Aus dem Institut für Informationssysteme
der Universität zu Lübeck
Direktor:
Prof. Dr. rer. nat. Volker Linnemann

Programmierung von XML-basierten Anwendungen unter Berücksichtigung der Sprachbeschreibung

Dissertation
zur
Erlangung der Doktorwürde
der Universität zu Lübeck
– Aus der Technisch-Naturwissenschaftlichen Fakultät –

Vorgelegt von
Sascha Martin Kempa
aus Berlin – Frohnau

Lübeck, Juli 2003

Zum Geleit

Innerhalb der letzten 10 Jahre hat das World-Wide Web eine beispiellose Entwicklung zu einem weltweiten Informationssystem vollzogen. Es ist aus vielen Bereichen nicht mehr wegzudenken und hat die tägliche Arbeit wesentlich verändert. Wichtige Informationen müssen nicht mehr mühsam telefonisch, per Email oder per Post angefordert werden, sondern können direkt interaktiv am Bildschirm abgefragt werden. Suchmaschinen erlauben das Auffinden der für die eigene Arbeit wichtigen Informationen.

Zur internen Darstellung von WWW-Seiten in Textform hat sich die Sprache HTML (**H**yper**t**ext **M**arkup **L**anguage) als Standard durchgesetzt. XML (E**x**tensible **M**arkup **L**anguage) als Verallgemeinerung von HTML spielt in zunehmendem Maße eine Rolle als Datenaustauschformat und zur Datenmodellierung.

Um WWW-Seiten schnell und übersichtlich zu entwerfen, werden geeignete Werkzeuge benötigt. Dies gilt insbesondere bei neueren Anwendungen, bei denen WWW-Seiten nicht statisch sind, sondern dynamisch bei jeder Anforderung durch einen Benutzer neu erzeugt werden. Beispiele sind Seiten für Börsenkurse oder für Wetterdaten. Diesen Anwendungen ist es gemeinsam, dass der Inhalt einer Webseite sich aus aktuellen, häufig veränderlichen Daten ergibt und daher ad hoc dynamisch erzeugt werden muss.

Die heute in der Praxis eingesetzten Werkzeuge für dynamische Web-Seiten sind unzureichend, da die Gültigkeit der generierten Seiten, d.h. die Korrektheit gemäß einer Sprachbeschreibung, im Allgemeinen nicht statisch am Generierungsprogramm abgelesen werden kann, sondern durch dynamische Testläufe überprüft werden muss. Dies gilt sowohl für HTML-Seiten als auch für XML-Dokumente. Wichtige Vertreter dieser Werkzeuge sind JAVA Servlets und JAVA Server Pages.

Hier setzt die Arbeit von Martin Kempa an. Es wird in der Arbeit die Sprache XOBE (**X**ML **OBJE**KTE) als Erweiterung der im WWW-Kontext inzwischen sehr weit verbreiteten objektorientierten Programmiersprache JAVA entwickelt. XOBE erlaubt eine einfache Implementierung von Anwendungen zur Generierung von XML-Dokumenten. HTML ist hierbei in der Form des XML-konformen XHTML ein wichtiger Spezialfall.

In XOBE wird die Gültigkeit der durch ein Programm generierbaren XML-Dokumente weitestgehend statisch garantiert. Dies geschieht dadurch, dass eine Sprachbeschreibung für XML-Dokumente, formuliert in XML Schema, direkt zur Typisierung verwendet wird. XML-Konstruktoren erlauben die Generierung neuer XML-Dokumentteile aus bereits vorher generierten Dokumentteilen. Hierdurch kann gewährleistet werden, dass ein XML-Konstruktor nur XML-Dokumentteile erzeugen kann, die dem zugrunde liegenden XML Schema in der Struktur entsprechen.

Für die Analyse der XML-Konstruktoren wird in der Arbeit ein geeignetes Typsystem formal entwickelt. Zur Typüberprüfung werden die aus der Literatur bekannten Heckengrammatiken (hedge grammars) herangezogen. Heckengrammatiken eignen sich in besonderer Weise zur Mo-

dellierung von XML-Sprachbeschreibungen. Der Algorithmus zur Typüberprüfung stellt eine Erweiterung und Modifizierung eines von Antimirov entwickelten Algorithmus zur Überprüfung von Ungleichungen von regulären Ausdrücken dar.

Zur Analyse und Traversierung von XML-Objekten verwendet die Arbeit die Sprache XPATH. Auch hier wird die Typinferenz formal definiert.

Die formal beschriebenen Algorithmen wurden implementiert und die Sprache XOBE im Rahmen eines Präprozessors für JAVA implementiert. Zwei Beispielanwendungen, nämlich die WML-Anbindung eines Medienarchivs und eine Übungsdatenverwaltung zeigen, wie man mit XOBE programmiert und wie die statische Korrektheit von generierten XML Strukturen gewährleistet werden kann.

Die Arbeit von Martin Kempa zeigt in hervorragender Weise, wie das praktische Problem der gültigen XML-Dokumente gelöst und durch Einsatz einer entsprechenden Theorie untermauert werden kann. Die Arbeit leistet einen herausragenden Beitrag zur sicheren Programmierung von Web-Anwendungen. Dies ist von besonderer Bedeutung angesichts der stürmischen und teilweise wenig systematischen Entwicklung im Bereich der Web-Programmierung.

Lübeck, im September 2003 Volker Linnemann

Danksagungen

Diese Arbeit ist das Resultat mehrerer langwieriger Forschungsphasen meiner gut fünf Jahre langen Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Informationssysteme der Universität zu Lübeck. Ausgehend von der Themensuche und der Einarbeitung in die Problemstellung, über die Erarbeitung von Lösungsideen und der Implementierung von Prototypen, bis hin zum Zusammenschreiben der Dissertation und dem Korrekturlesen habe ich vielfältige Unterstützung erfahren. An dieser Stelle möchte ich mich bei allen Beteiligten dafür bedanken.

Mein besonderer Dank gilt zunächst meinem Betreuer Prof. Dr. Volker Linnemann, in dessen Arbeitsgruppe diese Arbeit entstanden ist. Mit vielen inhaltlichen Diskussionen, Anregungen und Einwänden hat er meine Forschungsarbeit stets wohlwollend, aber inhaltlich kritisch begleitet. Herrn Prof. Dr. Walter Dosch danke ich für die Übernahme des zweiten Gutachtens, das mit einem nicht unerheblichen Arbeitsaufwand verbunden ist. Bei Prof. Dr. Jürgen Prestin möchte ich mich ebenfalls bedanken, der so freundlich war, den Vorsitz in der Prüfungskommission zu übernehmen.

Für anregende Diskussionen zum Inhalt der Arbeit geht mein Dank an meine Kollegen Beda Christoph Hammerschmidt und Sönke Magnussen. Für viele Verbesserungsvorschläge nach mühevoller Korrekturlesung gebührt der Dank Angela König, Henrike Schuhart und Torben Spiegler.

Abschließend möchte ich mich noch bei meiner Frau Susanne bedanken, ohne deren Rückhalt in allen weiteren Belangen des Lebens diese Arbeit nicht möglich gewesen wäre.

Lübeck, Juli 2003 S. Martin Kempa

Zusammenfassung

Die Kommunikation über das World-Wide Web mit Benutzern, seien es menschliche Anwender oder entfernt arbeitende Programme, wird in zunehmendem Maße zum integralen Bestandteil moderner Informationssysteme. Mit der Extensible-Markup-Language (XML) ist für den Austausch von Informationen über das Internet ein einheitliches Datenformat standardisiert worden, auf dessen Grundlage spezielle Auszeichnungssprachen für unterschiedliche Anwendungsgebiete definiert werden können. Heutige Web-Anwendungen zeichnen sich dadurch aus, dass sie in großem Umfang Dokumente einzelner Auszeichnungssprachen verarbeiten und dynamisch – also zur Laufzeit des Programmes – erzeugen. Die Implementierung dieser Web-Anwendungen erfolgt dabei in der Regel mit Werkzeugen, die die Korrektheit der erzeugten Dokumente nicht sicherstellen, was zusätzliche Testläufe notwendig macht. Es ist deshalb wünschenswert, eine Programmiersprache zur Verfügung zu haben, die die Kenntnis über die in einer Anwendung verwendeten Auszeichnungssprache nutzt, um fehlerfreie Anwendungen zu entwickeln.

In dieser Arbeit wird die Sprache XOBE (XML-Objekte), eine Erweiterung der objektorientierten Programmiersprache Java, vorgestellt, die eine einfache Implementierung von XML-basierten Anwendungen erlaubt. XML-Fragmente können dabei nach Deklaration der Sprachbeschreibung einer XML-Auszeichnungssprache im Programm als Instanzen von XML-Objekt-Klassen wie eingebaute Datentypen eingesetzt werden. Durch neu eingeführte Sprachkonstrukte ist es möglich, XML-Objekte zu erzeugen und Informationen oder Teile aus diesen zu selektieren. Der Vorteil der weitestgehenden Überprüfung der Gültigkeit für dynamisch erzeugte XML-Fragmente zum Zeitpunkt der Programmübersetzung wird bei diesem Ansatz im Gegensatz zu anderen Erweiterungen sichergestellt.

Die Analyse der Gültigkeit von XOBE-Programmen erfolgt mit dem auf XML-Typen zugeschnittenen Typsystem. Durch die aus der Literatur bekannten Heckengrammatiken ist es möglich, die durch die Sprachbeschreibung festgelegten XML-Typen, die im XOBE-Programm genutzt werden, zu formalisieren. Auf dieser Basis kommt zur Überprüfung einzelner Programmanweisungen ein neu entwickelter Subtyp-Algorithmus zum Einsatz. Die prototypische Implementierung der XOBE-Spracherweiterung, die als Präprozessor realisiert wurde, transformiert den XOBE-Quelltext in reines Java. Zur Repräsentation der XML-Objekte wird dabei der Schnittstellenstandard Dokument-Objektmodell (DOM) eingesetzt.

Die Programmiersprache XOBE ist besonders gut geeignet, Web-Anwendungen und Web-Services zu erstellen, die über das Internet zugreifbar sind. Dies wurde im Rahmen dieser Arbeit durch die Implementierung zweier prototypischer Web-Anwendungen bestätigt, die zusätzlich zeigen, dass mit XOBE Quelltexte entstehen, die im Vergleich zu Alternativen verständlicher und leichter zu warten sind. Damit leistet die Arbeit einen wichtigen Beitrag für die strukturierte Entwicklung korrekter XML-basierter Anwendungsprogramme.

Inhaltsverzeichnis

Kapitel 1

Einführung

Das World-Wide Web ist seit nahezu 10 Jahren das weltumspannende Informationssystem. Mit einer unüberschaubar großen Zahl von Rechnern tragen Anbieter aus allen Bereichen der Gesellschaft zum Informationssystem bei und eine noch viel größere Anzahl von Endanwendern nutzt das World-Wide Web täglich zur Informationsgewinnung. Die ersten Jahre des Webs waren geprägt von der Präsentation von Daten und Dokumenten in Form von statischen Hypertexten, die vom Anbieter für den Benutzer zur Verfügung gestellt wurden.

Populär wurden diese Systeme zu einem Zeitpunkt, als die Annahme vorherrschte, dass Änderungen an Dokumenten im World-Wide Web nicht oder nur selten vorgenommen werden. Diese Voraussetzung wurde mit der Zeit immer mehr abgeschwächt: Mehr und mehr Daten, die über das Web für den Benutzer zugänglich sind, erfordern eine dynamische Anpassung der Dokumente oder sogar den Dialog mit dem Benutzer. Möchte ein Anbieter beispielsweise eine Seite ins Web einspeisen, die die aktuellen Börsenkurse angibt, so würde der Inhalt dieses Dokuments im Verlaufe eines Tages ständig variieren. Eine der erste Anwendungen, die einen Dialog mit dem Endanwender benötigte, ist die Suchmaschine, die Anfragen nach Dokumenten mit gesuchtem Inhalt im Web beantworten kann.

Die neuen Anforderungen an das World-Wide Web führten zur Entwicklung von separaten, unabhängigen Web-Anwendungen einzelner Anbieter, die auf spezifische Aufgaben zugeschnitten sind. Diese Anwendungen sind durchaus vergleichbar mit traditionellen Informationssystemen, von denen sie sich im Wesentlichen durch die Kommunikation mit dem Endbenutzer über das World-Wide Web unterscheiden.

Weiterhin ist die Verknüpfung von Web-Anwendungen mit traditionellen Datenbanksystemen oder gar die Einbindung von Web-Anwendungen in eine bestehende Informationssysteminfrastruktur zu beobachten. Die Web-Anwendung dient dann als Schnittstelle zum Endbenutzer, während aus Benutzersicht das Datenbank- oder Informationssystem im Hintergrund der Anwendung wirkt. Viele der leistungsfähigsten Anwendungen im World-Wide Web nutzen inzwischen diese Möglichkeit. Die Web-Anwendungen der Banken ermöglichen inzwischen die Abwicklung

fast sämtlicher Bankgeschäfte über das World-Wide Web. Eine weitere umfangreiche Anwendung, die an ein bestehendes Informationssystem angebunden wurde, ist die Fahrplanauskunft der Bahn mit gleichzeitiger Einkaufsmöglichkeit der Fahrkarte.

Vergleichbar ist der aktuelle Stand der Programmierung von Web-Anwendungen mit den Anfangstagen des Einsatzes elektronischer Datenverarbeitung, in denen Programme ausschließlich von wenigen Spezialisten erstellt werden konnten: Eine Vielzahl unterschiedlicher Werkzeuge und Technologien sind notwendig, um eine leistungsfähige Web-Anwendung zu erstellen. Teilweise muss für ähnliche Aufgaben, abhängig davon, ob diese auf dem Rechner des Anbieters oder dem Rechner des Anwenders ausgeführt werden, auf unterschiedliche Programmiersprachen und -techniken zurückgegriffen werden.

In Zukunft soll die Idee der modularen Softwarekomponenten in Web-Anwendungen einfließen. Anstelle von eigenständigen, monolithischen Programmen soll dann eine Web-Anwendung aus einer Vielzahl kleiner, unabhängiger Softwarebausteine bestehen, die über das World-Wide Web hinweg austauschbar sind. Diese Web-Services werden dafür im Web zentral registriert und können anschließend bei Bedarf von anderen Web-Anwendungen eingesetzt werden.

Als Fernziel der Entwicklung von Web-Anwendungen ist wohl der persönliche, virtuelle Rechner im Web zu nennen. Jeder Benutzer hat dann auf einer von ihm frei wählbaren Oberfläche alle für ihn relevanten Anwendungen jederzeit und überall verfügbar. Einen ersten Schritt in diese Richtung stellen die omnipräsenten Emaildienste im Web dar, über die von jedem ans World-Wide Web angeschlossenen Rechner die persönliche Email gelesen und verschickt werden kann.

1.1 Motivation für statische Typüberprüfung

Von den erwähnten Fortschritten im Bereich der Web-Anwendungen sind alle Komponenten, aus denen ein solches System besteht, betroffen: Die Verwaltung der Daten in einem Datenbanksystem ist ebenso zu überdenken wie die Kommunikation zwischen den Anbietern und dem Endbenutzer. Gleiches gilt für die Programmarchitektur und insbesondere für die *Programmiersprachen* zum Implementieren der Anwendungen. Die Anpassung von Programmiersprachen auf die neuen Erfordernisse von Web-Anwendungen bilden den Schwerpunkt dieser Arbeit.

Die Entwicklung der Web-Anwendungen beschleunigte die Etablierung des Datenformats *Extensible-Markup-Language* (XML) zum Standard für das World-Wide Web. Da es sich bei XML um ein universelles Datenbeschreibungsformat handelt, war es möglich, auf der Basis von XML verschiedene Auszeichnungssprachen für die unterschiedlichsten Anwendungsgebiete zu definieren, die von der Präsentation von Dokumenten im Web bis zum simplen Austausch von Geschäftsdaten reichen. Standardisierungsgremien treiben den Entwicklungsprozess von XML-basierten Standards intensiv voran: So wurden bereits eine Sprache für Verweise, eine Selektionssprache und eine Transformationssprache für XML standardisiert; gearbeitet wird zur Zeit unter anderem an einer Anfragesprache für XML-Datenbanksysteme. Ähnlich rasant verläuft der

Prozess bei den Softwareherstellern, die versuchen, ihre aktuellen Produkte um Zugriffsmöglichkeiten über XML zu erweitern oder neue XML-spezifische Werkzeuge anzubieten.

Moderne Web-Anwendungen und Web-Services erzeugen im großen Maße XML-Dokumente dynamisch. Der Inhalt dieser Dokumente wird im Vergleich zu den traditionellen, statischen Web-Seiten, die für alle Benutzer zu jedem Zeitpunkt gleich sind, erst zur Laufzeit der Web-Anwendung erzeugt. Die Implementierungen dieser Web-Anwendungen und Web-Services erfolgt heutzutage durch den Einsatz von Standard-Programmiersprachen wie Java oder Visual Basic. Diese werden von den Softwareherstellern mit zusätzlichen Technologien ausgestattet, die die Programmiersprachen um Fähigkeiten zur einfacheren dynamischen Erzeugung von XML-Dokumenten erweitern.

Der Einsatz dieser Technologien garantiert die Korrektheit der dynamisch generierten Dokumente nicht oder nur bis zu einem sehr begrenzten Grad. Für das erzeugte XML wird in der Regel nur sichergestellt, dass es wohlgeformt ist, es sich also wirklich um ein XML-Dokument handelt. Dies war in Zeiten, in denen die Datenbestände des World-Wide Web fast ausschließlich statische Dokumente umfasste, auch sinnvoll und ausreichend. Da aber moderne Web-Anwendungen im großen Umfang XML-Dokumente bestimmter Auszeichnungssprachen erzeugen, ist festzustellen, dass diese Technologien den daraus erwachsenden neuen Anforderungen nicht mehr gerecht werden. Es reicht also nicht nur aus, zu überprüfen, ob es sich bei der generierten Struktur um ein XML-Dokument handelt, vielmehr muss auch garantiert werden, dass das Dokument zu einer definierten Auszeichnungssprache gehört, also gültig ist. Diese Eigenschaft der Gültigkeit muss bei den verfügbaren Technologien aber für jede Web-Anwendung durch aufwendige Testläufe nachgeprüft werden.

Um das genaue Problem der Generierung von gültigen XML-Dokumenten zu illustrieren, sei als Beispiel die Erzeugung einer Begrüßung genannt. Mit dem Einsatz der Technologie *JavaServer-Pages* (JSP) lautet ein solches Programm wie folgt:

```
<html>
    <head><title>A Simple Server Page</title></head>
    <body>
    <% if (Calendar.getInstance().get(Calendar.AM_PM) ==
       Calendar.AM) { %>
       <ul><li>Good Morning</li></ul>
    <% } else { %>
       <ul><li>Good Afternoon</li></ul>
    <% } %>
    </body>
</html>
```

Abhängig von der aktuellen Uhrzeit erzeugt diese simple Web-Anwendung ein Dokument mit unterschiedlichen Begrüßungstexten für den Vor- und Nachmittag. Der Interpreter für JSP akzeptiert dieses korrekte Programm. Die beiden durch das Programm generierten XML-Dokumente sind ebenfalls gültig gemäß der verwendeten Auszeichnungssprache.

Angenommen der Programmierer hätte nun vergessen, den Begrüßungstext in das XML-Element `li` einzufügen, ergibt sich der folgende Quelltext:

```
<html>
   <head><title>A Simple Server Page</title></head>
   <body>
   <% if (Calendar.getInstance().get(Calendar.AM_PM) ==
      Calendar.AM) { %>
      <ul>Good Morning</ul>
   <% } else { %>
      <ul>Good Afternoon</ul>
   <% } %>
   </body>
</html>
```

In diesem Fall würde der JSP-Interpreter erneut das Programm als korrekt akzeptieren, obwohl zur Laufzeit XML-Dokumente erzeugt werden, die ungültig sind, denn innerhalb eines Elements mit dem Namen `ul` muss für die gezeigte Auszeichnungssprache XHTML mindestens ein `li`-Element folgen. Fehler dieser Art sind offensichtlich schon frühzeitig erkennbar. Es wird damit deutlich, dass die Korrektheit der erzeugten XML-Dokumente mit JSP nicht in dem Maße sichergestellt wird, wie es möglich und sinnvoll wäre.

Nicht neu ist das Anliegen nach der Unterstützung von XML durch eine Programmiersprache: XML-Dokumente sollen in einer Programmiersprache nicht nur auf einfache Weise generiert werden können, sondern sollten gleichzeitig die Eigenschaft Gültigkeit für die erzeugten Dokumente weitgehend bereits zur Zeit der Programmübersetzung sicherstellen. Mit der Programmiersprache Java – und den bisher vorgenommenen Erweiterungen dieser Sprache – ist dieses Ziel nicht erreicht worden.

Ansätze aus jüngster Zeit zur Bewältigung dieses Problems weisen indes den Nachteil auf, dass die Gültigkeit der erzeugten Dokumente nur relativ eingeschränkt garantiert werden kann. Sie generieren aus der Sprachbeschreibung der Auszeichnungsprache Datentypen der verwendeten Programmiersprache, womit die Bedingungen der Auszeichnungssprache nun durch das Typsystem getestet werden können. Die Genauigkeit dieser Überprüfung, die zur Zeit der Programmübersetzung abläuft, beruht nun im Wesentlichen auf den Möglichkeiten des Typsystems der Programmiersprache und einer möglichst guten Abbildung der Bedingungen der Auszeichnungssprache in das Typsystem. Die meisten Abbildungen von Sprachbeschreibungen in Datentypen sind allerdings mit einem Verlust von Semantik verbunden, denn einige Eigenschaften der Gültigkeit sind nicht oder nur mit großen Umständen durch das Typsystem einer Standard-Programmiersprache ausdrückbar. Weiterhin ist dieses Vorgehen mit einem hohen Einarbeitungsaufwand für den Programmierer verbunden, dem sowohl die Kenntnis der Sprachbeschreibung der Auszeichnungssprache als auch das Wissen über die daraus erzeugten Datentypen abverlangt wird. Auch muss die Transformation für jede neue Auszeichnungssprache und für jede noch so kleine Änderung an einer vorhandenen Sprachbeschreibung erneut durchgeführt werden.

Zusätzlich unterschieden diese Techniken zwischen einer Repräsentation von XML-Dokumenten in Form von Zeichenketten und der Repräsentation durch Instanzen der aus der Sprachbeschreibung erzeugten Datentypen, wodurch eine explizite Konvertierung zwischen diesen beiden Darstellungen erforderlich wird. Besonders umständlich wird dieser Ansatz bei der Verwendung größerer konstanter XML-Dokumente, die entweder auf objektorientiertem Wege sehr mühsam erzeugt werden müssen oder durch die Konvertierung einer eingelesenen Zeichenkette erstellt werden können. Die unterschiedliche Darstellung von XML-Dokumenten ist ein ernster Bruch in dem eingesetzten objektorientierten Programmierparadigma. Wünschenswert ist deshalb eine Integration von XML-Dokumenten in eine Programmiersprache, die nur eine Repräsentation vorsieht.

Aus diesen Gründen und wegen dem eingangs geschilderten Wandel von statischen Web-Dokumenten zu komponentenbasierten Web-Anwendungen ergibt sich die Forderung nach leicht verwendbaren Programmiersprachen, die die Korrektheit der generierten XML-Dokumente bereits zur Zeit der Programmübersetzung sicherstellen.

Java ist zur Zeit die Programmiersprache der Wahl, wenn es um die Entwicklung von Web-Anwendungen geht. Sie bietet sich daher für eine Modifikation geradezu an. Die in dieser Arbeit vorgestellte Lösung präsentiert deshalb eine objektorientierte Integration von XML-Dokumenten in die Programmiersprache Java.

1.2 Zielsetzung und Aufbau der Arbeit

In dieser Arbeit wird eine Erweiterung für die Programmiersprache Java definiert und als Präprozessor implementiert, die die unterschiedlichen Repräsentationen von XML-Dokumenten in Form von Zeichenketten und durch eine Struktur von Objekten überwindet. Diese Java-Erweiterung – **X**ML-***Objekte*** (XOBE) – erlaubt es unter anderem erstmals mit einem erweiterten Typsystem die Gültigkeit der generierten XML-Dokumente bereits zur Zeit der Programmübersetzung so weit wie möglich sicherzustellen. Dabei werden die XML-Dokumente im XOBE-Programm ausschließlich durch XML-Syntax notiert.

Die Ziele dieser Erweiterung sind im Einzelnen:

1. Integration von XML-Dokumenten in das objektorientierte Klassenkonzept,
2. komfortable Zugriffsmöglichkeiten auf den Inhalt dieser XML-Dokumente und
3. weitestgehende Garantie der Gültigkeit der generierten Dokumente bereits zur Zeit der Programmübersetzung.

Das vorherige Beispiel, in dem ein XML-Dokument generiert werden soll, das einen von der aktuellen Uhrzeit abhängigen Begrüßungstext enthält, kann dann wie folgt formuliert werden:

```
ximport xhtml-transitional.dtd;
```

Als erstes muss im XOBE-Programm deklariert werden für welche Sprachbeschreibung XML-Dokumente verarbeitet werden, was durch das neue Schlüsselwort `ximport` erfolgt.

```
html welcomePage() {
  ul phrase;

  if (Calendar.getInstance().get(Calendar.AM_PM) ==
     Calendar.AM)
    phrase = <ul><li>Good Morning</li></ul>;
  else
    phrase = <ul><li>Good Afternoon</li></ul>;

  return <html>
           <head>
              <title>A Simple Server Page</title>
           </head>
           <body>{phrase}</body>
         </html>;
} // welcomePage
```

Im Anschluss kann eine Methode definiert werden, die das XML-Dokument, ein XML-*Objekt*, als Resultat zurückliefert. Durch die Verwendung von XML-Syntax werden in XOBE stets XML-Objekte erzeugt, das heißt, das Generieren und Analysieren von XML geschieht konzeptuell ausschließlich auf der Ebene von Objekten. Deshalb führt XOBE für jeden Elementtyp einer vereinbarten Sprachbeschreibung eine eigene Klasse ein, die nach der Deklaration wie eingebaute Klassen oder atomare Datentypen benutzt werden können. Eine explizite Generierung von Java-Klassen aus der Sprachbeschreibung entfällt deshalb. Durch den Bezeichner aus der Sprachbeschreibung wird eine solche Klasse angesprochen, wie es bei einer Variablen- oder Methodendeklaration nötig ist. XML-Objekte können wie alle Objekte in Java an Variablen zugewiesen und manipuliert werden; zusätzlich ist ein Einfügen in den Inhalt eines neuen XML-Objekts möglich.

Mit XOBE kann die Gültigkeit für generierte XML-Dokumente weitgehend bereits zur Zeit der Programmübersetzung sichergestellt werden. Dadurch ergeben sich für den Programmierer von Web-Anwendungen gegenüber der herkömmlichen Entwicklung die folgenden Vorteile:

1. XOBE-Programme sind effizienter, weil weniger dynamische Typumwandlungen und Überprüfungen der Gültigkeit zur Laufzeit benötigt werden.

2. Ein Programm in XOBE ist zuverlässiger, da auf die Programmierung von Recovery-Prozeduren, die nötig sind, um Fehler bei Typumwandlungen oder Gültigkeitsüberprüfungen abzufangen, verzichtet werden kann.

3. XOBE erlaubt eine schnellere Entwicklung von Implementierungen, weil intensive Testläufe wegfallen, die nötig sind, um die Korrektheit der dynamisch erzeugten XML-Dokumente plausibel zu machen.

4. Web-Anwendungen und Web-Services, die in XOBE implementiert wurden, sind besser zu warten, da die Programmstruktur der Quelltexte einfacher und übersichtlicher gegliedert ist.

Gliederung

Dieser Einführung folgt im Anschluss Kapitel 2, das die Grundlagen von Web-Anwendungen darlegt. Es beginnt dabei zunächst mit XML und den damit verbundenen Möglichkeiten zur Sprachbeschreibung von Auszeichnungssprachen. Anschließend werden zwei eng mit XML verbundene Standards vorgestellt; *XPath* dient zur Selektion von Inhalten aus einem XML-Dokument, während das *Dokument-Objektmodell* (DOM) die Schnittstelle für Programmiersprachen zu XML darstellt. Auf die Verarbeitung von syntaktischen Strukturen wird anschließend ebenfalls eingegangen. Das wichtigste globale Informationssystem ist mit seinen Web-Anwendungen zur Zeit das World-Wide Web, dessen Architektur mit den verschiedenen technologischen Möglichkeiten zur Programmierung von Web-Anwendungen in Abschnitt 2.5 erläutert wird. Dabei werden die wichtigsten Implementierungstechniken für Web-Anwendungen vorgestellt, die von rein statischen Dokumenten, über eine dynamisierte Benutzerseite bis hin zu vollwertigen Anwendungen auf der Anbieterseite reichen. Den Schluss des Kapitels bildet die Einordnung der vorliegenden Arbeit in den Kontext der beschriebenen Forschungsarbeiten.

Nach den Grundlagen folgt der Schwerpunkt dieser Arbeit, der in drei Kapitel unterteilt ist: In Kapitel 3 wird die Spracherweiterung XML-*Objekte* (XOBE) der Programmiersprache Java vorgestellt, die es erlaubt mit XML-Dokumenten in Java auf objektorientierte Weise zu arbeiten. XML-Dokumente werden in XOBE durch XML-Objekte repräsentiert, deren Klassen durch die Deklaration der Sprachbeschreibung der verwendeten Auszeichnungssprache automatisch bekannt sind. Für die Erzeugung von XML-Objekten und den Zugriff auf deren Inhalt werden neue Sprachkonstrukte definiert.

Kapitel 4 formalisiert das XOBE zu Grunde liegende Typsystem und stellt einen Algorithmus vor, mit dem es möglich wird, die Gültigkeit der in einem XOBE-Programm verarbeiteten XML-Objekte bereits zur Zeit der Programmübersetzung weitestgehend sicherzustellen. Es wird bewiesen, dass der Algorithmus für das Typsystem in XOBE korrekt arbeitet und stets terminiert.

Kapitel 5 definiert die Transformation der XOBE-Programme in reine Java-Programme. Dazu müssen die XML-Objekte in Java repräsentiert werden, was in dieser Arbeit mit dem DOM geschieht. Zusätzlich ist eine Abbildung der neu definierten Sprachkonstrukte notwendig. Das Kapitel endet mit der Präsentation von ersten Messdaten der Rechenleistung des im Rahmen dieser Arbeit implementierten Prototypen.

In Kapitel 6 wird die Praxistauglichkeit der vorgestellten Spracherweiterung durch die Implementierung zweier Web-Anwendungen untersucht. Die Arbeit schließt mit einer Zusammenfassung und dem Ausblick.

Kapitel 2

Grundlagen und verwandte Arbeiten

Die Extensible-Markup-Language (XML) bietet die Möglichkeit, für die verschiedensten Anwendungsgebiete eigene Auszeichnungssprachen zu definieren. Damit ist es für unterschiedliche Anwendungen möglich, über ein standardisiertes Datenformat systemübergreifend zu kommunizieren. Die Daten werden dabei in Form von Dokumenten ausgetauscht. Dokumente der gleichen Art werden zu einer Auszeichnungssprache zusammengefasst, die mittels einer Sprachbeschreibung definiert wird.

In diesem Kapitel werden die grundlegenden Begriffe aus dem Bereich der Extensible-Markup-Language eingeführt, die zum Verständnis der vorliegenden Arbeit notwendig sind. Für die Beschreibung von Auszeichnungssprachen werden zusätzlich die erweiterten Konzepte von XML-Schema vorgestellt. Zudem werden die Möglichkeiten von XPath zur Selektion von Daten aus einem Dokument dargelegt, sowie der Ansatz des Dokument-Objektmodells zur Repräsentation von XML im Programm präsentiert. Das verwandte Gebiet der Programmgenerierung ist Gegenstand des Abschnitts 2.4.

Im Anschluß daran wird das World-Wide Web (WWW) mit seinen Grundlagen als globales Datenhaltungssystem erläutert. Mit der Hypertext-Markup-Language ist das WWW die größte Anwendung von XML. Da es sich herausstellt, dass die statischen Präsentationsmöglichkeiten im WWW nicht den neuen Anforderungen genügen, die durch dynamisch zu erstellende Dokumente entstehen, werden danach Ansätze vorgestellt, die die Programmierung von Web-Anwendungen zum Ziel haben. Diese generieren und verarbeiten vielfach XML, weshalb im Anschluss Möglichkeiten zur Repräsentation von XML behandelt werden.

Nach einer Diskussion der Vor- und Nachteile der vorgestellten Ansätze mit einer anschließenden Einordnung der vorliegenden Arbeit in diesen Kontext wird dieses Kapitel beendet.

2.1 Extensible-Markup-Language

Die *Extensible-Markup-Language* (XML) [W3C98c] ist seit 1998 eine Empfehlung des World-Wide Web-Konsortiums (W3C), das mit seinen Empfehlungen De-Facto-Standards für das World-Wide Web setzt. Die Darstellung dieses Abschnitts folgt [ABS00]. Sie ist keine umfassende Beschreibung von XML sondern stellt den für das Verständnis dieser Arbeit nötigen Teil vor. Eine vollständige Definition liegt mit der Spezifikation [W3C98c] vor; eine ausführliche Behandlung des Themas findet sich ebenfalls in [Bra98, GP00].

Die Extensible-Markup-Language baut auf den Erfahrungen der Standard-Generalized-Markup-Language (SGML) [Gol90, Fly98] auf, einer gut 15 Jahre alten Entwicklung aus dem Bereich der Dokumentenverarbeitung, die inzwischen als ISO Standard (ISO 8879) vorliegt. Die Grundidee dieses Vorläufers besteht darin, die logische Struktur eines Dokuments konsequent von der Gestaltung für eine Präsentation des Dokuments, sei es an einem Bildschirm oder auf einem Drucker, zu trennen. Eine Anforderung, die für eine der wesentlichen Anwendungen von SGML, dem Austausch von Dokumenten im Verlagswesen, maßgeblich ist. Die Ausbreitung des World-Wide Webs und damit der Hypertext-Markup-Language (HTML), einer weiteren Anwendung von SGML, sorgt für eine erste Verschiebung des Anwendungsgebietes vom reinen Dokumentenaustausches hin zum Datenaustausch. Diese Verschiebung führt schließlich zur Spezifikation von XML als vereinfachte Variante von SGML.

Im Kern besteht XML aus nichts anderem als einer Syntax zum Austausch von Daten. Es gewinnt erst dadurch an Bedeutung, dass diese Syntax standardisiert ist und in einer Vielzahl von Gebieten und Programmen Anwendung findet. Beispielsweise bietet XML für eine Organisation oder Benutzergruppe die Möglichkeit, den Datentransfer zu spezifizieren, um Daten zwischen verschiedenen Anwendungen auszutauschen. Durch die breite Unterstützung, die XML zur Zeit erfährt, ist es sehr wahrscheinlich, dass XML in der nahen Zukunft zum Standard für den Datenaustausch im WWW wird.

Eine der Anforderungen an XML besteht darin, dass Dokumente für den Menschen lesbar sein sollen. Aus diesem Grund wird XML textuell repräsentiert. Ihre Struktur erhalten XML-Dokumente durch *Elemente*. Elemente beginnen stets mit einem *Start-Tag*, z. B. `<book>`, und enden mit einem *End-Tag*, beispielsweise `</book>`. Diese Tags werden auch Textauszeichnungen genannt. Zwischen einem Start- und einem End-Tag kann textueller Inhalt, also *Zeichendaten*, weitere Elemente oder eine Mischung aus beidem stehen. Ein Element besteht somit aus dem Start- und dem End-Tag, sowie dem Text und der Struktur zwischen den beiden Tags, dem sogenannten *Inhalt*. Steht ein Element im Inhalt eines anderen Elements spricht man von einem *Subelement*. Für Elemente, deren Inhalt leer ist, existiert mit `<offer/>` eine abkürzende Schreibweise; Start- und End-Tags werden also in einem Tag zusammengefasst. Mit den *Elementnamen* innerhalb der Tags und der Struktur des Inhalts werden die einzelnen Elemente in *Elementtypen* unterschieden. Die in einem Dokument auftretenden Elementtypen werden dabei vom Benutzer selbst definiert.

Ein weiterer Bestandteil von XML-Dokumenten sind *Attribute*, die den Elementen zugeordnet sind. Attribute bestehen aus einem Namen und einem Wert und werden innerhalb eines Start-

Tags angegeben. In dem Beispiel `<price currency="EUR">` wird für das Element `price` das Attribut `currency` auf den Wert `EUR` gesetzt. Analog zu den Elementtypen werden die Attributnamen, die Werte die die Attribute annehmen können sowie die Zuordnung zu den verschiedenen Elementtypen als *Attributtypen* ebenfalls vom Benutzer definiert. Zusätzlich können XML-Dokumente mit *Kommentar* versehen werden, z. B. durch `<!-- verkauft -->`.

Damit es sich bei einem Dokument um ein XML-Dokument handelt, müssen einige Bedingungen erfüllt sein. Zunächst müssen alle Elemente korrekt geschachtelt sein und damit eine klammerartigen Struktur bilden. Weiterhin müssen Attribute eindeutig sein. Das bedeutet, dass jedes Attribut in einem Element nur einmal auftreten darf. Dadurch unterscheiden sich Attribute wesentlich von Elementen, denn im Gegensatz zu Attributen dürfen Subelemente innerhalb eines Elements mehrfach vorkommen. Ein weiterer Unterschied besteht darin, dass die Werte von Attributen keine Elemente enthalten dürfen. Sind alle angesprochenen Anforderungen erfüllt, wird von einem *wohlgeformten* Dokument gesprochen. Die Eigenschaft wohlgeformt stellt eine ziemlich schwache Bedingung an XML-Dokumente, denn es wird lediglich sichergestellt, dass sich ein eingelesenes Dokument in einer baumartigen Struktur repräsentieren lässt. Die folgende vereinfachte Grammatik beschreibt den Aufbau von XML-Dokumenten.

Definition 2.1 (XML-Dokument)
Ein XML-*Dokument* ist nach folgender Grammatik aufgebaut:

<*Document*>	→	<*Element*>
<*Element*>	→	<*EmptyElementTag*> | <*STag*> <*Content*> <*ETag*>
<*STag*>	→	"<" <*Name*> (<*Attribute*>)* ">"
<*Attribute*>	→	<*Name*> "=" <*AttValue*>
<*ETag*>	→	"</" <*Name*> ">"
<*Content*>	→	(<*Element*> | <*CharData*> |<*Comment*>)*
<*EmptyElementTag*>	→	"<" <*Name*> (<*Attribute*>)* "/>"
<*Comment*>	→	"<!--" <*CharData*> "-->"

Das Nichtterminalsymbol <*AttValue*> steht hier für eine Zeichenkette in einfachen (') oder doppelten Hochkommata ("), <*Name*> für einen Elementname und <*CharData*> für alphanumerische Zeichendaten. □

Als durchgehendes Beispiel dient in dieser Arbeit das folgende Szenario; es zeigt ein Beispiel für ein XML-Dokument.

Beispiel 2.1
Eine Anwendung realisiert ein zentrales Verzeichnis antiquarischer Bücher, zu dem eine große Anzahl von unabhängigen Antiquariaten mit ihren Angebotslisten beitragen. In dem Verzeichnis können Benutzer der Anwendung nach Büchern suchen und erhalten Informationen über die Zustände der gefundenen Exemplare sowie über die von den Antiquariaten festgelegten Preise. Besteht ein Kaufinteresse von Seiten des Benutzers, kann er den oder die Titel in einen Einkaufskorb ablegen und bestellen. Die Bestellung wird von der Anwendung an das entsprechende Antiquariat weitergeleitet, welches sich dann um die Auslieferung der Bücher kümmern muss.

Für die Übermittlung der Angebotslisten an die Anwendung haben sich die Antiquariate auf ein

Datenformat in XML geeinigt. Das folgende Dokument wurde vom St. Jürgen Antiquariat am 20.2.2002 an die Anwendung übermittelt:

```
1  <aoml date="20.2.2002">
2      <antiquary>
3          <name>St. Jürgen Antiquariat</name>
4          <address>Ratzeburger Allee 40, 23562 Lübeck</address>
5          <email>st.juergenantiquariat@t-online.de</email>
6      </antiquary>
7      <offer>
8          <book catalog="Varia">
9              <title>Lotte in Weimar</title>
10             <author>Thomas Mann</author>
11             <condition>Einband fingerfleckig, Rücken verblaßt
                   </condition>
12             <price currency="EUR">8.00</price>
13         </book>
14         <book catalog="Varia">
15             <title>Buddenbrooks</title>
16             <author>Thomas Mann</author>
17             <condition>Einband verblichen, Besitzervermerk auf
                   Vs.</condition>
18             <price currency="EUR">25.00</price>
19         </book>
20     </offer>
21 </aoml>
```

Listing 2.1: Dokument in XML

Die gesamten Daten werden von dem Element `aoml` beinhaltet, für das das Attribut `datum` auf den Wert `20.2.2002` gesetzt wurde. Zunächst werden die Daten für das übermittelnde Antiquariat aufgeführt, die das Element `antiquary` umfasst. Neben dem Namen, im Element `name`, und der Adresse des Antiquariats, im Element `address`, wird dessen Emailadresse, im Element `email`, mit übertragen. Im Element `offer` erfolgt anschließend die Auflistung der vom Antiquariat angebotenen Artikel.

In diesem Fall werden zwei Bücher, erkennbar an den Elementtypen `book`, in das zentrale Verzeichnis eingestellt, die durch Titel, Autor, einer Zustandsangabe und dem Preis mit den Elementen `title`, `author`, `condition` und `price` beschrieben sind. Für das Element `book` kann man durch die Angabe des Attributs `catalog` bestimmen, unter welchen Rubriken das Buch im zentralen Verzeichnis auftreten soll. Im Element `price` wird mit dem Attribut `currency` angegeben, auf welche Währung sich die Preisangabe des Elements bezieht. □

2.1.1 Dokumenttypen für Auszeichnungssprachen

Wie im vorigen Abschnitt dargestellt, ist es mit XML möglich, Dokumente oder Daten durch Textauszeichnungen zu strukturieren. In vielen Anwendungen ist es allerdings sinnvoll, Dokumente mit gleichartiger Struktur zu einer Klasse von Dokumenten zusammenzufassen, um diese auf ähnliche Art und Weise zu verarbeiten. Eine Klasse gleichartiger Dokumente wird als *Auszeichnungssprache* („markup language“) bezeichnet und in XML durch eine *Dokumenttyp-Definition* (DTD) spezifiziert. Eine DTD abstrahiert dabei von den konkreten Dokumenten einer Auszeichnungssprache auf deren Struktur, ähnlich wie in der Theorie der Formalen Sprachen eine Sprache von Wörtern durch ihre Grammatik beschrieben wird.

Wie bereits beschrieben bilden Elemente und Elementtypen das wesentliche Strukturierungsmittel in XML. In der DTD besteht nun die Möglichkeit, Elementtypen durch Angabe einer Deklaration genauer zu spezifizieren, und damit den Inhalt dieser Elemente festzulegen. Eine *Elementtyp-Deklaration* besteht dabei aus der Zuordnung von einem regulären Ausdruck [Sal73], dem sogenannten *Inhaltsmodell* („content model“), zu einem Elementnamen. Der reguläre Ausdruck wird mittels Operatoren über den Elementnamen der DTD gebildet und kann sogar rekursiv sein. Unterstützt werden die beiden zweistelligen Operationen *reguläre Konkatenation* („sequence“) (Operator: `,`) und *reguläre Vereinigung* („choice“) (Operator: `|`) sowie die einstelligen Operationen *Kleene-Stern* (Operatoren: `*`, `+`)[1] und *Optional* (Operator: `?`). Um Zeichendaten im Inhalt eines Elements zu erlauben, ist der atomare Basisdatentyp *beliebige Zeichenkette* (`#PCDATA`) vorgesehen. Der reguläre Ausdruck darf aber auch *leer* (`EMPTY`) sein oder *beliebige Elementtypen* (`ANY`) zulassen.

Analog zur Elementtyp-Deklaration werden in der DTD Attributtypen durch eine *Attributtyp-Deklaration* spezifiziert. Da Attribute in einem XML-Dokument die Eigenschaften von Elementen beschreiben, ist jeder Attributtyp einem Elementtyp eindeutig zugeordnet. Da, wie bereits erwähnt, der Wert eines Attributs aus keinen Elementen bestehen darf, stehen für Attribute nur sehr einfache Typen zur Auswahl. Erlaubt sind Zeichendaten (`CDATA`) und definierbare Aufzählungstypen. Mit der Typangabe `ID` wird spezifiziert, dass es sich bei diesem Attribut, um ein *Schlüsselattribut* handelt. Die Werte dieser Attribute müssen Bezeichner sein, die für ein bestimmtes Dokument eindeutig sind. Attribute vom Typ `IDREF` und `IDREFS` sind *Schlüsselreferenzen* und verweisen auf einen oder mehrere dieser Schlüsselwerte. Für jeden Attributtypen muss angegeben werden, in welcher Form es in einem Element aufzutreten hat. Es kann als optional (`#IMPLIED`), verpflichtend (`#REQUIRED`) oder unveränderlich (`#FIXED`) deklariert werden. Weiterhin können Standardwerte für Attribute angegeben werden.

In einer DTD existiert mit den *Parameter-Entities* eine Möglichkeit häufig auftretende, längere reguläre Ausdrücke abkürzend zu bezeichnen. Innerhalb der DTD wird dann dieser Bezeichner anstatt des längeren Ausdrucks verwendet.

Definition 2.2 (Dokumenttyp-Definition)
Eine *Dokumenttyp-Definition* entspricht der folgenden Grammatik:

[1] Der Operator `*` steht für eine Liste, die auch leer sein darf, `+` für eine Liste mit mindestens einem Element.

<*DoctypeDecl*> → "<!DOCTYPE" <*Name*> "[" <*MarkupDecl*> "]" ">"
<*MarkupDecl*> → <*ElementDecl*> | <*AttListDecl*> | <*EntityDecl*>

<*ElementDecl*> → "<!ELEMENT" <*Name*> <*ContentSpec*> ">"
<*ContentSpec*> → "EMPTY" | "ANY" | <*Mixed*> | <*Children*>
<*Mixed*> → "(" "#PCDATA" ("|" <*Name*>)* ")*" | "(" "#PCDATA" ")"
<*Children*> → (<*Choice*> | <*Seq*>) ("?" | "*" | "+")?
<*Cp*> → (<*Name*> | <*PEReference*> | <*Choice*> | <*Seq*>) ("?" | "*" | "+")?
<*Choice*> → "(" <*Cp*> ("|" <*Cp*>)* ")"
<*Seq*> → "(" <*Cp*> ("," <*Cp*>)* ")"

<*AttrListDecl*> → "<!ATTLIST" <*Name*> <*AttDef*>* ">"
<*AttDef*> → <*Name*> <*AttType*> <*DefaultDecl*>
<*AttType*> → "CDATA" | "ID" | "IDREF" | "IDREFS" | <*Enumeration*>
<*Enumeration*> → "(" <*Name*> ("|" <*Name*>)* ")"
<*DefaultDecl*> → "#REQUIRED" | "#IMPLIED" | "#FIXED" <*AttValue*>

<*EntityDecl*> → "<!ENTITY" "%" <*Name*> <*EntityValue*> ">"
<*EntityValue*> → ("""" (<*CharData*> | <*PEReference*>)* """")
| ("'" (<*CharData*> | <*PEReference*>)* "'")
<*PEReference*> → "%" <*Name*> ";"

Das Nichtterminalsymbol <*Name*> steht hier für einen Elementnamen. Mit <*AttValue*> werden erneut Zeichenketten in einfachen (') oder doppelten Hochkommata (") und mit <*CharData*> alphanumerische Zeichendaten bezeichnet. □

Mit einer definierten Auszeichnungssprache, spezifiziert durch eine DTD, kann für ein gegebenes XML-Dokument getestet werden, ob es ein Dokument dieser Auszeichnungssprache ist. Werden die Anforderungen der DTD von einem Dokument erfüllt, spricht man von einem *gültigen* („valid“) Dokument. Anders als bei der Wohldefiniertheit ist für die Überprüfung der Gültigkeit eines Dokuments eine vorgegebene DTD notwendig. Damit ein Dokument als gültig erkannt wird, muss es sämtliche Anforderungen der DTD erfüllen. Dazu gehört, dass im Dokument nur Elemente auftreten, die auch in der DTD deklariert wurden, und diese korrekt verschachtelt sind. Dies bedeutet, dass die Inhalte der konkreten Elemente eines Dokuments den Inhaltsmodellen der Elementtypen entsprechen. Weiterhin darf ein Element nur Attribute enthalten, die auch für diesen Elementtypen deklariert wurden. Die Werte der Attribute müssen zu den deklarierten Wertebereichen der Attributtypen passen. Die Reihenfolge der Attribute eines Elements ist beliebig. Außerdem wird noch die Eindeutigkeit der Schlüsselattribute gefordert, sowie die Existenz der Schlüsselwerte, falls sie referenziert werden.

Eine ganze Reihe dieser Anforderungen lassen sich für XML-Dokumente, die von einem Programm dynamisch erzeugt werden, statisch, zum Zeitpunkt der Programmübersetzung, überprüfen. Ausgenommen sind die Eindeutigkeit der Schlüsselattribute sowie die Existenz der Schlüsselwerte. Auch ist es möglich, dass ein Programm versucht, aus einer bereits leeren Elementliste ein weiteres Element zu entfernen, was auch erst während des Programmablaufs festgestellt werden kann. Im weiteren Verlauf dieser Arbeit wird diese statisch überprüfbare Eigenschaft mit

statischer Gültigkeit bezeichnet.

Mit einer DTD lässt sich nun die Auszeichnungssprache des Dokuments aus dem letzten Beispiel spezifizieren.

Beispiel 2.2
Die *Antiquary-Offer-Markup-Language* (AOML) wird durch folgende DTD festgelegt:

```
1  <!DOCTYPE aoml [
2    <!ENTITY % fields  "article, condition, price" >
3
4    <!ELEMENT aoml       (antiquary, offer) >
5    <!ELEMENT antiquary  (name, address, email) >
6    <!ELEMENT name       (#PCDATA) >
7    <!ELEMENT address    (#PCDATA) >
8    <!ELEMENT email      (#PCDATA) >
9    <!ELEMENT offer      (book | record)* >
10   <!ELEMENT book       (title, author?, %fields;) >
11   <!ELEMENT record     (title, artist, %fields;) >
12   <!ELEMENT title      (#PCDATA) >
13   <!ELEMENT author     (#PCDATA) >
14   <!ELEMENT article    (#PCDATA) >
15   <!ELEMENT condition  (#PCDATA) >
16   <!ELEMENT artist     (#PCDATA) >
17   <!ELEMENT price      (#PCDATA) >
18
19   <!ATTLIST aoml   date      CDATA   #IMPLIED >
20   <!ATTLIST book   catalog   CDATA   #IMPLIED >
21   <!ATTLIST price  currency  CDATA   #REQUIRED >
22 ]>
```

Listing 2.2: Dokumenttyp-Definition der AOML

Die DTD deklariert das Entity `field` und die Elementtypen `aoml`, `antiquary`, `name`, `address`, `email`, `offer`, `book`, `record`, `title`, `author`, `article`, `condition`, `artist` und `price`. So muss beispielsweise ein Element vom Typ `antiquary` im Inhalt die Elemente `name`, `address` und `email` in dieser Reihenfolge umfassen. Darüber hinaus werden für den Elementtyp `aoml` ein Attribut `date`, für den Elementtyp `book` ein Attribut `catalog` und für den Elementtyp `price` das Attribut `currency` vereinbart. □

Das Beispiel, welches in dieser Arbeit durchgängig betrachtet wird, enthält alle Voraussetzungen, die zur Vorstellung der Probleme und Lösungswege, die hier verfolgt werden, notwendig sind. Es beinhaltet eine Anwendung, die dynamisch zur Laufzeit XML-Dokumente erzeugt. Die generierten Dokumente müssen dabei einer vorgegebenen DTD genügen. Der Sprachumfang der spezifizierten Auszeichnungssprache ist zwar stark eingeschränkt, doch würde eine Erweiterung

des Beispiels auf mehr Elemente und Attribute konzeptionell zu keinen weiteren Erkenntnissen führen. Vielmehr wäre eine Verminderung in der Klarheit der Darstellung zu erwarten.

Ein wesentlicher Kritikpunkt an DTDs ist, dass nur zwei atomare Basisdatentypen, nämlich `#PCDATA` als Elementinhalt und `CDATA` als Attributwert, vorgesehen sind. Dies mag für den Bereich der Dokumentenverarbeitung ausreichend sein, für den Datenaustausch im World-Wide Web ist es nicht. Im allgemeinen Datenaustausch ist es beispielsweise häufig sinnvoll, für den Elementtyp einer Auszeichnungssprache zu spezifizieren, dass das Inhaltmodell vom atomaren Basisdatentyp `integer` ist. Weitere Einschränkungen von DTDs entstehen durch die globale Spezifikation von Elementtypen. Lokale Deklarationen eines Elementnamens mit unterschiedlichen Inhaltsmodellen ist deshalb ausgeschlossen. Eine Beschränkung von Schlüsselreferenzen auf die Schlüssel bestimmter Elementtypen ist ebenfalls nicht möglich.

2.1.2 XML-Schema

XML-Schema liegt seit Mai 2001 als Empfehlung des W3C [W3C01c, W3C01d] vor. Es dient – wie die DTDs– zur Spezifikation von Auszeichnungssprachen, bietet aber genauere Möglichkeit zur Definition von Elementtypen, Attributtypen und weiteren Nebenbedingungen. Die Notation einer Sprachbeschreibung erfolgt mit XML-Schema in der Auszeichnungssprache XML-*Schema-Definition-Language* (XSDL), die selbst ein eigener XML-Dialekt ist. Die folgende Darstellung beschränkt sich auf die für diese Arbeit wichtigen Besonderheiten; ausführliche Beschreibungen finden sich in [W3C01b, vdV02].

In XML-Schema wird unterschieden zwischen Deklarationen, die Komponenten definieren, die in den Dokumenten der Auszeichnungssprache auftreten können, und Definitionen, die Komponenten spezifizieren, die nur schemaintern Verwendung finden. Im Gegensatz zu DTDs können Elementtypen nun global, geltend in der gesamten Sprachbeschreibung, oder lokal, nur im aktuellen Inhaltsmodell gültig, deklariert werden, wodurch unterschiedliche Elementtypen mit verschiedenen Inhaltsmodellen bei gleichem Elementnamen möglich werden. Die Deklaration eines Elementnamens geschieht in XSDL durch das Element `element` mit dem Attribut `name`. Für den Inhalt von Elementen kann definiert werden, dass er leer ist, nur Text umfasst, nur Elemente enthält oder gemischten Inhalt hat. Für die Inhaltsmodelle der Elementtypen kann neben Konkatenation (Element `sequence`) und Vereinigung (Element `choice`) durch die Operation `all` abkürzend ausgedrückt werden, dass Elementtypen in beliebiger Reihenfolge auftreten müssen. Durch die Attribute `minOccurs` und `maxOccurs` können Nebenbedingungen, die die Häufigkeiten des Auftretens von Inhaltsmodellen festlegen, genauer spezifiziert werden. Wird das Attribut `maxOccurs` mit dem Wert `unbounded` versehen, so darf sich das entsprechende Inhaltsmodell im Dokument beliebig oft wiederholen. Dies ist vergleichbar mit dem Kleene-Stern-Operator in DTDs.

XML-Schema differenziert zwischen *einfachen* und *komplexen Typen*. Bei einfachen Typen (Element `simpleType`) handelt es sich entweder um eingebaute atomare Basisdatentypen, wie beispielsweise `integer` oder `string`, Aufzählungstypen (Element `enumeration`) oder um

Ableitungen atomarer Basisdatentypen, deren Wertebereiche eingeschränkt wurden. Auch können durch einfache Typen Listen- oder Vereinigungstypen („union type“) über einfache Typen definiert werden. Komplexe Typen (Element `complexType`) sind dagegen Typen für Elementtypen, die aus einem Inhaltsmodell und optionalen Attributdeklarationen bestehen. Sie können genauso wie einfache Typen durch einen eindeutigen Typnamen bestimmt sein oder als *anonyme Typen* direkt in einer Elementtyp-Deklaration auftreten. Das folgende Beispiel zeigt eine Elementtyp-Deklaration. Bei der Deklaration eines Elementnamens kann dann entweder ein anonymer Typ definiert oder auf einen Typnamen verwiesen werden (Attribute `type`).

