, h)-Kontextualität so ist der erzeugt Automat im Gegensatz zum Verschmelzungskriterium für k-Kontextualität i.allg. nicht deterministisch, betrachte hierzu in Abbildung 3.8 (a) den Präfix-Baum-Automat zur Sprache $\{abc, dabc, ae, dae\}$ und in Abb. 3.8 (b) den zugehörigen (3,1)-kontextuellen Automaten nach Ablauf des Algorithmus, der nicht deterministisch ist.

  Das widerspricht auch nicht der Definition von Ahonen, es ist aber nicht klar, ob das die Intuition ist. Letztenendes gehen wir zu einem DEA über, wenn wir die Nicht-1-Mehrdeutigkeit erzeugen. Die Frage die sich hier stellt ist, ob hier ein Verschmelzen zu einem DEA die Sprache ändert – wie im allgemeinen Fall – oder nicht.

Mäkinen führt in [24] die Klasse der (k, h)-kontextuellen Sprachen als *identifizierbare* Klasse auf, wenn sich das aber auf die Definitionen von Ahonen stützt, ist ja diese Klasse noch nicht einmal wohldefiniert.

Setzen wir die hier gegebene *Automatendefinition* für (k, h)-kontextuelle Sprachen als korrekt voraus, so finden sich die Beweise für das Erzeugen der minimalen (k, h)-kontextuellen Sprache, die I enthält in [1], welche direkt auf den Beweisen von Muggleton für k-kontextuelle Sprachen in [26] aufbauen.

Erzeugung der Nicht-1-Mehrdeutigkeit

Der erzeugte (k, h)-kontextuelle Automat ist im Allgemeinen nicht nicht-1-mehrdeutig und auch nicht minimal. Abbildung 3.9 zeigt in (a) den 1-kontextuellen Automaten ausgehend von $I = \{ab, cb\}$ und in (b) den äquivalenten Minimalautomaten.

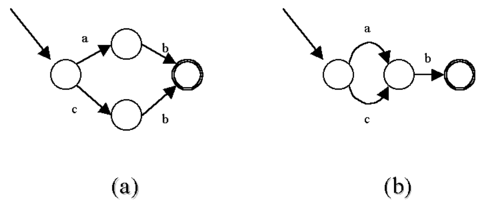


Abbildung 3.9: fehlende Minimalität

In [4] hat Ahonen einen Algorithmus nachgeliefert, der zu einem regulären Ausdruck r einen nicht-1-mehrdeutigen regulären Ausdruck r' liefert, wobei $L(r') \supseteq L(r)$ gilt, trotzdem r' aber nicht zu stark generalisiert ist. Die Grundlage hierfür liefert die genannte Arbeit [12] von Brüggemann-Klein und Wood, in der sich intensiv mit den nicht-1-mehrdeutigen Sprachen auseinandergesetzt wird.

Im ursprünglichen Artikel [2] beachten Ahonen und Manila die Nicht-1-Mehrdeutigkeit nicht weiter und verweisen zur Gewinnung des regulären Ausdruckes (bzw. Inhaltsmodelles) ausgehend vom generalisierten DEA auf den „Standardalgorithmus“ zur Erzeugung eines regulären Ausdruckes aus einem Automaten, der z.B. in [20] eingeführt wird.

Dieser Algorithmus hat eine Komplexität von $O(n^3)$ wenn n die Anzahl der Zustände ist und liefert im Allgemeinen einen regulären Ausdruck, der noch starken Vereinfachungen bedarf. Außerdem ist nicht sichergestellt, daß, wenn die Sprache des Automaten nicht-1-mehrdeutig ist, diese Tatsache auch für den generierten Ausdruck gilt.

Ausgehend vom Testalgorithmus von Brüggemann-Klein und Wood leitet Ahonen in [4] den folgenden Algorithmus ab, der einen Automaten zu einem nicht-1-mehrdeutigen Automaten generalisiert, allerdings verzichtet Ahonen weitestgehend auf eine Erläuterung:

Algorithmus Erzeuge_Nicht-1-Mehrdeutigkeit (Ahonen)

Eingabe: Ein DEA $M = (Z, \Sigma, \delta, z_0, E)$

Rückgabe: Ein gewisser NEA

(1) Falls M nur aus einem trivialen Orbit besteht, dann fertig
(2) Berechne Menge S der M-konsistenten Symbole
(3) Falls M nur aus einem einzigen nichttrivialen Orbit besteht und $S = \emptyset$ gilt,
(4) dann wähle ein Symbol a und einen Zustand $f(a) \in Z$ mit $f(a) \in \delta(q, a)$,
wobei q ein Endzustand ist
(5) Für jeden Zustand $q' \in E$ führe aus:
(6) Ist $q'' \in \delta(q', a)$ mit $q'' \neq f(a)$ und $q'' \in Z$,
(7) dann verschmelze $f(a)$ und q'' zu $f(a)$
(8) sonst $\delta(q', a) := \{f(a)\} \cup \delta(q', a)$
(9) $S := \{a\}$
(10) Berechne M_S
(11) Erzeuge_Orbit-Eigenschaft(M_S)
(12) Für alle K Orbit von M_S: für ein $x \in K$:
(13) Erzeuge_Nicht-1-Mehrdeutigkeit($(M_S)_x$)

Der oben angegebene Algorithmus ist formal nicht besonders korrekt, insbesondere wird vorausgesetzt, daß die Zustandsübergangsrelation beim rekursiven Aufruf nicht auf die Zustände der entsprechenden Orbits eingeschränkt wird, sondern quasi global verändert wird.

Genaugenommen erfolgt der rekursive Aufruf formal nicht mehr auf endlichen Automaten, der Algorithmus läßt sich so aber auf eine einfachere Art und Weise aufschreiben, als wenn mit globalen Variablen bzw. länglichen Mengenoperationen gearbeitet wird. Weiterhin wird δ unter Umständen so erweitert, daß es eine Zustandsübergangsrelation für einen NEA ist.

Der Algorithmus von Ahonen bricht ab, wenn der Automat nur aus einem trivialen Orbit besteht, ansonsten bestimmt er zunächst die Menge S aller M-konsistenten Symbole. Falls S leer ist und M nur aus einem einzigen nicht-trivialen Orbit besteht, wird ein M-konsistentes Symbol „erzeugt", hierfür wird ein Symbol $a \in \Sigma$ gewählt, für das gilt: $(z, a, z') \in \delta$ mit $z \in E$. z' übernimmt dann die Rolle von $f(a)$. Offenbar gibt es überhaupt so ein Tripel, denn entweder hat M nur einen Zustand – der dann auch Endzustand ist – und gäbe es keine Transition aus diesem Zustand hinaus wäre M trivial im Gegensatz zur Voraussetzung. Oder aber M hat mehr als einen Zustand und für jeden Endzustand gilt, es gibt irgendeine Transition aus diesem hinaus, denn wenn nicht, dann wären von diesem aus keine anderen Zustände erreichbar, also dieser Zustand gar nicht im selben Orbit (vgl. Orbit-Definition), was auch im Widerspruch zur Voraussetzung für diesen Fall steht.

Welches Symbol ggf. gewählt wird, wird mit Heuristiken bestimmt. Aus dem Satz 2 von Brüggemann-Klein und Wood folgt, daß wir so vorgehen müssen, denn ansonsten wäre M nicht nicht-1-mehrdeutig. Alle Endzustände von M, die einen Zustandsübergang für das Symbol a (innerhalb von M) besitzen, wobei der resultierende Zustand ungleich $f(a)$ ist, werden mit $f(a)$ verschmolzen, der neue resultierende Zustand übernimmt dann wieder die Rolle von $f(a)$. Alle anderen Endzustände von M erhalten einen neuen Zustandsübergang mit dem Symbol a zum Zustand $f(a)$.

Beachte daß dadurch, daß δ nicht auf Z eingeschränkt ist, Endzustände von M durchaus Übergänge mit dem Symbol $a \in \delta$ zu nicht in M enthaltenen Zuständen haben können. Durch den Verschmelzungsvorgang in Zeile (7) ist i.Allg. auch die Einschränkung von δ auf die Zustände von M nachher nicht mehr deterministisch.

Der Algorithmus *ErzeugeNicht-1-Mehrdeutigkeit* benutzt die Funktion *ErzeugeOrbitEigenschaft*. Ahonen hat den Algorithmus leicht fehlerhaft definiert, deshalb an dieser Stelle die korrekte Version:

Algorithmus ErzeugeOrbitEigenschaft

(1) Für jeden Orbit K von $M = (Z, \Sigma, \delta, z_0, E)$
(2) Seien $t_1, \ldots, t_k$ die Tore von K
(3) Falls existiert i: $t_i \in E$
(4) dann: $E := E \cup \{t_1, \ldots, t_k\}$
(5) Für $i = 1 \ldots k, a \in \Sigma$ wiederhole
(6) falls $(t_i, a, z) \in \delta$ mit $z \in Z \backslash K$
(7) dann $\delta := \delta \cup \{(t_1, a, z), \ldots, (t_k, a, z)\}$

Bemerkung:
Die Schleife in Zeile (5) kann natürlich in der Implementation je nach Darstellung der Automatenstruktur effizienter realisiert werden. Die Funktionsweise des Algorithmus dürfte direkt aus der Definition der Orbit-Eigenschaft hervorgehen, wenn wir nur E und δ ändern wollen, so führt der Algorithmus genau die minimal notwendigen Änderungen aus, um die Orbit-Eigenschaft zu erreichen. Der Algorithmus von Ahonen unterscheidet sich dahingehend, daß die Schleife wie folgt aussieht:

(5') Für jedes geordnete Paar (t_i, t_j)
(6') Falls existiert a mit $(t_i, a, z) \in \delta$ und $z \in Z \backslash K$ und $(t_j, a, z) \notin \delta$
(7') dann
(8') falls $(t_j, a, z') \in \delta$
(9') dann $\delta := \delta \cup \{(t_j, a, z)\}$

Dieser Algorithmus ist falsch und zwar aus zwei Gründen:
Zunächst einmal können o.B.d.A. $(t_i, a, z), (t_i, b, z'') \in \delta$ sein mit $a \neq b$ und $z, z'' \in Z \backslash K$.

Der Algorithmus in dieser Form würde aber o.B.d.A. nur für a die korrekte Erweiterung von δ durchführen. Zweitens ist die Erweiterung von δ selbst fehlerhaft, durch diese Konstruktion wird eine Erweiterung explizit dann *nicht* durchgeführt, falls $(t_j, a, z') \notin \delta$ ist – beachte daß Ahonen von z' nicht verlangt, daß es in K bzw. $Z \backslash K$ oder überhaupt in Z ist, denn wir betrachten δ ja nicht auf Z eingeschränkt – t_j ist aber nach Voraussetzung ein Tor von K, so daß sichergestellt werden muß, daß auch $(t_j, a, z) \in \delta$ ist. Auch die Iteration in Zeile (5') über „jedes geordnete Paar (t_i, t_j)“ führt nicht notwendig dazu.

Diese Konstruktion bleibt daher absolut unklar, vielleicht hat Ahonen auch nur die Zeile:

(10') sonst $\delta(t_j, a) := \{z\}$

vergessen, welche natürlich redundant i.d.S. ist, daß wir das ganze Konstrukt einfach durch:

(7”) dann
(8”) $\delta := \delta \cup \{(t_j, a, z)\}$

ersetzen könnten.

In Zeile (4) von dem Algorithmus *ErzeugeNicht-1-Mehrdeutigkeit* wird ein Symbol $a \in \Sigma$ und ein Zustand $f(a)$ gewählt, für das dann die M-Konsistenz hergestellt wird. Hierbei wird der Automat immer verallgemeinert, da entweder Zustandsübergänge hinzugefügt oder Zustände verschmolzen werden.

Beides kann weitere Zustandverschmelzungen erzwingen, weil der Automat deterministisch gemacht werden muß. Es ist jedoch sinnvoll, an dieser Stelle so wenig wie möglich zu generalisieren, da die Automatentransformation hier ja rein „technische“ Gründe hat und das eigentliche zielgerichtete Generalisieren schon stattfand.

Folgende Heuristiken werden von Ahonen zur Wahl von a vorgeschlagen: Wir wählen a und $f(a)$ so, daß die Zahl der Endzustände mit einer a-Transition nach $f(a)$ maximal wird, denn nur Endzustände, die keine a-Transition nach $f(a)$ haben werden modifiziert. Ahonen schlägt als Alternative vor, ein Symbol a zu nehmen, für das möglichst viele Endzustände eine Transition haben, aber zu verschiedenen Zielzuständen.

Letztere Heuristik halte ich für nicht so geeignet, da sie sehr viele Zustandsverschmelzungen nach sich zieht. Die erste Heuristik könnte man noch dahingehend modifizieren, daß wir a und $f(a)$ so wählen, daß $\{q | q \text{ } ist \text{ } Endzustand \text{ } und \text{ } ((q, a, f(a)) \in \delta \text{ } oder \text{ } \delta(q, a) = \emptyset)\}$ maximal wird, denn das ist dann die Maximalzahl von Zuständen, für die keine Verschmelzungen ausgeführt werden.

Natürlich könnte man ein konkretes Optimum definieren, z.B. unter allen Wahlen von a und $f(a)$ in allen Schritten so zu verfahren, daß nach Ablauf des gesamten Algorithmus z.B. ein Automat herauskommt, der eine maximal mögliche Zustandszahl besitzt. Andererseits könnte eine solche Optimierung es unter Umständen erfordern, alle Möglichkeiten auch tatsächlich durchzurechnen, was aus Effizienzgründen dann doch keine optimale Lösung ist.

Hier ein Beispiel aus [4] für die Arbeit des Algorithmus:
Abb. 3.10 (a) zeigt einen DEA, der nicht-1-mehrdeutig ist und aus den Orbits $\{1\}, \{2\}, \{3, 4, 5\}, \{6\}$ und $\{7\}$ besteht. Offensichtlich gilt beim Ablauf des Algorithmus $S = \emptyset$, weiterhin muß der Automat durch *ErzeugeOrbitEigenschaft* modifiziert werden, denn 3 und 4 sind Tore des Orbits $\{3, 4, 5\}$, so daß auch 4 ein Endzustand sein muß und weiterhin müssen wir $(3, R, 6)$ und $(4, R, 7)$ in δ aufnehmen.

Hiernach wird *ErzeugeNicht-1-Mehrdeutigkeit* rekursiv für jeden Orbit einmal aufgerufen, für den Orbit $\{3, 4, 5\}$ wählen wir 3 als Startzustand. Die Aufrufe auf den Orbits $\{1\}$, $\{6\}$ und $\{7\}$ brechen sofort ab, da diese Orbits trivial sind, für den Orbit $\{2\}$ wird $S = \{H\}$ berechnet, M_S ist dann auch der triviale Orbit. Interessant ist vor allem der Orbit $\{3, 4, 5\}$. Die Zustände 3 und 4 sind Endzustände, es gibt aber keine M-konsistenten Symbole, an dieser Stelle wird $(4, T, 5)$ in δ aufgenommen, damit der Orbitautomat $\{T\}$-konsistent ist, das Ergebnis zeigt Abb. 3.10 (b) und der $\{T\}$-Schnitt ist in (c) dargestellt.

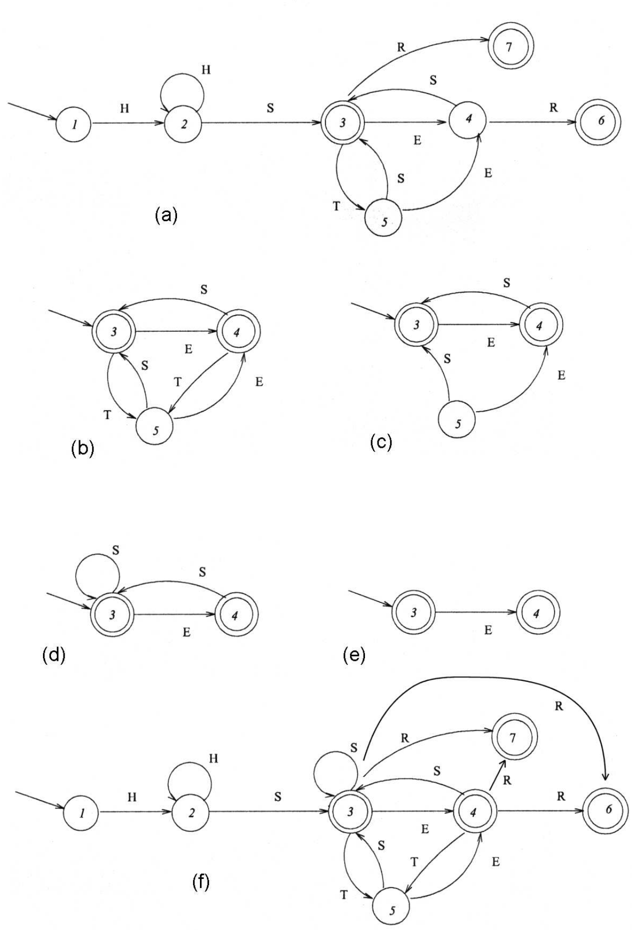


Abbildung 3.10: Erzeugung der Nicht-1-Mehrdeutigkeit (Abb. (a)-(e) aus [4])

Offenbar ist der Automat in die Orbits $\{5\}$ und $\{3, 4\}$ zerfallen, wobei $\{5\}$ trivial ist, also der rekursive Aufruf auf diesem Orbit gleich abbricht. Beim Aufruf auf $\{3, 4\}$ müssen wir wieder ein M-konsistentes Symbol erzeugen, hier S. Das Ergebnis zeigt Abb. 3.10 (d) und in (e) ist der entsprechende $\{S\}$-Schnitt dargestellt, wobei hier zwei triviale Orbits entstehen. Das Gesamtergebnis ist in (f) dargestellt.

Der Algorithmus wird in [4] aber überhaupt nicht bewiesen, bzw. nicht korrekt bewiesen und enthält ja auch den eben genannten Fehler. Es ist ja auch nicht so, daß Behauptungen, die nicht bewiesen oder gründlich durchdacht sind, auch unwahr sind. Daß der Algorithmus funktioniert, liegt an seiner „Nähe" zum Entscheidungsalgorithmus von Brüggemann-Klein und Wood, daß Ahonen den Fehler in *ErzeugeOrbitEigenschaft* nicht bemerkt hat, liegt vermutlich an dem geringen Ausmaß des Testens bzw. der Verifikation. Der Beweis für den Algorithmus soll nun an dieser Stelle nachgeholt werden.

Ahonen vergißt auch zu erwähnen, daß der Entscheidungsalgorithmus von Brüggemann-Klein und Wood nur für minimale DEAs gilt, womit schon das angegebene Beispiel in [4] falsch ist. Allerdings zeigen Brüggemann-Klein und Wood auch, daß wenn ein Automat die Orbit-Eigenschaft besitzt, diese beim Minimieren nicht verloren geht und wenn alle Orbitsprachen eines Automaten nicht-1-mehrdeutig sind, dieses auch für alle Orbitsprachen nach der Minimierung gilt. Hieraus folgt, daß wenn ein nicht-minimaler DEA diese Eigenschaften hat, er auch nicht-1-mehrdeutig ist, denn diese Eigenschaften gehen bei der Minimierung nicht verloren und die Sprache des Automaten ändert sich beim Minimieren auch nicht.

Zwar bemerkt Ahonen richtig, daß die Anwendung des Algorithmus auf einem Orbit-Automaten Nicht-Determinismus im enthaltenen Automaten verursachen kann, übersieht aber, daß das auch schon beim Erzeugen von S in Zeilen (4-9) geschehen kann und beim Nutzen der Funktion *ErzeugeOrbitEigenschaft*.

Die Idee von Ahonen ist nun, nach Ablauf des Algorithmus *ErzeugeNicht-1-Mehrdeutigkeit* im Falle von Nicht-Determinismus Zustände so zu verschmelzen, daß der Automat deterministisch wird und den Algorithmus *ErzeugeNicht-1-Mehrdeutigkeit* erneut auszuführen. Das wird wiederholt, bis sich der Automat nicht mehr ändert.

Das ist auch sinnvoll und notwendig, denn der Übergang zu einem äquivalenten DEA mit der üblichen Konstruktion (vgl. Hopcroft Ullmann [20], Potenzmengenautomat) vergrößert die Zustandsmenge exponentiell, wir möchten die Zustandsmenge aber monoton bis zum Abbruch des Algorithmus verkleinern.

Algorithmus verschmelze_zu_DEA
Eingabe: NEA $M = (Z, \Sigma, \delta, z_0, E)$
Ausgabe: DEA $M' = (Z', \Sigma', \delta', z_0', E')$

Solange existiert $(z_i, a, z_j), (z_i, a, z_h) \in \delta$ mit $z_j \neq z_h$ wiederhole
  Verschmelze z_j und z_h zu z_k

Bemerkung: Mit jedem Schritt nimmt die Zustandsanzahl um eins ab, also terminiert der Algorithmus irgendwann, da dann auch gilt: es existieren keine $(z_i, a, z_j), (z_i, a, z_h) \in \delta$ mit $z_j \neq z_h$ ist der resultierende Automat auch deterministisch und Zustandsverschmelzung war so definiert, daß der resultierende Automat auch eine Obermenge der Sprache des ursprünglichen Automaten erkennt.

Klar ist, wenn wir den Automaten M nach einem Durchlauf mit $verschmelze\_zu\_DEA$ deterministisch machen und der nächste Durchlauf den Automaten M nicht verändert hat, daß er dann nicht-1-mehrdeutig ist:

Wenn der Automat nicht verändert wurde, hat der Algorithmus entweder bei Zeile (1) terminiert, die Bedingung in Zeile (3) muß als „falsch“ ausgewertet worden sein, denn ansonsten wird der Automat geändert, die Funktion *ErzeugeOrbitEigenschaft* hat den Automaten nicht geändert, also hat er die Orbiteigenschaft, beachte hierbei, daß es keine Möglichkeit gibt, daß sich mehrere Änderungen vollständig kompensieren.

Für jeden Orbit K von M_S gilt für ein $x \in K$, daß der rekursive Aufruf den Automaten nicht geändert hat, der rekursive Aufruf erfolgt niemals auf dem ursprünglichen und bis jetzt unveränderten M selbst, denn sonst würde der Algorithmus überhaupt nicht terminieren, womit man per Induktion über die Automatenstruktur zeigen kann, daß M wirklich nicht-1-mehrdeutig ist unter Berücksichtigung der Aussage von Brüggemann-Klein und Wood zum Erhaltenbleiben der benötigten Eigenschaften beim Minimieren.

Lemma 14 *Der Algorithmus terminiert auf jedem DEA M.*

Beweis:
Die Rekursion erfolgt irgendwann immer mit einem Automaten, der weniger Zustände als M hat, entweder hat M mehrere Orbits, dann ist klar, daß jeder Orbitautomat weniger Zustände als M hat oder M hat nur einen einzigen Orbit.

Bei einem einzigen Orbit erfolgt entweder der Abbruch in Zeile (1) oder es wird der S-Schnitt M_S berechnet, sollte M_S immer noch nur genau einen Orbit besitzen, wird in der nächsten

Rekursion ein weitere S-Schnitt (mit o.B.d.A. $\{a\}$) durchgeführt usw., irgendwann muß M_S entweder trivial sein oder in mehr als einen Orbit „zerfallen“.

Beachte, daß wenn *ErzeugeOrbitEigenschaft* einmal auf M_S gelaufen ist und wir einen weiteren Schnitt machen, also $M_{S\cup\{a\}}$ erzeugen, dieser noch die Orbit-Eigenschaft hat, wenn er nur einen einzigen Orbit besitzt, denn die Tore sind gleich geblieben (und allesamt Endzustände) und Zustände außerhalb des einzigen Orbits gibt es ja nicht.
q.e.d.

Bemerkung:
Ahonen begründet ebenfalls die Terminierung des Automaten, allerdings falsch: Der Automat terminiert, weil entweder Zustände verschmolzen werden oder Zustandsübergänge hinzugefügt werden, wir natürlich aber Schranken dadurch gegeben haben, daß wir so entweder zu einem Automaten mit nur einem Zustand oder einem Automaten mit allen möglichen Zustandsübergängen gelangen, welche jeweils nicht-1-mehrdeutig sind.

Beachtet man oben angeführte Beobachtung, daß nicht-1-mehrdeutige DEAs Fixpunkte von *ErzeugeNicht-1-Mehrdeutigkeit* sind, dann stimmt es einfach nicht, daß immer Zustände verschmolzen werden oder Zustandsübergänge hinzu kommen.

Zusammenfassung:

Wir rufen den Algorithmus mit einem DEA M auf und führen auf dem Ergebnis-NEA M' $verschmelze\_zu\_DEA()$ aus und erhalten den DEA M''. Wie oben diskutiert, terminiert der Algorithmus auf M und irgendwann erhalten wir einen Fixpunkt $M = M''$, welcher nicht-1-mehrdeutig ist. Für diesen bilden wir den Minimalautomaten und können dann einen nicht-1-mehrdeutigen regulären Ausdruck ableiten:

Ableitung des regulären Ausdruckes

Aus den jetzt vorliegenden nicht-1-mehrdeutigen *minimalen* Automaten M wollen wir nun einen nicht-1-mehrdeutigen regulären Ausdruck ableiten, dieser wird rekursiv über Teilautomaten definiert.

Dazu zunächst wieder die Unterscheidung, ob $M = (Z, \Sigma, \delta, z_0, E)$ aus einem oder aus mehr als einem Orbit besteht:

Fall 1: M hat mehr als einen Orbit

Brüggemann-Klein und Wood zeigen ([12]; Theorem E), daß wenn M minimal ist, $L(M)$ nicht-1-mehrdeutig ist gdw. M hat die Orbiteigenschaft und alle Orbitsprachen sind nicht-1-mehrdeutig.

Ist $z_i \in Z$ so sei M^{z_i} der Automat, der aus M so hervorgeht, daß z_i der Startzustand ist und alles andere auf Zustände eingeschränkt wird, die in M von z_i aus erreichbar sind. Offenbar ist M^{z_i} minimal, wenn M es ist, also gibt es einen entsprechenden nicht-1-mehrdeutigen regulären Ausdruck E^{z_i}.

E_{z_0} sei der entsprechende nicht-1-mehrdeutige reguläre Ausdruck für M_{z_0}, für die Tore von $O(z_0)$ seien $a_1, \ldots, a_n$ die Symbole, mit denen $O(z_0)$ verlassen wird und $z_1, \ldots, z_n$ die entsprechenden Zustände zu denen diese Transitionen führen.

Sind die Tore von $O(z_0)$ keine Endzustände, so ergibt sich E wie folgt:

$E := E_{z_0}(a_1 E^{z_1} + a_2 E^{z_2} + \cdots + a_n E^{z_n})$

Anderenfalls

$E := E_{z_0}(a_1 E^{z_1} + a_2 E^{z_2} + \cdots + a_n E^{z_n})?$

Klarerweise ist für $a \in \Sigma$ und einen nicht-1-mehrdeutigen regulären Ausdruck E auch aE nicht-1-mehrdeutig. Dazu betrachten wir Lemma 1: Es gilt: $First(a_1 E')$ ist gleich $\{a_1\}$, also einelementig, wir müssen nur $Follow(a_1E')$ betrachten. Hier gilt: $Follow(a_1E') = First(E')$ und da E nicht-1-mehrdeutig ist, gilt für $x, y \in First(E')$: $um(x) \neq um(y)$. Also ist aE nicht-1-mehrdeutig.

Analog sehen wir, daß für $a_iE^{z_i}$ und $a_jE^{z_j}$ nicht-1-mehrdeutig auch $a_iE^{z_i} + a_jE^{z_j}$ nicht-1-mehrdeutig sind, da $a_i \neq a_j$ für $i \neq j$ gilt.

Ebenso folgt aus Lemma 1, daß E nicht-1-mehrdeutig ist gdw. $E?$ ist nicht-1-mehrdeutig.

Bilden wir FG aus zwei nicht-1-mehrdeutigen Ausdrücken F und G, so gilt:

$$First(F'G') = \begin{cases} First(F') & \epsilon \notin L(F') \\ First(F') \cup First(G') & \epsilon \in L(F') \end{cases}$$

o.B.d.A. $Sym(F') \cap Sym(G') = \emptyset$

Ist $\epsilon \notin L(F')$ so ist die erste Bedingung aus Lemma 1 erfüllt, denn F ist ja nicht-1-mehrdeutig, anderenfalls müssen wir $First(F) \cap First(G) = \emptyset$ fordern.

In beiden Fällen ergibt sich für die zweite Bedingung aus Lemma 1 und $z \in last(F')$: $Follow(F'G', z) = Follow(F', z) \cup First(G')$. Andere $Follow$-Mengen ändern sich nicht, es ist also hinreichend zu fordern:
$\forall z \in Last(F') : um(Follow(F', z)) \cap First(G) = \emptyset$

Da M die Orbiteigenschaft erfüllt, hat M_{z_0} keine a_i-Transitionen von einem Endzustand (beachte, daß die Endzustände Tore in M waren). Angenommen, für ein $z \in Last(E'_{z_0})$ ist ein $a \in Follow(E'_{z_0}, z)$ mit $um(a) = a_i$ für ein i, weil es ein Wort $uzav \in L(E'_{z_0})$ gibt, so muß das Wort $um(uzav)$ vom Automaten M erkannt werden.

Nach der Diskussion bei der Begründung von Lemma 1 gilt auch, daß $uz \in L(E'_{z_0})$ ist, also $um(uz)$ M in einen Endzustand überführt. Somit haben wir einen Widerspruch zur Annahme, da es keine a_i-Transition aus einem Endzustand gibt und M deterministisch ist.