Beispiel 2.3
In diesem Beispiel wird die Deklaration des Elementtyps `aoml` der DTD aus Beispiel 2.2 in XML-Schema formuliert; die vollständige Sprachbeschreibung findet sich in Anhang A:

```
2   <element name="aoml">
3     <complexType>
4       <sequence>
5         <element name="antiquary" type="t_antiquary"/>
6         <element name="offer" type="t_offer"/>
7       </sequence>
8       <attribute name="date" type="string"/>
9     </complexType>
10  </element>
```

Listing 2.3: Schemadefinition AOML

Es zeigt die Spezifikation von `aoml` durch einen anonymen komplexen Typen als globalen Elementtypen. Das Inhaltsmodell besteht wie in der DTD aus der Konkatenation eines `antiquary`- und eines `offer`-Elementtyps. Die Inhaltsmodelle und Attribute der lokal deklarierten Elementtypen `antiquary` und `offer` werden durch die komplexen Typen `t_antiquary` und `t_offer` definiert, deren Definitionen bringen konzeptionell nichts Neues, weshalb sie hier nicht weiter ausgeführt sind. Sie finden sich in Anhang A. Das Attribut `date` ist vom atomaren Basisdatentyp `string`. □

Ähnlich zur Definition von Parameter-Entities gibt es in XML-Schema die Möglichkeit Inhaltsmodelle mit einem Namen zu versehen. Mit diesen *benannten Gruppen* (Element `group`) ist es möglich die Definitionen von komplexen Typen abzukürzen, indem auf solche benannten Inhaltsmodelle referenziert wird.

Von einfachen und komplexen Typen können in XML-Schema, ähnlich wie in objektorientierten Programmiersprachen durch Vererbung, Ableitungen gebildet werden. Zu unterscheiden ist dabei zwischen einer *Einschränkung* (Element `restriction`) und einer *Erweiterung* (Element `extension`), die nur für komplexe Typen möglich ist. Bei einer Einschränkung müssen alle Instanzen des abgeleiteten Typs eine gültige Instanz des Basistyps sein, womit der abgeleitete Typ einen Subtyp des Basistyps bildet. Bei der Ableitung durch Erweiterung wird ein komplexer Basistyp um weitere Elementtypen oder Attributtypen ergänzt. In einer Dokumenteninstanz

kann dann anstelle des komplexen Basistyps der abgeleitete, erweiterte komplexe Typ auftreten, was mit *Typsubstitution* bezeichnet wird. Der aktuelle Typ des Elements muss dann allerdings durch ein spezielles Attribut (`type`) ausgezeichnet werden. Eine ähnliche Erweiterung wird durch *Substitutionsgruppen* („substitution group“) auf der Basis von Elementtypen eingeführt. XML-Schema erlaubt es, unterschiedliche Elementnamen mit gleichen Elementtypen zu einer Substitutionsgruppe zusammenzufassen. Im Dokument ist es dann zulässig, ein Element aus der Gruppe dort einzusetzen, wo ein anderer Elementtyp aus der Substitutionsgruppe erwartet wird.

Das folgende Beispiel zeigt die Sprachbeschreibung einer kleinen vollständigen Auszeichnungssprache mit XML-Schema.

Beispiel 2.4
Das *Shop-Interchange-Format* (SIF) ist ein Datenaustauschformat zur Kommunikation mit dem Warenkorb des zentralen Verzeichnis antiquarischer Bücher. Damit lassen sich Nachrichten formulieren, um den Warenkorb auszugeben, um Artikel hinzuzufügen oder um Artikel aus dem Warenkorb zu entfernen. Das Format ist wie folgt definiert:

```
1  <schema>
2     <element name="shopRequest" type="t_shopRequest"/>
3
4     <complexType name="t_shopRequest">
5        <sequence>
6           <element name="shoppingCart"
                              type="t_cartRequest"/>
7        </sequence>
8     </complexType>
9
10    <complexType name="t_cartRequest">
11       <sequence>
12          <element name="account" type="integer"/>
13          <choice>
14             <element name="add" type="integer"/>
15             <element name="remove" type="integer"/>
16             <element name="get"><complexType/></element>
17          </choice>
18       </sequence>
19    </complexType>
20
21
22    <element name="shopResponse" type="t_shopResponse"/>
23
24    <complexType name="t_shopResponse">
25       <sequence>
26          <element name="shoppingCart"
```

```
                                    type="t_cartResponse"/>
27          </sequence>
28      </complexType>
29
30      <complexType name="t_cartResponse">
31          <sequence>
32              <element name="account" type="integer"/>
33              <element name="request" type="t_request"/>
34              <element name="items" type="t_items"
                      minOccurs="0"/>
35          </sequence>
36      </complexType>
37
38      <complexType name="t_items">
39          <sequence>
40              <element name="article" type="integer"
                    minOccurs="0" maxOccurs="unbounded"/>
41              <element name="description" type="string"
                    minOccurs="0"/>
42          </sequence>
43      </complexType>
44
45      <simpleType name="t_request">
46          <restriction base="string">
47              <enumeration value="processed"/>
48              <enumeration value="fail"/>
49          </restriction>
50      </simpleType>
51  </schema>
```

Listing 2.4: Schemadefinition SIF

Die Sprachbeschreibung SIF deklariert die beiden globalen Elementtypen `shopRequest` und `shopResponse`. Darüber hinaus werden die komplexen Typen `t_shopRequest`, `t_cardRequest`, `t_shopResponse`, `t_cardResponse` und `t_items`, mit den lokalen Elementtypen `shoppingCart`, `account`, `add`, `remove`, `get`, `request`, `items`, `article` und `description` definiert. Außerdem wird der einfache Typ `t_request` durch Einschränkung des atomaren Typs `string` als Aufzählungstyp festgelegt. □

XML-Schema geht in einer ganzen Reihe von Punkten über das Dargestellte hinaus. Nicht weiter betrachtet werden in dieser Arbeit das Inhaltsmodell `any`, Attributgruppen, die Verhinderung von Typsubstitution (Attribute: `block`), das Erzwingen einer Ableitung durch abstrakte Typen und abstrakte Elementtypen, die Verhinderung von Ableitungen (Attribute: `final`) sowie Nebenbedingungen wie Eindeutigkeit, Schlüsselattribute und Referenzen auf diese. Für eine detai-

lierte Präsentation wird auf die für XML-Schema angeführte Literatur verwiesen.

2.2 XPath

XPath [W3C99a] ist eine vom W3C standardisierte Sprache zur Adressierung von Elementen und Teilen eines XML-Dokuments. Ursprünglich wurde es für den einheitlichen Gebrauch in der Transformationssprache der XML-Stylesheet-Language (XSLT) [W3C99b] und XML-Pointer-Language [W3C02b] entwickelt. Darüber hinaus werden von XPath Basisfunktionen zur Manipulation von Zeichenketten, Zahlen und booleschen Werten zur Verfügung gestellt.

In Definition 2.3 wird spezifiziert, was in dieser Arbeit unter einem XPath-Ausdruck verstanden werden soll. Dabei handelt es sich, bis auf wenige Ausnahmen, um den gesamten Sprachumfang von XPath in der Version 1.0. Nicht betrachtet werden abkürzende Notation, Verarbeitungsanweisungen, absolute Pfadangaben, Union- und Filter-Ausdrücke sowie die von XPath definierten Funktionen und Operationen auf Zeichenketten, Zahlen und booleschen Werten. Es sei darauf hingewiesen, dass die in Entwicklung befindliche Version 2.0 von XPath [W3C02a] in weiten Teilen umfangreicher ist. Grundsätzlich sind dann auch Bedingungen und Schleifenkonstrukte möglich. Die zusätzlichen Möglichkeiten der neuen Version bieten isoliert betrachtet zwar eine Erweiterung der Ausdrucksmöglichkeit, im Rahmen dieser Arbeit wird XPath aber stets als Ergänzung einer Programmiersprache behandelt, die bereits ähnliche Programmkonstrukte zur Verfügung stellt. Ein weiterer Vorteil der hier verwendeten Version 1.0 von XPath ist die Vorlage als festgelegter Standard. Umfassende Beschreibungen der beiden Versionen findet sich in den Spezifikationen des W3C [W3C99a, W3C02a].

Definition 2.3 (XPath-Ausdruck)
Ein *XPath-Ausdruck* ist nach folgender Grammatik aufgebaut:

<*LocationPath*>	→	<*RelativeLocationPath*>
<*RelativeLocationPath*>	→	<*Step*> | <*RelativeLocationPath*> "`/`" <*Step*>
<*Step*>	→	<*AxisSpecifier*> <*NodeTest*> <*Predicate*>*
<*AxisSpecifier*>	→	<*AxisName*> "`::`"
<*AxisName*>	→	"`ancestor`"|"`ancestor-or-self`"|"`attribute`"| "`child`"|"`descendant`"|"`descendant-or-self`"| "`following`"|"`following-sibling`"|"`parent`"| "`preceding`"|"`preceding-sibling`"|"`self`"
<*NodeTest*>	→	<*NameTest*> | <*NodeType*> "`(``)`"
<*Predicate*>	→	"`[`" <*PredicateExpr*> "`]`"
<*PredicateExpr*>	→	<*Expression*>
<*NameTest*>	→	"`*`" | <*Name*>
<*NodeType*>	→	"`comment`"|"`text`"|"`node`"

Mit <*Expression*> wird dabei ein boolescher Ausdruck bezeichnet, der hier nicht weiter ausgeführt wird. Er orientiert sich an Ausdrücken in gebräuchlichen Programmiersprachen. Das Nichtterminalsymbol <*Name*> steht für einen Elementnamen. □

Im Folgenden werden die für diese Arbeit relevanten Konstrukte näher erläutert. Jeder XPath-Ausdruck bezieht sich auf einen aktuellen *Kontextknoten* („context node“), bei dem es sich um einen beliebigen Knoten innerhalb des XML-Dokuments handeln kann. Dieser Knoten ist nötig, um festzulegen an welcher Position im Dokument die Auswertung des Ausdrucks beginnt. Während der Berechnung der Ergebnismenge zu einem Ausdruck kann sich, zum Ermitteln von Teilergebnissen, der Kontextknoten zeitweilig verändern.

Ein Pfadausdruck in XPath selektiert aus einem XML-Dokument einen einzelnen Knoten oder eine Menge von Knoten.[2] Er besteht aus beliebig vielen *Lokalisierungsschritten* („location step“), die durch das Zeichen / von einander getrennt werden. Vereinfacht dargestellt, bestehen sie aus folgender Struktur:

$$\langle Step \rangle_1 / \ldots / \langle Step \rangle_n$$

Die Semantik der Auswertung dieser Liste von Lokalisierungsschritten, die eine *Knotenmenge* („node set“) als Ergebnis liefert, lässt sich durch folgenden Algorithmus in Pseudocode-Notation beschreiben.

```
NodeSet process(Node context, List locationSteps) {
  NodeSet s1 = apply(locationSteps.first(),context);
  if (locationSteps.tail().isEmpty())
    return s1;
  else {
    NodeSet s2 = ∅;
    foreach (n ∈ s1)
      s2 = s2 ∪ process(n,locationSteps.tail());
    return s2;
  } // else
} // process
```

Listing 2.5: Algorithmus zur Auswertung der Lokalisierungsschritte

Besteht die Liste nur aus einem Lokalisierungsschritt, ist das Ergebnis der Auswertung dieses Schritts das Gesamtergebnis. Für eine Liste mit mehr als einem Lokalisierungsschritt wird zunächst der erste Schritt ausgewertet. Anschließend wird jeder Knoten in diesem Zwischenergebnis als Kontextknoten mit der Liste ohne den ersten Lokalisierungsschritt weiterverarbeitet. Die Teilresultate dieser rekursiven Aufrufe werden anschließend zum Gesamtergebnis vereinigt.

Jeder Lokalisierungsschritt besteht aus den drei Komponenten *Achse* („axis“), *Knotentest* („node test“) und beliebig vielen *Prädikaten* („predicate“). Dies führt, vereinfacht dargestellt, zu einer Struktur, wie folgt:

$$\langle Axis \rangle :: \langle NodeTest \rangle [\langle Predicate \rangle_1] \ldots [\langle Predicate \rangle_n]$$

In XPath existieren die folgenden Achsen, die in zwei Gruppen unterteilt werden: Zum einen gibt es Achsen, die Knoten in *Dokumentordnung* („document order“) selektieren, und zum anderen

[2] In XPath wird von einer Knotenmenge gesprochen, obwohl eine Ordnung für diese Menge existiert.

Achsen, die als Ergebnismenge eine Knotenmenge in *umgekehrter Dokumentordnung* („revers document order“) liefern. Mit Dokumentordnung wird dabei die Reihenfolge des Auftretens der ersten Zeichen von Elementen, Attributen, Text und Kommentaren im Dokument bezeichnet. Da Elemente vor ihrem Inhalt liegen, sind Elemente anhand des Auftretens ihrer Start-Tags in XML angeordnet. Die Attribute eines Elements liegen vor den Subelementen des Inhalts des Elements. Die umgekehrte Dokumentordnung ist definiert als die Umkehrung der Dokumentordnung.

Dies ist deshalb von Bedeutung, weil sich anschließende Prädikate auf die Positionen in der Liste beziehen können.

- Die Achsen mit Knoten in Dokumentordnung lauten:
  - Selbst-Achse („self axis“): Selektion des aktuellen Kontextknotens.
  - Kind-Achse („child axis“): Liefert die unmittelbaren Kinderknoten des Kontextknotens.
  - Nachfahr-Achse („descendant axis“): Gibt die Kinder sowie rekursiv alle Kindeskinder zurück.
  - Nachfahr-oder-Selbst-Achse („descendant-or-self axis“): Bezeichnet alle Kindeskinder des Kontextknotens inklusive dem aktuellen Kontextknoten.
  - Nachfolgende-Geschwister-Achse („following sibling axis“): Wählt alle folgenden Geschwisterknoten des Kontextknotens aus.
  - Nachfolger-Achse („following axis“): Selektiert alle nachfolgenden Knoten des aktuellen Kontextknotens.
  - Attribut-Achse („attribute axis“): Gibt alle Attribute des Kontextknotens zurück.
- Die Achsen mit Knoten in umgekehrter Dokumentordnung sind:
  - Eltern-Achse („parent axis“): Liefert den unmittelbaren Elternknoten des Kontextknotens.
  - Vorfahr-Achse („ancestor axis“): Gibt den Elternknoten sowie rekursiv alle weiteren Vorfahrenknoten zurück.
  - Vorfahr-oder-Selbst-Achse („ancestor-or-self axis“): Bezeichnet alle Vorfahrenknoten des Kontextknotens inklusive dem aktuellen Kontextknoten.
  - Vorherige-Geschwister-Achse („preceding sibling axis“): Wählt alle Geschwisterknoten des Kontextknotens aus, die vor dem Kontextknoten im Dokument stehen.
  - Vorgänger-Achse („preceding axis“): Selektiert alle vor dem aktuellen Kontextknoten auftretenden Knoten.

Beim anschließenden Knotentest wird die durch die Achse ausgewählte Liste von Knoten eingeschränkt. Grundsätzlich stehen folgende Möglichkeiten zur Verfügung:

- Durch die Angabe eines Elementnamen werden nur die Elemente aus der selektierten Knotenliste ausgewählt, die von diesem Elementtyp sind.
- Alternativ können mit der Angabe eines Knotentyps alle Knoten dieses Typs entlang der bezeichneten Achse selektiert werden. Es stehen dafür folgende Knotentypen zur Auswahl:
  - `node()`: Wählt alle Knoten aus, und bezeichnet damit den Grundtyp aller Knotentypen.
  - `text()`: Ausschließlich Textknoten werden selektiert.
  - `comment()`: Bezeichnet die Kommentarknoten im Dokument.

Durch die Angabe von einem oder mehreren Prädikaten kann die bereits reduzierte Knotenliste weiter vermindert werden. Ein Prädikat ist ein beliebiger boolescher Ausdruck, der für jeden Knoten in der Knotenliste ausgewertet wird. Erfüllt ein Knoten das angegebene Prädikat, wird er in die Ergebnisliste übernommen. XPath führt neben den üblichen Relationen und Funktionen auf Zeichenketten, Zahlen und booleschen Werten folgende, zusätzliche elementare Operationen ein:

- `position()`: Liefert die Position des Kontextknotens in der gegenwärtigen Knotenliste zurück.
- `last()`: Die Operation ermittelt die letzte Position für die selektierte Knotenliste. Dies entspricht damit der Länge der aktuellen Knotenliste.

Die Auswertung eines aus diesen drei Teilen bestehenden Lokalisierungsschritts erfolgt derart, dass zunächst alle Knoten gemäß der angegebenen Achse bezüglich des aktuellen Kontextknotens selektiert werden. Danach wird die Knotenmenge auf die Knoten eingeschränkt, die zunächst den Knotentest bestehen und im Anschluss daran nacheinander jedes Prädikat erfüllen. Die folgenden Beispiele verdeutlichen dieses Vorgehen.

Beispiel 2.5
Um die Anwendung von XPath zu demonstrieren, werden einige Beispiele zur Auszeichnungssprache AOML (Beispiel 2.2) angegeben, die sich beispielsweise auf das Dokument aus Beispiel 2.1 beziehen können.

1. Der folgende Ausdruck liefert sämtliche Kinderknoten mit dem Elementnamen `author` vom aktuellen Kontextknoten:

   ```
   child::author
   ```

   Bei Anwendung des Ausdrucks auf das Dokument aus Beispiel 2.1 mit dem `book`-Element aus Zeile 8 als Kontextknoten ergibt sich folgendes Ergebnis:

   ```
   <author>Thomas Mann</author>
   ```

2. Mit dieser Angabe werden die Kinderknoten des Kontextknotens selektiert, die den Elementtyp `book` haben und hinter dem fünften `book`-Element stehen:

   ```
   child::book[position() > 5]
   ```

   Sei der Elementknoten `offer` aus Zeile 7 der Kontextknoten für die Auswertung dieses Ausdrucks, so ergibt sich eine leere Resultatsmenge, da nur zwei Bücher im Angebot dieses Händlers enthalten sind.

3. Die Selektion der Nachfahren vom Elementtyp `price`, die ein Attribut mit dem Namen `currency` und dem Wert `EUR` besitzen, erfolgt mit diesem Pfadausdruck:

   ```
   descendant::price[string(attribute::currency) ==
         "EUR"]
   ```

   Für die beispielhafte Auswertung sei der Kontextknoten das Element `aoml` aus Zeile 1. Dann ergibt sich folgende Ergebnismenge:

   ```
   <price currency="EUR">8.00</price>
   <price currency="EUR">25.00</price>
   ```

□

Beispiel 2.6
Dieses Beispiel zeigt eine Anwendung für die Auszeichnungssprache SIF (Beispiel 2.4).

```
child::shopRequest/child::shoppingCart[position() == 1]
```

Der Ausdruck liefert das erste `shoppingCart`-Element, das das Kind eines `shopRequest`-Elements ist, welches wiederum ein Kind des gegenwärtigen Kontextknotens sein muss. □

Abschließend ist anzumerken, dass XPath-Ausdrücke unabhängig von einer Sprachbeschreibung formuliert werden und sich ausschließlich an der Dokumentenstruktur orientieren.

2.3 Dokument-Objektmodell

Die Daten eines XML-Dokuments liegen durch ihre einfache Form zunächst als reine Textdaten vor. Die Verarbeitung des Inhalts eines XML-Dokuments erfolgt in der Regel durch ein Programm, weshalb ein universeller Zugriff auf die Daten erforderlich wird. Es liegt nahe, dafür die logische Sicht auf das Dokument, die implizite Baumstruktur der geschachtelten Elemente, heranzuziehen.

Das *Dokument-Objektmodell* (DOM) spezifiziert die logische Struktur eines Dokuments, um aus einer Anwendung heraus über diese auf das Dokument zuzugreifen oder es zu manipulieren. Das DOM ermöglicht dem Programmierer das Erzeugen von wohlgeformten Dokumenten, die Navigation in deren Struktur, das Hinzufügen, Verändern oder Löschen von Elementen und Inhalt.

Alles was ein XML-Dokument enthält, kann durch das Dokument-Objektmodell angesprochen, verändert, gelöscht oder hinzugefügt werden.

Das Dokument-Objektmodell wurde vom W3C als Empfehlung [W3C98b, W3C00a] verabschiedet. In der Spezifikation werden sprach- und plattformneutrale Schnittstellen definiert. Zusätzlich werden vom DOM für einige Programmiersprachen sogenannte Sprachbindungen zur Verfügung gestellt. Eine Sprachbindung gibt an, wie die DOM-Schnittstellen für eine konkrete Programmiersprache umgesetzt werden.

In diesem Abschnitt werden die grundlegenden Schnittstellen der Spezifikation vorgestellt. Es wird eine formale Semantik angegeben, da [W3C98b] diese nur informell beschreibt. Nach einer kurzen Darstellung der Spezifikationsmethode, werden die Schnittstellen definiert. Es folgen Beispiele, die Beschreibung von Erweiterungen und Implementierungen sowie eine kritische Einschätzung des DOM.

2.3.1 Formalisierung

In diesem Abschnitt wird eine Möglichkeit zur Spezifikation von objektorientierten Schnittstellen vorgestellt. Dabei wird die Idee der *abstrakten Datentypen* [LZ74, WPP+83, LEW96] aufgegriffen und auf objektorientierte Schnittstellen übertragen. Die Einführung in dieser Arbeit erfolgt nur informell. Die formalen Grundlagen, wie objektorientierte Algebra, Belegungs- und Ausführungsfunktion, werden hier nicht erläutert. Detaillierte Darstellungen dazu finden sich in [LV96, Hug99], objektorientierte Typsysteme werden in [Ala97, Ala99] behandelt.

In abstrakten Datentypen wird die Semantik der Operationen durch Gleichungen spezifiziert. Auf ähnliche Weise soll hier die Semantik der objektorientierten Methoden angegeben werden. Dafür ist das Konzept einer Anweisungsgleichung notwendig.

Definition 2.4 (Anweisungsgleichung)
Eine *Anweisungsgleichung* besteht aus den zwei Anweisungen s_l und s_r und wird notiert durch:

$$[s_l] \Leftrightarrow [s_r]$$

□

Eine Anweisungsgleichung ist *gültig* innerhalb einer objektorientierten Algebra, falls für jeden gültigen Anfangszustand nach Auswertung von s_l die gleichen Variablenbelegungen und die gleichen Zustände für die beteiligten Objekte erreicht werden, wie nach der Auswertung von s_r. Für die beteiligten Objektreferenzen gilt dabei, dass sie bis auf Umbenennung gleich sind.

Als abkürzende Schreibweise wird im Weiteren auch folgendes verwendet:

$$l \overset{[s]}{=} r \quad \text{ist äquivalent zu} \quad [s; v := l] \Leftrightarrow [s; v := r]$$

Dies ist so zu interpretieren, dass nach der Auswertung von s die aufgeführte Gleichung $l = r$

gilt, deren Ausdrücke l und r nur Zugriffsoperationen beinhalten und dadurch die Zustände der Objekte nicht verändern.

Die Semantik einer objektorientierten Schnittstelle wird nun im Weiteren durch eine Menge von Anweisungsgleichungen spezifiziert. Eine objektorientierte Algebra ist ein Modell einer objektorientierten Schnittstelle, falls sämtliche Anweisungsgleichungen der Schnittstelle gültig sind.

Das folgende Beispiel demonstriert den verwendeten Formalismus.

Beispiel 2.7
Gegeben sei die Schnittstelle `Stack`, die die Methoden eines Kellers über ganzen Zahlen definiert, in typischer objektorientierter Form:

```
1 interface Stack {
2     static Stack newStack();
3     void push(int i);
4     void pop();
5     int top();
6     boolean isEmpty();
7 } // Stack
```

Ein zunächst leerer Keller wird mit der Operation `newStack` erzeugt. Die Methode `push` erlaubt das Ablegen einer ganzen Zahl auf dem Keller. Die oberste Zahl kann mit der Methode `pop` wieder entfernt werden. Ausgelesen werden kann sie mit der Methode `top`. Um zu überprüfen, ob der Keller leer ist, steht die Methode `isEmpty` zur Verfügung.

Die Spezifikation mittels der oben eingeführten Anweisungsgleichungen kann nun wie folgt vorgenommen werden; wobei ϵ für die leere Anweisung steht:

$$\begin{gathered}
\left[\ s := \texttt{newStack}(); b := s.\texttt{isEmpty}()\ \right] \Leftrightarrow \left[\ s := \texttt{newStack}(); b := \textit{true}\ \right] \\
\left[\ s.\texttt{push}(i); b := s.\texttt{isEmpty}()\ \right] \Leftrightarrow \left[\ s.\texttt{push}(i); b := \textit{false}\ \right] \\
\left[\ s.\texttt{push}(i); i := s.\texttt{top}()\ \right] \Leftrightarrow \left[\ s.\texttt{push}(i); i := i\ \right] \\
\left[\ s.\texttt{push}(i); s.\texttt{pop}()\ \right] \Leftrightarrow \left[\ \epsilon\ \right]
\end{gathered}$$

Für die Variablen gilt s : `Stack`, i : `int` und b : `boolean`. Unter Verwendung der abkürzenden Schreibweise können die ersten drei Gleichungen umgeschrieben werden zu:

$$\begin{gathered}
s.\texttt{isEmpty}() \overset{[s:=\texttt{newStack}()]}{=} \textit{true} \\
s.\texttt{isEmpty}() \overset{[s.\texttt{push}(i)]}{=} \textit{false} \\
s.\texttt{top}() \overset{[s.\texttt{push}(i)]}{=} i
\end{gathered}$$

Die erste Gleichung definiert, dass nach der Erzeugung eines Kellers dieser zunächst leer ist. Nach dem Ablegen eines Elements ist ein Keller nicht mehr leer, was die zweite Gleichung angibt. In der Dritten wird spezifiziert, dass nach dem Ablegen eines Elements auf dem Keller

dieses Element das oberste Element ist. Die Anwendung der beiden Methoden `push` und `pop` nacheinander führt wieder zum ursprünglichen Zustand; dies legt die letzte Gleichung fest. □

2.3.2 Schnittstellen und deren Semantik

Die Beschreibung der Schnittstellen des DOM in diesem Abschnitt beschränkt sich auf die Schnittstellen, die für die Konzepte Dokument, Element, Attribut und Kommentar notwendig sind. Weiterhin wird hier nur die konsequent objektorientierte Umsetzung dargestellt. Auf den vereinfachenden Ansatz, bei dem jedes Objekt im DOM als Knoten verstanden werden kann, wird hier aus Gründen der Klarheit verzichtet. Für eine weitergehende Spezifikation des DOM sei auf die Empfehlungen des W3C verwiesen. Die Definition der Syntax des DOM erfolgt in der sprach- und plattformneutralen Interface-Definition-Language (IDL) der OMG [Obj02]. Zur Beschreibung der Semantik wird hier der oben eingeführte formale Ansatz gewählt, während die genannten Referenzen eine informelle Beschreibung angeben.

Schnittstelle für Dokumente

Ziel des DOM ist es, ein Modell für XML-Dokumente bereitzustellen, weshalb es nötig ist, zunächst eine Schnittstelle für Dokumente festzulegen. In Listing 2.6 ist die Schnittstelle dargestellt. Jedes Dokument besteht, wie in Abschnitt 2.1 erwähnt, aus einem Wurzelelement. Dieses

```
1 interface Document {
2     attribute Element documentElement;
3     Element createElement(in DOMString tagName);
4     Text createTextNode(in DOMString data);
5     Comment createComment(in DOMString data);
6     Attr createAttribute(in DOMString name);
7 }
```

Listing 2.6: DOM-Schnittstelle `Document`

Wurzelelement wird mit dem Attribut `documentElement` angesprochen. Unter Vorwegnahme der Attribute `parentNode`, das auf den Elternknoten innerhalb der baumartigen Repräsentation eines Dokuments verweist, sowie `nextSibling` und `previousSibling`, die auf Vorgänger und Nachfolger zeigen, aus der Schnittstelle `Node` ist eine Formalisierung möglich:

- Das Wurzelelement verweist mittels `parentNode` auf das Dokumentobjekt.

$$d.\texttt{documentElement}.\texttt{parentNode} = d$$

- Das Wurzelelement hat keinen Nachfolger.[3]

$$d.\texttt{documentElement.nextSibling} = \texttt{nil}$$

- Das Wurzelelement hat keinen Vorgänger.

$$d.\texttt{documentElement.previousSibling} = \texttt{nil}$$

Weiterhin sind in dieser Schnittstelle die Konstruktoren für die Schnittstellen `Element`, `Text` und `Attribut` untergebracht. Da diese Teile eines Dokuments nach Auffassung des DOM nur innerhalb eines Dokuments auftreten dürfen, fungiert die Schnittstelle als Konstruktor-Klasse („abstract factory“), ein aus dem objektorientierten Design [GHJV95] bekanntes Muster. Der Konstruktor für die Dokumente selbst wird durch eine Implementierung des DOM definiert und ist hier nicht dargestellt.

Schnittstelle für Attribute

Für die Elemente in XML-Dokumenten besteht die Möglichkeit, über Attribute Eigenschaften festzulegen. Im DOM werden diese über die Schnittstelle `Attr` realisiert, die in Listing 2.7 zu sehen ist. Jedes XML-Attribut ist über das Attribut `ownerDocument` einem Dokument zuge-

```
1 interface Attr {
2     readonly attribute Document ownerDocument;
3     readonly attribute DOMString name;
4     attribute DOMString value;
5 }
```

Listing 2.7: DOM-Schnittstelle `Attr`

ordnet, besitzt einen unveränderlichen Namen `name` und verfügt über einen Wert `value`. Die Attribute sind wie folgt zu formalisieren:

- Nach der Konstruktion eines Attributes `a` mit dem Namen `n`, verweist `ownerDocument` auf das erzeugende Dokument, und `name` auf den Namen `n`.

$$\begin{gathered}\left[\ a := d.\texttt{createAttribute}(n)\ \right] \\ d = a.\texttt{ownerDocument} \\ n = a.\texttt{name}\end{gathered}$$

- Wird der Wert `value` eines Attributes `a` auf den Wert `s` gesetzt, so hat der Wert des Attributes anschließend diesen Wert `s`.

$$\begin{gathered}\left[\ a.\texttt{value} := s\ \right] \\ s = a.\texttt{value}\end{gathered}$$

[3] Mit `nil` wird der Wert einer nicht belegten Objektreferenz bezeichnet.

Schnittstellen für Knoten, Elemente, Text und Kommentar

Die Komponenten Element und Text, die, wie in Abschnitt 2.1 beschrieben, innerhalb eines Dokuments beliebig tief geschachtelt werden können, sind im DOM als Komponente-Kompositum-Struktur („composite component“) realisiert, ein Modellierung, die ebenfalls aus dem objektorientierten Design [GHJV95] stammt.

Mit `Node` in Listing 2.8 wird die Komponenten-Schnittstelle der verschachtelten Struktur des DOM definiert. Sie deklariert die Attribute und Methoden der Knoten, die in der baumartigen

```
1  interface Node {
2      const unsigned short ELEMENT_NODE = 1;
3      const unsigned short TEXT_NODE = 3;
4      const unsigned short COMMENT_NODE = 8;
5      readonly attribute unsigned short nodeType;
6      readonly attribute Document ownerDocument;
7      readonly attribute Node parentNode;
8      readonly attribute Node previousSibling;
9      readonly attribute Node nextSibling;
10
11     Node appendChild(in Node newChild);
12     Node insertBefore(in Node newChild, in Node refChild);
13     Node removeChild(in Node oldChild);
14     Node replaceChild(in Node newChild, in Node oldChild);
15     readonly attribute Node firstChild;
16     readonly attribute Node lastChild;
17     readonly attribute NodeList childNodes;
18 }
```

Listing 2.8: DOM-Schnittstelle `Node`

Repräsentation eines XML-Dokuments auftreten. Demnach ist das Attribut `nodeType` der Diskriminator der Schnittstellen und ermöglicht die Unterscheidung der Knoten in Kommentar-, Text- oder Elementknoten. Jeder Knoten erhält weiterhin die Möglichkeit über das Attribut `ownerDocument` auf das ihn enthaltende Dokument, durch `parentNode` auf den Vaterknoten im Baum und über `previousSibling` und `nextSibling` auf die Geschwisterknoten lesend zuzugreifen. Eine direkte Manipulation dieser Attribute wird allerdings durch die Attributeigenschaft `readonly` ausgeschlossen. Stattdessen wird eine Veränderung der untergeordneten Baumstruktur durch die Operationen `appendChild`, `insertBefore`, `removeChild` und `replaceChild` für Elementknoten ermöglicht. Für diese sind auch die Attribute `firstChild`, `lastChild` und `childNodes` definiert.

Mit Hilfe der Methode `appendChild`, die einen Knoten als letztes Kind eines Elements einfügt und erst in der Schnittstelle `Element` definiert wird, können die formalen Spezifikationen

erfolgen:

- Das Attribut `previousSibling` zeigt stets auf den vorangehenden und `nextSibling` auf den nachfolgenden Knoten.

$$\left[\begin{array}{l} n.\texttt{appendChild}(n_i); \\ n.\texttt{appendChild}(n_{i+1}) \end{array}\right]$$
$$n_i = n_{i+1}.\texttt{previousSibling}$$
$$n_{i+1} = n_i.\texttt{nextSibling}$$

- Das Attribut `parentNode` verweist auf den Elternknoten.

$$\left[\ n.\texttt{appendChild}(n_i)\ \right]$$
$$n = n_i.\texttt{parentNode}$$

- Ein neu angelegtes Element besitzt keinen Elternknoten, keinen Vorgänger, keinen Nachfolger und verweist auf das anlegende Dokument. Analoges gilt für Text- und Kommentarknoten.

$$\left[\ e := d.\texttt{createElement}(t)\ \right]$$
$$e.\texttt{ownerDocument} = d$$
$$e.\texttt{parentNode} = \texttt{nil}$$
$$e.\texttt{previousSibling} = \texttt{nil}$$
$$e.\texttt{nextSibling} = \texttt{nil}$$

Eine Spezifikation der restlichen Methoden ist erst später möglich und sinnvoll, weil diese nur für die Kompositum-Schnittstelle `Element` definiert sind.

Die Schnittstelle `Text`, die in Listing 2.10 definiert wird, ist eine Spezialisierung der Schnittstelle `CharacterData` (Listing 2.9). Sie beschreibt den Zugriff auf textuellen Inhalt innerhalb des Dokuments. Die Schnittstelle ist ein Blatt innerhalb der Komponente-Kompositum-Struktur

```
1 interface CharacterData : Node {
2     attribute DOMString data;
3 }
```

Listing 2.9: DOM-Schnittstelle `CharacterData`

und kann deshalb keine weiteren Kinder haben. Die formale Spezifikation beschränkt sich auf das Attribut `data` der Schnittstelle `CharacterData`:

```
1 interface Text: CharacterData {
2               };
```

Listing 2.10: DOM-Schnittstelle `Text`

- Das Attribut `nodeType` liefert den Wert für Textknoten und mit `data` kann auf den repräsentierten Text zugegriffen werden.

$$\begin{bmatrix} c := d.\texttt{createTextNode}(s) \end{bmatrix}$$
$$c.\texttt{nodeType} = \texttt{TEXT\_NODE}$$
$$c.\texttt{data} = s$$

- Wird `data` gesetzt, hat das Attribut beim Zugriff den gleichen Wert.

$$\begin{bmatrix} c.\texttt{data} := s \end{bmatrix}$$
$$c.\texttt{data} = s$$

```
1 interface Comment: CharacterData {
2               };
```

Listing 2.11: DOM-Schnittstelle `Comment`

Die Schnittstelle `Comment` in Listing 2.11 zur Repräsentation von Kommentarknoten ist analog zur Schnittstelle `Text` definiert.

In Listing 2.12 wird die Schnittstelle `Element` festgelegt, die für die Kompositum-Struktur im Dokument steht. Sie ermöglicht den Zugriff auf die Attribute eines Elements über die Methoden `getAttributeNode`, `setAttributeNode` und `removeAttributeNode` unter Übergabe der zu manipulierenden Attribute gemäß der Schnittstelle `Attr` aus Listing 2.7 bzw. unter Angabe der Attributnamen. Attribute können ausgelesen, gesetzt und gelöscht werden. Ebenfalls stehen die Methoden `getAttribute`, `setAttribute` und `removeAttribute` mit analoger Funktionalität zur Verfügung, die allerdings nicht über die Schnittstelle `Attr` sondern nur über den Attributnamen auf die Attribute zugreifen. Bevor die formale Spezifikation dieser Methoden erfolgt, sind zunächst die aufgeschobenen Methoden aus der Schnittstelle `Node` zu definieren.

Das Verhalten des Konstruktors beschreibt folgende Spezifikation:

- Das Attribut `nodeType` liefert den Wert für Elementknoten nach dem Erzeugen eines neuen Elements `e`, `tagName` den Tagnamen und `lastChild` sowie `firstChild` ver-

```
1  interface Element: Node {
2      readonly attribute DOMString tagName;
3      readonly attribute NameNodeMap attributes;
4      Attr getAttributeNode(in DOMString name);
5      Attr setAttributeNode(in Attr newAttr);
6      Attr removeAttributeNode(in Attr oldAttr);
7      DOMString getAttribute(in DOMString name);
8      void setAttribute(in DOMString name, in DOMString
           value);
9      void removeAttribute(in DOMString name);
10     NodeList getElementsByTagName(in DOMString name);
11 }
```

Listing 2.12: DOM-Schnittstelle `Element`

weisen auf keine Knoten, weil noch keine Kinderknoten eingefügt wurden.

$$\begin{aligned} [\ e := d.\texttt{createElement}(t)\ ] \\ e.\texttt{nodeType} = \texttt{ELEMENT\_NODE} \\ e.\texttt{tagName} = t \\ e.\texttt{lastChild} = \texttt{nil} \\ e.\texttt{firstChild} = \texttt{nil} \end{aligned}$$

Mit der Methode `appendChild` kann ein Knoten als letztes Kind unterhalb eines Elements eingefügt werden, sie ist der Konstruktor für die baumartige Hierarchie. Um die Übersichtlichkeit in der formalen Spezifikation zu verbessern, wird der Rückgabewert der Methode nur in der ersten Definition betrachtet:

- Die Methode `appendChild` liefert den eingefügten Knoten.

$$[\ n_i := e.\texttt{appendChild}(n_i)\ ] \Leftrightarrow [\ e.\texttt{appendChild}(n_i)\ ]$$

- Der Knoten n_i soll bei e_2 eingefügt werden, obwohl er schon in dem Dokument an e_1 existiert, dann entspricht dies dem einmaligen Einfügen von n_i in e_2. Mit anderen Worten wird n_i implizit aus e_1 entfernt und dann in e_2 eingefügt.

$$\left[\begin{array}{l} e_1.\texttt{appendChild}(n_i); \\ e_2.\texttt{appendChild}(n_i) \end{array}\right] \Leftrightarrow [\ e_2.\texttt{appendChild}(n_i)\ ]$$

Mit der Methode `insertBefore` werden Knoten in Elemente eingefügt. Falls der zweite Parameter gesetzt ist, wird vor diesem Knoten ansonsten als letztes Kind dieses Elements in der Hierarchie eingefügt. Der Rückgabewert der Methode wird nur in der ersten Spezifikation angegeben:

- Die Methode `insertBefore` liefert den eingefügten Knoten.

$$\left[\ n_i := e.\texttt{insertBefore}(n_i, n_j)\ \right] \Leftrightarrow \left[\ e.\texttt{insertBefore}(n_i, n_j)\ \right]$$

- Der Knoten n_i soll vor n_{i+1} eingefügt, werden, also muss er direkt vor dem Einfügen von n_{i+1} als letzter Knoten eingefügt werden.

$$\left[\begin{array}{l} e.\texttt{appendChild}(n_{i+1}); \\ e.\texttt{insertBefore}(n_i, n_{i+1}) \end{array}\right] \Leftrightarrow \left[\begin{array}{l} e.\texttt{appendChild}(n_i); \\ e.\texttt{appendChild}(n_{i+1}) \end{array}\right]$$

- Der Knoten n_i soll vor n_{i+1} eingefügt werden, der aber nicht als letzter eingefügt wurde, dann kann man auch erst n_i einfügen und dann den Knoten n_k als letzten einfügen.

$$\left[\begin{array}{l} e.\texttt{appendChild}(n_k); \\ e.\texttt{insertBefore}(n_i, n_{i+1}) \end{array}\right] \Leftrightarrow \left[\begin{array}{l} e.\texttt{insertBefore}(n_i, n_{i+1}); \\ e.\texttt{appendChild}(n_k) \end{array}\right]$$

Mit der Methode `removeChild` werden Knoten aus der Hierarchie entfernt. Sie liefert den gelöschten Knoten als Rückgabewert, der wieder nur in der ersten formalen Spezifikation betrachtet wird:

- Die Methode `removeChild` liefert den gelöschten Knoten.

$$\left[\ n_i := e.\texttt{removeChild}(n_i)\ \right] \Leftrightarrow \left[\ e.\texttt{removeChild}(n_i)\ \right]$$

- Die drei Attribute `parentNode`, `previousSibling` und `nextSibling` sind nicht mehr gesetzt, nachdem ein Knoten n_i gelöscht wurde.

$$\begin{aligned} &\left[\ e.\texttt{removeChild}(n_i)\ \right] \\ n_i.\texttt{parentNode} &= nil \\ n_i.\texttt{previousSibling} &= nil \\ n_i.\texttt{nextSibling} &= nil \end{aligned}$$

- Soll das Element n_i aus der Hierarchie entfernt werden, nachdem es zuvor eingefügt wurde, heben sich die beiden Methoden auf.

$$\left[\begin{array}{l} e.\texttt{appendChild}(n_i); \\ e.\texttt{removeChild}(n_i) \end{array}\right] \Leftrightarrow \left[\ \epsilon\ \right]$$

- Soll ein Knoten n_i gelöscht werden, nachdem ein anderer Knoten n_k zunächst als letzter Knoten eingefügt wurde, kann auch erst das Element gelöscht und dann das letzte Element eingefügt werden.

$$\left[\begin{array}{l} e.\texttt{appendChild}(n_k); \\ e.\texttt{removeChild}(n_i) \end{array}\right] \Leftrightarrow \left[\begin{array}{l} e.\texttt{removeChild}(n_i); \\ e.\texttt{appendChild}(n_k) \end{array}\right]$$

Die Methode `replaceChild` ersetzt einen Kinderknoten an einem Element durch einen neuen Knoten. Der alte Knoten wird aus der Hierarchie entfernt. Der Rückgabewert wird wieder zunächst in der ersten Spezifikation festgelegt und dann vernachlässigt:

- Die Methode `replaceChild` liefert den zu ersetzenden Knoten.

$$\left[\ n_j := e.\texttt{replaceChild}(n_i, n_j)\ \right] \Leftrightarrow \left[\ e.\texttt{replaceChild}(n_i, n_j)\ \right]$$

- Wird erst ein Knoten n_j eingefügt und dann durch einen Knoten n_i ersetzt, so kann auch nur der Knoten n_i eingefügt werden.

$$\left[\begin{array}{l} e.\texttt{appendChild}(n_j); \\ e.\texttt{replaceChild}(n_i, n_j) \end{array}\right] \Leftrightarrow \left[\ e.\texttt{appendChild}(n_i)\ \right]$$

- Soll ein Knoten n_j durch n_i ersetzt werden und wurde zuvor ein weitere Knoten n_k eingefügt, dann kann auch erst die Ersetzung vor dem Einfügen vorgenommen werden.

$$\left[\begin{array}{l} e.\texttt{appendBefore}(n_k); \\ e.\texttt{replaceChild}(n_i, n_j) \end{array}\right] \Leftrightarrow \left[\begin{array}{l} e.\texttt{replaceChild}(n_i, n_j); \\ e.\texttt{appendChild}(n_k) \end{array}\right]$$

Das Attribut `lastChild` ermöglicht einen Zugriff auf das letzte Kind eines Elements, falls dieses existiert:

- Nach dem Einfügen eines Knotens n_k an letzter Stelle, verweist das Attribut `lastChild` auf diesen.

$$\begin{array}{c} \left[\ e.\texttt{appendChild}(n)\ \right] \\ e.\texttt{lastChild} = n \end{array}$$

Mit dem Attribut `firstChild` wird der Zugriff auf das erste Kind eines Elements ermöglicht, falls dieses existiert:

- Wird ein Knoten n als erster Kinderknoten in ein Element e eingefügt, so verweist das Attribut `firstChild` auf diesen.

$$\begin{array}{c} \left[\begin{array}{l} e := d.\texttt{createElement}(t); \\ e.\texttt{appendChild}(n) \end{array}\right] \\ e.\texttt{firstChild} = n \end{array}$$

- Wurden mindestens zwei Knoten n_i und n_{i+1} in ein Element e eingefügt und wird anschließend auf den ersten Knoten mittels `firstChild` zugegriffen, so kann man auch vor dem Einfügen von n_{i+1} auf das erste Element zugreifen.

$$\left[\begin{array}{l} e.\texttt{appendChild}(n_i); \\ e.\texttt{appendChild}(n_{i+1}); \\ n := e.\texttt{firstChild} \end{array}\right] \Leftrightarrow \left[\begin{array}{l} e.\texttt{appendChild}(n_i); \\ n := e.\texttt{firstChild}; \\ e.\texttt{appendChild}(n_{i+1}) \end{array}\right]$$

```
1 interface NodeList {
2     Node item(in unsigned long index);
3     readonly attribute unsigned long length;
4 }
```

Listing 2.13: DOM-Schnittstelle `NodeList`

Die Methode `childNodes` liefert eine Liste mit den Kinderknoten eines Elements. Die Schnittstelle für die Liste `NodeList` ist in Listing 2.13 dargestellt. Das Attribut `length` gibt die Länge einer Liste an und die Methode `item` ermöglicht die indizierte Selektion einzelner Elemente aus der Liste. Leider wird in der Schnittstelle kein Konstruktor definiert, so dass eine formale Spezifikation nur über die Methode `childNodes` für Knoten möglich ist:

- Ist ein Knoten n als letztes Kind an einem Element n eingefügt worden, so ermöglicht die Methode `item` den Zugriff auf dieses, indem die Länge der Liste als Index übergeben wird.

$$\left[\ e.\texttt{appendChild}(n)\ \right]$$
$$n = e.\texttt{childNodes}.\texttt{item}(e.\texttt{childNodes}.\texttt{length})$$

- Soll ein Element der Liste extrahiert werden, das nicht das letzte Element ist, so ist dies unabhängig davon, ob das letzte Element vor oder nach dem Zugriff eingefügt wurde.

$$\left[\begin{array}{l} e.\texttt{appendChild}(n_k); \\ l := e.\texttt{childNodes}.\texttt{length}; \\ n_i := e.\texttt{childNodes}.\texttt{item}(l-j) \end{array}\right] \Leftrightarrow \left[\begin{array}{l} l := e.\texttt{childNodes}.\texttt{length}+1; \\ n_i := e.\texttt{childNodes}.\texttt{item}(l-j); \\ e.\texttt{appendChild}(n_k) \end{array}\right]$$

Ziemlich analog zur Spezifikation der Methode `childNodes` kann die Formalisierung der Methode `getElementsByTagName` erfolgen, wobei auf die entsprechenden Tagnamen der Elemente Rücksicht genommen werden muss. Es werden dann nur die Kinderknoten in einer Knotenliste zurückgegeben, die sowohl Elemente sind, als auch mit ihrem Tag dem Parameter entsprechen.

Das Attribut `attributes` der Schnittstelle `Element` erlaubt den Zugriff auf die Attribute eines Elements. Diese werden in der Struktur `NamedNodeMap` vorgehalten, deren Schnittstelle in Listing 2.14 zu sehen ist. Es sind drei Methoden vorgegeben, die den Zugriff auf Attribute, das Einfügen und das Löschen von Attributen regeln und folgender Formalisierung unterliegen:

- Wird ein Attribut a erzeugt, auf den Wert S gesetzt und mit `setNamedItem` in die Map m eingefügt, so liefert `getNamedItem` anschließend dieses Attribut a.

$$\left[\begin{array}{l} a := d.\texttt{createAttribute}(n); \\ a.\texttt{value} := s; \\ m.\texttt{setNamedItem}(a) \end{array}\right]$$
$$a = m.\texttt{getNamedItem}(n)$$

```
1 interface NamedNodeMap {
2     Attr getNamedItem(in DOMString name);
3     Attr setNamedItem(in Attr arg);
4     Attr removeNamedItem(in DOMString name);
5 }
```

Listing 2.14: DOM-Schnittstelle `NamedNodeMap`

- Wird ein Attribut a_1 mit Namen n_1 erzeugt und in die Map m durch `setNamedItem` eingefügt und wird außerdem ein Attribut a_2 mittels seines Namens n_2 über `getNameItem` ausgelesen, so ist das Ergebnis unabhängig von der Reihenfolge der beiden Operationen, falls sich n_1 und n_2 unterscheiden.

$$\left[\begin{array}{l} a_1 := d.\texttt{createAttribute}(n_1); \\ a_1.\texttt{value} := s; \\ m.\texttt{setNamedItem}(a_1); \\ a_2 = M.\texttt{getNamedItem}(n_2) \end{array}\right] \Leftrightarrow \left[\begin{array}{l} a_2 = m.\texttt{getNamedItem}(n_2); \\ a_1 := d.\texttt{createAttribute}(n_1); \\ a_1.\texttt{value} := s; \\ m.\texttt{setNamedItem}(a_1) \end{array}\right]$$

- Ein zunächst mittels `setNamedItem` eingefügtes Attribut a_1 ist nach der Anwendung der Operation `removeNamedItem` nicht mehr in der Map m.

$$\left[\begin{array}{l} a := D.\texttt{createAttribute}(n); \\ a.\texttt{value} := s; \\ m.\texttt{setNamedItem}(a); \\ m.\texttt{removeNamedItem}(n) \end{array}\right] \Leftrightarrow \left[\begin{array}{l} a := d.\texttt{createAttribute}(n); \\ a.\texttt{value} := s \end{array}\right]$$

- Die Operationen `setNamedItem` und `removeNamedItem` sind unabhängig in ihrer Reihenfolge, falls die Operationen sich auf unterschiedliche Attribute beziehen.

$$\left[\begin{array}{l} a := d.\texttt{createAttribute}(n_1); \\ a.\texttt{value} := s; \\ m.\texttt{setNamedItem}(a); \\ m.\texttt{removeNamedItem}(n_2) \end{array}\right] \Leftrightarrow \left[\begin{array}{l} m.\texttt{removeNamedItem}(n_2); \\ a := d.\texttt{createAttribute}(n_1); \\ a.\texttt{value} := s; \\ m.\texttt{setNamedItem}(a) \end{array}\right]$$

Die Methoden `setAttributeNode`, `getAttributeNode`, `removeAttributeNode` werden analog den Methoden der Schnittstelle `NamedNodeMap` formalisiert, weshalb auf deren Spezifikation an dieser Stelle verzichtet wird.

Mit der Methode `setAttribute` wird der Wert eines Attributs eines Elements neu gesetzt. Ist das Attribut noch nicht vorhanden, wird es erzeugt. Die Methode `getAttribute` liefert den aktuellen Wert eines Attributs, während das Löschen eines Attributs mit der Methode `removeAttribute` erfolgt. Das Verhalten der Methoden wird durch folgende formale Spezifikation bestimmt:

- Die Methode `getAttribute` wird durch die Methode `getNamedItem` aus der Schnittstelle `NamedNodeMap` (siehe Listing 2.14) definiert.

$$e.\texttt{getAttribute}(n) = e.\texttt{getAttributeNode}(n).\texttt{value}$$

- Für die Methode `setAttribute` erfolgt eine Abbildung auf die Methode `setNamedItem` unter vorheriger Erzeugung eines neuen Attributs.

$$\left[\ e.\texttt{setAttribute}(n, v)\ \right] \Leftrightarrow \left[\begin{array}{l} a := e.\texttt{ownerDocument}.\\ \qquad\qquad\qquad\texttt{createAttribute}(n);\\ a.\texttt{value} := v;\\ e.\texttt{setAttributeNode}(a) \end{array}\right]$$

- Die Methode `removeAttribute` wird durch die Methode `removeNamedItem` spezifiziert.

$$\left[\ e.\texttt{removeAttribute}(n)\ \right] \Leftrightarrow \left[\ e.\texttt{removeAttributeNode}(n)\ \right]$$

Das Attribut `attribute` ermöglicht einen lesenden Zugriff auf die Attribute eines Elements über die Schnittstelle `NamedNodeMap`.

- Wird auf das Attribut `attributes` eines Elements e die Operation `getNamedItem` angewendet, entspricht dies der Ausführung der Methode `getAttributeNode` für das Element e.

$$e.\texttt{attributes}.\texttt{getNamedItem}(n) = e.\texttt{getAttributeNode}(n)$$

Anwendungsbeispiele

Im restlichen Abschnitt werden zwei Beispiele für die Anwendung des DOM präsentiert.

Beispiel 2.8
In diesem Beispiel wird das XML-Fragment

```
8   <book catalog="Varia">
9     <title>Lotte in Weimar</title>
10    <author>Thomas Mann</author>
11    <condition>Einband fingerfleckig, Rücken
        verblaßt</condition>
12    <price currency="EUR">8.00</price>
13  </book>
```

des Dokuments aus Listing 2.1 betrachtet. Eine Erzeugung dieses Fragments aus einem Programm heraus wird unter Verwendung des DOM mit folgenden Anweisungen erreicht. Die Variable `d` ist dabei eine Variable der Schnittstelle `Document` und verweist auf ein Objekt, das diese implementiert.

```
1  bk = d.createElement("book");
2  bk.setAttribute("catalog","Varia");
3  ttl = d.createElement("title");
4  ttl.appendChild(d.createTextNode("Lotte in Weimar"));
5  athr = d.createElement("author");
6  athr.appendChild(d.createTextNode("Thomas Mann"));
7  cndtn = d.createElement("condition");
8  cndtn.appendChild(d.createTextNode("Einband fingerfleckig
       , Rücken verblaßt"));
9  prc = d.createElement("price");
10 prc.appendChild(d.createTextNode("8.00"));
11 prc.setAttribute("currency","EUR");
12 bk.appendChild(ttl);
13 bk.appendChild(athr);
14 bk.appendChild(cndtn);
15 bk.appendChild(prc);
```

Es zeigt sich, dass mit der Anwendung der Methode `createElement` (1,3,5,7,9), `createTextNode` (4,6,8,10), `setAttribute` (2,11) und `appendChild` (4,6,8,10,12-15) aus dem DOM das XML-Fragment auf einfache Weise im Programm kreiert werden kann. □

Da das DOM eine universelle Schnittstelle für XML-Dokumente bereitstellt, also für jede Auszeichnungssprache verwendbar ist, können auch ungültige Dokumente erzeugt werden, wie das nachstehende Beispiel demonstriert.

Beispiel 2.9
Es bezieht sich auf die Programmanweisung aus dem vorherigen Beispiel. In diesem sei die folgende Zeile ausgetauscht.

```
5  athr = d.createElement("artist");
```

Dies führt zu DOM-Instanzen, die nachstehendes XML-Fragment repräsentieren:

```
8        <book catalog="Varia">
9           <title>Lotte in Weimar</title>
10          <artist>Thomas Mann</artist>
11          <condition>Einband fingerfleckig , Rücken
               verblaßt</condition>
12          <price currency="EUR">8.00</price>
13       </book>
```

Für dieses Fragment ergibt die Überprüfung der Gültigkeit gemäß der Sprachbeschreibung aus Beispiel 2.2 eine Verletzung dieser Eigenschaft. □

Abschließend kann demnach festgestellt werden, dass das DOM eine universelle Schnittstelle für die Verarbeitung von XML in einer Programmiersprache bereitstellt. Sie realisiert einen ein-

heitlichen und austauschbaren Zugriff für XML-basierte Anwendungen. Eine Überprüfung der Gültigkeit gemäß einer zu Grunde liegenden Sprachbeschreibung findet, wie das letzte Beispiel zeigt, nicht statt. So kann im repräsentierten XML-Dokument beliebig eingefügt und gelöscht werden, solange nur die Baumstruktur nicht verletzt wird.

2.3.3 Implementierungen und Erweiterungen

Nachdem im letzten Abschnitt die wichtigsten Schnittstellen des DOM definiert und deren Anwendung an Beispielen illustriert wurden, zählt dieser Abschnitt aktuelle DOM-Implementierungen auf und gibt einen Einblick in den aktuellen Stand des Standardisierungsprozesses. Das DOM definiert, wie gezeigt, lediglich Schnittstellen und legt nicht fest, wie diese zu implementieren sind. Dies bedeutet für ein Programm, das das DOM einsetzen möchte, die Einbindung einer DOM-Implementierung. Für die Programmiersprache Java sind inzwischen sowohl Implementierungen namhafter Unternehmen, wie

- XML Parser for Java [IBM03] von IBM,
- Java API for XML Precessing (JAXP) [Sun01b] von Sun,
- XML Developer's Kit for Java [Ora02] von Oracle,

als auch Open-Source-Entwicklungen, u. a.

- Xerces Java Parser [Apa01] vom Apache XML Project und
- GNU JAXP Project [Fre01] der Free Software Foundation

verfügbar. Durch die Festlegung aller Implementierungen auf den gemeinsamen Standard DOM ergibt sich für den Entwickler der Vorteil, dass die eingesetzte DOM-Implementierung nahezu beliebig austauschbar ist.

Die Spezifikation des DOM gliedert sich in drei Ebenen („level"), die aufeinander aufbauen. Inzwischen wurde die 2. Ebene (DOM Level 2) als Empfehlung des W3C [W3C00a] herausgegeben. Gegenüber der 1. Ebene wird das Modell um zusätzliche Funktionalitäten erweitert. So gibt es seitdem Schnittstellen zum Traversieren eines Dokuments und zur Selektion von Dokumentbereichen. Zusätzlich wird ein Ereignismodell definiert und der Zugriff auf Präsentationsinformationen, sogenannte *Style-Sheets* [W3C96, W3C98a], erlaubt. Auch können nun Dokumente durch unterschiedliche Sichten („view") angesprochen werden. Die gerade in Vorbereitung befindliche dritte Ebene der Spezifikation definiert weiter Schnittstellen zum Laden und Speichern von Dokumenten und zum Zugriff auf das Dokument über XPath-Ausdrücke. Darüber hinaus gibt es Methoden, um die Gültigkeit eines Dokuments während der Laufzeit der Anwendung zu überprüfen. Trotz dieser vielversprechenden Neuerung kann das DOM aber die statische Gültigkeit nicht zum Zeitpunkt der Programmübersetzung garantieren.

2.4 Verarbeitung syntaktischer Strukturen

Applikationen, die auf XML basieren, haben zweifellos mit der Verarbeitung syntaktischer Strukturen [Lin79, Lin81] zu tun, einer gut 20 Jahre alten Idee, mit der sich unter anderem das Gebiet des Übersetzerbaus befasst. Im Folgenden soll der Aspekt der Verarbeitung von syntaktischen Strukturen, d. h. der Generierung und Umformung von Strukturen grammatikbasierter Sprachen, näher betrachtet werden.

Programmgeneratoren sind Programme, die mittels einer Eingabe Programmcode als Ausgabe erzeugen. Typische Beispiele hierfür sind Compiler-Compiler aus dem Übersetzerbau und CASE-Werkzeuge aus der Softwaretechnik. Beim Generieren von Programmen per Programm ist es wesentlich, dass die erzeugten Programmteile syntaktisch korrekt sind. Ist dies nicht der Fall, kann das erzeugte Programm nicht ausgeführt werden. Für eine korrigierte Version muss das erzeugende Programm mit sämtlichen Eingaben erneut ablaufen.

Das Problem bei der Programmierung von Programmgeneratoren liegt nun darin, Sprachmittel zur Verfügung zu stellen, die es erlauben, bereits zur Zeit der Programmübersetzung eine Aussage darüber zu treffen, ob die vom übersetzten Programm erzeugten Programme syntaktisch korrekt sind. Es ist einsichtig, dass Ansätze zur Programmierung von Programmgeneratoren, die die zu erzeugenden syntaktischen Strukturen – die Programme oder Programmteile – mit Operationen auf Zeichenkettenebene verarbeiten, dieser Anforderung nicht gerecht werden können. Fehler können bei diesem Verfahren lediglich empirisch, durch aufwendiges Testen ausgeschlossen werden. Zur Lösung dieses Problems muss eine Sprachbeschreibung, die in der Regel in Form einer Grammatik definiert ist, für die zu erzeugenden syntaktischen Strukturen vorliegen. Dabei werden für die Nichtterminalsymbole der Grammatik neue Datentypen definiert und zugeordnet, was im Wesentlichen der abstrakten Syntax entspricht. Zusätzlich werden Operationen eingeführt, die es erlauben, aus Werten der Basisdatentypen diese Nichtterminaltypen zu erzeugen, sowie weitere Operationen, um nach unterschiedlichen Regelvarianten zu selektieren.

Die Konstruktoroperationen, die Werte der Nichtterminaltypen erzeugen, erhalten als Parameter syntaktische Strukturen in konkreter Syntax, die zusätzlich auch, an zur zu Grunde liegenden Sprachbeschreibung konformen Positionen, Variablen der Nichtterminaltypen enthalten können. Damit ist es möglich, dass bereits generierte Strukturen ineinander geschachtelt werden können, und nicht nur von links nach rechts erzeugt werden müssen, was bei einer Verarbeitung auf Zeichenkettenebene erforderlich ist. Dies hat den Vorteil, dass der Anwendungsprogrammierer nicht ausschließlich mit der ungewohnten abstrakten Syntax arbeiten muss, sondern weiterhin die vertrautere konkrete Syntax einsetzen kann.

Eine Übersetzung dieser Generatorprogramme geschieht mit einem erweiterten Compiler, der neben dem eigentlichen Programm zusätzlich die zu Grunde liegende Grammatik der zu generierenden syntaktischen Strukturen einliest und berücksichtigt. Der wesentliche Vorteil dieses Vorgehens besteht, neben einer übersichtlicheren Programmstruktur, in der Sicherstellung der syntaktischen Korrektheit der durch das Programm generierten syntaktischen Strukturen zur Zeit der Programmübersetzung. Die Einschränkungen werden darin gesehen, dass für große Gram-

matiken möglicherweise eine hohe Anzahl von Nichtterminaltypen entstehen und darin, dass bei Prozeduren, die auf unterschiedlichen Nichtterminaltypen arbeiten, aber die gleiche Funktionalität erfüllen, durch das strikte Typsystem eine mühsame und aufwendige Implementierung notwendig wird.

Es liegt auf der Hand, dass sich der beschriebene Ansatz sehr gut für eine Programmiermethodik XML-basierter Anwendungen adaptieren lässt. Denn die zu verarbeitenden XML-Dokumente sind ebenfalls syntaktische Strukturen und eine Sprachbeschreibung ist in Form einer DTD oder eines XML-Schemas gegeben.

2.5 Web-Anwendungen

Nachdem in Abschnitt 2.1 die Extensible-Markup-Language vorgestellt wurde, die eine universelle Möglichkeit für den Datenaustausch vorsieht, wendet sich dieser Abschnitt dem *World-Wide Web* (WWW) zu, dem mittlerweile größten und am meisten genutzten Informationssystem. Mit der *Hypertext-Markup-Language* (HTML) realisiert das WWW sicherlich die am weitesten verbreitete Anwendung von XML.

Das World-Wide Web, das sich im wissenschaftlichen Umfeld entwickeln konnte, wird inzwischen zum Großteil kommerziell genutzt. Betrachtet man die breiten Nutzungsmöglichkeiten, die das Web inzwischen bietet, angefangen von Bankanwendungen bis hin zum Versandhaus, so kann man nicht mehr nur von einer bloßen Informationssammlung oder einem Hyperlinksystem im ursprünglichen Sinne sprechen. Vielmehr werden von einzelnen Anbietern vollwertige Anwendungen realisiert, die das WWW lediglich als Infrastruktur der Implementierung nutzen; Anwendungen dieser Art werden im folgenden als *Web-Anwendungen* bezeichnet.

Der Abschnitt beginnt mit einer grundlegenden Einführung in den Aufbau des Internets, um ein Verständnis für die Problematik zu schaffen, bevor auf die verschiedenen Aspekte der Informationspräsentation im WWW eingegangen wird. Als weiterführende Literatur sei an dieser Stelle auf [Kro95, Tol97b] verwiesen. Die traditionelle Präsentation statischer Dokumente wird in Abschnitt 2.5.2 vorgestellt, während im Nachfolgenden die erweiterten Ansätze für dynamisch erstellten Inhalt diskutiert werden.

2.5.1 Das Internet und seine Dienste

Das Internet wird gebildet von einem weltweiten Verbund von Rechnern, die über ihre Verknüpfungen untereinander Daten austauschen. Die Computer werden durch sogenannte *Internet-Protokoll-Adressen* (IP-Nummern) eindeutig identifiziert. Trotz der Notwendigkeit von eindeutigen IP-Nummern sind die Rechner im Internet nicht hierarchisch strukturiert, was ursächlich in den militärischen Anfängen begründet liegt. Eine der damaligen Hauptanforderungen an das Netz war eine möglichst hohe Ausfallsicherheit. Mit einer dezentralen, netzartigen Organisati-

on, die wesentlich zum Erfolg des Internets beitrug, kann im Gegensatz zu einer hierarchischen Struktur diese Anforderung besser erfüllt werden. Trotzdem muss die Eindeutigkeit der IP-Nummern sichergestellt werden, was durch eine zentrale Vergabe geschieht. Zusätzlich werden durch das *Domain-Name-System* (DNS) Rechnernamen vergeben, denen die IP-Nummern der Rechner eindeutig zugeordnet werden, um das Arbeiten mit Computern im Internet für die Anwender zu vereinfachen.

Damit eine Kommunikation im Internet über unterschiedliche Software- und Hardware-Grenzen hinweg überhaupt funktioniert, müssen die eingesetzten Protokolle standardisiert sein. Im Internet erfolgt dies durch die *Request-For-Comments-Dokumente* (RFC). In diesen werden die einzelnen Protokolle definiert und beschrieben, die sich unterschiedlichen logischen Schichten (ähnlich dem ISO/OSI-Schichtenmodell (ISO 7498) [Int94, Tan96]) zuordnen lassen. Jede Schicht implementiert und kapselt eine bestimmte Funktionalität, auf die die nächsthöhere Schicht aufbauen kann. In dieser hierarchischen Anordnung befinden sich auf der untersten Ebene die Netzprotokolle der lokalen Netzwerksysteme, wie z. B. das Ethernet. Auf der nächsten Ebene, auf der auch das *Internet-Protocol* (IP) anzusiedeln ist, werden Netzverbindungsprotokolle realisiert, die für Verknüpfungen zwischen lokalen Netzwerksystemen sorgen. Darauf aufbauend liegt die Schicht der Transportprotokolle, in der das *Transfer-Control-Protocol* (TCP) für eine zuverlässige Kommunikation sorgt. Damit wird die korrekte Auslieferung von Daten zwischen verschiedenen Rechnern sichergestellt. Ganz oben in der Protokollhierarchie liegen die sogenannten *Dienstprotokolle*, die im Weiteren angesprochen werden. Der Benutzer braucht keine genaue Kenntnis über die technischen Details und den Aufbau der einzelnen Protokollschichten zu haben, lediglich die Funktionalität der Dienstprotokolle sollte dem Benutzer bekannt sein, damit er diese sinnvoll einsetzen kann.

Dienstprotokolle, oder kurz Dienste, realisieren praktisch schon spezielle Anwendungen, durch die das Internet erst sinnvoll genutzt wird. Die verschiedenen Dienste verrichten eine Vielzahl von unterschiedlichen Funktionalitäten. Um einen Dienst auf einem entfernt liegenden Rechner zu nutzen, muss dieser den Dienst implementieren, d. h. es muss ein Programm gestartet sein und ablaufen, das das zugehörige Dienstprotokoll versteht. Ist dies der Fall, ist es möglich durch ein entsprechendes ausführbares Programm von einem anderen Computer aus, den entfernt liegenden Rechner mittels dieses Dienstes zu erreichen. Beispielsweise ist es mit dem sehr einfachen Dienst *ping* möglich, zu ermitteln, ob ein bestimmter Rechner vom aktuellen Computer aus über das Internet erreichbar ist. Weitere Dienste sind der Dienst *telnet*, mit dem es möglich ist, sich auf einem entfernt liegenden Rechner anzumelden und auf diesem zu arbeiten, der Dienst *ftp* (File Transfer Protocol), mit dem Dateien zwischen zwei Rechnern hin- und herkopiert werden können, und der Dienst zum Verschicken von elektronischer Mail (*Email*) mittels SMTP (Simple Mail Transfer Protocol).

Der populärste Dienst des Internets ist aber inzwischen das World-Wide Web, für das der Begriff Internet bereits zum Synonym geworden ist. Das *Hypertext-Transfer-Protocol* (HTTP) realisiert diesen Dienst. Die folgenden Zahlen verdeutlichen mit welcher raschen Geschwindigkeit das Wachstum des Internets erfolgte. Waren im Januar 1993 weltweit erst 1.313.000 Rechner ans Internet angeschlossen, betrug die Anzahl im Januar 2003 bereits 171.638.297 [Int03]. Ähnlich

rasant ist die Entwicklung bei den im Internet arbeitenden WWW-Servern, die von 130 Rechner im Juni 1993 auf 40.444.778 im Mai 2003 anwuchsen [Net03]. Damit hat das WWW wesentlich zur Verbreitung des Internets beigetragen.

Die Arbeitsweise des Webs folgt einer typischen Client/Server-Architektur. WWW-Server halten dabei Daten und Informationen vor, die von Client-Rechnern im Internet bei Bedarf abgerufen werden können. Client-Rechner können sich also, falls notwendig, mit Servern verbinden und mittels eines geeigneten Programms dort abgelegte Informationen darstellen und verarbeiten. Dies leisten unter anderem die weit verbreiteten grafischen WWW-Browser *Netscape Communicator* oder *Internet Explorer*. Jeder Internetteilnehmer kann aber nicht nur über einen Browser im WWW abgelegte Informationen nutzen, sondern er kann zusätzlich auch einen eigenen WWW-Server einrichten, dort eigene Daten ablegen und damit zum Informationsangebot im Web beitragen. Ein solcher Rechner sollte aber sinnvollerweise permanent arbeiten und dauerhaft mit dem Internet verbunden sein.

2.5.2 Präsentation von statischen Dokumenten

Wie bereits im letzten Abschnitt erwähnt stellen WWW-Server Daten und Informationen zur Verfügung. Diese Daten werden sehr häufig in Form einer Datei abgelegt. Jede Datei wird dabei mit dem *Uniform-Resource-Locator* (URL) [BLMM94] im Web eindeutig identifiziert. Diese URL-Adresse besteht im ersten Teil zunächst aus der Art der Übertragung, dem Dienstprotokoll. Durch diese Angabe ist es möglich, dass WWW-Browser nicht nur die Kommunikation über den Dienst HTTP realisieren, sondern zusätzlich auch eine Verbindung über andere Dienste wie beispielsweise ftp erlauben. In einem zweiten Teil wird der Server eindeutig identifiziert, was sowohl über seinen vom DNS verwalteten Rechnernamen als auch durch seine IP-Adresse geschehen kann. Der letzte Teil besteht aus einer Pfadangabe und dem Dateinamen der bereitgestellten Datei.

Das World-Wide Web legt nicht fest, in welchem Format die auf den Web-Servern abgelegten Dateien vorliegen müssen. Damit besteht die Möglichkeit, Informationen in jeder Form abzulegen. Verbreitet sind Daten im Textformat (ASCII-Dateien), als Dokumente (PS- und PDF-Dateien) bis hin zu Bildern (JPEG, GIF u. a.) sowie Audio- und Video-Dateien. Selbst die Übertragung von ausführbaren Programmen in Binärformat oder Bytecode (Java) ist möglich.

Die meisten Dateien oder Dokumente, die Informationen im WWW bereithalten, liegen allerdings im Format der Hypertext-Markup-Language (HTML) [RLHJ97] vor, für deren Übertragung im Internet das Hypertext Transfer Protocol (HTTP) vorgesehen ist. Bei HTML handelt es sich um eine Anwendung von SGML und inzwischen unter dem Namen XHTML auch von XML [W3C00b], die für die Speicherung von Informationen in Form eines Hypertextsystems ausgelegt ist. Die im Hypertextsystem abgelegten Daten, man spricht auch von Seiten, werden durch einen Browser dem Benutzer geeignet präsentiert. Ein Hypertextsystem zeichnet sich durch sogenannte *Hyperlinks* aus, die verschiedene Dokumente oder Dokumentteile miteinander verknüpfen. Diese Verknüpfung wird vorgenommen, um weitere Informationen zu den vorliegenden Daten

in Beziehung zu setzen. Die durch einen Hyperlink referenzierte Datei kann sich wiederum auf beliebigen WWW-Servern im Internet befinden und wird mittels ihrer URL adressiert. Dem Benutzer bietet sich während der Präsentation einer Seite im Browser die Möglichkeit durch Anwählen von Hyperlinks, auf verwiesene Seiten zu verzweigen. Eine ausführliche Beschreibung von HTML findet sich in [Tol97a]; das nachstehende Beispiel gibt einen Eindruck von HTML.

Beispiel 2.10
Im Folgenden ist eine HTML-Seite als Ergebnis einer möglichen Suche im zentralen Verzeichnis antiquarischer Bücher zu sehen:

```
<html>
  <head><title>Suchergebnis</title></head>
  <body>
    <h1>Suchergebnis im Verzeichnis vom 20.2.2002</h1>
    <p>Die folgenden Titel wurden gefunden:</p>
    <ul>
      <li>
        <p>Thomas Mann,<i>Lotte in Weimar</i>,
          Einband fingerfleckig, Rücken verblaßt.
            EUR 8.00</p>
        <p><i>St. Jürgen Antiquariat</i>, Bestellung
          an:
          <a href="mailto:st.juergenantiquariat@t-
            online.de">
          st.juergenantiquariat@t-online.de</a></p>
      </li>
      <li>

      ⋮

      </li>
    </ul>
    <hr></hr>
    <a href="suche.html">Neue Suche</a>
  </body>
</html>
```

Listing 2.15: Dokument in HTML

Sie zeigt die Auflistung der gefundenen Bücher. Zusätzlich wird für jedes Buch aufgeführt, welches Antiquariat es führt. Über einen Hyperlink ist es möglich, eine Email als Bestellung an das entsprechende Antiquariat zu versenden. Am Fuße der Seite kann mit einem weiteren Hyperlink erneut auf die Seite für die Suche verzweigt werden.

Die Darstellung des Dokuments im Browser ist in Abbildung 2.1 zu sehen. □

Abbildung 2.1: Anzeige des HTML-Dokuments im Browser.

HTML ist eine Auszeichnungssprache zur Präsentation von Web-Seiten. Die ursprüngliche Idee von SGML, die logischen Dokumentenstruktur von der eigentlichen Präsentationsgestaltung durch den Browser strikt zu trennen, ist in HTML mit der Zeit mehr und mehr abgeschwächt worden. Durch eine Vielzahl neuer Elementtypen in den jüngsten Versionen, die ausschließlich der Formatierung und Präsentation des Inhalts dienen, wird HTML dem Anspruch von SGML immer weniger gerecht. Die Unzufriedenheit an dieser Entwicklung ist mit ein Grund, der zur Spezifikation von XML beitrug.

2.5.3 Dynamisierung des Webs auf der Client-Seite

Mit statischen HTML-Seiten ist es möglich, Informationen über das WWW einer breiten Öffentlichkeit zu präsentieren. Was nicht realisiert werden kann, ist ein Dialog mit dem Nutzer der angebotenen Informationen. Diese Einschränkung war die Hauptmotivation, die zur Entwicklung einer Reihe von Technologien führte, um den Dialog zwischen Benutzer und dem Web-Server zu ermöglichen. Informationssysteme, die diese Art von Kommunikation realisieren, werden im Folgenden als *Web-Anwendungen* bezeichnet. Typische Web-Anwendungen sind beispielsweise Banken, Versandhäuser und Auktionshäuser im WWW. Viele dieser Programme sind außerdem an ein eigenständiges Datenbanksystem angeschlossen, und verknüpfen damit dieses mit dem Web.

Nötig für einen Dialog im WWW ist einerseits die Eingabe von Daten auf der Clientseite, als auch eine Übertragung der Daten auf den Server, sowie deren dortige Verarbeitung. Mit Formu-

laren in HTML existiert bereits eine einfache Form zur Übermittlung von Daten vom Benutzer an die Web-Anwendung. Weitere Möglichkeiten, die zusätzlich mit einer stärkeren Dynamisierung der Anwendung auf der Client-Seite einhergehen, wie ECMA-Script und Applets, werden in diesem Abschnitt vorgestellt. Dadurch wird es möglich, dass Web-Anwendungen für den Benutzer ein ähnliches Eingabeverhalten zeigen, wie dieser es von Standardprogrammen her gewohnt ist.

ECMA-Script

Die von der *European Computer Manufacturers Association* (ECMA) standardisierte Programmiersprache *ECMA-Script* [ECM99] findet sich im Browser unter der Bezeichnung *Java-Script* [Net97a] wieder. Damit ist es möglich, Programmteile in eine HTML-Seite zu integrieren. Der Java-Script-Teil wird dabei durch ein spezielles Element von HTML abgegrenzt.

Da es sich bei Java-Script um eine Skriptsprache handelt, wird das Programm erst zur Laufzeit übersetzt und ausgeführt. Dies geschieht allerdings nicht auf der Server- sondern auf der Client-Seite einer Web-Anwendung, nachdem das Dokument heruntergeladen wurde. Selbst Teile der aktuellen HTML-Seite können damit dynamisch generiert werden, was entweder auf der Ebene von Zeichenketten erfolgt oder durch den Zugriff auf die DOM-Repräsentation des Dokuments geschieht. Ein großer Vorteil von Java-Script ist die mögliche Verlagerung von Rechenschritten vom Server auf den Client-Rechner. Für den Benutzer einer Anwendung stellt die Interpretation von unbekannten Programmen allerdings ein Sicherheitsproblem dar, denn er kann nicht sicher sein, dass nur unschädlicher Code ausgeführt wird. Um dieses Problem zu entschärfen, ist die Ausführung von Java-Script-Programmen nur unter erheblichen Einschränkungen gestattet; beispielsweise ist kein Zugriff auf die lokale Festplatte möglich.

Java Applets

Die Programmiersprache Java [AG98] erhebt den Anspruch, die Programmiersprache für Web-Anwendungen zu sein. Sie ist objektorientiert und besitzt ein striktes Typsystem. Der Java-Übersetzer erzeugt einmalig prozessorunabhängigen Zwischencode, den sogenannten Bytecode, der zur Ausführung von der *Laufzeitumgebung* („Java Virtual Machine“) (JVM) interpretiert wird. Durch die Generierung des Bytecodes, sind Java-Programme auf nahezu allen Rechner-Architekturen lauffähig, also plattformunabhängig. Aus diesem Grund ist die Sprache gut für Anwendungsteile auf der Client-Seite geeignet. Java-Programme, die auf dem Client-Rechner ablaufen, heißen *Java Applets*. Für deren Ausführung wird der Bytecode vom Server auf den Client übertragen und in der dortigen JVM des Browsers interpretiert. Im Vergleich zu Anwendungen in prozessorabhängigem Maschinencode sind Programme als Bytecode in der Ausführung meist langsamer. Durch die Integration von sogenannten *Just-In-Time-Übersetzern* („JIT compiler“) in die JVM wird aber versucht, diesem Nachteil zu begegnen.

Java selbst ist eine vollwertige höhere Programmiersprache, die dem Programmierer einiges bietet. So steht mit der *Java-API* („application programming interface“) [Sun01a] eine sehr um-

fangreiche Klassenbibliothek zur Verfügung. Dadurch ist es unter anderem möglich, auf einfache Weise eine komfortable grafische Benutzeroberfläche zu gestalten. Problemlos möglich ist auch die Anbindung einer Anwendung an ein Datenbanksystem. Dafür existieren die datenbankunabhängig definierten Schnittstellen JDBC („Java Database Connectivity“) [EHF01] und SQLJ („Standard Query Language for Java“) [SBK+99]. Mit diesen kann eine Java-Anwendung durch das Erzeugen entsprechender SQL-Anweisungen Anfragen an eine Datenbank stellen und Daten in der Datenbank manipulieren oder löschen. Damit der Zugriff über JDBC oder SQLJ von Java aus auf ein Datenbanksystem möglich wird, muss dieses einen entsprechenden Treiber bereitstellen, der die Verbindung zwischen Java und der Datenbank herstellt.

2.5.4 Dynamisierung des Webs auf der Server-Seite

In diesem Abschnitt werden verschiedene, existierende technische Ansätze vorgestellt, die es erlauben, HTML-Dokumente oder, allgemeiner, XML-Dokumente dynamisch auf dem Web-Server zu generieren.

Common Gateway-Interface

Eine der ältesten Möglichkeiten für die Erzeugung dynamischer Dokumente durch eine Web-Anwendung wurde durch das *Common Gateway-Interface* (CGI) [CAR98, Gai95, Gun96] bereits 1995 definiert. Diese Schnittstelle erlaubt es, ein beliebiges Programm mit einer festen URL zu verknüpfen. Bei der Anwahl dieser URL durch einen Benutzer, wird vom WWW-Server nicht eine statische Seite an den Browser zurückgeschickt. Stattdessen wird das zugehörige, externe Programm vom WWW-Server zur Ausführung gebracht. Dieses externen Programm ist nun dafür zuständig, eine korrekte Web-Seite zu erzeugen.

Durch CGI wird eine Schnittstelle normiert, die regelt, wie dem externen Programm Daten und Parameter vom Web-Server übergeben und wie vom externen Programm die generierten Web-Seiten zum Web-Server zurückgeliefert werden. Mit CGI können durch ein externes Programm aktuelle Informationen in Form von HTML für Anwender präsentiert werden. Ein manuelles Erstellen von Seiten ist in diesem Fall nicht notwendig und in der Regel nicht möglich, da die dynamisch erzeugten Seiten häufig von aktuellen Parametern und Daten abhängen. Für das Einlesen von Daten zur Übertragung an das externe Programm können beispielsweise Formulare in HTML genutzt werden.

Erfolgt die Implementierung einer CGI-Anwendung in einer Programmiersprache, die ihre ausführbaren Programme in nativen Programmcode übersetzen, so wird das Programm dadurch plattformabhängig, eine Eigenschaft, die man bei Web-Anwendungen meist vermeiden möchte. Um trotz der Realisierung mittels CGI eine Plattformunabhängigkeit zu erreichen, wird die Web-Anwendung häufig in einer Skriptsprache wie Perl [WS92] oder Ruby [Mat01] realisiert. Auch bieten Skriptsprachen im Vergleich zu gängigen Standardprogrammiersprachen eher eine

Unterstützung von CGI durch Bibliotheksfunktionen.

Ein weiterer Nachteil von CGI ist, dass durch das zustandslose Übertragungsprotokoll HTTP eine Kommunikation des Benutzers mit der Web-Anwendung über mehrere Web-Seiten hinweg nur mit erhöhtem Aufwand möglich ist. Man spricht bei HTTP auch von einer gedächtnislosen Verbindung. Die Erweiterung FastCGI [Bro96] ermöglicht solche dauerhaften Verbindungen.

Die externen Programme in CGI werden in einem eigenen Prozess ausgeführt. Das Ablaufen in einem neuen Prozess für jedes externe Programm wird als Nachteil von CGI angesehen, da das Erzeugen eines neuen Prozesses durch das Betriebssystem mit erheblichen Aufwand verbunden ist. Andererseits hat es den Vorteil, dass bei einem Absturz des externen Programms durch einen unberücksichtigten Fehler in der Anwendung nicht auch der Web-Server selbst beendet wird. Dieses Verhalten kann bei Web-Anwendungen auftreten, die nicht durch CGI sondern über eine spezielle Schnittstelle des Web-Servers [Net97b, Apa00] Parameter und Daten austauschen. Diese alternative Implementierungstechnik von Web-Anwendungen wird häufig auch als *Server-API* bezeichnet. Sie wird in dieser Arbeit nicht weiter vertieft; es sei auf die angegebene Literatur verwiesen.

Server-Side Includes

Eine andere Möglichkeit, um HTML-Seiten mit dynamischen Daten zu erzeugen, ist mit den *Server-Side Includes* (SSI) [Gun96, Inf97a, Apa03] gegeben. SSI, die auch als „parsed HTML“ bezeichnet werden, reichern statische HTML-Seiten zur Laufzeit um weitere Daten, die beispielsweise aus einer Datenbank stammen können, an. Dabei haben die statischen HTML-Seiten zusätzlich spezielle Markierungen, die über HTML hinausgehen, um deutlich zu machen, an welchen Stellen welche Änderungen zur Laufzeit vorgenommen werden sollen. Die Anweisungen in den Markierungen der erweiterten Seiten werden vom Web-Server selbst oder von einem speziellen CGI-Programm ausgeführt. Die Ergebnisse werden anschließend in die umgebende HTML-Seite integriert und an den Anwender übermittelt. Die zusätzlichen Markierungen einer für SSI ausgelegten, erweiterten HTML-Seite werden in der Regel von einem Web-Server, der kein SSI unterstützt, als Kommentar behandelt.

Java Servlets

Die *Java Servlets* [Cow01, HC98, Wil99] sind eine Implementierungsmöglichkeit für Web-Anwendungen, bei der die Funktionalität einer Programmiersprache, in diesem Fall Java [AG98], in den Web-Server integriert wurde. Die Laufzeitumgebung von Java läuft dabei permanent im Hintergrund des Web-Servers, der bei Bedarf einen neuen Java-Thread eines Servlets startet. Servlets sind dabei ganz normale Java-Anwendungen. Der Programmierer einer Web-Anwendung erhält mit der Entwicklungsumgebung für Servlets neben der Standard-Java-Bibliothek weitere Bibliotheksfunktionen, die den Komfort erhöhen. So gibt es Methoden zum automatischen Einlesen und Dekodieren von HTTP-Anfragen, zum Lesen und Schreiben von HTML sowie zur Verarbei-

tung von sogenannten *Sitzungen*, also dauerhaften Verbindungen zum Anwender. Dadurch kann eine Web-Anwendung auch eine permanente Datenbankverbindung etablieren. Die Kommunikation eines Servlets mit dem Web-Server erfolgt direkt und der Austausch von Daten zwischen einzelnen Servlets ist ohne Umwege möglich.

Im Vergleich zu CGI bieten Servlets einige Vorteile. Zunächst ist eine als Servlet implementierte Web-Anwendung plattformunabhängig und damit ohne erneute Übersetzung portierbar allein durch die Wahl der Programmiersprache Java. Weiterhin ist ein Servlet effizienter als ein CGI-Programm, weil nicht stets ein neuer Prozess gestartet werden muss. Stattdessen wird bei Bedarf ein weniger aufwendiger Java-Thread aktiviert. Wird eine Web-Anwendung, die mit CGI umgesetzt wurde, mehrfach aufgerufen, so liegt auch der ausführbare Programmtext dieser Anwendung mehrfach im Speicher. Bei einer Servlet-Realisierung, die mehrfach abläuft, wird der Programmtext stattdessen nur einmal in den Speicher geladen. Die Servlet-Engine startet nämlich in diesem Fall mehrere Threads, die alle auf die selbe Java-Klasse im Speicher zurückgreifen. Außerdem besitzt die Servlet-Engine noch Optimierungsmöglichkeiten für die zu ladenden Java-Klassen durch Caching.

JavaServer-Pages

Was SSI für CGI ist, sind die *JavaServer-Pages* (JSP) [PL01, PLC99, FK00] für Java Servlets. JSP bestehen aus statischen XML- oder HTML-Dokumenten, die mit reinen Java-Anweisungen angereichert sind. Diese Anweisungen generieren zur Laufzeit weitere XML- oder HTML-Fragmente, die in die statische Seite an den entsprechenden Stellen eingesetzt werden. Da JSP vom JSP-Präprozessor in Servlets umgesetzt werden, kann man JSP als Servlet-Aufsatz interpretieren. Durch unterschiedliche Typen spezieller Elemente in JSP können die eingebetteten Java-Anweisungen in JSP so voneinander unterschieden werden, dass diese an verschiedenen Stellen in das resultierende Servlet übersetzt werden. Damit wird der Programmaufbau von Servlet-Klassen in JSP reflektiert. Während der Ausführung der Web-Anwendung werden dann die aus den JSP erzeugten Servlets ausgeführt. Liegt für eine Seite noch kein Servlet vor, wird dieses dynamisch generiert und anschließend ausgeführt.

Als großer Vorteil von JSP wird die durch sie mögliche Trennung von Präsentationslayout und Programmlogik angeführt. Dabei soll in einer Web-Anwendung die Präsentation mit Hilfe von JSP realisiert werden, während die Programmführung mit Servlets implementiert wird. Eine Anwendung besteht demnach aus einer Kombination von JSP und Servlets.

2.5.5 Diskussion

Es folgt eine kurze Zusammenfassung der vorgestellten Web-Technologien.

Die Fülle unterschiedlicher Techniken für die Implementierung von Web-Anwendungen, die in diesem Abschnitt dargestellt wurden, lässt erkennen, dass sich mit der fortschreitenden Nutzung

des World-Wide Webs die Anforderungen an das System veränderten. Jede neue Aufgabe brachte dabei ein neue technische Lösung mit sich, die nur selten eine vorherige erweiterte. Mit den statischen Dokumenten eines Hypertextsystems, auf die der Benutzer lediglich lesend zugreifen konnte, begann die Anwendung des WWW. Es folgte schnell die Forderung nach Web-Anwendungen mit dynamischen Web-Seiten und Interaktivität zur Übermittlung von Benutzereingaben. Diese weitreichenden Veränderungen erforderten sowohl eine Erweiterung der Client-Seite (Java-Script, Applets), als auch eine Anpassung auf der Server-Seite (CGI). Da durch die Weiterentwicklung der Server-Seite für jeden Benutzer stets ein neuer Prozess gestartet wird, wurde sie schnell als zu schwerfällig kritisiert. Es kam zur Entwicklung direkter Schnittstellen zum Server, um die Anwendung im Serverprozess selbst ablaufen zu lassen (Server-API). Gleichzeitig wurde eine einfache Unterstützung für weitgehend konstante Web-Dokumente erarbeitet (SSI).

Doch die vollständige Integration der Programmteile einer Web-Anwendung in den Server birgt den Nachteil, dass bei fehlerhafter Implementierung der gesamte Web-Server zusammenbrechen kann. Weitere Nachteile, wie Abhängigkeit von bestimmten Serverimplementierungen und von bestimmten Rechnerarchitekturen, auf denen der Server mit der Anwendung ausgeführt wird, konnten mit dem Einsatz der Programmiersprache Java für Web-Anwendungen (Servlets) überwunden werden. Java bietet zusätzlich den Vorteil, dass das Programm nicht in einem separaten, aufwendigen Prozess ablaufen muss, sondern in einem eigenständigen, weniger aufwendigen *Thread* ausgeführt wird, der dadurch die Stabilität des Web-Servers nicht gefährdet. Selbst die einfache Einbindung weitgehend konstanter Web-Seiten ist vorgesehen (JSP).

Die folgende Tabelle fasst die Eigenschaften in den Spalten für alle vorgestellten Technologien, die in den Zeilen aufgeführt sind, zusammen. Trifft für eine Technik eine Eigenschaft zu, so wird

	programmiersprachen-unabhängig	eigenständiger **P**rozess/**T**hread	server-unabhängig	portier-bar	plattform-unabhängig
CGI	+	P	+	+/−	+/−
Server-API	−	−	−	−	−
SSI	−	+/−	−	−	−
Servlets	−	T	+	+	+
JSP	−	T	+	+	+

der Eintrag mit einem + markiert, ansonsten wird ein − angegeben. Beispielsweise ist CGI unabhängig von einer konkreten Programmiersprache definiert (+), läuft in einem eigenen Prozess (P) und ist unabhängig von einer konkreten Serverimplementierung (+). Die Portierbarkeit und die Plattformunabhängigkeit[4] hängen bei CGI aber von der gewählten Programmiersprache ab (+/−). Für Java wird zusätzlich der Ablauf im eigenen Thread (T) angezeigt.

Durch die inzwischen weitverbreitete kommerzielle Nutzung des WWW ist erkannt worden, dass eine Standardisierung von Auszeichnungssprachen, Protokollen und Programmierschnitt-

[4]Der wesentliche Unterschied zwischen den Eigenschaften Portierbarkeit und Plattformunabhängigkeit liegt darin, dass der Quelltext portierbarer Programme für unterschiedliche Rechnerarchitekturen angepasst und neu übersetzt werden darf. Dies ist bei plattformunabhängigen Programmen nicht erlaubt.

stellen notwendig ist. Im Rahmen des W3C, dem Zusammenschluss der an einer Standardisierung interessierten Firmen und Organisationen, wird dies durchgeführt und vorangetrieben. Das Ziel ist ein weiteres Divergieren der Darstellungsmöglichkeiten von Informationen im Web sowie von deren Übertragungsarten zu unterbinden.

2.6 Verarbeitung und Repräsentation von XML

Der letzte Abschnitt präsentierte Technologien zur Implementierung von Web-Anwendungen. Dieser Abschnitt konzentriert sich auf technische Möglichkeiten zur Verarbeitung und Repräsentation von XML-Dokumenten und XML-Fragmenten in Programmiersprachen. Dies ist eine für Web-Anwendungen wichtige Fähigkeit, da diese die übermittelten Daten auf dem Server verarbeiten und mit zur Laufzeit generierten XML-Fragmenten beantworten. Es werden verschiedene, existierende technische Ansätze kurz vorgestellt, die es erlauben, HTML-Dokumente oder, allgemeiner, XML-Dokumente zu verarbeiten. Primär unterscheiden sie sich in der Repräsentation der XML-Fragmente.

Der Abschnitt gliedert sich in vier Teile. Als erstes wird auf die Verarbeitung von XML auf der Basis von Zeichenketten eingegangen. Anschließend werden Methoden, die erst einfache, später höhere Objektmodelle verwenden, beschrieben und abschließend erfolgt die Darstellung von Ansätzen, die die statische Gültigkeit zur Zeit der Programmübersetzung garantieren.

2.6.1 Verarbeitung von XML als Zeichenkette

Der einfachste Weg, um XML in einem Programm zu verarbeiten, besteht in der Verwendung der von der Programmiersprache zur Verfügung gestellten Datentypen für Zeichenketten und den darauf aufbauenden Operationen. XML-Fragmente werden dann wie gewöhnliche Zeichenketten ohne jegliche Struktur behandelt. Mit den verbreiteten Web-Technologien CGI, SSI, Servlets und JSP ist eine solche Verarbeitung möglich und gängige Praxis.

Generell kann man sagen, dass die Verarbeitung von XML auf der Basis von Zeichenketten einen wesentlichen Nachteil mit sich bringt. Es kann zur Zeit der Programmübersetzung weder sichergestellt werden, ob die generierten XML-Fragmente wohlgeformt sind, noch ist eine Überprüfung der Gültigkeit möglich. Die sowohl in CGI als auch in Servlets fehlende Unterstützung für größere, konstante XML-Fragmente, die eine umständliche Generierung notwendig macht, ist durch SSI und JSP aufgehoben worden.

2.6.2 Einfache Objektmodelle

Eine Verbesserung gegenüber der Verarbeitung von XML-Fragmenten auf der Ebene von Zeichenketten liegt in der Definition eines *einfachen Objektmodells* mit Klassen für die Knoten eines XML-Fragments. Dadurch werden beliebige XML-Dokumente durch eine objektorientierte Datenstruktur repräsentiert, die ausgelesen und verändert werden kann. Der wichtigste Vertreter dieses Ansatzes ist das DOM. Daneben existiert für die Programmiersprache Java ein unabhängiges Objektmodell für XML mit dem Namen *Java*-DOM (JDOM) [JDO], das besser auf die Eigenheiten der Sprache abgestimmt ist. Diese Arbeit geht auf JDOM nicht näher ein, sondern verweist auf die detaillierte Darstellung in [Har02].

Dokument-Objektmodell

Das *Dokument-Objektmodell* (DOM) wurde bereits in Abschnitt 2.3 ausführlich vorgestellt. Der Ansatz ist weit verbreitet und wird von vielen Programmiersprachen unterstützt. Es ist der bisher einzige standardisierte und sprachneutrale Weg zur Verarbeitung von XML.

Konstante XML-Fragmente müssen in einfachen Dokumentmodellen entweder in streng objektorientierter Weise ausprogrammiert werden, was sehr mühsam ist, oder durch das Einlesen des XML-Fragments als Zeichenkette erzeugt werden, was eine Überprüfung der Gültigkeit zur Laufzeit erforderlich macht. Die Eigenschaft Wohlgeformtheit wird im Vergleich zur Verarbeitung als Zeichenketten jedoch für die dynamisch generierten XML-Fragmente zur Zeit der Programmübersetzung garantiert. Die Überprüfung der Gültigkeit dagegen ist ebenfalls erst zur Laufzeit möglich.

2.6.3 Höhere Objektmodelle

Neben den einfachen Objektmodellen können XML-Fragmente auch durch *höhere Objektmodelle* repräsentiert werden. Deren Verwendung setzt voraus, dass eine Anwendung sich auf die Verarbeitung von Dokumenten einer oder mehrere fester Auszeichnungssprachen beschränkt. Diese Annahme stellt zwar im Allgemeinen eine Einschränkung dar, wird aber von den meisten Web-Anwendungen eingehalten. Die Idee dieses Ansatzes ist, aus der Sprachbeschreibung der verwendeten Auszeichnungssprache Datentypen oder Klassen der Programmiersprache zu generieren. Dabei werden die Datentypen und Klassen derart definiert, dass sie die durch die Sprachbeschreibung intendierte Semantik der Dokumentstruktur so gut wie möglich in der Programmiersprache reproduzieren.

Eine ganze Reihe von Vorschlägen [Bou02] für höhere Objektmodelle, an denen sich nahezu jeder namhafte Softwarehersteller, wie Sun mit JAXB, Microsoft mit .Net Framework, Exolab mit Castor, Delphi mit Data Binding Wizard und Oracle mit XML Class Generator [Sun03, Mic01, Exo02, Bor01, Ora01] beteiligt, sind in jüngster Zeit vorgelegt worden. In der Regel sind die-

se Ansätze auf spezielle Programmiersprachen zugeschnitten. Eine Standardisierung ist zur Zeit nicht vorgesehen. Diese Arbeit beschränkt sich auf eine kurze Darstellung von JAXB und dem Validating-DOM, einer Entwicklung aus früheren Forschungsarbeiten.

Java Architecture for XML Binding

Für die Programmierung von Web-Anwendungen und Web-Services wird von der Firma Sun das Entwicklungswerkzeug *Java Architecture for* XML *Binding* (JAXB) [Sun03] zur Verfügung gestellt. Dieses verfügt über einen *Schema-Übersetzer*, den der Programmierer mit einer Sprachbeschreibung, sei es eine DTD oder ein XML-Schema, startet. Als Ergebnis erhält er den Quellcode von Typen- und Klassendefinitionen für Elementdeklarationen und Typdefinitionen der Sprachbeschreibung. Jede Klasse definiert eigene Instanzvariablen, in denen die Inhalte und Attribute der repräsentierten XML-Fragmente abgelegt werden. Spezielle Zugriffsmethoden erlauben das Auslesen und Ändern dieser Daten. Weiterhin werden sogenannte `unmarshal`-Methoden erzeugt, die ein einfaches Einlesen von XML in die generierten Klassenstrukturen ermöglichen. Umgekehrt können aber auch über Methoden namens `marshal` aus den Instanzen der internen Darstellung die repräsentierten XML-Fragmente erstellt werden. Mit der ebenfalls generierten Methode `validate` kann ein Test auf Gültigkeit der aktuellen Objekte zur Sprachbeschreibung durchgeführt werden. Dem Entwickler steht es nun frei, die generierten Klassen um anwendungsspezifische Methoden zu erweitern und sie in den Programmen der Anwendung einzusetzen. Auch kann er die Voreinstellung des Schema-Übersetzers durch sogenannte *Bindungsschemata* („binding schema“) überschreiben. Damit ist eine Veränderung der generierten Klassen-, Variablen- und Methodennamen sowie ein Einfluss auf erzeugte Typkonvertierungen, Klassenkonstruktoren, typsichere Aufzählungsklassen und Schnittstellen möglich.

In JAXB erfolgt eine Überprüfung auf Gültigkeit in den `marshal`- und `unmarshal`-Methoden und kann zusätzlich während der Laufzeit durch Aufruf einer `validate`-Methode angestoßen werden. Zusätzlich wird durch die speziellen Zugriffsmethoden der generierten Klassen und das Java-Typsystem schon eine Vielzahl von möglicherweise ungültigen Objektzuständen ausgeschlossen. Aber gerade bei nicht trivialen, verschachtelten Inhaltsmodellen zeigt JAXB Schwächen, denn es ergeben sich entweder sehr komplizierte Datenstrukturen oder welche, die das Inhaltsmodell nur sehr ungenügend reflektiert. Die statische Gültigkeit während der gesamten Lebenszyklen der Objekte ist damit in JAXB nicht garantiert. Ein weiterer Nachteil ist darin zu sehen, dass eine etwaige Änderung der Sprachbeschreibung eine erneute Generierung der Klassen nach sich zieht. Auch wird für den Programmierer die Komplexität der Softwareentwicklung künstlich erhöht. Denn er muss sich neben der Sprachbeschreibung zusätzlich in das Bindungsschema einarbeiten, welches für nicht triviale Fälle schnell kompliziert wird.

Validating-DOM

Validating-DOM (VDOM) [KL02] ist eine Erweiterung des DOM zum höheren Objektmodell.

Ähnlich wie bei JAXB wird für jede Elementdefinition in der Sprachbeschreibung ein neue Schnittstellendefinition generiert. Diese Schnittstellen, die Erweiterungen der DOM-Schnittstelle `Element` sind, enthalten spezielle Methoden zum Einfügen und Entfernen von Attributen und Inhalten. Die Deklaration der Methodensignaturen ist dabei in einer solchen Weise konstruiert, dass das Inhaltsmodell der Elementtypen durch das Typsystem weitgehend sichergestellt werden kann. Zusätzlich werden nur zulässige Attribute und Attributwerte akzeptiert. In VDOM werden konstante XML-Fragmente durch ein neues Sprachkonstrukt mit dem Namen *Parameterized-*XML (PXML) unterstützt.

Höhere Objektmodelle verfügen wie einfache Objektmodelle, mit Ausnahme des VDOM, über keine Erleichterung zur Verarbeitung von konstanten XML-Fragmenten. Aus diesem Grund muss die Generierung von konstanten XML-Fragmenten entweder durch verschachtelte Konstruktor- und Methodenaufrufe oder durch das Einlesen einer festen Zeichenkette erfolgen. Das erste Vorgehen ist für den Programmierer sehr mühsam, während das zweite einer Überprüfung der Gültigkeit zur Laufzeit bedarf. Die Wohlgeformtheit von dynamisch generierten Dokumenten zur Zeit der Programmübersetzung wird durch höhere Objektmodelle sichergestellt. Die Gültigkeit für diese Dokumente kann meist nur eingeschränkt garantiert werden. Ausschlaggebend für die Qualität dieser Überprüfung ist die Wahl des Objektmodells und der verwendeten Programmiersprache.

2.6.4 Garantie der statischen Gültigkeit

In diesem Abschnitt werden Ansätze zur Repräsentation von XML vorgestellt, die es erlauben, bereits zur Zeit der Programmübersetzung zu garantieren, dass die von der Web-Anwendung erzeugten Dokumente statisch gültig sind. Nur diese Entwicklungen sind wirklich vergleichbar mit der Spracherweiterung, die diese Arbeit vorstellt.

XDuce

Die Sprache XDuce [HVP00] ist eine funktionale Programmiersprache, die speziell zur Verarbeitung von XML entwickelt wurde. Sie definiert sogenannte *reguläre Ausdruckstypen*, deren Werte XML-Fragmente repräsentieren. Diese Werte werden durch spezielle Konstruktoren erzeugt und können, wie in funktionalen Programmiersprachen verbreitet, durch *Pattern-Matching* analysiert werden. XDuce unterstützt Typinferenz für Pattern und Variablen, es führt eine Subtyp-Analyse auf der Basis von Baumautomaten [RS97] durch, um die Gültigkeit der Instanzen von regulären Ausdruckstypen bereits zum Zeitpunkt der Programmübersetzung sicherzustellen.

BigWig

BigWig [BMS01] ist eine iterative Programmiersprache zur Entwicklung von interaktiven Web-Services. Sie führt getypte XML-*Dokument-Schablonen* („templates“) ein, die ausgezeichnete Lücken enthalten können. Um im Programm dynamisch XML-Dokumente zu erzeugen, besteht die Möglichkeit, diese Lücken zur Laufzeit mit anderen Schablonen oder Zeichenketten zu substituieren. Für sämtliche Schablonen überprüft BigWig die dynamisch berechneten Dokumente auf Gültigkeit bzgl. einer gegebenen DTD bereits zum Zeitpunkt der Programmübersetzung. Dies geschieht durch zwei Datenflussanalysen [NNH99], die einen Graphen konstruieren, der sämtliche mögliche Dokumente endlich darstellt. Der Graph wird dann analysiert, um Schablonen zu erkennen, die gegen die Gültigkeit verstoßen. Der BigWig-Quelltext wird in eine Kombination von Programmen unterschiedlicher Standard-Web-Technologien wie HTML, CGI, Applets und JavaScript übersetzt. Seit kurzer Zeit ist BigWig auch für die Programmiersprache Java unter dem Namen JWig [CMS02] verfügbar.

XL und XQuery

Einen weiteren anspruchsvollen Ansatz präsentiert die Sprachspezifikation von XL [FGK02]. XL ist eine eigenständige XML-Programmiersprache speziell zur Implementierung von Web-Services. Sie übernimmt die Sprachkonstrukte aus XQuery [W3C02c] und erweitert diese um höhere und deklarative Sprachkonstrukte zu einer vollständigen Programmiersprache. Zusätzlich werden imperative Sprachkonstrukte wie Fallunterscheidungen, Schleife und Ausnahmen integriert, wodurch XL zu einer Mischung aus funktionaler und imperativer Programmiersprache wird. Wie im zukünftigen Standard für Anfragesprachen von XML-Datenbanksystemen XQuery [W3C02c] soll XL, das auf XQuery beruht, die statische Gültigkeit für dynamisch erzeugte XML-Fragmente unterstützen.

2.6.5 Diskussion

In diesem Abschnitt werden die dargestellten Ansätze zur Verarbeitung von XML-Fragmenten zusammengefasst.

Die folgende Tabelle stellt in ihren Spalten dar, in wie fern die unterschiedlichen Verarbeitungsformen von XML, die in den Zeilen aufgeführt sind, konstante XML-Fragmente unterstützen, sowie die Eigenschaften Wohlgeformtheit und statische Gültigkeit bereits zum Zeitpunkt der Programmübersetzung garantieren. Durch die Angabe eines $+$ wird illustriert, dass der Ansatz über die entsprechende Eigenschaft verfügt. Ist dies nicht der Fall, wird ein $-$ aufgeführt. Bei der Eigenschaft der statischen Gültigkeit wird für Techniken, die diese nicht im vollen Umfang sicherstellen, ein $_+$ angezeigt. Zum Beispiel ist eine einfache Einbindung von konstanten XML-Fragmenten in JAXB nicht vorgesehen ($-$), die Eigenschaft der Wohlgeformtheit wird dagegen garantiert ($+$), während die statische Gültigkeit nur begrenzt zugesichert werden kann ($_+$).

	konstante XML-Fragmente	Übersetzungsgarantien	
		Wohlgeformtheit	statische Gültigkeit
Zeichenkette	−	−	−
SSI, JSP	+	−	−
DOM, JDOM	−	+	−
JAXB, CASTOR	−	+	+
VDOM	+	+	+
XDuce, BigWig, XL, XOBE	+	+	+

Die Hauptbeobachtung aus der Tabelle ist, dass nur XDuce, BigWig und XL wirklich vergleichbar sind mit dem Vorschlag, den diese Arbeit mit XML-*Objekte* (XOBE) vorstellt. Die anderen Ansätze verwenden zur Repräsentation von XML-Fragmenten entweder nur Zeichenketten oder trennen streng zwischen einer Repräsentation als Zeichenkette und einer Repräsentation als Objekt. Im restlichen Abschnitt werden kurz die wesentlichen Unterschiede zwischen XDuce, BigWig und XL zu dem in den nächsten Kapiteln eingeführten XOBE beschrieben.

Im Vergleich zu XOBE implementiert XDuce ebenfalls einen Subtyp-Algorithmus, der aber auf regulären Baumautomaten basiert. Er operiert deshalb auf einer zusätzlichen internen Repräsentation für die regulären Ausdruckstypen, was durch die notwendigen Konvertierungen eine Ineffizienz des Verfahrens darstellt. Auf diese interne Repräsentation verzichtet der Algorithmus dieser Arbeit, der in Abschnitt 4.6 vorgestellt wird. Weiterhin ist es mit XOBE, durch Erweiterung der Programmiersprache Java, einfacher, ein Programm an andere Komponenten wie beispielsweise Datenbanksysteme anzubinden.

Verglichen mit XOBE können die in BigWig eingeführten Schablonen als Methoden begriffen werden, die XML-Objekte zurückliefern. Die Argumente dieser Methoden korrespondieren dann mit den Lücken der Schablonen. Damit ist dieses Sprachmittel in XOBE voll darstellbar. Der Algorithmus zur Typanalyse in BigWig basiert auf einer Datenflussanalyse und ist deshalb nur schwer mit dem Algorithmus in XOBE vergleichbar. Weiterhin scheint das Typsystem dieser Arbeit verglichen mit BigWig ausdrucksstärker zu sein, weil es möglich ist, den Subtyp-Algorithmus sehr natürlich um Typerweiterungen und Typeinschränkungen aus XML-Schema zu erweitern (Abschnitt 4.8). Dies ist in BigWig wohl nur schwer erreichbar.

XL ist definiert als eigenständige Programmiersprache für Web-Services. Im Unterschied dazu ist XOBE eine Erweiterung von Java, einer bereits weit verbreiteten und bewährten Programmiersprache für Web-Anwendungen und Web-Services. Es ist naheliegend, dass XOBE mit der Verwendung von bereits entwickelten Bibliotheken und Programmen in Java erheblich profitieren kann.

2.7 Einordnung dieser Arbeit

In diesem Abschnitt werden zunächst die Eigenschaften der zuvor beschriebenen Techniken zur Verarbeitung und Repräsentation von XML, zusammengefasst, bevor wiederholt die Zielsetzung dieser Arbeit klargestellt wird. Die Verarbeitung und Repräsentation von XML wird verstärkt bei Web-Anwendungen (2.5) eingesetzt, für die die unterschiedlichsten Implementierungstechniken existieren.

Die Verarbeitung von XML mit den Mitteln von Zeichenketten, wie dies bei SSI über CGI und JSP auf der Grundlagen von Servlets erfolgt, ist völlig unzureichend. Zwar werden konstante XML-Fragmente relativ komfortabel unterstützt, doch wird für generierte XML-Fragmente weder die Eigenschaft wohldefiniert noch die statische Gültigkeit zur Zeit der Programmübersetzung garantiert. Selbst eine Überprüfung der Eigenschaften zur Laufzeit der Web-Anwendung ist nicht vorgesehen.

Einfache Objektmodelle sind zur Zeit der einzige standardisierte Weg zur Verarbeitung von XML-Fragmenten. Sie sind so allgemein definiert, dass eine Verarbeitung von XML-Dokumenten beliebiger Auszeichnungssprachen, deren Sprachbeschreibung der Anwendung nicht einmal bekannt sein muss, durchgeführt werden kann. Die Einbindung konstanter XML-Fragmente ist nicht vorgesehen. Die Eigenschaft der Wohldefiniertheit wird zur Zeit der Programmübersetzung für alle generierten XML-Fragmente garantiert, während die statische Gültigkeit bei gegebener Sprachbeschreibung für den selben Zeitpunkt nicht sichergestellt werden kann. Allerdings wird einer Überprüfung zur Laufzeit, die zusätzliche Rechenzeit in Anspruch nimmt, unterstützt.

Bei der Verarbeitung von XML-Dokumenten mit höheren Objektmodellen müssen alle verwendeten Auszeichnungssprachen zum Zeitpunkt der Programmierung feststehen. Mit dieser Annahme kann sich die interne Repräsentation von XML eng an den Bedingungen der Sprachbeschreibung orientieren, wodurch die Überprüfung auf Gültigkeit erleichtert wird. Höhere Objektmodelle stellen die Wohldefiniertheit während der Programmübersetzung sicher, die statische Gültigkeit wird nur mit Einschränkung garantiert. Die Überprüfung der gesamten statischen Gültigkeit ist zur Laufzeit der Anwendung möglich. Bisher ist keine Unterstützung von konstanten XML-Fragmenten integriert.

Ansätze, die die statische Gültigkeit bei der Verarbeitung von XML-Fragmenten zur Zeit der Programmübersetzung sicherstellen, gibt es nur vereinzelt. Eine Entwicklung realisiert eine funktionale Programmiersprache mit eingeschränktem Sprachumfang, weshalb ein Verwendung für Web-Anwendungen und Web-Services nur begrenzt möglich ist. Desweiteren ist in der in Standardisierung befindlichen Anfragesprache für XML-Datenbanksysteme eine Überprüfung auf statische Gültigkeit vorgesehen. Darauf aufbauend wurde eine Programmiersprache für Web-Services definiert, die sowohl über deklarative als auch iterative Sprachkonstrukte verfügt. Für letztere Sprachen ist eine erste Implementierung zur Zeit noch in der Entwicklungsphase. Verfügbar dagegen ist eine Java-Erweiterung, die ebenfalls die statische Gültigkeit garantiert, aber Einschränkungen hinsichtlich der erweiterten Beschreibungsmöglichkeiten von XML-Schema aufweist.

Insgesamt ist folgende Schlussfolgerung zu ziehen: Es ist – bis auf eine Ausnahme – bisher *nicht* möglich, mit einer objektorientierten Programmiertechnik Web-Anwendungen zu erstellen, die eine Verarbeitung von XML-Fragmenten unter Garantie der statischen Gültigkeit vorsieht. (Lediglich die parallel zu dieser Arbeit entstandene Java-Erweiterung JWig geht einen Schritt in diese Richtung.) Stattdessen muss mit Hilfe von intensiven Testläufen die Korrektheit der erzeugten XML-Fragmente plausibel gemacht werden. Dieser Nachteil besteht bei einer ganzen Reihe der beschriebenen und weit verbreiteten Technologien zur Verarbeitung und Repräsentation von XML. Andere Entwicklungen können zwar die statische Gültigkeit in sehr begrenztem Maße sicherstellen, verlangen aber vom Programmierer neben der Kenntnis der eingesetzten Auszeichnungssprache, noch eine Abbildung der Sprachbeschreibung in Datentypen der Programmiersprache.