Für den Fall, daß $\epsilon \in L(M_{z_0})$, muß gelten, daß der Startzustand ein Endzustand ist. Da die Endzustände Tore in M waren, gibt es wie oben bereits gesagt, aus diesen keine a_i-Transitionen, also muß $a_i \notin First(E_{z_0})$ gelten. Offenbar ist $First(a_1E^{z_1} + a_2E^{z_2} + \cdots + a_nE^{z_n}) = \{a_1, \ldots, a_n\}$, also $First(F) \cap First(G) = \emptyset$.

Fall 2: M hat nur einen einzigen Orbit

Ist der einzige Orbit von M trivial, so liegt der nicht-1-mehrdeutige Ausdruck auf der Hand, denn die Orbitsprache ist $\{\epsilon\}$ und der entsprechende reguläre Ausdruck ϵ ist nicht-1-mehrdeutig, was direkt aus der Definition folgt.

Brüggemann-Klein und Wood zeigen, daß wenn M ein DEA ohne überflüssige Zustände und S eine Menge M-konsistenter Symbole ist, M genau dann nicht-1-mehrdeutig ist, wenn für alle $z \in Z$ gilt: $L(M_S^q)$ ist nicht-1-mehrdeutig.

Aus der Definition des S-Schnittes M_S und der Definition der M-Konsistenz, folgt direkt, daß gelten muß: $L(M) = L(M_S)(\bigcup_{a \in S}[aL(M_S^{f(a)})])^*$

$\subseteq$:
Erkennen wir $w = b_1 ... b_i ... b_n \in L(M)$ und gelangen wir nach b_i o.B.d.A. zum ersten Mal in einen Endzustand z_e, so ist $b_{i+1} \in S$ oder nicht. Im zweiten Fall können wir die b_{i+1}-Transition wie bisher weiter verfolgen. Falls aber $b_{i+1} \in S$, so ist die b_{i+1}-Transition von z_e aus entfernt worden. Dann haben wir aber $b_{i+1}L(M_S^{f(b_{i+1})})$ zur Ableitung zu Verfügung. Diese Argumentation ist bei jedem Endzustand gleich, w wird also auch auf der rechten Seite erkannt bzw. abgeleitet.

$\supseteq$:
Dadurch, daß S nur M-konsistente Symbole enthält wird klar, daß wir mit der rechten Seite nicht mehr Worte ableiten können. Jeder Ableitung eines Wortes auf der rechten Seite können wir direkt eine Erkennung auf der linken Seite zuordnen.
q.e.d.

Durch die Ausführung des S-Schnittes und der bekannten Aussage, daß hier $S \neq \emptyset$ gelten muß, muß M_S entweder trivial sein oder in mehr als einen Orbit zerfallen, letztenendes gelangen wir immer zu trivialen Orbits.

Beispiel:
Wir wollen den Ausdruck des Automaten aus Abb. 3.10 (f) auf Seite 52 ableiten:

- wir sehen gleich, daß die Orbitsprachen von $\{1\}$, $\{6\}$ und $\{7\}$ trivial sind
- zunächst haben wir mehrere Orbits, d.h.
  $L(M) := E_1(H\,E^2)$ mit $E_1 = \epsilon$
- auch bei E^2 haben wir mehrere Orbits, also
  $E^2 := E_2(S\,E^3)$, wobei offenbar $E_2 = H^*$ gilt
- ebenso haben wir bei E^3 mehrere Orbits, also
  $E^3 := E_3(R\,E^6 + R\,E^7)?$, wobei $E^6 = E^7 = \epsilon$
- bei E_3 (d.h. der Orbit $\{3, 4, 5\}$) müssen wir den $\{S, T\}$-Schnitt durchführen, wonach wir die trivialen Orbits $\{3\}$, $\{4\}$ und $\{5\}$ haben, mit $E^3 = E?$; also:
  $E_3 := E?(S\,E? + T\,(E + S\,E?))^*$
- zusammen:
  $L(M) := H\,H^*(S\,E?(S\,E? + T\,(E + S\,E?))^*\,R?)$

Brüggemann-Klein und Wood zeigen, daß dieses Vorgehen exponentiell in der Größe des Automaten ist.

Weiterhin können auf dem Ergebnisausdruck noch Umformungen zur Vereinfachung sinnvoll sein, hierbei ist natürlich zu beachten, daß durch diese die Nicht-1-Mehrdeutigkeit nicht verletzt wird.

Weiteres Vorgehen

Konkret benutzt Ahonen k-Gramme für die Implementation, auf welche an dieser Stelle nicht eingegangen wird, welche aber eine sehr effiziente Implementation für k-kontextuelle Sprachen ermöglichen, vgl. hierzu auch Kapitel 6.

Weiterhin diskutiert Ahonen noch Verbesserungen auf den erhaltenen Ergebnissen, erwähnenswert sind hier vor allem die *Isolierung von Modellgruppen* und die *Inklusionen*.

Aufgrund gewisser Maße bzgl. des benachbarten Auftretens von Elementen wird im ersten Fall versucht, durch das Definieren von *parameter entities* [7] die Lesbarkeit zu erhöhen, im zweiten Fall wird versucht, Dinge zu erkennen, die an beliebiger Stelle stehen können, was aber noch nicht erkannt wurde. Dies sind bei dokumenten-orientierten XML-Dokumenten z.B. Fußnoten und Abbildungen.

[7] Ahonen schlägt als Möglichkeit auch vor, neue Elemente statt parameter entities zu definieren, und dementsprechend die Dokumente zu migrieren, was wir aber hier nicht wollen.

3.2.4 Fernau

Fernau präsentiert in [16] einen generischen Algorithmus der direkt aus den theoretischen Betrachtungen zum Lernen von Sprachen abgeleitet ist. Auch hier werden Attribute und deren Typen vernachlässigt, die Nicht-1-Mehrdeutigkeit wird nicht diskutiert.

Die Formalisierungen in [16] unterscheiden sich syntaktisch stark von den bisher gebrachten, XML-Dokumente werden nicht als Baum aufgefaßt, es werden XML-Grammatiken definiert, deren Terminale praktisch den Start- und Endtags entsprechen usw. Den Verzicht auf die Baumdarstellung erkauft man sich mit einer Fülle von Definitionen, welche aber nichts zum eigentlichen Problem beitragen. Da die Notwendigkeit nicht besteht, werden die meisten verwendeten Formalisierungen hier nicht wiedergegeben, die verwendeten XML-Grammatiken machen im Prinzip nicht wesentlich mehr, als die korrekte Start- und Endtagklammerung sicherzustellen.

Fernau behauptet, die Methode von Ahonen zur ausschließlichen Ableitung von rechten Seiten von Grammatikregeln – wenn man eine DTD als eCFG auffaßt – sei nicht notwendig hinreichend für eine korrekte XML-Grammatik. Diese Behauptung ist nicht korrekt, das Problem der Bestimmung einer DTD – wenn man von Attributen absieht – ist aber auf die Teilprobleme der Bestimmung von Inhaltsmodellen für jeden Elementtyp zurückführbar.

Der eigentliche Algorithmus von Fernau ist in dem Sinne generisch, daß wir ihm noch eine Unterscheidungsfunktion f übergeben müssen.

Definition:
Sei F eine beliebige endliche Menge: eine Funktion $f : \Sigma^* \rightarrow F$ heißt **Unterscheidungsfunktion** :gdw.
$\forall u, v, w \in \Sigma^* : f(u) = f(v) \Rightarrow f(uw) = f(vw)$

Definition:
Für eine konkrete Unterscheidungfunktion f heißt eine Sprache L **f-unterscheidbar** :gdw.
$\forall u, v, w, z \in \Sigma^* : f(w) = f(z)$ und $wu, wv \in L \Rightarrow (zu \in L \Leftrightarrow zv \in L)$
Die Familie aller f-unterscheidbarer Sprachen über Σ wird mit (f, Σ)-DL[8] bezeichnet.

Bekanntermaßen ist die Klasse der regulären Sprachen nicht identifizierbar, d.h. wir können keinen Algorithmus angeben, der uns nur unter Zugrundlegung positiver Beispiele, also Worte, die wirklich in der Sprache sind, einen endlichen Automaten bzw. einen regulären Ausdruck liefert, der exakt die gesuchte Sprache liefert. Damit gilt natürlich dieselbe Aussage für die Klasse der XML-Sprachen, da wir für diese eine Menge regulärer Ausdrücke ableiten müssen.

Bestimmte Teilmengen der Klasse der regulären Sprache sind jedoch identifizierbar, insbesondere gilt das für (f, Σ)-DL, was Fernau zeigt.

Die Unterscheidungsfunktion f ist also das Generalisierungskriterium, welches gleichzeitig die Identifizierbarkeit der Sprache sicherstellt.

Ein DEA $M = (Z, \Sigma, \delta, z_0, E)$ ist **f-unterscheidbar** :gdw.
Seien $v, w \in \Sigma^* : \delta(z_0, v) = \delta(z_0, w) = z' \Rightarrow f(v) = f(w)$
und
Seien $p, q \in Z$ mit $p \neq q$: $(p, q \in E$ oder $\exists a \in \Sigma$ mit $\delta(p, a) = \delta(q, a) = z') \Rightarrow f(p) \neq f(q)$

Es gilt: L ist genau dann f-unterscheidbar, wenn ein DEA M existiert mit $L(M) = L$ und M ist f-unterscheidbar.
Da f für alle Worte v, w, die z_0 nach z' überführen, denselben Wert hat, können wir f auch über Z definieren: Ist $\delta(z_0, w) = z'$, dann $f(z') := f(w)$

Die Frage, wie man f für unser Szenario sinnvoll wählen kann wird von Fernau in [16] allerdings nicht diskutiert, in [15] wird aber die Approximation von beliebigen regulären Sprachen mit f-unterscheidbaren Sprachen diskutiert. In unserem Fall ist I (bedingt durch die Endlichkeit) auch

[8] DL steht für *distingushable languages*

immer charakteristisch für eine f-unterscheidbare Sprache.

Das Vorgehen ist wie folgt:
Sei $I = \{w_1, ..., w_k\}$ wie auch in den anderen Ansätzen die Menge der Worte, für welche das Inhaltsmodell abgeleitet werden soll.
Es wird nun ein NEA M' definiert, der genau I erkennt. Mit Hilfe der Unterscheidungsfunktion f wird auf den Zuständen von M' eine Äquivalenzrelation $\equiv_f$ definiert und danach jeweils alle Zustände einer Klasse zu einem einzigen Zustand verschmolzen. Fernau hat gezeigt, daß das Ergebnis M ein f-unterscheidbarer DEA ist, dessen Sprache $L(M)$ die kleinste f-unterscheidbare Sprache ist, welche I enthält, womit gem. der Diskussion in Abschnitt 3.2 dieser Algorithmus die Sprache in den Grenzen gem. Gold identifiziert.

Ausgehend von I erfolgt die Konstruktion des Skelett-Automaten (engl. skeletal automaton) wie folgt, hierbei sei $\forall i \in \{1...k\} : w_i = a_{i1}...a_{in_i}$.

$$\begin{array}{rcl} M' & := & (Z, \Sigma, \delta, Z_0, E) \\ \text{mit} & & \\ Z & := & \{z_{i,j} \,|\, 1 \leq i \leq k, 1 \leq j \leq n_i + 1\} \\ \delta & := & \{(z_{i,j}, a_{i,j}, z_{i,j+1}) \,|\, 1 \leq i \leq k, 1 \leq j \leq n_i\} \\ Z_0 & := & \{z_{i,1} \,|\, 1 \leq i \leq k\} \\ E & := & \{z_{i,n_i+1} \,|\, 1 \leq i \leq k\} \end{array}$$

M' ist ein NEA mit mehreren Startzuständen, weiterhin gilt aufgrund der Konstruktion, daß das einzige Wort, mit dem $z_{i,j}$ erreichbar ist, genau $a_{i,1} \ldots a_{i,j-1}$ ist (bei Fernau bezeichnet als *head string;* $HS(z_{i,j})$) und das einzige Wort, mit dem wir ausgehend von $z_{i,j}$ einen Endzustand (nämlich z_{i,n_i+1}) erreichen $a_{i,j} \ldots a_{i,n_i}$ ist (bei Fernau bezeichnet als *frontier string;* $FS(z_{i,j})$).

Demnach ist die Erweiterung von f auf Z mittels $f(z) := f(HS(z))$ wohldefiniert, es sei $\equiv_f := (\rightleftharpoons_f)^+$, also die transitive Hülle von $\rightleftharpoons_f$, welche wiederrum eine reflexive und symmetrische Relation auf den Zuständen darstellt mit: $z_{i,j} \rightleftharpoons_f z_{k,l}$:gdw. $(HS(z_{i,j}) = HS(z_{k,l})\, oder\, FS(z_{i,j}) = FS(z_{k,l}))\, und\, f(z_{i,j}) = f(z_{k,l})$

In [15] zeigt Fernau, daß der von ihm erzeugte Automat M' für jede Wahl von I und einer Unterscheidungsfunktion f isomorph zum Ergebnisautomaten des Algorithmus (1) mit dem Verschmelzungskriterium für f ist, welches sich aus der Definition des f-unterscheidbaren Automaten ergibt.

Bemerkung:
Benutzen wir $f(w) := \sigma_k(w)$ wobei $\sigma_k(w)$ der k-Suffix von w ist für $|w| \geq k$ bzw. $\sigma_k(w) := w$ für $|w| < k$, so erhalten wir genau die Klasse der k-umkehrbaren Sprachen.

Beweis:
$\Rightarrow$: Ist L eine k-umkehrbare Sprache, so existiert natürlich ein k-umkehrbarer DEA M mit $L(M) = M$. Wir betrachten die Definition der f-unterscheidbaren Sprachen: Unter der Voraussetzung $|w| \geq k, \sigma_k(w) = \sigma_k(z)$ und $wv, wu \in L$ und o.B.d.A. $zv \in L$ überführen sowohl w als auch z den Zustand z_0 in denselben Zustand q, was aus der Eigenschaft der k-Umkehrbarkeit folgt, denn w und z haben ja denselben k-Suffix, die gegenteilige Annahme führt sofort zum Widerspruch[9], somit gilt auch $zw \in L$
$\Leftarrow$: L sei σ_k-unterscheidbar mit $zv, wv \in L$ und $\sigma_k(z) = \sigma_k(w)$. Aus der Definition der f-unterscheidbaren Sprachen folgt dann unmittelbar $T_L(z) = T_L(w)$. Das ist aber genau die Definition der k-umkehrbaren Sprachen (vgl. Seite 35).

[9] Mit der Fallunterscheidung $v = \epsilon$ und $v \neq \epsilon$

3.2.5 XTRACT

Der Ansatz des XTRACT-Systems [18] und [17] unterscheidet sich von den anderen in der Herangehensweise. Das XTRACT-System arbeitet nicht auf XMLtrees, Graphen oder endlichen Automaten sondern direkt auf der Menge der Ausdrücke, d.h. deren textueller Repräsentation. Weiterhin wird im Gegensatz zu den anderen nicht zielgerichtet auf eine DTD hin gearbeitet, sondern vielmehr ein Suchraum aufgespannt.

Zum Einschränken des Suchraumes werden Heuristiken benutzt, für die Suche Approximationsalgorithmen, da die Optimumsermittlung NP-hart ist. Die Zielfunktion ist hierbei Ansätzen aus der Informationstheorie entnommen und zwar wird das MDL (Minimum Descriptor Length)-Prinzip zu Grunde gelegt. Informal ist nach dem MDL-Prinzip diejenige DTD die optimale, für die gilt, daß für ihre eigene Kodierung und die Kodierung aller XML-Dokumente der Menge die wenigsten Bits benötigt werden.

Das Vorgehen nach dem MDL-Prinzip wird von den Autoren wie folgt motiviert:
Die zu erzeugende DTD soll *kompakt* und *präzise* sein, kompakt bedeutet hier von geringer Größe und präzise bedeutet, daß die DTD möglichst wenige Worte abdeckt, die nicht in den Eingabedokumenten enthalten sind.

Die Heuristik hierfür ist, daß die Kompaktheit einer DTD mit der Anzahl der zur Kodierung der DTD selbst benötigten Bits korreliert und die Präzision mit der Anzahl der Bits, die zur Kodierung der Dokumente selbst benötigt werden.

Gibt die DTD ziemlich exakt die Beispieldokumente wieder (quasi als Aufzählung), so brauchen wir wenig Bits für die Kodierung der Dokumente (Listennummer in der Aufzählung) aber viele Bits für die Kodierung der DTD selbst, da diese sehr groß wird. Ist andererseits die DTD sehr stark generalisert (etwa Inhaltsmodell der Art $(a_1|...|a_n)^*$), so benötigen wir für die DTD selbst wenig Bits, aber entsprechend viele für die Dokumente, da wir sämtliche Strukturinformationen in den Dokumenten kodieren müssen.

Das MDL-Prinzip wurde in der aktuellen Form von Rissanen 1982 eingeführt und wird hier aus [19] wiedergegeben:

Die beste Theorie um eine Datenmenge zu erklären ist diejenige, welche die Summe minimiert, welche sich aus der Länge (in Bits) der Theoriebeschreibung und der Länge (in Bits) der Daten, kodiert mit genau dieser Theorie, ergibt.

Im Fall der Ableitung von Grammatiken entspricht die Theorie einer Grammatik und die Datenmenge der Beispielwortmenge.

Vor der Darstellung des Ansatzes im Detail sei schon vorweggenommen, daß der Aspekt der Nicht-1-Mehrdeutigkeit nicht beachtet wird. Eine Rückfrage bei einem der Autoren ergab, daß dieser Punkt wohl schlichtweg vergessen wurde, ebenso wurde vergessen, in die in [18] aufgeführte Test-DTD-Menge Produktionen aufzunehmen, welche ein Symbol mehr als einmal enthalten. De facto sind reguläre Ausdrücke, die kein Symbol mehrfach beeinhalten immer nicht-1-mehrdeutig.

Abhilfe würde natürlich der Algorithmus von Ahonen [4] schaffen, der einen DEA in einen nicht-1-mehrdeutigen DEA umformt. Andererseits haben wir bei XTRACT nach dem Ablauf des Algorithmus schon einen fertigen und i.d.R. kompakten und gut lesbaren regulären Ausdruck, so daß mutzumaßen ist, daß wir viele Auswirkungen des MDL-Prinzipes zunichte machen, wenn wir daraus wieder den Automaten bilden, diesen manipulieren und wieder einen regulären Ausdruck ableiten, welcher zwar nicht-1-mehrdeutig, nicht aber gerade kompakt ist.

Besser wäre es, den regulären Ergebnisausdruck von XTRACT auf Nicht-1-Mehrdeutigkeit zu testen und falls nötig gewisse Regeln anzuwenden, die eine Generalisierung durchführen, dabei aber danach streben, gewisse Eigenschaften (wie z.B. die Kompaktheit) zu erhalten. Der Test selbst kann nach Lemma 1 erfolgen oder über die Konstruktion eines Glushkovautomaten, welcher ein spezieller EA ist der zu einem regulären Ausdruck korrespondiert und genau dann deterministisch ist, wenn der entsprechende reguläre Ausdruck nicht-1-mehrdeutig ist. Der Glushkovautomat wird u.a. bei Brüggemann-Klein und Wood in [12] beschrieben.

Es ist natürlich nicht so einfach – sonst wären viele andere Arbeiten gar nicht nötig gewesen – einen Satz von Regeln auf syntaktischer Ebene anzugeben, welcher einen regulären Ausdruck zu einem nicht-1-mehrdeutigen Ausdruck generalisiert. Die regulären Ausdrücke bei XTRACT sind allerdings nicht beliebig, also evt. wäre es in diesem Fall möglich, eine entsprechenden Vorgehensweise zu konstruieren.

XTRACT behandelt nicht die Erkennung der Attributtypen, ebensowenig werden gemischte Inhaltsmodelle betrachtet, was aber o.B.d.A. für die Generierung von DTD auch keine wirkliche theoretische Relevanz hat, da in diesem Fall sowieso nur Inhaltsmodelle der oben bereits erwähnten Formen erlaubt sind.

Die Generierung des Inhaltsmodelles erfolgt für jeden Elementtyp separat, im folgenden sei deshalb stets vorausgesetzt, daß es um einen bestimmten festen Elementtyp e geht. Zunächst wird die Menge I erstellt, die alle Sequenzen enthält, die e in irgendeinem der XML-Dokumente als Inhalt hat. Der Algorithmus funktioniert schrittweise
(siehe Beispiel aus [18]: $I := \{ab, abab, ac, ad, bc, bd, bbd, bbbbe\}$):

- der Generalisierungsschritt, die Menge I wird um Sequenzen angereichert, die den Kleene-Stern (*) enthalten; diese Sequenzen werden unter Anwendung bestimmter Heuristken aus den ursprünglichen Sequenzen erzeugt.

  im Beispiel: zu I werden $(ab)*, (a|b)*, b*d, b*e$ hinzugefügt.

- Danach findet der Faktorisierungsschritt statt, der die Menge I um weitere Sequenzen anreichert, hierfür werden Sequenzen aus I zusammengefaßt, z.B. wird die Sequenz $a(b|c)$ zu I hinzugefügt, wenn in I die Sequenzen ab und ac enthalten sind. Auch hierbei werden Heuristiken angewandt, um nur „sinnvolle“ Faktorisierungen anzuwenden, damit der Suchraum nicht zu groß wird.

  im Beispiel: zu I werden $(a|b)(c|d), b*(d|e)$ hinzugefügt

- Der abschließende letzte Schritt, der MDL-Schritt, sucht nun eine Teilmenge aus I, so daß wenn man die Elemente dieser Teilmenge mit dem Auswahloperator ‚|‘ zusammenfaßt, das so entstehende Inhaltsmodell die Ausgangsmenge der Sequenzen überdeckt und das MDL-Prinzip realsiert wird. Hierfür wird jedoch aus Komplexitätsgründen nicht exakt optimiert sondern nur approximiert.

  im Beispiel: das Inhaltsmodell $(ab)*|(a|b)(c|d)|b*(d|e)$ wird abgeleitet, beachte, daß dieses Inhaltsmodell u.a. schon dadurch nicht-1-mehrdeutig ist, da $a \in First(ab*) \cap First((a|b)(c|d))$ gilt.

Generalisierungsschritt

Es werden zusätzliche Ausdrücke produziert, welche den Kleene-Stern (*) enthalten, die ?- und +-Operatoren werden nicht berücksichtigt.

Die exakten Funktionsbeschreibungen entnehme man [17], die Funktion $DiscoverSeqPattern(s, r)$ sucht im übergebenen Ausdruck s benachbarte Auftreten von gleichen Teilworten x, wobei mindestens r Stück nebeneinander stehen, diese werden durch ein Hilfssymbol A ersetzt und der gesamte Ablauf wiederholt, bis keine solchen Auftreten mehr gefunden werden. Das Hilfssymbol A hat die Semantik $(x)*$ und es wird sichergestellt, daß $(x)*$ nur maximal ein Hilfssymbol zugegeordnet wird, denn es kann ja auch bei einem anderen Wort aus I auftreten.

Beispiel aus [18]: $s = abababcababc$ und $r = 2$, dann gilt nach dem ersten Durchlauf $s = A_1cA_1c$ und nach dem zweiten Durchlauf $s = A_2$ mit $A_1 = (ab)*$ und $A_2 = (A_1c)* = ((ab)*c)*$

Bei XTRACT wird für jedes Wort aus I die Funktion $DiscoverSeqPattern()$ für die Werte $r = 2, 3, 4$ angewendet.

Nachfolgend wird die Funktion $DiscoverOrPattern(s, d)$ angewandt, welche Ausdrücke der Form $(a_1|a_2|...|a_m)*$ ermittelt.
Hierzu wird die Heuristik benutzt, daß wenn der ursprüngliche reguläre Ausdruck $(a_1|a_2|...|a_m)*$ als Teil enthält, dann in den zugehörigen Worten z.B. häufig das Symbol a_i mehrfach aber nicht weit voneinander entfernt auftritt.
Hierzu wird s in kleinstmögliche Teilworte $\{s_1, ..., s_n\}$ mit $s = s_1s_2...s_n$ zerlegt, so daß für jedes Symbol a in jedem Teilwort s_i gilt: im d-Suffix von $s_1...s_{i-1}$ bzw. im d-Präfix von $s_{i+1}...s_n$ ist a nicht enthalten.
Enthält s_i die verschiedenen Symbole $a_1, ..., a_m$ mit $m > 1$, dann wird s_i in s durch das Hilfssymbol $A := (a_1|...|a_m)*$ ersetzt.
Durch die fehlende Formalisierung[10] in [18] bringen die Autoren ein fehlerhaftes Beispiel: für $abcbac$ wird für $d = 2$ die Sequenz aA_1ac mit $A_1 = (b|c)*$ produziert, was aber nach dem präsentierten Algorithmus nicht korrekt ist.

Ein korrektes Beispiel:
Ist $s = abcbaec$ so ergibt $DiscoverSeqPattern(s, d)$:

- $s = aA_1aec$ für $d = 2$, wobei $A_1 = (b|c)*$
- $s = aA_2$ für $d = 3$, wobei $A_2 = (a|b|c|e)*$
- $s = A_2$ für $d = 4$, wobei $A_2 = (a|b|c|e)*$

Konkret wird für jedes Ergebnis s' aus $DiscoverSeqPattern()$ die Funktion $DiscoverOrPattern(s', d)$ mit den Parametern $d \in \{0.1 * |s'|, 0.5 * |s'|, |s'|\}$ aufgerufen.

Es werden für jedes s aus I bis zu 9 Generalisierungen hinzugefügt, die jeweils die Eigenschaft haben, s zu überdecken.

Faktorisierungsschritt

Den exakten Algorithmus entnehme man [17]. Die Grundidee ist es, aus I' verschiedene Teilmengen S_i auszuwählen, für die gilt, daß ein Element aus S_i mit anderen (möglichst vielen) Elementen aus S_i einen (möglichst langen) gemeinsamen Suffix oder Präfix besitzt und andererseits für $d_1, d_2 \in S_i$ gilt, daß d_1 und d_2 nur möglichst wenige der Originalsequenzen in I gemeinsam überdecken.

Außerdem soll für $i \neq j$ auch $S_i \cap S_j$ möglichst klein sein. Die Heuristik hierfür ist, daß wir innerhalb von S_i möglichst gut faktorisieren können und ansonsten Redundanzen vermeiden, die häufig in hohen MDL-Kosten resultieren.

Die einzelnen S_i werden dann jeweils mit einem effizienten (heuristischen) Algorithmus faktorisiert.

[10] die hier wiedergegebene Semantik ist dem Algortihmus $Partition()$ aus [18] entnommen, nicht den dortigen Beispielen

MDL-Schritt

Sei I' die Menge an regulären Ausdrücken, die nach den beiden vorangegangen Schritten produziert wurde. In diesem Schritt geht es darum, $r_1, \ldots, r_n$ aus I' so auszuwählen, daß gilt: $L(r_1|\ldots|r_n) \supseteq I$ und

$$cost(r_1|\ldots|r_n) = \min_{p_1,\ldots,p_m \in I' : L(p_1|\ldots|p_m) \supseteq I} \{cost(p_1|\ldots|p_n)\}$$

Die Kosten $cost()$ ergeben sich aus der Summe der Anzahl der Bits die für die Kodierung der DTD benötigt werden und der Anzahl der Bits, die für die Kodierung der Worte aus I mit eben jener DTD gebraucht werden.

Kodierung der DTD

Hierbei geht es konkret nur um die Kodierung des DTD-Teiles (also das Inhaltsmodells) für das zu I korrespondierende Element. Hierzu betrachten wir das Inhaltsmodell als Zeichenkette über $\Sigma' := \Sigma \cup \{|, *, +, ?, (,)\}$.
Hat das Inhaltsmodell die Länge n, so benötigen wir $n * \lceil \log_2 |\Sigma'| \rceil$ Bits zur Kodierung.

Kodierung der Worte aus I

Die Autoren von [18] definieren die Funktion $seq(CM, s)$ über Wörten s aus I zur Bit-Darstellung von s mit Hilfe von CM wie folgt:

1. $seq(CM, s) := \epsilon$ gdw. CM enthält kein Zeichen aus $\{|,^*,^+,?,(,)\}$
2. $seq(CM_1 \ldots CM_k, s_1 \ldots s_k) := seq(CM_1, s_1) \ldots seq(CM_k, s_k)$ wenn $CM = CM_1 \ldots CM_k$, $s = s_1 \ldots s_k$ und $\forall i = 1 \ldots k : s_i \in L(CM_i)$
   Beachte, daß der erste Fall Priorität hat.
3. $seq(CM_1|\ldots|CM_k, s) := i\, seq(CM_i, s)$, wenn $s \in L(CM_i)$
   Für die Kodierung von i werden $\lceil \log_2 k \rceil$ Bits benötigt in normaler Binärdarstellung.
4. $seq(CM^*, s_1 \ldots s_k) := \begin{cases} k\, seq(CM, s_1) \ldots seq(CM, s_k) & k > 0 \\ 0 & k = 0 \end{cases}$
   wenn $s = s_1 \ldots s_k$. Der Wert von k kann beliebig groß sein, deshalb wird k so kodiert, daß zunächst $\lceil \log_2 k \rceil$ unär dargestellt wird (also eine Folge von 1en, die der Anzahl der Binärstellen von k entspricht), hiernach eine 0 als Trennsymbol folgt und danach k selbst in binärer Darstellung.