Mit dieser Arbeit soll durch eine Erweiterung der Programmiersprache Java eine objektorientierte Integration von XML-Fragmenten erreicht werden. Die Erweiterung soll dabei für die verarbeiteten XML-Fragmente die statische Gültigkeit bereits zur Zeit der Programmübersetzung sicherstellen, wodurch auf intensive Testläufe und aufwendige Überprüfungen zur Laufzeit verzichtet werden kann. Einzellösungen zur komfortablen Formulierung statischer XML-Teile, wie sie mit SSI und JSP in 2.5.4 ausdrückbar sind, können mit dieser Java-Erweiterung ebenfalls realisiert werden.

Die Grundlagen der Java-Erweiterung bildet die Deklaration einer XML-Sprachbeschreibung, sei es eine DTD oder ein XML-Schema, im Programm, wodurch die Sprachbeschreibung zum Zeitpunkt der Programmübersetzung feststeht. Aufbauend auf der Sprachbeschreibung werden neue Sprachkonstrukte eingeführt, um XML-Fragmente zu erzeugen und im Programm zu verarbeiten. Auch ist die Selektion von Inhalten und Elementen aus XML-Fragmenten über XPath (2.2) möglich. Da jedem XML-Fragment ein eindeutiger Typ aus der Sprachbeschreibung zugeordnet ist, kann anhand einer Typanalyse zum Zeitpunkt der Programmübersetzung überprüft werden, ob das Programm ausschließlich statisch gültige XML-Fragmente verarbeitet. Eine prototypische Implementierung der Java-Erweiterung wurde als Präprozessor realisiert und transformiert die XOBE-Programme in reinen Java-Quelltext. Als interne Repräsentation der XML-Fragmente kommt dabei das DOM (2.3) zur Anwendung.

Kapitel 3

XML-Objekte

Bei der Programmierung von Anwendungen mit JSP, wie im letzten Kapitel demonstriert, hat sich gezeigt, dass eine Integration von XML-Syntax in eine Programmiersprache für Web-Anwendungen wie Java sinnvoll ist. Trotzdem ist der Ansatz der JSP, wie im Abschnitt 2.5 ausführlich diskutiert, nicht ausreichend. Die Unzulänglichkeit liegt einerseits in der fehlenden statischen Überprüfung der XML-Syntax hinlänglich der Eigenschaften Wohlgeformtheit und Gültigkeit und andererseits im unklaren Objektmodell, das hinter den XML-Konstrukten steht.

Die unstrittigen Vorteile von XML-Syntax in der Programmiersprache Java werden durch die Java-Erweiterung **X**ML-***Obje**kte* (XOBE) aufgegriffen. XOBE führt weiterhin ein klares Objektmodell für XML-Dokumente mit Elementen, Attributen und Zeichendaten ein, die sogenannten XML-*Objekte*. XML-Objekte werden mit Hilfe von XML-Konstruktoren, die in XML-Syntax notiert werden, erzeugt. Die Selektion von Werten und Inhalten aus XML-Objekten erfolgt in der standardisierten und in Abschnitt 2.2 vorgestellten Sprache XPath, um die die Programmiersprache Java ebenfalls erweitert wird.

Das Kapitel beginnt mit einer informellen Einführung über die neuartigen Sprachkonstrukte in XOBE. Im Anschluss daran erfolgt eine formale Definition der Syntax mit einer Beschreibung der implizierten Semantik. Es folgt ein Abschnitt, der die Anwendung von XOBE mit Beispielen demonstriert.

3.1 Einführung

Die Spracherweiterung XOBE ergänzt die Programmiersprache Java um XML-*Objekte* [LK02]. Mit XML-Objekten werden sowohl die baumartige Struktur eines XML-Fragments als auch die darin enthaltenen Informationen und Daten repräsentiert. Sie sind als eingebaute Datenobjekte konstruiert, so dass sie wie Werte von Basisdatentypen im Programm verwendet werden können. Die Struktur von XML-Objekten muss einer vorgegebenen Sprachbeschreibung entsprechen, in-

dem diese vorab deklariert wird. Damit werden die Definitionen und Deklarationen der deklarierten Sprachbeschreibung als implizite Definitionen von XML-*Objektklassen* aufgefasst und zur Typisierung der unterschiedlichen XML-Objekte eingesetzt. Die korrekte Typisierung wird zum Zeitpunkt der Programmübersetzung überprüft.

Vergleichbar mit Zeichenketten vom Typ `String`, die durch konstante Zeichenketten im Programm erzeugt werden können, steht als Konstruktor für XML-Objekte deren XML-Syntax zur Verfügung. Mit dem folgenden kleinen Beispiel soll die Idee veranschaulicht werden.

Beispiel 3.1
In diesem Beispiel wird die Auszeichnungsprache AOML aus Beispiel 2.2 verwendet.

```
1  author a = <author>Thomas Mann</author>;
```

Die Zeile zeigt die Zuweisung eines XML-Elements an die Variable `a`. Die Variable ist deklariert als Variable der XML-Objektklasse `author`.

```
2  int eu = 8;
3  price p = <price currency="EUR">{eu}</price>;
```

In der letzten Zeile wird der Variablen `p` der XML-Objektklasse `price` ein Element zugewiesen. Das Element hat keinen Inhalt. Stattdessen wird dort der Wert der `int`-Variablen `eu` eingesetzt.

```
4  book b = <book catalog="Varia">
5              <title>Lotte in Weimar</title>
6              {a}
7              <condition>Einband fingerfleckig, Rücken
                  verblaßt</condition>
8              {p}
9          </book>;
```

Hier wird der Variablen `b` der XML-Objektklasse `book` das Fragment eines XML-Dokuments zugewiesen. Diesmal werden die XML-Objekte auf die die Variablen `a` und `p` verweisen, die zuvor als XML-Objektklassen deklariert wurden, in den Inhalt eingefügt (6,8).

```
10 String eu2 = (b/child::price/child::text()).item(0);
```

Die letzte Zeile des Beispiels selektiert aus dem durch die Variable `b` referenzierten XML-Objekt den Inhalt des Kindes `price`. □

3.2 Syntax und Semantik

In diesem Abschnitt folgt nach obiger informeller Darstellung die Definition der neuen Sprachkonstrukte in XOBE. Dabei werden die neuen Konstrukte in die Java-Grammatik integriert. Auf

eine Vorstellung der gesamten Java-Syntax wird an dieser Stelle verzichtet; sie findet sich in [GJS96, GJSB00].

3.2.1 Objektmodell

Neben der konkreten Syntax eines XML-Dokuments oder -Fragments ist für XOBE die abstrakte, logische Struktur von XML wesentlich. Die Struktur eines XML-Dokuments ähnelt dabei einem Baum, während die Struktur eines XML-Fragments, das aus mehreren Elementen bestehen kann, einer Reihe von Bäumen, einer sogenannten *Hecke*, gleicht. Ein XML-Dokument kann damit auch als Spezialfall eines XML-Fragments angesehen werden, weshalb hier im Weiteren nur noch von XML-Fragmenten gesprochen wird.

In XOBE wird die Struktur und der eigentliche Dateninhalt von XML-Fragmenten durch Instanzen eines *Objektmodells* repräsentiert. Dieses Objektmodell ist nicht nur eine Datenstruktur für XML-Fragmente, sondern ein Objektmodell im Sinne des traditionellen objektorientierten Designs [RBP+91]. Es umfasst damit nicht nur die Struktur und Daten der XML-Fragmente, sondern schließt auch Identität und Verhalten der repräsentierenden Objekte ein, die als XML-Objekte bezeichnet werden. Für die einzelnen XML-Objekte der baumartigen Objektstruktur wird zwischen den folgenden Objektklassen unterschieden:

- Elementklassen mit Superklasse `Element`,
- Attributklassen mit Superklasse `Attribute`,
- Kommentarklasse `Comment`.

Das wichtigste Strukturierungsmittel in XML sind Elementtypen, die in der Sprachbeschreibung deklariert wurden. Elemente unterscheiden sich durch verschiedene Elementnamen, Attributtypen und Inhaltsmodelle. Anhand dieser Eigenschaften erfolgt in XOBE eine Unterteilung der Elementobjekte in *spezielle Elementklassen*, die von der allgemeinen Elementklasse `Element` abgeleitet werden. Durch die Definitionen und Deklarationen der verwendeten Sprachbeschreibung werden Spezialisierungen der allgemeinen Superklasse erreicht. Für jedes Element im zu repräsentierenden XML-Fragment existiert eine Instanz einer solchen speziellen Elementklasse in der Objektstruktur. Jedes Elementobjekt referenziert seine Attributobjekte sowie die im Inhalt befindlichen Element-, Text- und Kommentarobjekte, die auch als *Kinder* bezeichnet werden. Weiterhin findet sich ein Verweis auf das *Elternobjekt* eines Elementobjekts.

Jedes Elementobjekt verweist auf eine zugeordnete Menge von Attributobjekten. Jedes Attributobjekt verweist auf seinen Attributnamen und den Attributwert. Anders als im Datenmodell von XPath [W3C99a] wird in XOBE nicht verlangt, dass ein Attribut auf sein Elementobjekt verweist. Attributobjekte werden analog zu den Elementobjekten in unterschiedliche *spezielle Attributklassen* unterteilt, die sich von der allgemeinen Attributklasse `Attribute` ableiten.

Die in der Sprachbeschreibung deklarierten Eigenschaften Attributname und Attributtyp definieren die verschiedenen Spezialisierungen der allgemeinen Superklasse. Attributtypen, für die in der Sprachbeschreibung ein Vorgabewert vorliegt, werden in der Objektstruktur so behandelt, als wären sie im Element aufgeführt.

Im Inhalt von Elementen können in XML Zeichendaten auftreten, die in XOBE durch Instanzen der Klasse `String` aus der Java-Bibliothek repräsentiert werden. Zeichenketten, die Zeichendaten repräsentieren, müssen immer aus mindestens einem Zeichen bestehen. Andernfalls werden sie aus der Objektstruktur entfernt. Zeichenketten gehen in der Objektstruktur niemals einer anderen Zeichenkette als direkter Geschwisterknoten voraus oder folgen unmittelbar einer solchen. Für jeden Kommentar im repräsentierten XML-Fragment existiert ein Kommentarobjekt der *Kommentarklasse* `Comment` in der Objektstruktur. Diese verweisen auf den Kommentartext, der ebenfalls in Form einer Zeichenkette vorliegt.

Im XOBE-Objektmodell wird auf eine spezielle Klasse zur Repräsentation ganzer XML-Fragmente verzichtet. Instanzen einer solchen Klasse, wie sie im DOM oder im Datenmodell von XPath definiert sind, dienen lediglich dazu, den Einstiegspunkt oder die Wurzel der Objektstruktur zu markieren und verweist auf die Zeichenketten, Element- und Kommentarobjekte, die für den Inhalt des zu repräsentierenden XML-Fragments auf der äußersten Ebene stehen. Ein Auftreten solcher Objekte innerhalb des verschachtelten Baumes ist dort nicht erlaubt.

Wie in XPath (siehe Abschnitt 2.2) wird auch in XOBE eine Ordnung für die Objekte der Objektstruktur definiert. Mit *Dokumentordnung* wird dabei die Reihenfolge aller Objekte einer Objektstruktur bezeichnet, die mit der Reihenfolge des Auftreten des ersten Zeichens der XML-Repräsentation eines jeden Objekts im Dokument korrespondiert. Das Dokumentobjekt ist stets das erste Objekt in Dokumentordnung, während Elternobjekte vor ihren Kindobjekten liegen. Dadurch sind Elementobjekte anhand des Auftretens ihrer Start-Tags in XML angeordnet. Die Attributobjekte eines Elements liegen vor den Kinderobjekten des Inhalts des Elements. Die *umgekehrte Dokumentordnung* ist definiert als die Umkehrung der Dokumentordnung.

3.2.2 Klassen

Die Bestandteile von Daten oder Dokumenten in XML, die Elemente und Attribute, haben im Allgemeinen unterschiedliche Elementtypen und Attributtypen, wie in 2.1.1 beschrieben wurde. Diese Typen werden in einer Sprachbeschreibung deklariert und definiert. In XOBE werden aus der Sprachbeschreibung die Deklarationen der Elementnamen mit der angegebenen Inhaltsdefinition und die Definitionen benannter Gruppen sowie komplexer Typen berücksichtigt. Dafür muss die Sprachbeschreibung im XOBE-Programm explizit deklariert werden.

Definition 3.1 (Schemadeklaration)
Eine *Schemadeklaration* in XOBE erfolgt über die folgende Syntax:
<*ImportDeclaration*> → ... | "`ximport`" <*URI*> "`;`"
Mit <*ImportDeclaration*> wird ein Nichtterminal aus der Java-Grammatik bezeichnet und durch

die Zeichen ... wird angezeigt, dass die bestehende Grammatikregel der Java-Syntax erweitert wird. Eine <*URI*> referenziert eine DTD oder ein XML-Schema. □

Nach der obigen Grammatik ist es erlaubt, dass ein XOBE-Programm mehrere Sprachbeschreibungen deklariert. Dies ist bei unterschiedlichen Bezeichnern in den Sprachbeschreibungen auch problemlos möglich. Werden allerdings Bezeichner in unterschiedlichen Sprachbeschreibungen mehrfach verwendet, so ist der Einsatz qualifizierender Klassenbezeichner notwendig. Dies stellt eine zukünftige Erweiterungsmöglichkeit von XOBE dar.

Die Typen der im Programm deklarierten Sprachbeschreibung werden implizit als Klassendefinitionen für XML-Objekte interpretiert. Sie stehen nach der Deklaration über den Elementnamen, Gruppennamen oder Typnamen unmittelbar zur Verfügung. Eine explizite Erzeugung von Java-Quelltext für die XML-Objekt-Klassen findet nicht statt. XML-Objekte sind demnach Instanzen von Elementtypen, Gruppen oder Basistypen, die in der deklarierten Sprachbeschreibung definiert wurden. Damit repräsentiert der Wert jedes XML-Objekts ein XML-Fragment, das verschachtelte Elemente und Zeichendaten beinhalten kann. Durch eine Schemadeklaration werden in einem XOBE-Programm XML-Objekt-Klassen vereinbart. Diese impliziten Klassen sind als endgültig (Java-Terminus: `final`) vereinbart. Damit ist es nicht möglich, von diesen Klassen weitere Subklassen abzuleiten.

Im XOBE-Programm wird XML ausschließlich unter Verwendung von XML-Objekten verarbeitet, d. h. auf die Repräsentation als Zeichenkette wird während des Programmablaufs verzichtet. Trotzdem gibt es Fälle, in denen eine Umwandlung in eine Zeichenkette erforderlich wird. Dies wird notwendig, wenn von einem Programm XML-Daten an die Außenwelt kommuniziert werden, zum Beispiel als Resultat eines Java-Servlets. Für diesen Zweck wird die Methode `toString` für XML-Objekte zur Verfügung gestellt.

3.2.3 Deklaration von Variablen

Wie für jede Klasse in Java ist es mit XOBE möglich, Variablen für XML-Objektklassen zu deklarieren. Die Variablen dieser Klassen werden auch als XML-Variablen bezeichnet.

Definition 3.2 (XML-Variablendeklaration)
Eine XML-*Variablendeklaration* ist durch folgende Grammatik definiert:

<*Type*> → (<*BasicType*> | <*XMLType*> | <*Name*>) ("[" "]")*
<*XMLType*> → "xml" "<" (<*Name*> | "(" <*ChoiceType*> ")") ("+" | "*")? ">"
<*ChoiceType*> → <*Name*> | <*Name*> "|" <*ChoiceType*>

Mit <*Type*>, <*BasicType*> und <*Name*> werden Nichtterminale aus der Java-Grammatik bezeichnet. □

Die Definition zeigt, dass XML-Variablen durch das Schlüsselwort `xml` gefolgt von spitzen Klammern (`<`, `>`) mit dem Typbezeichner aus der Sprachbeschreibung deklariert werden. Mit der Anweisung `xml<title> t;` wird beispielsweise die Variable `t` der XML-Objektklasse deklariert, die durch die Elementtypdeklaration `title` in der Sprachbeschreibung impliziert

wird. In eindeutigen Fällen kann auf die Angabe des Schlüsselworts sowie der spitzen Klammern verzichtet werden. Die abkürzende Schreibweise lautet dann `title t;`. Zusätzlich ist es möglich Variablen zu deklarieren, die zur Laufzeit auf unterschiedliche XML-Objekte verweisen können. Mit `xml<(book|record)> i;` wird eine solche Variable deklariert, der entweder ein XML-Objekt vom Typ `book` oder vom Typ `record` zugewiesen werden kann. Mit Hilfe einer *Spezialisierung* „down-cast“ `(xml<book>) i` kann eine solche Referenz in eine der Varianten umgewandelt werden. Die abkürzende Schreibweise ist bei Variablendeklarationen für unterschiedliche XML-Objekte und für die in Abschnitt 3.2.5 eingeführten Elementlisten nicht zulässig.

3.2.4 Konstruktoren

In XOBE-Programmen werden XML-Objekte mit Hilfe von Ausdrücken erzeugt, die als XML-Objekt-Konstruktoren bezeichnet werden. Die Syntax dieser Ausdrücke besteht nicht nur aus wohldefiniertem XML, sondern muss auch die Bedingungen der statischen Gültigkeit erfüllen.

Als Erweiterung zur reinen XML-Syntax wird es gestattet, andere Java-Werte, Java-Objekte oder XML-Objekte in einen XML-Objekt-Konstruktor einzufügen. Syntaktisch werden diese Einfügepunkte mit geschweiften Klammern von der umgebenen XML-Notation abgegrenzt. Dies entspricht der Vorgehensweise, wie sie auch in XML-Anfragesprachen wie XQuery [W3C02c] zu finden ist. Die eingefügten Werte und Objekte sind allerdings nur an Stellen erlaubt, die nicht die Eigenschaften Wohldefiniertheit und statische Gültigkeit verletzen.

Definition 3.3 (XML-Objekt-Konstruktor)
Ein XML-*Objekt-Konstruktor* ist nach folgender Grammatik aufgebaut:

<Literal>	→	... | *<Element>*
<Element>	→	*<EmptyElementTag>* | *<STag>* *<Content>* *<ETag>*
<STag>	→	"<" *<Name_XML>* (*<Attribute>*)* ">"
<Attribute>	→	*<Name_XML>* "=" (*<AttValue>* | "{"*<Expression>*"}")
<ETag>	→	"</" *<Name_XML>* ">"
<Content>	→	(*<Element>* | *<CharData>* | *<Comment>* | "{"*<Expression>*"}")*
<EmptyElementTag>	→	"<" *<Name_XML>* (*<Attribute>*)* "/>"
<Comment>	→	"<!--" *<CommentData>* "-->"

<AttValue> ist dabei eine Zeichenkette in einfachen (') oder doppelten Hochkommata ("), *<Name_XML>* ein Elementname und *<CharData>* sowie *<CommentData>* bedeuten alphanumerische Zeichendaten. Mit *<Literal>* und *<Expression>* werden Nichtterminalsymbole aus der Java-Grammatik bezeichnet. □

Für XML-Konstruktoren wird gefordert, dass die Bedingungen der Sprachbeschreibung eingehalten werden. Erlaubt wird aber, dass `int`-Werte an Stellen eingesetzt werden, an denen laut Sprachbeschreibung Werte vom Basisdatentyp `integer` erwartet werden. Analoges gilt für Zeichenketten der Klasse `String`, die an `string`-Positionen zulässig sind. Bereits in Ab-

schnitt 3.1 wurden Beispiele für XML-Konstruktoren gezeigt.

3.2.5 Elementliste

XOBE stellt als zusätzliche Klasse eine Liste von XML-Objekten zur Verfügung. Da es sich hier in der Regel um Listen von XML-Elementen handelt, wird auch verkürzend von *Elementlisten* gesprochen, obwohl allgemeine Listen von XML-Objekten gemeint sind. Gerade für Schleifenkonstrukte und Rekursionen sind Elementlisten gut geeignet, um eine beliebige Anzahl gleicher oder unterschiedlicher Elemente aufzunehmen.

Für die Deklaration einer Elementliste gibt es zwei Varianten. Eine Erste kann über die in Abschnitt 3.2.3 eingeführten Operatoren + und * vorgenommen werden. Beispielsweise deklariert die Anweisung `xml<author*> al;` die Variable `al` als eine Liste von `author`-Elementen. Die zweite Möglichkeit besteht darin, dass der Name einer in der deklarierten Sprachbeschreibung definierten benannten Gruppe herangezogen wird, die bereits durch das Attribut `maxOccurs="unbounded"` als Liste definiert wurde. Die Syntax und Semantik einer Liste von XML-Objekten wird durch die Definition einer spezifizierten Schnittstelle festgelegt.

Definition 3.4 (Liste über XML-Objekte)
Eine *Liste über* XML*-Objekte* sei durch folgende Schnittstelle definiert:

```
1 interface XMLList<XMLObject>: XMLObject {
2     static XMLList <>();
3     static XMLList +(in XMLList l, in XMLObject o);
4     XMLObject item(in int i);
5     int getLength();
6     static XMLList +(in XMLObject o, in XMLList l);
7     static XMLList +(in XMLList l_1, in XMLList l_2);
8 } // XMLList
```

Die Operationen und Methoden der Schnittstelle werden durch folgende Anweisungsgleichungen mit den Variablen l, l_1, l_2 : `XMLList` und o, o_1, o_2 : `XMLObject` und i : `int` spezifiziert:

$$l.\texttt{getLength}() \overset{[l := \texttt{<>}]}{=} 0$$

$$\left[\begin{array}{l} l_2 := l_1 \texttt{ + } o; \\ i := l_2.\texttt{getLength}() \end{array}\right] \Leftrightarrow \left[\begin{array}{l} i := l_1.\texttt{getLength}() + 1; \\ l_2 := l_1 \texttt{ + } o \end{array}\right]$$

$$l_2.\texttt{item}(0) \overset{[l_2 := l_1 \texttt{ + } o]}{=} o$$

$$\left[\begin{array}{l} l_2 := l_1 \texttt{ + } o_1; \\ o_2 := l_2.\texttt{item}(i) \end{array}\right] \Leftrightarrow \left[\begin{array}{l} o_2 := l_1.\texttt{item}(i-1); \\ l_2 := l_1 \texttt{ + } o_1 \end{array}\right]$$

$$\texttt{<> + } o \overset{[\epsilon]}{=} o \texttt{ + <>}$$

$$(o_1 \texttt{ + } l) \texttt{ + } o_2 \overset{[\epsilon]}{=} o_1 \texttt{ + } (l \texttt{ + } o_2)$$

$$<> + l \overset{[\epsilon]}{=} l$$

$$(o + l_1) + l_2 \overset{[\epsilon]}{=} o + (l_1 + l_2)$$

□

Wie die Definition zeigt, wird mit <> die leere Elementliste bezeichnet. Der Operator + realisiert die Konkatenation der Elementliste in drei Varianten, eine zum Hinzufügen eines Elements am Ende der Liste, eine zum Hinzufügen am Anfang der Liste und eine zum Zusammenfügen zweier Listen. Zusätzlich liefert die Methode `getLength` die Länge der Elementliste und die Methode `item` selektiert das Element der Liste an der angegebenen Position. Das erste Element der Liste steht an der Position 0.

Da es sich bei Elementlisten wiederum um XML-Objekte handelt, stellt es kein Problem dar, sie in XML-Objekt-Konstruktoren einzufügen, wenn dies nach der deklarierten Sprachbeschreibung erlaubt ist.

Anmerkung: Obwohl eine Elementliste ein XML-Objekt ist und deshalb streng genommen ebenfalls ein Element der Liste sein könnte, wird dies von diesem Datentyp nicht unterstützt. Stattdessen wird die Konkatenation zweier Listen verwendet. Es wird also stets auf einer flachen Liste gearbeitet.

3.2.6 Selektion

Für die Selektion von Daten aus XML-Objekten wird in XOBE die Syntax von XPath, die in Abschnitt 2.2 vorgestellt wurde, verwendet. Die Semantik von XPath wird dafür auf das XOBE-Objektmodell adaptiert. XPath stellt, wie beschrieben, einen Mechanismus zur Verfügung, um mittels sogenannter Pfadausdrücke bestimmte Knoten aus einem XML-Fragment zu selektieren. Da in XOBE XML-Fragmente ausschließlich durch XML-Objekte repräsentiert werden, kann mittels dieser Pfadausdrücke auf Inhaltsdaten und verschachtelte XML-Objekte eines XML-Objekts zugegriffen werden.

Ein Pfadausdruck in XPath bezieht sich stets auf einen Kontextknoten. In XOBE wird dieser durch eine XML-Objekt-Variable vorgegeben und dem Pfadausdruck syntaktisch vorangestellt. Das Resultat eines Pfadausdrucks besteht in XOBE aus einer Elementliste, die, wie erwähnt, ebenfalls ein XML-Objekt darstellt. Dies ermöglicht es, die Ergebnisse von Pfadausdrücken in XML-Objekt-Konstruktoren weiter zu verwenden.

Definition 3.5 (XPath-Ausdruck)
Ein *XPath-Ausdruck* in XOBE ist nach folgender Grammatik aufgebaut:

<*Expression*>	→	... | <*XPathExpression*>
<*XPathExpression*>	→	<*Name*> "/" <*RelativeLocationPath*> | <*LocationPath*>
<*LocationPath*>	→	<*RelativeLocationPath*> | <*AbsoluteLocationPath*>
<*AbsoluteLocationPath*>	→	"/" <*RelativeLocationPath*>?
<*RelativeLocationPath*>	→	<*Step*> | <*RelativeLocationPath*> "/" <*Step*>

<Step>	→	*<AxisSpecifier>* *<NodeTest>* *<Predicate>**
<AxisSpecifier>	→	*<AxisName>* `"::"`
<AxisName>	→	`"ancestor"`|`"ancestor-or-self"`|`"attribute"`| `"child"`|`"descendant"`|`"descendant-or-self"`| `"following"`|`"following-sibling"`|`"parent"`| `"preceding"`|`"preceding-sibling"`|`"self"`
<NodeTest>	→	*<NameTest>* | *<NodeType>* `"("")"`
<Predicate>	→	`"["` *<PredicateExpr>* `"]"`
<PredicateExpr>	→	*<Expression>*
<NameTest>	→	`"*"` | *<Name>*
<NodeType>	→	`"comment"`|`"text"`|`"node"`

Bei *<Expression>* und *<Name>* handelt es sich um Nichtterminale aus der Java-Grammatik. □

Anmerkung: Die zweite Syntax-Variante des Nichtterminalen *<XPathExpression>* ist nur innerhalb von Prädikaten gestattet.

Anmerkung: Die erzeugten Listen der XPath-Ausdrücke sind nicht „live“. Damit ist gemeint, dass eine Veränderung des XML-Dokuments keine geänderte Liste nach sich zieht. Die Liste ist nicht mit dem Dokument gekoppelt. Etwaige Änderungen im Dokument werden also nicht in der Liste reflektiert und umgekehrt. Wird das Dokument modifiziert während eine Iteration über die Liste stattfindet, ist das Resultat der Iteration undefiniert. Trotzdem haben Änderungen der XML-Objekte in der Liste auch Auswirkungen auf das eigentliche Dokument, denn die Elementliste besteht, wie in Java üblich, aus Referenzen auf XML-Objekte.

3.3 Anwendungsbeispiele

In diesem Abschnitt werden anhand eines detaillierten Beispiels die in XOBE eingeführten Sprachkonstrukte verdeutlicht. Das Beispiel zeigt einen Web-Dienst, der eine Shop-Anwendung realisiert, die mit der Außenwelt über ein spezielles XML-Datenformat kommuniziert. Das Datenformat, das Verwendung findet, ist das Shop-Interchange-Formart (SIF) aus Abschnitt 2.1.2.

Implementiert wird das Beispiel mit einer Klasse `Cart`, die einen Einkaufskorb für den Shop realisiert und in Abbildung 3.1 als UML-Darstellung [BRJ99, RJB99] illustriert ist. Die Klasse hat eine Komponente `accountNr` vom Typ `int`, die den Einkaufskorb für jeden Kunden eindeutig identifiziert. Eine weitere Komponente `articles` der Schnittstelle `List` aus der Java-Bibliothek registriert die ausgewählten Artikel im Einkaufskorb. Dafür werden die Artikelnummern, die vom Typ `int` sind, in der Liste gespeichert. Zusätzlich deklariert die Klasse die drei Methoden `addArticle`, `removeArticle` und `getArticles`. Die Methode `addArticle` fügt einen Artikel zum Einkaufskorb hinzu, `removeArticle` entfernt einen Artikel aus diesem und `getArticles` liefert den Inhalt des Einkaufskorb mit den ausgewählten Artikeln.

Das folgende Beispiel zeigt die Realisierung der Methode `addArticle`.

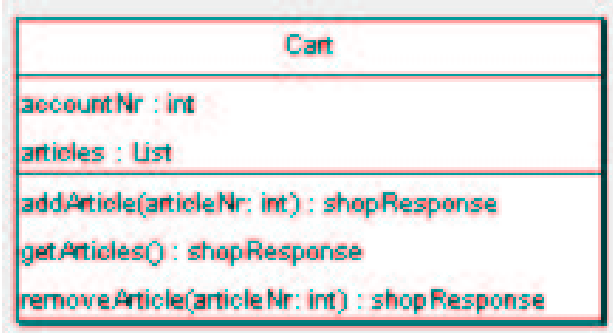

Abbildung 3.1: Klasse `Cart`

Beispiel 3.2
Die Methode `addArticle` liefert ein `shopResponse`-Objekt gemäß dem SIF (Beispiel 2.4) zurück. Als Parameter erhält die Methode die Nummer des Artikels, der hinzugefügt werden soll.

```
1 shopResponse addArticle(int articleNr){
2     this.articles.add(articleNr);
3     return <shopResponse>
4                <shoppingCart>
5                    <account>{this.accountNr}</account>
6                    <request>processed </request>
7                </shoppingCart>
8            </shopResponse>;
9 } // addArticle
```

Die Methode fügt zunächst die Artikelnummer der Artikelliste mit der Methode `add` aus der `List`-Schnittstelle hinzu (2). Anschließend wird mit einem XML-Objekt-Konstruktor, wie im vorherigen Abschnitt definiert, das XML-Objekt der Klasse `shopResponse` erzeugt und zurückgegeben (3-8). Der Konstruktor fügt in den Inhalt des `account`-Elements die Kundennummer, die aus der Komponente `accountNr` extrahiert wird, ein (5). Hier wird die schon beschriebene Eigenschaft der Konstruktoren ausgenutzt, dass `int`-Werte akzeptiert werden, an Stellen wo nach der deklarierten Sprachbeschreibung `integer`-Werte erwartet werden. □

Das Beispiel der Methode `addArticle` macht plausibel, dass eine Überprüfung der statischen Gültigkeit gemäß dem SIF zur Zeit der Programmübersetzung möglich ist, da sämtliche dafür relevanten Typinformationen zur Verfügung stehen. Das nächste Beispiel zeigt die Implementierung der Methode `removeArticle` der Klasse `Cart`.

Beispiel 3.3
Die Methode `removeArticle` erhält als Parameter eine Artikelnummer und liefert ein XML-Objekt der Klasse `shopResponse`.

```
1  shopResponse removeArticle(int articleNr){
2     request done;
3     shopResponse response;
4
5     if (this.articles.remove(articleNr))
6        done = <request>processed</request>;
7     else
8        done = <request>fail</request>;
9     response = <shopResponse>
10                   <shoppingCart>
11                   <account>{this.accountNr}</account>
12                   {done}
13                   </shoppingCart>
14                </shopResponse>;
15    return response;
16 } // removeArticle
```

In der Methode werden die zwei lokalen XML-Objekt-Variablen `done` und `response` unterschiedlicher XML-Objekt-Klassen deklariert (2,3). Die Zuweisung eines XML-Objekts an die Variable `done` hängt vom Erfolg der Löschung des Artikels mit der Nummer `articleNr` aus der Liste `articles` ab (5). Ist diese erfolgreich, wird ein Element mit Inhalt `processed` erzeugt (6), ansonsten mit Inhalt `fail` (8). Der Variablen `response` wird mit Hilfe eines Konstruktors ein `shopResponse`-Objekt zugewiesen (9-14). Es enthält ebenfalls wieder die Kundennummer des Einkaufskorbs (11) und zusätzlich die eingefügte Variable `done` (12). Das Beispiel zeigt wie XML-Objekt-Variablen innerhalb von Konstruktoren verwendet werden können. □

Es wird erneut offensichtlich, dass die statische Gültigkeit dieser Methode mit den angegebenen Typinformationen zur Programmübersetzung überprüfbar ist. Eine solche Typüberprüfung würde für die beiden vorgestellten Beispiele ergeben, dass die XML-Objekte gültig, also korrekt, sind. Im folgenden Beispiel wird diese Bedingung nicht eingehalten.

Beispiel 3.4
Dieses Beispiel zeigt nicht gültigen XOBE-Quelltext, für den der XOBE-Übersetzer eine entsprechende Fehlermeldung liefert.

```
1  request done;
2  shopResponse response;
3
4  done = <request>fail</request>;
5  response = <shopResponse>
6                <shoppingCart>
7                   <account>{this.accountNr}
```

```
                                        {done}</account>
8                   </shoppingCart>
9                </shopResponse>;
```

Nach Deklaration der beiden Variablen `done` und `response` als XML-Objekt-Variablen (1,2), erfolgt eine Zuweisung eines `request`-Elements an die Variable `done` (4). In der zweiten Zuweisung wird mittels eines Konstruktors ein `shopResponse`-Objekt an die Variable `response` zugewiesen (5-9). Dabei wird sowohl die Kundennummer `accountNr` als auch das Objekt der Variablen `done` in ein `account`-Element eingebettet (7). Nach der Sprachbeschreibung des SIF ist aber ein `request`-Element nicht im Inhalt eines `account`-Elements erlaubt. Somit muss dieser Quelltext vom XOBE-Übersetzer abgelehnt werden. □

Nach dem letzten Beispiel mit inkorrektem Programmtext, folgt nun ein Beispiel für Elementlisten, das die Methode `getArticles` realisiert.

Beispiel 3.5
Die Methode `getArticles` wird ohne Parameter aufgerufen und liefert erneut ein XML-Objekt der Klasse `shopResponse`.

```
1  shopResponse getArticles() {
2     int i;
3     xml<article*> content;
4
5     content = <>;
6     for (i = 0; i < this.articles.size(); i = i + 1)
7        content = content + <article>{this.articles.get(i)
            } </article>;
8     return <shopResponse>
9                <shoppingCart>
10                  <account>{this.accountNr}</account>
11                  <request>processed</request>
12                  <items>{content}</items>
13               </shoppingCart>
14            </shopResponse>;
15 } // getArticles
```

Die Methode deklariert zwei lokale Variablen; `i` vom Typ `int` als Laufvariable in der Schleife und `content` als XML-Objekt-Variable der Listenklasse `article*` (2,3). Der Variablen `content` wird als erstes eine leere Liste zugewiesen (5). Diese wird anschließend in der `for`-Schleife (6-7), die über die Artikelnummern der Komponente `articles` des Einkaufskorbes iteriert, um `article`-Objekte erweitert (7). Das Beispiel zeigt die Verwendung der Konkatenation `+` für Elementlisten. Die resultierende Elementliste `content` wird in einen Konstruktor, der ein `shopResponse`-Objekt erzeugt, eingebettet (8-14). Das `shopResponse`-Objekt wird

als Ergebnis der Methode zurückgegeben. □

Das letzte Beispiel dieses Abschnittes geht auf die Selektionsmöglichkeiten in XOBE ein. Es zeigt die Implementierung der Methode `processRequest`, die eine eingehende Anfrage eines Kunden verarbeitet.

Beispiel 3.6
Die Methode `processRequest` erhält als Parameter ein XML-Objekt der Klasse `shopRequest` und liefert als Ergebnis ein XML-Objekt der Klasse `shopResponse`, welches als Antwort an den anfragenden Kunden zurückgeschickt wird.

```
1  shopResponse processRequest(shopRequest rq) {
2      Cart c;
3      xml<shopRequest.shoppingCart> sc;
4
5      sc = (rq/child::shopRequest/child::shoppingCart).item
           (0);
6      c = allCarts.get((sc/child::account).item(0));
7
8      if ((sc/child::add).getLength() == 1)
9          return c.addArticle((sc/child::add).item(0));
10     else if ((sc/child::remove).getLength() == 1)
11         return c.removeArticle((sc/child::remove).item(0));
12     else if ((sc/child::get).getLength() == 1)
13         return c.getArticles();
14 } // processRequest
```

Die Methode reicht die eingehende Anfrage an den Einkaufskorb des kontaktierenden Kunden weiter und liefert die resultierende Antwort der entsprechenden Methoden des Einkaufskorbs. Um den Einkaufskorb des anfragenden Kunden zu finden, wird die globale Variable `allCarts` verwendet, die den Zugriff auf alle registrierten Einkaufskörbe des Web-Dienstes ermöglicht. Von der übergebenen Anfrage `rq` ermittelt die Methode das `shoppingCart`-Objekt und weist es der Variablen `sc` zu (5). Anschließend wird das entsprechende `Cart`-Objekt aus der Menge der in Verarbeitung befindlichen Einkaufskörbe `allCarts` durch die Methode `get` ausgewählt (6). In der abschließenden Fallunterscheidung (8-13) werden die drei möglichen Anfragetypen `add`, `remove` oder `get` differenziert und an die passenden Methoden des `Cart`-Objekts delegiert. Die Artikelnummern, die von den Methoden `addArticle` und `removeArticle` als Parameter benötigt werden, werden aus dem XML-Objekt der Variablen `sc` selektiert.

Die Methode `processRequest` verwendet zur Selektion des Inhalts eines XML-Objekts Pfadausdrücke (5-12), wie sie in Abschnitt 3.2.6 beschrieben werden. Die Pfadausdrücke liefern generell eine Liste von XML-Objekten, so dass für einen Zugriff auf ein bestimmtes Element selbst, die Listenmethode `item` angewendet werden muss (5,6,9,11). Die Länge der Resultatsliste eines Pfadausdrucks kann mittels der Methode `getLength` (8,10,12) untersucht werden. □

3.4 Bewertung

Zur Konstruktion von XOBE lässt sich zusammenfassend sagen, dass mit der Java-Erweiterung ein mächtiges und effizientes Werkzeug für die Programmierung von Web-Anwendungen gewonnen wurde. Von Interesse ist dabei einerseits die Möglichkeit, XML-Syntax auf einfache Art und Weise in Java zu verwenden, was durch das Konzept der XML-Objekte erreicht wurde. XML-Fragmente bezeichnen dadurch in einem XOBE-Programm stets XML-Objekte. Eine Unterscheidung zwischen einer Repräsentation als Zeichenkette und einer Repräsentation als Objektstruktur ist damit aufgehoben.

Andererseits wird mit Spracherweiterung gleichzeitig, neben der Wohlgeformtheit, die Eigenschaft der statischen Gültigkeit bereits zum Zeitpunkt der Programmübersetzung für sämtliche XML-Objekte, die auch dynamisch erzeugt werden dürfen, sichergestellt. Dies wird durch die Typüberprüfung der Java-Erweiterung, die Gegenstand des nächsten Kapitels sein wird, erreicht.

Obwohl XOBE mit der statischen Gültigkeit die meisten Anforderungen der Eigenschaft Gültigkeit von XML-Schema während der Programmübersetzung garantieren kann, kann für einige wenige Ausnahmen auf eine zusätzliche Überprüfung zur Laufzeit nicht verzichtet werden. Die Ursache dafür liegt darin, dass einige Objekteigenschaften erst zur Laufzeit der Anwendung feststehen, und damit außerhalb des XOBE-Typsystems liegen. Dies ist vergleichbar mit Feldern konstanter Größe („array“) in Standardprogrammiersprachen, bei denen der Zugriffsindex im deklarierten Intervall liegen muss. So wäre es in XOBE durchaus denkbar, dass aus einer Elementliste, also einem mit dem Attribut `maxOccurs="unbounded"` deklarierten Elementtyp, die leer ist, noch ein weiteres Element gelöscht werden soll. Weitere Laufzeitüberprüfungen sind notwendig für Identitätsbeschränkungen („identity constraints“) sowie abgeleitete Basisdatentypen, wie eingeschränkte Zeichenkettentypen („restricted string types“) und eingeschränkte numerische Typen („facets on numeric types“). Fehler dieser Art können erst während des Programmablaufs erkannt werden und sind dann durch geeignete Ausnahmebehandlungen abzufangen. Trotz dieser zusätzlichen Laufzeitberechnungen ist ein Effizienzgewinn gegenüber der vollständigen Überprüfung der Gültigkeit zur Laufzeit, wie es bei Verwendung des DOM notwendig ist, zu erwarten, da die Überprüfung nur für einige wenige Bedingungen durchgeführt werden muss.

Abschließend sind als Auswirkungen der Spracherweiterung auf die Programmiersprache Java zu erwähnen, dass eine Ergänzung der eingebauten Wertebereiche für Daten um XML-Fragmente wie auch eine dafür sinnvolle Erweiterung des Typsystems vorgenommen wurde. Wegen der herausragenden Rolle von XML in der Welt der Programmierung von Web-Anwendungen und Web-Services, sowie möglicherweise der gesamten Software-Entwicklungsbranche, scheint dies ein nur konsequenter Schritt zu sein. Die Nachteile, die durch die Erweiterung einer bestehenden Programmiersprache entstehen, anstatt eine neue Programmiersprache für genau diesen Zweck zu definieren, sind vernachlässigbar gering. Stattdessen überwiegen die Vorteile, die durch die Nutzungsmöglichkeiten von bereits entwickeltem Quelltext und von Bibliotheken entstehen.

Kapitel 4

Ein Typsystem für XOBE

Wie sich bei den Anwendungen des Dokument-Objektmodells im Kapitel 2.3 zeigt, ist die Erzeugung von Dokumenten möglich, die gemäß einer Sprachbeschreibung (DTD oder XML-Schema) ungültig sind. Ein wesentlicher Nachteil des DOM besteht darin, dass eine Überprüfung der Gültigkeit der in Verarbeitung befindlichen Dokumente erst zur Laufzeit der Anwendung vorgenommen werden kann. Da die Abwesenheit von Programmteilen, die die Gültigkeit verletzen, nicht garantiert werden kann, ist im DOM für die dadurch notwendige Überprüfung der Verbrauch zusätzlicher Rechenzeit unvermeidlich.

Um bei der Verarbeitung von Dokumenten deren Gültigkeit auf einfache Art sicherzustellen, ist es sinnvoll, eine Typüberprüfung für XML-Objekte zu entwickeln, die die Angaben der Sprachbeschreibung als Typsystem zu Grunde legt. Bereits in [Hos00] wurde aufbauend auf der Sprachbeschreibung ein Typsystementwurf für eine derartige Verarbeitung im Rahmen einer funktionalen Programmiersprache vorgestellt. Diese Idee wird hier weiterverfolgt und ausgebaut; insbesondere wird hier der Ansatz in den Kontext des objektorientierten Paradigmas gesetzt. Dafür wird das XOBE-Typsystem definiert, dessen Ausdruckskraft sich an XML-Schema orientiert.

Das Kapitel beginnt mit einer informellen Einführung der Typen in XOBE. Im Anschluss erfolgt eine Formalisierung der Typen und Sprachbeschreibungen, worauf aufbauend die Typinferenz für XML-Objekte und XPath-Ausdrücke definiert werden. Es folgt der Algorithmus zur Typüberprüfung. Den Abschluss des Kapitels bilden die Nachweise der Korrektheit des Algorithmus sowie Erläuterungen zu Erweiterungen und Vereinfachungen.

4.1 Einführung

Die in Kapitel 3 beschriebene Java-Erweiterung XOBE verwendet zur Programmierung XML-Objekte unterschiedlicher XML-Objekt-Klassen. XML-Klassen werden dabei ausschließlich in der Sprachbeschreibung, die eine Auszeichnungssprache bestimmt, definiert und stehen dem

XOBE-Programm durch Deklaration implizit zur Verfügung. Weitere Definitionen von XML-Klassen im XOBE-Programm sind nicht vorgesehen. Aufgabe des Typsystems ist es zunächst, die Typen[1] der im XOBE-Programm verwendeten Programmausdrücke zu inferieren. Anschließend wird anhand der inferierten Typen überprüft, ob die Programmausdrücke an zulässigen Positionen im Programm auftreten.

An einem kleinen Beispiel soll das Vorgehen gezeigt werden.

Beispiel 4.1
Das Beispiel 3.1 aus Abschnitt 3.1 zeigte die Zuweisung eines `book`-Elements an die Variable `b` der Klasse `book`. Zuvor wurden die Bezeichner `a` und `p` als Variablen der Klassen `author` und `price` deklariert und initialisiert.

```
4 book b = <book catalog="Varia">
5              <title>Lotte in Weimar</title>
6              {a}
7              <condition>Einband fingerfleckig , Rücken
                  verblaßt </condition>
8              {p}
9          </book>;
```

Die Aufgabe des XOBE-Typsystems ist es, für diese Zuweisung die Typen der beteiligten Ausdrücke zu inferieren. Für eine Zuweisung sind dies die Typen der linken und rechten Seiten, die im Weiteren mit s und r bezeichnet werden. Auf der linken Seite der Zuweisung steht die Variable `b`, für die das Typsystem gemäß der Deklaration den Typen $s = \mathit{book}$ inferiert. Die rechte Seite der Zuweisung besteht aus dem Element **`book`**, für das der folgende Typ ermittelt wird:[2]

$$r = \texttt{book}[\texttt{@catalog}[\texttt{string}]; \texttt{title}[\texttt{string}]; \mathit{author}; \texttt{condition}[\texttt{string}]; \mathit{price}]$$

Für diese beiden Typen wird im Anschluss an die Inferenz vom Typsystem die sogenannte Subtyp-Beziehung überprüft. Denn nur wenn die Ungleichung

$$r \leq s$$

gilt, wobei mit der Relation $\leq$ die Subtyp-Beziehung bezeichnet wird, ist der Typ der rechten Seite ein Subtyp des Typs der linken Seite. Und nur dann ist die obige Zuweisung korrekt und wird vom Typsystem akzeptiert. □

Wie das Beispiel zeigt, besteht das XOBE-Typsystem zur Überprüfung der Typkorrektheit aus mehreren Komponenten. Hinzu kommt noch das Einlesen der zu Grunde liegenden Sprachbeschreibung. Damit gliedert sich das XOBE-Typsystem in drei Teile:

- Die Formalisierung der Sprachbeschreibung übersetzt die deklarierte Sprachbeschreibung in eine für das Typsystem lesbare Form.

[1] Da im Weiteren das Typsystem von XOBE vorgestellt wird, wird hier von Typen oder XML-Typen anstatt von Klassen gesprochen.

[2] Eine Einführung in die formale Notation für XML-Typen erfolgt im nächsten Abschnitt.

- Die Typinferenz, die für XML-Konstruktoren und XPath-Ausdrücke zu unterscheiden ist, dient zur Ermittlung der XML-Typen.
- Der Subtyp-Algorithmus überprüft die Zulässigkeit der ermittelten XML-Typen. Dies ist die Hauptaufgabe bei der Untersuchung der Typkorrektheit von XOBE-Programmen.

Im nächsten Abschnitt wird sich zeigen, dass sich XML-Fragmente durch *reguläre Heckensprachen* formalisieren lassen. Nach [BKMW01] lassen sich *reguläre Baumautomaten* zu regulären *Heckenautomaten* erweitern, die dann reguläre Heckensprachen erkennen. Damit wäre für die Überprüfung der Subtyp-Beziehung der klassische Algorithmus zur Entscheidung der Sprachinklusion zweier Automaten möglich. Dieser berechnet für zwei gegebene Automaten M und $M^{'}$ zunächst das Komplement $\overline{M^{'}}$ von $M^{'}$ bevor anschließend der Durchschnitt zwischen M und $\overline{M^{'}}$ gebildet wird. Eine Sprachinklusion liegt genau dann vor, wenn das Resultat der Durchschnittsoperation leer ist.

Der Algorithmus arbeitet demnach mit der Berechnung des Komplements, die eine Determinisierung – also eine Konvertierung eines nicht-deterministischen Automaten in einen deterministischen Automaten – beinhaltet. In der Regel wird diese mit der Berry-Sethi-Methode [BS86] durch Teilmengen-Konstruktion durchgeführt. Dabei korrespondiert jeder Zustand des zu erzeugenden deterministischen Automaten mit einer Teilmenge von Zuständen des ursprünglichen nicht-deterministischen Automaten. Durch diese Konstruktion kann die Zustandsmenge im Allgemeinen exponentiell anwachsen.

Zu erhöhtem Aufwand führt zusätzlich, dass der Automat, dessen Terminalsymbole die Basisdatentypen der XML-Typen sind, jegliche Informationen über komplexe Typen und deren Typdefinitionen verliert. So sind beispielsweise bei der Überprüfung der Subtyp-Beziehung $(\mathit{book}*; \mathit{record}*) \leq (\mathit{book}|\mathit{record})*$ die Typdefinitionen von *book* und *record* irrelevant. Mit anderen Worten: Die Beziehung kann auch ohne das Wissen der Typdefinitionen verifiziert werden. Der klassische Algorithmus berücksichtigt eine solche Erleichterung nicht; stattdessen werden alle Typen bis zu den Basisdatentypen hin analysiert.

Der Algorithmus von Aiken und Murphy zur Überprüfung der Subtyp-Beziehung von *regulären Baumausdrücken* [AM91] ist in der Lage, mit komplexen Typen umzugehen. Er arbeitet, wie für Subtyp-Algorithmen vieler Typsystem üblich, von oben nach unten („top-down“). Dies bedeutet, dass er mit einem Paar von Typen, für das die Subtyp-Beziehung überprüft werden soll, beginnt. Da ein komplexer Typ sich mittels Typkonstruktoren zusammensetzt, kann in jedem Schritt überprüft werden, ob die obersten Konstruktoren beider Seiten zusammenpassen. Es erfolgt eine rekursive Anwendung auf die Teilkomponenten der Typen, bis Typnamen erreicht werden, die dann einfach zu überprüfen sind. Da die Typnamen nicht nur für Basisdatentypen, sondern auch für komplexe Typen stehen können, ist die angesprochene Überprüfungsvereinfachung möglich.

In [Hos00] wird der Algorithmus von Aiken und Murphy für XML adaptiert und auf *Baumautomaten mit Rangzahl* („ranked tree automata“) angewendet. Bei dieser Methode entsteht der Nachteil, dass eine häufige Umwandlung von XML-Typen in Baumautomaten notwendig wird.

Der Algorithmus der in diesem Kapitel (Abschnitt 4.6) vorgestellt wird, kommt ohne diese aufwendige Konvertierung aus.

Notation

Zur Definition der Formalisierung von Sprachbeschreibungen, für die Darstellung der Typinferenz und zur Beschreibung des Subtyp-Algorithmus werden in dieser Arbeit *Typisierungs-* oder *Inferenzregeln* verwendet. Die Notation, die ursprünglich im Bereich der Logik entwickelt [Tak75, Pra65, Fre67] wurde, ist zur Beschreibung von Typsystemen und Semantik von Programmiersprachen [Mit96, Car97] inzwischen weit verbreitet. Eine Inferenzregel besteht aus einer Linie mit einer oder mehreren Prämissen über der Linie und einer Konklusion unter der Linie. Gelten nun sämtliche Prämissen über der Linie, dann darf die Konklusion unterhalb der Linie abgeleitet werden. Folgendes Beispiel illustriert eine solche Inferenzregel:

Beispiel 4.2
Das Beispiel zeigt eine Vereinfachung einer Regel, wie sie in einem der folgenden Abschnitte verwendet wird. Dabei wird durch $cm : r$ notiert, dass das Inhaltsmodell cm vom XML-Type r ist.

$$\frac{\begin{array}{c} cm_1 : r_1 \\ cm_2 : r_2 \end{array}}{cm_1 cm_2 : (r_1; r_2)} \qquad \text{(SIMP CONC)}$$

Die Regel SIMP CONC sagt folgendes aus: Falls ableitbar ist, dass cm_1 vom Typ r_1 und cm_2 vom Typ r_2 ist, dann ist ebenfalls ableitbar, dass die Konkatenation $cm_1 cm_2$ vom Typ $r_1; r_2$ ist. Zum Beispiel sei $cm_1 =$ `<b>1</b>` und $cm_2 =$ `<i>zwei<i>` mit den Typen $r_1 =$ **b**[**integer**] und $r_2 =$ **i**[**string**]. Dann kann durch die Regel `<b>1</b><i>zwei<i>` : **b**[**integer**]; **i**[**string**] abgeleitet werden, weil sowohl `<b>1</b>` : **b**[**integer**] als auch `<i>zwei<i>` : **i**[**string**] gelten. □

Die Menge von Prämissen über der Linie kann in speziellen Fällen auch leer sein. In einer solchen Regel, die auch als *Axiom* bezeichnet wird, gilt die Konklusion unterhalb der Linie immer.

Klassischerweise werden Typisierungsregeln verwendet, um *Typisierungsurteile* („typing judgment“) abzuleiten. Ein Typisierungsurteil hat die Form $\Gamma \vdash m : r$. Dabei ist Γ ein gegebener Kontext, der beispielsweise aus Variablendeklarationen bestehen kann, m ist ein typisierbarer Ausdruck und r der Typ von m. Bei leerem Kontext wird auch statt $\emptyset \vdash m : r$ abkürzend $m : r$ angegeben. In dieser Arbeit werden darüber hinaus weitere Relationen mit Inferenzregeln definiert. So erfolgt die Formalisierung der Sprachbeschreibung als auch des Subtyp-Algorithmus unter Angabe von Inferenzregeln.

4.2 Formalisierung

Nach der obigen informellen Einführung definiert dieser Abschnitt die formalen Grundlagen, die nötig sind, um später den Algorithmus zur Überprüfung der Typkorrektheit zu formulieren. Dabei wird die Menge der natürlichen Zahlen mit $\mathcal{N}$ (die Menge der natürlichen Zahlen inklusive der Null mit $\mathcal{N}_0 = \mathcal{N} \cup \{0\}$) und die Menge der booleschen Werte mit $\mathcal{B}$ angegeben; mit $\mathcal{P}(s)$ wird die Potenzmenge von s bezeichnet und $\uplus$ steht für die disjunkte Vereinigung.

In XOBE werden XML-Typen durch *reguläre Heckenausdrücke* („regular hedge expression“) [Mur01] formalisiert, die *reguläre Heckensprachen* („regular hedge languages“) repräsentieren. Eine Einführung zu regulären Heckensprachen findet sich in [BKMW01]. Heckensprachen sind eng verwandt mit den *Baumsprachen* („tree languages“), denn eine *Hecke* („hedge“) ist eine geordnete Sequenz von Bäumen. In der Literatur [Neu99] wird gelegentlich auch von Wald („forest“) gesprochen, doch ist dieser Begriff bereits in der Graphentheorie [Die00, Tur96] als ungeordnete Menge von Bäumen definiert. Eine ausführliche Einführung zu regulären Baumsprachen findet sich in [RS97, CDG+97].

Die folgenden Definitionen für Hecke und reguläre Heckenausdrücke verwendet eine ähnliche Notation wie [W3C01a].

Definition 4.1 (Hecke)
Eine *Hecke* über einer Menge von Terminalsymbolen $T = B \uplus E$ mit der Menge B von Basisdatentypen und der Menge E von Elementnamen ist induktiv definiert:

$$
\begin{aligned}
&\epsilon \text{ ist die leere Hecke,}\\
&b \text{ mit } b \in B \text{ ist eine Hecke,}\\
&e[v] \text{ mit } e \in E \text{ und der Hecke } v \text{ ist eine Hecke und}\\
&vw \text{ mit den Hecken } v \text{ und } w \text{ ist eine Hecke.}
\end{aligned}
$$

Die *Menge aller Hecken* sei mit T^* bezeichnet und die *Menge aller Hecken ohne die leere Hecke* mit T^+ notiert. □

Wie die Definition zeigt, besteht eine Hecke entweder aus der leeren Hecke, aus einem Basisdatentypen, aus einem Elementnamen mit einer Hecke als Inhalt oder aus der Konkatenation zweier Hecken. Für Hecken ist es möglich reguläre Heckenausdrücke zu definieren, von denen im Weiteren der Arbeit auch kurz von regulären Ausdrücken gesprochen wird.