Beispiel:
$seq((ab|c)^*(de|fg^*), abccabfggg) = 1110100\,0\,1\,1\,0\,1\,11011$

Drei Aspekte müssen dringend Berücksichtigung finden:

- Die gewählte Kodierung sowohl für CM selbst als auch die Worte aus I ist die empirisch naheliegendste Variante, nicht aber die einzig mögliche. Wir können durch Vereinbarung von Prioritätsregeln die Anzahl der nötigen Klammern in CM beeinflußen und auch insbesondere für den Fall der Kodierung eines Wortes mi CM^* sind andere Varianten denkbar. Die gewählte Variante scheint die günstigste, ob sie das aber ist – und es überhaupt eine günstigste gibt – ist nicht klar und natürlich hat die Art der Kodierung wesentlichen Einfluß auf das Ergebnis durch das MDL-Prinzip.
- Es findet eine Diskretisierung statt, die dazu führen kann, daß Unterschiede in der Informationsmenge nicht mehr klar unterschieden werden können. Tatsächlich ist bei XTRACT als Seiteneffekt die Konstruktion eines Inhaltsmodells von allen anderen Inhaltsmodellen

abhängig, was natürlich keine wünschenswerte Eigenschaft ist. Sichtbar wird das dadurch, daß zu Kodierung von CM selbst $n * \lceil \log_2 |\Sigma'| \rceil$ Bits benötigt werden (wenn CM die Länge n hat), wobei die Größe von Σ' nicht von der Zahl der auftretenden Elementtypen in I sondern der *Gesamtzahl* an Elemententypen in der Beispiel-Dokumentmenge abhängt. Ob der Effekt wesentlich ist, ist eine andere Frage, aber er ist definitiv vorhanden, wodurch das Ergebnis nicht allein von I abhängt.

- Last, but not least ist $seq()$ nicht wohldefiniert, wie bereits erwähnt, haben die Autoren den Nicht-1-Mehrdeutigkeitsaspekt nicht berücksichtigt. Nicht einmal die Mehrdeutigkeit selbst wurde diskutiert und i.allg. haben zwei verschiedene Ableitungen desselben Wortes aus einem einzigen (mehrdeutigen) regulären Ausdruck nicht die gleiche Bitzahl entsprechend obiger Kodierung. Da uns nicht die Kodierung selbst sondern nur die Anzahl der Bits interessiert, können wir an dieser Stelle aber vereinbaren, eine jeweils kürzeste Kodierung zu wählen, falls es mehr als eine Möglichkeit gibt. Da die Autoren es nicht erwähnen, wird dies aber nicht realisiert sein, weiterhin müßten wir dann zusätzlich auch das Problem des Findens *aller* möglichen Ableitungen eines Wortes aus einem regulären Ausdruck betrachten.

Nachdem wir die Kodierung festgelegt haben, wird die Minimierung durchgeführt. Die Autoren zeigen hierbei, daß dieses Problem dem *Facility Location Problem* entspricht, welches NP-hart ist. Es gibt jedoch einen Algorithmus, der unter gewissen Voraussetzungen eine gute Approximation (d.h. mit Aussagen über die Güte) liefert. Die Autoren verwenden diesen Algorithmus obwohl die Voraussetzung nicht erfüllt ist, werten die Ergebnisse aber trotzdem als sehr gut.
Die Komplexität dieses Approximationsverfahrens geben die Autoren mit $O(|I'|^2 * \log |I'|)$ an, wodurch schon klar wird, daß gerade die Größe des Suchraums einen entscheidenen Einfluß auf die Komplexität des Gesamtproblemes hat.

Die von den Autoren vorgestellten Beispiel mit denen die XTRACT testeten lieferten sehr gute Ergebnisse, d.h. im allgemeinen wurden auch komplexe Inhaltsmodelle wieder erkannt. Hierbei wurden Inhaltsmodelle – sowohl theoretische, als auch welche aus der Praxis – benutzt, um Beispieldaten zu generieren, so daß $|I|$ in der Größenordnung zwischen 500 und 1000 lag. Allerdings war kein Beispiel vorhanden, in welchem ein Symbol mehr als einmal auftrat und auch keines, welches eine Schachtelungstiefe bzgl. der Klammerung von mehr als 2 hatte.

Der Ansatz von XTRACT sieht sehr vielversprechend aus, die Autoren haben sehr viel Arbeit in die Entwicklung von Heuristiken gesteckt um den Suchraum gezielt und gleichzeitig effizient aufzuspannen. Für die zukünftige Arbeit ist auch noch die Integration von (?) geplant. Einzig der Aspekt der Nicht-1-Mehrdeutigkeit wurde vergessen und es ist nicht klar, wie der aufgespannte Suchraum unter Nutzung des MDL-Prinzipes mit der Nicht-1-Mehrdeutigkeit harmoniert. Es ist auch nicht zu erwarten, daß wir unter Anwendung des MDL-Prinzipes die Nicht-1-Mehrdeutigkeit automatisch erreichen, zumal ja auch der Suchraum nicht sehr weit aufgespannt wird und hier tatsächlich dann auch nicht mit der exakten Lösung sondern einer Approximation gearbeitet wird.

3.3 von DTDs zu XML-Schema

Wenn wir übersichtsartig die Möglichkeiten von DTDs und XML-Schemata vergleichen wollen, so stoßen wir bei der Vernachlässigung von Entity-Referenzen vor allem auf folgende Punkte:

- Mehr Typen und Typhierachie

  Bei einer DTD haben wir eine beschränkte vordefinierte Menge an Typen für Attribute, welche ja nur Stringinhalt haben. Bei Elementen können wir entweder ein Inhaltsmodell angeben, daß von der Mächtigkeit einer regulären Sprache gleichkommt und somit festlegen, wie und welche Elemente als Kinder vorkommen, wobei das Element keinen Stringinhalt haben darf oder aber Stringinhalt zulassen, wobei wir noch angeben können, welche Elemente noch als Kinder auftreten können ohne Restriktionen an Häufigkeit und Anordnung. Den Stringinhalt von Elementen selbst können wir aber keinen Restriktionen unterwerfen.

Bei XML-Schema hingegen haben wir mehr vordefinierte Typen für Attribute und Elemente, die ausschließlich Stringinhalt besitzen. Der Typbegriff bei XML-Schema ist wie folgt: Es gibt einfache und komplexe Typen (engl. simple and complex types) . Einfache Typen sind Einschränkungen an Stringinhalte, also gewisse Teilmengen von Σ^*, Attribute können nur einfache Typen haben.

Elemente können entweder einfache Typen haben oder komplexe Typen. Ein komplexer Typ umfaßt das Inhaltsmodell und die Attribute sowie die Angabe, ob gemischter Inhalt erlaubt ist. Ausgehend von dem definierten anyType existiert eine (azyklische) Typhierachie mittels den Beziehungen der *Einschränkung*, der *Erweiterung* und bei einfachen Typen der *Vereinigung* und *Listenbildung*.

Die *Erweiterung* dient dazu, einen komplexen Typ mit mehr Attributen oder einem Inhaltsmodell mit hinzukonkatenierten Elementen bzgl. des Basistypes zu definieren, die *Einschränkung* dient dazu komplexe bzw. einfache Typen zu definieren, welche als Wertebereich nur eine Teilmenge des Wertebereichs des Basistypes haben.

Nachfolgend einige einfache Beispiele für Erweiterung und Einschränkung:

```
<complexType name="a_typ">
  <sequence>
    <element name="b" type="b_typ" maxOccurs="unbounded"/>
  </sequence>
</complexType>
```

Oben wird ein komplexer Typ namens `a_typ` definiert, dieser wird zu `a1_typ` erweitert:

```
<complexType name="a1_typ">
  <complexContent>
    <extension base="a_typ">
      <sequence>
        <element name="c" type="c_typ"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

Das Element mit Namen `c` wird an das Inhaltsmodell von `a_typ` „angehängt“. Einschränkuungen können von vielerlei Art sein, bei einfachen Typen handelt es sich um Restriktionen auf Zeichenkettenebene, bei komplexen Typen um Einschränkungen an Attribute und das Inhaltsmodell (Kardinalitäten, Typen von Elementen,...). Nachfolgender Typ `a2_typ` schränkt die Anzahl der `b`s ein:

```
<complexType name="a2_typ">
  <complexContent>
    <restriction base="a_typ">
      <sequence>
        <element name="b" type="b_typ" maxOccurs="3"/>
      </sequence>
    </restriction>
  </complexContent>
</complexType>
```

- Trennung von Elementnamen und Elementtyp

  Bei DTDs haben wir eine eindeutige Zuordnung von Elementnamen zu Elementtyp (also Inhaltsmodell, vorhandene Attribute usw.), DTD-XML-Sprachen waren ja baum-lokal. Bei XML-Schema hingegen folgt aus dem Elementnamen nicht sein Typ. Es gibt allerdings folgende Restriktion[11]:

  Zwei Kindelemente eines Elemente, welche gleiche Namen haben, besitzen auch den selben Typ.

  Hieraus folgt im Übrigen auch die Ein-Typigkeit der korrespondierenden regulären Baumgrammatik, denn die Regeln der Baumgrammatik ergeben sich wie folgt (als Beispiel):

  ```
  <complexType name="c_typ">
    <sequence>
      <element name="a" type="a_typ"/>
      <element name="b" type="b_typ"/>
    </sequence>
  </complexType>
  ```

  zu $c\_typ \rightarrow A, B \in P_2$, $A \rightarrow a\, a\_typ, B \rightarrow b\, b\_typ \in P_1$ wobei der Einfachheit halber disjunkte Mengen an Typ- und Elementnamen angenommen wurden und $a\_typ, b\_typ, c\_typ \in N_2$, $A, B \in N_1$ und $a, b \in \Sigma$ gilt.

  Aus der Restriktion (siehe [31] Abschnitt 3.8.6 Punkt: Schema Component Constraint: Element Declarations Consistent) folgt dann, daß für $a = b$ auch $a\_typ = b\_typ$ gelten muß, wir also gleiche rechte Seiten haben und demnach auch die linken Seiten (die ja sowieso künstlich eingeführt wurden) zusammenfallen lassen können, womit wir letztenendes keine Verletzung der Bedingung für die Ein-Typigkeit mehr haben.

- Substitution-Groups

  Wir können eine Top-Level-Elementdeklaration zum Kopf einer Ersetzungsgruppe machen und andere Top-Level-Elementdeklarationen[12] dieser Ersetzungsgruppe zuordnen.

  In einer Instanz sind dann alle Elemente aus der Ersetzungsgruppe anstelle des Kopfes der Ersetzungsgruppe ohne besondere Kennzeichnung benutzbar. Die Typen der Mitglieder der Ersetzungsgruppe müssen aber vom Typ des Kopfes der Ersetzungsgruppe abgeleitet worden sein.

  Jede Top-Level-Elementdeklaration definiert automatisch eine Ersetzungsgruppe, die Mitgliedschaft in Ersetzungsgruppen ist reflexiv und transitiv (aber nicht symmetrisch), d.h. ist A eine Top-Level-Elementdeklaration, enthält die Deklaration für B `substitutionGroup="A"` und die Deklaration für C enthält `substitutionGroup="B"`, so sind in der Ersetzungsgruppe zu A die Elementdeklarationen für A, B und C enthalten.
  Eine Elementdeklaration kann auch höchstens in einer Ersetzungsgruppe enthalten sein, die nicht ihre eigene ist.

- Inhaltsmodelle

  Inhaltsmodelle selbst sind gleich in ihrer Mächtigkeit, wobei bei XML-Schema der `all`-Operator syntaktisch hinzukommt, außerdem können wir Kardinalitäten exakter spezifizieren. Bei DTDs hatten wir nur die Kardinalitäten $[0, 1]$ alias ‚? ‘, $[1, 1]$, $[0, n]$ alias ‚*‘ und $[1, n]$ alias ‚+‘zu Verfügung, bei XML-Schema können wir beliebige Kardinaltäten $[x, y]$ mit $x \leq y$ und $x \in N_0$ und $y \in N_0 \cup \{n\}$ angeben.

[11] siehe hierzu auch Anhang: Interpretation der XML-Schema Spezifikation

[12] siehe hierzu auch Anhang C

XML-Schema ist aber dadurch nicht mächtiger, z.B. läßt sich $x[2,3]$ ausdrücken als $xx|xxx$ und $y[4,n]$ als $yyyy+$. Der `all`-Operator ist eine stark eingeschränkte Form des SGML &-Operators. Er ist nicht überall einsetzbar (nur als höchstes Element eines Inhaltsmodells) und darf ausschließlich Elemente als Inhalt haben, d.h. keine Sequenzen oder Auswahlen. Die enthaltenen Elemente müssen auch ein `maxOccurs` von 0 oder 1 aufweisen. Ein Ausdruck $a\&b\&c$ läßt sich in den äquivalenten Ausdruck $abc|acb|bac|cab|bca|cba$ umformen, was deshalb prinzipiell auch mit dem `all`-Operator geht.

- Integritätsbedingungen

  XML-Schema bietet uns Möglichkeiten der Angabe von Schlüssel-, Fremdschlüssel- und Eindeutigkeitsbedingungen. Außerdem können wir Kardinalitäten „x bis y mal“ $(x \leq y)$ nun exakter ausdrücken, als bei DTDs, mehr dazu bei dem Abschnitt über Integritätsbedingungen.

- Konzept der Name-Spaces

  Mit XML-Schema wird das Konzept der Namensräume (engl. name spaces) vervollständigt, konkret dienen XML-Schemata zum Populieren von Namensräumen. In den Beispieldokumenten können jetzt mehrere Namensräume verwendet werden, an dieser Stelle wollen wir jedoch voraussetzen, daß alle Elemente und Attribute aus dem gleichen Namensraum kommen, mit Ausnahme von `xsi:nil`, `xsi:type`, ... Sollten Attribute aus anderen Namensräumen auftreten, so werden diese vor der Bearbeitung entfernt, sollten Elemente aus anderen Namensräumen auftreten, so werden diese ebenfalls entfernt, allerdings müssen wir uns an den entsprechenden Stellen merken, daß dort `any` bzw. `anyAttribute` steht.

  Unser Namensraum wird – falls vorhanden – durch den Namensraum des Wurzelelementes bestimmt, der in allen Dokumenten gleich sein soll. Wir nehmen dann dementsprechend `targetNamespace="..."` in das `schema`-Element auf.

- `xsi:nil`

  Elemente können das Attribut `xsi:nil` mit dem Wert $true$ haben, welches dazu dient, einen Nullwert zu signalisieren. In diesem Fall muß der Elementinhalt leer sein, allerdings darf das Element Attribute besitzen.
  Damit `xsi:nil="true"` verwandt werden kann, muß in der Element-Deklaration `nillable="true"` aufgeführt werden. Trägt das Element das Attribut `xsi:nil="true"`, hat keinen Inhalt und die anderen Attribute sind korrekt vorhanden, so wird das Element validiert – sogar dann, wenn das Inhaltsmodell ϵ nicht zuläßt.
  Daher dürfen wir solche Elementinstanzen bei der Schemaableitung nicht berücksichtigen, wissen aber, daß wir bei der entsprechenden Elementdeklaration `nillable="true"` zu setzen haben.

3.3.1 Mögliche Vorgehensweisen

Naheliegend für die Generierung eines XML-Schemas sind nun drei Möglichkeiten:

- einen DTD-Ansatz benutzen und dann transformieren

  Die einfachste Möglichkeit ist es, einen der genannten Ansätze für die Generierung von DTDs zu benutzen und das Ergebnis einfach in ein XML-Schema zu transformieren. Nachfolgend folgt ein Transformationsalgorithmus. Hiermit wird allerdings das Potential von XML-Schema in keinster Weise ausgenutzt.

- Dokumente aufbereiten, DTD-Ansatz benutzen, transformieren und ergänzen

  Um XML-Schema besser auszunutzen, muß es möglich sein, daß der Elementname nicht den Elementtyp impliziert. Über geeignete Heuristiken müssen wir also die Menge I zum Elementnamen e auftrennen in $I = I_1 \cup ... \cup I_n$, wobei der Inhalt eines e-Elementes in genau einem I_j ist. Wir legen also für jedes e-Element den Typ $T_j, j = 1...n$ fest, sammeln dann für alle e-Elemente vom Typ T_j die Inhalte in I_j und führen mit den vorhandenen Algorithmen die Typ-Ableitung durch.

  Wenn wir die Algorithmen nicht ändern wollen, können in einem separaten Schritt vor dem Ablauf des Algorithmus alle Element so umbenannt werden, daß gilt: *Elementname impliziert Typ* und nach Ablauf des Algorithmus wird die Umbenennung wieder rückgängig gemacht.

  Hiernach erfolgt dann wieder die Transformation in ein XML-Schema und abschließend wird noch versucht, dieses XML-Schema um Datentypinformationen und Integritätsbedingungen anzureichern. Die Anreicherung wird dann in den entsprechenden Abschnitten diskutiert.

  Vorteil ist, daß wir XML-Schema sehr gut ausnutzen, wobei allerdings erstens noch die Heuristiken zur Typfestlegung von Elementen diskutiert werden müssen und weiterhin noch keine Vorkehrungen für Konstrukte wie den `all`-Operator getroffen sind. Weiterhin ist unklar, ob die Algorithmen mit den erweiterten Ausdruckmöglichkeiten bei den Kardinalitäten und Typhierachien harmonieren.

- Wir können ausgehend von einem der obigen Ansätze versuchen, die XML-Schema-Möglichkeiten zu integrieren.
  Hiermit würden wir natürlich XML-Schema optimal ausnutzen, andererseits kann es sein, daß das Vorgehen nach dem zweiten Vorschlag mit einem konkreten Algorithmus schon das optimale Ergebnis liefert, was Implementationsaufwand sparen würde.

Nachfolgend nun ein Satz mit dem zugehörigen Algorithmus, der bei den ersten beiden Möglichkeiten erwähnt wurde:

Satz 5 *Jede XML-Sprache L die mit einer DTD D definiert ist, läßt sich auch mit einem XML-Schema S definieren.*

Beweis:
Im Anhang D befindet sich ein Algorithmus zur Konvertierung einer DTD in ein XML-Schema.
q.e.d.

An dieser Stelle wird nun die Möglichkeit der Benutzung vorhandener Algorithmen genauer diskutiert. Die Menge der Dokumente muß zunächst vorbehandelt werden:

- Identifiziere Mengen von Elementen, die gleiche Namen und gleiche Typen besitzen. Sind ausgehend vom jeweiligen Wurzelelement zwei Elemente über den gleichen XPath-Ausdruck $e_1/e_2/.../e_n$ erreichbar und sind die Wurzelelemente gleichbenannt, so müssen sie gleiche Typen besitzen. Das folgt mit Induktion über die Pfadlänge ausgehend von der oben genannten Restriktion, die besagt, daß Kindelemente desselben Elementes gleiche Typen besitzen müssen, wenn sie gleiche Namen haben. Gibt es in I Dokumente mit verschieden benannten Wurzelelementen, so ist jeder möglicher Pfad ausgehend von jedem möglichem Wurzelelement zu bilden.

  Wie oben angesprochen bewirkt das eine Zerlegung von I in $I_1 \cup ... \cup I_k$, wobei der Benutzer zusätzliche semantische Information einbringen kann, in dem er angibt, welche Mengen wieder vereinigt werden können.

Falls der Fall eintritt, daß ein abgeleiteter Typ benutzt wird, so wird das durch die Nutzung des `xsi:type`-Attributes kenntlich gemacht. Trotz evt. gleicher Pfade müssen die Inhalte solcher Elemente separat gesammelt werden, eine spezielle Menge wird festgelegt pro auftretendem Wert dieses Attributes. Zur Vereinfachung kann angenommen werden, daß bei Nicht-Vorhandensein dieses Attributes der Originaltyp Verwendung findet.

Mit den Mengen I_i werden dann die vorgestellten Algorithmen benutzt, jeder Menge I_i ist entweder ein Pfad der Form $e_1/e_2/.../e_n$ oder ein Wert des `xsi:type`-Attributes zugeordnet, so daß wir nachher hieraus das XML-Schema beginnend beim Wurzelelement konstruieren können.

- Die vorgestellten Algorithmen setzen jeweils voraus, daß die Element entweder keinen oder nur Elementinhalt haben, der Fall mit `#PCDATA` wird bei DTDs ob der Einfachheit separat gehandhabt. Bei XML-Schema ist dies jedoch anders, lassen wir zunächst die Textinhalte stehen, so unterscheiden wir drei Fälle:
  - jedes Element aus I_i enthält keinen Text, sondern nur XML-Elemente:
in diesem Fall brauchen wir nichts mehr ändern
  - alle Elemente aus I_i enthalten nur Text, aber keine XML-Element:
Hier brauchen wir kein Inhaltsmodell erkennen, sondern nur einen einfachen Datentyp, hierzu siehe Kapitel 4
  - wir haben gemischten Inhalt vorliegen:
Wir merken uns zur Menge I_i das Attribut `mixed="true"` welches später in den entsprechenden Typ mit aufgenommen wird, entfernen alle Textinhalte aus I_i und verfahren dann wie im ersten Fall.
- enthält I_i Attribute bzw. Elemente mit Namespace-Teilen im Namen, so werden diese entfernt, wenn der Namensraum des Wurzelelementes ein anderer (oder nicht vorhanden) ist. Zu diesem I_i wird dann aber vermerkt, daß im jeweiligen Typ dann später entsprechende `any`- bzw. `anyAttribute`-Elemente aufgenommen werden. Hierbei geht es aber um die normalisierten Namen, d.h. hat ein Elementname syntaktisch keinen Namespace-Teil, ist aber in einem übergeordneten Element ein Namensraum angegeben, der für das untergeordnete Element gilt, so ist genauso zu verfahren, als wenn das Element diesen Namespace-Teil im Namen hätte.

Mit den so vorbehandelten bzw. gebildeten Mengen wird eines der vorgestellten Verfahren benutzt und nachher das XML-Schema konstruiert. Generell (also auch bei der Anpassung eines Verfahrens für XML-Schema) sollten die folgenden Schritte ausgeführt werden:

- Sammeln der Attribute für jeden Typ
- Erkennen der (einfachen) Datentypen für Attribute und Elemente, die dies betrifft, hierzu Kapitel 4
- Ableiten der Integritätsbedingungen wie Schlüssel, Eindeutigkeitsbedingungen und Fremdschlüssel

Haben wir eines der vorgestellten Verfahren nach obiger Vorbehandlung benutzt, sollten noch folgende Schritte stattfinden, teilweise sogar vor der Konstruktion des XML-Schemas:

- Wir können ausgehend von den Ergebnissen versuchen herauszufinden, ob das Inhaltsmodell zweier Elemente mit gleichem Namen identisch ist. Hierzu können wir Heuristiken anwenden:

  Gibt es bestimmte Attribute oder Kindelemente in nur einem der beiden, so sind es wahrscheinlich verschiedene Typen (allerdings können diese bzgl. der Strukturvererbung in

Beziehung stehen), anderenfalls müssen wir die Ergebnisse mit einer gewissen Metrik vergleichen.

Im Gleichheitsfall (wenn der Abstand der Ergebnisse bzgl der Metrik sehr klein ist) können wir eine Zusammenführung auf syntaktischer Ebene versuchen (insbesondere dann, wenn eine Teilmengenbeziehung gilt) oder aber wir fassen die ursprünglichen Mengen I_i und I_j zusammen und führen das Verfahren auf $I_i \cup I_j$ erneut aus. Wir können auch an dieser Stelle den Nutzer bitten, solche Zusammenlegungen zu initiieren.

- Die vorgestellten Algorithmen beachten nicht die Möglichkeiten der Kardinalitätsangabe bei XML-Schema. Wie hier verfahren werden kann, wird in Abschnitt 5.6 diskutiert.

- Ebensowenig wird `all` erkannt, hier haben wir die Möglichkeit, ausgehend vom Ergebnis zu begutachten, ob ein `all`-Element in Frage kommt. Hierzu muß das Ergebnis CM ein Inhaltsmodell der Form: $(CM_1|...|CM_n)$ sein, wobei einerseits gelten soll, daß für alle i gilt: CM_i ist eine Sequenz aus Elementen, die höchstens mit ? quantifiziert sind und jedes Element tritt in CM_i nur einmal auf und $|Sym(CM)| - \epsilon \leq n \leq |Sym(CM)|$[13] für ein gewisses ϵ.
  In diesem Fall können wir ein `all`-Element einführen, daß – entsprechend quantifiziert – jedes Element aus $Sym(CM)$ genau einmal enthält. Problematisch wird es, wenn die Verfahren hier übergeneralisiert (* oder +) haben, denn eine Auflistung von $n!$ Elementpermutationen könnte je nach Verfahren zu einem schlechten MDL-Wert, zu einer Übergeneralisierung bei endlichen Automaten oder zur Anwendung einer Regel, die gute Lesbarkeit erzwingt, führen.

Ebenso stellt sich die Frage der Erkennung von Ersetzungsgruppen (engl. substitution groups). Das Anwenden eines Elementes aus der Ersetzungsgruppe anstelle des Originalelementes wird nicht gekennzeichnet, einzig der Typ des Elementes ist aus dem Typ des Originalelementes abgeleitet.

In einer Instanz können auch ohne Probleme zwei Elemente der selben Ersetzungsgruppe auftreten, angenommen, das ursprüngliche Inhaltsmodell habe die Form aa und b, c sind in der Ersetzungsgruppe zu a, so ist die Instanz bc erlaubt. Es ist nicht zu erwarten, für diesen Fall Heuristiken zu finden, d.h. der Benutzer muß nach Möglichkeit diese Information beisteuern.

Gleichfalls das Finden von Strukturvererbung ist nicht einfach. Wurde `xsi:type` benutzt, so wissen wird, daß gewisse Typen von anderen abgeleitet wurden, nicht aber wie und ob direkt oder indirekt. Bei direkter Vererbung könnte man versuchen, die Inhaltsmodelle zu vergleichen bzgl. der in XML-Schema vorgegebenen Restriktionen an die Vererbung. Minimale Änderungen am Inhaltsmodell (konkret daß nicht ganz exakte Erkennen des ursprünglichen Modelles) können aber diesen Ansatz scheitern lassen.

Es scheint in diesem Fall ratsam, den Nutzer manuell Vererbungstrukturen erkennen zu lassen und diese dann auf Plausibilität zu prüfen.

Diese beiden Punkte gelten natürlich auch für einen speziellen Algorithmus für XML-Schema. Was man aber bei diesem konkret implementieren könnte, wäre eine Unterstützung für den `all`-Operator. Naheliegend ist das bei Ansätzen, die Regelsystem verwenden (wie GB-Engine) oder dem XTRACT-Ansatz, bei welchem man die Kandidatenmenge und die Kodierung entsprechen um `all`-Konstrukte erweitern könnte, die dann im MDL-Schritt mit bewertet werden.

[13] vorausgesetzt, kein CM_i wird von einem CM_j mit $i \neq j$ subsummiert

Kapitel 4

Datentypen

In diesem Kapitel wird die Ableitung von Datentypen für Attribute und Elemente mit ausschließlichem Zeichenketteninhalt diskutiert. Hierbei ist vorausgesetzt, daß die konkreten Textinhalte der Instanzen selbst bezüglich der Zeichenkodierung korrekt sind, zu einem früheren Zeitpunkt haben wir ja auch schon angenommen, daß die Beispieldokumentmenge alle Restriktionen bezüglich der Wohlgeformtheit erfüllt.

Die Instanzmenge an Zeichenketten zu einem Element bzw. Attribut sei hier mit J bezeichnet, ausgehend von den Mengen I_i[1] der Elementinhalte werden diese wie folgt gebildet: Handelt es sich (im Fall von XML-Schema) um den Inhalt eines Elementes, so $J := I_i$, anderenfalls betrifft es ein Attribut a, welches von dem Elementtyp e, der zu I_i korrespondiert, besessen wird. Im zweiten Fall wird J so gebildet, daß zu jedem e-Element, welches zu I_i korrespondiert, der Inhalt des a-Attributes zu J hinzugefügt wird, sofern a vorhanden ist.

Prinzipiell können wir keine Attribute erkennen, welche in der ursprünglichen DTD bzw. dem ursprünglichen XML-Schema als *optional* gekennzeichnet waren und in der Beispieldokumentmenge niemals auftreten, es sei deshalb angenommen, daß die Beispieldokumentmenge die Möglichkeiten des ursprünglichen Schemas „ausschöpft".

Generell können wir entscheiden, ob für ein Attribut ein gewisser Typ möglich ist oder nicht. Bei vielen Typen ist hierfür sogar ausschließlich der Inhalt hinreichend. Es tritt jedoch meistens – insbesondere bei XML-Schema – die Situation auf, daß wir mehrere Möglichkeiten haben, einem Attribut ein Typ zuzuordnen. In solchen Fällen – den Typ ID hier einmal herausgenommen – ist es sinnig, über den möglichen Attributen eine Ordnung $\prec$ zu definieren, wobei der kleinste Typ gem. $\prec$ der „wahrscheinlichste" Typ ist. Wie wahrscheinlich ein Typ ist, wird mit Heuristiken bestimmt. Diese Heuristiken kann man beinahe zu beliebiger Komplexität ausbauen, was an dieser Stelle zu weit führen würde, deshalb werden nachfolgend nur einige grundlegende Heuristiken benutzt.

Die Aufgabe ist es nun, alle für eine Attributdeklaration benötigten Informationen zu erschließen, dazu gehören neben dem eigentlichen Datentyp auch noch Qualifikatoren (z.B. `#FIXED` bei DTDs). Das Attribut sei immer mit a bezeichnet, der Element*typ* mit e_typ.

Viele Datentypdefinitionen werden auf einige Definitionen aus der XML-Spezifikation [11] zurückgeführt, beachte hierbei, daß es für die Kodierung verschiedene Möglichkeiten (z.B. UTF-8 und UTF-16) gibt, die hier nicht festgelegt wird:

[1] Durch die Zuordnung zu I_i und nicht zu Elementnamen ersparen wir uns an dieser Stelle die erneute Diskussion wenn bei XML-Schema der Fall auftritt, daß es gleiche Elementnamen mit verschiedenen Typen gibt.

```
Char     ::= #x9 | #xA | #xD | [#x20-#xD7FF] | [#xE000-#xFFFD]
             | [#x10000-#x10FFFF]

S        ::= (#x20 | #x9 | #xD | #xA)+

NameChar ::= Letter | Digit | CombiningChar | Extender | '.' | '-'
             | '_' | ':'

Name     ::= (Letter | '_' | ':') (NameChar)*

Names    ::= Name (S Name)*

Nmtoken  ::= (NameChar)+

Nmtokens ::= Nmtoken (S Nmtoken)*
```

wobei `Letter, Digit, CombiningChar` und `Extender` in der XML-Spezifikation ([11]; Anhang B) definiert sind. Die `S`-Produktion steht für Leerzeichen (engl. white spaces). Ausgehend von den genannten Produktionen folgt:

- $L(Name) \subset L(Nmtoken)$
- $L(Name) \subset L(Names)$
- $L(Nmtoken) \subset L(Nmtokens)$

Weitere Zusammenhänge werden durch die Transitivität von $\subset$ impliziert.

4.1 DTD

4.1.1 Datentypen

Bei DTDs haben wir nur eine Menge vordefinierter Typen, die im folgenden diskutiert werden:

CDATA CDATA stellt keine Restriktionen[2] an den Stringinhalt und ist daher immer anwendbar.

ID Die Funktion von ID-Attributen wird im Kapitel 5 ausführlicher vorgestellt.

Der Typ ID ist für das a-Attribut *nicht* möglich, wenn mindestens eines der folgenden wahr ist:

- der Attributwert von a für *ein* e_typ-Element nicht konform der $Name$-Produktion,
- einem Attribut b mit $b \neq a$ des e_typ-Elementes ist schon der Typ ID zugeordnet,
- Attributen b_i anderer Elementtypen ist schon ID als Typ zugeordnet und $\exists i : I_a \cap I_{b_i} \neq \emptyset$,
- dem Attribut a ist keiner der Qualifikatoren #IMPLIED oder #REQUIRED zugeordnet[3],
- es gibt n Elemente vom e_typ im *aktuellen* Dokument d_k mit vorhandenem a-Attribut und $||J|_{d_k}| \neq n$

[2] die über die Zeichendarstellung selbst hinausgehen

[3] es ist nicht notwendig, zuerst alle Qualifikatoren und dann die Typen zu erschließen, die Reihenfolge kann zweckmäßig angepaßt werden

IDREF Der Typ IDREF ist für das a-Attribut *nicht* möglich, wenn mindestens eines der folgenden wahr ist:

- der Attributwert von a für *ein* e_typ-Element nicht konform der $Name$-Produktion,
- es gibt kein Element in dem *aktuellen* XML-Dokument, welches ein Attribut vom Typ ID besitzt und der Wert dieses ID-Attributes stimmt mit dem Wert des a-Attributes überein.

IDREFS Der Typ IDREFS ist für das a-Attribut *nicht* möglich, wenn mindestens eines der folgenden wahr ist:

- der Attributwert von a für *ein* e_typ-Element nicht konform der $Names$-Produktion,
- es gibt für einen $Name$ aus der Werteliste des a-Attributes kein Element in dem *aktuellen* XML-Dokument, welches ein Attribut vom Typ ID besitzt und der Wert dieses ID-Attributes stimmt mit dem Wert des $Name$ überein.

ENTITY Der Wert eines Attributes vom Typ ENTITY muß der $Name$-Produktion genügen, weiterhin muß es eine *unparsed entity* mit einem Namen geben, der dem Wert dieses Attributes entspricht.

Wie auch weiter unten bei NOTATION diskutiert, können wir *unparsed entities* nicht erschließen und haben nach Voraussetzung auch keine definiert, da wir keine interne oder externe DTD besitzen, somit können wir den Typ ENTITY nicht ableiten.

ENTITIES Analog ENTITY, nur daß das der Attributwert der $Names$-Produktion genügen muß und mehrere *unparsed entities* referenziert.

NMTOKEN Die Werte aus J müssen der `Nmtoken`-Produktion genügen.

NMTOKENS Die Werte aus J müssen der `Nmtokens`-Produktion genügen.

NOTATION Wir hatten angenommen, daß kein Dokument aus der Beispieldokumentmenge eine interne DTD besitzt. Attribute vom Typ NOTATION verweisen mit ihrem Wert auf eine in der DTD definierte *Notation*. Da Notationen nicht ableitbar sind – sie müssen explizit definiert werden – und wir daher keine definierten Notationen besitzen, können wir für ein Attribut niemals den Typ NOTATION ableiten.

Beispiel (aus [7]):
Ausschnitt aus einer DTD:

```
...
<!NOTATION JPEG PUBLIC "ISO/IEC 10918:1993//NOTATION
            Digital Compression and Coding of
            Continuous-tone Still Images
            (JPEG)//EN"         >
<!ELEMENT e (#PCDATA)>
<!ATTLIST e
  a NOTATION #REQUIRED>
...
```

Ausschnitt aus einem zugehörigen XML-Dokument:

```
...
<e a="JPEG">
...
</e>
...
```

Aufzählung Die Werte bei einer Aufzählung müssen der $Nmtoken$-Produktion genügen.

Beispiel:

```
...
<!ATTLIST e
  farbe (rot | gruen | blau) #IMPLIED>
...
```

4.1.2 Vorgehen

Zunächst stellt sich die Frage, ob der Nutzer eine Typerkennung von ID-Attributen vornehmen lassen will oder nicht. Falls das gewünscht ist, ist es zweckmäßig, zunächst *alle* Typzuordnungen vom Typ ID vorzunehmen und danach die restlichen Typzuordnungen durchzuführen, welche jeweils unabhängig von der Typzuordnung zu anderen Attributen, von denen der Typ noch unbekannt ist, sind.

Für jeden Elementtyp e_typ der die Attribute $a_i : i = 1 \dots n$ besitzt, kann unter Vernachlässigung der anderen Elementtypen jedes a_i genau dann vom Typ ID sein, wenn es bei jedem Auftreten in *einem* Dokument – wir schränken hierfür J ein auf $J|_{d_k}$ wobei d_k eines der Beispieldokumente sei – einen anderen Wert besitzt und das für *jedes* Beispieldokument gilt.

Die Menge dieser Attribute für das e_typ-Element sei mit $IDs(e\_typ)$ bezeichnet, natürlich kann nur *höchstens einem* Attribut dieser Menge der Typ ID zugeordnet sein. Sind $e_i : i = 1 \dots m$ alle auftretenden Elementtypen und wählen wir aus einigen der Mengen $IDs(e_i)$ entsprechend Attribute a_i aus, so müssen wir sicherstellen, daß gilt:

$$\forall d_k \in I : \forall e_i, e_j \; mit \; i \neq j : J_{a_i}|_{d_k} \cap J_{a_j}|_{d_k} = \emptyset$$

Dafür gibt es i.allg. mehrere Möglichkeiten, die dem Nutzer natürlich alle zur Auswahl zur Verfügung gestellt werden können. Hierfür gibt es $\Pi_{i=1\dots m}(|IDs(e_i)| + 1)$ Möglichkeiten, die $+1$ rührt aus der Tatsache, daß wir natürlich für einen Elementtyp auch *kein* ID-Attribut haben müssen.

Alternativ könnten wir nach einer Maximalzahl von ID-Attributen suchen, d.h. wenn wir den Graphen $G = (V, E)$ mit $V := \bigcup_{i=1\dots m} IDs(e_i)$ – wobei die IDs-Mengen o.B.d.A. disjunkt sind – und $\{a_i, a_j\} \in E$:gdw. a_i, a_j gehören beide zum selben Elementtyp oder $\exists d_k \in I : J_{a_i}|_{d_k} \cap J_{a_j}|_{d_k} \neq \emptyset$, dann ist das äquivalent zur Suche nach einer maximalen unabhängigen Knotenmenge, ein Problem, daß NP-vollständig ist[4].

Für Attribute vom Typ ID sind wir an dieser Stelle fertig, deswegen sei nachfolgend angenommen, daß dem a-Attribut nicht der Typ ID zugeordnet wurde. Im folgenden sei die Funktion $checkType(production, string)$ angenommen, die per Def. genau dann $wahr$ ist, wenn $string$ der $production$-Produktion genügt, andernfalls $falsch$ liefert.
$checkType()$ sei auf Mengen von Zeichenketten wie folgt erweitert:
$checkType(production, \{w_1, \dots, w_n\}) := \bigwedge_{i=1\dots n} checkType(production, w_i)$

Wir bilden jetzt für I die Menge $possibleTypes(J) \subseteq$
$\{CDATA, IDREF, IDREFS, NMTOKEN, NMTOKENS, Aufzählung\}$.

Es gilt:

- $possibleTypes(J) = \bigcap_{i \in J} possibleTypes(\{i\})$
- $CDATA \in possibleTypes(J)$

[4] Das Problem CLIQUE ist NP-vollständig (vgl. [29]), und jede CLIQUE ist eine unabhängige Knotenmenge im Komplementärgraphen.

- $NMTOKEN \in possibleTypes(J) : gdw. \, checkType(Nmtoken, J)$
- $NMTOKENS \in possibleTypes(J) : gdw. \, checkType(Nmtokens, J)$
- $NMTOKEN \in possibleTypes(J) \Leftrightarrow Aufzählung \in possibleTypes(J)$
- $IDREF \in possibleTypes(J) \Rightarrow IDREFS \in possibleTypes(J)$
- $NMTOKEN \in possibleTypes(J) \Rightarrow NMTOKENS \in possibleTypes(J)$
- $IDREF \in possibleTypes(J) \Rightarrow NMTOKEN \in possibleTypes(J)$
- $IDREFS \in possibleTypes(J) \Rightarrow NMTOKENS \in possibleTypes(J)$
- $IDREFS, NMTOKEN \in possibleTypes(J) \Rightarrow IDREF \in possibleTypes(J)$

Für IDREF(S) muß gelten $checkType(Name(s), J) = wahr$ und weiterhin muß der Wert (bzw. müssen die Werte) bei Attributen vom Typ ID im *selben* Dokument auftreten, formal:

$$\forall {d_k}^5 \in I : J|_{d_k} \subseteq \bigcup_{a_i \, hat \, Typ \, ID} J_{a_i}|_{d_k}$$

Heuristisch gesehen erschließen wir die „meiste“ Semantik, wenn wir einen Typ zuordnen, der unter allen Möglichkeiten die meisten Restriktionen liefert. Dieser Typ ist natürlich nicht eindeutig festgelegt, deshalb ist es sinnvoll, dem Nutzer eine Liste mit allen Elementen aus $possibleTypes(I)$ anzubieten, welche nach Heuristiken bzgl. der Wahrscheinlichkeit absteigend geordnet ist und dem Nutzer die Wahl zu überlassen oder aber alternativ den ersten Typ in der Liste zu wählen.

- Nach Voraussetzung können wir jedem a-Attribut den Typ CDATA zuordnen, jede Zuordnung eines anderen Types ist eine stärkere Restriktion.
- Haben wir die Möglichkeit, sowohl NMTOKEN als auch NMTOKENS (bzw. IDREF und IDREFS) zuzuordnen, so ergibt die Nicht-Listenvariante eine größere Restriktion, hier ist also NMTOKEN (bzw. IDREF) vorzuziehen.
- Haben wir die Möglichkeit, sowohl IDREF als auch NMTOKEN (bzw. IDREFS als auch NMTOKENS) als Typ zu wählen, so ist die Typfestlegung auf IDREF (bzw. IDREFS) die größere Restriktion.
- Sind sowohl Aufzählung und NMTOKEN möglich, so ist es naheliegend, dann eine Aufzählung als Typ anzunehmen, wenn die Anzahl verschiedener Werte gering ist. Also wenn n die Anzahl der *e_typ*-Elemente in der Beispielmenge ist, für die a vorhanden ist und $|J| \ll n$ gilt, so ist ein Aufzählungstyp wahrscheinlich.

Das ist natürlich nur eine Möglichkeit, Typen für DTDs festzulegen. Die Entwicklung weitergehender Heuristiken führt über den Rahmen dieser Diplomarbeit hinaus. Denkbar wäre z.B. sowohl Attributnamen als auch die Attributwerte in die Typableitung mit einzubeziehen. So spricht ein Attributname von id mit großer Wahrscheinlichkeit auch für den Attributtyp ID und Attributwerte von *rot, grün, blau, ...* sprechen mit großer Wahrscheinlichkeit für einen Aufzählungstyp über den Farben.

[5] Beachte, daß i.Allg. $J|_{d_{k_1}} \cap J|_{d_{k_2}} \neq \emptyset$

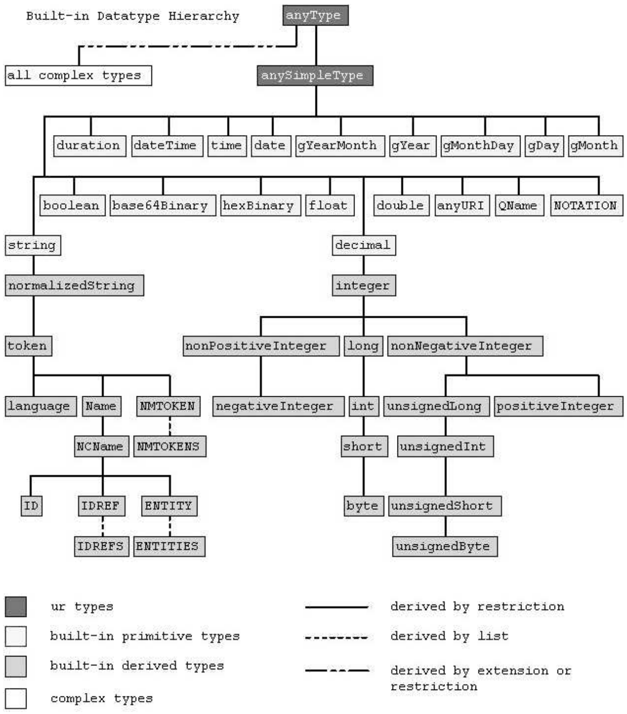


Abbildung 4.1: Die Typ-Hierarchie der einfachen Typen von XML-Schema (Abb. aus [9])

4.2 XML-Schema

Bei XML-Schema wird die Möglichkeit, Attributen Datentypen zuzuordnen, dadurch erweitert, daß auch Elementen mit ausschließlichem Stringinhalt solche *einfachen Datentypen* zugeordnet werden können. Weiterhin gibt es eine wesentliche größere Anzahl vordefinierter Datentypen und die Möglichkeit, aus vorhandenen Datentypen weitere Datentypen abzuleiten, entweder mittels *Restriktion*, *Vereinigung* oder *Liste*.

4.2.1 Datentypen

Dementsprechend unterteilen sich die Datentypen in:

- atomare Datentypen
- Listendatentypen
- Vereinigungsdatentypen

Orthogonal dazu gibt es die Unterscheidung in *eingebaute* und *nutzerdefinierte* Datentypen, sowie die Trennung nach *primitiven* und *abgeleiteten* Datentypen. Mehr dazu in [9].

Einen Überblick über die Typhierachie liefert die Abbildung 4.1 aus [9].

4.2.2 Facetten

Die Werteraum (engl. value space) und der zugehörige lexikalische Raum (engl. lexical space) sind über sogenannte Facetten (engl. facets) definiert. Die Fundamentalen Facetten, wie z.B. die Kardinalität, die *endlich* oder *abzählbar unendlich* annehmen kann sind für die primitiven Datentypen in [9] definiert und ergeben sich für abgeleitete Typen entsprechend der Art der Ableitung.

Beim Ableiten selbst können wir die nicht-fundamentalen Facetten (engl. constraining oder non-fundamental facets) verwenden. Welche wir im einzelnen verwenden können, hängt bei der Ableitung mit Restriktion vom Basisdatentyp ab, bei der Ableitung mit der Vereinigung oder Liste sind gewisse Facetten unabhängig von den Basisdatentypen möglich.

Eine Übersicht über die nicht-fundamentalen Facetten:

$length$ Anzahl der Zeichen bei $string$ und Ableitungen davon, Anzahl der Listenelemente bei Listenableitungen, sowie Anzahl der Bytes bei $hexBinary$ und $base64Binary$ und Ableitungen von diesen.
Der Wert von $length$ muß exakt eingehalten werden.

$minLength$ s.o., die Zahl der Zeichen, Bytes bzw. Listenelemente muß mindestens dem Wert von $minLength$ sein.

$maxLength$ s.o., die Zahl der Zeichen, Bytes bzw. Listenelemente darf maximal gleich dem Wert von $maxLength$ sein.

$pattern$ Ein gewisser regulärer Ausdruck[6] beschränkt den lexikalischen Raum – und somit den Werteraum – des Basistypes.

$enumeration$ Der Wertraum des Basistypes wird auf die angegebene endliche Teilmenge beschränkt.

$whiteSpace$ Gem. XML-Spezifikation [11] gibt es verschiedene Möglichkeiten mit Leerzeichen[7] zu verfahren. Diese Facette ist wichtig für den Datentyp $string$ und davon abgeleitete.

$maxInclusive$ rechte Intervallgrenze[8] für geschlossenes Intervall

$maxExclusive$ dto. für offenes Intervall

$minInclusive$ linke Intervallgrenze für geschlossenes Intervall

$maxExclusive$ dto. für offenes Intervall

$totalDigits$ Gesamtzahl der Ziffern für $decimal$ und abgeleitete Typen

$fractionDigits$ Zahl der Nachkommaziffern für $decimal$ und abgeleitete Typen

Welche der nicht-fundamentalen Facetten jeweils anwendbar sind und welche sich gegenseitig ausschließen (z.B. $length$ und $minLength$) bzw. welche Bedingungen bzgl. der Werte der Facetten gelten, entnehme man [9].

Wir haben in J konkrete Werte aus dem lexikalischen Raum vorliegen, bei vielen Datentypen gilt hier aber eine Bijektion zum Werteraum. Die Typhierachie ist nicht ganz klar, zum einen kann sich die XML-Schema-Spezifikation nicht entscheiden, ob $anySimpleType$ existiert oder

[6] in Perl-Notation, Definition in [9]
[7] Hiermit ist nicht nur das einfach Leerzeichen #x20 sondern auch #x9, #xD, #xA gemeint.
[8] $(max|min)(In|Ex)clusive$ sind natürlich nur für Datentypen mit einer definierten Ordnung

nur virtuell vorhanden ist, weiterhin gilt natürlich bei der Vererbung mit Restriktion die Teilmengenbeziehung, d.h. z.B. ist $L(integer) \subset L(decimal)$, andererseits tritt in dieser Hierachie der Fall auf, daß solche Teilmengenbeziehungen auch zwischen benachbarten Typen gelten, die nicht in einer Vererbungsbeziehung stehen.

Konkret umfaßt die Wertemenge für $string$ die Wertemenge *aller* anderer Typen und z.B. stehen $float$ und $double$ in keiner Vererbungsbeziehung. Weiterhin ist der Typ $token$ eine Folge von Teilworten, die mit einem einzelnen Leerzeichen getrennt sind aber alle abgeleiteten Typen enthalten keine Leerzeichen mehr, nur um später aus NMTOKEN, IDREF und ENTITY wieder Listen abzuleiten.

Betrachten wir zunächst den Fall, daß wir keine nutzerdefinierten Typen zuordnen wollen. Die Funktion $checkType()$ sei analog wie bei den DTDs definiert, nur daß wir hier gleich alle Typen prüfen und in der Menge $possibleTypes()$ sammeln.

Für den Typ ID können wir genauso wie bei den DTDs verfahren, wobei generell zu beachten ist, daß die Typen ID, IDREF, IDREFS, NOTATION, NMTOKEN, NMTOKENS, ENTITY und ENTITIES *nur* bei Attributen benutzt werden sollen. Es ist besser, wenn es um einen Elementtyp geht, diese Typen bei der Betrachtung gleich außen vor zu lassen. Nachfolgend setzen wir voraus, daß ggf. die Typen ID, IDREF und IDREFS schon erkannt wurden.

Den Typ $string$ können wir immer zuordnen (analog CDATA bei den DTDs), einem möglichen spezielleren Typ wird immer der Vorzug gegeben. Generell ist ein Typ immer spezieller als sein Basistyp, bezeichnet mit $\prec$, hiermit haben wir aber nur eine Halbordnung bzgl. $\prec$ gegeben, wir werden für die anderen wichtigen Fälle $\prec$ definieren[9]. Wir definieren also: jeder Typ ist spezieller als $string$.

Betrachtet man die lexikalische Repräsentation genauer, so erkennt man, daß die Typen $duration, dateTime, time, date, gYearMonth, gYear,$
$gMonthDay, gDay, gMonth, boolean$ paarweise keinen Wert mit gleicher lexikalischer Repräsentation haben, somit können keine zwei aus diesen gemeinsam in $possibleTypes(J)$ sein.

Wir nehmen jetzt sukzessive die anderen Typen hinzu und geben Heuristiken an, wie zu verfahren ist. Natürlich fallen bei der konkreten Prüfung alle die Typen heraus, welche nicht in $possibleTypes(J)$ sind.

$decimal$ und Ableitungen

Es gilt: $L(boolean) \cap L(decimal) = \{0, 1\}$, wir definieren $boolean \prec decimal$ Weiterhin hat $decimal$ mit $gYear$ Werte gemeinsam, hier soll $gYear$ als Typ gewählt werden, wenn alle auftretenden Werte mit den Ziffern 19 oder 20 beginnen, ansonsten eine $decimal$-Ableitung.

Im Teilbaum unter $decimal$ in der Typ-Hierarchie soll der speziellste möglichste Typ gewählt werden, wobei natürlich alle Typen spezieller als ihre Basistypen sind und weiterhin soll gelten: $nonPositiveInteger$ bzw. $nonNegativeInteger$ sind spezieller als $long$ und $unsignedLong \prec positiveInteger$.

Wir erweitern an dieser Stelle transitiv: Ist A von $decimal$ abgeleitet und gilt $decimal \prec B$ (bzw. $B \prec decimal$) dann gilt auch $A \prec B$ (bzw. $B \prec A$).

$float$ und $double$

Es soll gelten: $float \prec double$, da $L(float) \subset L(double)$.
Weiterhin gilt: $L(float) \cap L(decimal) \neq \emptyset$, und dieser Schnitt enthält sogar die gleichen Werte,

[9] Beachte, daß wir eine Halbordnung erweitern wollen, also sich gewisse Verhältnisse von Typen zueinander durch die Transitivität ergeben. Ist z.B. $decimal \prec float$, so sind auch alle $decimal$-Ableitungen $\prec float$.

die bei $decimal$ zur Überschneidung mit $boolean$ und $gYear$ führten. An dieser Stelle definieren wir: $decimal \prec float$.

QName

Der Typ $QName$[10] steht für Namen, die optional mit einem Namensraum qualifiziert sind, es wäre sinnvoller, diesen Typ unterhalb von $Name$ neben $NCName$[11] anzuordnen. Klarerweise definieren wir dann $QName \prec Name$ und $NCName \prec QName$.

$\mathbf{anyURI}, \mathbf{base64Binary}$ und $\mathbf{hexBinary}$

Da $anyURI, base64Binary$ und $hexBinary$ im Normalfall sehr spezielle Zeichenketten als Wert hat, und nur in Ausnahmefällen alle Werte eines Attributes dieser Typen [12] auch einen anderen Typ (außer $string$) haben können, definieren wir, $anyURI, base64Binary$ und $hexBinary$ sind spezieller als $duration$, $dateTime$, $time$, $date$, $gYearMonth$, $gYear$, $gMonthDay$, $gDay$, $gMonth$, $boolean$, $decimal$, $float$, $QName$, $NOTATION$

$\mathbf{NOTATION}, \mathbf{ENTITY}, \mathbf{ENTITIES}$

Hier gilt das bei DTDs gesagte, solche Typen werden prinzipiell nicht erkannt.

$\mathbf{string}$ und Ableitungen

Da ID, IDREF und IDREFS ggf. schon erkannt wurden und die eben genannten Typen nicht erkannt werden, brauchen wir nur noch definieren:
$language \prec Name$ und $Name \prec NMTOKEN$.

Weiterhin gilt: $NMTOKEN \prec NMTOKENS$ und $ENTITY \prec ENTITIES$, da wir so mehr Semantik wiedergeben.

Auch hier erweitern wir transitiv wie bei $decimal$ mit Ausnahme des Typs $QName$: Ist A von $string$ abgeleitet und gilt $string \prec B$ (bzw. $B \prec string$) dann gilt auch $A \prec B$ (bzw. $B \prec A$).

4.2.3 Vorgehen

Zunächst berechnen wir für J und für jeden Datentyp den Wert von $checkType()$, hierbei brauchen wir den Datentyp $string$ nicht zu betrachten, weiterhin gilt $checkType(B, J) \Rightarrow checkType(A, J)$, wenn B mittels Restriktion aus A oder A mittels Liste aus B abgeleitet wurde. Somit ersparen wir uns einige Tests.

Hieraus erhalten wir dann die Menge $possibleTypes(a)$ und der bzgl. $\prec$ kleinste Typ ist der Typ, den wir dem Attribut zuordnen.

Beispiel:
Sei $J = \{23, 45, 3.45, 8.9\}$, dann gilt:
$possibleTypes(J) = \{string, float, double, decimal, normalizedString, token\}$
und: $decimal \prec float \prec double \prec token \prec normalizedString \prec string$, also wählen wir als Typ $decimal$.

[10]vgl. [10], `Qname ::= (NCName ':')? NCName`
[11]vgl. [10], $Name$ gem. XML-Spezifikation ohne ‚:‘
[12]Beachte, daß wir gleich eine ganze *Menge* an Werten - nämlich J - betrachten.

4.2.4 Ableitungen von Datentypen

Bei XML-Schema haben wir die Möglichkeit, eigene Datentypen abzuleiten, obiges Vorgehen betraf ja nur die eingebauten Datentypen.

Diese Thematik kann an dieser Stelle nicht erschöpfend diskutiert werden. Vielmehr werden einige Ansatzpunkte für weitere Überlegungen vorgestellt.

Listen

Viele Datentypen erlauben keine Leerzeichen, so daß eine Liste von $duration$ nach obigem Vorgehen als $string$ bzw. eine $string$-Ableitung erkannt würde.

Als Vorgehensweise bietet sich an, nach Leerzeichen in den Werten zu suchen und festzustellen, ob es einen Datentyp A gibt, so daß die Teilworte zwischen den Leerzeichen immer zu diesem gehören.

In diesem Fall könnten wir folgenden Typ neu definieren:

```
<simpleType name="neu">
  <list itemType="A"/>
</simpleType>
```

Komplizierter wird es, wenn wir Listen aus nicht-vordefinierten Typen erkennen wollen.

Vereinigungen

Normalerweise nehmen Typen A und B an einer Vereinigung teil, für die keine Teilmengenbeziehung $A \subseteq B$ oder $B \subseteq A$ gilt. Das führt nach obigem Vorgehen dazu, daß der speziellste gemeinsame (evt. indirekte) Basistyp bzw. $string$ zugeordnet wird. Z.B. $positiveInteger \cup negativeInteger$ führt zum Erkennen von z.B. $integer$.

Um so etwas zu erkennen, können wir die Funktion $checkType()$ so umdefinieren, daß sie nicht $wahr$ bzw. $falsch$ liefert, sondern ein Wert aus [0,1], der die relative Häufigkeit der Typzugehörigkeit für Elemente aus J angibt.

Ist $checkType()$ hinreichend groß, so versuchen wir, eine Überdeckung aus verschiedenen Typen für J zu finden, wobei wir weiterhin darauf achten, nicht zuviele Typen (maximal 3) an der Vereinigung teilnehmen zu lassen.

Auch hier gibt es Schwierigkeiten, wenn wir auch Vereinigungen aus nicht-vordefinierten Typen erkennen wollen.

Aufzählungen

Bei Aufzählungen ist die Frage, ob wir nicht zu starke Restriktionen erzeugen. Deshalb sei hier als Kriterium das gleiche wie bei DTDs angewandt und weiterhin ist eine Aufzählung nur bei $string$, $integer$ und den Ableitungen davon wahrscheinlich.

Einschränkungen bei Zahlen, Listen und Strings

Einschränkungen bzgl. den Facetten $length$, $minLength$, $maxLength$, $pattern$, $(min|max)(In|Ex)clusive$, $totalDigits$ und $fractionDigits$ bergen ebenfalls die Gefahr in sich, zu starke Restriktionen abzuleiten.

Hier müßten noch weiterführende Heuristiken entwickelt oder der Nutzer gefragt werden. Insbesondere die Erkennung der richtigen Generalisierung von $pattern$, welches die Einschränkung mit einem (speziellen) regulären Ausdruck darstellt, ist von der Schwierigkeit mit der in Kapitel 3 diskutierten Ableitung für das Inhaltsmodell vergleichbar.

4.3 Qualifikatoren

Die Möglichkeiten für die Qualifikatoren bei einer Attributdeklaration unterscheiden sich bei DTDs und XML-Schemata[13] nur syntaktisch, weshalb sie hier nicht getrennt diskutiert werden.

4.3.1 Konstanten

Wir können einen Attributwert als *konstant* definieren, hierfür dient bei DTD `#FIXED +` $Wert$ und bei XML-Schema die Angabe `fixed=`$Wert$ in der Attributdeklaration[14]. Bei DTDs muß ein `#FIXED`-Attribut nicht notwendig vorhanden sein, bei XML-Schema wird das über `use=`$Wert$ bestimmt, wobei:
$Wert \in$ {`"optional", "prohibited", "required"`}.

Beispiel bei DTDs:

```
<!ATTLIST e
  a CDATA #FIXED "konstanter Text">
```

Beispiel bei XML-Schema:

```
<attribute name="a" type="string" fixed="konstanter Text" />
```

Es gilt: Wenn $|J| > 1$ dann kann das Attribut keine Konstante sein. Andererseits spricht $|J| = 1$ mit großer Wahrscheinlichkeit für ein konstantes Attribut, weshalb wir in diesem Fall als Qualifikator *konstant* wählen. Bei XML-Schema setzen wir zusätzlich `use="required"` falls das Attribut immer vorhanden ist, anderenfalls `use="optional"`.

Beachte, daß Attribute vom Typ ID nicht konstant sein dürfen!

4.3.2 Optionale Attribute ohne Default-Wert

Bei DTDs wird ein Attribut mit `#IMPLIED` qualifiziert, wenn es nicht konstant, optional und ohne Default-Wert ist, bei XML-Schema geschieht das über die Angabe `use="optional"` wobei dann `fixed` und `default` nicht vorhanden sein dürfen.

Beispiel bei DTDs:

```
<!ATTLIST e
  a CDATA #IMPLIED>
```

Beispiel bei XML-Schema[15]:

```
<attribute name="a" use="optional" />
```

Attribute können *immer* auf diese Art und Weise qualifiziert sein.

[13] `use="prohibited"` ist für uns nicht interessant
[14] Wenn `fixed=`$Wert$ in der Attributdeklaration vorhanden ist, darf nicht gleichzeitig `default=`$Wert$ vorhanden sein und umgekehrt
[15] `use="optional"` ist auch der Default-Wert

4.3.3 Notwendige Attribute ohne Default-Wert

Bei DTDs wird ein Attribut mit `#REQUIRED` qualifiziert, wenn es nicht konstant, optional und ohne Default-Wert ist, bei XML-Schema geschieht das über die Angabe `use="required"` wobei dann `fixed` und `default` nicht vorhanden sein dürfen.

Beispiel bei DTDs:

```
<!ATTLIST e
  a CDATA #REQUIRED>
```

Beispiel bei XML-Schema:

```
<attribute name="a" use="required" />
```

Es muß hier gelten, daß jedes e_typ-Element ein a-Attribut besitzt.

4.3.4 Default-Werte

Bei DTDs und XML-Schema können optionale Attribute (und nur diese) mit einem Default-Wert versehen sein. Hierfür fehlt bei DTDs die Angabe von `#FIXED`, `#IMPLIED` oder `#REQUIRED` und bei XML-Schema ist `default=`$Wert$ vorhanden, wobei `use="optional"` gelten muß.

Beispiel bei DTDs:

```
<!ATTLIST e
  a CDATA "Default-Text">
```

Beispiel bei XML-Schema:

```
<attribute name="a" use="optional" default="Default-Text" />
```

Es ist nicht in jedem Fall zu erwarten, einen Default-Wert erschließen zu können, denn wenn der Nutzer des Schemas diesen kennt, wird er diesen vielleicht niemals explizit angeben.

Wurden die XML-Dokumente aber automatisch generiert und evt. sogar schon normalisiert, dann tritt der Default-Wert durchaus in den Instanzen auf.

4.3.5 Vorgehen

Wenn möglich ordnen wir den Qualifikator *konstant* zu.

Ansonsten ist ein Wert aus J, der bei vielen Instanzen auftritt, ein Indiz für einen Default-Wert, falls die Dokumente automatisch generiert wurden. Gilt (immer bezogen auf e_typ-Elemente):

$$\exists w \in J : \frac{\text{Anzahl der } a\text{-Attribute mit } Wert(a) = w}{\text{Anzahl der vorhandenen } a\text{-Attribute}} \geq p$$

für ein gewähltes $p \in [0, 1]$, so ist als Default-Wert w anzunehmen und natürlich muß das Attribut optional sein.
Anderenfalls wird kein Default-Wert angenommen und je nach dem, ob das Attribut immer vorhanden ist oder nicht, wird das Attribut mit *notwendig* bzw. *optional* qualifiziert.

Alternativ kann auch der Nutzer nach einem Default-Wert gefragt werden.

Kapitel 5

Integritätsbedingungen

5.1 Überblick

Zusätzlich zu den bereits diskutierten Thematiken des Inhaltsmodells und der Datentypen können in einem XML-Dokument bzw. in einer Menge von XML-Dokumenten noch weitere Integritätsbedingungen gelten.

Begrifflich haben wir bisher Datentypen im Sinne von XML-Schema getrennt in die Inhaltsmodelle, die (einfachen) Datentypen für Stringderivate und ggf. den Elementen zugeordnete Attributmengen. In gewisser Hinsicht sind natürlich schon die Angabe von Inhaltsmodellen, vorhandenen Attributen und die Angabe von Datentypen für Attribute bzw. auch Stringinhalt von Elementen Integritätsbedingungen an ein wohlgeformtes XML-Dokument.

Es macht jedoch Sinn, hier eine Trennung zwischen Strukturteil und (expliziten) Integritätsbedingungen durchzuführen, was auch beim relationalen Datenmodell und bei objektorientierten Datenbankmodellen so praktiziert wird. Anschaulich bedeutet dies, daß wir beim relationalen Modell durch die Angabe des (einfachen) Relationenschemas oder (einfachen) Datenbankschemas zunächst die Grundmenge an möglichen Instanzen aufspannen, auf welchen dann die lokalen bzw. globalen Integritätsbedingungen als Abbildung von Relationen bzw. Datenbanken nach „wahr“ oder „falsch“ definiert sind.

Auch bei attributierten Grammatiken kann so vorgegangen werden, eine kontextfreie Grammatik gibt eine kontextfreie Sprache als Grundlage vor, letztendlich kann ein Ableitungsbaum dieser grundlegenden Grammatik dennoch über die Berechnung der Attribute als nicht zulässig verworfen werden, wenn man damit z.B. eine Sprache darstellen will, die in Wirklichkeit keine kontextfreie Sprache ist.

Beachte auch, daß Integritätsbedingungen aus der „realen Welt“ bei der Modellierung mit verschiedenen Datenmodellen nicht unbedingt bei allen als explizite Integritätsbedingungen wieder auftauchen: Haben wir z.B. ein Objekt mit einem Komponentenobjekt in der Realität, so können wir diesen Sachverhalt mit einem objektorientierten Datenbankmodell meist ziemlich direkt modellieren, beim relationalen Modell hingegen wird das Komponentenobjekt oftmals in einer anderen Relation dargestellt und wir müssen zur Darstellung des Zusammenhangs unter anderem eine Inklusionsabhängkeit einführen.

Was ist bei XML-Dokumenten bzw. Mengen von XML-Dokumenten (zu einem XML-Schema bzw. einer DTD) ein sinnvoller Definitionsbereich für Integritätsbedingungen?

Beim relationalen Modell unterscheiden wir zwischen lokalen und globalen Integritätsbedingungen. Lokale Integritätsbedingungen sollen hierbei innerhalb einer Relation gelten, globale hingegen über einem Datenbankschema, d.h. sie drücken Bedingungen zwischen Relationen aus.

Naheliegend sind folgende drei Möglichkeiten des Definitionsbereiches für Integritätsbedingungen:

- Menge von XML-Dokumenten

  Stellt unser XML-Schema bzw. unsere DTD z.B. einen Studenten dar, der z.B. ein Attribut $Immatrikulationsnummer$ besitzt, so könnte eine sinnvolle Integritätsbedingung die Festlegung sein, daß die $Immatrikulationsnummer$ ein *Schlüssel* für einen Studenten ist. Da jeder Student zu genau einer Datei innerhalb der (abgegrenzten) Menge korrespondiert, muß diese Integritätsbedingung natürlich überhalb einer *Menge* von XML-Dokumenten definiert sein. Ein XML-Dokument übernimmt hier quasi die Rolle eines Tupels (bzw. Objektes) in einer Relation (bzw. Extension), welche wiederum durch die Menge aller XML-Dokumente charakterisiert wird.

- Ein XML-Dokument

  Betrachten wir eben genanntes Studentenbeispiel, so können wir natürlich ebensogut eine DTD bzw. ein XML-Schema für die ganze Studentenschaft definieren, welche obige Studentendefinition beinhaltet, als Wurzelelement aber z.B. in DTD-Syntax das Element Studentenschaft besitzt:
  `<!ELEMENT Studentenschaft (Student)*>`
  Hierbei entspricht dann praktisch ein XML-Dokument einer Relation (bzw. Extension) und alle Kind-Elemente des Wurzelelementes entsprechen den Tupeln (bzw. Objekten). Wollen wir hier die Schlüsseleigenschaft der $Immatrikulationsnummer$ mit einer Integritätsbedingung darstellen, so ist es sinnvoll, diese als Abbildung eines einzigen XML-Dokumentes nach „wahr“ oder „falsch“ zu spezifizieren.

- Teile eines XML-Dokumentes

  Eine Integritätsbedingung muß natürlich nicht zwingend über allen Elementen eines bestimmten Types innerhalb einer XML-Datei sinnvoll sein. Es könnte unter Umständen sinnvoll sein, diese Menge einzuschränken, z.B. mit Pfadangaben innerhalb des XML-trees. Studenten können (mehrere) Hobbies haben, ein Student hat aber kein Hobby zweimal, verschiedene Studenten können aber das selbe Hobby haben. Klarerweise wäre hier der Definitionsbereich einer Integritätsbedingung ein einzelner Student.

Später werden wir noch sehen, daß uns XML-Schema für die beiden letztgenannten Punkte Konstrukte zur Verfügung stellt, wobei diese beiden Möglichkeiten praktisch zusammenfallen, da die Selektion der relevanten Elementmenge über einen eingeschränkten XPath-Ausdruck erfolgt, welcher uns natürlich auch den Zugriff auf alle Elemente eines bestimmten Types erlaubt.

Sowohl DTDs als auch XML-Schema stellen keine Vorkehrungen zur Verfügung, Bedingungen über ein Dokument hinaus zu definieren, sieht man von dem simplen Inkludieren eines XML-Dokumentes in ein anderes hinein ab. An dieser Stelle sollen deshalb solche globalen XML-Integritätsbedingungen nicht betrachtet werden, sie machen aber trotzdem z.B. an der Stelle Sinn, wenn sie in einem Datenbanksystem definiert werden, welches XML-Dokumente verwaltet.

Allgemein gilt für uns deshalb jetzt, eine Integritätsbedingung I ist eine Abbildung der Form:

$I : WF \rightarrow \{wahr, falsch\}$

wobei WF die Menge aller wohlgeformten XML-Dokumente sei.

Häufige gebrauchte Arten von Integritätsbedingungen sind:

- Schlüsselbedingungen

  In unserem Kontext bedeutet dies, ein bestimmtes Element wird durch bestimmte Informationen identifiziert und diese Menge an Informationen ist in gewisser Hinsicht minimal. Für diese Informationen kommen hier Attribute, Stringinhalte, Kindelemente und natürlich Kombinationen davon in Betracht.

  Die Minimalität ist bei Schlüsseln so zu verstehen, daß es Instanzen gibt, welche die Schlüsselbedingung erfüllen, nicht aber die Schlüsselbedingung für eine bestimmte echte Teilmenge des Schlüssels. Für eine konkrete Instanz wird die Minimalität klarerweise nicht gefordert, denn z.B. ist im Relationenmodell bei einer Relation, welche nur ein einziges Tupel enthält, jedes einzelne Attribut ein Schlüssel (dieser Relation), wir möchten aber natürlich auch ausdrücken können, daß solche Relationen auch komplexere Schlüsselbedingungen erfüllen.

- Funktionale Abhängigkeiten

  Funktionale Abhängigkeiten im Kontext des relationalen Modells bedeuten anschaulich, daß sich der Wert bestimmter Attribute innerhalb einer Relation funktional aus Werten bestimmter anderer Attribute ergibt. Ohne formale Einführung des Relationenmodells sei R die Attributmenge einer Relation, $X, Y \subseteq R$.
  Eine Relation r über R erfüllt die funktionale Abhängigkeit $X \rightarrow Y$ gdw.
  $\forall t_1, t_2 \in r : t_1(X) = t_2(X) \Rightarrow t_1(Y) = t_2(Y)$
  Schlüsselbedingungen sind also Spezialfälle von funktionalen Abhängigkeiten in dem Sinne, daß $Y = R$ gilt und X inklusionsminimal ist.

- Inklusionsabhängigkeiten

  Eine bestimmte Menge von Objekten ist auch irgendwo anders enthalten. Das Duale dazu sind Exklusionsbedingungen.

- Fremdschlüssel

  Bestimmte Informationen, also hier eine Kombination aus Attributen, Stringinhalten und Kindelementen eines Elementes, sind irgendwo anders enthalten und dort auch Schlüssel. Fremdschlüssel sind praktisch die Kombination einer Schlüsselbedingung und einer Inklusionsabhängigkeit

- Disjunktheits- und Überdeckungsbedingungen

  Wie die Bezeichnungen schon implizieren, soll eine Disjunktheitsbedingung dafür sorgen, daß gewisse Mengen von Dingen paarweise disjunkt sind, bzw. die Überdeckungsbedingung sicherstellen, daß die Vereinigung gewisser Mengen eine Obermenge einer bestimmten anderen Menge ist.

- Kardinalitäten

  Kardinalitätsbedingungen geben Anzahlbeschränkungen vor. In unserem Fall werden wir sie für die Verfeinerungen der implizit im Strukturteil enthaltenen Kardinalitäten gebrauchen, da bisher im Inhaltsmodell nur Kardinalitäten wie „0 oder 1“, „genau 1“, „genau k“, „mindestens 1“, „beliebig viele“ und ähnliches enthalten ist.

Beim Ableiten von Integritätsbedingungen aus einer Beispielmenge haben wir es mit dreiwertiger Logik zu tun: Die Beispielmenge impliziert die Gültigkeit gewisser Integritätsbedingungen, sie impliziert die Ungültigkeit gewisser anderer Integritätsbedingungen und es gibt Integritätsbedingungen, deren Gültigkeit aus der Beispielmenge nicht abgeleitet werden kann, denn die Beispielmenge ist i.allg. nicht vollständig.

In Arbeiten zum Reengineering von Datenbanken wie [21] wird auf die Nützlichkeit von negierten Bedingungen der oben genannten häufig auftretenden Bedingungen hingewiesen. Diese lassen sich oft leichter berechnen und haben den Vorteil, daß sie auch ohne Closed-World-Assumption gültig sind.

Die Closed-World-Assumption ist die Annahme, daß alle Integritätsbedingungen, die auf der Datenmenge gelten, die wir zur Ableitung von Integritätsbedingungen zur Verfügung haben, generell gelten und umgekehrt. Ist zum Beispiel die $Personalausweisnummer$ ein Schlüssel auf 100 Beispieldatensätzen, so besagt die Closed-World-Assumption, daß $Personalausweisnummer$ auch Schlüssel auf allen Personendatensätzen ist, alle geltenden Integritätsbedingungen lassen sich also schon aus der Beispieldatenmenge ableiten. Unter der Closed-World-Assumption läßt sich also für jede Integritätsbedingung die Gültigkeit oder Nichtgültigkeit ableiten.

Diese Closed-World-Assumption ist sinnvoll, um dem Anwender gewisse Integritätsbedingungen zu präsentieren, die für die übergebene Menge gelten und den Anwender entscheiden zu lassen, ob sie auch für die Generalisierung gelten sollen, denn beim Schemaerkennen führen wir ja genau eine solche Generalisierung durch.

Gilt diese Annahme nicht, so lassen sich generell z.B. keine Schlüsselbedingungen ableiten, denn ist eine Attributmenge X Schlüssel für die Beipielrelation $r(R)$, so können wir aber noch weitere Tupel so hinzufügen, daß X kein Schlüssel mehr ist. Analog könnte ein Schlüssel für unsere XML-Dokument-Menge kein Schlüssel für eine Obermenge selbiger sein.

Gehen wir zu den Negationen einiger bekannter Integritätsbedingungen über, wie z.B. negierten Schlüsseln oder negierten funktionalen Abhängigkeiten, so zeigt sich, daß diese invariant sind gegen Obermengenbildung. Somit können wir diese auch bei fehlender Closed-World-Assumption aus unserer gegebenen XML-Dokumentmenge ableiten.

Definition:
Eine Integritätsbedingung $I : D \to \{wahr, falsch\}$ heißt **invariant gegen Obermengenbildung** (bzw. **Teilmengenbildung**) :gdw.
$\forall d, d' \in D \, mit \, d \subseteq d' : I(d) = „wahr“ \Rightarrow I(d') = „wahr“ \, (bzw. \, I(d') \Rightarrow I(d))$

In [21] werden die Integritätsbedingungen, welche aus der Negation der „Standardintegritätsbedingungen“ – vgl. die Liste oben, aber ohne Anspruch auf Vollständigkeit – auch als *negierte* oder *negative Integritätsbedingungen* bezeichnet. Beachte, daß I nicht notwendig eine von beiden Eigenschaften hat und sich beide Eigenschaften auch nicht ausschließen. Des weiteren sei bemerkt, daß Integritätsbedingungen immer auf Mengen von Dingen definiert sind, auch wenn sie nur ein einziges XML-Dokument nach „wahr“ oder „falsch“ abbilden, so wird innerhalb dieses Dokumentes doch i.d.R. eine Menge von Elementen (oder ähnliches) berücksichtigt.

Nur bestimmte Arten von Integritätsbedingungen sind für uns aber interessant (das sind hauptsächlich die genannten) und von diesen sind wiederum nicht notwendig alle mit unserem Datenmodell darstellbar.

Niemals möglich ist z.B. die Darstellung nichtberechenbarer Integritätsbedingungen, es sei hier einfach einmal angenommen, als Inhalt eines Elementes ist eine Turingmaschine kodiert und die Integritätsbedingung soll genau dann „wahr“ sein, wenn das spezielle Halteproblem für diese Turingmaschine „wahr“ ist, welches bekanntermaßen nicht-entscheidbar ist. Oft nicht möglich ist überhaupt eine Algorithmusangabe, wie z.B. eine Integritätsbedingung für Kardinalitäten, welche nur Primzahlen zuläßt.

Aus diesem Grund interessieren wir uns hier auch ausschließlich für Integritätsbedingungen, welche wir darstellen können:

5.2 Möglichkeiten bei DTDs

Die Möglichkeiten bei DTDs sind sehr eingeschränkt. Elemente können maximal ein Attribut vom Typ `ID` besitzen, der Wert eines Attributes vom Typ `ID` ist im gesamten Dokument und unabhängig von dem Elementtyp dieses Attributes eindeutig. Beachte, daß bei DTDs eine Bijektion zwischen Elementnamen und Elementtypen besteht. Attribute vom Typ `IDREF` bzw. `IDREFS` verweisen auf ein anderes Element (bzw. andere Elemente) im gleichen Dokument. Praktisch können wir folgende Integritätsbedingungen darstellen:

Durch die Definition eines Attributes vom Typ `ID` z.B. beim Element Studenten wird sichergestellt, daß *dieses* Attribut ein Schlüssel von Studenten ist. Wir können also nicht ein vorhandenes Attribut $Matrikelnummer$ als Schlüssel explizit kennzeichnen, die einzige Möglichkeit besteht eben darin, gerade $Matrikelnummer$ vom Typ `ID` zu wählen.

Die Probleme sind, daß wir nicht mehr als einen Schlüssel pro Elementtyp definieren können – es darf laut XML Spezifikation nur maximal ein Attribut vom Typ `ID` geben – und daß dieser Attributwert über alle Attribute vom Typ `ID` innerhalb dieses XML-Dokumentes eindeutig sein muß, was Probleme aufwirft, wenn z.B. ein Auto-Element ein Kennzeichen hat, was gleich einer $Matrikelnummer$ ist.

Auch bei den Verweisen mittels `IDREF` bzw. `IDREFS` von einem anderen Element können wir keine Restriktion erzwingen, daß auf ein Student-Element verwiesen wird, genausogut könnte eine `IDREFS`-Referenzliste aus einem Studentenschaft-Element auf Studenten und Autos gemischt verweisen, was nicht unbedingt sinnig ist. Das Einzige was sichergestellt ist, ist die Tatsache, daß sich die referenzierten Elemente auch in diesem XML-Dokument befinden.

`ID`-Attribute taugen also im allgemeinen nur als Surrogate für die Darstellung der Elementidentität nicht aber zur vollwertigen Darstellung von Schlüsselabhängigkeiten.

Wenn wir also die Attributtypen richtig erkannt haben, wobei wir für `ID`, `IDREF` und `IDREFS` die eben genannten Aspekte berücksichtigen müssen, ist es nicht sinnvoll zu versuchen, weitere Integritätsbedingungen abzuleiten, da wir sie mit einer DTD sowieso nicht darstellen können.

5.3 Möglichkeiten bei XML-Schema

XML-Schema beinhaltet zunächst aus Kompatibilitätsgründen die eben genannten Möglichkeiten bei den DTDs, favorisiert aber die neu hinzugekommenden Konstrukte für die Darstellung von Integritätsbedingungen, Beispiele wurden teilweise aus [13] übernommen, dort finden sich dann auch die vollständigen Beispiele.

Beachte, daß diese Integritätsbedingungen nicht einem Typ, sondern einer Elementdeklaration zugeordnet sind, vgl. [31].

Im einzelnen offeriert XML-Schema folgende Möglichkeiten:

- Eindeutigkeit (engl. Uniqueness)

  Wir definieren mit `selector` den Definitionsbereich der Eindeutigkeit über einen (eingeschränkten) XPath-Ausdruck relativ zu Instanzen der Elementdeklaration, für die wir die Eindeutigkeit definieren. Der Definitionsbereich muß eine Menge untergeordneter Elemente dieser Instanz sein, d.h. Kindelemente oder Kindelemente von Kindelementen usw. Relativ zu diesen bestimmen wir mit `field` Attribute oder Elemente mit Stringinhalt, wobei die Auswertung des XPath-Ausdruckes jedes `field`-Elementes bezogen auf einen Knoten des Definitionsbereiches höchstens einen (Attribut- bzw. Element-)Knoten liefern darf.

  Beispiel aus [13]:

```
<unique name="dummy1">
  <selector xpath="r:regions/r:zip"/>
  <field xpath="@code"/>
  <field xpath="r:part/@number"/>
</unique>
```

Für ein Element, dessen Deklaration obige Eindeutigkeitsbedingung enthält, wird als Definitionsbereich dieser Bedingung der XPath `r:regions/r:zip` ausgewertet, das sind alle `zip`-Kindelemente aller `regions`-Kindelemente des aktuellen Elementes. Für jedes Element innerhalb des Definitionsbereiches wird praktisch ein Tupel mit den angegebenen `field`-Elementen gebildet, es darf nur maximal ein `code`-Attribut geben (ist laut XML-Spezifikation auch gar nicht anders möglich) aber auch `part/@number` darf nur maximal ein `number`-Attribut liefern.

Jedes Tupel ist automatisch getypt – das ergibt sich aus den Typen der entsprechenden Attribute oder Elemente mit Stringinhalt – allerdings hat nicht jedes Tupel zwingend den gleichen Typ. Der Vergleich zweier Werte von Tupelkomponenten erfolgt auch unter Beachtung des Types, d.h. ist der Typ jeweils eine Zahl, so ist $3.0 = 3$, ist der Typ hingegen jeweils eine einfache Zeichenkette, so ist $3.0 \neq 3$.

Die Eindeutigkeitsbedingung stellt sicher, daß wenn p und q Elemente des Definitionsbereiches sind, für die weiterhin gilt, daß jedes `field`-Element zu einer nicht-leeren Menge evaluiert wird, daß die Tupel zu p und q ungleich sind bzgl. der Gleichheitsdefinition von XML-Schema, welche die Typen der Tupelelemente berücksichtigt.

- Schlüssel

  Eine Schlüsselbedingung unterscheidet sich nur durch ein Detail von einer Eindeutigkeitsbedingung, nämlich der Forderung, daß die Auswertung jedes `field`-Element eine nicht-leere Menge liefert:

  Die Schlüsselbedingung stellt sicher, daß wenn p und q Elemente des Definitionsbereiches sind, daß jedes `field`-Element zu einer nicht-leeren Menge evaluiert wird und daß die Tupel zu p und q ungleich sind bzgl. der Gleichheitsdefinition von XML-Schema, welche die Typen der Tupelelemente berücksichtigt.

  Beispiel aus [13]:

```
<key name="pNumKey">
  <selector xpath="r:parts/r:part"/>
  <field xpath="@number"/>
</key>
```

- Fremdschlüssel

  Syntaktisch sind Fremdschlüsselbedingungen ähnlich den oben genannten, zusätzlich wird ein andersweitig definierter Schlüssel referenziert. Sei p ein Element des Definitionsbereiches der Fremdschlüsselbedingung, für welches alle `field`-Elemente zu nicht-leeren Mengen evaluiert werden, dann stellt die Fremdschlüsselbedingung sicher, daß das Tupel t zu p (Vorgehensweise hier wie bei Schlüssel- und Eindeutigkeitsbedingung) auch in der Menge der Tupel des referenzierten Schlüssel (bzgl. des aktuellen Elementes) auftritt, hierzu auch die nachfolgenden Erläuterungen:

  Die Erläuterungen in der Spezifikation sind zwar seit der Version vom 2.Mai 2001 konsistent und vollständig, aber nicht leicht verständlich, deshalb an dieser Stelle eine informale Semantikbeschreibung:

Gem. [31]; Absätze 3.11.4 und 3.11.5, wird die Fremdschlüsselbedingung bei der Validierung von Elementen (aus einem XML-Dokument) geprüft, in deren zugehöriger Elementdeklaration der Fremdschlüssel $keyref$ definiert ist. Sei das aktuell zu validierende Element mit e_v bezeichnet, die zugehörige[1] Elementdeklaration mit $e_{keyref}decl$. Die Fremdschlüsselbedingung referenziert einen Schlüssel key, der Namensraum für Integritätsbedingungen ist schemaweit eindeutig, d.h. es gibt genau eine Elementdeklaration, die den referenzierten Schlüssel enthält, bezeichnet mit $e_{key}decl$. Ausgehend von e_v wird analog den vorgestellten Schlüssel- und Eindeutigkeitsbedingungen der Definitionsbereich gebildet und die jeweiligen Tupel zugeordnet.

e_v bekommt in seinem *infoset* eine *identity constraint table* zugeordnet, vgl. [31]; Absatz 3.11.5 . Diese enthält Einträge für key gdw. zu e_v oder einem direkten oder indirekten Kindelement von e_v gehört die Elementdeklaration $e_{key}decl$. Informal wird zu jedem dieser Elemente die Schlüsselbedingung ausgewertet, die Paare *(Element aus Definitionsbereich, Tupel)* gesammelt und alle diese Mengen vereinigt.

Sollten Konflikte auftreten, also $Tupel$ in dieser Menge nicht eindeutig sein, so werden alle Paare mit $Tupel$ entfernt, es sei denn, sie gehören direkt zu e_v, also nicht zu einem Kindelement. In dieser Menge muß nun das oben genannte t vorhanden sein, Gleichheit hier wieder gem. der XML-Schema Spezifikation

Beispiel aus [13]:

```
<keyref name="dummy2" refer="r:pNumKey">
  <selector xpath="r:zip/r:part"/>
  <field xpath="@number"/>
</keyref>
```

Beachte, daß es keine syntaktischen Beschränkungen gibt, in welcher Elementdeklaration wir `keyref` plazieren, geschieht dies aber in einer Elementdeklaration, die weder identisch zu der des referenzierten Schlüssels ist, noch Elementdeklarationen direkt oder indirekt enthält, in welchen der Schlüssel definiert ist, dann ist diese Fremdschlüsselbedingung semantisch sinnlos, denn die oben angesprochene Tabelle enthält niemals Einträge für den referenzierten Schlüssel.

Beachte, daß bei allen drei genannten Bedingungen sich die Auswertung der XPath-Ausdrücke auf die normalisierten XML-Dokumente bezieht, d.h. unter anderem, daß wenn ein Attribut im XML-Dokument nicht auftritt, aber einen Defaultwert besitzt, dieses Attribut nach der Normalisierung mit dem Defaultwert vorhanden ist.

Die Festlegung von Datentypen wurde im vorhergehenden Kapitel dieser Arbeit diskutiert, hierbei wurde dann auch ggf. herausgefunden, ob ein Attribut optional ist und evt. ein Defaultwert festgelegt usw. Mit Hilfe dieser Informationen müssen alle XML-Dokumente in der Eingabemenge zunächst normalisiert werden, bevor versucht wird, einige der genannten Abhängigkeiten zu erschließen.

5.4 Schlüssel- und Eindeutigkeitsbedingungen

Die gebräuchlichsten Algorithmen setzen alle auf dem relationalen Modell auf. Hierbei ist der Definitionsbereich der Abhängigkeit, also die Relation (bzw. Datenbank), fest vorgegeben. Bei einem XML-Dokument wissen wir von vorneherein nicht, was der Definitionsbereich einer Abhängigkeit ist.

[1] Das ist keine Bijektion, d.h. es gibt i.allg. mehrere Elemente aus dem XML-Dokument, die der gleichen Elementdeklaration zugeordnet sind.

Es spricht vieles dafür, sich zumindest diesen Definitionsbereich vom Benutzer vorgeben zu lassen, denn er kann aus Elementen mit verschiedenen Namen – aber nicht notwendig allen Elementen mit einem bestimmten Namen – bestehen und der Name eines Elementes impliziert den Typ eines Elementes nicht.

Würden wir versuchen, alle Schlüsselbedingungen zu finden, die innerhalb der übergebenen XML-Dokumentmenge gelten, so müßten wir zunächst die Elementdeklaration bestimmen, innerhalb welcher die Schlüsselbedingung definiert ist und ausgehend von dieser alle durch XPath-Ausdrücke gegebenen Möglichkeiten, den Definitionsbereich der Integritätsbedingung festzulegen. Das Inhaltsmodell legt fest, welche XPath-Ausdrücke überhaupt eine nicht-leere Menge an Elementen liefern können, ohne die Anzahl jetzt exakt zu bestimmen, erkennt man, daß es z.B. möglich ist, alle untergeordneten Elemente mit bestimmten Namen zu selektieren.

In der Namensmenge kann jeder Elementname enthalten sein oder nicht, so daß es hier bei einem Inhaltsmodell welches n verschiedene Namen von (nicht notwendig direkt) untergeordneten Elementen erlaubt, 2^n verschiedene Möglichkeiten gibt, also exponentiell viele und das sind noch lange nicht alle Möglichkeiten. Wenn der Benutzer hingegen den Definitionsbereich vorgibt, also konkret sagt, für welche Elementmenge er einen Schlüssel sucht, kann man sich nicht unerheblichen Aufwand sparen.

Hierfür ist die Kenntnis von zwei Angaben erforderlich:

- Die Informationen, für welche Elementdeklaration die Schlüsselbedingung abgeleitet werden soll.
- Die Angabe eines XPath zu untergeordneten Elementen für Elemente der oben angegebenen Elementdeklaration, das ist das `xpath`-Attribut des `selector`-Elementes.

Weiterhin arbeiten die Algorithmen auf einer (Beispiel)-Relation (oder Datenbank) und liefern auch die Bedingung für diese Relation (oder Datenbank). Eine Schlüsselbedingung in XML-Schema gilt für *ein* XML-Dokument, aber in *jedem* XML-Dokument. Wir haben zur Ableitung einer Integritätsbedingung also nicht ein Beispiel sondern mehrere, die Algorithmen für das relationale Modell gehen aber i.d.R. von einem Beispiel (Relation bzw. Datenbank) aus.

Weiterhin kann eine Schlüsselbedingung mehrfach in einem Dokument angewendet worden sein, gibt es z.B. eine Schlüsselbedingung im Typ von `Firmen`-Elementen, die besagt, daß die Mitarbeiternummern der `Mitarbeiter`-Kindelemente innerhalb einer Firma eindeutig sind, so kann es aber in einem XML-Dokument evt. mehrere `Firmen` geben und die Mitarbeiternummern von Mitarbeitern verschiedener Firmen sind nicht notwendig disjunkt.

Liegt der Definitionsbereich der zu ermittelnden Integritätsbedingung fest, so definiert diese konkrete Beispielelementmengen für welche die Integritätsbedingung gelten soll, diese stammen aus verschiedenen oder auch gleichen XML-Dokumenten.

Bevor wir mit dem Testen der Inhalte von untergeordneten Elementen zum Finden von (möglichen) Schlüssel- und Eindeutigkeitsbedingungen anfangen, können wir die Mengen von Feldern (`fields`) folgendermaßen einschränken:

- Die Menge von Feldern muß überhaupt durch einige für `fields` zugelassene XPath-Ausdrücke beschreibbar sein.
- Pro Element des Definitionsbereiches muß jedes Feld zu genau einem (im Fall von Schlüsselbedingungen) bzw. höchstens einem (im Fall von Eindeutigkeitsbedingungen) Attribut oder Element mit Stringinhalt ausgewertet werden.

Im zweiten Fall können wir sofort einige nicht-geltende Schlüssel- oder Eindeutigkeitsbedingungen sammeln, nämlich alle, die mindestens ein Feld enthalten, welches zu mehr als einem

Attribut bzw. Element ausgewertet wird, oder bei Schlüssel auch Felder, die zu leeren Menge ausgewertet werden.

Wie iterieren wir nun am besten über alle XPath-Ausdrücke für ein Feld? Klarerweise kann es zwei XPath-Ausdrücke geben, die *Schema-äquivalent* zu unserem – bis auf die Integritätsbedingungen – vorhandenen XML-Schema sind. *Schema-äquivalent* bedeutet im Gegensatz zu *äquivalent*, daß die zwei XPath-Ausdrücke nur dann gleiche Knotenmengen zu liefern brauchen, wenn das Dokument gültig bzgl. des XML-Schemas ist.

XPath erlaubt diverse Möglichkeiten, mit Abkürzungen äquivalente Xpath-Ausdrücke darzustellen, z.B. ist die *child*-Achse der Standardwert und kann deshalb weggelassen werden, o.B.d.A. sei deshalb die Syntax für die erlaubten XPath-Ausdrücke wie folgt, wobei die ersten drei Produktionen für die Ausdrücke in `selector` und `field` gelten:

```
Selector ::= Path ('|' Path)*
Step     ::= '.' | NameTest
NameTest ::= QName | '*' | NCName ':' '*'
```

Für `selector` hat die Path-Produktion folgende Form:

```
Path ::= ('.//')? Step ( '/' Step )*
```

Für `field` hat die Path-Produktion eine Form, die auch Attribute zuläßt:

```
Path ::= ('.//')? ( Step '/' )* ( Step | '@' NameTest )
```

Nach Regel 3.11.14 Punkt 2 muß der Pfadausdruck für `selector` auf Elemente *unterhalb* des aktuellen Elementes verweisen, für `selector` ist also z.B. ‚.' nicht gestattet, was einerseits semantisch sowieso nicht viel bringt und andererseits setzen wir ja voraus, daß der Nutzer uns einen gültigen XPath für `selector` übergeben hat.

Betrachten wir die Grammatik für die erlaubten XPath-Ausdrücke, so sehen wir, daß sie im Vergleich zu allen XPath-Ausdrücken sehr stark eingeschränkt ist, was uns die Arbeit damit aber sehr erleichtert, da z.B. die Auswertung – und viel wichtiger: die Konstruktion – von Prädikaten entfällt.

Der Algorithmus ermittelt nun alle Schlüssel- und Eindeutigkeitsbedingungen, die in den Beispielen gelten, hierfür werden alle negierten Schlüssel- und Eindeutigkeitsbedingungen ermittelt. Die ermittelten Schlüssel- und Eindeutigkeitsbedingungen werden dem Nutzer mitgeteilt, diese gelten aber nur für die Beispielmenge und sind nicht invariant gegenüber Obermengenbildung. Der Nutzer muß also entscheiden, gilt diese Bedingung auch allgemein, bzw. welche `field`-Elemente müssen noch hinzugetan werden, damit diese Bedingung allgemein gilt, wobei natürlich die genannten Restriktionen für die `field`-Elemente zu beachten sind.

Nachfolgend die Grundidee für den Algorithmus *Negierte_Schlüssel*, die Beispieldokumentmenge enthalte :

eins.xml

```
<root>
  ...
  <a>
    <b key="123" />
    <b>
      <key>145</key>
      <c>333</c>
      <c>444</c>
    </b>
  </a>
  ...
  <a>
    <c key_c="234">
    <b>
      <key="111"/>
      <d><e>777</e></d>
    </b>
  </a>
  ...
</root>
```

zwei.xml

```
<root>
  ...
  <a>
    <b key="123"/>
    <b key="222"/>
  </a>
  ...
</root>
```

- Der Nutzer hat die Elementdeklaration festgelegt, für den die Schlüsselbedingung ermittelt werden soll und ausgehend von diesem einen XPath P, der den Definitionsbereich des Schlüssels darstellt, P ist dann der Wert des `xpath`-Attributes des `selector`-Elementes des Schlüssels. Im nachfolgenden Beispiel sei das Element a – wir nehmen zu Vereinfachung an, daß alle a-Elemente denselben Typ haben – und der XPath P für den Definitionsbereich sei `b|c`.

  Daraus müssen wir einzelne XML-Teilbäume bilden, die jeweils die betreffenden a-Elemente als Wurzel haben. Diese werden in der Menge I gesammelt, die dann dem eigentlichen Algorithmus übergeben werden.

  Im Beispiel ergibt sich I zu:

```
<a>
  <b key="123" />
  <b>
    <key>145</key>
    <c>333</c>
    <c>444</c>
  </b>
</a>
```

```
<a>
  <c key_c="234">
  <b>
    <key="111"/>
    <d>
      <e>777</e>
    </d>
  </b>
</a>
```

```
<a>
  <b key="123"/>
  <b key="222"/>
</a>
```

- Der Rolle von Attributen entsprechen die `field`-Elemente, die jeweils eine Selector-Produktion der oben genannten Grammatik besitzen. Im Gegensatz zu Attributen liegen diese Produktionen aber nicht fest, wir können und müssen sie aus der Menge I entnehmen. Sinnvoll ist zunächst einmal jeder XPath ohne ‚|' der ausgewertet auf einem XML-Teilbaum $i \in I$ zu einem Element mit einem einfachen Typ[2] oder einem Attribut führt.

  Im Beispiel ist dies die Menge {`@key, @key_c, key, d/e, c`}

[2] d.h. simple type gem. XML-Schema Spezifikation

Dieser erste Schritt wird in der Funktion $bilde\_alle\_Pfade()$ realisiert. Wird irgendeiner dieser Pfade auf irgendeinem Teil-XML-Baum $i \in I$ zu mehr als einem Element evaluiert, so ist er nicht gültig und muß wieder entfernt werden, im Beispiel betrifft das den Pfad `c` .

Die Selector-Produktion erlaubt jedoch auch XPaths mit ‚|'. Logischerweise kommen hierfür nur welche der soeben gebildeten Pfadmenge in Betracht, denn alle anderen Pfade sind unsinnig, da sie entweder zu 0-elementigen oder 2+-elementigen Menge ausgewertet würden. Wird eine Pfadkomponente bzgl. ‚|' *immer* zu einer 0-elementigen Menge ausgewertet, so ist sie redundant, wird sie hingegen auch nur *einmal* zu einer 2+-elementigen Menge ausgewertet, so ist automatisch die gesamte Produktion mit ‚|' ungültig. Die Funktion $zusammengesetztePfade()$ bildet nun alle erlaubten Kombinationen. Hierbei kann natürlich der Fall eintreten, daß der gebildete Pfad mit ‚|' irgendwo zu einer 2-elementigen Menge ausgewertet wird, obwohl das die Komponentenpfade nicht taten. Solche Kombinationen sind zu verwerfen.

Im Beispiel kommen alle Kombinationen von `@key, @key_c, key, d/e` in Betracht, hier sind offenbar alle Kombinationen gültig, die `key` und `d/e` nicht gemeinsam enthalten, das sind:

```
@key            @key|@key_c       @key_c|d/e
@key_c          @key|key          @key|@key_c|key
key             @key|d/e          @key|@key_c|d/e
d/e             @key_c|key
```

Für Schlüsselbedingungen muß aber jeder Pfad zu einer *genau* 1-elementigen Menge ausgewertet werden.
Dieser gesamte Punkt wird von der Funktion $bildeKandidatenmenge()$ abgehandelt und liefert im Beipiel die Menge `{@key|@key_c|key}`

- Aus der Menge I nehmen wir ein Element i und werten den Pfadausdruck P aus, um den Definitionsbereich in Bezug auf i zu erhalten. Für das erste Element aus I im obigen Beispiel erhalten wir so die Menge (im Algorithmus später an J zugewiesen):

```
<b key="123" /> ,              <b>
                                 <key>145</key>
                                 <c>333</c>
                                 <c>444</c>
                               </b>
```

  Jeder dieser Teilbäume entspricht einem Tupel in einer Relation, jeder der gebildeten Pfadausdrücke im letzten Schritt einem Attribut. Wir wählen ein Paar aus dieser Menge (hier natürlich nur eine Möglichkeit) und ermitteln die Menge der Pfade, für welches dieses Paar gleiche Werte hat. Hier haben wir nur einen Pfad (`@key|@key_c|key`) zur Verfügung, auf welchem das einzig vorhandene Paar nicht übereinstimmt. Diese Menge an Pfaden (hier: $\emptyset$) ist ein negierter Schlüssel.

- Wir verfahren für jedes $i \in I$ gemäß obigem Punkt und erhalten eine Menge negierter Schlüssel, wobei jeder negierte Schlüssel eine Menge an Pfaden (aus der ermittelten Kandidatenmenge) ist. Unser Beispiel liefert allerdings immer $\emptyset$ als negierten Schlüssel. Ein negierter Schlüssel ist redundant, wenn er eine echte Teilmenge eines anderen negierten Schlüssels ist. Wir entfernen an dieser Stelle jetzt alle redundanten negierten Schlüssel (im Beispiel natürlich keine).

- zum Abschluß haben wir eine Menge nicht-redundanter negierter Schlüssel (hier: $\{\emptyset\}$) und eine Kandidatenmenge an Pfaden, deren Auswertung immer 1-elementige Mengen liefert (hier: `{@key|@key_c|key}`). Jede Teilmenge der Kandidatenmenge die echte Obermenge eines der ermittelten negierten Schlüssels ist, ist ein Schlüssel unter der Closed-World-Assumption. Da $\emptyset$ allerdings trivial ist, brauchen wir es in die Menge der negierten Schlüssel nicht aufzunehmen.

Im Beispiel ist `{@key|@key_c|key}` der einzige Schlüssel (unter der Closed-World-Assumption).

Algorithmus Negierte_Schlüssel

Eingabe:

I Menge an XMLtrees, die aus den ursprünglichen normalisierten XMLtrees für die Beispieldokumentmenge dadurch hervorgehen, daß sie alle deren Teilbäume enthält, die als Wurzelelement ein Element von dem Typ haben, für den die Schlüsselbedingung gesucht ist. o.B.d.A. sei $I = \{i_1, ..., i_n\}$

S ein XML-Schema, welches im wesentlichen die Inhaltsmodelle und Typen enthält

P ein XPath-Ausdruck, der den Definitionsbereich des Schlüssels festlegt

Ausgabe:

M_1 eine Familie von Mengen, die aus Selector-Produktionen der Grammatik für `field`-Elemente bestehen, wobei diese in M_2 enthalten sind

M_2 eine Menge von aus Selector-Produktionen der Grammatik für `field` -Elemente, für die die Auswertung immer eine Menge mit genau einem (bzw. maximal einem) Element ergab.

$M_1 := \emptyset$
$J := \emptyset$ [3]
für $k := 1$ bis n führe aus:
    $J := J \cup tree(Werte\,P\,auf\,i_k\,aus)$ [4]
M_2[5]$:= bildeKandidatenmenge(J, „key“)$ (*)

Es sei o.B.d.A. $M_2 = \{m_1, ..., m_l\}$

für $a := 1$ bis n führe aus[6]
    $J := tree(Werte\,P\,auf\,i_a\,aus)$ [7]
    Es sei o.B.d.A. $J = \{j_1, ..., j_t\}$
    für $b := 1$ bis $t - 1$ führe aus
        für $c := 2$ bis t führe aus[8]
            $m := \emptyset$ [9]
            für $d := 1$ bis l führe aus
                falls Wert(Werte m_d auf j_b aus)=[10]Wert(Werte m_d auf j_c aus) *(**)*
                    dann: $m := m \cup \{m_d\}$
            falls $m \neq \emptyset$
                dann: $M_1 := M_1 \cup \{m\}$

[3] J enthält die Teil-XML-Bäume
[4] Teilbäume des i_k-XMLtrees gem. P
[5] M_2 entspricht der Attributmenge, ist Liste von Selector-Produktionen für `field`-Elemente
[6] d.h. iteriere über jedes $i \in I$
[7] o.B.d.A. P auf i_a definiert, denn sonst Schlüsselbedingung trivialerweise erfüllt
[8] d.h. iteriere über jedes Paar von „Tupeln“
[9] m ist Menge gleicher „Attribute“ für 2 „Tupel“ aus J
[10] Gleichheit gem. der XML-Schema Spezifikation, $Wert()$ liefere den Zeichenketteninhalt.

o.B.d.A. sei $M_1 = \{h_1, ..., h_k\}$
für $a := 1$ bis $k - 1$ führe aus
  für $b := a + 1$ bis k führe aus
    falls $h_a \subset h_b$
      dann: $M_1 := M_1 \backslash \{h_a\}$

Um Eindeutigkeitsbedingungen abzuleiten, sind folgende Änderungen notwendig: Ersetze (*) durch „$M_2 := bildeKandidatenmenge(J, „unique“)$“ und interpretiere (**) beim Auftreten des Falls, daß die Auswertung auf mindestens einer Seite die leere Menge liefert, als „ungleich“.

Implementierungshinweis:
Ist der zu prüfende XPath m_d in Zeile (**) so aufgebaut, daß $m_d = m_{d_1}|...|m_{d_n}$ gilt, so können wir bei Auswertung von (**) als „wahr“ auch gleich jeden XPath $m_{d_{i_1}}|...|m_{d_{i_k}}$ mit $\{i_1, ..., i_k\} \subseteq \{1, ..., n\}$ als „wahr“ annehmen.
Solche Teilpfade sind aber nur bei Eindeutigkeitsbedingungen in M_2 enthalten, da sie ja manchmal zu 0-elementigen Mengen evaluiert werden.

Der Algorithmus geht direkt aus dem Algorithmus von [21] hervor, nach Ablauf des Algorithmus erhalten wir die Mengen M_1 und M_2. Es gilt dann: Jede nicht-leere Menge m von Ausdrücken der oben genannten Selector-Produktion kann genau dann *kein* Schlüssel (bzw. eindeutig) sein, wenn gilt: $m \not\subseteq M_2$ oder $\exists m' \in M_1$ mit $m \subseteq m'$.

Andersherum ausgedrückt: m ist ein möglicher Schlüssel genau dann, wenn gilt: $m \subseteq M_2$ und es gilt, $m \supset m'$ für ein $m' \in M_1$. Die Beispieldokumentmenge erfüllt dann nämlich die Schlüssel- bzw. Eindeutigkeitsbedingung, die derart konstruiert wird, daß der XPath-Ausdruck des `selector`-Elementes den vom Nutzer erhaltenen XPath-Ausdruck bekommt und für jedes Element von m ein `field`-Element mit identischem XPath-Ausdruck definiert wird. Da Schlüsselbedingungen nicht invariant gegenüber Obermengenbildung sind, muß der Nutzer entscheiden, ob so ein Schlüssel auch allgemein gelten soll.

Übergang zu Schlüsseln und Eindeutigkeitsbedingungen

Die *Menge aller Schlüssel* aus denen der Benutzer welche wählen kann, ergibt sich derart, daß man einen negierten Schlüssel aus der Menge M_1 wählt und zu einer echten Obermenge übergeht, wobei diese Obermenge aber eine Teilmenge von M_2 sein muß. Daß jede solche Obermenge wirklich ein Schlüssel für die Beispieldokumentmenge ist, wurden im letzten Teil des Algorithmus sichergestellt, wo alle negierten Schlüssel entfernt wurden, für die es eine Obermenge gibt, die ebenfalls negierter Schlüssel ist.

Eindeutigkeitsbedingungen ergeben sich nach Ablauf des modifizierten Algorithmus derart, daß wir ebenfalls zu einer echten Obermenge m einer negierten Eindeutigkeit $m' \in M_1$ übergehen, welche aber wiederum Teilmenge von M_2 ist.

Ist $m' \in M_1$, so verletzt ein $i \in I$ die zu m' korrespondierende Eindeutigkeitsbedingung an den Elementen e_1 und e_2 aus i. m als echte Obermenge von m' enthält eine weitere Selector-Produktion p, die falls sie sowohl auf e_1 als auch e_2 definiert ist, dort verschiedene Werte liefert, da m' inklusionsmaximal ist. Ist p auf mindestens einem von beiden nicht definiert, so ist die Eindeutigkeitsbedingung trivialerweise erfüllt, denn es sind nur die Elemente aus dem Definitionsbereich zu prüfen, für die alle `field`-Elemente zu genau 1-elementigen Mengen evaluiert werden.

Das schwierige an dem genannten Algorithmus, der offenbar im Kern eine Zeitkomplexität von $O(|I| * |J(I)|^2 * |M_2|)$ besitzt, ist die Bestimmung der Menge M_2, also die Menge der Selector-Produktionen für `field`-Elemente, die jeweils zu einelementigen (bzw. null- oder einelementigen) Mengen für alle Bäume werden.

Betrachten wir als Beispiel folgende (partielle) DTD:

```
<!ELEMENT p (n,p?)>
<!ELEMENT n #PCDATA>
```

Offenbar validiert diese DTD XML-Dokumente mit einer beliebig tiefen Schachtelung von p-Elementen, wobei jedes XML-Dokument natürlich per Def. endlich ist, wir aber keine Möglichkeit haben, die Schachtelungstiefe irgendwie einzuschränken.

Theoretisch kann es sein, daß eine Schlüsselbedingung gilt, für alle n-Elemente, die sich in der Schachtelung z.B. in 3. Stufe befinden:

```
<p>
  <n>...</n>
  <p>
    <n>...</n>
    <p>
      <n>Wert</n>
    </p>
  </p>
</p>
```

der entsprechende XPath-Ausdruck also bezogen auf das oberste p dann $p/p/n$ lauten würde. Da das eine Schlüsselbedingung ist, muß der Wert definiert sein, wir stellen also sicher, daß die Schachtelung mindestens 3 Stufen tief ist. Andererseits ist das offenbar ein Missbrauch von Schlüsselbedingungen, denn eine 3+-Stufigkeit ließe sich in XML-Schema sauber so realisieren[11]:

```
<complexType name="p1">
  <sequence>
    <element name="n" type="string"/>
    <element name="p" type="p2"/>
  </sequence>
</complexType>

<complexType name="p2">
  <sequence>
    <element name="n" type="string"/>
    <element name="p" type="p3"/>
  </sequence>
</complexType>