Definition 4.2 (Regulärer Heckenausdruck)
Die Menge der *regulären Heckenausdrücke Reg* über einer Menge von Terminalsymbolen $T = B \uplus E$ mit der Menge B von Basisdatentypen und der Menge E von Elementnamen sowie einer

Menge N von Nichtterminalsymbolen ist rekursiv definiert durch:

$$\begin{aligned}
\emptyset &\in \textit{Reg} && \text{für die leere Menge,}\\
\epsilon &\in \textit{Reg} && \text{für die leere Hecke,}\\
b &\in \textit{Reg} && \text{für Basisdatentypen,}\\
n &\in \textit{Reg} && \text{für komplexe Typen,}\\
e[r] &\in \textit{Reg} && \text{für Elementtypen,}\\
(r|s) &\in \textit{Reg} && \text{für die Operation reguläre Vereinigung,}\\
(r;s) &\in \textit{Reg} && \text{für die Operation Konkatenation und}\\
(r)* &\in \textit{Reg} && \text{für die Operation Kleene-Stern}
\end{aligned}$$

für alle $b \in B$, $n \in N$, $e \in E$ und $r, s \in$ *Reg*. □

Anmerkung: Für die Operatoren werden die Vorrangregeln in der absteigenden Reihenfolge $*$, ; und | festgelegt. Auf die Klammerung eines regulären Heckenausdrucks kann dann in eindeutigen Fällen auch verzichtet werden. Die aus der DTD bekannten Operatoren $+$ und $?$ sind mit regulären Heckenausdrücken ebenfalls darstellbar. So entspricht $r+$ dem Ausdruck $r|r*$ und $r?$ dem Ausdruck $r|\epsilon$.

Aufbauend auf regulären Ausdrücken kann die durch einen regulären Ausdruck bezeichnete Sprache festgelegt werden.

Definition 4.3 (Sprache eines regulären Heckenausdrucks)
Die *Heckensprache* $L(r)$ über einen regulären Heckenausdruck $r \in$ *Reg* bei einer gegebenen Menge von Produktionen P (wie sie in Definition 4.5 festgelegt werden) sei definiert durch:

$$\begin{aligned}
L(\emptyset) &= \{\}\\
L(\epsilon) &= \{\epsilon\}\\
L(b) &= \{b\}\\
L(n) &= L(r) \text{ mit } n \to r \in P\\
L(e[r]) &= \{e[u] | u \in L(r)\}\\
L(r|s) &= L(r) \cup L(s)\\
L(r;s) &= \{uv | u \in L(r), v \in L(s)\}\\
L(r*) &= \{\epsilon\} \cup L(r;r*)
\end{aligned}$$

für alle $b \in B$, $n \in N$, $e \in E$ und $r, s \in$ *Reg*. □

Es ist einsichtig, dass es reguläre Ausdrücke gibt, die eine Sprache repräsentieren, die die leere Hecke beinhalten. Für alle diese regulären Ausdrücke soll das Prädikat *isNullable?* erfüllt sein.

Definition 4.4 (Leere-Hecke-Prädikat)
Das *Leere-Hecke-Prädikat isNullable?* : *Reg* $\to \mathcal{B}$ entscheidet für einen gegebenen regulären Ausdruck $r \in$ *Reg* und eine Menge P von Produktionen (wie sie in Definition 4.5 definiert

werden), ob die leere Hecke in der Sprache $L(r)$ liegt. Es ist wie folgt rekursiv definiert:

$$\begin{aligned}
\textit{isNullable?}(\emptyset) &= \textit{false} \\
\textit{isNullable?}(\epsilon) &= \textit{true} \\
\textit{isNullable?}(b) &= \textit{false} \\
\textit{isNullable?}(n) &= \textit{isNullable?}(r) \text{ mit } n \to r \in P \\
\textit{isNullable?}(e[r]) &= \textit{false} \\
\textit{isNullable?}(r|s) &= \textit{isNullable?}(r) \vee \textit{isNullable?}(s) \\
\textit{isNullable?}(r;s) &= \textit{isNullable?}(r) \wedge \textit{isNullable?}(s) \\
\textit{isNullable?}(r*) &= \textit{true}
\end{aligned}$$

mit $b \in B$, $n \in N$, $e \in E$ und $r, s \in \textit{Reg}$. □

Auf der Basis dieser Definitionen lässt sich nun die reguläre Heckengrammatik einführen, die in der Lage ist, eine XML-Sprachbeschreibung formal darzustellen.

Definition 4.5 (Reguläre Heckengrammatik)
Eine *reguläre Heckengrammatik* sei definiert durch $G = (T, N, s, P)$ mit

$T = B \uplus E$ einer Menge von Terminalsymbolen, bestehend aus Basisdatentypen B und einer Menge E von Elementnamen (Tags),

N einer Menge von Nichtterminalsymbolen (Namen von Gruppen oder komplexen Typen),

s einem Startausdruck mit $s \in \textit{Reg}$ und

P einer Menge von Produktionsregeln der Form $n \to r$ mit $n \in N$ und $r \in \textit{Reg}$, einem regulären Heckenausdruck über T und N.

Für die Regeln der Produktionsmenge P gelten die folgenden zwei Bedingungen:

1. Bei rekursiven Nichtterminalsymbolen tritt die rekursive Anwendung nur an letzter Position einer Produktion auf.

2. Bei rekursiven Nichtterminalsymbolen gilt für den Ausdruck s vor der rekursiven Anwendung $\neg \textit{isNullable?}(s)$.

□

Die in der Definition der Heckengrammatik angegebenen Bedingungen für rekursive Produktionen sind notwendig, um die Regularität der Grammatik zu gewährleisten. Dies entspricht der Rechtslinearität der regulären Grammatiken [HU79]. Eine Verletzung der Bedingungen würde

zu Grammatiken führen, die mindestens so ausdrucksstark wie kontextfreie Grammatiken sind, wie folgendes Beispiel zeigt:

$$\mathit{offer} \rightarrow \texttt{book}[\texttt{string}]; \mathit{offer}; \texttt{book}[\texttt{string}] \mid \epsilon$$

Da aber sowohl das Entscheidungsproblem für die Sprachinklusion zweier kontextfreier Sprachen, als auch die Entscheidung, ob eine kontextfreie Grammatik eine reguläre Sprache erzeugt, unentscheidbar sind [HU79], werden die beiden syntaktischen Bedingungen eingeführt.[3] Es wird von einer *wohlgeformten* Grammatik gesprochen, falls jede Produktion der Grammatik diese Bedingungen erfüllt.

Anmerkung: Analog zu XML-Schema ist es in Heckengrammatiken erlaubt, einen Namen sowohl als Elementnamen in E als auch als Nichtterminalsymbole in N zu verwenden. Eine Unterscheidung wird in dieser Arbeit durch unterschiedliche Schriftarten vorgenommen.

Zusätzlich sei erwähnt, dass der Kleene-Stern-Operator $r*$ eines Heckenausdrucks in einer Hekkengrammatik stets durch eine rekursive Regel $n \rightarrow r; n|\epsilon$ darstellbar ist. Um aber die Produktionsmengen in der Darlegung dieser Arbeit nicht durch zusätzliche künstliche Produktionen unnötig zu erweitern, wird am Kleene-Stern-Operator festgehalten.

Will man ermitteln, ob eine Heckengrammatik die geforderten Bedingungen der Wohlgeformtheit einhält, ist ein weiteres Prädikat notwendig. Für dieses wird zunächst als Hilfsprädikat die *Bewachtheit* auf regulären Ausdrücken hinsichtlich eines Nichtterminalsymbols definiert. Es gibt an, ob das Nichtterminalsymbol nur im Inhalt eines Elementtyps – also bewacht – auftritt.

Definition 4.6 (Bewachtheit)
Die *Bewachtheit* sei ein Prädikat $\mathit{guarded} : N \times \mathit{Reg} \rightarrow \mathcal{B}$, das für einen regulären Ausdruck $r \in \mathit{Reg}$ entscheidet, ob ein Nichtterminalsymbol $x \in N$ ausschließlich im Inhalt eines Elements auftritt:

$$\begin{aligned}
\mathit{guarded}_x(\emptyset) &= \mathit{true} \\
\mathit{guarded}_x(\epsilon) &= \mathit{true} \\
\mathit{guarded}_x(b) &= \mathit{true} \\
\mathit{guarded}_x(n) &= \begin{cases} \mathit{false} & \text{falls } x = n \\ \mathit{guarded}_x(r) \text{ mit } n \rightarrow r \in P & \text{sonst} \end{cases} \\
\mathit{guarded}_x(e[r]) &= \mathit{true} \\
\mathit{guarded}_x(r|s) &= \mathit{guarded}_x(r) \wedge \mathit{guarded}_x(s) \\
\mathit{guarded}_x(r; s) &= \mathit{guarded}_x(r) \wedge \mathit{guarded}_x(s) \\
\mathit{guarded}_x(r*) &= \mathit{guarded}_x(r)
\end{aligned}$$

mit $b \in B$, $n \in N$, $e \in E$ und $r, s \in \mathit{Reg}$. □

[3] Als Alternative wären auch analoge Bedingungen möglich, die der Linkslinearität bei regulären Grammatiken entsprechen.

Mit dem Prädikat ist es nun möglich ein Prädikat zu definieren, das für einen regulären Ausdruck verifiziert, ob dieser hinsichtlich eines Nichtterminalsymbols wohlgeformt ist.

Definition 4.7 (Wohlgeformtheit eines regulären Ausdrucks)
Die *Wohlgeformtheit* eines regulären Ausdrucks $r \in \mathit{Reg}$ bezüglich eines Nichtterminalsymbols $x \in N$ sei ein Prädikat $\mathit{wellformed} : N \times \mathit{Reg} \to \mathcal{B}$, das wie folgt definiert ist:

$$
\begin{aligned}
\mathit{wellformed}_x(\emptyset) &= \mathit{true} \\
\mathit{wellformed}_x(\epsilon) &= \mathit{true} \\
\mathit{wellformed}_x(b) &= \mathit{true} \\
\mathit{wellformed}_x(n) &= \begin{cases} \mathit{false} & \text{falls } x = n \\ \mathit{wellformed}_x(r) \text{ mit } n \to r \in P & \text{sonst} \end{cases} \\
\mathit{wellformed}_x(e[r]) &= \mathit{true} \\
\mathit{wellformed}_x(r|s) &= \mathit{wellformed}_x(r) \wedge \mathit{wellformed}_x(s) \\
\mathit{wellformed}_x(r;s) &= \begin{cases} \mathit{guarded}_x(r) \wedge \mathit{wellformed}_x(s) & \text{falls } \mathit{isNullable?}(r) \\ \mathit{guarded}_x(r) \wedge \mathit{wellformed}'_x(s) & \text{sonst} \end{cases} \\
\mathit{wellformed}_x(r*) &= \mathit{guarded}_x(r)
\end{aligned}
$$

Dabei sei das Hilfsprädikat $\mathit{wellformed}'$ wie *wellformed* definiert mit den folgenden Änderungen für Nichtterminalsymbole und reguläre Konkatenation:

$$
\begin{aligned}
\mathit{wellformed}'_x(n) &= \begin{cases} \mathit{true} & \text{falls } x = n \\ \mathit{wellformed}'_x(r) \text{ mit } n \to r \in P & \text{sonst} \end{cases} \\
\mathit{wellformed}'_x(r;s) &= \mathit{guarded}_x(r) \wedge \mathit{wellformed}'_x(s)
\end{aligned}
$$

mit $b \in B$, $n \in N$, $e \in E$ und $r, s \in \mathit{Reg}$. □

Somit kann die Wohlgeformtheit einer ganzen Heckengrammatik überprüft werden.

Definition 4.8 (Wohlgeformtheit einer Heckengrammatik)
Eine reguläre Heckengrammatik $G = (T, N, s, P)$ ist *wohlgeformt*, falls gilt:

$$\bigwedge_{n \to r \in P} \mathit{wellformed}_n(r)$$

mit $n \in N$ und $r \in \mathit{Reg}$. □

Anmerkung: Die Bedingung der Wohlgeformtheit wird von einer Heckengrammatik, die durch die Formalisierung einer DTD oder eines XML-Schemas entsteht, stets erfüllt. Dies liegt daran, dass XML-Schema keine rekursiven Gruppen erlaubt und in einer DTD Gruppen nur durch Parameter-Entities nachgebildet werden können, die ebenfalls nicht rekursiv definiert sein dürfen.

Der folgende Satz lässt sich für reguläre Heckengrammatiken aufstellen.

Satz 4.1
Die von regulären Heckengrammatiken erzeugten Sprachen entsprechen regulären Baumsprachen.

Beweis: In [Hos00] werden unter der Bezeichnung *reguläre Ausdruckstypen* ebenfalls Heckengrammatiken verwendet. Bis auf den Kleene-Stern unterscheiden sich diese nicht von den Heckengrammatiken in dieser Arbeit. Da ein Kleene-Stern, wie erwähnt, stets durch eine zusätzliche Produktion ausgedrückt werden kann, stellt dieser keine Erweiterung des Sprachumfangs dar. In [Hos00] wird die Konstruktion eines regulären Baumautomaten aus einer regulären Heckengrammatik angegeben, der exakt die gleiche Sprache erkennt, die die Heckengrammatik erzeugt. Weiterhin ist für reguläre Baumautomaten bekannt, dass diese reguläre Baumsprachen akzeptieren [RS97]. Somit erzeugen die Heckengrammatiken dieser Arbeit ebenfalls reguläre Baumsprachen. □

Dass eine Heckengrammatik eine Baumsprache erzeugt, ist zunächst verwunderlich. In [Hos00] wird aber gezeigt, dass eine Hecke stets durch einen Baum darstellbar ist.

Die Hauptaufgabe des Typsystems ist, die Subtyp-Beziehung zweier XML-Typen zu überprüfen. Die Subtyp-Beziehung lässt sich durch eine reguläre Ungleichung ausdrücken und ist über die Heckensprachen der regulären Ausdrücke definiert.

Definition 4.9 (Reguläre Ungleichung)
Eine *reguläre Ungleichung*, oder kurz Ungleichung, zweier regulärer Ausdrücke sei definiert durch:

$$r \leq s \Leftrightarrow L(r) \subseteq L(s)$$

für $r, s \in \mathit{Reg}$. □

Es wird von einer *trivial inkonsistenten* Ungleichung gesprochen, falls die leere Hecke in der Sprache des linken regulären Ausdrucks enthalten ist, aber nicht in der Sprache des rechten Ausdrucks.

Definition 4.10 (Inkonsistenz)
Eine reguläre Ungleichung heißt *inkonsistent*, falls gilt:

$$\mathit{inc}(r \leq s) = \mathit{isNullable?}(r) \wedge \neg \mathit{isNullable?}(s)$$

mit $r, s \in \mathit{Reg}$. □

Für die spätere Formulierung des Subtyp-Algorithmus wird eine Funktion benötigt, die die führenden Terminalsymbole eines regulären Ausdrucks ermittelt, was die folgende Funktion leistet.

Definition 4.11 (Führende Terminalsymbole)
Die Menge der *führenden Terminalsymbole* eines regulären Ausdrucks $r \in \mathit{Reg}$ sei definiert

durch die Funktion *term* : *Reg* $\rightarrow \mathcal{P}(T)$. Sie ist rekursiv definiert durch:

$$
\begin{aligned}
\textit{term}(\emptyset) &= \{\} \\
\textit{term}(\epsilon) &= \{\} \\
\textit{term}(b) &= \{b\} \\
\textit{term}(n) &= \textit{term}(r) \text{ mit } n \rightarrow r \in P \\
\textit{term}(e[r]) &= \{e\} \\
\textit{term}(r|s) &= \textit{term}(r) \cup \textit{term}(s) \\
\textit{term}(r;s) &= \begin{cases} \textit{term}(r) \cup \textit{term}(s) & \text{falls } \textit{isNullable?}(r) \\ \textit{term}(r) & \text{falls } \neg\textit{isNullable?}(r) \end{cases} \\
\textit{term}(r*) &= \textit{term}(r)
\end{aligned}
$$

mit $b \in B$, $n \in N$, $e \in E$ und $r, s \in$ *Reg*. □

Die Operation ist rekursiv definiert und liefert die führenden Terminalsymbole eines regulären Ausdrucks. Trifft die Operation auf ein Nichtterminalsymbol zu, wird die Operation mit dessen Produktionsdefinition aufgerufen. Für die reguläre Konkatenation wird das Prädikat *isNullable?* herangezogen, um zu ermitteln, ob die Operation rekursiv auf beide oder nur auf den ersten Teilausdruck angewendet werden muss.

4.3 XML-Schema als Heckengrammatik

Die Typüberprüfung eines XOBE-Programms erfolgt mit einem Algorithmus, der auf der Basis von Heckengrammatiken arbeitet, wie sie im letzten Abschnitt eingeführt wurden. Aus diesem Grund ist es notwendig, dass XML-Typen, die durch Deklaration der Sprachbeschreibung einem XOBE-Programm bekannt sind, in dieser Form dargestellt werden. Dieser Abschnitt definiert die Formalisierung von XML-Schemata als Heckengrammatiken. Die formale Beschreibung besteht aus zwei Relationen.

Definition 4.12 (Formalisierung einer Sprachbeschreibung)
Gegeben sei ein XML-Schema S, dann ist die Formalisierung von S definiert durch zwei Relationen:

$S \hookrightarrow r$	r formalisiert die Komponente S als regulären Heckenausdruck.
$S \mapsto p$	p formalisiert die Komponente S als Produktionsregel.

Dabei sei $r \in$ *Reg* und $p \in P$. □

In der Definition der Formalisierungsrelationen steht der Operator $\&$ als abkürzende Schreibweise für sämtliche Permutationen von Konkatenationen der beteiligten regulären Ausdrücke. Die

Hilfsfunktion *occurs*, die im Folgenden definiert wird, erweitert einen regulären Ausdruck derart, dass die Bedingungen der Attribute `minOccurs` und `maxOccurs` berücksichtigt werden. So ergibt sich beispielsweise für $\mathit{occurs}(r, 2, 4)$ das Resultat $r; r; (r|\epsilon); (r|\epsilon)$.

Definition 4.13 (Funktion *occurs*)
Die Funktion $\mathit{occurs} : \mathit{Reg} \times \mathcal{N}_0 \times \mathcal{N} \cup \{*\} \rightarrow \mathit{Reg}$ ist definiert durch:

$$\begin{aligned}
\mathit{occurs}(r, 0, 1) &= (r|\epsilon) \\
\mathit{occurs}(r, 1, 1) &= r \\
\mathit{occurs}(r, 0, *) &= r* \\
\mathit{occurs}(r, m, *) &= (r; \mathit{occurs}(r, m-1, *)) \\
\mathit{occurs}(r, 0, k) &= ((r|\epsilon); \mathit{occurs}(r, 0, k-1)) \\
\mathit{occurs}(r, m, k) &= (r; \mathit{occurs}(r, m-1, k-1)) \text{ falls } m \leq k
\end{aligned}$$

mit $r \in \mathit{Reg}$, $m \in \mathcal{N}$ und $k \in \mathcal{N} \backslash \{1\}$. □

Mit den anschließenden Regeln werden aus den Komponenten des XML-Schemas die regulären Heckenausdrücke ermittelt. In diesen wird die in Abschnitt 4.1 beschriebene Notation verwendet. Die Relationen verknüpfen einen Ausdruck des XML-Schemas auf der linken Seite mit einem Ausdruck aus der Heckengrammatik auf der rechten Seite. Dabei wird ein Attribut eines Elements aus dem XML-Schema auf der linken Seite der Relation durch die Angabe von @ selektiert. Dadurch müssen nicht alle möglichen Attributkombinationen aufgeführt werden.

Definition 4.14 (Formalisierung mittels Ausdrucksrelation)
Die *Ausdrucksrelation* $\hookrightarrow$ ist durch folgende Regeln definiert:

$$\frac{}{\texttt{integer} \hookrightarrow \texttt{integer}} \quad \text{(INT)}$$

$$\frac{}{\texttt{string} \hookrightarrow \texttt{string}} \quad \text{(STR)}$$

$$\frac{}{o \hookrightarrow o} \quad \text{(IDENT)}$$

$$\frac{cm_1 \hookrightarrow r_1, \ldots, cm_n \hookrightarrow r_n}{\texttt{<all>}cm_1 \ldots cm_n\texttt{</all>} \hookrightarrow (r_1 \& \ldots \& r_n)} \quad \text{(ALL)}$$

$$\frac{cm_1 \hookrightarrow r_1, \ldots, cm_n \hookrightarrow r_n}{\begin{array}{l}\texttt{<sequence } ag\texttt{>}cm_1 \ldots cm_n\texttt{</sequence>} \\ \texttt{@minOccurs} = m, \texttt{@maxOccurs} = k\end{array} \hookrightarrow \mathit{occurs}((r_1; \ldots; (r_{n-1}; r_n) \ldots), m, k)} \quad \text{(SEQ)}$$

$$\frac{cm_1 \hookrightarrow r_1, \ldots, cm_n \hookrightarrow r_n}{\texttt{<choice } ag\texttt{>}cm_1 \ldots cm_n\texttt{</choice>} \quad \texttt{@minOccurs} = m, \texttt{@maxOccurs} = k \hookrightarrow \textit{occurs}((r_1 | \ldots | (r_{n-1} | r_n) \ldots), m, k)} \quad \text{(CHOICE)}$$

$$\frac{i \hookrightarrow o}{\begin{array}{l}\texttt{<group } ag\texttt{/>} \\ \texttt{@ref} = i, \qquad \hookrightarrow \textit{occurs}(o, m, k) \\ \texttt{@minOccurs} = m, \texttt{@maxOccurs} = k\end{array}} \quad \text{(GROUP REF)}$$

$$\frac{i \hookrightarrow o}{\begin{array}{l}\texttt{<element } ag\texttt{/>} \\ \texttt{@ref} = i, \qquad \hookrightarrow \textit{occurs}(o, m, k) \\ \texttt{@minOccurs} = m, \texttt{@maxOccurs} = k\end{array}} \quad \text{(ELEM REF)}$$

$$\frac{i \hookrightarrow i', t \hookrightarrow r}{\begin{array}{l}\texttt{<attribute } ag\texttt{/>} \\ \texttt{@name} = i, \texttt{@type} = t \quad \hookrightarrow \texttt{@}i'[r]\end{array}} \quad \text{(ATTR)}$$

$$\frac{\texttt{<complexType } ag\texttt{>}cm\texttt{</complexType>} \mapsto (o \to r)}{\texttt{<complexType } ag\texttt{>}cm\texttt{</complexType>} \hookrightarrow o} \quad \text{(COMPL)}$$

$$\frac{\texttt{<element } ag\texttt{>}cm\texttt{</element>} \mapsto (o \to r)}{\texttt{<element } ag\texttt{>}cm\texttt{</element>} \hookrightarrow o} \quad \text{(GLOB ELEM)}$$

$$\frac{\texttt{<element } ag\texttt{>}cm\texttt{</element>} \mapsto (o \to r)}{\begin{array}{l}\texttt{<element } ag\texttt{>}cm\texttt{</element>} \\ \texttt{@minOccurs} = m, \texttt{@maxOccurs} = k \quad \hookrightarrow \textit{occurs}(o, m, k)\end{array}} \quad \text{(LOC ELEM)}$$

mit $m \in \mathcal{N}_0$, $n, k \in \mathcal{N}$, $\texttt{@}i' \in T$ und $o \in N$ und $r, r_1, \ldots, r_n \in \textit{Reg}$. □

In den Regeln INT und STR werden die Datentypen aus dem XML-Schema auf Terminalzeichen der resultierenden Heckengrammatik abgebildet. Mittels IDENT werden die Bezeichner aus der Sprachbeschreibung als Terminal- oder Nichtterminalsymbole übernommen. Durch die Definitionen der Regeln ALL, SEQ und CHOICE werden die Inhaltsmodelle, so wie sie in der Sprachbeschreibung formuliert sind, in Heckenausdrücke überführt.

Für Referenzen auf Gruppendefinitionen oder Elementdeklarationen wird durch die Regeln GROUP REF und ELEM REF in der Heckengrammatik das entsprechende Nichtterminalsymbol verwendet. Dies ist immer möglich, da es für jede Gruppendefinition und Elementdeklaration in der Heckengrammatik ein korrespondierendes Nichtterminalsymbol gibt. Mit der Operation *occurs* werden ebenfalls die Zusatzbedingungen bezüglich der Häufigkeit des Auftretens reflektiert.

Durch ATTR wird die Attributdeklaration in einen Heckenausdruck transformiert. Attributtypen werden in der Heckengrammatik analog zu Elementtypen behandelt. Sie unterscheiden sich lediglich durch spezielle Terminalzeichen (mit führendem `@`) und einem einfacheren Inhaltstypen. Durch die Definitionen der Regeln COMPL, GLOB ELEM und LOC ELEM wird auf die im restlichen Abschnitt definierte Produktionenrelation $\mapsto$ zur Formalisierung zurückgegriffen, die für die Definition eines komplexen Typs oder einer lokalen Elementdeklaration eine Produktion der resultierenden Heckengrammatik erzeugt. Als Ergebnis der Regel wird dann das Nichtterminalsymbol dieser erzeugten Produktion zurückgegeben, das bei lokalen Elementdeklarationen mit der Hilfsoperation *occurs* gemäß den Bedingungen der Attribute `minOccurs` und `maxOccurs` angepasst wird.

Anmerkung: Für die Formalisierung eines XML-Schemas wird in dieser Arbeit angenommen, dass alle Bezeichner bereits in einer eindeutigen Form vorliegen. In [W3C01a] wird eine solche Prozedur während der Normalisierung einer Sprachbeschreibung definiert, auf die hier aber nicht näher eingegangen wird. Dort wird jeder Bezeichner durch den vollständigen Bezeichnerpfad vom Wurzelelement bis zur Definition oder Deklaration in der Sprachbeschreibung (mit dem Trennsymbol `/`) eindeutig qualifiziert. Da eindeutige Bezeichner in Beispielen die Komplexität unnötig erhöhen, werden im Folgenden die Bezeichner meist eindeutig gewählt, und in diesen Fällen die einfachere Ausgangsform beibehalten.

Die zweite Relation die nötig ist, um ein XML-Schema zu formalisieren, benötigt für komplexe Typen, die Zeichenketten im Inhalt zulassen, die folgende Definition der Hilfsfunktion *mixed*. Sie erzeugt den benötigten Heckenausdruck für den betroffenen Inhaltstyp.

Definition 4.15 (Funktion *mixed*)
Die Funktion *mixed* : *Reg* $\times \mathcal{B} \rightarrow$ *Reg* ist definiert durch:

$$\begin{aligned} \mathit{mixed}((r_1;\ldots;r_n),\mathit{true}) &= (\mathtt{string};r_1;\mathtt{string};\ldots;\mathtt{string};r_n;\mathtt{string}) \\ \mathit{mixed}((r_1|\ldots|r_n),\mathit{true}) &= (r_1|\ldots|r_n|\mathtt{string}) \\ \mathit{mixed}(r,\mathit{false}) &= r \end{aligned}$$

mit $r, r_1, \ldots, r_n \in$ *Reg*. □

Bei regulären Ausdrücken, die aus einer Konkatenation bestehen, kann Text zwischen sämtlichen Teilausdrücken auftreten, allerdings nur auf der obersten Ebene. Für reguläre Vereinigungen wird der Text als zusätzliche Alternative zugelassen. Damit kann die Relation, die mit Produktionsregeln formalisiert wird, angegeben werden.

Definition 4.16 (Formalisierung mittels Produktionenrelation)
Die *Produktionenrelation* $\mapsto$ ist durch folgende Regeln definiert:

$$\frac{i \hookrightarrow e, t \hookrightarrow r}{\begin{array}{l}\texttt{<element } ag\texttt{/>} \\ \texttt{@name} = i, \texttt{@type} = t\end{array} \quad \mapsto e \rightarrow e[r]} \qquad (\text{ELEM1})$$

$$\frac{i \hookrightarrow e, cm \hookrightarrow r}{\texttt{<element } ag\texttt{>}cm\texttt{</element>} \quad \texttt{@name} = i \;\mapsto e \to e[r]} \quad \text{(ELEM2)}$$

$$\frac{i \hookrightarrow o, cm \hookrightarrow r}{\texttt{<group } ag\texttt{>}cm\texttt{</group>} \quad \texttt{@name} = i \;\mapsto o \to r} \quad \text{(GROUP)}$$

$$\frac{i \hookrightarrow o, cm \hookrightarrow r_{cm}, am \hookrightarrow r_{am}}{\texttt{<complexType } ag\texttt{>}cm\ am\texttt{</complexType>} \quad \texttt{@name} = i, \texttt{@mixed} = b \;\mapsto o \to r_{am}; \textit{mixed}(r_{cm}, b)} \quad \text{(COMPL)}$$

$$\frac{i \hookrightarrow o, t \hookrightarrow r_{cm}, am \hookrightarrow r_{am}}{\begin{array}{l}\texttt{<complexType } ag\texttt{><simpleContent>}\\ \quad\texttt{<extension base="}t\texttt{">}am\texttt{</extension>}\\ \texttt{</simpleContent></complexType>}\\ \texttt{@name} = i\end{array} \quad \mapsto o \to r_{am}; r_{cm}} \quad \text{(SIMPL)}$$

$$\frac{\begin{array}{l} de_1 \mapsto n_1 \to r_1 \text{ mit } T_1, N_1 \text{ und } P_1,\\ \vdots\\ de_k \mapsto n_k \to r_k \text{ mit } T_k, N_k \text{ und } P_k\end{array}}{\texttt{<schema } ag\texttt{>}de_1 \ldots de_k\texttt{</schema>} \mapsto G = (\cup T_i, \cup N_i, (n_1| \ldots |n_k), \cup P_i) \text{ mit } 1 \leq i \leq k} \quad \text{(SCHEMA)}$$

mit $b \in \mathcal{B}$, $k \in \mathcal{N}$, $e, o, n_1, \ldots, n_k \in N$, $e \in E$ und $r, r_{cm}, r_{am}, r_1, \ldots, r_k \in \textit{Reg}$. □

Durch die Regeln ELEM1 und ELEM2 werden für Elementdeklarationen, die innerhalb von komplexen Typen lokal oder auf obersten Ebene global auftreten können, Produktionen erzeugt. Dabei muss zwischen einer Deklaration mit Typreferenz und einer Deklaration mit anonymen Typen unterschieden werden. Für Gruppen- und Elementdefinitionen werden mit GROUP, COMPL und SIMPL ebenfalls eigene Produktionen generiert. Mit der letzten Regel SCHEMA wird aus den einzelnen Produktionen, die durch die Transformationen der einzelnen Komponenten der Sprachbeschreibung gewonnen wurden, eine Heckengrammatik aufgebaut.

Anhand eines Beispiels wird die Formalisierung eines XML-Schemas verdeutlicht.

Beispiel 4.3
Dieses Beispiel zeigt die Formalisierung des komplexen Typs `t_items` aus dem XML-Schema SIF (Abschnitt 2.1) als Produktion einer regulären Heckengrammatik durch die obigen Relatio-

nen und Regeln. Zur Vereinfachung der Darstellung soll gelten:

$ed_1 =$ `<element name="article" type="integer" minOccurs="0" maxOccurs="unbounded"/>`

$ed_2 =$ `<element name="description" type="string" minOccurs="0"/>`

Abbildung 4.1 zeigt nun den vollständigen Transformationsbaum, der bei der Generierung der Produktion entsteht. Es wird deutlich, dass neben der Produktion für *t_items* noch Produktionen

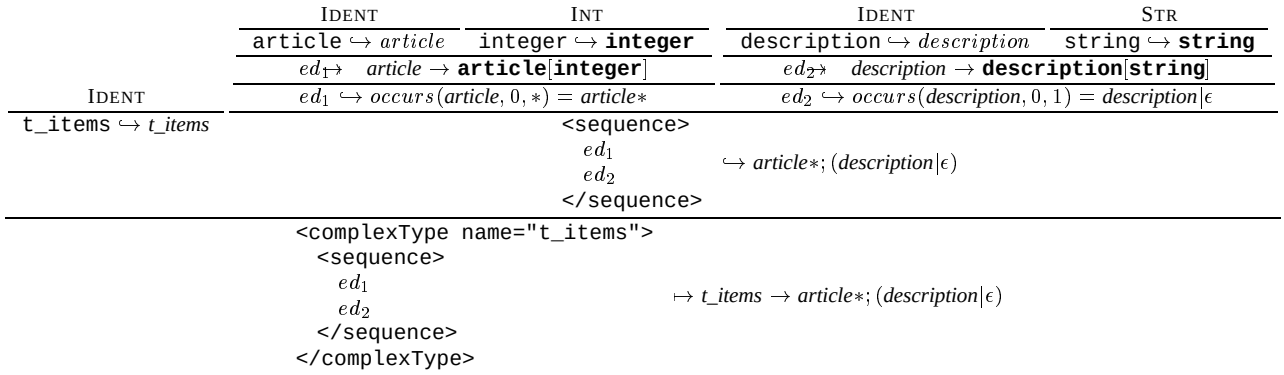

Abbildung 4.1: Formalisierung einer Schemakomponente

für die Elementtypen *article* und *description* entstehen. Insgesamt ergibt sich die Grammatik $G = (B \cup E, N, (\textit{shopRequest}|\textit{shopResponse}), P)$ mit:

$$
\begin{aligned}
B &= \{\textbf{integer}, \textbf{t\_request}\},\\
E &= \{\textbf{shopRequest}, \textbf{shopResponse}, \textbf{shoppingCart}, \textbf{account}, \textbf{add},\\
&\quad \textbf{remove}, \textbf{get}, \textbf{request}, \textbf{items}, \textbf{article}, \textbf{description}\},\\
N &= \{\textit{t\_shopRequest}, \textit{t\_cartRequest}, \textit{t\_shopResponse}, \textit{t\_cartResponse},\\
&\quad \textit{t\_items}, \textit{shopRequest}, \textit{shopResponse}, \textit{t\_shopRequest/shoppingCart},\\
&\quad \textit{t\_shopResponse/shoppingCart}, \textit{account}, \textit{add}, \textit{remove}, \textit{get}, \textit{request},\\
&\quad \textit{items}, \textit{article}, \textit{description}\}\text{und}\\
P &= \{ \quad \textit{t\_shopRequest} \to \textit{t\_shopRequest/shoppingCart},\\
&\qquad \textit{t\_cartRequest} \to (\textit{account}; (\textit{add}|\textit{remove}|\textit{get})),\\
&\qquad \textit{t\_shopResponse} \to \textit{t\_shopResponse/shoppingCart},\\
&\qquad \textit{t\_cartResponse} \to (\textit{account}; \textit{request}; (\textit{items}|\epsilon)),\\
&\qquad \textit{t\_items} \to (\textit{article}*; (\textit{description}|\epsilon)),\\
&\qquad \textit{shopRequest} \to \textbf{shopRequest}[\textit{t\_shopRequest}],\\
&\qquad \textit{t\_shopRequest/shoppingCart} \to \textbf{shoppingCart}[\textit{t\_cartRequest}],\\
&\qquad \textit{account} \to \textbf{account}[\textbf{integer}],
\end{aligned}
$$

$$
\begin{aligned}
\mathit{add} &\rightarrow \texttt{add}[\texttt{integer}],\\
\mathit{remove} &\rightarrow \texttt{remove}[\texttt{integer}],\\
\mathit{get} &\rightarrow \texttt{get}[\epsilon],\\
\mathit{shopResponse} &\rightarrow \texttt{shopResponse}[\mathit{t\_shopResponse}],\\
\mathit{t\_shopResponse/shoppingCart} &\rightarrow \texttt{shoppingCart}[\mathit{t\_cartResponse}],\\
\mathit{request} &\rightarrow \texttt{request}[\texttt{integer}],\\
\mathit{items} &\rightarrow \texttt{items}[\mathit{t\_items}],\\
\mathit{article} &\rightarrow \texttt{article}[\texttt{integer}],\\
\mathit{description} &\rightarrow \texttt{description}[\texttt{string}]\}.^{4}
\end{aligned}
$$

□

XML-Schema bietet sehr weitreichende Möglichkeiten, um die Dokumente einer Auszeichnungssprache zu beschreiben. Aufgrund dieses beträchtlichen Umfangs konzentriert sich diese Arbeit auf den grundlegenden Kern. Nicht betrachtet werden in dieser Arbeit Inhaltsmodelle vom Typ `any`, Einschränkungen einfacher Typen („simple type restriction“), Attributgruppen, Elementtypen mit `nil`-Werten (Attribut `nillable`), Verhinderung von Typsubstitution (Attribut `block`), Erzwingen einer Ableitung durch abstrakte komplexe Typen (Attribut `type`) und abstrakte Elementdeklarationen („substitution group“), Verhinderung von Ableitungen (Attribut `final`) sowie alle weiteren Nebenbedingungen (Elemente `unique`, `key`, `keyref`).

4.4 Typinferenz für XML-Konstruktoren

In diesem Abschnitt wird die Typinferenz definiert, die notwendig ist, um für einen XML-Konstruktor den zugehörigen Typen zu ermitteln. Wie bereits oben beschrieben, wird bei der Typanalyse eines XOBE-Programms der Typ sämtlicher XML-Konstruktoren inferiert. Da alle Variablen vor der Verwendung deklariert sein müssen, gestaltet sich die Typinferenz als nicht besonders schwierig. Die formale Beschreibung besteht zunächst aus einem Typisierungsurteil.

Definition 4.17 (Typisierungsurteil für XML-Konstruktor)
Gegeben sei eine Menge von Variablendeklarationen Γ, dann wird das folgende Typisierungsurteil definiert:

$$\Gamma \vdash c : r \qquad\qquad c \text{ ist ein wohlgeformter XML-Konstruktor vom Typ } r \text{ in } \Gamma$$

Dabei sei $r \in \mathit{Reg}$. □

Mit den anschließenden Typisierungsregeln kann der Typ eines XML-Konstruktors berechnet werden. Die Definition verwendet die Operation *lexType*, die den von der lexikalischen Analyse identifizierten Basisdatentypen liefert.

[4] Der Elementtyp **`shoppingCart`** ist durch zwei unterschiedlich lokale Deklarationen definiert, die durch eindeutige Bezeichner unterschieden sind.

Definition 4.18 (Typinferenz XML-Konstruktor)
Der *Typ eines* XML*-Konstruktors* sei definiert durch folgende Inferenzregeln:

$$\frac{}{\Gamma \vdash va : \mathit{lexType}(va)} \quad \text{(VAL)}$$

$$\frac{\Gamma \vdash va : r, id \hookrightarrow e}{\Gamma \vdash id\texttt{=}va : @e[r]} \quad \text{(ATTR)}$$

$$\frac{\begin{array}{c}\Gamma \vdash at_1 : r_1 \\ \vdots \\ \Gamma \vdash at_n : r_n\end{array}}{\Gamma \vdash at_1 \ldots at_n : r_1; \ldots; r_n} \quad \text{(ATTRS)}$$

$$\frac{id : t \in \Gamma}{\Gamma \vdash \{id\} : t} \quad \text{(VAR)}$$

$$\frac{}{\Gamma \vdash cd : \mathit{lexType}(cd)} \quad \text{(DATA)}$$

$$\frac{\begin{array}{c}id \hookrightarrow e \\ \Gamma \vdash am : r_{am} \\ \Gamma \vdash cm : r_{cm}\end{array}}{\Gamma \vdash \texttt{<}id\ am\texttt{>}cm\texttt{</}id\texttt{>} : e[r_{am}; r_{cm}]} \quad \text{(ELEM)}$$

$$\frac{\Gamma \vdash am : r, id \hookrightarrow e}{\Gamma \vdash \texttt{<}id\ am\texttt{/>} : e[r]} \quad \text{(EMPTY)}$$

$$\frac{\begin{array}{c}\Gamma \vdash cm_1 : r_1 \\ \vdots \\ \Gamma \vdash cm_n : r_n\end{array}}{\Gamma \vdash cm_1 \ldots cm_n : (r_1; \ldots; r_n)} \quad \text{(CONC)}$$

mit $t \in B \cup N$, $e \in E$, $r, r_1, \ldots, r_n \in \mathit{Reg}$ und der Ausdrucksrelation $\hookrightarrow$. □

Mit den Regeln VAL, ATTR und ATTRS wird ermittelt, wie sich der Typ eines Attributwertes, eines Attributs und einer Attributliste zusammensetzt. Die Typen von Variablen, die bereits durch die Deklaration der Variablen bekannt sind, werden innerhalb eines XML-Konstruktors durch die Regel VAR eingesetzt. Dabei wird nicht zwischen Variablen der Programmiersprache Java und Variablen von XML-Objekt-Klassen unterschieden.

Im Inhalt eines Elements können neben Elementen auch Zeichendaten auftreten, deren Typ durch die Regel DATA mit der Operation *lexType* bestimmt wird. Die Regel ELEM ermittelt den Typen

für ein XML-Element mit nicht leerem Inhalt, während mit EMPTY dies – ganz analog – für die abkürzende Schreibweise der leeren Elemente geschieht. Die letzte Regel CONC inferiert den zusammengesetzten Typ für die Konkatenation von Elementen oder Zeichendaten, wie sie im Inhalt von Elementen auftreten kann.

Mit dem folgenden Beispiel wird die Arbeitsweise der Typinferenz illustriert.

Beispiel 4.4
In der Methode `addArticle` des Beispiels aus Abschnitt 3.3 wird ein XML-Objekt mittels einer `return`-Anweisung zurückgegeben. Der Typ der Variable `accountNr` wurde zuvor als `int` deklariert.

```
1 return <shopResponse>
2            <shoppingCart>
3                <account>{this.accountNr}</account>
4                <request>processed </request>
5            </shoppingCart>
6        </shopResponse>;
```

Für diesen XML-Konstruktor kann nun anhand der definierten Regeln der aktuelle Typ inferiert werden. In der Abbildung 4.2 wird die Berechnung für die inneren beiden Elemente dargestellt.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \texttt{\{this.accountNr\}} : \textbf{integer}}\,\textsc{Var}}{\Gamma \vdash \texttt{<account>\{this.accountNr\}</account>} : \textbf{account}[\textbf{integer}]} \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash \texttt{processed} : \textbf{string}}\,\textsc{Data}}{\Gamma \vdash \texttt{<request>processed</request>} : \textbf{request}[\textbf{string}]}}{\Gamma \vdash \begin{array}{l}\texttt{<account>\{this.accountNr\}</account>}\\ \texttt{<request>processed</request>}\end{array} : \textbf{account}[\textbf{integer}]; \textbf{request}[\textbf{string}]}$$

Abbildung 4.2: Berechnung der Typinferenz

Für den gesamten XML-Konstruktor ergibt sich dann folgender Typ:

$$\textbf{shopResponse}[\textbf{shoppingCart}[\textbf{account}[\textbf{integer}]; \textbf{request}[\textbf{string}]]]$$

□

4.5 Typinferenz für XPath-Ausdrücke

Dieser Abschnitt formalisiert die Typinferenz für XPath-Ausdrücke in XOBE-Programmen. Die Typanalyse benötigt für die Überprüfung der Typkorrektheit von XOBE-Programmen die Typen der XPath-Ausdrücke, die anhand von Regeln inferiert werden können. Zur Verfügung dafür steht analog zur Typinferenz der XML-Konstruktoren eine Umgebung mit den Typen der Variablen, da alle Variablen im Programm vor der Verwendung deklariert sein müssen. Mit einem Typisierungsurteil wird die Typinferenz von XPath-Ausdrücken formal beschrieben.

Definition 4.19 (Typisierungsurteil für XPath-Ausdrücke)
Gegeben sei eine Menge von Variablendeklarationen Γ, dann ist das folgende Typisierungsurteil definiert:

$$\Gamma \vdash x : r \qquad\qquad x \text{ ist ein wohlgeformter XPath-Ausdruck vom Typ } r \text{ in } \Gamma$$

Dabei sei $r \in$ *Reg*. □

Anmerkung: Für die Typen der XPath-Ausdrücke gilt die Besonderheit, dass es sich um Typen der Form $(e_1[r_1]|\ldots|e_n[r_n])*$ mit $e_i \in E$ und $r_i \in$ *Reg* handelt. Die Ursache dafür liegt in der Definition von XPath deren Ausdrücke stets eine Liste von XML-Knoten (eigentlich eine Knotenmenge) zurückgeben. Im Weiteren wird die reguläre Vereinigung von Elementtypen $e_1[r_1]|\ldots|e_n[r_n]$ innerhalb des Kleene-Stern-Operators auch mit *XPath-Typ* bezeichnet.[5]

Um die Typinferenz für XPath-Ausdrücke zu definieren werden einige Hilfsfunktionen benötigt. Die erste Hilfsfunktion *nodeTest*, die angegeben wird, ermittelt den XPath-Typen für einen Knotentest. Anschließend folgt mit *self*, *child*, *attribute*, *descendant*, *parent*, *ancestor*, *followingSibling* und *precedingSibling* für jede Achse in XPath eine weitere Hilfsfunktion.

Definition 4.20 (Funktion *nodeTest*)
Die Funktion *nodeTest* : $E \times$ *Reg* $\rightarrow$ *Reg* liefert für einen Elementnamen $e \in E$ die Typen aus dem XPath-Typ $r \in$ *Reg*, die den Elementnamen e haben. Die Definition ist rekursiv:

$$\begin{aligned} \textit{nodeTest}_e(\emptyset) &= \emptyset \\ \textit{nodeTest}_e(f[r]) &= \begin{cases} e[r] & \text{falls } e = f \\ \emptyset & \text{sonst} \end{cases} \\ \textit{nodeTest}_e(r|s) &= \textit{nodeTest}_e(r)|\textit{nodeTest}_e(s) \end{aligned}$$

mit $e, f \in E$ und $r, s \in$ *Reg*. □

Die Funktion *nodeTest* selektiert aus einem XML-Typen genau die Elementtypen, die den angegebenen Elementnamen aufweisen.

Mit der Funktion *self* wird der XPath-Typ für die Selbst-Achse ermittelt.

Definition 4.21 (Funktion *self*)
Die Funktion *self* : *Reg* $\rightarrow$ *Reg* liefert den XPath-Typ eines regulären Ausdrucks $r \in$ *Reg* und wird durch die zweistellige Hilfsfunktion *self* definiert:

$$\textit{self}(r) = \textit{self}(r, \{\})$$

[5] Um XPath-Typen möglichst kompakt darzustellen, werden Wiederholungen wie $t|t$ implizit aufgelöst zu t.

Die Hilfsfunktion *self* : *Reg* $\times \mathcal{P}[N] \rightarrow$ *Reg* ist rekursiv definiert durch:

$$
\begin{aligned}
\mathit{self}(\emptyset, N_v) &= \emptyset \\
\mathit{self}(\epsilon, N_v) &= \emptyset \\
\mathit{self}(b, N_v) &= \emptyset \\
\mathit{self}(n, N_v) &= \begin{cases} \emptyset & \text{falls } n \in N_v, \\ \mathit{self}(r, N_v \cup \{n\}) \text{ mit } n \rightarrow r \in P & \text{sonst} \end{cases} \\
\mathit{self}(e[r], N_v) &= \begin{cases} e[r] & \text{falls } e \neq \texttt{@}f, \\ \emptyset & \text{sonst} \end{cases} \\
\mathit{self}(r|s, N_v) &= \mathit{self}(r, N_v)|\mathit{self}(s, N_v) \\
\mathit{self}(r; s, N_v) &= \mathit{self}(r, N_v)|\mathit{self}(s, N_v) \\
\mathit{self}(r*, N_v) &= \mathit{self}(r, N_v)
\end{aligned}
$$

mit $b \in B$, $n \in N$, $e, \texttt{@}f \in E$,$r, s \in$ *Reg* und $N_v \subseteq N$. □

Die Hilfsfunktion *self* ermittelt für den angegebenen regulären Ausdruck, die im Ausdruck enthaltenen Elementtypen. Diese werden in Form einer regulären Vereinigung kombiniert, wie es für XPath-Typen notwendig ist. Dafür wird in einer Menge N_v von Nichtterminalsymbolen notiert, welche Nichtterminalsymbole bereits bearbeitet wurden. Bei noch nicht behandelten Nichtterminalsymbolen erfolgt eine rekursive Anwendung der Funktion auf die Definition der Produktionsregel, während bei einem schon bekannten Nichtterminalsymbol der Rekursionszweig beendet wird. Auf diese Art und Weise ist es möglich, Endlosschleifen zu vermeiden. Attributtypen bleiben genauso wie Basisdatentypen, reguläre Konkatenation und Kleene-Stern unberücksichtigt.

Die Funktion *child* berechnet die XPath-Typen zur Kind-Achse.

Definition 4.22 (Funktion *child*)
Die Funktion *child* : *Reg* $\rightarrow$ *Reg* liefert die XPath-Typen der Kinderknoten eines XPath-Typs $r \in$ *Reg* und ist rekursiv definiert:

$$
\begin{aligned}
\mathit{child}(\emptyset) &= \emptyset \\
\mathit{child}(e[r]) &= \mathit{self}(r) \\
\mathit{child}(r|s) &= \mathit{child}(r)|\mathit{child}(s)
\end{aligned}
$$

mit $e \in E$ und $r, s \in$ *Reg*. □

Die Funktion *child* ist für XPath-Typen rekursiv definiert und berechnet mit Hilfe der Funktion *self* die Elementtypen der Kinderknoten des übergebenen XPath-Typen.

Um die XPath-Typen der Attribut-Achse zu ermitteln, wird die Funktion *attribute* definiert.

Definition 4.23 (Funktion *attribute*)
Die Funktion *attribute* : *Reg* $\rightarrow$ *Reg* liefert die XPath-Typen der Attribute eines XPath-Typs

$r \in$ *Reg* und ist rekursiv definiert:

$$\begin{aligned} \textit{attribute}(\emptyset) &= \emptyset \\ \textit{attribute}(e[r]) &= \textit{attribute}(r, \{\}) \\ \textit{attribute}(r|s) &= \textit{attribute}(r)|\textit{attribute}(s) \end{aligned}$$

mit $e \in E$ und $r, s \in$ *Reg*. Die Hilfsfunktion *attribute* : *Reg* $\times \mathcal{P}(N) \rightarrow$ *Reg* ist bis auf folgende Ausnahme analog zur Hilfsfunktion *self* : *Reg* $\times \mathcal{P}(N) \rightarrow$ *Reg* definiert:

$$\textit{attribute}(e[r], N_v) = \begin{cases} e[r] & \text{falls } e = \texttt{@}f, \\ \emptyset & \text{sonst} \end{cases}$$

mit $e, \texttt{@}f \in E$, $r \in$ *Reg* und $N_v \subseteq N$. □

Die Attributtypen eines regulären Ausdrucks werden mit der zweistelligen Hilfsfunktion *attribute* extrahiert. Darauf aufbauend können die Attribute eines XML-Typen durch die Funktion *attribute* rekursiv ermittelt werden.

Mit der Funktion *descendant* werden die Typen gemäß der Nachfahr-Achse berechnet.

Definition 4.24 (Funktion *descendant*)
Die Funktion *descendant* : *Reg* $\rightarrow$ *Reg* liefert XPath-Typen der Nachfahren eines regulären Ausdrucks $r \in$ *Reg* und ist rekursiv definiert:

$$\begin{aligned} \textit{descendant}(\emptyset) &= \emptyset \\ \textit{descendant}(e[r]) &= \textit{descendantOrSelf}(r, \{\}) \\ \textit{descendant}(r|s) &= \textit{descendant}(r)|\textit{descendant}(s) \end{aligned}$$

für alle $e \in E$ und $r, s \in$ *Reg*. Die Hilfsfunktion *descendantOrSelf* : *Reg* $\times \mathcal{P}(N) \rightarrow$ *Reg* ist bis auf nachstehende Ausnahme analog zur Hilfsfunktion *self* : *Reg* $\times \mathcal{P}(N) \rightarrow$ *Reg* definiert:

$$\textit{descendantOrSelf}(e[r], N_v) = \begin{cases} \textit{descendantOrSelf}(r, N_v)|e[r] & \text{falls } e \neq \texttt{@}f, \\ \textit{descendantOrSelf}(r, N_v) & \text{sonst} \end{cases}$$

mit $e, \texttt{@}f \in E$, $r \in$ *Reg* und $N_v \subseteq N$. □

Ähnlich zur Definition der Funktion *attribute* wird die Funktion *descendant* konstruiert. Der Unterschied liegt darin, dass für die Achse der Nachfahren nun auch die Inhalte der Elementtypen rekursiv betrachtet werden.

Für die Eltern-Achse wird die Funktion *parent* festgelegt.

Definition 4.25 (Funktion *parent*)
Die Funktion *parent* : *Reg* $\rightarrow$ *Reg* liefert XPath-Typen der Elternknoten eines regulären Aus-

drucks $r \in$ *Reg* und ist durch eine vierstellige Hilfsfunktion *parent* definiert:

$$\textit{parent}(t) = \bigm| _{n_i \in N} \textit{parent}(t, N_s, \emptyset, r_i) \text{ mit } n_i \to r_i \in P$$
$$\text{und } N_s = \{n | t \leq \textit{self}(r) \text{ mit } n \to r \in P\}$$

Die Hilfsfunktion *parent* : *Reg* $\times \mathcal{P}(N) \times$ *Reg* $\times$ *Reg* $\to$ *Reg* ist rekursiv definiert wie folgt:

$$\begin{aligned}
\textit{parent}(t, N_s, p, \emptyset) &= \emptyset \\
\textit{parent}(t, N_s, p, \epsilon) &= \emptyset \\
\textit{parent}(t, N_s, p, b) &= \emptyset \\
\textit{parent}(t, N_s, p, n) &= \begin{cases} p & \text{falls } n \in N_s \\ \emptyset & \text{sonst} \end{cases} \\
\textit{parent}(t, N_s, p, e[r]) &= \begin{cases} \textit{parent}(t, N_s, e[r], r) | p & \text{falls } t \leq e[r] \\ \textit{parent}(t, N_s, e[r], r) & \text{sonst} \end{cases} \\
\textit{parent}(t, N_s, p, r|s) &= \textit{parent}(t, N_s, p, r) | \textit{parent}(t, N_s, p, s) \\
\textit{parent}(t, N_s, p, (r; s)) &= \textit{parent}(t, N_s, p, r) | \textit{parent}(t, N_s, p, s) \\
\textit{parent}(t, N_s, p, r*) &= \textit{parent}(t, N_s, p, r)
\end{aligned}$$

mit $b \in B$, $n \in N$, $e \in E$, $p, r, s, t \in$ *Reg* und $N_s \subseteq N$. □

Die Funktion *parent* arbeitet mit Unterstützung der Hilfsfunktion *parent*, deren Aufruf mit vier Parametern erfolgt. Der Parameter t ist dabei der Elementtyp, von dem die Elementtypen der Eltern gesucht werden. Die Menge N_s besteht aus den Nichtterminalsymbolen, die t als Subtyp im XPath-Typ der Selbst-Achse einschließt. Der Parameter p akkumuliert den aktuellen Elterntyp und r steht für den gerade betrachteten regulären Heckenausdruck.

Die Idee der Berechnung ist, dass für jedes Nichtterminalsymbol in der Heckengrammatik überprüft wird, ob dessen Produktion den übergebenen Elementtyp als Subtyp enthält. Wird dann in der Betrachtung des regulären Ausdrucks ein solches Nichtterminalsymbol gefunden, wird der akkumulierte Elterntyp zurückgegeben. Nichtterminalsymbole werden demnach nicht rekursiv behandelt. Stattdessen wird getestet, ob ein Nichtterminalsymbol in Menge N_s ist, die vorab erstellt wurde.

Für die Vorfahr-Achse wird die Funktion *ancestor* angegeben.

Definition 4.26 (Funktion *ancestor*)
Die Funktion *ancestor* : *Reg* $\to$ *Reg* liefert XPath-Typen der Vorfahrknoten eines regulären Ausdrucks $r \in$ *Reg* und ist durch eine vierstellige Hilfsfunktion *ancestor* definiert:

$$\textit{ancestor}(t) = \bigm| _{n_i \in N} \textit{ancestor}(t, N_s, \emptyset, r_i) \text{ mit } n_i \to r_i \in P$$
$$\text{und } N_s = \{n | t \leq \textit{self}(r) \text{ mit } n \to r \in P\}$$

Die Hilfsfunktion *ancestor* : *Reg* $\times \mathcal{P}(N) \times$ *Reg* $\times$ *Reg* $\rightarrow$ *Reg* ist bis auf folgende Ausnahme analog zur Hilfsfunktion *parent* definiert:

$$\textit{ancestor}(t, N_s, a, e[r]) \quad = \quad \begin{cases} \textit{ancestor}(t, N_s, e[r], r)|a & \text{falls } t \leq e[r] \\ \textit{ancestor}(t, N_s, a|e[r], r) & \text{sonst} \end{cases}$$

mit $e \in E$, $a, r, t \in$ *Reg* und $N_s \subseteq N$. □

Die Definition der Funktion *ancestor* entspricht der Idee, die bei der Realisierung der Funktion *parent* verfolgt wurde. Der Unterschied liegt darin, dass in der vierstelligen Hilfsfunktion *ancestor* nicht nur der aktuelle Elterntyp akkumuliert wird, sondern der Typ sämtlicher Vorfahren.

Auch für die Berechnung des Typs der Nachfolgende-Geschwister-Achse wird eine Funktion festgelegt.