<complexType name="p3">
  <sequence>
    <element name="n" type="string"/>
    <element name="p" type="p3" minOccurs="0" />
  </sequence>
</complexType>
```

Zunächst könnte M_2 natürlich alle Pfadausdrücke enthalten, die theoretisch mit dem Inhaltsmodell bildbar sind, von diesen müßten wir dann diejenigen selektieren, die die Bedingung erfüllen,

[11] vgl. hierzu Anhang C zur Interpretation der XML-Schema Spezifikation

immer nur zu maximal einem Element bzw. Attribut-Knoten ausgewertet zu werden. Dieses Vorgehen ist so nicht machbar, da die Menge der Pfadausdrücke potentiell – siehe obenstehendes Beispiel – unendlich ist.

Andererseits brauchen wir bei Eindeutigkeitsbedingungen natürlich keine Pfade zu M_2 hinzufügen, welche auf jedem Beispieldokument zu einer leeren Menge evaluiert werden, denn das wären triviale Eindeutigkeitsbedingungen auf der Beispieldokumentmenge, wobei es möglicherweise unendlich viele gibt, so daß die Suche nach solchen wirklich keinen Sinn macht.

Bei Schlüsselbedingungen brauchen wir sogar nur solche Produktionen in M_2 aufnehmen, die auf allen Bäumen aus J zu genau einem Element bzw. Attribut ausgewertet werden. D.h. kombinieren wir das Generieren der Pfade mit dem Test der Mächtigkeit der jeweils zugehörigen Menge, so kommen wir in keine Probleme mit der Unendlichkeit der möglichen Pfade, M_2 ist natürlich endlich bedingt durch die Endlichkeit der Bäume in J.

Betrachten wir die Grammatik genauer, erkennen wir, daß es zu jedem Attribut bzw. Element in jedem Dokument genau einen Pfad ohne `.//` und `*` gibt, der von der Wurzel zu diesem Attribut bzw. Element führt, dabei kann ein Pfad natürlich zu mehreren Attributen bzw. Elementen führen. Da die Schachtelung wie bereits erwähnt beliebig tief sein kann, können wir gewisse Pfade die mit `.//` beginnen bei gewissen Inhaltsmodell nicht als Kombination einiger Pfade ohne `.//` darstellen. Andererseits sind unsere Beispieldokumente endlich, so daß auch die relevanten Pfade endlich sind und wir uns zunächst auf die Arbeite mit Pfaden ohne `.//` beschränken können. Haben wir nachher einen möglichen Schlüssel, so können wir testen, ob er innerhalb der Beispieldokumentmenge mit einem `.//`-Konstrukt abkürzbar wäre, ist dies der Fall, so können wir das als weiteren zusätzlichen Schlüssel anbieten, äquivalent ist er ob fehlender Closed-World-Assumption nicht.

Analog bilden wir nachher Pfade mit `*` als Abkürzung für eine Menge von Pfaden, auch dieser zusätzliche negierte Schlüssel ist gültig aber nicht äquivalent.

Beispiele:
$m = \{person/name|tier/name, ...\}, m' \in M_1, m' \subset m \subseteq M_2$ und es gibt keinen weiteren Pfad in M_2, der mit $name$ endet.
Dann sind auch $\{.//name, ...\}$ und $\{*/name, ...\}$ mögliche Schlüssel, aber i.allg. nicht äquivalent zu m.

Nachfolgend der Algorithmus zum Finden aller Kandidaten für gültige Selector-Produktionen der genannten Grammatik:

Die Pfadliste, die $FindeKandidaten()$ zurückliefert enthält Einträge der Form:
$(name, loccount, globcount, invalid, indexlist)$ und ist nach $name$ sortiert.

- $name$ ist dabei die Selector-Produktion selbst, also ein Pfad ohne ‚|' oder mehrere mittels ‚|' zusammengesetzte Pfade
- $globcount$ ist ein Zähler, der angibt, in wievielen Bäumen aus J $name$ zu genau einem Element bzw. Attribut ausgewertet wurde.
- $indexlist$ enthält die Liste von Nummern derjenigen Bäume aus J, bei denen $name$ zu genau einem Element oder Attribut ausgewertet wurde, es gilt also $|indexlist| = globcount$.
- $loccount$ und $invalid$ werden nur intern genutzt.

Zum Einfügen stehe die Funktion $FügeEin(\mathrm{Pfadliste}, \mathrm{pfad}, \mathrm{j})$ zur Verfügung:
Beim Einfügen in die Pfadliste wird, falls der Pfad noch nicht enthalten ist, der Eintrag $(\mathrm{pfad}, 1, 0, false, \mathrm{j})$ sortiert eingefügt, anderenfalls wird $loccount$ beim vorhandenen Eintrag um 1 erhöht und j in die $indexlist$ sortiert mit aufgenommen.

Die von der Funktion $FindeKandidaten()$ zurückgelieferte Liste hat bei einem Eintrag für $globcount$ genau dann den Wert $|J|$, wenn der entsprechende Pfad überall genau zu einer 1-elementigen Menge von Attributen bzw. Elementen ausgewertet wurde. Je nach Modus werden nur Pfade zurückgegeben, die immer zu genau 1-elementigen Mengen evaluieren ($modus$ = „key") oder Pfade, die immer zu 0 oder 1-elementigen Mengen evaluieren ($modus$ = „$unique$").

Funktion FindeKandidaten(J,modus)

Eingabe:

J	Menge J an XMLtrees
$modus$	$modus \in \{$„key", „$unique$"$\}$

Ausgabe:

$pliste$	Liste von Selector-Produktionen der obigen Grammatik für `field`-Elemente

$pliste := \emptyset$
Für alle $j \in J$
    $Bilde\_alle\_pfade(j, \epsilon, j)$
    $Updatepliste(pliste)$
$Entferne\_ungültige\_Pfade(pliste)$[12]

$pliste' := pliste$
$pliste'' := pliste$
wiederhole
    $pliste'' := zusammengesetztePfade(pliste, pliste'')$
    $pliste' := pliste' \cup pliste''$
bis $pliste'' = \emptyset$
falls $modus$ = „key"
    $Entferne\_nicht\_volldefinierte\_Pfade(pliste)$[13]
return $pliste'$

Funktion Bilde_alle_Pfade(j, teilpfad, j')
Eingabe:

j	Knoten in einem XML-Baum
$teilpfad$	ausgehend von j' zurückgelegter Teilpfad
j'	Wurzelknoten des XML-Teilbaumes, wird als Index in $pliste$ verwandt und deshalb durchgereicht bei rekursiven Aufrufen

Ausgabe:

$pliste$	vgl. oben, wird als globale Variable verwendet

für alle a_i Attribut von j:
    $FügeEin(pliste, teilpfad/@a_i, j')$
für alle e_i Kindelement von j:
    falls $Typ(e_i) \neq string$[14]
        dann: $bilde\_alle\_Pfade(j.e_i$[15]$, teilpfad/name(e_i), j')$
        sonst: $FügeEin(pliste, teilpfad/name(e_i), j')$

[12] alle Pfade mit $invalid$ =true aus Liste entfernen
[13] Pfade mit globcount<|J| entfernen
[14] oder ein von string abgeleiteter einfacher Datentyp; die Datentypen sind bereits bekannt
[15] das ist der Teilbaum, der sich durch Herabsteigen zu e_i ergibt.

Funktion UpdateListe(pliste)
für jedes $t = (name, loccount, globcount, invalid, indexlist)$ in $pliste$
  falls $loccount > 0$
    dann: $globcount := globcount + 1$
  falls $loccount > 1$
    dann: $invalid := true$
  $loccount := 0$

Offenbar können wir genau dann zwei Pfade „zusammensetzen“, wenn die Indexlisten disjunkt sind.

Funktion zusammengesetztePfade(pliste,pliste’)
Eingabe:

$pliste$	erste Liste von Pfade
$pliste'$	zweite Liste von Pfaden

Ausgabe:

$Rückgabeliste$	Liste von *gültigen* Pfaden, die durch Zusammensetzung mit ‚|‘ aus einem Pfad der ersten und einem Pfad der zweiten Liste entstanden

$Rückgabeliste := \emptyset$
für alle $i \in pliste$
  für alle $j \in pliste'$
    falls $i.indexlist \cap j.indexlist = \emptyset$
      dann: $Rückgabeliste := Rückgabeliste$
      $\cup\{(i.name|j.name, ..., i.globcount + j.globcount, i.indexlist \cup j.indexlist)\}$

Bemerkung:
Durch die Vereinigung am Ende wird die Duplikatfreiheit sichergestellt, zum Duplikaterkennen muß die |-Konkatenation daher sortiert erfolgen.

Implementierungshinweise:

- Die Indexlisten sind sortiert, es kann daher in einem Schritt die Vereinigungsindexliste gebildet werden und gleichzeitig die Disjunktheit überprüft werden.
- Evt. kann auch schon vorher auf Duplikate geprüft werden und natürlich können wir gleich abbrechen, wenn die Summe der einzelnen $globcount$-Werte größer als $|J|$ ist.

Die von $FindeKandidaten()$ zurückgelieferte Menge an Selector-Produktionen ist im Falle von Schlüsselbedingungen genau die Menge aller XPath-Pfade, die immer zu 1-elementigen Mengen ausgewertet werden, bzw. im Fall von Eindeutigkeitsbedingungen genau die Menge von XPath-Pfaden, die immer zu 0 oder 1-elementigen Mengen ausgewertet werden und nicht trivial sind. Trivial sind XPath-Pfade, die immer zu 0-elementigen Mengen ausgewertet werden.

Zeitkomplexität

p_1 sei die Anzahl der XPath-Ausdrücke ohne ‚|‘, die immer zu 0- oder 1-elementigen Mengen ausgewertet werden.
Für $i \in I$ sei $knP(i)$ die Anzahl der Knoten, zu denen P ausgewertet wird.
$|M_2|$ ist in $O(2^{p_1})$ für Eindeutigkeitsbedingungen, da die Situation konstruierbar ist, daß jeder Pfad nur an genau einer Stelle zu einer 1-elementigen Menge ausgewertet wird, wir also alle Kombinationen mittels ‚|‘ als Kandidaten bilden können. Für Schlüssel ist hingegen $|M_2|$ in $O(p_1)$, denn entweder alle solche Pfade werden immer zu 1-elementigen Mengen evaluiert oder aber die Kombinationen mittels ‚|‘ von diesen, wobei die einzelnen Pfade dann nicht in M_2 sind.

Für beide Fälle werden aber im Worst Case zunächst $O(2^{p_1})$ XPaths gebildet, wobei jeweils Mengenvereinigungen stattfanden, die aber in $O(|J|)$ liegen, wobei das J beim Aufruf der Funktion $FindeKandidaten()$ gemeint ist, $|J| = O(\sum_{i \in I} knP(i))$. Im Worst Case dominiert daher $FindeKandidaten()$ den restlichen Teil, der eine Komplexität von $\sum_{i \in I} O(knP(i)^2 * |M_2|)$ besitzt, sinnvolle Abschätzungen für den Average Case zu machen zählt zu den noch offenen Problemen.

Wir können aber die beiden folgenden Heuristiken anwenden, welche die Laufzeit effizient beschränken:

- Angabe von d_1 als Beschränkung der XPath-Tiefe (d.h. das Auftreten von ‚/')
- Angabe von n_1 als Beschränkung der XPath-Teile (d.h. das Auftreten von ‚|')

Modifizieren wir die Funktion $FindeKandidaten()$ entsprechend, so werden nur maximal $\binom{p_1}{n_1}$XPath's gebildet, weiterhin wird mit der Angabe von d_1 die Zahl der zu prüfenden XPaths ohne ‚|' i.d.R. wesentlich kleiner als p_1.

In [21] sind noch etliche andere Heuristiken und Verfahren zum Finden von Schlüsseln aufgeführt, z.B. ist bei einer Relation die Wahrscheinlichkeit für ein Attribut in einem Schlüssel aufzutreten empirisch korreliert mit der Zahl der verschiedenen Werte, die dieses Attribut in der Relation annimmt.

Solche heuristischen Verfahren für das Relationenmodell sind teilweise auch für unser Problem übertragbar, was aber über den Rahmen dieser Arbeit hinausführt.

5.5 Fremdschlüssel

Die Semantik von Fremdschlüsseln wurde oben bereits erläutert. Zur Vereinfachung wollen wir annehmen, daß Schlüssel bereits erkannt wurden, d.h. der Nutzer hat einige Schlüssel aus der Menge der möglichen Schlüssel ausgewählt, weiterhin stehe schon zu jedem Element aus der Beispieldokumentmenge die entsprechende *identity constraint table* bereit, die Menge der Schlüssel sei mit $Keys$ bezeichnet.

Wir müssen folgende Dinge klären:

- Zu welcher Elementdeklaration gehört der Fremdschlüssel?
- Wie ist der Definitionsbreich festgelegt?
- Welcher Schlüssel wird referenziert?
- Wie sind die Tupel definiert?

An dieser Stelle wollen wir zur Vereinfachung gem. der Diskussion bei Schlüsseln annehmen, daß die ersten beiden Punkte vom Nutzer vorgegeben sind.

Beachte, daß auch Fremdschlüsselbedingungen nicht invariant gegen Obermengenbildung sind. Weiterhin sind sie sogar bei einer geltenden Schlüsselbedingung nicht invariant gegen Teilmengenbildung. Da die Erkennung von Fremdschlüssel sehr zeitintensiv ist, wird im folgenden nur die Grundidee für einen Algorithmus angegeben:

Wir können mit $findeKandidaten(J,$ „$unique$") ebenso wie in $Negierte\_Schlüssel()$ die Menge M_2 der möglichen „Attribute" (also die Werte der `field`-Elemente) für den Fremdschlüssel bestimmen, J sei wie im Algorithmus $Negierte\_Schlüssel()$ gebildet.

Hiernach iterieren wir über alle $k \in Keys$ und versuchen (bei festliegender Elementdeklaration und Definitionsbereich) alle nicht-trivialen Fremdschlüssel zu finden, die k referenzieren.

Einen Fremdschlüssel wollen wir mit *nicht-trivial* bezeichnen, wenn auf irgendeinem Element des nicht-leeren Definitionsbereiches das Tupel *vollständig* definiert ist.

Der Fremdschlüssel muß genauso viele Tupelkomponenten wie der referenzierte Schlüssel haben, die Anzahl sei hier gleich n. Konkret müssen wir die Menge $\underbrace{M_2 \times \cdots \times M_2}_{n}$ prüfen, denn anders als bei Schlüsseln kann das doppelte Vorkommen eines `field`-Elementes mit gleichem Pfad schon Sinn machen.

Für jedes $j \in J$ und alle dieser $p_1 \times \cdots \times p_n$ Möglichkeiten bilden wir alle Tupel, erhalten so die Menge $T(j, p_1, \ldots, p_n)$ und sehen in der *identity constraint table* zum Wurzelelement aus j nach, ob $T(j, p_1, \ldots, p_n)$ beim Eintrag zu k als Teilmenge enthalten sind.

Zur algorithmischen Vereinfachung können wir folgende Aussagen benutzen:

- Sind die Typen bereits festgelegt und wissen wir, daß die Mengen der lexikalischen Repräsentationen eines p_i disjunkt zu der des entsprechenden Eintrages im referenzierten Schlüssels ist, so können wir den Test für $p_1 \times \cdots \times p_n$ sofort abbrechen.
- Anstatt jedesmal die Menge T zu bilden, bilden wir für alle $p_i \in M_2$ die Mengen T_i und prüfen für jedes dieser T_i, alle $j \in J$ und jede Komponente des Schlüssels das Enthaltensein in jeder der entsprechenden Tabellen. Schlägt der Test auch nur bei einem $j \in J$ fehl, so markieren wir das p_i mit der entsprechenden Komponentennummer und brauchen den Test für p_i und andere $h \in J$ für dieselbe Komponente auch nicht mehr ausführen.
  Am Ende sind genau die $p_1 \times \cdots \times p_i \times \cdots \times p_n$ mögliche Fremdschlüssel (zu k), für die für alle i gilt: p_i ist *nicht* mit i markiert.
- Das obige T_i kann auch gespeichert werden, damit es für den nächsten Schlüssel nicht neu berechnet werden muß.

Zu jedem $k \in Keys$, wobei die Anzahl der Komponenten $|k|$ von k gleich n sei, müssen bis zu $|M_2| * n * |J|$ Mengenvergleiche ausgeführt werden.

In [21] werden noch Heuristiken zur Suche nach Inklusionabhängigkeiten im relationalen Modell beschrieben, die man modifiziert ebenfalls anwenden kann.

5.6 Kardinalitäten

Die bisher diskutierte Möglichkeit ein XML-Schema zu erzeugen, war es, eine DTD als Grundlage zu nehmen. Kardinalitäten bei DTDs und XML-Schemata haben die Form „x bis y mal“ mit $x \leq y$, wobei x und y die Werte 0 und 1 annehmen können und y zusätzlich auch den Wert n, der die Anzahl nach oben nicht beschränkt. Bei XML-Schema entfällt die Beschränkung auf 0 und 1, hier sind jetzt alle natürlichen Zahlen (einschließlich der 0) erlaubt. Um Kardinalitäten sinnvoll erkennen zu können, müssen wir untersuchen, wie die jeweiligen Algorithmen beim Generalisieren prinzipiell vorgehen, da die Beschreibung der Kardinalitäten in XML-Schema mit `minOccurs`=x und `maxOccurs`=y recht „zeichenintensiv“ ist, wird im folgenden [x,y] verwendet:

Das Problem sind die Kriterien, nach denen diese Algorithmen eine Generalisierung durchführen. Das abstrakte Kriterium „Lesbarkeit“ differiert sehr zwischen DTD und XML-Schema, z.B. ist `aa|aaaa|aaaaa` schwerer lesbar als `a[2,5]` und wird deshalb schnell zu `a*` generalisiert, obwohl in einem XML-Schema `a[2,5]` sinnvoller wäre. Natürlich müssen wir auch hier eine Heuristik benutzen, so daß wir auch z.B. zu `a[2,n]` generalisieren können.
Betrachten wir überblicksartig die vorgestellten Verfahren, hierbei können wir einfache (`a[x,y]`) und schwierige (`CM[x,y]`) Wiederholungen unterscheiden:

- Bei DTD-Miner können wir nichts begutachten, da wir keine ausreichende Spezifikation haben.
- Bei GB-Engine wird verhältnismäßig schnell generalisiert, die Regel *Repeating Atoms* faßt schon ab zweimaligen Auftreten zu [1,n] zusammen, allerdings kann man diese Regel von der Anwendung ausschließen, aber nur komplett.
- Bei Ahonen und Manila gibt es im einfachen Fall keinen Pfad länger k, der nur mit a's gekennzeichnet ist, ist y größer k, so wird hier also übergeneralisiert. Haben wir eine Heuristik, ab welchem Wert für y wir n nehmen, können wir k so wählen. Wie der Algorithmus im schwierigen Fall vorgeht, müßte man experimentell ermitteln, den Grad der Generalisierung können wir aber immer mit k steuern.
- Der Ansatz von Fernau ist generisch, es wird keine konkrete Unterscheidungsfunktion festgelegt, so daß wir auch hier nichts diskutieren können.
- Bei Extract müssen wir die Kandidaten-Menge und die Zielfunktion diskutieren. Was nicht in die Kandidaten-Menge kommt, kann natürlich auch nicht im Ergebnis stehen. Angenommen, wir wollen `CM[2,n]` ableiten, als Kandidaten kommen bei Extract aber nur `CM` und `CM*` (und Konkatenationen davon) in Betracht.

  Haben wir am Ende `CMCMCM*`, so können wir dieses im XML-Schema durch `CM[2,n]` substituieren. Die ersten beiden Auftreten von CM werden mit je 0 Bits kodiert (zzgl. den Bits für die Kodierung innerhalb von CM), so daß wir für die Kodierung eines konkreten Dokumentteiles $CMCMCM^n$ i.allg. bei $CMCMCM*$ weniger Bits benötigen, als bei $CMCM*$ oder $CM*$, allerdings können es evt. auch gleich viele Bits durch die Verwendung von $Ceiling()$ sein.

  $CM*$ läßt sich aber in der DTD kürzer kodieren als $CMCMCM*$, wobei es hier zu einer Anormalität kommen kann, wenn recht viele Bits für die Kodierung von CM benötigt werden und entweder wenig Dokumente vorliegen oder die benötigten Bits für die Kodierung der Dokumente sich für $CMCMCM*$ und $CM*$ nicht unterscheiden. Das MDL-Prinzip scheint aber trotzdem geeignet, wir könnten die Kandidatenbildung und Dokumentkodierung so belassen, aber in der DTD-Kodierung Konstrukte wie $CM[x,y]$ erlauben, wobei wir dann die Kodierung so wählen müssen, daß $CM[2,n]$ weniger Bits benötigt als $CMCMCM*$.

Generell können wir entweder einen der genannten Algorithmen verwenden, wobei wir ihn entsprechend parametrisieren (Ahonen/Manila) oder abändern (XTRACT), damit wir eine Übergeneralisierung vermeiden und dann nach Ablauf folgende Regeln anwenden, bis sich die Schemabeschreibung nicht mehr ändert:

Regel:
Sei CM ein beliebiges Inhaltsmodell im Schema, seien p_1, p_2 Particle[16] in CM
Sind $p_1[x_1, y_1], p_2[x_2, y_2]$ benachbart und $p_1.name = p_2.name$, dann ersetze beide durch $p_1[x_1 + x_2, y_1 + y_2]$.

Beachte, daß in XML-Schema gleiche Namen innerhalb eines Inhaltsmodells auch gleiche Typen implizieren, wir die Typdefinitionen also nicht vergleichen müssen.

Führt der Algorithmus hingegen eine Übergeneralisierung durch, so können wir das Schema auch nachträglich in der Struktur ändern, indem wir nochmals die ursprüngliche Menge I betrachten und für jedes $*$ und $+$ in jedem Inhaltsmodell nochmals die exakten Anzahlen für alle $i \in I$ ermitteln und die entsprechende Untergrenze dann als Minimum dieser Anzahl und die Obergrenze als Maximum oder n neu setzen.

Beide Vorgehen sind von der Zeitkomplexität linear in der Größe des Schemas bzw. der Gesamtgröße (d.h. Element- und Attributzahl) der Menge I.

[16]vergleiche hierzu die XML-Schema Spezifikation

Kapitel 6

Implementierung

Die Implementation soll dazu dienen, beispielhaft einige der vorgestellten Ansätze bzw. Algorithmen zu demonstrieren.

6.1 Funktionsbeschreibung

Das Programm kann:

- Erstellen einer vollständigen DTD bzw. eines vollständigen XML-Schemas
- Extraktion aller Beispielmengen (d.h. die sample sets der zu erkennenden Sprachen) aus einer Menge von XML-Dateien
- Generieren es (k, h)-kontextuellen Automaten für jede Sprache
- Erzeugung der Nicht-1-Mehrdeutigkeit für diese Automaten
- Erkennen von Schlüsseln
- Aufsammeln vorhandener Attribute

Der Nutzer hat Einfluß auf:

- Wahl, ob DTD oder XML-Schema generiert wird
- Wahl der Parameter k und h für die (k, h)-Kontextualität
- Wahl des Verzeichnisses für die Eingabedateien, die temporären Dateien und Names der Ausgabedatei
- Wahl der Suffixlänge (d.h. entweder konkrete Länge oder kompletter Suffix) zur Abgrenzung der Datentypen.

  Wie bereits diskutiert sind zwei Elemente in beliebigen Dokumenten aus der Kollektion dann vom gleichen Typ, wenn sie mit dem gleichen Pfad (ohne Wildcards) erreichbar sind, bei DTDs reicht die Namensgleichheit der Elemente.

  Der Nutzer kann hier eine Relaxation dieser Bedingung einstellen, indem zwei Elemente aus beliebigen Dokumenten der Kollektion als schon dann zum gleichen Typ gehörend angesehen werden, wenn die beiden Pfade (ohne Wildcards) mit denen die Elemente erreichbar sind, denselben n-Suffix[1] haben.

- Anzahl der Schlüssel, bei jedem Schlüssel:
  - Auswahl des Elementes für den der Schlüssel gelten soll
  - Auswahl des Definitionsbereiches
  - Auswahl eines maximalen Nicht-Schlüssels und Ergänzung selbigens aus der präsentierten Liste mit gültigen Pfaden

Nicht implementiert wurden:

- Erkennen der konkreten einfachen Typen (engl. simple types) gem. XML-Spezifikation für Elemente mit nicht-Elementinhalt und Attribute
- bei Attributen keine Erkennung der Qualifikatoren (konstant, optional, usw.)
- Erkennung von `keyref` und `unique`
- Alle vom Nutzer angegebenen XPaths dürfen keine Wildcards enthalten, müssen also ausschließlich aus Elementnamen getrennt mit / bestehen.
- weiterführende Ansätze zur Zuordnung eines Elementes aus der Beispielmenge zu einem Typ, Berücksichtigung der `xsi:type`-Attribute
- komplexere Vereinfachungen auf dem erzeugten Inhaltsmodell, welche die Nicht-1-Mehrdeutigkeit wahren (u.a. Erkennen von `all`-Gruppen, Zusammenfassungen bei `minOccurs` und `maxOccurs`)
- Ausgabe von Schlüssel für Elementdeklarationen, die nicht Top Level-Elementdeklarationen sind
- Elementnamen dürfen keinen Unterstrich (_) enthalten
- Namensräume (engl. namespaces) werden nicht unterstützt

6.2 Bemerkungen zu Implementation

Die Implementation folgt generell den in dieser Arbeit vorgestellten Algorithmen, insbesondere die Algorithmen zur Schlüsselerkennung sind bereits ziemlich implementationsnah dargestellt.

6.2.1 Erzeugung des Automaten

Die Erzeugung des (k, h)-kontextuellen Automaten[2] aus der Beispielmenge teilt sich in zwei Schritte:

[1] n > 0; ist n=1, so folgt bereits aus dem Elementnamen der Elementtyp
[2] (k, h)-kontextuell nach der *Automatendefinition*

Erzeugung des k-kontextuellen Automaten

Hierfür wird der Ansatz der *k-Gramme* benutzt, der auch von Ahonen [2] vorgestellt wird. Hier soll der Ablauf nur kurz skizziert, nicht aber bewiesen werden[3]:

- jedem Wort der Beispielmenge (zu der zu erkennenden Sprache) werden k Stück Sonderzeichen (#) vorangestellt und eines angefügt
- die k-Gramme sind die Menge aller Teilworte der Länge $k+1$ aus allen Worten der (modifizierten) Beispielmenge
- der k-Präfix jedes Wortes charakterisiert den Zustand eindeutig, der 1-Suffix bezeichnet ein Symbol, mit dem von dem bezeichneten Zustand aus ein Übergang möglich ist und zwar genau in den Zustand, der mit dem k-Suffix dieses Wortes bezeichnet wird; ist der 1-Suffix das Sonderzeichen (#), so dient das Wort nicht zur Beschreibung eines Überganges sondern zur Kennzeichnung, daß der durch den k-Präfix charakterisierte Zustand ein Endzustand ist. Ein Wort korrespondiert zum Startzustand, wenn es mit k Stück Sonderzeichen (#) beginnt.
  Die Zustände und Übergänge werden gem. dieser Diskussion gebildet.

Umwandlung dieses Automaten in einen (k,h)-kontextuellen Automaten

Der oben erzeugte Automat ist k-kontextuell, aber i.Allg. noch nicht (k,h)-kontextuell. Man mache sich klar, daß wir den $(k,k-1)$-kontextuellen Automaten erhalten, wenn wir:

- die Richtung aller Transitionen aus δ umkehren,
- alle Zustände miteinander verschmelzen, welche dem Determinismus des Automaten widersprechen, wobei δ zu Grunde gelegt wird[4],
- wieder die Richtung aller Transitionen aus δ umkehren,

Den (k,h)-kontextuellen Automaten erhalten wir, indem wir das genannte Vorgehen $(k-h)$-mal ausführen, wobei wir δ nur zu Beginn und zu Ende umkehren müssen. Außerdem ist der Algorithmus zur Erzeugung des Determinismus gleich so ausgelegt, daß er auch nach einer einstellbaren Anzahl an Iterationen abbricht und nicht erst bei Erreichen des Determinismus, so daß für diesen Schritt keine zusätzlichen Algorithmen nötig sind, da wir die Routinen zur Erzeugung der Determinismus hier einfach benutzen.

6.2.2 Erzeugung der Nicht-1-Mehrdeutigkeit und Ableiten des Ausdruckes

Die vorgestellten Algorithmen zur Erzeugung der Nicht-1-Mehrdeutigkeit sind auf abstrakterem Niveau als mit Java direkt realisierbar. Generell wird in der Implementation ein *einziger* Automat genutzt und nicht mehrere, die später wieder zusammengefügt werden müssen. Bei der Durchführung des S-Schnittes werden deshalb die Kanten nicht entfernt, sondern nur ausgeblendet, gleichzeitig wird das Array `orbit` benutzt, um die Arbeit auf Automatenteile einzugrenzen.

Zum Berechnen der Orbits selbst wird Tarjans Algorithmus (entnommen aus [27]) verwendet, der eine Zeitkomplexität von $O(|Transitionen| + |Zustände|)$ besitzt. Nach der Durchführung eines S-Schnittes wird auch dieser Algorithmus nur auf dem aktuellen Automatenteil neu

[3] man kann sich den Beweis sehr schnell selbst überlegen

[4] d.h. impliziert δ nicht das Verschmelzen von z_i mit z_j, sondern wird dieses erst von vorangehenden Verschmelzungen impliziert, so werden z_i und z_j *nicht* verschmolzen

ausgeführt, es findet immer nur eine Verfeinerung *eines* Orbits in *mehrere* Orbits nach einem S-Schnitt statt.

Nach der Ausführung des vorgestellten Algorithmus zur Erzeugung der Nicht-1-Mehrdeutigkeit wird der Automat deterministisch gemacht. Diese zwei Schritte werden solange wiederholt, bis der Automat sich nicht mehr ändert.

Falls das Erzeugen eines M-konsistenten Symbols nötig ist, so wird versucht die Wahl so zu treffen, daß der Automat möglichst wenig generalisiert wird. Die Heuristik hierfür ist, zunächst die Menge an Symbolen zu bestimmen, für die ausgehend von allen Endzuständen in eine minimale Zahl von Zuständen überführt wird (denn genau diese minimale Zahl von Zuständen werden miteinander verschmolzen) und aus dieser Menge ein Symbol zu wählen, für das möglichst wenige Transitionen hinzugefügt werden müssen.

Der Algorithmus zum Erzeugen des Determinismus arbeitet ebenso iterativ, bis sich der Automat nicht mehr ändert (oder wie oben angesprochen, eine gewisse Zahl von Iterationen stattgefunden hat). Pro Iteration werden alle die Verschmelzungen durchgeführt, die von δ *direkt* impliziert wurden, nicht aber nötige Verschmelzungen, die von vorangegangenen Verschmelzungen impliziert werden. Nach einer Iteration wird δ aktualisiert.

6.2.3 Schlüsselerkennung

Die Algorithmen zu Schlüsselerkennung sind sehr nah an der Darstellung im Kapitel 5 implementiert.

6.3 Test

Zum Testen standen zwei Kollektionen zur Verfügung, zum einen eine Sammlung von 37 Shakespeare-Werken im XML-Format (von Jon Bosak nach XML konvertiert) zusammen mit einer zugehörigen DTD. Diese Shakespeare-Texte kann man als dokumentenorientierte XML-Daten auffassen.

Weiterhin diente das Periodensystem der Element im XML-Format als Beispiel für datenorientierte XML-Dateien.

Die Ergebnisse auf den Shakespeare-Werken machen einen brauchbaren Eindruck (bei $(2, 1)$-Kontextualität), die Original-DTD und die generierte DTD sind im Anhang B aufgeführt.

Hingegen ist das Ergebnis bzgl. des Periodensystems wenig brauchbar, auch bei verschiedensten Parametrisierungen des Programmes. Das Originalinhaltsmodell für `ATOM` hat in etwa die Form:
`A,B,C,D,E,F?,G,H,I,J?,K,L,M?,N,O?,P,Q,R`

Durch das Ablesen des nicht-1-mehrdeutigen Ausdruckes wird ein unbrauchbar kompliziertes Inhaltsmodell erzeugt.

Die Ergebnisse zeigen, daß man bessere Generalisierungsverfahren vor allem für (?) braucht (z.B. entsprechende Suchraumerweiterung bei XTRACT, nach Angaben der Autoren funktioniert XTRACT bei solchen Fällen auch nicht zufriedenstellend) oder effektive Verfahren, den regulären Ausdruck unter Wahrung der Nicht-1-Mehrdeutigkeit zu vereinfachen.

Kapitel 7

Zusammenfassung und Ausblick

An dieser Stelle soll noch einmal zusammengefaßt werden, was in dieser Diplomarbeit gezeigt bzw. erreicht wurde und weiterhin eine Übersicht der noch offenen Probleme bzw. noch zu realisierenden Arbeiten gegeben werden.

Die gesamte Aufgabe, d.h. das Erzeugen eines XML-Schemas (bzw. einer DTD) für eine übergebene Beispielmenge an XML-Dokumenten wurde in drei Teilprobleme unterteilt:

- Inhaltsmodell
- (einfache) Datentypen
- Integritätsbedingungen

7.1 Inhaltsmodell

Zu Beginn wurde eine grundlegende theoretische Einführung der Problematik gebracht. Dieses umfaßt eine Wiederholung der Grundlagen für reguläre Sprachen, weiterhin eine kurze Einführung in die Grundlagen für das Erkennen von Sprachen soweit dies für diese konkrete Problemstellung erforderlich ist und auch eine Einführung in die Problematik der Nicht-1-Mehrdeutigkeit. Ergänzend wurde eine Formalisierung von XML-Dokumenten, nämlich der *XMLtree* eingeführt.

Nachfolgend wurden verschiedene existierende Ansätze für das Erzeugen einer DTD untersucht. Hierbei stellte sich heraus, daß bei einem ansonsten vielversprechenden Ansatz (XTRACT) das Problem der Nicht-1-Mehrdeutigkeit übersehen wurde. Weiterhin wurden in den Arbeiten von Ahonen verschiedene Fehler ausgemacht: es wurde gezeigt, daß Ahones zwei Definitionen der (k, h)-Kontextualität inkonsistent sind. Weiterhin wurde ein Fehler im Algorithmus zur Erzeugung der Nicht-1-Mehrdeutigkeit korrigiert und ein korrekter Beweis für die Korrektheit des Algorithmus gegeben, da der Beweis von Ahonen mangelhaft war.

Abschließend wurde diskutiert, inwiefern die Ansätze für das Erzeugen einer DTD auf das Erzeugen eines XML-Schemas übertragbar sind.

7.2 Datentypen

In diesem Abschnitt wurde die Problematik der Erkennung von (einfachen) Datentypen vorgestellt und – da die Thematik beliebig ausbaubar ist – ein *einfacher* Ansatz für das Erkennen von Datentypen präsentiert.

7.3 Integritätsbedingungen

Hier wurden vor allem die Probleme des Erkennens von Kardinalitäten und Schlüsselbedingungen diskutiert. Ausgehend von Meike Klettkes Algorithmus aus [21] für das relationale Modell wurde ein Algorithmus für die konkrete Problemstellung entwickelt. Der Unterschied besteht darin, daß zum einen nicht eine Relation, sondern mehrere „Relationen" vorliegen, wobei hier *keine* 1:1-Beziehung zu den XML-Dateien vorliegt und weiterhin die Menge der „Attribute" nicht feststeht, so daß auch diese erst gefunden werden müssen. Durch den letzten Punkt ist i.allg. (d.h. dem Finden *aller* Schlüsselkandidaten) und im schlechtesten Fall leider eine exponentielle Laufzeit zu erwarten.

7.4 Offene Probleme und zukünftige Arbeiten

- wie bereits unter Abschnitt 6.3 angeführt, wird ein Verfahren zum Vereinfachen regulärer Ausdrücke unter Wahrung der Nicht-1-Mehrdeutigkeit benötigt
- es ist eine vergleichende Studie notwendig, die mit bestimmten, in der Praxis auftretenden XML-Kollektionen die Leistungsfähigkeit der verschiedenen Ansätze zur Erschließung des Inhaltsmodells untersucht; nach Meinung des Autors ist der MDL-Ansatz der vielversprechendste, bei diesem muß der Suchraum noch weiter sinnvoll aufgespannt werden, ohne ihn aber zu groß werden zu lassen,
- es ist zu untersuchen, inwieweit die Generalisierungen der vorgestellten Ansätze mit der weiteren Generalisierung bei der Erzeugung der Nicht-1-Mehrdeutigkeit harmonieren,
- es ist notwendig, geeignete Ansätze und Heuristiken für das Erschließen (bzw. Generieren) von Datentypen zu entwickeln, insbesondere unter dem Aspekt einer vorhandenen *erweiterbaren* Typ-Hierachie,
- in die Beispielimplementation müßten noch alle in Kapitel 6 angesprochenen, noch ausstehenden Punkte eingebracht werden

Anhang A

Benutzung des Programmes

A.1 Systemvoraussetzungen, Bezug

Das Beispielprogramm wurde mit Java Version 1.2.2 und Xerces Version 1.4.0 erstellt, es ist sicherzustellen, daß die zu Xerces gehörenden Klassen (alle zusammen in `xerces.jar`) im `CLASSPATH` vorhanden sind.

Die Kompilierung erfolgt mit:
`javac MAIN.java`

Das Programm mit den Quelltexten wird auch gerne zum Experimentieren und Weiterentwickeln zur Verfügung gestellt.

Sollte das Programm diesem Exemplar der Diplomarbeit nicht auf einem Datenträger beiliegen, so ist es aus dem WWW beziehbar:

http://wwwdb.informatik.uni-rostock.de/xml

A.2 Benutzung

Der Aufruf erfolgt mit:
`java MAIN` *Parameter*

Alle Parameter sind optional, die folgende Aufzählung gibt eine Übersicht:

-v n der Verbose-Parameter:

- n=0 keine Ausgaben
- n=1 wichtige Ausgaben (*Default*)
- n=2 sehr viele Ausgaben zum Programmablauf

-m n der Modus-Parameter:

- n=DTD DTD-Modus
- n=0 zwei Elemente haben gleichen Typ gdw. die Pfade (ohne Wildcards) zu ihnen sind identisch (*Default*)

$n > 0$ zwei Elemente haben gleichen Typ gdw. die Pfade (ohne Wildcards) zu ihnen haben identische n-Suffixe

-i $Pfad$ der Eingabepfad, dieser darf ausschließlich die XML-Kollektion enthalten, keine anderen Dateien! (*Default*=`xmlfiles/`)

-t $Pfad$ das temporäre Verzeichnis, dieses muß leer sein und wird bei Bedarf angelegt (*Default*=`output/`)

-o $Datei$ die Ausgabedatei, eine vorhandene Datei gleichen Namens wird ggf. überschrieben (*Default*=`result.xsd`)

-k n die erzeugten Automaten sind (n,h)-kontextuell

-h n die erzeugten Automaten sind (k,n)-kontextuell

Shakespeare DTD

Die Original-DTD:

```
<!-- DTD for Shakespeare    J. Bosak    1994.03.01, 1997.01.02 -->
<!-- Revised for case sensitivity 1997.09.10 -->
<!-- Revised for XML 1.0 conformity 1998.01.27 (thanks to Eve Maler) -->

<!ENTITY amp "&#38;">

<!ELEMENT PLAY     (TITLE, FM, PERSONAE, SCNDESCR, PLAYSUBT,
                    INDUCT?, PROLOGUE?, ACT+, EPILOGUE?)>
<!ELEMENT TITLE    (#PCDATA)>
<!ELEMENT FM       (P+)>
<!ELEMENT P        (#PCDATA)>
<!ELEMENT PERSONAE (TITLE, (PERSONA | PGROUP)+)>
<!ELEMENT PGROUP   (PERSONA+, GRPDESCR)>
<!ELEMENT PERSONA  (#PCDATA)>
<!ELEMENT GRPDESCR (#PCDATA)>
<!ELEMENT SCNDESCR (#PCDATA)>
<!ELEMENT PLAYSUBT (#PCDATA)>
<!ELEMENT INDUCT   (TITLE, SUBTITLE*, (SCENE+|(SPEECH|STAGEDIR|SUBHEAD)+))>
<!ELEMENT ACT      (TITLE, SUBTITLE*, PROLOGUE?, SCENE+, EPILOGUE?)>
<!ELEMENT SCENE    (TITLE, SUBTITLE*, (SPEECH | STAGEDIR | SUBHEAD)+)>
<!ELEMENT PROLOGUE (TITLE, SUBTITLE*, (STAGEDIR | SPEECH)+)>
<!ELEMENT EPILOGUE (TITLE, SUBTITLE*, (STAGEDIR | SPEECH)+)>
<!ELEMENT SPEECH   (SPEAKER+, (LINE | STAGEDIR | SUBHEAD)+)>
<!ELEMENT SPEAKER  (#PCDATA)>
<!ELEMENT LINE     (#PCDATA | STAGEDIR)*>
<!ELEMENT STAGEDIR (#PCDATA)>
<!ELEMENT SUBTITLE (#PCDATA)>
<!ELEMENT SUBHEAD  (#PCDATA)>
```

Die generierte DTD:

```
<!ELEMENT TITLE    (#PCDATA)>
<!ELEMENT P        (#PCDATA)>
<!ELEMENT FM       (P,P*)>
<!ELEMENT PERSONA  (#PCDATA)>
<!ELEMENT GRPDESCR (#PCDATA)>
<!ELEMENT PGROUP   (PERSONA,PERSONA*,GRPDESCR)>
<!ELEMENT PERSONAE (TITLE,((PGROUP,(PGROUP|PERSONA)*)|
                   (PERSONA,(PGROUP|PERSONA)*)))>
<!ELEMENT SCNDESCR (#PCDATA)>
<!ELEMENT PLAYSUBT (#PCDATA)>
<!ELEMENT STAGEDIR (#PCDATA)>
<!ELEMENT SPEAKER  (#PCDATA)>
<!ELEMENT LINE     (#PCDATA|STAGEDIR)*>
<!ELEMENT SPEECH   (SPEAKER,SPEAKER*,((STAGEDIR,SUBHEAD?,
                   ((STAGEDIR,SUBHEAD?)|(LINE,SUBHEAD?))*)|
                   (LINE,SUBHEAD?,((STAGEDIR,SUBHEAD?)|
                   (LINE,SUBHEAD?))*)))>
<!ELEMENT SCENE    (TITLE,((STAGEDIR,STAGEDIR*,SUBHEAD?,
                   (SPEECH,((STAGEDIR,STAGEDIR*,SUBHEAD?)|SUBHEAD)?)*)|
                   (SPEECH,((STAGEDIR,STAGEDIR*,SUBHEAD?)|SUBHEAD)?,
                   (SPEECH,((STAGEDIR,STAGEDIR*,SUBHEAD?)|SUBHEAD)?)*)))>
<!ELEMENT ACT      (TITLE,((SCENE,SCENE*,EPILOGUE?)|
                   (PROLOGUE,SCENE,SCENE*,EPILOGUE?)))>
<!ELEMENT SUBHEAD  (#PCDATA)>
<!ELEMENT PLAY     (TITLE,FM,PERSONAE,SCNDESCR,PLAYSUBT,((ACT,ACT*)|
                   (INDUCT,ACT,ACT*)|(PROLOGUE,ACT,ACT*)))>
<!ELEMENT EPILOGUE (TITLE,((STAGEDIR,SPEECH,STAGEDIR?)|(SPEECH,STAGEDIR?)|
                   (SUBTITLE,SPEECH)))>
<!ELEMENT INDUCT   (TITLE,((SCENE,SCENE)|
                   (STAGEDIR,STAGEDIR,SPEECH,STAGEDIR)))>
<!ELEMENT PROLOGUE (TITLE,((STAGEDIR,STAGEDIR*,SPEECH,STAGEDIR)|SPEECH))>
<!ELEMENT SUBTITLE (#PCDATA)>
```

Anhang C

Interpretation der XML-Schema-Spezifikation

Die XML-Schema-Spezifikation [31] gehört zu den Spezifikationen mit den größten Präzisionsansprüchen und interessanterweise gleichzeitig zu denen mit den meisten Unklarheiten, sie wirft einige Fragen auf, die nicht unmittelbar beantwortet werden, soweit möglich werden diese Fragen an dieser Stelle ausdiskutiert, andernfalls wird die Semantik festgelegt:

- Obwohl in [31]; Abschnitt 2.2.2.2 geschrieben wird, daß nur Top-Level-Elementdeklarationen Mitglieder einer Ersetzungsgruppe (engl. substitution group) sein können, wird diese Restriktion an keine Stelle im formalen Teil – Kapitel 3 von [31] – wiedergegeben, es wird aus formaler Sicht nur sichergestellt, daß der *Kopf* einer Ersetzungsgruppe eine Top-Level-Elementdeklaration ist. Auch keine andersweitig Restriktion verhindert das zu meinem Wissen indirekt, denn zwar muß der Typ eines Mitglieds der Ersetzungsgruppe vom Typ des Kopfes *gültig abgeleitet* sein, das können wir aber z.B. dadurch sicherstellen, daß wir in beiden Fällen keine Inline-Typdefinition haben, sondern die selbe Typdefinition mittels `type="..."` referenzieren. Wir gehen hier aber davon aus, daß die Absicht aus Abschnitt 2.2.2.2 die maßgebende ist.
  Im Übrigen hat man die *sehr* umständliche Variante gewählt, daß Ersetzungsgruppen mindestens zwei Elemente haben, anstatt einfach generell auch ein-elementige Ersetzungsgruppen zuzulassen.
- Wo darf `all` auftreten?
  Aus Abschnitt 3.8.6 *Schema Component Constraint: All Group Limited* folgt u.a., daß zu `all` gehörende Modellgruppen nur innerhalb von `group` oder als oberstes bei `complexType` auftreten dürfen. In `sequence` oder auch `choice` dürfen aber beliebig Referenzen mit `group` genutzt werden und zwar beliebig oft und gemischt mit anderen Kindelementen wie `element`, `choice`, `sequence` und `any`. Es gibt keinerlei Restriktionen, daß hier `groups` die `all` enthalten nicht auftreten dürfen, de facto gibt es also keine Restriktion bzgl. des Auftretens von `all`, daß `all` widerrum nur `elemente` enthalten darf, ist natürlich klar.

  Das widerspricht aber wahrscheinlich der Intention, wir wollen annehmen, daß Modellgruppen mit `all` im abstrakten Datenmodell von XML-Schema nur als oberster Operator eines Inhaltsmodells stehen darf.
- es ist nicht klar, ob `anySimpleType` nur virtuell ist oder richtig existiert, was aber im Kontext dieser Arbeit nicht weiter relevant ist.
- bei Punkt 3.7 (`group`) sind bei der Version vom 30.März 2001 auf den 2.Mai 2001 Teile verlorengegangen, vgl. hierzu den Absatz über die XML-Darstellung, wo sich bei beiden Versionen z.B. auf das `minOccurs`-Attribut bezogen wird, dieses aber in der Version vom 2.Mai 2001 nicht mehr vorhanden ist.
- eine wichtige Frage wurde bei *www-xml-schema-comments@w3c.org* eingereicht, aber noch nicht beantwortet, hier ein Zitat aus der Anfrage vom 14.Juli 2001:

I would like to model the following in xml schema:

```
<!ELEMENT p (n,p?)>
<!ELEMENT n #PCDATA>
...
```

Two possibilities are obvious:

```
<element name="p" type="p_type"/>
<complexType name="p_type">
  <sequence>
    <element name="n" type="string"/>
    <element minOccurs="0" name="p" type="p_type" />
  </sequence>
</complexType>
```

and

```
<element name="p">
  <complexType>
    <sequence>
      <element name="n" type="string"/>
      <element minOccurs="0" ref="p"/>
    </sequence>
  </complexType>
</element>
```

In my opinion, at least one of them should be permitted by xml schema because otherwise xml schema would be a restriction of DTD regarding this aspect. But after reading the xml schema specification, it seems both are not permitted due to:

```
3.8.6 Constraints on Model Group Schema Components
All modelgroups (see Model Groups (§3.8)) must
satisfy the following constraints.

Schema Component Constraint: Model Group Correct

All of the following must be true: ... 2 Circular
groups are disallowed. That is, within the particles
of a group there may not be at any depth a particle
whose term is the group itself.
```

In both cases the <sequence> corresponds to a particle(#1) that contains a model group(#2):
first case: #2 contains two particles, one for the n-element (#3) and the other one for the p-element (#4), #4's term is an *Element Declaration Schema Component* (#5) #5's type definition is the type definition corresponding to the contained <complexType> (#6) #6's type definition is besides other properties the content type property, that contains in our case the value `element-only` and a particle (#7) that is the same as #1. So #2 contains itself at a certain depth.
second case: nearly like the first case, except that #4's term has to be resolved due to the presence of the `ref`. This is a global element declaration that corresponds to an *Element Declaration Schema Component* (#5) Besides that the argumentation is the same.
So the quoted restriction is applicable with the result, that both attempts to model the given part of a DTD are not permitted.

Diese ursprüngliche Restriktion bestand früher nicht, womit praktisch kontextfreie Grammatiken realisierbar waren (siehe hierzu Anmerkungen aus [23]) und ist zu restriktiv. Die Intention ist wohl vielmehr daß das `group`-Element eine rein syntaktische Abkürzung ist, sich also praktisch alle `group`-Elemente durch sukzessives Einsetzen der Inhalte eleminieren lassen.
Eine Antwort auf die Frage steht aber noch aus.

Dieser Anhang bezieht sich auf die XML-Schema Version von 2.Mai 2001.

Anhang D

Konvertierung DTD nach XML-Schema

Dieser Anhang enthält eine Algorithmusbeschreibung zur Konvertierung einer DTD in ein äquivalentes XML-Schema.

Wir nehmen an dieser Stelle an, daß z.B. *parameter entities* u.ä. schon aufgelöst wurden, vernachlässigen Namensräume, andere Enitity- und Notation-Definitionen, die sich aber prinzipiell auch mit XML-Schema darstellen oder simulieren lassen. O.B.d.A. sei D eine separate Datei und enthalte ausschließlich Produktionen der Art: <!ELEMENT ... > sowie <!ATTLIST ... >. Ebenso sei D syntaktisch korrekt und erfülle auch alle Restriktionen der XML Spezifikation.

S sei dann entsprechend wie folgt definiert:
Als Wurzelknoten (Sichtweise als Baum) enthalte S:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
</xsd:schema>
```

Für jede Produktion P :=<!ELEMENT e ...> existiert o.B.d.A. höchstens eine Produktion P_{att} :=<!ATTLIST e ...>, weiterhin haben alle ELEMENT-Produktionen paarweise verschiedene Elementnamen e.
Je nach Art von P wird ein bestimmter Knoten als Kindknoten von dem xsd:schema-Knoten in S aufgenommen, für die Erzeugung dieses Knotens (als Zeichenkettenfolge) wird die Funktion $Convert()$ definiert.
$Convert()$ bekommt eine Produktion der Form <!ELEMENT e r> übergeben, wobei r ein Inhaltsmodell ist. Desweiteren bekommt $Convert()$ die korrespondierende Produktion der Form <!ATTLIST e a> übergeben. $Convert()$ erzeugt eine Textdarstellung, kursive Teile bezeichnen weitere Funktionsaufrufe, deren (Text-)Ergebnisse anstelle der Aufrufe selbst in das Ergebnis übernommen werden.
Wenn zu dem Element e keine Attribute definiert sind, dann soll o.B.d.A. die Produktion <!ATTLIST e a> existieren, wobei $a = \epsilon$.

Fall 1: P =<!ELEMENT e EMPTY>

$Convert($<!ELEMENT e EMPTY>, <!ATTLIST e a>$) :=$

```
<xsd:element name="e" >
  <xsd:complexType>
    + convA(a) +
  </xsd:complexType>
</xsd:element>
```

Fall 2: P =<!ELEMENT e ANY>

$Convert($<!ELEMENT e ANY>, <!ATTLIST e a>$) :=$

```
<xsd:element name="e" >
  <xsd:complexType mixed="true">
    <xsd:any processContents="strict"/>
    + convA(a) +
  </xsd:complexType>
</xsd:element>
```

Fall 3: P =<!ELEMENT e r> und r ist o.B.d.A. [1] ein Term gem. der folgenden Grammatik für Inhaltsmodelle über allen in D auftretenden Elementnamen, wobei die Syntax im Gegensatz zur Definition regulärer Ausdrücke in [20] wie folgt sei:

Σ:=Menge aller Elementnamen

Term	::=	(Choice | Sequence) Count
Cp	::=	(Sigma | Choice | Sequence) Count
Choice	::=	'(' Cp ('|' Cp)+ ')'
Sequence	::=	'(' Cp (',' Cp)* ')'
Count	::=	('?' | '*' | '+')?

Bemerkung:
Man sieht sehr leicht, daß die Terme dieser Form sich auf normale reguläre Ausdrücke über Sigma abbilden lassen, wenn die Symbole aus Σ gleich den Elementnamen sind. Beachte außerdem die Eindeutigkeit der Grammatik. $Convert()$ wird für diesen Fall wie folgt definiert:

$Convert($<!ELEMENT e r>, <!ATTLIST e a>$) :=$

```
<xsd:element name="e" type="e" /> <xsd:complexType name="e" >
  + convT(r) +
  + convA(a) +
</xsd:complexType>
```

Beachte, daß XML-Schema die Restriktion enthält, daß wenn zwei Kindelemente gleichen Namens auftreten, diese denselben Typ haben müssen, wobei der Typ nicht inline angegeben werden darf.

$$ConvCount(c) := \begin{cases} \texttt{minOccurs="1" maxOccurs="1"} & c = \epsilon \\ \texttt{minOccurs="0" maxOccurs="1"} & c = ? \\ \texttt{minOccurs="0" maxOccurs="unbounded"} & c = * \\ \texttt{minOccurs="1" maxOccurs="unbounded"} & c = + \end{cases}$$

e sei Element Σ, w_i beliebige Zeichensequenz, mit der Einschränkung, wenn w_i die Zeichen ,|'oder ,,'enthält, dann beginnt und endet w_i mit ,('bzw. ,)'.

$ConvT((w_1, w_2, \ldots, w_n)\, c) :=$

```
<xsd:sequence + ConvCount(c) + >
    + ConvT($w_1$) +
    + ConvT($w_2$) +
    ...
    + ConvT($w_n$) +
</xsd:sequence>
```

$ConvT((w_1|w_2|\ldots|w_n)\, c) :=$

```
<xsd:choice + ConvCount(c) +>
    + ConvT(w_1) +
    + ConvT(w_2) +
    ...
    + ConvT(w_n) +
</xsd:choice>
```

[1] Die XML-Definition erlaubt an verschiedenen Stellen Spaces, also Leerzeichen u.ä. Diese haben keinen Einfluß auf die Semantik und können deshalb o.B.d.A. hier in der Betrachtung weggelassen werden.

$ConvT(ec) :=$

```
<xsd:element name="e" type="e" + ConvCount(c) + />
```

Fall 4: Im Gegensatz zu Fall 3, wo ein Element ausschließlich andere Elemente als Inhalt hat (sog. Element Content) gibt es noch den Fall des sog. Mixed Content Types, d.h. ein Element darf auch normale Zeichenketten als Inhalt haben. $P =$`<!ELEMENT` e r`>` und r ist o.B.d.A. ein $Term\_mixed$ der folgenden Grammatik:

Term_mixed ::= '(#PCDATA' ('|' Sigma)* ')*' | '(#PCDATA)'

Convert wird für diesen Fall erweitert:

$Convert($`<!ELEMENT e (#PCDATA)>, <!ATTLIST e a>`$) :=$

```
<xsd:element name="e" >
  <xsd:complexType mixed="true">
    + convA(a) +
  </xsd:complexType>
</xsd:element>
```

bzw. $Convert($`<!ELEMENT e (#PCDATA |e1|...|en)*>, <!ATTLIST e a>`$) :=$

```
<xsd:element name="e" type="e" /> <xsd:complexType name="e"
mixed="true">
  <xsd:sequence minOccurs="0" maxOccurs="unbounded" >
    <xsd:element name="e1" type="e1" />
     ...
    <xsd:element name="en" type="en" />
  </xsd:sequence>
  + convA(a) +
</xsd:complexType>
```

Bemerkung:
Die angegebenen Funktionen prüfen nicht auf syntaktische und semantische Korrektheit der übergebenen Argumente, diese wird jeweils vorausgesetzt.

Die Funktion $convA()$ dient der Umsetzung der Attributdeklarationen. Die Attributdeklarationen seien hier wiederum als syntaktisch richtig vorausgesetzt. Die Liste der Attributdeklarationen enthält Einträge der Art:

$Attributname\ Attributtyp\ Defaultwert$

Alle für DTDs definierten Attributtypen haben ihre Entsprechung in XML Schema:

CDATA	xsd:string
ID	xsd:ID
IDREF	xsd:IDREF
IDREFS	xsd:IDREFS
ENTITY	xsd:ENTITY
ENTITIES	xsd:ENTITIES
NMTOKEN	xsd:NMTOKEN
NMTOKENS	xsd:NMTOKENS
NOTATION	xsd:NOTATION

Es gilt generell: $Defaultwert \in \{\#REQUIRED, \#IMPLIED, \#FIXED\ \mathbf{Wert}, \mathbf{Wert}\}$

$$convDef(w) := \begin{cases} \texttt{use="required"} & \texttt{w ="\#REQUIRED"} \\ \texttt{use="optional"} & \texttt{w ="\#IMPLIED"} \\ \texttt{fixed=}Wert & \texttt{w ="\#FIXED } Wert\texttt{"} \\ \texttt{default=}Wert & \texttt{w ="}Wert\texttt{"} \end{cases}$$

Für $Attributtyp \in \{ID, IDREF, IDREFS, ENTITY, ENTITIES, NMTOKEN, NMTOKENS, NOTATION\}$ sei definiert:

$convA(\mathbf{a\,typ\,default}) :=$
`<xsd:attribute name="`a`" type="xsd:`typ`"` $+convDef(default)+$ `/>`

Für $Attributtyp = CDATA$:

$convA(\mathbf{a}\,CDATA\,\mathbf{default}) :=$
`<xsd:attribute name="`a`" type="xsd:string "` $+convDef(default)+$ `/>`

Und für $Attributtyp = Aufzählung$:

$convA(\mathbf{a(NMTOKEN_1, NMTOKEN_2, ..., NMTOKEN_N)default}) :=$

```
<xsd:attribute name="a" + convDef(default) + >
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="NMTOKEN1"/>
      <xsd:enumeration value="NMTOKEN2"/>
      ...
      <xsd:enumeration value="NMTOKEN_N"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:attribute>
```

Zustandsverschmelzung formal

Definition: Verschmelzen von Zuständen formal

Sei $M = (Z, \Sigma, \delta, z_0, E)$ der Ausgangsautomat mit $q_i, q_j \in Z$ und $q_i \neq q_j$.
Sei $M' = (Z', \Sigma, \delta', z_0', E')$ der Automat nach dem Verschmelzen von q_i und q_j.
Dann ist M' wie folgt definiert:
Sei $q_k \notin Z : Z' := Z \backslash \{q_i, q_j\} \cup \{q_k\}$
Ist $q_i \in E$ oder $q_j \in E$
  dann: $E' := E \backslash \{q_i, q_j\} \cup \{q_k\}$
  sonst: $E' := E$
Ist $q_i = z_0$ oder $q_j = z_0$
  dann: $z_0' := q_k$
  sonst: $z_0' := z_0$
Setze $\delta' := \emptyset$
$\forall (z_l, a, z_m) \in \delta$:
  Falls $\{z_l, z_m\} = \{q_i, q_j\}$ oder $(z_l = z_m$ und $z_l \in \{q_i, q_j\})$
  dann: $\delta' := \delta' \cup \{(q_k, a, q_k)\}$
  sonst: falls $z_l \in \{q_i, q_j\}$
    dann: $\delta' := \delta' \cup \{(q_k, a, z_m)\}$
    sonst: falls $z_m \in \{q_i, q_j\}$
      dann: $\delta' := \delta' \cup \{(z_l, a, q_k)\}$
      sonst: $\delta' := \delta' \cup \{(z_l, a, z_m)\}$

Literatur

[1] H. Ahonen. *Generating grammars for structured documents using grammatical inference methods*. PhD thesis, University of Helsinki, Department of Computer Science, 1996.
`http://citeseer.nj.nec.com/ahonen96generating.html`

[2] H. Ahonen, H. Mannila, and E. Nikunen. Generating grammars for SGML tagged texts lacking DTD. In *M. Murata and H. Gallaire, editors, Proceedings of the Workshop on Principles of Document Processing '94*, 1994.
`http://citeseer.nj.nec.com/article/ahonen94generating.html`

[3] Helena Ahonen. Automatic generation of SGML content models. *Electronic Publishing Origination, Dissemination, and Design*, 8(2/3):195–206, /1995.
`http://citeseer.nj.nec.com/ahonen95automatic.html`

[4] Helena Ahonen. Disambiguation of SGML content models. In *PODP*, pages 27–37, 1996.
`http://citeseer.nj.nec.com/ahonen97disambiguation.html`

[5] Helena Ahonen, Barbara Heikkinen, Oskari Heinonen, Jani Jaakkola, and Mika Klemettinen. Analysis of Document Structures for Element Type Classification. In *PODDP*, pages 24–42, 1998.
`http://citeseer.nj.nec.com/55535.html`

[6] Dana Angluin. Inductive inference of formal languages from positive data. *Information and Control*, 45:117–135, 1980.

[7] Henning Behme and Stefan Mintert. *XML in der Praxis*. Addison-Wesley, 1998.

[8] Ralf Behrens, Gerhard Buntrock, Carsten Lecon, and Volker Linnemann. Is XML really enough?
`http://citeseer.nj.nec.com/254519.html`

[9] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes, Mai 2001.
`http://www.w3.org/TR/xmlschema-2/`

[10] Tim Bray, Dave Hollander, and Andrew Layman. Namespaces in XML, January 1999.
`http://www.w3.org/TR/REC-xml-names`

[11] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (Second Edition), October 2000.
`http://www.w3.org/TR/REC-xml`

[12] A. Brüggemann-Klein and D. Wood. One-Unambiguous Regular Languages. Information and Computation, 140(2): 229–253 and 142(2): 182–206, 1998., 1998.
`http://citeseer.nj.nec.com/333579.html`

[13] David C. Fallside. XML Schema Part 0: Primer, Mai 2001.
`http://www.w3.org/TR/xmlschema-0/`

[14] Wenfei Fan and Leonid Libkin. On XML integrity constraints in the presence of DTDs. In *Symposium on Principles of Database Systems*, 2001.
`http://citeseer.nj.nec.com/432589.html`

[15] Henning Fernau. Approximative Learning of Regular Languages, 2001.

[16] Henning Fernau. Learning XML Grammars, 2001.
`http://citeseer.nj.nec.com/fernau01learning.html`

[17] Minos Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim. XTRACT: a system for extracting document type descriptors from XML documents. *Bell Labs Tech. Memorandum*, 1999.

[18] Minos N. Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim. XTRACT: A System for Extracting Document Type Descriptors from XML Documents. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, volume 29, pages 165–176. ACM, 2000.

[19] P. Grünwald. A minimum description length approach to grammar inference. In G. Scheler, S. Wernter, and E. Riloff, editors, Connectionist, Statistical and Symbolic Approaches to Learning for Natural Language, volume 1004 of Lecture Notes in AI, pages 203–216. Berlin: Springer Verlag., 1994.
`http://citeseer.nj.nec.com/grunwald94minimum.html`

[20] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesley, 1979.

[21] Meike Klettke. *Akquisition von Integritätsbedingungen in Datenbanken*. PhD thesis, Universität Rostock, Fachbereich Informatik, 1997. infix: DISBIS 51.

[22] Meike Klettke and Holger Meyer. Optimisation Techniques for Mapping documents to databases.
`http://wwwdb.informatik.uni-rostock.de/xml/devcon2001_lon%don_vortrag.ps`

[23] D. Lee, M. Mani, and M. Murata. Reasoning about XML Schema Languages using Formal Language Theory. Technical Report, IBM Almaden Research Center, RJ# 10197, Log# 95071, Nov. 2000.
`http://www.cs.ucla.edu/~dongwon/paper`

[24] Erkki Makinen. Inferring Uniquely Terminating Regular Languages from Positive Data. *Information Processing Letters*, 62(2):57–60, 1997.
`http://citeseer.nj.nec.com/makinen97inferring.html`

[25] Chuang-Hue Moh, Ee-Peng Lim, and Wee Keong Ng. DTD-Miner: A Tool for Mining DTD from XML Documents. In *WECWIS 2000*, pages 144–151, 2000.

[26] Stephen Muggleton. *Inductive Acquisition of Expert Knowledge*. Addison-Wesley, 1990.

[27] Esko Nuutila and Eljas Soisalon-Soininen. On Finding the Strongly Connected Components in a Directed Graph. *Information Processing Letters*, 49:9–14, 1993.
`http://citeseer.nj.nec.com/nuutila94finding.html`

[28] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 Specification, December 1999.
`http://www.w3.org/TR/html401`

[29] Uwe Schöning. *Theoretische Informatik - kurzgefaßt*. Spektrum akademischer Verlag, 3. edition, 1997.

[30] Keith E. Shafer. Creating DTDs via the GB-Engine and Fred.
`http://www.oclc.org/fred/docs/sgml95.html`

[31] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures, Mai 2001.
`http://www.w3.org/TR/xmlschema-1/`

Selbständigkeitserklärung

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

Rostock, 30.September 2001

Thesen

1. Für eine XML-Anwendung ist die Benutzung einer DTD bzw. eines XML-Schemas wünschenswert, um sicherzustellen, daß die zu verarbeitenden Dokumente gewissen Restriktionen an Struktur und Inhalt genügen.

2. Die automatische Generierung einer DTD bzw. eines XML-Schemas für eine Kollektion von XML-Dokumenten ist möglich.

3. Bei der Erschließung der DTD bzw. des XML-Schemas sind verschiedene Ansätze aus der Theorie des Lernens von Sprachen sinnvoll, u.a. das MDL-Prinzip oder die Festlegung von Sprachklassen, die identifizierbar in den Grenzen sind.

4. Eine große Schwierigkeit liegt in der Forderung der Nicht-1-Mehrdeutigkeit für die Inhaltsmodelle von XML.

5. Die Umwandlung eines endlichen Automaten in einen nicht-1-mehrdeutigen endlichen Automaten ist automatisiert möglich, hierbei findet i.allg. eine Generalisierung seiner Sprache statt.

6. Mehrere Ansätze (z.B. XTRACT) vernachlässigen die Nicht-1-Mehrdeutigkeit und kreieren somit ungültige Ergebnisse.

7. Der Algorithmus von Ahonen zum Erzeugen der Nicht-1-Mehrdeutigkeit bei endlichen Automaten ist fehlerhaft. Er wird in dieser Arbeit korrigiert und bewiesen.

8. Die beiden Definitionen der (k, h)-Kontextualität von Ahonen in [1] und [2] sind inkonsistent, was in dieser Arbeit gezeigt wird.

9. Diese Arbeit zeigt einige in der XML-Schema Spezifikation enthaltene Inkonsistenzen auf, welche aber korrigiert werden könnten.

10. Zu einer gegebenen XML-Kollektion ist es weiterhin möglich, alle maximalen Nicht-Schlüssel zu finden, wodurch dem Anwender sinnvolle Hilfestellungen zum Finden von Schlüsseln gegeben werden können.

11. Es gibt grundlegende Heuristiken zum Erschließen von Datentypen und Attributen.