Definition 4.27 (Funktion *followingSibling*)
Die Funktion *followingSibling* : *Reg* $\rightarrow$ *Reg* liefert XPath-Typen der nachfolgenden Geschwisterknoten eines regulären Ausdrucks $r \in$ *Reg* und ist durch eine vierstellige Hilfsfunktion *followingSibling* definiert:

$$\textit{followingSibling}(t) = \bigm| \limits_{n_i \in N} \textit{followingSibling}(t, N_s, \emptyset, r_i) \text{ mit } n_i \rightarrow r_i \in P \text{ und } N_s = \{n | t \leq \textit{self}(r) \text{ mit } n \rightarrow r \in P\}$$

Die Hilfsfunktion *followingSibling* : *Reg* $\times \mathcal{P}(N) \times$ *Reg* $\times$ *Reg* $\rightarrow$ *Reg* ist rekursiv definiert wie folgt:

$$\begin{aligned}
\textit{followingSibling}(t, N_s, f, \emptyset) &= \emptyset \\
\textit{followingSibling}(t, N_s, f, \epsilon) &= \emptyset \\
\textit{followingSibling}(t, N_s, f, b) &= \emptyset \\
\textit{followingSibling}(t, N_s, f, n) &= \begin{cases} f & \text{falls } n \in N_s \\ \emptyset & \text{sonst} \end{cases} \\
\textit{followingSibling}(t, N_s, f, e[r]) &= \begin{cases} \textit{followingSibling}(t, N_s, \emptyset, r)|f & \text{falls } t \leq e[r] \\ \textit{followingSibling}(t, N_s, \emptyset, r) & \text{sonst} \end{cases} \\
\textit{followingSibling}(t, N_s, f, r|s) &= \textit{followingSibling}(t, N_s, f, r)| \\
&\quad \textit{followingSibling}(t, N_s, f, s) \\
\textit{followingSibling}(t, N_s, f, (r; s)) &= \textit{followingSibling}(t, N_s, f|\textit{self}(s), r)| \\
&\quad \textit{followingSibling}(t, N_s, f, s) \\
\textit{followingSibling}(t, N_s, f, r*) &= \textit{followingSibling}(t, N_s, f|\textit{self}(r), r)
\end{aligned}$$

mit $b \in B$, $n \in N$, $e \in E$, $f, r, s, t \in$ *Reg* und $N_s \subseteq N$. □

Die Grundidee der vierstelligen Hilfsfunktion *followingSibling*, mit der die Funktion *following-Sibling* definiert ist, folgt ebenfalls wieder der Hilfsfunktion *parent*. Diesmal werden aber die Typen der folgenden Geschwisterknoten in dem Parameter f akkumuliert.

Es folgt ein Beispiel, das die Funktion *followingSibling* auf einen regulären Ausdruck anwendet.

Beispiel 4.5
Dieses Beispiel ermittelt den Typen der folgenden Geschwisterknoten für den regulären Ausdruck $t = \texttt{title}[\texttt{string}]$ gemäß der durch die Sprachbeschreibung AOML (Beispiel 2.2) induzierten Heckengrammatik. Für die Menge N_s ergibt sich $N_s = \{\mathit{title}\}$. Nun wird die vierstellige Hilfsfunktion *followingSibling* für jedes Nichtterminalsymbol berechnet, beginnend mit *aoml*:

$$\begin{aligned}\mathit{followingSibling}(t, N_s, \emptyset, \texttt{aoml}[\mathit{antiquary}; \mathit{offer}]) \\ &= \mathit{followingSibling}(t, N_s, \emptyset, \mathit{antiquary}; \mathit{offer}) \\ &= \mathit{followingSibling}(t, N_s, \mathit{self}(\mathit{offer}), \mathit{antiquary}) \\ &\quad | \ \mathit{followingSibling}(t, N_s, \emptyset, \mathit{offer}) \\ &= \emptyset\end{aligned}$$

Analoge Resultate ergeben sich für die Nichtterminalsymbole *antiquary, name, address, email, offer, author, condition, artist, price* und *fields*. Für die Nichtterminalsymbole *book* und *record* werden dagegen relevante Typen gefunden:

$$\begin{aligned}\mathit{followingSibling}(t, N_s, \emptyset, \texttt{book}[\mathit{title}; (\mathit{author}|\epsilon); \mathit{fields}]) \\ &= \mathit{followingSibling}(t, N_s, \emptyset, \mathit{title}; (\mathit{author}|\epsilon); \mathit{fields}) \\ &= \mathit{followingSibling}(t, N_s, \mathit{self}((\mathit{author}|\epsilon); \mathit{fields}), \mathit{title}) \\ &\quad |\mathit{followingSibling}(t, N_s, \mathit{self}(\mathit{fields}), \mathit{author}|\epsilon) \\ &\quad |\mathit{followingSibling}(t, N_s, \emptyset, \mathit{fields}) \\ &= \mathit{self}((\mathit{author}|\epsilon); \mathit{fields})\end{aligned}$$

$$\mathit{followingSibling}(t, N_s, \emptyset, \texttt{record}[\mathit{title}; \mathit{artist}; \mathit{fields}]) = \mathit{self}(\mathit{artist}; \mathit{fields})$$

Der Typ *title* kann dagegen nicht nach t stehen, was sich in diesem Aufruf zeigt:

$$\begin{aligned}\mathit{followingSibling}(t, N_s, \emptyset, \texttt{title}[\texttt{string}]) &= \mathit{followingSibling}(t, N_s, \emptyset, \texttt{string}) \\ &= \emptyset\end{aligned}$$

Insgesamt erhält man das Resultat:

$$\mathit{followingSibling}(t) = \mathit{self}(\mathit{author})|\mathit{self}(\mathit{artist})|\mathit{self}(\mathit{fields})$$

□

Die Definition der Funktion *precedingSibling* für die Vorherige-Geschwister-Achse erfolgt analog zur vorherigen Funktion *followingSibling*.

Definition 4.28 (Funktion *precedingSibling*)
Die Funktion *precedingSibling* : *Reg* $\rightarrow$ *Reg* liefert XPath-Typen der zuvor stehenden Geschwisterelemente eines regulären Ausdrucks $r \in$ *Reg* und ist analog zu *followingSibling* definiert. □

Mit diesen Hilfsfunktionen kann mit den anschließenden Regeln festgelegt werden, welche Typen für einen XPath-Ausdruck abzuleiten sind.

Definition 4.29 (Typinferenz XPath-Ausdrücke)
Der *Typ eines XPath-Ausdrucks* sei definiert durch folgende Inferenzregeln:

$$\frac{va : r \in \Gamma}{\Gamma \vdash va : \mathit{self}(r)*} \quad \text{(VAR)}$$

$$\frac{\Gamma \vdash ls : r*}{\Gamma \vdash ls\texttt{/child} : \mathit{child}(r)*} \quad \text{(CHILD)}$$

$$\frac{\Gamma \vdash ls : r*}{\Gamma \vdash ls\texttt{/descendant} : \mathit{descendant}(r)*} \quad \text{(DESC)}$$

$$\frac{\Gamma \vdash ls : r*}{\Gamma \vdash ls\texttt{/parent} : \mathit{parent}(r)*} \quad \text{(PAR)}$$

$$\frac{\Gamma \vdash ls : r*}{\Gamma \vdash ls\texttt{/ancestor} : \mathit{ancestor}(r)*} \quad \text{(ANC)}$$

$$\frac{\Gamma \vdash ls : r*}{\Gamma \vdash ls\texttt{/following\_sibling} : \mathit{followingSibling}(r)*} \quad \text{(FOLL SIB)}$$

$$\frac{\Gamma \vdash ls\texttt{/ancestor-or-self} : r*}{\Gamma \vdash ls\texttt{/following} : \mathit{descendant}(\mathit{followingSibling}(r))*} \quad \text{(FOLL)}$$

$$\frac{\Gamma \vdash ls : r*}{\Gamma \vdash ls\texttt{/preceding\_sibling} : \mathit{precedingSibling}(r)*} \quad \text{(PREC SIB)}$$

$$\frac{\Gamma \vdash ls\texttt{/ancestor-or-self} : r*}{\Gamma \vdash ls\texttt{/preceding} : \mathit{descendant}(\mathit{precedingSibling}(r))*} \quad \text{(PREC)}$$

$$\frac{\Gamma \vdash ls\texttt{/descendant} : r_1*, \Gamma \vdash ls\texttt{/self} : r_2*}{\Gamma \vdash ls\texttt{/descendant-or-self} : (r_1|r_2)*} \quad \text{(DESCOS)}$$

$$\frac{\Gamma \vdash ls\texttt{/ancestor} : r_1*, \Gamma \vdash ls\texttt{/self} : r_2*}{\Gamma \vdash ls\texttt{/ancestor-or-self} : (r_1|r_2)*} \quad \text{(ANCOS)}$$

$$\frac{\Gamma \vdash ls : r*}{\Gamma \vdash ls\texttt{/attribute} : \mathit{attribute}(r)*} \quad \text{(ATTR)}$$

$$\frac{id \hookrightarrow e, \Gamma \vdash ls\texttt{/}ax : r*}{\Gamma \vdash ls\texttt{/}ax\texttt{::}id : \mathit{nodeTest}_e(r)*} \quad \text{(TEST)}$$

mit $e \in E$, $r, r_1, r_2 \in$ *Reg* und der Ausdrucksrelation $\hookrightarrow$. □

Für eine Variable in einem XPath-Ausdruck wird durch die Regel VAR der deklarierte Typ unter Anwendung der Hilfsfunktion *self* zu einem XPath-Typ umgeformt. Mit den Regeln CHILD, ATTR, DESC, PAR, ANC, FOLL SIB, FOLL, PREC SIB, PREC, DESCOS und ANCOS berechnen sich die XPath-Typen der Knoten, die durch die jeweiligen Achsen selektiert werden. Dabei kommen die Funktionen *child*, *attribute*, *descendant*, *parent*, *ancestor*, *followingSibling* und *precedingSibling* zur Anwendung. Der Knotentest in XPath schränkt den Typen der Knoten einer Achse ein, was die Funktion *nodeTest*, die in Regel TEST angewendet wird, ausdrückt.

Die Arbeitsweise der Typinferenz illustrieren die nachstehenden Beispiele. Das erste einfache Beispiel beschränkt sich auf Anwendung der Regeln aus Definition 4.29, die auf den eingeführten Hilfsfunktionen *nodeTest*, *self* und *child* basieren.

Beispiel 4.6
In diesem Beispiel sei die Variable `b` vom Typ `book` und gesucht wird der Typ des XPath-Ausdrucks `b/child::author` aus Beispiel 2.5 (Abschnitt 2.2). Die Abbildung 4.3 zeigt wie

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{b} : \mathit{self}(\mathit{book})*}\,\text{VAR}}{\texttt{b/child} : \mathit{child}(\textbf{book}[\mathit{title}; (\mathit{author}|\epsilon); \mathit{fields}])*}}{\texttt{b/child::author} : \mathit{nodeTest}_{\textbf{author}}((\textbf{title}[\textbf{string}]|\textbf{author}[\textbf{string}]|\textbf{article}[\textbf{string}]|\textbf{condition}[\textbf{string}]|\textbf{price}[\textbf{string}]))*}$$

Abbildung 4.3: Typinferenz eines Ausdrucks mit Kind-Achse

unter Anwendung der Regeln VAR, CHILD und TEST aus Definition 4.29 der Typ des Ausdrucks inferiert wird. Nach Auswertung der Funktion *nodeTest* in der letzten Zeile der Abbildung ergibt sich abschließend:

$$\texttt{b/child::author} : (\textbf{author}[\textbf{string}])*$$

□

Das folgende Beispiel illustriert ebenfalls die Arbeitsweise der Typinferenz.

Beispiel 4.7
Die Variable `t` sei für dieses Beispiel vom Typ `title`. Inferiert werden soll der Typ des XPath-Ausdrucks `t/parent::book`. In der Abbildung 4.4 ist der Ableitungsbaum für den Ausdruck

$$\dfrac{\dfrac{\dfrac{}{\texttt{t} : \mathit{self}(\mathit{title})*}\,\text{VAR}}{\texttt{t/parent} : \mathit{parent}(\textbf{title}[\textbf{string}])*}}{\texttt{t/parent::book} : \mathit{nodeTest}_{\textbf{book}}(\textbf{book}[\mathit{title}; (\mathit{author}|\epsilon); \mathit{fields}]|\textbf{record}[\mathit{title}; \mathit{artist}; \mathit{fields}])*}$$

Abbildung 4.4: Typinferenz eines Ausdrucks mit Eltern-Achse

dargestellt. Anwendung finden die Regel VAR, PAR und TEST aus Definition 4.29. Bei der Auswertung der Funktion *parent* in Regel PAR ergibt sich für die Menge der bezüglich dieser Achse relevanten Nichtterminalsymbole $N_s = \{\mathit{title}\}$. Dies bedeutet, dass all die Elementtypen, in

deren Inhaltsmodellen das Nichtterminalsymbol *title* auftritt, mögliche Elementtypen der Elternknoten sind. Die Auswertung der Funktion *nodeTest* in der letzten Zeile der Ableitung resultiert schließlich in der folgenden Typzuordnung:

$$\texttt{t/parent::book} : (\textbf{book}[\mathit{title}; (\mathit{author}|\epsilon); \mathit{fields}])*$$

□

4.6 Algorithmus zur Typüberprüfung

Nachdem die formalen Grundlagen eingeführt sind und die Typinferenz für XML-Konstruktoren und XPath-Ausdrücke feststeht, folgt in diesem Abschnitt die formale Definition des Algorithmus zur Überprüfung der Subtyp-Beziehung für XML-Typen.

Das Vorgehen des Algorithmus fundiert auf der algorithmischen Idee von Antimirov [Ant94] zur Überprüfung von Ungleichungen von regulären Ausdrücken. Antimirov zeigt, dass für jede ungültige reguläre Ungleichung mindestens eine reduzierte Ungleichung existiert, die trivial inkonsistent ist. Dabei ist eine reguläre Ungleichung genau dann trivial inkonsistent, wenn das leere Wort Element der Sprache, repräsentiert durch den Ausdruck der linken Seite, ist, aber nicht Element der Sprache der rechten Seite. Diese Eigenschaft lässt sich auf einfache Art überprüfen. Ebenfalls ist es nicht schwer, die reduzierten Ungleichungen zu berechnen. Da es sich aber im Falle von XML-Typen nicht um reguläre Ausdrücke handelt, sondern um reguläre Heckenausdrücke, die auf einer Heckengrammatik basieren, bedarf es einer Erweiterung von Antimirovs Algorithmus auf Heckengrammatiken.

Die Darstellung des Subtyp-Algorithmus beginnt mit den Definitionen von partiellen Ableitungen für reguläre Heckenausdrücke und regulären Ungleichungen. Anschließend erfolgt die Definition des Algorithmus selbst mit der Anwendung auf ein kleines Beispiel. Der Abschnitt endet mit Anmerkungen zur Komplexität.

Im Algorithmus werden reduzierte Ungleichungen berechnet, die aus reduzierten regulären Ausdrücken bestehen. Diese entstehen durch die sogenannte *partielle Ableitung regulärer Ausdrücke*. Eine partielle Ableitung reduziert einen regulären Ausdruck um das führende Terminalsymbol. Für die Definition der partiellen Ableitung ist eine erweiterte Konkatenation für reguläre Ausdrücke nötig, die Mengen von Tupeln als ersten Parameter zulässt. Sie ist wie folgt definiert.

Definition 4.30 (Erweiterte Konkatenation auf Mengen von Tupeln)
Die Erweiterung der *Konkatenation auf Mengen von Tupeln* $; : \mathcal{P}(\mathit{Reg} \times \mathit{Reg}) \times \mathit{Reg} \rightarrow \mathcal{P}(\mathit{Reg} \times$

Reg) sei wie folgt definiert:

$$\begin{aligned} R;\emptyset &= \{\} \\ \{\};t &= \{\} \\ \{(r,\emptyset)\};t &= \{\} \\ \{(\emptyset,s)\};t &= \{\} \\ \{(r,\epsilon)\};t &= \{(r,t)\} \\ \{(r,s)\};t &= \{(r,s;t)\} \\ (R \cup R');t &= (R;t) \cup (R';t) \end{aligned}$$

mit $R, R' \subseteq \mathit{Reg} \times \mathit{Reg}$, $r, t \in \mathit{Reg}\backslash\{\emptyset\}$ und $s \in \mathit{Reg}\backslash\{\emptyset, \epsilon\}$. □

Damit kann die partielle Ableitung mit folgender Definition formal spezifiziert werden.

Definition 4.31 (Partielle Ableitung eines regulären Ausdrucks)
Die *partielle Ableitung* hinsichtlich des Terminalsymbols $x \in T$ wird berechnet durch die Funktion $\mathit{der}_x : \mathit{Reg} \to \mathcal{P}(\mathit{Reg} \times \mathit{Reg})$. Sie ist rekursiv definiert durch:

$$\begin{aligned} \mathit{der}_x(\emptyset) &= \mathit{der}_x(\epsilon) = \{\} \\ \mathit{der}_x(n) &= \mathit{der}_x(r) \text{ mit } n \to r \in P \\ \mathit{der}_x(b) &= \begin{cases} \{(\epsilon,\epsilon)\} & \text{falls } b = x \text{ und } x \in B, \\ \{\} & \text{sonst} \end{cases} \\ \mathit{der}_x(e[r]) &= \begin{cases} \{(r,\epsilon)\} & \text{falls } e = x \text{ und } x \in E, \\ \{\} & \text{sonst} \end{cases} \\ \mathit{der}_x(r|s) &= \mathit{der}_x(r) \cup \mathit{der}_x(s) \\ \mathit{der}_x(r;s) &= \begin{cases} \mathit{der}_x(r);s & \text{falls } \neg \mathit{isNullable?}(r) \\ \mathit{der}_x(r);s \cup \mathit{der}_x(s) & \text{falls } \mathit{isNullable?}(r) \end{cases} \\ \mathit{der}_x(r*) &= \mathit{der}_x(r);r* \end{aligned}$$

mit $b \in B$, $n \in N$, $e \in E$ und $r, s \in \mathit{Reg}$. □

Um einen Eindruck zu bekommen, was die Operation *der* angewendet auf einen regulären Ausdruck leistet, wird nun ein kurzes Beispiel betrachtet.

Beispiel 4.8
Dieses Beispiel bezieht sich auf die Heckengrammatik aus Beispiel 4.3. Es wird der reguläre Ausdruck

$$r \equiv \texttt{account}[\texttt{integer}];\texttt{request}[\mathit{t\_request}]$$

betrachtet. Für diesen wird die partielle Ableitung hinsichtlich des Terminalsymbols **account** berechnet. Es ergibt sich die Ergebnismenge

$$\mathit{der}_{\texttt{account}}(r) = \{(\texttt{integer}, \texttt{request}[\mathit{t\_request}])\}.$$

Die Menge besteht in diesem Fall nur aus einem Tupel. Die erste Komponente des Tupels ist der reguläre Ausdruck, der den Inhalt des reduzierten Terminalsymbols beschreibt. In diesem Fall also den Inhalt des reduzierten **`account`**-Elements. Der reguläre Ausdruck der zweiten Komponente ist der um das Element **`account`** reduzierte Ausdruck r. □

Das Beispiel zeigt nicht, dass für einen regulären Ausdruck im Allgemeinen eine Menge regulärer Ausdrücke als partielle Ableitung entstehen kann. Dies ist bei Ausdrücken mit regulärer Vereinigung oder einer Konkatenation mit einem führenden Ausdruck der Fall, dessen Sprache die leere Hecke enthält. Jedes Element der partiellen Ableitung ist ein Paar, dessen Komponenten mit den zwei Dimensionen einer regulären Hecke korrespondieren. Die erste Komponente steht für die Vater-Kind-Dimension, während die zweite Komponente der Geschwister-Dimension entspricht.

Aufbauend auf der partiellen Ableitung für reguläre Ausdrücke wird von Antimirov die *partielle Ableitung für reguläre Ungleichungen* eingeführt. Für den Fall der Heckengrammatiken wird diese Definition schwierig, weil durch die partielle Ableitung der regulären Ausdrücke Paare entstehen. Ausgenutzt werden kann aber eine mengentheoretische Beobachtung von Hosoya, Vouillon und Pierce [HVP00], die hier als Satz formuliert wird. Ein Beweis findet sich in Anhang B.

Satz 4.2 (Teilmengenbeziehung des Karthesischen Produkts)
Gegeben seien die Mengen a, b, $c_1, \ldots, c_n$ und $d_1, \ldots, d_n$, dann gilt:

$$a \times b \subseteq (c_1 \times d_1) \cup \cdots \cup (c_n \times d_n)$$

$$\Leftrightarrow$$

$$(a \subseteq \cup_{i \in I_1} c_i \vee b \subseteq \cup_{i \in \overline{I}_1} d_i) \wedge \cdots \wedge (a \subseteq \cup_{i \in I_{2^n}} c_i \vee b \subseteq \cup_{i \in \overline{I}_{2^n}} d_i)$$

$$\text{mit } \mathcal{P}(\{1, \ldots, n\}) = \{I_1, \ldots, I_{2^n}\} \text{ und } \overline{I}_i = \{1, \ldots, n\} \backslash I_i \; .$$

Dabei bezeichnet $\mathcal{P}$ die Potenzmenge einer Menge. □

Der Satz ermöglicht die Umformung der Teilmengenbeziehung auf der linken Seite der Äquivalenz, die für Karthesische Produkte gilt, in Teilmengenbeziehungen auf der rechten Seite, die sich auf einfache Mengen beziehen. Das nachstehende Beispiel illustriert die mögliche Anwendung der Satzes.

Beispiel 4.9
Die gegebenen Teilmengenbeziehung

$$\mathcal{B} \times \mathcal{N}_0 \subseteq (\mathcal{B} \times \{0\}) \cup (\{\mathit{true}\} \times \mathcal{N}) \cup (\{\mathit{false}\} \times \mathcal{N})$$

wird mit Hilfe von Satz 4.2 zu

$$\begin{aligned}
&(\mathcal{B} \subseteq \mathcal{B} \vee \mathcal{N}_0 \subseteq \emptyset) \wedge (\mathcal{B} \subseteq \mathcal{B} \vee \mathcal{N}_0 \subseteq \mathcal{N}) \wedge \\
&(\mathcal{B} \subseteq \mathcal{B} \vee \mathcal{N}_0 \subseteq \mathcal{N}) \wedge (\mathcal{B} \subseteq \mathcal{B} \vee \mathcal{N}_0 \subseteq \{0\}) \wedge \\
&(\mathcal{B} \subseteq \mathcal{B} \vee \mathcal{N}_0 \subseteq \mathcal{N}) \wedge (\mathcal{B} \subseteq \{\mathit{true}\} \vee \mathcal{N}_0 \subseteq \mathcal{N}_0) \wedge \\
&(\mathcal{B} \subseteq \{\mathit{false}\} \vee \mathcal{N}_0 \subseteq \mathcal{N}_0) \wedge (\mathcal{B} \subseteq \emptyset \vee \mathcal{N}_0 \subseteq \mathcal{N}_0)
\end{aligned}$$

umgeformt. □

Da der Subtyp-Algorithmus, der im Folgenden definiert wird, nur mit Teilmengenbeziehungen auf einfacher Mengenebene nicht aber mit Teilmengenbeziehungen auf Karthesischen Produkten umgehen kann, ist diese Umformung essentiell.

Um Satz 4.2 bei der Definition der partiellen Ableitung regulärer Ungleichungen anzuwenden, ist zunächst daran zu erinnern, dass reguläre Ausdrücke Mengen von Hecken beschreiben. Erfolgt nun eine Reduktion von r, s einer regulären Ungleichung $r \leq s$ mittels der partiellen Ableitung für reguläre Ausdrücke hinsichtlich eines führenden Terminalsymbols x, so gilt weiterhin die Ungleichung

$$(c_r \times r_r) \subseteq (c_s^1 \times r_s^1) \cup \cdots \cup (c_s^n \times r_s^n) \text{ mit } \mathit{der}_x(s) = \{(c_s^1, r_s^1), \ldots, (c_s^n, r_s^n)\}$$

für alle $(c_r, r_r) \in \mathit{der}_x(r)$. Diese Teilmengenbeziehung ist nun nach Satz 4.2 äquivalent zu

$$(c_r \subseteq \cup_{i \in I_1} c_s^i \vee r_r \subseteq \cup_{i \in \overline{I}_1} r_s^i) \wedge \cdots \wedge (c_r \subseteq \cup_{i \in I_{2^n}} c_s^i \vee r_r \subseteq \cup_{i \in \overline{I}_{2^n}} r_s^i)$$
$$\text{mit } \mathcal{P}(\{1, \ldots, n\}) = \{I_1, \ldots, I_{2^n}\} \text{ und } \overline{I}_i = \{1, \ldots, n\} \backslash I_i \text{ und}$$
$$\mathit{der}_x(s) = \{(c_s^1, r_s^1), \ldots, (c_s^n, r_s^n)\}.$$

In der folgenden Definition der partiellen Ableitung regulärer Ungleichungen werden die entstandenen Disjunktionen von Ungleichungen der obigen Konjunktion in einer Menge zusammengefasst. Ebenfalls werden wieder die Operationen auf regulären Ausdrücken anstatt der Mengenoperatoren verwendet. Somit wird die Mengenvereinigung $\cup$ durch die reguläre Vereinigung $|$ und die Teilmengenrelation $\subseteq$ durch das Symbol der regulären Ungleichung $\leq$ ersetzt.

Definition 4.32 (Partielle Ableitung regulärer Ungleichungen)
Die *partielle Ableitung* einer regulären Ungleichung $r \leq s$ mit $r, s \in \mathit{Reg}$ hinsichtlich eines Terminalsymbols $x \in T$ ist definiert durch:

$$\mathit{part}_x(r \leq s) = \{(c_r \leq \Big| _{i \in I} c_s^i) \vee (r_r \leq \Big| _{i \in \overline{I}} r_s^i) |$$
$$(c_r, r_r) \in \mathit{der}_x(r) \wedge$$
$$\mathit{der}_x(s) = \{(c_s^1, r_s^1), \ldots, (c_s^n, r_s^n)\} \text{ mit}$$
$$I \in \mathcal{P}(\{1, \ldots, n\}) \text{ und } \overline{I} = \{1, \ldots, n\} \backslash I\}$$

mit $c_r, r_r, c_s^i, r_s^i \in \mathit{Reg}$. □

Die partielle Ableitung einer regulären Ungleichung ist damit eine Menge, deren Elemente aus zweistelligen Disjunktionen von regulären Ungleichungen bestehen. Das folgende Beispiel veranschaulicht diese Definition.

Beispiel 4.10
In diesem Beispiel gelte nach der Heckengrammatik aus Beispiel 4.3

$$r \equiv \mathit{description}; (\texttt{account}[\texttt{integer}]; \mathit{description})*$$
$$s \equiv (\texttt{description}[\texttt{string}]; \texttt{account}[\texttt{integer}])*; \mathit{description}$$

für die reguläre Ungleichung $r \leq s$, für die die partielle Ableitung hinsichtlich **`description`** berechnet werden soll. Die dafür notwendigen partiellen Ableitungen der linken und rechten Seiten sind folgende:

$$der_{\texttt{description}}(r) = \{(\texttt{string}, (\texttt{account}[\texttt{integer}]; \textit{description})*)\}$$
$$der_{\texttt{description}}(s) = \{(\texttt{string}, \texttt{account}[\texttt{integer}]; s), (\texttt{string}, \epsilon)\}$$

Dies ergibt für die partielle Ableitung der Ungleichung die Menge

$$\begin{aligned}
&part_{\texttt{description}}(r \leq s) = \\
&\quad \{(\texttt{string} \leq \texttt{string} | \texttt{string} \vee \\
&\qquad\qquad (\texttt{account}[\texttt{integer}]; \textit{description})* \leq \emptyset), \\
&\quad (\texttt{string} \leq \texttt{string} \vee \\
&\qquad\qquad (\texttt{account}[\texttt{integer}]; \textit{description})* \leq \epsilon), \\
&\quad (\texttt{string} \leq \texttt{string} \vee \\
&\qquad\qquad (\texttt{account}[\texttt{integer}]; \textit{description})* \leq \texttt{account}[\texttt{integer}]; s), \\
&\quad (\texttt{string} \leq \emptyset \vee \\
&\qquad\qquad (\texttt{account}[\texttt{integer}]; \textit{description})* \leq (\texttt{account}[\texttt{integer}]; s) | \epsilon)\}
\end{aligned}$$

mit vier Disjunktionen. □

Nach der Definition der partiellen Ableitung für reguläre Ungleichungen ist es nun möglich, die Regeln für den Subtyp-Algorithmus anzugeben.

Der Algorithmus testet, ob es sich bei der zu überprüfenden Ungleichung um eine trivial inkonsistente Ungleichung handelt. Ist dies nicht der Fall, kann nicht direkt entschieden werden, ob die Ungleichung korrekt ist. Stattdessen werden von der zu überprüfenden regulären Ungleichung sämtliche ableitbaren partiellen Ableitungen berechnet. Dafür werden zunächst die führenden Terminalsymbole der linken Seite der Ungleichung durch die Funktion *term* ermittelt. Um diese Terminalsymbole erfolgt die Verkürzung der regulären Ungleichung mit der Operation *part*. Auf die erzeugten Ungleichungen in den Disjunktionen der partiellen Ableitungen kann der Algorithmus rekursiv angewendet werden. Ist eine dieser Disjunktionen nicht korrekt, ist die zu überprüfende Ungleichung falsch. In allen anderen Fällen wird die gegebene reguläre Ungleichung als korrekt erkannt.

Der Subtyp-Algorithmus ist definiert durch zwei *Subtyp-Urteile* $\Gamma \vdash r \leq s \Rightarrow \Gamma'$ und $\Gamma \vdash^* r \leq s \Rightarrow \Gamma'$ mit der Menge Γ von regulären Ungleichungen der Form $t \leq u$. Beide Urteile werden interpretiert als: „Der Algorithmus überprüft $r \leq s$ und alle Ungleichungen $t \leq u$ in Γ sind nicht trivial inkonsistent. Als Resultat wird die Menge Γ' mit sämtlichen Ungleichungen der partiellen Ableitungen von $r \leq s$ zurückgeliefert."

Definition 4.33 (Subtyp-Urteile für reguläre Ungleichung)
Gegeben sei eine Menge regulärer Ungleichungen Γ, dann werden die folgenden beiden Subtyp-

Urteile unterschieden:

$$\Gamma \vdash r \leq s \Rightarrow \Gamma' \qquad r \leq s \text{ ist eine gültige Ungleichung in } \Gamma.$$
$$\Gamma \vdash^* r \leq s \Rightarrow \Gamma' \qquad r \leq s \text{ ist eine gültige Ungleichung in } \Gamma.$$

Dabei seien $r, s \in Reg$ und Γ' die resultierende Menge von regulären Ungleichungen. □

In jedem Schritt des Algorithmus entstehen neu partielle Ableitungen, deren Ungleichungen zur Menge Γ hinzugefügt werden. Es entsteht eine neue, erweiterte Menge Γ'.

Da die Nichtterminalsymbole in der Heckengrammatik rekursiv definiert sein können, kann es vorkommen, dass durch die rekursive Berechnung sämtlicher partieller Ableitungen bereits berechnete Ungleichungen zu einem späteren Zeitpunkt erneut auftreten. Damit für diese Fälle die Terminierung des Algorithmus sichergestellt ist, werden sämtliche bereits berechneten Ungleichungen in der Menge Γ gespeichert. Andererseits darf aber nicht sofort nach dem Hinzufügen einer neuen Ungleichung zu Γ dieser rekursive Zweig erfolgreich abgebrochen werden, weil dies dazu führen würde, dass nicht sämtliche ableitbaren partiellen Ableitungen berechnet werden. Aus diesem Grund wird nach dem Hinzufügen einer Ungleichung zu Γ vom ersten Subtyp-Urteil $\vdash$ in das zweite Urteil $\vdash^*$ umgeschaltet. Ein Zurückschalten erfolgt erst nach der Berechnung von neuen regulären Ungleichungen. Auf diesem Weg ist die Erzeugung sämtlicher partieller Ableitungen sichergestellt. Die Menge Γ ist zu Beginn des Subtyp-Algorithmus leer.

Definition 4.34 (Subtyp-Algorithmus)
Der Subtyp-Algorithmus ist durch folgende Regeln definiert:

$$\frac{r \leq s \in \Gamma}{\Gamma \vdash r \leq s \Rightarrow \Gamma} \qquad \text{(HYP)}$$

$$\frac{\begin{array}{c} r \leq s \notin \Gamma, \\ \Gamma \cup \{r \leq s\} \vdash^* r \leq s \Rightarrow \Gamma' \end{array}}{\Gamma \vdash r \leq s \Rightarrow \Gamma'} \qquad \text{(ASSUM)}$$

$$\frac{\begin{array}{c} \neg \mathit{inc}(r \leq s), \\ \text{für alle } i \in \{1, \ldots, k\} \text{ mit } k = |\mathit{part}_x(r \leq s)| \text{ und } x \in \mathit{term}(r) \text{ gilt} \\ \Gamma_{i-1} \vdash c_r \leq c_s \Rightarrow \Gamma_i \vee \Gamma_{i-1} \vdash r_r \leq r_s \Rightarrow \Gamma_i \\ \text{mit } (c_r \leq c_s \vee r_r \leq r_s) \in \mathit{part}_x(r \leq s) \end{array}}{\Gamma_0 \vdash^* r \leq s \Rightarrow \Gamma_k} \qquad \text{(REC)}$$

□

In der Regel HYP wird getestet, ob die zu überprüfende Ungleichung bereits in der Menge der bereits berechneten Ungleichungen enthalten ist. Dies bedeutet, dass in diesem Rekursionszweig keine neuen Ungleichungen erzeugt werden können. Ist dies der Fall beendet der Algorithmus an dieser Stelle die Berechnung erfolgreich. Die Regel ASSUM dient zum Umschalten zwischen

den beiden Subtyp-Urteilen $\vdash$ und $\vdash^*$. Ist die zu überprüfende Ungleichung noch nicht in der Menge Γ wird sie hinzugefügt und zum Urteil $\vdash^*$ übergegangen.

Regel REC ist nur anwendbar, falls es sich nicht um eine trivial inkonsistente Ungleichung handelt, was mit dem Prädikat *inc* überprüft wird. Es wird die partielle Ableitung der Ungleichung mit der Operation *part* berechnet, deren Ungleichungen rekursiv zu überprüfen sind. Ist eine Anwendung der Regel nicht möglich, bricht der Algorithmus für diese inkorrekte Ungleichung ab. Listing 4.1 zeigt mit der Methode `prove` zum besseren Verständnis den Algorithmus nochmals

```
1  boolean prove_Γ(r ≤ s) {
2     if (inc(r ≤ s))
3        return false;
4     elsif ((r ≤ s) ∈ Γ)                          // HYP
5        return true;
6     else {
7        boolean ok := true;
8        Set pd := ∅;
9        Set ns := term(r);
10       foreach (x ∈ ns)
11          pd := pd ∪ part_x(r ≤ s);
12       Γ := Γ ∪ {r ≤ s};                         // ASSUM
13       foreach ((c_r ≤ c_s) ∨ (r_r ≤ r_s) ∈ pd)   // REC
14         ok := ok && (prove_Γ(c_r ≤ c_s) || prove_Γ(r_r ≤ r_s));
15       return ok;
16    } // else
17 } // prove
```

Listing 4.1: Subtyp-Algorithmus in Pseudocode

in Pseudocode.

Das folgende Beispiel zeigt die Anwendung des Algorithmus.

Beispiel 4.11
Betrachtet werden in diesem Beispiel erneut die beiden regulären Ausdrücke

$$r \equiv \textit{description}; (\texttt{account}[\texttt{integer}]; \textit{description}) * \text{ und}$$
$$s \equiv (\texttt{description}[\texttt{string}]; \texttt{account}[\texttt{integer}])*; \textit{description}$$

aus Beispiel 4.10. Wieder wird die reguläre Ungleichung $r \leq s$ betrachtet, die mit dem Subtyp-Algorithmus auf ihre Richtigkeit hin überprüft werden soll. Zu Beginn der Berechnung ist $\Gamma = \{\}$ und der Algorithmus ermittelt die führenden Terminalsymbole der rechten Seite mit dem Ergebnis $\textit{term}(r) = \{\texttt{description}\}$. Für das Element dieser Menge wurde bereits in

Beispiel 4.10 die partielle Ableitung gebildet. Seien im Weiteren noch

$$r^{'} \equiv (\texttt{account}[\texttt{integer}]; \textit{description}) * \text{ und}$$
$$s^{'} \equiv \texttt{account}[\texttt{integer}]; s$$

so ergeben sich für die weitere Berechnung des Algorithmus die Teilergebnisse, die Abbildung 4.6 zeigt. Ersichtlich ist, dass einige Ungleichungen, die während der Berechnung auftreten, scheitern. Andere werden als korrekt erkannt und in die Menge Γ aufgenommen.

Offen bleibt in der Abbildung noch die Überprüfung der Ungleichung $r^{'} \leq s^{'}|\epsilon$. Da die Menge der führenden Terminalsymbole von $r^{'}$ nur aus **account** besteht, ergibt sich mit der partiellen Ableitung

$$\begin{aligned} \textit{part}_{\texttt{account}}(r^{'} \leq s^{'}|\epsilon) = \{&(\texttt{integer} \leq \texttt{integer} \vee r \leq \emptyset), \\ &(\texttt{integer} \leq \emptyset \vee r \leq s)\} \end{aligned}$$

der weitere Verlauf des Algorithmus, wie er in Abbildung 4.7 dargestellt ist. Der Algorithmus akzeptiert in diesem Zweig die Ungleichung $r^{'} \leq s^{'}$ mit der Menge aller berechneten Ungleichungen Γ_6. In Abbildung 4.5 ist dargestellt, wie sich die Menge Γ während der Ausführung verändert. Die zu Beginn leere Menge füllt sich allmählich mit neuen regulären Ungleichungen,

$$\begin{aligned} \Gamma_1 &= \{r \leq s\} \\ \Gamma_2 &= \Gamma_1 \cup \{\texttt{string} \leq \texttt{string}|\texttt{string}\} \\ \Gamma_3 &= \Gamma_2 \cup \{\epsilon \leq \epsilon\} \\ \Gamma_4 &= \Gamma_3 \cup \{\texttt{string} \leq \texttt{string}\} \\ \Gamma_5 &= \Gamma_4 \cup \{r^{'} \leq s^{'}|\epsilon\} \\ \Gamma_6 &= \Gamma_5 \cup \{\texttt{integer} \leq \texttt{integer}\} \end{aligned}$$

Abbildung 4.5: Entwicklung der Menge Γ

bis der Algorithmus keine neuen Ungleichungen mehr erzeugt.

Schließlich terminiert der Algorithmus mit dem Resultat, dass es sich bei $r \leq s$ um eine gültige Ungleichung handelt. Denn der Algorithmus konnte für keine Disjunktion beide Ungleichungen als inkonsistent ermitteln. □

Komplexität

Die Komplexität der Subtyp-Überprüfung ist, wie in [Sei90] gezeigt, EXPTIME-vollständig und ist damit im schlimmsten Fall exponentiell. Trotzdem ist der Algorithmus im Vergleich zum klassischen Verfahren auf der Basis von Baumautomaten (siehe Abschnitt 4.1) eine Verbesserung.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\textsc{Rec}}{\Gamma_3 \vdash^* \epsilon \le \epsilon \Rightarrow \Gamma_3}}{\Gamma_2 \vdash \epsilon \le \epsilon \Rightarrow \Gamma_3} \vee \dfrac{\dfrac{\mathit{Fehler}}{\Gamma_2 \cup \{\epsilon \le \emptyset\} \vdash^* \epsilon \le \emptyset}}{\Gamma_2 \vdash \epsilon \le \emptyset}}{\Gamma_2 \vdash^* \mathtt{string} \le \mathtt{string}|\mathtt{string} \Rightarrow \Gamma_3}}{\Gamma_1 \vdash \mathtt{string} \le \mathtt{string}|\mathtt{string} \Rightarrow \Gamma_3} \vee \dfrac{\dfrac{\mathit{Fehler}}{\Gamma_1 \cup \{r' \le \emptyset\} \vdash^* r' \le \emptyset}}{\Gamma_1 \vdash r' \le \emptyset}, \quad \dfrac{\dfrac{\dfrac{\textsc{Hyp}}{\Gamma_4 \vdash \epsilon \le \epsilon \Rightarrow \Gamma_4} \vee \dfrac{\dfrac{\mathit{Fehler}}{\Gamma_4 \cup \{\epsilon \le \emptyset\} \vdash^* \epsilon \le \emptyset}}{\Gamma_4 \vdash \epsilon \le \emptyset}}{\Gamma_4 \vdash^* \mathtt{string} \le \mathtt{string} \Rightarrow \Gamma_4}}{\Gamma_3 \vdash \mathtt{string} \le \mathtt{string} \Rightarrow \Gamma_4} \vee \dfrac{\dfrac{\begin{matrix}\mathit{Fehler}\\ \vdots\end{matrix}}{\Gamma_3 \cup \{r' \le \epsilon\} \vdash^* r' \le \epsilon}}{\Gamma_3 \vdash r' \le \epsilon}, \\ \dfrac{\textsc{Hyp}}{\Gamma_4 \vdash \mathtt{string} \le \mathtt{string} \Rightarrow \Gamma_4} \vee \dfrac{\dfrac{\mathit{Fehler}}{\Gamma_4 \cup \{r' \le s'\} \vdash^* r' \le s'}}{\Gamma_4 \vdash r' \le s'}, \quad \dfrac{\dfrac{\begin{matrix}\mathit{Fehler}\\ \vdots\end{matrix}}{\Gamma_4 \cup \{\mathtt{string} \le \emptyset\} \vdash^* \mathtt{string} \le \emptyset}}{\Gamma_4 \vdash \mathtt{string} \le \emptyset} \vee \dfrac{\text{(siehe Abbildung 4.7)}}{\Gamma_4 \vdash r' \le s'|\epsilon \Rightarrow \Gamma_6}}{\Gamma_1 \vdash^* r \le s \Rightarrow \Gamma_6}}{\emptyset \vdash r \le s \Rightarrow \Gamma_6}$$

Abbildung 4.6: Berechnung für Ungleichung $r \le s$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\textsc{Hyp}}{\Gamma_6 \vdash \epsilon \le \epsilon \Rightarrow \Gamma_6} \vee \dfrac{\dfrac{\mathit{Fehler}}{\Gamma_6 \cup \{\epsilon \le \emptyset\} \vdash^* \epsilon \le \emptyset}}{\Gamma_6 \vdash \epsilon \le \emptyset}}{\Gamma_6 \vdash^* \mathtt{integer} \le \mathtt{integer} \Rightarrow \Gamma_6}}{\Gamma_5 \vdash \mathtt{integer} \le \mathtt{integer} \Rightarrow \Gamma_6} \vee \dfrac{\dfrac{\begin{matrix}\mathit{Fehler}\\ \vdots\end{matrix}}{\Gamma_5 \cup \{r \le \emptyset\} \vdash^* r \le \emptyset}}{\Gamma_5 \vdash r \le \emptyset}, \dfrac{\dfrac{\begin{matrix}\mathit{Fehler}\\ \vdots\end{matrix}}{\Gamma_6 \cup \{\mathtt{integer} \le \emptyset\} \vdash^* \mathtt{integer} \le \emptyset}}{\Gamma_6 \vdash \mathtt{integer} \le \emptyset} \vee \dfrac{\textsc{Hyp}}{\Gamma_6 \vdash r \le s \Rightarrow \Gamma_6}}{\Gamma_5 \vdash^* r' \le s'|\epsilon \Rightarrow \Gamma_6}}{\Gamma_4 \vdash r' \le s'|\epsilon \Rightarrow \Gamma_6}$$

Abbildung 4.7: Berechnung für Ungleichung $r' \le s'|\epsilon$

Der klassische Algorithmus, der nach dem Komplement eines Automaten die Vereinigung berechnet, erzeugt immer einen minimalen deterministischen Automaten mit allen Zuständen, was exponentiell lange dauern kann. Diese Determinisierung ist aber nicht immer notwendig, wie Antimirov in [Ant94] zeigt. Es gibt Ungleichungen wie $r \leq s$ mit $r = (\mathit{book}*; \mathit{record})*; \mathit{book}^n; \mathit{book}*$ und $s = (\mathit{book}|\mathit{record})*; \mathit{book}; (\mathit{book}|\mathit{record})^{n-1}$ in denen das Verfahren dieser Arbeit polynomiell arbeitet, weil nur $3n - 3$ Ungleichungen erzeugt werden, während der klassische Algorithmus mit 2^n Zuständen exponentiellen Aufwand [Per90] benötigt. Dies liegt daran, dass der Subtyp-Algorithmus dieser Arbeit die rechte Seite einer Ungleichung nur so weit determinisiert, wie es für die Überprüfung der Ungleichung notwendig ist. Auf eine vollständige Determinisierung kann dadurch in manchen Fällen verzichtet werden.

4.7 Korrektheit des Algorithmus

Das im letzten Abschnitt dargestellte Beispiel macht plausibel, dass der Algorithmus die Subtyp-Beziehung zweier regulärer Heckenausdrücke überprüft; damit stellt sich die Frage, ob der Algorithmus korrekt und vollständig arbeitet sowie in allen Fällen terminiert. In diesem Abschnitt wird deshalb diese Fragestellung näher untersucht und abschließend positiv beantwortet.

4.7.1 Korrektheit

Um die Korrektheit des Algorithmus in diesem Abschnitt zu zeigen, werden zunächst einige Definitionen eingeführt, die im Beweis Verwendung finden. Zusätzlich wird zwischen Basisdatentypen und Elementnamen nicht mehr unterschieden. Stattdessen werden Basisdatentypen nun wie Elementtypen mit leerem Inhalt behandelt. Es gilt also für alle $b \in B$: $b \Leftrightarrow b[\epsilon]$.

Zunächst wird für jede Hecke eine Menge von Heckenpräfixen definiert.

Definition 4.35 (Menge aller Heckenpräfixe)
Die *Menge aller Heckenpräfixe Prefix* einer Heckensprache L sei definiert durch:

$$\mathit{Prefix}(L) = \{\mathit{prefix}(v) | v \in L\}$$

Für die Menge der *Heckenpräfixe einer Hecke prefix* : $T^* \to \mathcal{P}(T^*)$ gilt:

$$\begin{aligned} \mathit{prefix}(\epsilon) &= \{\epsilon\} \\ \mathit{prefix}(a[v]) &= \{a[u] | u \in \mathit{prefix}(v)\} \\ \mathit{prefix}(vw) &= \mathit{prefix}(v) \cup \{vw\} \end{aligned}$$

Die *Größe eines Heckenpräfixes* sei rekursiv definiert durch:

$$\begin{aligned} |\epsilon| &= 0 \\ |a[v]| &= |v| + 1 \\ |vw| &= |v| + |w| \end{aligned}$$

mit $a \in T$ und $v, w \in T^*$. □

Es folgt eine Erweiterung der Heckensprache auf Tupel von regulären Ausdrücken.

Definition 4.36 (Erweiterte Heckensprache)
Die *Erweiterung der Heckensprache* auf Tupel von regulären Ausdrücken sei definiert durch:

$$L((r, s)) = L(r) \times L(s)$$

mit $r, s \in$ *Reg*. □

Desweiteren wird das Leere-Hecken-Prädikat auf Tupel von regulären Ausdrücken erweitert.

Definition 4.37 (Erweitertes Leere-Hecke-Prädikat)
Die *Erweiterung des Leere-Hecke-Prädikats isNullable?* auf Tupel ist rekursiv definiert durch:

$$\textit{isNullable?}((r, s)) = \textit{isNullable?}(r) \wedge \textit{isNullable?}(s)$$

mit $r, s \in$ *Reg*. □

Die triviale Inkonsistenz, die in Definition 4.10 für reguläre Ungleichungen definiert ist, wird nun auf Konjunktionen und Disjunktionen von regulären Ungleichungen erweitert.

Definition 4.38 (Erweiterte Inkonsistenz)
Die *Erweiterung der Inkonsistenz inc* auf Konjunktionen von Disjunktionen sei rekursiv definiert durch:

$$\begin{aligned} \textit{inc}(d_1 \wedge \cdots \wedge d_n) &= \textit{inc}(d_1) \vee \cdots \vee \textit{inc}(d_n) \\ \textit{inc}(c_1 \vee c_2) &= \textit{inc}(c_1) \wedge \textit{inc}(c_2) \end{aligned}$$

Dabei seien $d_1, \ldots, d_n$ Disjunktionen und c_1 und c_2 entweder Konjunktionen oder reguläre Ungleichungen. □

Weiterhin wird die partielle Ableitung eines regulären Ausdrucks von Terminalsymbolen auf ganze Heckenpräfixe erweitert.

Definition 4.39 (Erweiterte partielle Ableitung eines regulären Ausdrucks)
Die *Erweiterung der partiellen Ableitung* eines regulären Ausdrucks hinsichtlich eines Heckenpräfixes ist rekursiv definiert durch:

$$\textit{der}_{a[v]w}(r) = \textit{der}_v(x) \times \textit{der}_w(y) \text{ mit } (x, y) \in \textit{der}_a(r)$$

mit $a \in T$, $v, w \in T^*$ und $x, y, r \in$ *Reg*. □

Eine analoge Erweiterung wird für die partielle Ableitung regulärer Ungleichungen vorgenommen.

Definition 4.40 (Erweiterte partielle Ableitung einer regulären Ungleichung)
Die *Erweiterung der partiellen Ableitung* einer regulären Ungleichung hinsichtlich eines Heckenpräfixes ist rekursiv definiert durch:

$$\mathit{part}_{a[v]w}(r \leq s) = \{\bigwedge \mathit{part}_v(x) \vee \bigwedge \mathit{part}_w(y) | x \vee y \in \mathit{part}_a(r \leq s)\}$$

mit $a \in T$, $v, w \in T^*$ und $x, y, r, s \in \mathit{Reg}$. □

Damit besteht die Möglichkeit, eine Menge aller partiellen Ableitungen regulärer Ungleichungen zu definieren.

Definition 4.41 (Menge aller partiellen Ableitungen)
Die *Menge aller partiellen Ableitungen* einer regulären Ungleichung ist dann definiert durch:

$$\mathit{PAR}(r \leq s) = \bigcup_{v \in T^+} \mathit{part}_v(r \leq s)$$

mit $r, s \in \mathit{Reg}$. □

Die Menge aller partiellen Ableitungen steht in einem engen Zusammenhang zum Subtyp-Algorithmus dieser Arbeit (Definition 4.34), denn dieses Verfahren berechnet genau diese Menge. Es beruht auf der Beobachtung, dass, falls $r \leq s$ eine gültige Ungleichung ist, alle partiellen Ableitungen $\mathit{PAR}(r \leq s)$ nicht trivial inkonsistent, also gültig, sind. Ist dagegen $r \leq s$ nicht gültig, so enthält die Menge $\mathit{PAR}(r \leq s)$ mindestens eine Konjunktion, die trivial inkonsistent ist.

Schließlich wird die Konkatenation von Hecken noch erweitert auf die Konkatenation eines Heckenpräfixes mit einem Tupel von Hecken.

Definition 4.42 (Erweiterte Konkatenation)
Die *Erweiterung der Konkatenation* von Hecken auf die Konkatenation eines Heckenpräfixes mit einem Tupel von Hecken sei definiert durch:

$$\begin{aligned} a[v]w \cdot (x, y) &= a[v \cdot x]w \cdot y \\ w \cdot x &= wx \\ \epsilon \cdot x &= x \end{aligned}$$

mit $a \in T$ und $v, w, x, y \in T^*$. □

Nach der Angabe der für die Korrektheitsbeweise nötigen Definitionen, kann ein erster Hilfssatz formuliert werden, der ebenfalls notwendig wird. Dieser besagt, dass die Hecken der Sprache eines durch partielle Ableitung hinsichtlich eines Terminalsymbols reduzierten regulären Heckenausdrucks identisch sind mit den Hecken der Sprache des ursprünglichen regulären Ausdrucks, die um das führende Terminalsymbol verkürzt wurden.

Hilfssatz 4.1
Sei $L(r)$ eine reguläre Sprache mit $r \in \mathit{Reg}$ dann gilt:

$$\{a[v]w | a[v]w \in L(r) \text{ und } a \in \mathit{term}(r)\} = \bigcup_{s_i \in \mathit{der}_a(r)} \{a[v]w | (v, w) \in L(s_i)\}$$

Beweis:
Vollständige Induktion über Länge des regulären Ausdrucks r:

- Induktionsanfang:
  - Angenommen es gilt $r = \emptyset$, dann folgt:

$$\{a[v]w | a[v]w \in L(\emptyset) \wedge a \in \mathit{term}(\emptyset)\} = \emptyset = \bigcup_{s_i \in \mathit{der}_a(\emptyset)} \{a[v]w | (v,w) \in L(s_i)\}$$

  - Angenommen es gilt $r = \epsilon$, dann folgt:

$$\{a[v]w | a[v]w \in L(\epsilon) \wedge a \in \mathit{term}(\epsilon)\} = \emptyset = \bigcup_{s_i \in \mathit{der}_a(\epsilon)} \{a[v]w | (v,w) \in L(s_i)\}$$

  - Angenommen es gilt $r = a$, dann folgt:

$$\begin{aligned} \{a[v]w | a[v]w \in L(a) \wedge a \in \mathit{term}(a)\} &= \{a\} \\ &= \{a[v]w | (v,w) \in \{(\epsilon,\epsilon)\}\} \\ &= \bigcup_{s_i \in \mathit{der}_a(a)} \{a[v]w | (v,w) \in L(s_i)\} \end{aligned}$$

- Induktionsschluss:
  - Angenommen es gilt $r = t*$, dann folgt:

$$\begin{aligned} & \{a[v]w \mid a[v]w \in L(t*) \wedge a \in \mathit{term}(t*)\} \\ = \; & \{a[v]w \mid a[v]w \in L(t;t*) \wedge a \in \mathit{term}(t)\} \\ = \; & \{a[v]xy \mid a[v]x \in L(t) \wedge y \in L(t*) \wedge a \in \mathit{term}(t)\} \\ \overset{\text{I. V.}}{=} \; & \bigcup_{s_i \in \mathit{der}_a(t)} \{a[v]xy \mid x \in L(s_i) \wedge y \in L(t*)\} \\ = \; & \bigcup_{s_i \in \mathit{der}_a(t);t*} \{a[v]w \mid (v,w) \in L(s_i)\} \\ = \; & \bigcup_{s_i \in \mathit{der}_a(t*)} \{a[v]w \mid (v,w) \in L(s_i)\} \end{aligned}$$

– Angenommen es gilt $r = t_1|t_2$, dann folgt:

$$\begin{aligned}
& \{a[v]w|a[v]w \in L(t_1|t_2) \wedge a \in \mathit{term}(t_1|t_2)\} \\
= \; & \{a[v]w|a[v]w \in L(t_1) \wedge a \in \mathit{term}(t_1) \vee a[v]w \in L(t_2) \wedge a \in \mathit{term}(t_2)\} \\
= \; & \{a[v]w|a[v]w \in L(t_1) \wedge a \in \mathit{term}(t_1)\} \cup \\
& \qquad \{a[v]w|a[v]w \in L(t_2) \wedge a \in \mathit{term}(t_2)\} \\
\overset{\text{I. V.}}{=} \; & \bigcup_{s_i \in \mathit{der}_a(t_1)} \{a[v]w|(v,w) \in L(s_i)\} \cup \bigcup_{s_i \in \mathit{der}_a(t_2)} \{a[v]|(v,w) \in L(s_i)\} \\
= \; & \bigcup_{s_i \in \mathit{der}_a(t_1) \cup \mathit{der}_a(t_2)} \{a[v]w|(v,w) \in L(s_i)\} \\
= \; & \bigcup_{s_i \in \mathit{der}_a(t_1|t_2)} \{a[v]w|(v,w) \in L(s_i)\}
\end{aligned}$$

– Angenommen es gilt $r = t_1; t_2$ und $\neg \mathit{isNullable?}(t_1)$, dann folgt:

$$\begin{aligned}
& \{a[v]w|a[v]w \in L(t_1;t_2) \wedge a \in \mathit{term}(t_1;t_2)\} \\
= \; & \{a[v]xy|a[v]x \in L(t_1) \wedge y \in L(t_2) \wedge a \in \mathit{term}(t_1)\} \\
\overset{\text{I. V.}}{=} \; & \bigcup_{s_i \in \mathit{der}_a(t_1)} \{a[v]xy|(v,x) \in L(s_i) \wedge y \in L(t_2)\} \\
= \; & \bigcup_{s_i \in \mathit{der}_a(t_1);t_2} \{a[v]xy|(v,xy) \in L(s_i)\} \\
= \; & \bigcup_{s_i \in \mathit{der}_a(t_1;t_2)} \{a[v]w|(v,w) \in L(s_i)\}
\end{aligned}$$

– Angenommen es gilt $r = t_1; t_2$ und $\mathit{isNullable?}(t_1)$, dann folgt:

$$\begin{aligned}
& \{a[v]w|a[v]w \in L(t_1;t_2) \wedge a \in \mathit{term}(t_1;t_2)\} \\
= \; & \{a[v]xy|a[v]x \in L(t_1) \wedge y \in L(t_2) \wedge a \in \mathit{term}(t_1) \vee \\
& \qquad a[v]xy \in L(t_2) \wedge a \in \mathit{term}(t_2)\} \\
= \; & \{a[v]xy|a[v]x \in L(t_1) \wedge y \in L(t_2) \wedge a \in \mathit{term}(t_1)\} \cup \\
& \qquad \{a[v]w|a[v]w \in L(t_2) \wedge a \in \mathit{term}(t_2)\} \\
\overset{\text{I. V.}}{=} \; & \bigcup_{s_i \in \mathit{der}_a(t_1)} \{a[v]xy|(v,x) \in L(s_i) \wedge y \in L(t_2)\} \cup \\
& \qquad \bigcup_{s_i \in \mathit{der}_a(t_2)} \{a[v]w|(v,w) \in L(s_i)\} \\
= \; & \bigcup_{s_i \in \mathit{der}_a(t_1);t_2 \cup \mathit{der}_a(t_2)} \{a[v]w|(v,w) \in L(s_i)\} \\
= \; & \bigcup_{s_i \in \mathit{der}_a(t_1;t_2)} \{a[v]w|(v,w) \in L(s_i)\}
\end{aligned}$$

□

Eine Erweiterung des letzten Hilfssatzes auf ganze Heckenpräfixe, lässt sich ebenfalls angeben.

Hilfssatz 4.2
Sei $L(r)$ eine reguläre Sprache mit $r \in \mathit{Reg}$ dann gilt:

$$\{v \cdot w | v \cdot w \in L(r) \text{ und } v \in \mathit{Prefix}(L(r))\} = \bigcup_{s_i \in \mathit{der}_v(r)} \{v \cdot w | w \in L(s_i)\}$$

Beweis:
Vollständige Induktion über die Größe des Präfixes v:

- Induktionsanfang: Angenommen es gilt $|v| = 1$, dann folgt:

$$\begin{aligned}
& \{v \cdot w | v \cdot w \in L(r) \wedge v \in \mathit{Prefix}(L(r))\} \\
= \quad & \{a[\epsilon]w | a[\epsilon]w \in L(r) \text{ und } a \in \mathit{term}(r)\} \\
\overset{\text{Hilfss. 4.1}}{=} \quad & \bigcup_{s_i \in \mathit{der}_a(r)} \{a[\epsilon]w | (\epsilon, w) \in L(s_i)\} \\
= \quad & \bigcup_{s_i \in \mathit{der}_v(r)} \{v \cdot w | w \in L(s_i)\}
\end{aligned}$$

- Induktionsschluss: Angenommen es gilt $|v| > 1$, dann folgt:

$$\begin{aligned}
& \{v \cdot w | v \cdot w \in L(r) \wedge v \in \mathit{Prefix}(L(r))\} \\
= \quad & \{a[x]y \cdot w | a[x]y \cdot w \in L(r) \text{ und } a \in \mathit{term}(r)\} \\
\overset{\text{Hilfss. 4.1}}{=} \quad & \bigcup_{s_i \in \mathit{der}_a(r)} \{a[x]y \cdot w | (x, y \cdot w) \in L(s_i)\} \\
\overset{\text{I. V. u. I. V. u. Def. } \mathit{der}}{=} \quad & \bigcup_{s_i \in \mathit{der}_{a[x]y}(r)} \{a[x]y \cdot w | w \in L(s_i)\} \\
= \quad & \bigcup_{s_i \in \mathit{der}_v(r)} \{v \cdot w | w \in L(s_i)\}
\end{aligned}$$

□

Um den Hilfssatz zu beweisen, wird eine vollständige Induktion über die Größe der Heckenpräfixe angewendet und auf Hilfssatz 4.1 zurückgegriffen.

Nach diesem Hilfssatz folgt ein weiterer Hilfssatz, der auf dem vorherigen basiert. Dieser besagt, dass es zu jeder Hecke einer Sprache, die durch einen regulären Ausdruck beschrieben wird, mindestens eine partielle Ableitung des regulären Ausdrucks hinsichtlich dieser Hecke gibt, die die leere Hecke repräsentiert.

Hilfssatz 4.3
Sei $L(r)$ eine reguläre Sprache mit $r \in \mathit{Reg}$, dann gilt:

$$v \in L(r) \Leftrightarrow \exists s \in \mathit{der}_v(r) \text{ mit } \mathit{isNullable?}(s)$$

Indirekter Beweis:
Angenommen es gilt $\nexists s \in \mathit{der}_v(r)$ mit $\mathit{isNullable?}(s)$, dann folgt:

$$\begin{aligned} &\overset{\text{Def. } L}{\Leftrightarrow} && v \notin \bigcup_{s_i \in \mathit{der}_v(r)} \{v \cdot w | w \in L(s_i)\} \\ &\overset{\text{Hilfss. 4.2}}{\Leftrightarrow} && v \notin \{v \cdot w | v \cdot w \in L(r) \text{ und } v \in \mathit{Prefix}(L(r))\} \end{aligned}$$

□

Der Beweis wird indirekt geführt und wendet Hilfssatz 4.2 an.

Die *Korrektheit* des Subtyp-Algorithmus liegt dann vor, falls für jedes positive Resultat des Algorithmus die Eingabe eine gültige Ungleichung ist. Mit Satz 4.3 lässt sich diese Bedingung formulieren.

Satz 4.3 (Korrektheit)
Liefert der Subtyp-Algorithmus $\Gamma \vdash r \leq s$, dann ist $r \leq s$ eine gültige Ungleichung, also:

$$\nexists f \in \mathit{PAR}(r \leq s) \text{ mit } \mathit{inc}(f) \Rightarrow r \leq s$$

Indirekter Beweis:
Angenommen es gilt $r \not\leq s$, dann folgt:

$$\begin{aligned} &\overset{\text{Def. } \leq}{\Rightarrow} && \exists v \in L(r) \text{ mit } v \notin L(s) \\ &\overset{\text{Hilfss. 4.3}}{\Rightarrow} && \exists x \in \mathit{der}_v(r) \text{ mit } \mathit{isNullable?}(x) \text{ und } \nexists y \in \mathit{der}_v(s) \text{ mit } \mathit{isNullable?}(y) \\ &\overset{\text{Def. } \mathit{der}}{\Rightarrow} && \exists f \in \mathit{PAR}(r \leq s) \text{ mit } \mathit{inc}(f) \end{aligned}$$

□

Die Korrektheit des Subtyp-Algorithmus wird indirekt bewiesen, wobei zur Unterstützung Hilfssatz 4.3 herangezogen wird.

4.7.2 Vollständigkeit

Damit der Subtyp-Algorithmus als *vollständig* gilt, muss dieser für jede reguläre Ungleichung ein positives Ergebnis ermitteln. Formal wird diese Eigenschaft durch Satz 4.4 beschrieben.

Satz 4.4 (Vollständigkeit)
Gilt die Ungleichung $r \leq s$, dann liefert der Subtyp-Algorithmus $\Gamma \vdash r \leq s$, also:

$$r \leq s \Rightarrow \nexists f \in \textit{PAR}(r \leq s) \text{ mit } \textit{inc}(f)$$

Indirekter Beweis:
Angenommen es gilt $\exists f \in \textit{PAR}(r \leq s)$ mit $\textit{inc}(f)$, dann gilt:

$$\begin{aligned}
&\overset{\text{Def. } \textit{PAR}}{\Rightarrow} && \exists v \in \textit{Prefix}(L(r)) \text{ mit } f \in \textit{part}_v(r \leq s) \text{ mit } \textit{inc}(f) \\
&\overset{\text{Def. } \textit{der} \text{ und Hilfss. 4.3}}{\Rightarrow} && v \in L(r) \wedge v \notin L(s) \\
&\overset{\text{Def. } \leq}{\Rightarrow} && r \not\leq s
\end{aligned}$$

□

Der Beweis der Vollständigkeit des Subtyp-Algorithmus wird erneut indirekt geführt und beruht ebenfalls im Kern auf Hilfssatz 4.3.

4.7.3 Terminierung

Die Terminierung des Subtyp-Algorithmus wird in diesem Abschnitt untersucht. Dafür wird die Nerode-Kongruenz $\sim$ herangezogen, da für reguläre Baumsprachen nur endlich viele Klassen existieren [RS97].

Definition 4.43 (Nerode-Kongruenz)
Zwei Heckenpräfixe $v, w \in \textit{Prefix}(L(r))$ sind kongruent, falls gilt:

$$v \sim w \quad \Leftrightarrow \quad \forall x \in \bigcup_{s_i \in \textit{der}_v(r)} L(s_i) \text{ gilt } v \cdot x \in L(r) \text{ und } w \cdot x \in L(r)$$

□

Zwei Heckenpräfixe v und w sind genau dann kongruent, falls sowohl die Konkatenation von v mit einer Fortsetzung x als auch die Konkatenation von w mit x in der betrachteten Sprache enthalten sind.

Damit der Subtyp-Algorithmus terminiert, muss sichergestellt sein, dass es endlich viele partielle Ableitungen für eine reguläre Ungleichung gibt. Da die partielle Ableitung einer regulären Ungleichung mittels der partiellen Ableitungen der regulären Ausdrücke definiert ist, genügt es zu zeigen, dass nur endlich viele $der_v(r)$ mit $v \in \textit{Prefix}(L(r))$ existieren. Die Menge $der_v(r)$ selbst muss endlich sein, weil r fest und v endlich ist. Mit dem folgenden Satz wird gezeigt, dass die Anzahl der Mengen $der_v(r)$ für $v \in \textit{Prefix}(L(r))$ endlich ist.

Satz 4.5
Sei $L(r)$ eine reguläre Sprache mit $r \in \textit{Reg}$ und $v, w \in \textit{Prefix}(L(r))$, dann gilt:

$$der_v(r) = der_w(r) \Leftrightarrow v \sim w$$

Beweis:

"⇒" Hinrichtung: Indirekter Beweis

Angenommen es gilt $v \nsim w$, dann folgt:

$$\overset{\text{Def.}\nsim}{\Leftrightarrow} \quad \exists x \in \bigcup_{s_i \in der_v(r)} L(s_i) \text{ mit } v \cdot x \in L(r) \text{ und } w \cdot x \notin L(r)$$

$$\overset{\text{Hilfss. 4.2}}{\Rightarrow} \quad x \in \bigcup_{s_i^1 \in der_v(r)} L(s_i^1) \text{ und } x \notin \bigcup_{s_i^2 \in der_w(r)} L(s_i^2)$$

$$\overset{\text{Def.}=}{\Rightarrow} \quad der_v(r) \neq der_w(r)$$

"⇐" Rückrichtung: Direkter Beweis

Angenommen es gilt $der_v(r) = der_w(r)$, dann folgt:

$$\overset{\text{Def.}=}{\Leftrightarrow} \quad s_i \in der_v(r) \Leftrightarrow s_i \in der_w(r)$$

$$\overset{\text{Hilfss. 4.2}}{\Rightarrow} \quad \forall x \in \bigcup_{s_i \in der_v(r)} L(s_i) \text{ gilt } v \cdot x \in L(r) \text{ und } w \cdot x \in L(r)$$

$$\overset{\text{Def.}\sim}{\Leftrightarrow} \quad v \sim w$$

□

Der Beweis, der in die eine Richtung indirekt und in die andere direkt erfolgt, macht sich erneut den Hilfssatz 4.2 zu Nutze.

Abschließend kann anhand der Nachweise für die Korrektheit, Vollständigkeit und Terminierung des Subtyp-Algorithmus festgestellt werden, dass der Algorithmus wie erwartet arbeitet.

4.8 Erweiterungen und Vereinfachungen

Dieser Abschnitt beschreibt zunächst die Erweiterung des Subtyp-Algorithmus auf die weiterführenden Konzepte von XML-Schema. Im Anschluss folgt eine Diskussion zu möglichen Vereinfachungen des Algorithmus für die bestehenden Definitionen von DTD und XML-Schema unter Berücksichtigung der Auswirkungen auf das Laufzeitverhalten.

4.8.1 Substitutionsgruppen, Typerweiterung und Typeinschränkung

Wie im Abschnitt 2.1 angesprochen wurde, erweitert XML-Schema die Ausdruckskraft von DTD unter anderem durch die Mechanismen Substitutionsgruppen, Typerweiterung und Typeinschränkung. Dadurch wird das Typsystem in XML durch die sogenannte *Bezeichnertypisierung* („named typing“) ergänzt. In der Bezeichnertypisierung gibt es die Möglichkeiten Bezeichner mit

Typdefinitionen zu verknüpfen und zwischen diesen Bezeichnern Beziehungen, wie zum Beispiel eine Vererbungshierarchie, zu deklarieren. In XML-Schema ist dies mit den komplexen Typen durch Substitutionsgruppen, Typerweiterung und Typeinschränkung möglich. Neben der Bezeichnertypisierung existiert in XML die *Strukturtypisierung* („structural typing“). In dieser wird anhand der Struktur von XML-Fragmenten und Inhaltsmodellen eine Beziehung hergestellt.

In Substitutionsgruppen werden Elementnamen gleicher Elementtypen zu einer Menge zusammengefasst. Für ein Dokument ist dann zulässig, dass die Instanz eines Elementtyps aus der Substitutionsgruppe an einer Position im Dokument auftritt, an dem ein anderer Elementtyp aus der Substitutionsgruppe erwartet wird. Zur Formalisierung von Substitutionsgruppen wird im Typsystem von XOBE die reflexive und transitive Substitutionsgruppenrelation *SubGr* eingeführt. Für das zweite Konzept in XML-Schema, der Erweiterung und Einschränkung komplexer Typen, gilt ähnliches. Durch diese dürfen Instanzen der erweiterten oder eingeschränkten Typen an Stellen im Dokument eingesetzt werden, an denen die nicht erweiterten oder uneingeschränkten Typen erwartet werden. Für die formale Behandlung im Subtyp-Algorithmus von XOBE wird zusätzlich die Bezeichnertyprelation *Inh* definiert.

Im Folgenden wird die Abkürzung BZT für die modifizierten Definitionen verwendet, die durch die Konzepte der Bezeichnertypisierung entstehen. Es ergeben sich für die Formalisierung eines XML-Schemas folgende zusätzliche Regeln.

Definition 4.44 (Produktionsrelation (BZT))
Die *Produktionsrelation (BZT)* ergänzt die Produktionsrelation $\mapsto$ aus Definition 4.16 um folgende Regeln:

$$\frac{i \hookrightarrow e, t \hookrightarrow r, s \hookrightarrow n}{\begin{array}{l}\texttt{<element } ag\texttt{/>} \\ \texttt{@name} = i, \texttt{@type} = t, \\ \texttt{@substitutionGroup} = s\end{array} \quad \mapsto e \to e[r] \text{ und } (e \leq n \in \textit{SubGr})} \tag{SUB1}$$

$$\frac{i \hookrightarrow e, cm \hookrightarrow r, s \hookrightarrow n}{\begin{array}{l}\texttt{<element } ag\texttt{>}cm\texttt{</element>} \\ \texttt{@name} = i, \texttt{@substitutionGroup} = s\end{array} \quad \mapsto e \to e[r] \text{ und } (e \leq n \in \textit{SubGr})} \tag{SUB2}$$

$$\frac{i \hookrightarrow o, t \hookrightarrow n, cm \hookrightarrow r_{cm}, am \hookrightarrow r_{am}}{\begin{array}{l}\texttt{<complexType } ag\texttt{><complexContent>} \\ \quad \texttt{<extension base="}t\texttt{">} \\ \quad\quad cm\ am \\ \quad \texttt{</extension>} \\ \texttt{</complexContent></complexType>} \\ \texttt{@name} = i, \texttt{@mixed} = b\end{array} \quad \begin{array}{l} \\ \\ \\ \\ \mapsto \\ \\ \\ \\ \\ \\ \end{array} \begin{array}{l} o \to r_{am_n} \& r_{am}; \textit{mixed}(r_{cm_n}; r_{cm}, b) \\ \text{mit } n \to r_{am_n}; r_{cm_n} \in P \\ \text{und } (o \leq n \in \textit{Inh})\end{array}} \tag{EXT}$$

$$\frac{i \hookrightarrow o, t \hookrightarrow n, cm \hookrightarrow r_{cm}, am \hookrightarrow r_{am}}{\begin{array}{l}\texttt{<complexType } ag\texttt{><complexContent>} \\ \quad\texttt{<restriction base="}t\texttt{">} \\ \quad\quad cm\ am \\ \quad\texttt{</restriction>} \\ \texttt{</complexContent></complexType>} \\ \texttt{@name} = i, \texttt{@mixed} = b\end{array} \quad \mapsto \quad \begin{array}{l} o \to r_{am_n} \& r_{am}; \textit{mixed}(r_{cm}, b) \\ \text{mit } n \to r_{am_n}; r_{cm_n} \in P \\ \text{und } (o \leq n \in \textit{Inh})\end{array}} \quad \text{(REST)}$$

mit $b \in \mathcal{B}$, $e, o, n \in N$, $e \in E$ und $r, r_{cm}, r_{am}, r_{am_{t'}}, r_{cm_{t'}} \in \textit{Reg}$. □

Die Regel SUB1 und SUB2 erzeugen für Elementdeklarationen, die Substitutionsgruppen zugeordnet sind, Produktionen und erweitern die Substitutionsgruppenrelation *SubGr*. Mit den Regeln EXT und REST werden die Produktionen für Definitionen komplexer Typen mit Erweiterung beziehungsweise Einschränkung erstellt. Die notwendige Ausdehnung der Bezeichnersubtyprelation *Inh* wird ebenfalls vorgenommen.

Nachdem die erweiterte Sprachbeschreibung ergänzend zur Heckengrammatik mit eine Bezeichnersubtyprelation formalisiert wird, muss eine Regel für die Typinferenz eines XML-Konstruktors ergänzt werden.

Definition 4.45 (Typinferenz eines XML-Konstruktors (BZT))
Die *Typinferenz eines* XML*-Konstruktors (BZT)* ergänzt Definition 4.18 um folgende Regel:

$$\frac{\begin{array}{c} id \hookrightarrow e, t \hookrightarrow n \\ \Gamma \vdash am : r_{am}, \\ \Gamma \vdash cm : r_{cm}, \\ e[r_{am}; r_{cm}] \leq n \end{array}}{\Gamma \vdash \begin{array}{l}\texttt{<}id\ am\texttt{>}cm\texttt{</}id\texttt{>} \\ \texttt{@type} = t\end{array} : n} \quad \text{(ELEM INH)}$$

mit $n \in N$, $e \in E$, $r_{am}, r_{cm} \in Reg$ und der Ausdrucksrelation $\hookrightarrow$. □

Die Regel ELEM INH bezieht sich auf Elemente, die durch das Attribut `type` ihren aktuellen Typ annotieren. Für diese Elemente wird der angegebene Typ n als Typ des Konstruktors zurückgegeben und gleichzeitig mit dem Subtyp-Algorithmus verifiziert, ob der inferierte Typ des Elements $e[r_{am}; r_{cm}]$ zum annotierten Typ passt.

Weiterhin ist die Anpassung des Algorithmus zur Berechnung der Subtyp-Beziehung notwendig. Die Idee bei der Integration der Bezeichnersubtyprelation in den Algorithmus ist, dass die regulären Ungleichungen nicht nur mittels führender Terminalsymbole, sondern, falls möglich, auch durch führende Nichtterminalsymbole reduziert werden. Bei diesem Vorgehen ist die Berücksichtigung der Vorgaben durch die Bezeichnersubtyprelation möglich.

Notwendig ist die Neudefinition der im Abschnitt 4.2 eingeführten Funktion *term*. Anstatt bei

einem Nichtterminalsymbol die Produktionsdefinition zu analysieren, beendet die neue Variante diesen Rekursionszweig.

Definition 4.46 (Führende Terminalsymbole (BZT))
Die *führenden Terminalsymbole (BZT)* eines regulären Ausdrucks $r \in \mathit{Reg}$ liefert die modifizierte Funktion $\mathit{term} : \mathit{Reg} \to \mathcal{P}[T]$ aus Definition 4.11. Die Änderung ist definiert für Nichtterminalsymbole:

$$\mathit{term}(n) = \{\}$$

mit $n \in N$. □

Zusätzlich wird eine Funktion non definiert, die analog zur Funktion *term* die Menge der Nichtterminalsymbole aus einem regulären Ausdruck ermittelt.

Definition 4.47 (Führende Nichtterminalsymbole)
Die *führenden Nichtterminalsymbole* eines regulären Ausdrucks $r \in \mathit{Reg}$ liefert die Funktion $\mathit{non} : \mathit{Reg} \to \mathcal{P}[N]$ und sei analog zur Funktion *term* aus Definition 4.11 mit folgenden Änderungen für Nichtterminalsymbole und Basisdatentypen definiert:

$$\mathit{non}(b) = \{\}$$
$$\mathit{non}(n) = \{n\}$$

mit $b \in B$ und $n \in N$. □

Die in Abschnitt 4.6 eingeführte Funktion *der* muss ebenfalls abgewandelt werden. Die Funktion liegt nun in zwei Varianten vor, wobei die eine für Terminalzeichen und die andere für Nichtterminalsymbole definiert ist. Die Funktion *der* für Terminalsymbole berücksichtigt dabei die Angaben der Substitutionsgruppenrelation *SubGr*.

Definition 4.48 (Partielle Ableitung hinsichtlich eines Terminalsymbols (BZT))
Die *partielle Ableitung* hinsichtlich des Terminalsymbols $x \in T$ wird berechnet durch die Funktion $\mathit{der}_x : \mathit{Reg} \to \mathcal{P}[\mathit{Reg} \times \mathit{Reg}]$. Sie ist definiert wie die Funktion *der* aus Definition 4.31 mit folgender Änderung für Elementtypen:

$$\mathit{der}_x(e[r]) \quad = \quad \begin{cases} \{(r, \epsilon)\} & \text{falls } e \leq x \in \mathit{SubGr} \text{ und } x \in E, \\ \{\} & \text{sonst} \end{cases}$$

mit $e \in E$ und $r \in \mathit{Reg}$. □

Bei der Festlegung der Funktion *der* für Nichtterminalsymbole wird entsprechend die Bezeichnersubtyprelation einbezogen. Zusätzlich kann ein Basisdatentyp oder ein Elementname nicht um ein Nichtterminalsymbol reduziert werden, weshalb in diesen Fällen eine leere Menge das Ergebnis bildet.

Definition 4.49 (Partielle Ableitung hinsichtlich eines Nichtterminalsymbols)
Die *partielle Ableitung* hinsichtlich des Nichtterminalsymbols $x \in N$ wird berechnet durch die Funktion $\mathit{der}_x : \mathit{Reg} \to \mathcal{P}[\mathit{Reg} \times \mathit{Reg}]$. Sie ist definiert wie die Funktion *der* aus Definition 4.31

mit folgender Änderung für Basisdatentypen, Nichtterminalsymbole und Elementnamen:

$$
\begin{aligned}
\mathit{der}_x(b) &= \emptyset \\
\mathit{der}_x(n) &= \begin{cases} \{(\epsilon, \epsilon)\} & \text{falls } x = n \text{ oder } x \leq n \in \mathit{Inh}, \\ \mathit{der}_x(r) \text{ mit } n \to r \in P & \text{sonst} \end{cases} \\
\mathit{der}_x(e[r]) &= \emptyset
\end{aligned}
$$

für alle $b \in B$, $n \in N$, $e \in E$ und $r \in \mathit{Reg}$. □

Da eine Reduktion einer regulären Ungleichung mit einem Nichtterminalsymbol nicht immer erfolgreich sein muss, ist es sinnvoll in diesen Fällen ein führendes Nichtterminalsymbol durch dessen Produktionsdefinition zu ersetzen, um anschließend die Reduktion erneut zu versuchen. Dazu wird eine Funktion *sub* spezifiziert.

Definition 4.50 (Substitution führender Nichtterminalsymbole)
Die *Substitution* eines führenden Nichtterminalsymbols $x \in N$ durch die Produktionsdefinition von x wird berechnet durch die Funktion $\mathit{sub}_x : \mathit{Reg} \to \mathit{Reg}$ und ist rekursiv definiert durch:

$$
\begin{aligned}
\mathit{sub}_x(\emptyset) &= \emptyset \\
\mathit{sub}_x(\epsilon) &= \epsilon \\
\mathit{sub}_x(b) &= b \\
\mathit{sub}_x(n) &= \begin{cases} r \text{ mit } n \to r \in P & \text{falls } n = x \\ n & \text{sonst} \end{cases} \\
\mathit{sub}_x(e[r]) &= e[r] \\
\mathit{sub}_x(r|s) &= \mathit{sub}_x(r)|\mathit{sub}_x(s) \\
\mathit{sub}_x(r; s) &= \begin{cases} \mathit{sub}_x(r); s & \text{falls } \neg \mathit{isNullable?}(r) \\ \mathit{sub}_x(r); \mathit{sub}_x(s) & \text{falls } \mathit{isNullable?}(r) \end{cases} \\
\mathit{sub}_x(r*) &= \mathit{sub}_x(r)*
\end{aligned}
$$

mit $b \in B$, $n \in N$, $e \in E$ und $r, s \in \mathit{Reg}$. □

Mit diesen Angaben ist es nun möglich, den Subtyp-Algorithmus aus Abschnitt 4.6 entsprechend umzuschreiben.

Definition 4.51 (Subtyp-Algorithmus (BZT))
Der *Subtyp-Algorithmus (BZT)* ersetzt die Regel REC aus Definition 4.34 durch folgende Regel:

$$\frac{\begin{array}{c}\neg \mathit{inc}(r \leq s),\\ \\ \text{für alle } i \in \{1,\ldots,k\} \text{ mit } k = |\mathit{part}_x(r \leq s)| \text{ und } x \in \mathit{term}(r) \text{ gilt}\\ \Gamma_{i-1} \vdash c_r \leq c_s \Rightarrow \Gamma_i \vee \Gamma_{i-1} \vdash r_r \leq r_s \Rightarrow \Gamma_i\\ \text{mit } (c_r \leq c_s \vee r_r \leq r_s) \in \mathit{part}_x(r \leq s),\\ \\ (\text{für alle } j \in \{1,\ldots,l\} \text{ mit } l = |\mathit{part}_x(r \leq s)| \text{ und } x \in \mathit{non}(r) \text{ gilt}\\ \Gamma_{k+j-1} \vdash c_r \leq c_s \Rightarrow \Gamma_{k+j} \vee \Gamma_{k+j-1} \vdash r_r \leq r_s \Rightarrow \Gamma_{k+j}\\ \text{mit } (c_r \leq c_s \vee r_r \leq r_s) \in \mathit{part}_x(r \leq s))\\ \vee\ \Gamma_k \vdash \mathit{sub}_n(r) \leq s \Rightarrow \Gamma_{k+l} \text{ mit } n \to t \in P\end{array}}{\Gamma_0 \vdash^* r \leq s \Rightarrow \Gamma_{k+l}} \text{(REC)}$$

□

Wie Regel REC zeigt, extrahiert der Algorithmus für eine reguläre Ungleichung zunächst die führenden Terminalzeichen, als auch die führenden Nichtterminalsymbole. Für die führenden Terminalzeichen wird die Ungleichung, wie in Definition 4.34, reduziert. Analoges erfolgt für die Nichtterminalzeichen, wobei für den Fall, dass diese Reduktion zu keinem Ergebnis führt, versucht wird, durch die Substitution des führenden Nichtterminalsymbols zum Erfolg zu gelangen.

4.8.2 Vereinfachungen

Sowohl in der Spezifikation von XML [W3C98c], als auch in der Spezifikation von XML-Schema [W3C01c] werden für die Definition von XML-Typen Einschränkungen festgelegt. Dies führt dazu, dass die Heckengrammatiken, die durch die Formalisierung von DTDs und XML-Schemata entstehen, bestimmte Eigenschaften aufweisen, die zu wesentlichen Vereinfachungen des Subtyp-Algorithmus führen. Verbunden damit ist eine Verbesserung des Laufzeitverhalten um Größenordnungen.

Elementtypen in Inhaltsmodellen

In XML-Schema gilt für die Deklaration von Elementtypen, dass Typen mit dem selben Elementnamen innerhalb eines Inhaltsmodells gleich zu sein haben. Damit müssen sowohl die Inhaltsmodelle, als auch die Attribute der Elementtypen identisch sein. In DTDs gilt diese Bedingung ohnehin, da für jeden Elementnamen nur ein Elementtyp definiert werden kann. Das folgende Beispiel zeigt eine Produktion, die die genannte Anforderung verletzt:

Beispiel 4.12
Sei in dem XML-Schema der AOML (Anhang A) die Definition des komplexen Typs `t_book` wie folgt ersetzt:

```
<complexType name="t_book">
  <sequence>
    <element name="title" type="integer"/>
    <element name="title" type="string"/>
  </sequence>
</complexType>
```

Die Instanzen des komplexen Typs `t_book` müssen nun nach dieser Definition aus zwei Elementen bestehen, die beide den Elementnamen `title` tragen, aber unterschiedlichen Inhalt besitzen. Der Inhalt des ersten Elements ist von Typ `integer`, während das zweite Element eine Zeichenkette beinhaltet. Damit sind die Typen dieser beiden Elemente verschieden, was der geschilderten Bedingung widerspricht. □

Für die durch die Sprachbeschreibung induzierte Heckengrammatik kann die Bedingung formal formuliert werden.

Definition 4.52
Für alle Produktionen $n \to r \in P$ einer Heckengrammatik mit *Inhaltsmodellen, die für gleiche Elementnamen nur identische Elementtypen zulassen*, gilt:

$$c_1 = c_2 \text{ für } (c_1, r_1) \in \mathit{der}_v(r) \text{ und } (c_2, r_2) \in \mathit{der}_v(r)$$

mit $v \in T^+$, $n \in N$ und $r, c_1, c_2, r_1, r_2 \in \mathit{Reg}$. □

Die Definition zeigt für die partiellen Ableitungen eines regulären Ausdrucks, dass ausschließlich Tupel entstehen, die sich in den ersten Komponenten (c_1 und c_2) nicht unterscheiden. Damit sind reguläre Ungleichungen wie beispielsweise

$$\mathbf{book}[\mathit{title}; \mathit{price}?]* \leq \mathbf{book}[\mathit{title}] * \mid \mathbf{book}[\mathit{title}]*; \mathbf{book}[\mathit{title}; \mathit{price}]; \mathbf{book}[\mathit{title}; \mathit{price}?]*$$

nicht mehr zugelassen. Durch diese Eigenschaft lässt sich die Definition der partiellen Ableitungen erheblich einfacher angeben.

Definition 4.53 (Partielle Ableitung regulärer Ungleichungen (Simp1))
Die *partielle Ableitung* einer regulären Ungleichung $r \leq s$ mit $r, s \in \mathit{Reg}$ hinsichtlich eines Terminalsymbols $x \in T$ ist definiert durch:

$$\mathit{part}_x(r \leq s) = \{(c_r \leq c_s) \ \vee \ (\epsilon \leq \emptyset), (\epsilon \leq \emptyset) \ \vee \ (r_r \leq \Big| \limits_{i \in \{1,\ldots,n\}} r_s^i) | \\ (c_r, r_r) \in \mathit{der}_x(r) \ \wedge \ \mathit{der}_x(s) = \{(c_s, r_s^1), \ldots, (c_s, r_s^n)\}\}$$

mit $c_r, c_s, r_r, r_s \in \mathit{Reg}$. □

Für die partielle Ableitung einer regulären Ungleichung entstehen durch die Einschränkung nur noch zwei Disjunktionen, von denen eine Ungleichung ($\epsilon \leq \emptyset$) stets trivial inkonsistent ist. Die

beiden anderen Ungleichungen beziehen sich auf das Inhaltsmodell des reduzierten Terminalzeichens ($c_r \leq c_s$) und auf die nachfolgenden Elementtypen ($r_r \leq \left|_{i\in\{1,\ldots,n\}} r_s^i\right.$).

Im schlechtesten Fall ist der Algorithmus aus Abschnitt 4.6 exponentiell, wofür es zwei Ursachen gibt. Die Anzahl der regulären Ungleichungen kann sich zum einen durch die Tupelmenge der partiellen Ableitungen der regulären Ausdrücke exponentiell vergrößern und zum zweiten durch die Anwendung von Satz 4.2 erweitern. Durch die Einschränkung einer Sprachbeschreibung auf Inhaltsmodelle, für die nur gleiche Elementtypen mit gleichem Elementnamen zulässig sind, ergibt sich bezüglich des Aufwands eine Verbesserung. Die Anwendung von Satz 4.2 ist nun nicht mehr erforderlich.[6] Trotzdem bleibt der Aufwand exponentiell und entspricht nun genau dem Aufwand von Antimitovs Algorithmus, der PSPACE-vollständig ist [Ant94].

Einseindeutigkeit

Die *Einseindeutigkeit* („one-unambiguous“) [BKW98] ist eine weitere Einschränkung an die Inhaltsmodelle, die sowohl für eine DTD („deterministic content models“) (Referenz E in der Spezifikation [W3C98c]) als auch für ein XML-Schema („unique particle attribution“) (§3.8.6 in [W3C01c]) gefordert wird. Es wird von einem einseindeutigen Inhaltsmodell gesprochen, falls es möglich ist unter Berücksichtigung der Konkatenations-, Vereinigungs- und Kleene-Stern-Operatoren einen Pfad durch das Inhaltsmodell so zu bestimmen, dass jedes Element im Inhalt eines Dokuments mit einem Elementtyp im Inhaltsmodell korrespondiert. Kann ein Element im Inhalt mit mehr als einem Elementtypen im Inhaltsmodell korrespondieren, ist die Eigenschaft verletzt. Das folgende Beispiel zeigt ein nicht einseindeutiges Inhaltsmodell.

Beispiel 4.13
Dieses Beispiel bezieht sich auf die DTD der AOML (Beispiel 2.2) und zeigt ein nicht einseindeutiges Inhaltsmodell.

```
<!ELEMENT book        ((title, author)|(title, editor))>
```

Bei der Verarbeitung des Inhalts eines `book`-Elements wäre für das erste `title`-Element unklar mit welchem Elementtyp `title` im Inhaltsmodell dieses korrespondiert. Nur mit einer Vorausschau auf das dem `title`-Element folgende Element wäre eine eindeutige Zuordnung möglich. □

Bei vielen nicht einseindeutigen Inhaltsmodellen ist eine Umformung in einen äquivalenten einseindeutigen Ausdruck möglich. Aber nicht immer kann ein solcher Ausdruck gefunden werden, was in [BKW98] festgestellt wurde. Beispielsweise ist für das Inhaltsmodell `((title|author)*;title;(title|author))`, das die Einseindeutigkeit verletzt, keine äquivalente Umwandlung in einen einseindeutigen Ausdruck möglich.

[6] Eine weitere Verbesserung des Algorithmus ist in diesem Fall dadurch möglich, dass der &-Operator für Attribute oder Inhaltsmodelle nicht durch sämtliche Permutationen sondern eigenständig repräsentiert wird.

Die Formalisierung dieser Eigenschaft für Heckengrammatiken ist mit Hilfe der partiellen Ableitungen möglich.

Definition 4.54
Für alle Produktionen $n \to r \in P$ einer Heckengrammatik mit *einseindeutigen Inhaltsmodellen* gilt:

$$|\mathit{der}_v(r)| = 1$$

mit $v \in T^+$, $n \in N$ und $r \in \mathit{Reg}$. □

Die Charakterisierung für einseindeutige Inhaltsmodelle führt zu einelementigen partiellen Ableitungen der regulären Ausdrücke. Damit kann für die partiellen Ableitungen der regulären Ungleichungen eine weiter vereinfachte Definition angegeben werden.

Definition 4.55 (Partielle Ableitung regulärer Ungleichungen (Simp2))
Die *partielle Ableitung* einer regulären Ungleichung $r \leq s$ mit $r, s \in \mathit{Reg}$ hinsichtlich eines Terminalsymbols $x \in T$ ist definiert durch:

$$\mathit{part}_x(r \leq s) = \{(c_r \leq c_s) \ \vee \ (\epsilon \leq \emptyset), (\epsilon \leq \emptyset) \ \vee \ (r_r \leq r_s) | \\ \mathit{der}_x(r) = \{(c_r, r_r)\} \ \wedge \ \mathit{der}_x(s) = \{(c_s, r_s)\}\}$$

mit $c_r, c_s, r_r, r_s \in \mathit{Reg}$. □

Die partielle Ableitung einer regulären Ungleichung vereinfacht sich in der Form, dass nun in der Ungleichung für nachfolgende Elementtypen ($r_r \leq r_s$) des reduzierten Terminalzeichens keine reguläre Vereinigung über sämtliche Tupel aus $\mathit{der}_x(s)$ gebildet werden muss. Die Menge $\mathit{der}_x(s)$ enthält nämlich nur ein Element.

Mit der Beschränkung der Inhaltsmodelle einer Sprachbeschreibung auf einseindeutige Inhaltsmodelle verbessert sich der Aufwand des Subtyp-Algorithmus wesentlich. Da die partiellen Ableitungen der regulären Ausdrücke nun nur noch aus einem Tupel bestehen, entstehen in jedem Reduktionsschritt exakt zwei neue Ungleichungen. Der Aufwand wird damit linear zur Anzahl der Terminalsymbole in beiden regulären Ausdrücken. Dies ist eine Verbesserung um Größenordnungen.

Es hat sich gezeigt, dass die Einschränkungen der Inhaltsmodelle zu wesentlichen Effizienzsteigerungen des Subtyp-Algorithmus führen. Da sowohl von DTDs als auch von XML-Schema die genannten Beschränkungen gefordert werden, können XOBE-Programme mit diesen Sprachbeschreibungen davon profitieren. Es wäre aber auch denkbar, die Forderung der Einseindeutigkeit für Inhaltsmodelle aufzugeben. Damit wäre der Aufwand des Subtyp-Algorithmus zwar nicht mehr linear, was aber für die relativ kleinen Inhaltsmodelle realistischer Sprachbeschreibungen, die auch angewendet werden, vertretbar wäre.

Kapitel 5

Übersetzung von XOBE-Programmen

Um aus einem Programm ein ausführbares Programm zu erhalten, ist eine Programmübersetzung notwendig. Ein XOBE-Programm ist ein Java-Programm, das, wie in Kapitel 3 eingeführt, im Wesentlichen um XML-Objekte und XPath-Ausdrücke zur Selektion von Daten aus XML-Objekten ergänzt wurde. Bedingt durch diese Erweiterungen ist eine Verarbeitung mit einem Java-Übersetzer weder syntaktisch möglich, noch kann damit die Typkorrektheit, wie sie in Kapitel 4 vorgestellt wurde, überprüft werden.

Es ist eine eigene Übersetzung der XOBE-Programme notwendig, für die im Rahmen dieser Arbeit eine prototypische Implementierung vorgenommen wurde. Diese besteht aus einem Präprozessor, der das XOBE-Programm in ein reines Java-Programm übersetzt, nachdem die Typkorrektheit festgestellt wurde. Das erzeugte reine Java-Programm verwendet zur internen Repräsentation der XML-Objekte das in Abschnitt 2.3 präsentierte DOM.

Im Folgenden wird zunächst die Architektur des Übersetzungsprozesses dargestellt, der aus einem XOBE-Programm ein reines Java-Programm erzeugt. Der nachfolgende Abschnitt definiert die Transformation der XML-Objekte im Einzelnen. Daraufhin wird in einem Abschnitt die Transformation der XPath-Ausdrücke mit zusätzlichen Algorithmen, die für die Umsetzung notwendig sind, vorgestellt. Abschließend werden Erfahrungen und Leistungsdaten diskutiert, sowie auf Erweiterungsmöglichkeiten der Implementierung eingegangen.

5.1 Architektur des Präprozessors

Für die Übersetzung der XOBE-Programme wurde eine Implementierung als Präprozessor gewählt. Diese Umsetzung geschieht nicht aus zwingender Notwendigkeit, sondern stellt eine Designentscheidung dar. Denkbar wäre ebenfalls eine Integration der XOBE-Spracherweiterung in einen Standard-Java-Übersetzer, eine Möglichkeit die im letzten Abschnitt dieses Kapitels kurz diskutiert wird. In diesem Abschnitt wird die Architektur des Prototyps vorgestellt, die in Abbil-

dung 5.1 schematisch dargestellt ist. Wie zu sehen ist, gliedert sich der XOBE-Präprozessor in

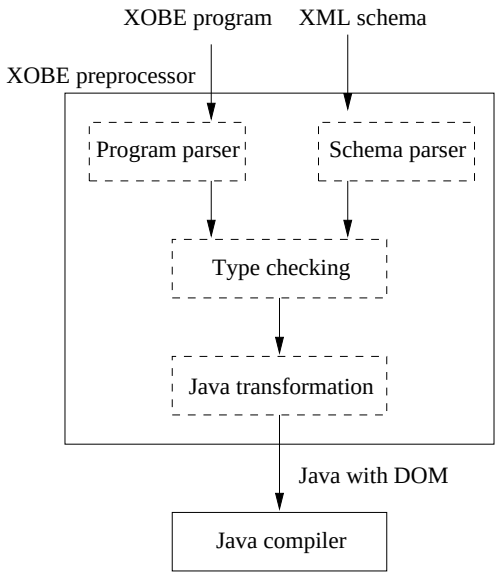

Abbildung 5.1: Architektur des XOBE-Präprozessors

drei aufeinander folgende Phasen:

1. Der lexikalischen und syntaktischen Analyse,
2. der Typanalyse und
3. der Transformation in reinen Java-Quelltext.

In der lexikalischen und syntaktischen Analyse wird das XOBE-Programm eingelesen und in eine interne Repräsentation überführt. Ein XOBE-Programm besteht neben den Standard-Java-Sprachkonstrukten aus der leicht modifizierten XML-Syntax der XML-Konstruktoren, aus der XPath-Syntax und aus der XML-Schema- bzw. DTD-Syntax der importierten Sprachbeschreibungen. Die XML-Syntax ist leicht verändert, weil die XML-Konstruktoren Variablen enthalten können, was in der standardisierten XML-Syntax nicht vorgesehen ist. Es kommen damit in der ersten Phase fünf unterschiedliche Syntaxregeln zum Einsatz. Die Standard-Java-, die XML- und die XPath-Regeln werden zu einem *XOBE-Programmparser* zusammengefasst. Die Regeln aus XML-Schema und der DTD-Syntax finden Eingang in den *XOBE-Schemaparser*. In der vorliegenden prototypischen Implementierung wird mit Hilfe des Compiler-Compilers JavaCC [Web02] der XOBE-Programmparser generiert. Für den XOBE-Schemaparser wird einerseits der XML-Parser Xerces [Apa01] eingesetzt, um die XML-Schema-Syntax zu analysieren, andererseits wird der DTD-Parser von Wuttka [Wut] verwendet, der die Sprachbeschreibungen in DTD-Syntax einliest. Die Festlegung der internen Repräsentation für eingelesene XOBE-Programme geschieht mit Unterstützung des Java-Tree-Builders (JTB) [TWP00], der aus Syntaxregeln eine abstrakte Syntax generiert.

In der Phase der *Typanalyse* überprüft der Präprozessor, ob es sich bei dem eingelesenen Quelltext um ein wohlgetyptes Programm handelt. Ein XOBE-Programm ist wohlgetypt, falls alle XML-Objekte und XPath-Ausdrücke gültig bezüglich der deklarierten Sprachbeschreibung sind. Vorab aber muss die Typanalyse untersuchen, ob die importierten Sprachbeschreibungen selbst korrekt sind. Danach erfolgt die Typüberprüfung der Java-Anweisungen, die XML-Objekte oder XPath-Ausdrücke enthalten, wie beispielsweise Zuweisungen und Methodenaufrufe, gemäß der Beschreibung von Kapitel 4. Wie dort beschrieben ist, findet zunächst eine Typinferenz für XML-Konstruktoren, XML-Objekt-Variablen und XPath-Ausdrücke statt. Anschließend werden mit dem Subtyp-Algorithmus die durch XOBE erweiterten Anweisungen verifiziert. Das Typsystem einer Auszeichnungssprache, die eine DTD beschreibt, ist sehr strikt und lässt sich vollständig durch reguläre Heckengrammatiken, wie sie in Abschnitt 4.2 beschrieben wurden, formalisieren. Damit kann der Subtyp-Algorithmus aus Abschnitt 4.6 zur Typüberprüfung verwenden werden. In XML-Schema wird dieses strikte Typsystem durch Typerweiterungen und Typeinschränkungen (siehe Abschnitt 2.1.2) aufgeweicht. Trotzdem können mit Hilfe der Erweiterung des Suptyp-Algorithmus aus Abschnitt 4.8 auch diese Anweisungen überprüft werden.

In der letzten Phase des Präprozessors wird die *Transformation* des XOBE-Programms in reinen Java-Quelltext durchgeführt. Für die Implementierung der XOBE-Konstrukte in reinem Java-Quelltext sind mehrere verschiedene Alternativen denkbar, die mit unterschiedlichen XML-Objekt-Repräsentationen arbeiten. Die vorliegende Realisation [Kra02] verwendet die Standardrepräsentation des DOM, das in Abschnitt 2.3 vorgestellt wurde. Auf weitere Implementierungsmöglichkeiten geht die Diskussion im letzten Abschnitt dieses Kapitels kurz ein. Die Transformation ersetzt die XML-Konstruktoren und XPath-Ausdrücke des XOBE-Programms durch passende DOM-Anweisungen. Sie erfolgt auf der internen Repräsentation des Quelltextes. Es werden die Teilbäume, die die XOBE-Konstrukte repräsentieren, durch solche neu erzeugten Teilbäume ersetzt, die für den passenden DOM-Quelltext stehen. Das Transformationsresultat, der reine Java-Quelltext, kann nach der Ausgabe mit einem Standard-Java-Übersetzer verarbeitet werden, womit im Anschluss an die Transformation des Präprozessors die Umwandlung in ein lauffähiges Programm erfolgen kann. Obwohl das DOM als ungetypte XML-Implementierung die statische Gültigkeit der Elemente selbst nicht sicherstellt, sind die transformierten XML-Objekte des XOBE-Programms gültig. Dies gilt, weil die Typanalyse des Präprozessors diese Eigenschaft für den resultierenden Java-Quelltext garantiert.

Die Notation, die in den folgenden Abschnitten verwendet wird, dient zur einfachen Darstellung von Transformationen. Eine Transformationsvorschrift wird dabei durch eine Regel der Form $[\![s]\!]^p_r \Rightarrow t$ beschrieben. Dabei entspricht s einem Ausdruck der Quellsprache, aus dem der Ausdruck t in der Zielsprache generiert wird. Die Parameter-Annotation p der Vorschrift steht für Werte, die für die Transformation notwendig sind, und bei Anwendung mit übergeben werden. Die resultierende Annotation r wird dagegen erst durch die Transformation selbst gesetzt und kann deshalb erst nach der Anwendung der Transformation verwendet werden.

Anmerkung: Vergleichbar sind die Annotation p mit den vererbten („inherited“) Attributen und die Annotation r mit den synthetisierten („synthesized“) Attributen, wie sie von den attributierten Grammatiken [ASU86] her bekannt sind.

5.2 Implementierung für XML-Objekt-Konstruktoren

Für die in Abschnitt 3.2.4 eingeführten Sprachkonstrukte wird in diesem Abschnitt das Vorgehen zur Umsetzung in reinen Java-Quelltext definiert und an kurzen Beispielen illustriert. Die Implementierung erfolgt dabei mit Hilfe der Operationen aus dem DOM, das im Abschnitt 2.3 vorgestellt und formal definiert wurde.

Bei der Umsetzung der XML-Objekt-Konstruktoren und der XPath-Ausdrücke im folgenden Abschnitt wird davon ausgegangen, dass das XOBE-Programm bereits in abstrakter Syntax vorliegt. Damit sind sowohl lexikalische als auch syntaktische Fehler ausgeschlossen. Weiterhin ist die Überprüfung der Typkorrektheit bereits erfolgreich abgeschlossen. Die für diese Aufgabe notwendigen Schritte wurden in Kapitel 4 definiert und diskutiert.

Bei XML-Objekt-Konstruktoren handelt es sich um Ausdrücke, die im XOBE-Programm innerhalb unterschiedlichster Anweisungen auftreten, wie z. B. in Methodenaufrufen, Initialisierungen, Fallunterscheidungen oder Schleifenbedingungen. Die Umwandlung dieser Konstruktoren erzeugt, wie zu sehen sein wird, in der Regel eine Reihe von Java-Anweisungen. Mehrere generierte Anweisungen können aber häufig nicht in die ursprüngliche Anweisung eingesetzt werden. Aus diesem Grund ist eine sogenannte *bedeutungsgleiche* Umsetzung notwendig. Beispielsweise muss bei einem Auftreten eines XML-Objekt-Konstruktors als Parameter eines Methodenaufrufs in der transformierten Implementierung zunächst das DOM-Objekt erzeugt werden, bevor mit diesem dann als Parameter die Methode aufgerufen werden kann. Diese Untersuchung bedeutungsgleicher Umsetzungen wird in dieser Arbeit nicht weiter vertieft, da sie für XML-Objekt-Konstruktoren und XPath-Ausdrücke stets ohne Schwierigkeit erreicht werden kann.

Um teilweise konstruierte DOM-Objekte weiter zu verarbeiten, werden unbenutzte, temporäre Variablen benötigt. In dieser Darstellung wird davon ausgegangen, dass für die Transformation diese in unbegrenzter Menge zur Verfügung stehen. Die prototypische Implementierung stellt dies sicher. Zusätzlich ist für die Umsetzung eine Unterscheidung von XML-Objekt-Variablen und Java-Variablen notwendig, da eine unterschiedliche Verarbeitung erfolgt. Für die Vorschriften der Transformation wird diese Einteilung als gegeben angenommen. Im Prototypen kann diese Information von der vorangehenden Typüberprüfung übernommen werden.

Eine Implementierung von XOBE mittels DOM benötigt weiterhin zur Repräsentation von XML-Objekten stets ein Objekt der DOM-Klasse `Document`. In der folgenden Darstellung wird deshalb angenommen, dass eine solche Instanz stets mit der Deklaration der XML-Objekt-Variablen zur Verfügung steht. Ebenfalls werden für die Deklaration von XML-Objekt-Variablen geeignete DOM-Anweisungen erzeugt, die hier nicht wiedergegeben sind. Für Variablen von Elementklassen werden Variablen der DOM-Klasse `Element` erzeugt und für Elementlisten Variablen der im nächsten Abschnitt eingeführten Schnittstelle `XobeNodeList`. Die Operationen `+`, `item` und `getLength` der Elementlisten, werden auf die entsprechenden Methoden (`addFirst`, `add`, `addAll`, `item` und `getLength`) abgebildet. Eine leere Liste (`<>`) wird durch Anwendung des Konstruktors einer `XobeNodeList`-Implementierung erzeugt.

Die Transformation von XML-Objekt-Konstruktoren ist definiert durch nachfolgende Transformationsvorschriften.

Definition 5.1 (Transformation eines leeren Elements)
Die *Transformation für ein leeres Element* mit dem Elementnamen N, der Attributliste A und der Java-Variablen für ein Dokument d ist definiert durch:

$$[\![\texttt{<}N\ \ A\texttt{/>}]\!]^d_{\mathsf{n}} \Rightarrow \begin{array}{l}\texttt{Element n = }d\texttt{.createElement("}N\texttt{");}\\ [\![A]\!]^{\mathsf{n}}\end{array}$$

Dabei referenziert die Variable n einen Elementknoten. □

Das leere Element wird in eine Java-Anweisung transformiert, die einen Elementknoten mittels der entsprechenden DOM-Methode erzeugt. Danach erfolgt rekursiv die Transformation der Attributliste unter Übergabe des erzeugten Knotens als Parameter-Annotation. Der erzeugte Elementknoten wird als resultierende Annotation zurückgegeben.

Definition 5.2 (Transformation eines nicht leeren Elements)
Die *Transformation für ein nicht leeres Element* mit dem Elementnamen N, der Attributliste A, der Inhaltsliste C und der Java-Variablen für ein Dokument d ist definiert durch:

$$[\![\texttt{<}N\ \ A\texttt{>}\ \ C\ \ \texttt{</}N\texttt{>}]\!]^d_{\mathsf{n}} \Rightarrow \begin{array}{l}\texttt{Element n = }d\texttt{.createElement("}N\texttt{");}\\ [\![A]\!]^{\mathsf{n}}\\ [\![C]\!]^{\mathsf{n}}_{(n_1,\ldots,n_k)}\\ \texttt{n.addChild(}n_1\texttt{);}\\ \vdots\\ \texttt{n.addChild(}n_k\texttt{);}\end{array}$$

Dabei verweisen die Variablen n, $n_1, \ldots n_k$ auf Elementknoten. □

Aus einem nicht leeren Element werden mehrere Anweisung generiert. Zunächst wird für das Element ein Elementknoten erzeugt. Weiter werden die Transformationen für die Attributliste und die Inhaltsliste rekursiv aufgerufen. Die durch die Transformation der Inhaltsliste erzeugten Knoten werden abschließend als Kinder zum Elementknoten hinzugefügt. Der erzeugte Elementknoten wird als resultierende Annotation zurückgegeben.

Definition 5.3 (Transformation einer Inhaltsliste)
Die *Transformation für eine Inhaltsliste* $C_1 \ldots C_k$ mit der Java-Variablen für ein Dokument d ist definiert durch:

$$[\![C_1 \ldots C_k]\!]^d_{(n_1,\ldots,n_k)} \Rightarrow [\![C_1]\!]^d_{n_1} \ldots [\![C_k]\!]^d_{n_k}$$

Dabei referenzieren die Variable $n_1, \ldots n_k$ Elementknoten. □

Für eine Inhaltsliste erfolgt eine rekursive Anwendung der Transformation auf die Elemente der Liste. Die resultierenden Annotationen werden als Resultat zurückgeliefert.

Definition 5.4 (Transformation einer Attributliste)
Die *Transformation für eine Attributliste* $A_1 \ldots A_k$ mit der Java-Variablen für einen Elementknoten n ist definiert durch:

$$[\![A_1 \ldots A_k]\!]^n \Rightarrow [\![A_1]\!]^n \ldots [\![A_k]\!]^n$$

□

Für eine Attributliste erfolgt ebenfalls eine rekursive Anwendung der Transformation auf die Elemente der Liste, die Attribute.

Definition 5.5 (Transformation eines Attributs)
Die *Transformation für ein Attribut* mit dem Attributnamen N, dem Attributwert V und der Java-Variablen für einen Elementknoten n ist definiert durch:

$$[\![N \texttt{ = } V]\!]^n \Rightarrow n\texttt{.setAttribute("}N\texttt{",}[\![V]\!]\texttt{);}$$

□

Für ein Attribut erfolgt das Setzen des Wertes für den angegebenen Attributnamen mittels der entsprechenden Methode aus dem DOM.

Definition 5.6 (Transformation eines Attributwertes)
Die *Transformation für einen konstanten Attributwert* mit den Zeichendaten V ist definiert durch:

$$[\![\texttt{"}V\texttt{"}]\!] \Rightarrow \texttt{"}V\texttt{"}$$

□

Für einen konstanten Attributwert wird eine konstante Java-Zeichenkette gleichen Inhalts generiert.

Definition 5.7 (Transformation von Zeichendaten)
Die *Transformation für Zeichendaten* im Inhalt eines XML-Objekt-Konstruktors D mit der Java-Variablen für ein Dokument d ist definiert durch:

$$[\![D]\!]^d_{\mathsf{n}} \Rightarrow \texttt{Text n = }d\texttt{.createTextNode("}D\texttt{");}$$

Dabei verweist die Variable n auf einen Elementknoten. □

Für Zeichendaten im Inhalt eines Elements wird ein Textknoten erzeugt und als resultierende Annotation zurückgeliefert.

Definition 5.8 (Transformation einer XML-Objekt-Variablen)
Die *Transformation für eine Variable* I als Bezeichner für eine XML-Objekt-Variable mit der Java-Variablen für ein Dokument d ist definiert durch:

$$[\![\texttt{\{}I\texttt{\}}]\!]^d_I \Rightarrow \epsilon$$

□

Für eine XML-Objekt-Variable in einem XML-Objekt-Konstruktor werden keine weiteren Java-Anweisungen erzeugt. Es wird lediglich die Variable selbst als Resultat zurückgegeben.

Definition 5.9 (Transformation einer Java-Variablen)
Die *Transformation für eine Variable* I als Bezeichner für eine Java-Variable im Inhalt eines XML-Objekt-Konstruktors mit der Java-Variablen für ein Dokument d ist definiert durch:

$$[\![\{I\}]\!]^d_{\mathsf{n}} \Rightarrow \texttt{Text n = } d\texttt{.createTextNode("" + } I\texttt{);}$$

Dabei referenziert die Variable n einen Elementknoten. □

Für eine Java-Variable im Inhalt von XML-Objekt-Konstruktoren wird ein Textknoten erzeugt, der den Wert der Variablen als Zeichenkette beinhaltet. Der erzeugte Knoten wird als Resultat zurückgeliefert. Als letzte Transformation wird die Vorschrift für Kommentar in XML angegeben.

Definition 5.10 (Transformation von Kommentar)
Die *Transformation von Kommentar* mit dem Kommentartext D ist definiert durch:

$$[\![\texttt{<!- } D \texttt{ ->}]\!]^d_{\mathsf{n}} \Rightarrow \texttt{Comment n = } d\texttt{.createComment("}D\texttt{");}$$

Dabei referenziert die Variable n einen Elementknoten. □

In einen Kommentarknoten, erzeugt durch die entsprechende DOM-Methode, wird der XML-Kommentar transformiert. Als resultierende Annotation wird der erzeugte Kommentarknoten zurückgegeben.

Die folgenden Beispiele, die die Sprachbeschreibung aus Abschnitt 2.1.2 verwenden, veranschaulichen die formale Transformation durch die Umwandlung von XML-Objekt-Konstruktoren in reinen Java-Quelltext. Das erste Beispiel zeigt die Konvertierung eines einfachen Elements.

Beispiel 5.1
Eine XOBE-Anweisung der Art

```
1 author a = <author>Thomas Mann</author>;
```

wird durch die Transformation in folgende Java-Anweisungen umgewandelt:

```
1 Element a = d.createElement("author");
2 Text t1 = d.createTextNode("Thomas␣Mann");
3 a.addChild(t1);
```

Dabei werden für das Element ein Elementknoten (1) und für den Inhalt, bestehend aus Zeichendaten, ein Textknoten (2) erzeugt. Der Textknoten wird anschließend als Kind des Elementknotens eingefügt (3). □

Im nächsten Beispiel hat ein Element ein Attribut und der Inhalt wird über eine Java-Variable festgelegt.

Beispiel 5.2
Aus den Anweisungen

```
1 float n = 8.00;
2 price p = <price currency="EUR">{n}</price>
```

in XOBE-Syntax wird durch die Transformation die Anweisungsfolge

```
1 float n = 8.00;
2 Element p = d.createElement("price");
3 p.setAttribute("currency","EUR");
4 Text t2 = d.createTextNode("" + n);
5 p.addChild(t2);
```

in Java erzeugt. Der Wert der Java-Variable n wird hier als Zeichenkette in die DOM-Repräsentation eingefügt (4). Das Setzen des Attributwertes (3) erfolgt über die entsprechende Methode aus dem DOM. □

Das letzte Beispiel zur Transformation von XML-Objekt-Konstruktoren beschäftigt sich mit einem komplexeren Element. Es beinhaltet verschachtelte Elemente und mehrere XML-Objekt-Variablen.

Beispiel 5.3
Die XOBE-Anweisung

```
1 book b = <book catalog="Varia">
2              <title>Lotte in Weimar</title>
3              {a}
4              <condition>Einband fingerfleckig , Rücken
                  verblaßt </condition>
5              {p}
6          </book>;
```

transformiert zum Java-Quelltext:

```
1 Element b = d.createElement("book");
2 b.setAttribute("catalog","Varia");
3
4 Element t3 = d.createElement("title");
5 Text t4 = d.createTextNode("Lotte␣in␣Weimar");
6 t3.addChild(t4);
7
8 Element t5 = d.createElement("condition");
9 Text t6 = d.createTextNode("Einband␣fingerfleckig,␣Rücken
     ␣verblaßt");
10 t5.addChild(t6);
```

```
11
12 b.addChild(t3);
13 b.addChild(a);
14 b.addChild(t5);
15 b.addChild(p);
```

Sowohl das äußere Element als auch die Elemente im Inhalt werden als Elementknoten (1,4,8) erzeugt. Die Inhalte der inneren Elemente werden als Textknoten kreiert (5,9) und in die entsprechenden Elementknoten eingefügt. Am Ende wird der Inhalt des äußeren Elements, der aus den inneren Elementen und XML-Objekt-Variablen besteht, durch Kinderknoten zu dem Elementknoten, der das äußere Element repräsentiert, hinzugefügt (12-15). □

5.3 Implementierung der XPath-Ausdrücke

In diesem Abschnitt wird für die in Abschnitt 3.2.6 eingeführten Sprachkonstrukte die Übersetzung in reinen Java-Quelltext spezifiziert und an kurzen Beispielen erläutert. Die Implementierung erfolgt dabei ebenfalls mit den Operationen des DOM aus Abschnitt 2.3.

Die Semantik eines XPath-Ausdrucks ist dahingehend festgelegt, dass durch diesen eine Liste von Knoten selektiert wird. Dies führt in der Implementierung zu einer neuen Klasse für die Repräsentation dieser Liste. Naheliegend ist, dafür die durch die DOM-Schnittstelle `NodeList` eingeführte Liste zu verwenden. Leider ist für diese aber keine Methode zum Hinzufügen von Elementen definiert, weshalb diese Schnittstelle für die hier bestehenden Anforderungen nicht ausreicht. Aus diesem Grund wird für die Implementierung der XOBE-Programme die Schnittstelle `NodeList` unter dem Namen `XobeNodeList` erweitert und um die benötigte Methoden ergänzt.

```
1  public interface XobeNodeList extends NodeList {
2
3    public boolean add(Node node);
4    public boolean addFirst(Node node);
5    public boolean addAll(NodeList nodes);
6
7    public Iterator iterator();
8
9    public XobeNodeList getChildNodes(String test);
10   public XobeNodeList getDescendantNodes(String test);
11   public XobeNodeList getParentNodes(String test);
12   public XobeNodeList getAncestorNodes(String test);
13   public XobeNodeList getFollowingSiblingNodes(String
       test);
14   public XobeNodeList getPrecedingSiblingNodes(String
```

```
        test);
15    public XobeNodeList getFollowingNodes(String test);
16    public XobeNodeList getPrecedingNodes(String test);
17    public XobeNodeList getAttributeNodes(String test);
18    public XobeNodeList getSelfNodes(String test);
19    public XobeNodeList getAncestorOrSelfNodes(String test)
        ;
20    public XobeNodeList getDescendantOrSelfNodes(String
        test);
21
22  } // XobeNodeList
```

Die Knotenliste `XobeNodeList` ist eine Erweiterung der DOM-Schnittstelle `NodeList`. Es werden zusätzlich Methoden zum Hinzufügen von Elementen deklariert (3-5). Mit der Methode `iterator` (7) ist es möglich, die Elemente einer Liste der Reihe nach durch eine Instanz der Klasse `Iterator` aus der Java-Standardbibliothek [Sun01a] zu bearbeiten. Die restlichen Methoden (9-20) dienen zur Selektion von Knoten bzgl. einer der in XPath definierten Achse. Sie werden für die Implementierung der XPath-Ausdrücke verwendet, wie sie durch die Transformation, die in diesem Abschnitt beschrieben wird, entsteht.

Weiterhin wird davon ausgegangen, dass eine Implementierung der Schnittstelle `XobeNodeList` zur Verfügung steht. Da sich die Operationen bis auf die Methoden zur Knotenselektion, auf die in diesem Abschnitt noch eingegangen wird, nicht von den Methoden der Schnittstelle `List` aus der Java-Bibliothek unterscheiden, wird hier von einer Angabe abgesehen.

Die Transformation von XPath-Ausdrücken wird durch folgende Vorschriften definiert.

Definition 5.11 (Transformation eines XPath-Ausdrucks)
Die *Transformation eines XPath-Ausdrucks* mit der XML-Objekt-Variablen V und den Lokalisierungsschritten S_i mit $i \in \{1, \ldots, n\}$ ist definiert durch:

$$[\![V/S_1/\ldots/S_n]\!]_{\mathtt{t}} \Rightarrow \begin{array}{l} \texttt{XobeNodeList t = new XobeNodeListImpl(}V\texttt{);} \\ [\![S_1]\!]^{\mathtt{t}} \\ \vdots \\ [\![S_n]\!]^{\mathtt{t}} \end{array}$$

Dabei referenziert die Variable `t` eine Knotenliste. □

Ein XPath-Ausdruck im XOBE-Programm wird in mehrere Anweisungen reines Java transformiert. Das Ergebnis eines XPath-Ausdrucks besteht aus einer Liste von XML-Objekten. Diese werden in einer Knotenliste `t` gespeichert, die zunächst aus dem aktuellen Knoten V besteht. Auf diese initiale Knotenliste werden anschließend die durch den XPath-Ausdruck angegebenen Lokalisierungsschritte bzgl. der Achsen und Knotentests durchgeführt. Die Anweisungen für diese Lokalisierungsschritte werden durch die Anwendung der entsprechenden Transformationen erzeugt. Die Knotenliste `t` wird abschließend als Resultatsannotation als Ergebnis zurückgeliefert.

Definition 5.12 (Transformation eines Schritts)
Die *Transformation eines Lokalisierungsschritts* mit dem Achsenbezeichner A, dem Knotentest N, den Prädikaten P_i mit $i \in \{1, \ldots, n\}$ und der Knotenliste t ist definiert durch:

$$[\![A\texttt{::}N\texttt{[}P_1\texttt{]}\ldots\texttt{[}P_n\texttt{]}]\!]^t \Rightarrow \begin{array}{l} t\texttt{.}\,[\![A\texttt{::}N]\!]^t\texttt{;} \\ [\![\texttt{[}P_1\texttt{]}]\!]^t \\ \vdots \\ [\![\texttt{[}P_n\texttt{]}]\!]^t \end{array}$$

□

Für einen Lokalisierungsschritt wird zunächst die Transformation für den Achsenbezeichner und den Knotentest durchgeführt. Diese operiert auf der mittels der Parameterannotation übergebenen Knotenliste t. Zum Schluss erfolgt die Transformation der Prädikate, die die Knotenliste weiter einschränken.

Definition 5.13 (Transformation eines Knotentests)
Die *Transformation eines Knotentests* bzgl. der folgenden Achsen für den Elementnamen E und der Knotenlistenvariablen t ist definiert durch:

- Selbst-Achse („self axis“)

$$[\![\texttt{self::}E]\!]^t \Rightarrow t \texttt{ = } t\texttt{.getSelfNodes("}E\texttt{")}$$

- Kind-Achse („child axis“)

$$[\![\texttt{child::}E]\!]^t \Rightarrow t \texttt{ = } t\texttt{.getChildNodes("}E\texttt{")}$$

- Nachfahr-Achse („descendant axis“)

$$[\![\texttt{descendant::}E]\!]^t \Rightarrow t \texttt{ = } t\texttt{.getDescendantNodes("}E\texttt{")}$$

- Nachfahr-oder-Selbst-Achse („descendant-or-self axis“)

$$[\![\texttt{descendant-or-self::}E]\!]^t \Rightarrow t \texttt{ = } t\texttt{.getDescendantOrSelfNodes("}E\texttt{")}$$

- Nachfolgende-Geschwister-Achse („following sibling axis“)

$$[\![\texttt{following\_sibling::}E]\!]^t \Rightarrow t \texttt{ = } t\texttt{.getFollowingSiblingNodes("}E\texttt{")}$$

- Nachfolger-Achse („following axis“)

$$[\![\texttt{following::}E]\!]^t \Rightarrow t \texttt{ = } t\texttt{.getFollowingNodes("}E\texttt{")}$$

- Attribut-Achse („attribute axis“)

$$[\![\texttt{attribute::}E]\!]^t \Rightarrow t \texttt{ = } t\texttt{.getAttributeNodes("}E\texttt{")}$$

- Eltern-Achse („parent axis“)

$$[\![\texttt{parent::}E]\!]^t \Rightarrow t \texttt{ = } t\texttt{.getParentNodes("}E\texttt{")}$$

- Vorfahr-Achse („ancestor axis“)

$$[\![\texttt{ancestor::}E]\!]^t \Rightarrow t \texttt{ = } t\texttt{.getAncestorNodes("}E\texttt{")}$$

- Vorfahr-oder-Selbst-Achse („ancestor-or-self axis“)

$$[\![\texttt{ancestor-or-self::}E]\!]^t \Rightarrow t \texttt{ = } t\texttt{.getAncestorOrSelfNodes("}E\texttt{")}$$

- Vorherige-Geschwister-Achse („preceding sibling axis“)

$$[\![\texttt{preceding\_sibling::}E]\!]^t \Rightarrow t \texttt{ = } t\texttt{.getPrecedingSiblingNodes("}E\texttt{")}$$

- Vorgänger-Achse („preceding axis“)

$$[\![\texttt{preceding::}E]\!]^t \Rightarrow t \texttt{ = } t\texttt{.getPrecedingNodes("}E\texttt{")}$$

□

Die Implementierung eines Knotentests bzgl. einer Achse erfolgt durch einen einfachen Methodenaufruf für die Achse und den Knotentest. Für jede Achse existiert dafür eine eigene Methode, die die entsprechenden Knoten extrahiert. Dabei wird von der Methode die der Achse zugeordnete Dokumentordnung berücksichtigt (siehe dazu Abschnitt 2.2). Der nachfolgende Knotentest erfolgt durch den Aufruf der Methode `nodeTest` innerhalb der Achsenmethode und schränkt die Knotenliste der Achse auf die im XPath-Ausdruck angegebenen Elemente ein.

Wie in der Transformation eines Knotentests bzgl. einer Achse ersichtlich ist, wird für jede Achse eine eigene Methode aufgerufen. Diese Methoden selektieren die jeweiligen Knoten der angegebenen Achse. In diesem Abschnitt wird beispielhaft die Implementierung der Methode `getChildNodes` für die Kind-Achse vorgestellt. Da sich die Routinen für die anderen Achsen auf ähnliche Weise umsetzen lassen, wird an dieser Stelle auf deren Angabe verzichtet. Eine vollständige Aufführung findet sich im Anhang D.

```
1   public XobeNodeList getChildNodes(String test) {
2     int i;
3     XobeNodeList result;
4
5     result = new XobeNodeListImpl();
6     for (i = 0; i < this.getLength(); i = i + 1)
7       result.addAll(getChildNodes(test, this.item(i)));
8     return result;
9   } // getChildNodes
```

```
10    private static XobeNodeList getChildNodes(String test,
        Node node) {
11      int i;
12      NodeList children;
13      XobeNodeList result;
14
15      result = new XobeNodeListImpl();
16      children = node.getChildNodes();
17      for (i = 0; i < children.getLength(); i = i + 1)
18        if (nodeTest(test,children.item(i)))
19          result.add(children.item(i));
20      return result;
21    } // getChildNodes
```

Die einstellige Methode `getChildNodes` (1-9) berechnet für die aufgerufene Knotenliste die Kinder der einzelnen Knoten in der Liste. Dies geschieht durch den Aufruf der zweistelligen Methode `getChildNodes` (7). Die Resultate der einzelnen Aufrufe werden in der Knotenliste `result` (3) abgelegt und am Ende der Methode zurückgegeben (8). In der zweistelligen Methode `getChildNodes` (10-21) werden die Kinder für den als Parameter übergebenen Knoten `node` dem Knotentest `nodeTest` (18) unterzogen. Nur die Knoten, die diesen bestehen, werden in die Ergebnisknotenliste `result` (13) aufgenommen (19). Die Kinder eines Knotens erhält die Routine durch die DOM-Methode `getChildNodes` (16).

Die aufgerufene Methode `nodeTest` überprüft, ob es sich bei einem Knoten um eine bestimmte Knotenvariante handelt.

```
1     private static boolean nodeTest(String test, Node item)
        {
2       if (test.equals("text()"))
3         test = "#text";
4       else if (test.equals("comment()"))
5         test = "#comment";
6       else if (test.equals("node()"))
7         test = "*";
8       if (test.equals("*") || item.getNodeName().equals(
          test))
9         return true;
10      else
11        return false;
12    } // nodeTest
```

Der übergebene Knoten `item` soll der Variante `test` entsprechen. Nur dann wird von der Routine `true` zurückgeliefert (9). Um welche Form es sich bei einem Knoten handelt, kann mittels der DOM-Methode `getNodeName` (8) analysiert werden. Der Platzhalter `*` erfüllt für jeden

Knoten in XPath den Knotentest.

Das folgende Beispiel eines einfachen XPath-Ausdrucks zeigt die Anwendungen der definierten Transformationsvorschriften.

Beispiel 5.4
Das Beispiel selektiert mit Hilfe eines XPath-Ausdrucks aus dem XML-Objekt `off`, das ein `offer`-Element repräsentiert, die `book`-Elemente. Die Liste wird der Variablen `ts` zugewiesen.

```
1 xml<book*> ts = off/child::book;
```

Für die Implementierung des XPath-Ausdrucks ergeben sich nach der Ableitung gemäß der definierten Transformationsregeln folgende Java-Anweisungen.

```
1 XobeNodeList t = new XobeNodeListImpl(off);
2 t = t.getChildNodes("book");
3 XobeNodeList ts = t;
```

Mit der Variablen `off` wird eine temporäre Knotenliste `t` erzeugt (1). Für diese wird die Methode `getChildNodes` für die Kind-Achse aufgerufen (2), die auch den Knotentest auf den Elementnamen `book` durchführt. Als letztes (3) wird die temporäre Liste `t` der im Beispiel angegebenen Listenvariable `ts` zugewiesen. □

Die Transformation von Prädikaten ist bis auf zwei Besonderheiten nicht weiter schwierig. Das Prädikat selbst wird als Schleife realisiert, in der der Prädikatausdruck für jedes Element in der Knotenliste ausgewertet wird. Ist der Ausdruck für einen Knoten ungültig, wird dieser aus der Liste herausgefiltert.

Der Prädikatausdruck muss nur unwesentlich transformiert werden. Nur an Stellen, wo erneut XPath-Ausdrücke verschachtelt auftreten, muss eine Veränderung vorgenommen werden. Weil der Ausdruck ansonsten unverändert bleibt, wird in dieser Arbeit nur eine Transformationsvorschrift für die Vergleichsrelation angegeben. Alle weiteren Relationen und Operationen der Ausdrücke lassen sich auf ähnliche Weise ganz analog umsetzen, weshalb auf diese Transformationsvorschriften an dieser Stelle verzichtet wird.

Die Parameter, die die Transformation benötigt, werden bis zu den möglichen XPath-Ausdrücken innerhalb eines Prädikats oder den elementaren XPath-Operationen rekursiv durchgereicht. Nur dort können zusätzliche Java-Anweisungen erzeugt werden, die der eigentlichen Auswertung des Ausdrucks vorangestellt werden müssen. Der eigentliche Ausdruck bleibt ansonsten praktisch unverändert und wird dann als Resultat zurückgeliefert.

Die erste Besonderheit muss für XPath-Ausdrücke innerhalb von Prädikaten berücksichtigt werden, mit denen Bedingungen über die zu selektierenden Knoten formuliert werden können. Da sich solch ein XPath-Ausdruck während der Selektion auf den jeweils aktuellen Knoten bezieht, muss im Vergleich zu allgemeinen XPath-Ausdrücken eine modifizierte Transformation angewendet werden. Die zweite Sonderbehandlung erfahren die zwei elementaren XPath-Operationen, die innerhalb von XPath-Ausdrücken auftreten können und sich auf die selektierte Knoten-

liste beziehen.

Die Transformation von Prädikaten ist durch nachfolgende Vorschriften definiert.

Definition 5.14 (Transformation eines Prädikats)
Die *Transformation eines Prädikats* P mit der Knotenliste t ist definiert durch:

$$[\![[P]]\!]^{t} \Rightarrow \begin{array}{l} \texttt{int p = 1; int l = } t\texttt{.getLength();} \\ \texttt{Iterator it = } t\texttt{.iterator();} \\ \texttt{while (it.hasNext()\{} \\ \quad \texttt{Node n = it.next();} \\ \quad [\![P]\!]_{e}^{(\texttt{p,l,n})} \\ \quad \texttt{if (!}e\texttt{) it.remove();} \\ \quad \texttt{p = p + 1;} \\ \texttt{\} // while} \end{array}$$

□

Die Einschränkung der Knotenliste durch ein Prädikat kann erst nach dem letzten Knotentest erfolgen. Die Implementierung erfolgt über eine Schleife, die über die Knoten in der Knotenliste t iteriert. Innerhalb der Schleife wird für jeden Knoten die Transformation des Prädikatausdrucks P ausgeführt. Das Ergebnis wird dann mit dem booleschen Ausdruck e ermittelt. Ist das Prädikat nicht erfüllt, wird der behandelte Knoten aus der Knotenliste mit der Methode `remove` gelöscht.

Definition 5.15 (Transformation einer Vergleichsrelation)
Die *Transformation der Vergleichsrelation* für die beiden Ausdrücke E_l und E_r mit der aktuellen Position p in der Knotenliste, der letzten Position l in der Knotenliste und der Java-Variablen für den aktuellen Knoten n ist definiert durch:

$$[\![E_l \texttt{ == } E_r]\!]^{(p,l,n)}_{(e_l \texttt{ == } e_r)} \Rightarrow \begin{array}{l} [\![E_l]\!]^{(p,l,n)}_{e_l} \\ [\![E_r]\!]^{(p,l,n)}_{e_r} \end{array}$$

□

Die Transformation wird rekursiv auf den linken und rechten Ausdruck der Vergleichsrelation angewendet. Als Resultat der Transformation wird der Vergleich zwischen dem Ergebnis e_l der linken und dem Ergebnis der rechten Transformation e_r als Java-Ausdruck abgeliefert.

Definition 5.16 (Transformation eines XPath-Ausdrucks innerhalb eines Prädikats)
Die *Transformation eines XPath-Ausdrucks innerhalb eines Prädikats* mit den Lokalisierungsschritten S_i und $i \in \{1, \ldots, k\}$, der aktuellen Position p in der Knotenliste, der letzten Position l in der Knotenliste und der Java-Variablen für den aktuellen Knoten n ist definiert durch:

$$[\![S_1\texttt{/.../}S_k]\!]^{(p,l,n)}_{\texttt{t}} \Rightarrow \begin{array}{l} \texttt{XobeNodeList t = new XobeNodeListImpl(}n\texttt{);} \\ [\![S_1]\!]^{\texttt{t}} \\ \vdots \\ [\![S_k]\!]^{\texttt{t}} \end{array}$$

Dabei verweist die Variable `t` auf eine Knotenliste. □

Die Transformation unterscheidet sich von der Transformation der allgemeinen XPath-Ausdrücke dahingehend, dass der aktuelle Knoten n, ein Parameter der Transformation, initial die Knotenliste `t` bildet. Die rekursive Transformation der einzelnen Schritte des Ausdrucks und die Knotenliste `t` als Resultat der Transformation gleichen der Definition für allgemeine XPath-Ausdrücke.

Definition 5.17 (Transformation elementarer XPath-Operationen)
Die *Transformation der elementaren XPath-Operationen innerhalb eines Prädikats* mit der aktuellen Position p in der Knotenliste, der letzten Position l in der Knotenliste und der Java-Variablen für den aktuellen Knoten n ist definiert durch:

$$[\![position()]\!]_p^{(p,l,n)} \Rightarrow \epsilon$$

$$[\![last()]\!]_l^{(p,l,n)} \Rightarrow \epsilon$$

□

Für die elementaren Operationen innerhalb von Prädikaten werden keine zusätzlichen Java-Anweisungen generiert. Es werden lediglich Variablen als Java-Ausdruck als resultierende Annotationen zurückgeliefert, für die Operation `position` die aktuelle Position p und für die Operation `last` die letzte Position in der Knotenliste l.

Im restlichen Abschnitt werden zwei XPath-Beispiele vorgestellt und deren durch Transformation erzeugte Implementierung beschrieben.

Beispiel 5.5
Für dieses Beispiel wird erneut davon ausgegangen, dass die XML-Objekt-Variable `off` als `offer`-Element deklariert wurde. Es werden sämtliche `book`-Elemente, die nach dem fünften Buch im Inhalt des `offer`-Elements auftreten, selektiert und der Knotenliste `ts` zugewiesen.

```
1 xml<book*> ts = off/child::book[position() > 5];
```

Die Transformationsvorschriften generieren für diesen XPath-Ausdruck die folgende Implementierung:

```
1  XobeNodeList t = new XobeNodeListImpl(off);
2  t = t.getChildNodes("book");
3  int pos = 1;
4  int last = t.getLength();
5  Iterator it = t.iterator();
6  while (it.hasNext()) {
7    Node cnode = it.next();
8    if (!(pos > 5))
9      it.remove();
10   pos = pos + 1;
11 } // while
12 XobeNodeList ts = t;
```

Es wird eine temporäre Knotenliste `t` mit dem Element `off` initialisiert (1). Anschließend werden die Kinderelemente mit Namen `book` selektiert (2). Danach erfolgt die Initialisierung einer Variablen `pos` (3), die die Position des aktuellen Knotens im XPath-Ausdruck angibt, und einer Variablen `last` (3), die die Position des letzten Elements und damit die Länge der Knotenliste vor der Selektion durch das Prädikat angibt. Mit Hilfe einer Schleife (6-11) wird über die Knoten in der Liste iteriert. Der aktuelle Knoten wird der Variablen `cnode` zugewiesen. Für jeden Knoten wird der Prädikatausdruck ausgewertet (8). Ist dieser nicht erfüllt, wird der Knoten aus der Liste entfernt (9). Am Ende (12) wird die temporäre Liste `t` der im XOBE-Programm deklarierten Liste `ts` zugewiesen. □

Das nächste Beispiel verwendet innerhalb des Prädikats erneut einen XPath-Ausdruck.

Beispiel 5.6
Dieses Beispiel selektiert aus dem Inhalt eines `book`-Elements die `price`-Elementkinder. Die XML-Objekt-Variable `b` steht also für ein Element mit Elementnamen `book`. Die Elemente mit Namen `price` müssen ein Attribut `currency` haben, das mit dem Wert `EUR` belegt ist.

```
1 xml<title*> ts = b/child::price[string(attribute::
    currency) == "EUR"];
```

Für diesen Ausdruck wird eine ähnliche Implementierung wie im letzten Beispiel erzeugt.

```
1  XobeNodeList t = new XobeNodeListImpl(b);
2  t = t.getChildNodes("price");
3  int pos = 1;
4  int last = t.getLength();
5  Iterator it = t.iterator();
6  while (it.hasNext()) {
7    Node cnode = it.next();
8    XobeNodeList t1 = new XobeNodeListImpl(cnode);
9    t1 = t1.getAttributeNodes("currency");
10   if (!(string(t1) == "EUR"))
11     it.remove();
12   pos = pos + 1;
13 } // while
14 XobeNodeList ts = t;
```

Wie zu sehen ist, unterscheidet sich dieses Beispiel vom vorhergehenden hauptsächlich in der Umsetzung des Prädikatsausdrucks. Da in diesem ein XPath-Ausdruck auftritt werden für diesen zusätzliche Java-Anweisungen generiert (8-9), die der Auswertung des eigentlichen Prädikatsausdrucks (10) vorangestellt werden müssen. Im Prädikatsausdruck wird der XPath-Ausdruck durch die temporäre Knotenliste `t1` ersetzt. Die im Ausdruck verwendete Methode `string` ist eine Methode aus der XPath-Bibliothek, von der angenommen wird, dass sie für die XOBE-Implementierung zur Verfügung steht. □

5.4 Erfahrungen und Leistungsdaten

Die wichtigsten Erfahrungen mit der prototypischen Implementierung des XOBE-Präprozessors gibt dieser Abschnitt wieder. Neben der Präsentation von Leistungsmessungen werden im Anschluss weitere Entwicklungsmöglichkeiten vorgestellt.

5.4.1 Leistungsdaten

Für die Bewertung der Leistung der prototypischen Implementierung sind zwei Aspekte von Interesse. Zu beachten ist zunächst, wieviel Rechenzeit der Präprozessor für die Übersetzung der XOBE-Programme benötigt und welcher Anteil davon auf den Algorithmus zur Typüberprüfung entfällt. Von Belang sind zum Zweiten die Ausführungszeiten der resultierenden Servlets, die nach der Transformation auf dem DOM basieren. Verglichen werden diese Testprogramme mit Standard-Servlet-Implementierungen, die ohne das DOM realisiert wurden.

Zu rechnen ist bei dieser Gegenüberstellung mit einer leichten Verschlechterung der Ausführungszeiten für die resultierenden XOBE-Programme, da in der DOM-Implementierung komplexere Objekt-Strukturen aufgebaut werden als bei einer Verarbeitung auf der Grundlage von Zeichenketten. Bei einem Vergleich mit reinen Java-Programmen, die ebenfalls eine DOM-Implementierung nutzen, wären für die XOBE-Programme allerdings keine Laufzeitverluste zu erwarten, weil es sich bei XOBE um ein statisches Typsystem handelt. Die deklarierten Bedingungen der Sprachbeschreibungen dienen lediglich zur Typüberprüfung während der Übersetzung und sind bis auf wenige Ausnahmen, die nur dynamisch überprüfbar sind, nicht zur Laufzeit des Programms erforderlich. Die Messungen wurden durchgeführt auf einer SUN Blade 1000 mit zwei Ultra Sparc 3 (600 Mhz) Prozessoren und 1 GByte Hauptspeicher unter dem Betriebssystem Solaris 8 (SunOS 5.8).

Die folgenden Testprogramme wurden in die Leistungsmessung einbezogen:

Estate ist kurzes Programm, welches Web-Seiten in XHTML für den Web-Server eines Immobilienmaklers generiert. Es geht davon aus, dass Immobilienbeschreibungen in Form von XML-Dateien vorliegen, die einer kleinen, nicht standardisierten Sprachbeschreibung folgen. Die Anwendung liest diese Dokumente als Eingabe ein und konvertiert die Daten in eigenständige XHTML-Dokumente zur Präsentation im Web.

MobileArchive ist eine Web-Anwendung auf der Basis von Servlets, die eine Anbindung für mobile Geräte über die *Wireless-Markup-Language* (WML) für ein medizinisches Medienarchiv realisiert. Sie ermöglicht neben der Navigation durch die Ablagestruktur des Archivs, die einem Verzeichnisbaum des Dateisystems ähnelt, die Suche nach bestimmten Medienobjekten. Medienobjekte einiger Formate können, soweit es das Wiedergabegerät zulässt, auch angezeigt werden. Detailliertere Ausführungen zu MobileArchive finden sich im nächsten Kapitel.

Übungsdatenverwaltung (ÜDV) ist eine Web-Anwendung, die ebenfalls auf Servlets basiert, und die Benutzerschnittstelle für ein akademisches Übungsdatenverwaltungssystem realisiert. Die Anwendung ermöglicht dem Benutzer einerseits Eingaben, zum Beispiel von Klausurergebnissen, die an das Datenbanksystem im Hintergrund weitergereicht werden, und andererseits auch Einblick in den aktuellen Datenbestand. Eine genauere Beschreibung zum Funktionsumfang liefert das folgende Kapitel.

Die Implementierung des ersten Testprogramms besteht aus mehreren, einfachen, iterativen Methoden, die eine einfache Traversierung der Eingabe mit gleichzeitiger Berechnung der Ausgabe durchführen. Die zweite Anwendung ermöglicht, wie beschrieben, eine WML-Verbindung zu einem Medienarchiv. Auf das Medienarchiv wird über eine von diesem zur Verfügung gestellte Programmschnittstelle zugegriffen. Die Anwendung selbst speichert die Position des aktuellen Benutzers (Clients) in der Struktur des Archivs. Die dritte Testanwendung kommuniziert über JDBC mit dem Datenbankmanagementsystem des Herstellers Informix [Inf97b]. Die Eingabe des Benutzers wird in Datenbankanfragen umgewandelt und an die Datenbank weitergeleitet. Etwaige Antworten oder Resultate der Anfragen werden in XHTML eingebettet und an den Benutzer zurückgeschickt. In den Programmen Estate und ÜDV wird die Sprachbeschreibung von XHTML (genauer XHTML-transitional) importiert, während die Anwendung MobileArchive die Sprachbeschreibung WML verwendet.

Anmerkung: Für die beiden Anwendungen MobileArchive und ÜDV existierten bereits Implementierungen in Standard-Java-Servlet-Technik, die im Rahmen dieser Arbeit für eine Umsetzung mit XOBE verwertet wurden. Bei dieser Umsetzung wurden viele einfache, unentdeckte Fehler in den alten Implementierungen gefunden, die trotz sorgfältiger Testläufe der Implementierungen übersehen wurden. Die häufigsten Fehler waren ungültig generierte XML-Fragmente.

	Zeilenanzahl		Übersetzungszeit (s)		Ausführungszeit (s)	
Anwendung	XOBE	Schema	gesamt	Typanalyse	Standard	XOBE
Estate	158	1231	2.16	0.05	-	0.9
MobileArchive	1045	355	4.49	0.16	0.03	0.04
ÜDV	195	1196	4.61	0.07	0.01	0.01

Die Tabelle präsentiert die Anzahl der Zeilen der gesamten XOBE-Programme und die Anzahl der Zeilen der deklarierten Sprachbeschreibungen in den ersten beiden Spalten. In der zweiten Gruppe von Spalten zeigt die Tabelle die Zeit, die für die Vorübersetzung der XOBE-Programm verbraucht wurde. Sie beinhaltet die Zeit des Einlesens, der Typinferenz, der Anwendung des Subtyp-Algorithmus und der Quelltext-Transformation in reines Java, wie sie in den letzten Abschnitten beschrieben wurde. Die Zeit, die der Subtyp-Algorithmus benötigt, wird in der Spalte „Typanalyse“ dargestellt. Die dritte Spaltengruppe der Tabelle vermittelt einen Eindruck davon, wie die Leistungsfähigkeit der Servlets von der DOM-basierten Implementierung beeinflusst wird. Die Spalte „Standard“ zeigt zum Vergleich die Zeit, die während der Ausführung der nicht mit XOBE implementierten Servlets vergeht. Die letzte Spalte gibt schließlich die Laufzeit der XOBE-Programme an.

Wie die Tabelle zeigt, arbeitet der Algorithmus zur Typüberprüfung für die vorgestellten Anwendungen mit akzeptabler Geschwindigkeit. Selbst Anwendungen, die relativ große XML-Typen aus der Sprachbeschreibung von XHTML oder WML nutzen, werden in vielversprechender Zeit übersetzt. Die Ausführungszeiten der DOM-basierten Servlet-Implementierungen ist, wie erwartet, etwas langsamer als die der Standard-Servlets, liegen aber noch in einem erträglichen Rahmen. Der Geschwindigkeitsvorteil der Standard-Servlet-Implementierungen ist mit der fehlenden Garantie für korrekt generierte XML-Dokumente zu erklären. Aus diesem Grund ist der dargestellte Vergleich den XOBE-Programmen gegenüber nicht besonders fair. Gerechter ist eine Gegenüberstellung der XOBE-Implementierungen mit DOM-basierten Realisierungen, die dynamisch die Gültigkeit überprüfen. Für diesen Vergleich ist mit einem Vorteil für die XOBE-Programme zu rechnen.

5.4.2 Erweiterungen des Prototyps

Zur Erweiterung des aktuellen Prototyps gibt es vielfältige Möglichkeiten. So sind Verbesserungen bei der Implementierung der XML-Objekte als auch der XPath-Ausdrücke denkbar. Die Integration in einen eigenständigen Java-Übersetzer wäre ebenfalls vorstellbar.

Für die Implementierung von XOBE das DOM zu nutzen ist eine naheliegende Vorgehensweise. Trotzdem sind aber auch andere Implementierungen realisierbar. Besonders sinnvoll erscheint eine Umsetzung, die einen weitergehenden Zugriff auf den Inhalt eines Elements ermöglicht, der sich stärker an der Struktur des Inhaltsmodells orientiert. Die Idee hinter diesem strukturorientierten Zugriff ist, dass der Benutzer den Inhalt eines Elements über die in der Sprachbeschreibung definierte Struktur anspricht. Diese kann, wie in Abschnitt 2.1 beschrieben, aus ineinander verschachtelten Konkatenationen, reguläre Vereinigungen usw. bestehen. Im DOM ist ein solcher Zugriff nicht vorgesehen, da dort der Inhalt eines Elements zu einer Knotenliste von Kindern verschmolzen wird. Trotzdem ist mit dem DOM, falls die verwendete Sprachbeschreibung dem Benutzer bekannt ist, eine gleichwertige Selektion des Elementinhalts möglich, die aber nicht ohne zusätzliche, kostspielige Berechnungen auskommt. Dieser zusätzliche algorithmische Aufwand ist notwendig, weil die Struktur der Inhaltsmodelle nicht in der DOM-Implementierung gespeichert wird. Würde eine XOBE-Implementierung die Struktur des Elementinhalts ebenfalls repräsentieren, könnten bestimmte Zugriffe auf den Inhalt in konstanter Zeit erfolgen. Dies hätte eine Beschleunigung des Zugriffes um eine Größenordnung zur Folge, da die Selektion eines Kindknotens im DOM linear zur Anzahl der Kinder eines Elements ist.

Die Implementierung der Auswertung der XPath-Ausdrücke ließe sich zunächst ganz naiv dadurch verbessern, dass eine Auswertung der Prädikate bereits während der Auswertung der Achsen erfolgt. Diese Optimierung ist allerdings nicht immer möglich, da sich Prädikate auch auf Eigenschaften der resultierenden Knotenliste einer Achse beziehen können. Um eine echte Verbesserung bei der Auswertung von XPath zu erzielen, sollte besser auf die Implementierung von [GKP02] zurückgegriffen werden. Dort wird ein Algorithmus angegeben, der sämtliche XPath-Ausdrücke mit polynomiellem statt exponentiellem Aufwand berechnet. Für einge-

schränkte XPath-Anfrage ist sogar eine lineare Auswertung möglich, was eine Verbesserung um Größenordnungen darstellt.

Interessant wäre auch die Integration von XOBE in einen Java-Übersetzer. Dadurch könnten die zusätzlichen Typinformationen eines XOBE-Programms für weitergehende Programmoptimierungen eingesetzt werden. Beispielsweise wäre die Repräsentation konstanter XML-Fragmente in optimierter Form möglich. Auch könnten Attribut- und Elementnamen einmalig in den betreffenden XML-Objektklassen gespeichert werden, statt diese Information redundant in jedem XML-Objekt vorzuhalten.

Kapitel 6

Web-Anwendungen programmiert in XOBE

Um die in den vorherigen Kapiteln erarbeiteten Sprachkonstrukte von XOBE und deren Transformation zu nutzen, wurden im Rahmen dieser Arbeit mehrere prototypische Implementierungen von Web-Anwendungen vorgenommen. Zwei dieser Anwendungen werden in diesem Kapitel beschrieben. Damit wird gezeigt, dass XOBE für die Realisierung von Web-Anwendung nutzbar und sinnvoll ist.

Die erste Anwendung implementiert einen Zugang zu einem Medienarchiv für mobile Geräte, die XML in der Form der Wireless-Markup-Language interpretieren können. Als zweite Web-Anwendung wird ein Informationssystems vorgestellt, das den Übungsbetrieb von Lehrveranstaltungen an Universitäten verwaltet. Es ist über eine XHTML-Schnittstelle dem Benutzer zugänglich. Die Datenhaltung dieser Anwendung erfolgt in einer relationalen Datenbank. Beide Implementierungen nutzen die Sprachkonstrukte von XOBE intensiv.

Der erste Abschnitt dieses Kapitels stellt den Zugang zu einem Medienarchiv über die Wireless-Markup-Language vor. Im folgenden Abschnitt erfolgt die Darstellung des Informationssystems für den Übungsbetriebs einer Universität. Beide Abschnitte sind so gegliedert, dass zunächst die Arbeitsweise der Anwendungen vorgestellt werden und anschließend auf Architektur und Details der Implementierung eingegangen wird.

6.1 WML-Anbindung eines Medienarchivs

Dieser Abschnitt beschreibt die Anwendung *MobileArchive*, einen Zugang zu einem Medienarchiv für mobile Endgeräte, realisiert in der Java-Erweiterung XOBE. Ein Vorgänger der Anwendung auf der Basis von Java-Servlets ist im Rahmen einer Studienarbeit [Kra01] am Institut für Informationssysteme entstanden. Für die vorliegende Arbeit wurde diese Implementierung

mittels XOBE neu umgesetzt.

Die WML-Anbindung erfolgt für das Medienarchiv der Universität zu Lübeck MONTANA [BL00], das am Institut für Informationssysteme entwickelt wurde. Ein Medienarchiv dient der zentralen Archivierung und Verwaltung von Mediendaten. Ein Laden und Speichern dieser Daten wird von entfernten Rechnern im Netzwerk ermöglicht. Die Mediendaten oder -objekte sind digitale oder digitalisierte Daten der verschiedensten Formate, wie z. B. Bilder, Filme oder Audiosequenzen, die als Dateien abgelegt werden. Die Dateien im Archiv werden ähnlich einem Dateisystem in Verzeichnissen hierarchisch angeordnet. Verzeichnisse werden wie Medienobjekte als Archivobjekte aufgefasst. Der Zugriff auf die Daten wird über eine Benutzerrechteverwaltung kontrolliert.

Die Benutzerschnittstelle des Medienarchivs ist als Web-Schnittstelle konzipiert. Die Kommunikation erfolgt über einen Browser, der die Navigation durch die Verzeichnisstruktur ermöglicht. Medienobjekte können zum Archiv hinzugefügt und aus dem Archiv heruntergeladen werden. Die Schnittstelle ermöglicht die Manipulation von Objekteigenschaften der Medienobjekte, die Suche nach Mediendaten anhand der Eigenschaften und in bestimmten Fällen eine Präsentation der Mediendaten selbst über den Browser.

Für das Anzeigen von Informationen auf mobilen Endgeräten, wie Handys oder PDAs, ist in XML die Auszeichnungssprache Wireless-Markup-Language (WML) spezifiziert worden. Die definierten Elemente erinnern stark an HTML. Beispielsweise werden Elemente zur Textformatierung, für Hyperlinks, Bilder und Eingabefelder zur Verfügung gestellt. Der wichtigste Unterschied ist, dass ein WML-Dokument, das sogenannte „*deck*“, nicht nur aus einer, sondern aus mehreren anzeigbaren Seiten besteht, den „*cards*“. WML ist ein Standard des WAP-Forums, das sich ebenfalls mit dem darunter liegenden Protokoll Wireless-Application-Protocol (WAP) beschäftigt. Die Spezifikationen zu WAP und WML finden sich in [Wir00], einführende Beschreibungen zum Thema können in [KT01] nachgelesen werden.

6.1.1 Arbeitsweise und Benutzerzugang

Die Anwendung verwirklicht den Zugriff auf das Medienarchiv über ein mobiles Endgerät. Ein typischer Sitzungsablauf lässt sich wie folgt beschreiben. Der Anwender, der mit dem Medienarchiv Kontakt aufnehmen will, wählt in seinem WML-fähigen Gerät die Seite des Medienarchivs an. Die Aufforderung zur Eingabe der Zugangsdaten beantwortet er mit seinem Benutzernamen und dem zugehörigen Kennwort. Der Anwender ist nun im Archiv angemeldet. Ihm stehen vier verschiedene Operationen zur Verfügung:

1. Navigation durch die Verzeichnishierarchie des Medienarchivs, zum gezielten Auffinden von Medienobjekten,
2. Anzeigen der Eigenschaften von ausgewählten Medienobjekten,
3. Anzeigen von Medienobjekten selbst für einige wenige Formate und

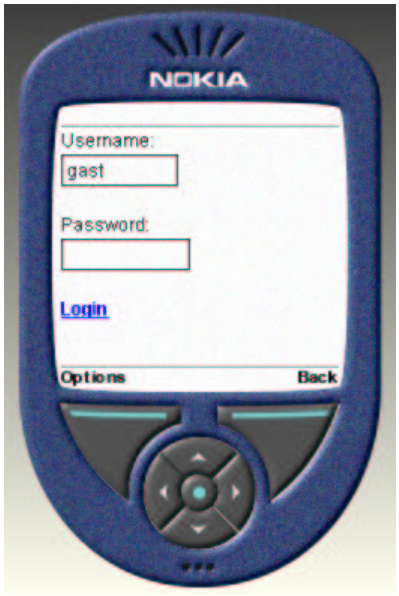

Abbildung 6.1: Loginseite

4. eine allgemeine Suchanfrage nach Medienobjekten.

Diese Operationen können beliebig oft wiederholt und dabei wahllos kombiniert werden. Meldet der Benutzer sich zum Schluss vom Medienarchiv ab, ist die Sitzung damit beendet.

Die Umsetzung der Anbindung zum Medienarchiv in WML ist eine durchaus anspruchsvolle Aufgabe. Schwierig erscheint vor allem die Darstellung, bei der eine Reduktion auf das nötigste erfolgen muss. So ist es in WML nicht sinnvoll, grafische Elemente, wie in HTML üblich, in die Benutzerführung einfließen zu lassen, da die Anzeigen sehr klein sind. Die Menüführung ist, soweit möglich, komprimiert und wird mittels Auswahllisten und verzweigenden Hyperlinks realisiert. Nach der Anmeldung im Archiv mit Benutzernamen und Kennwort, bietet sich dem Anwender die Möglichkeit in den Verzeichnissen mit Medienobjekten zu navigieren. Eine Auswahlliste präsentiert dabei die Unterverzeichnisse des aktuellen Verzeichnisses, eine weitere gibt die Namen der Medienobjekte aus. Über den Hyperlink eines Unterverzeichniseintrags ist der Inhalt dieses Unterverzeichnisses zu erreichen, der auf einer gesonderten Seite dargestellt wird.

Ein zentraler Aspekt der Anwendung ist die Möglichkeit zur Formulierung von Suchanfragen an das Medienarchiv. Diese Anfragen können sich auf die Eigenschaften der Medienobjekte beziehen, die beim Einstellen der Objekte in das Archiv durch beschreibende Attribute festgelegt wurden. Die Ergebnisse werden in Listen mit maximal zehn Einträgen aufbereitet. Damit wird eine einigermaßen übersichtliche und sinnvolle Aufteilung des Resultats gewährleistet.

Die Anzeigemöglichkeiten der WML-Browser sind sehr beschränkt, weshalb die meisten Medienobjekte nicht angezeigt werden können. Möglich ist eine Darstellung von Texten der Formate WML und ASCII sowie von Bildern im Format Wireless-BMP (WBMP). Bilder der Formate

Abbildung 6.2: Anzeige der Unterverzeichnisse

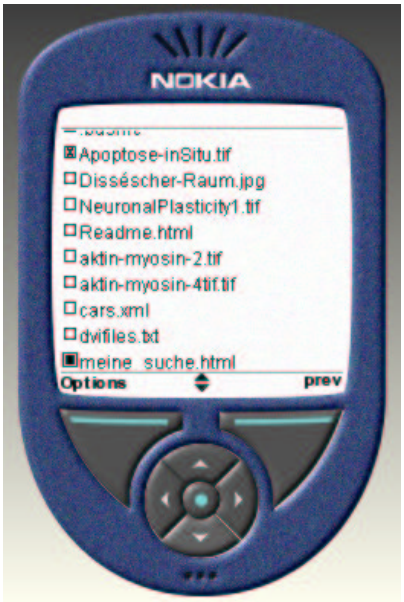

Abbildung 6.3: Anzeige der Medienobjekte

BMP, GIF, JPEG und TIFF können ebenfalls angezeigt werden, wobei der Inhalt in das Format WBMP konvertiert wird, weshalb mit starken Einschränkungen in der Darstellung gerechnet werden muss. Bei allen anderen Medienobjekten werden die Objekteigenschaften statt des Inhalts angezeigt. Die Eigenschaften von Medienobjekten werden in tabellarischer Form angezeigt. Dazu zählen das Format des Objekts, der Name des Autors und das Datum der Erstellung, um nur einige zu nennen.

6.1.2 Architektur und Implementierungsdetails

Die Architektur der Anwendung gliedert sich in drei mehr oder weniger unabhängige Schichten. Eine Schicht steht für das Datenmodell zur Repräsentation der beteiligten Daten, eine zweite steuert die Ein- und Ausgabe durch einen Automaten und die Präsentationsschicht erzeugt die Ausgabe in der Auszeichnungssprache WML und überträgt Eingaben in Ereignisse des Automaten. In diesem Abschnitt werden die drei Schichten in der Reihenfolge Datenmodell, Automat und Präsentation kurz vorgestellt.

Das Datenmodell der Anwendung ist sehr einfach und in Abbildung 6.8 als Klassendiagramm dargestellt. Die Klasse *WMLSession* repräsentiert die Sitzung von Anwendern zum Medienarchiv über die WML-Anbindung. Jedes Objekt dieser Klasse besteht aus einem aktuellen Archivobjekt der Klasse *ArchivObject* und einer optionalen Anfrage der Klasse *Query*.

Die Klasse *ArchivObject* ist eine abstrakte Klasse von der die Klassen *Directory* und *MediaObject* abgeleitet sind. Mit *Directory* werden Verzeichnisse im Medienarchiv repräsentiert, Medien-

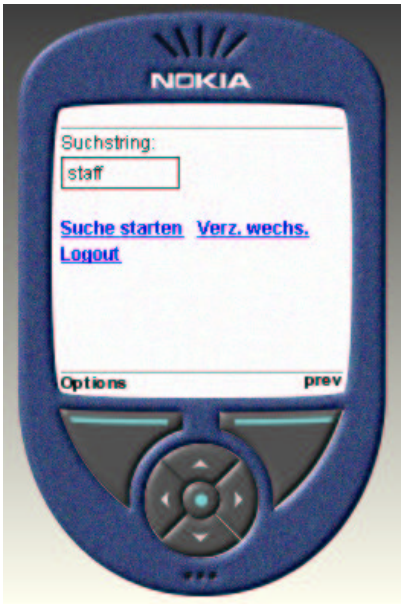

Abbildung 6.4: Eingabe der Anfrage

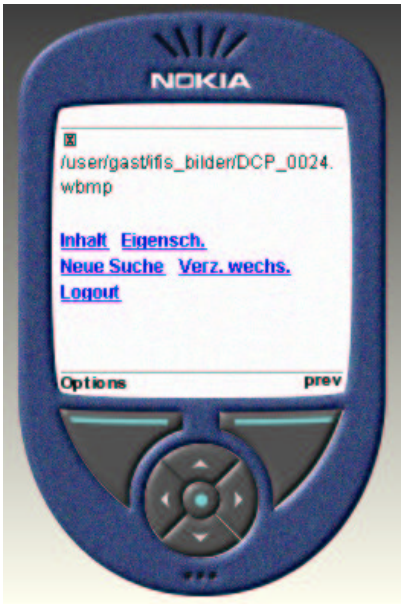

Abbildung 6.5: Anzeige des Suchergebnisses

objekte des Archivs durch Instanzen der Klasse *MediaObject*. Von der Klasse *MediaObject* existiert noch eine spezialisierte Klasse *DisplayableMedia* für die Medienobjekte, die auf einem mobilen Endgerät dargestellt werden können.

Anfragen an das Medienarchiv werden durch Objekte der Klasse *Query* modelliert. Sie bestehen aus einem Anfragetext *query*, nach dem im Archiv gesucht wird. Die Ergebnisse einer Anfrage werden mit dem Attribut *result* ebenfalls im *Query*-Objekt abgelegt.

Das Ein- und Ausgabeverhalten wird durch den Automaten, der in Abbildung 6.9 dargestellt ist, beschrieben. Wie zu sehen ist, besteht der Automat auf der obersten Ebene aus den vier Zuständen *Login request*, *Display archiv object*, *Display query* und *Good bye message*. Der Automat startet im Zustand *Login request*, von dem aus in den Zustand *Display archiv object* verzweigt wird. Von diesem sind sowohl die Zustände *Display query* als auch *Good bye message* erreichbar. Der letzte Zustand beendet den Ablauf des Automaten. Sämtliche Ereignisse, die zu Zustandsübergängen führen, werden durch den Anwender ausgelöst, indem dieser Benutzereingaben vornimmt.

Die beiden Zustände *Display archiv object* und *Display query* gliedern sich in mehrere Unterzustände. Der Zustand *Display archiv object* unterscheidet zunächst, ob ein Verzeichnis oder ein Medienobjekt dargestellt werden soll. Je nachdem verzweigt der Automat zu den Zuständen *Display subdirectories* und *Display media objects* oder *Display properties* und *Display content*. In den Zustand *Display content* verzweigt der Automat nur, falls es sich beim ausgewählten Medienobjekt um ein darstellbares handelt. Im Zustand *Display query* werden die Unterzustände *Enter search string* und *Display search result* durchlaufen. Beim Übergang zwischen diesen bei-

Abbildung 6.6: Anzeige eines Medienobjekts

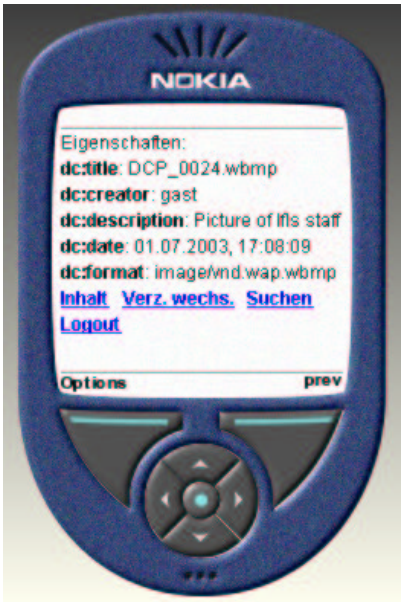

Abbildung 6.7: Anzeige der Eigenschaften

den Zuständen erfolgt die Suchanfrage an das Medienarchiv. Werden auf die Anfrage hin mehr Ergebnisse gefunden, als auf einer Seite darstellbar sind, werden diese auf mehrere Seiten verteilt, zwischen denen hin und her geschaltet werden kann. Dabei wechselt der Automat nicht den Zustand.

Zum Ende dieses Abschnitts folgt die Implementierung von zwei Methoden. Sie realisieren in der Anwendung einen Teil der Ein- und Ausgabe.

```
1  public p queryRequestAsWML() {
2    String url;
3    p para;
4
5    url = encodeURL("WMLSession?link=execSearch&searchQuery
       =$(searchQuery)");
6
7    // Suchstring eingeben
8    para = <p>Suchstring:<br/>
9             <input type="text" name="searchQuery" value=""
                  maxlength="32"/>
10            <br/>
11            <a accesskey="7" href="{url}">Suche starten</a
                 >
12          </p>;
13
```

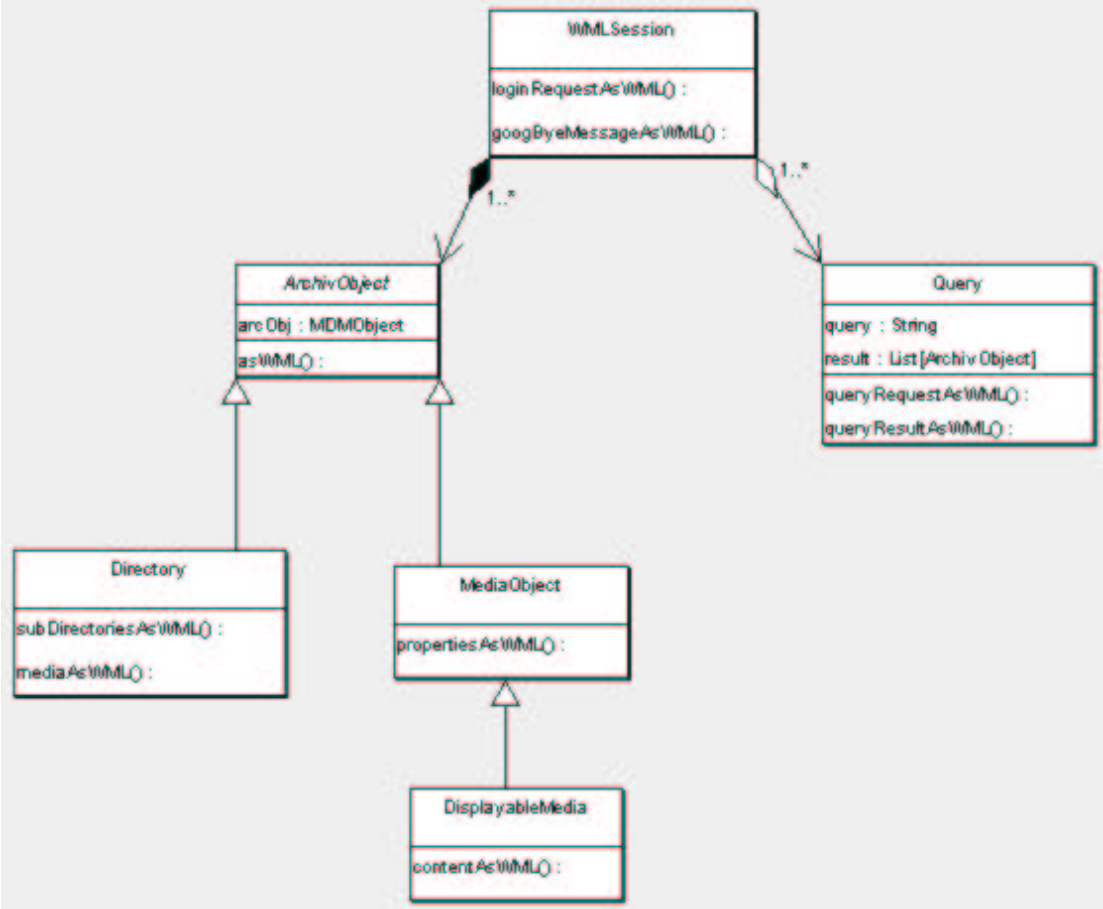

Abbildung 6.8: Datenmodell der WML-Anbindung

```
14    return para;
15  } // queryRequestAsWML
```

Die Methode `queryRequestAsWML` erzeugt ein WML-Fragment mit dem der Benutzer zur Eingabe einer Zeichenkette aufgefordert wird, nach der im Medienarchiv gesucht werden soll. Es ist ersichtlich, dass dafür ein `p`-Element mit einem Formularfeld (9) im Inhalt der XML-Objekt-Variablen `para` zugewiesen wird (8-12). Durch einen Hyperlink (11) kann der Anwender die Suche aktivieren. Die Referenz (7), die durch dieses Ereignis aktiviert wird, übergibt den aktivierten Link und die eingegebene Zeichenkette.

Das etwas längere Beispiel zeigt die Methode `subDirectoriesAsWML`.

```
1  public p subDirectoriesAsWML() {
2    String path, name, subDir, parentDir;
3    String[] subDirs;
4    p para;
```

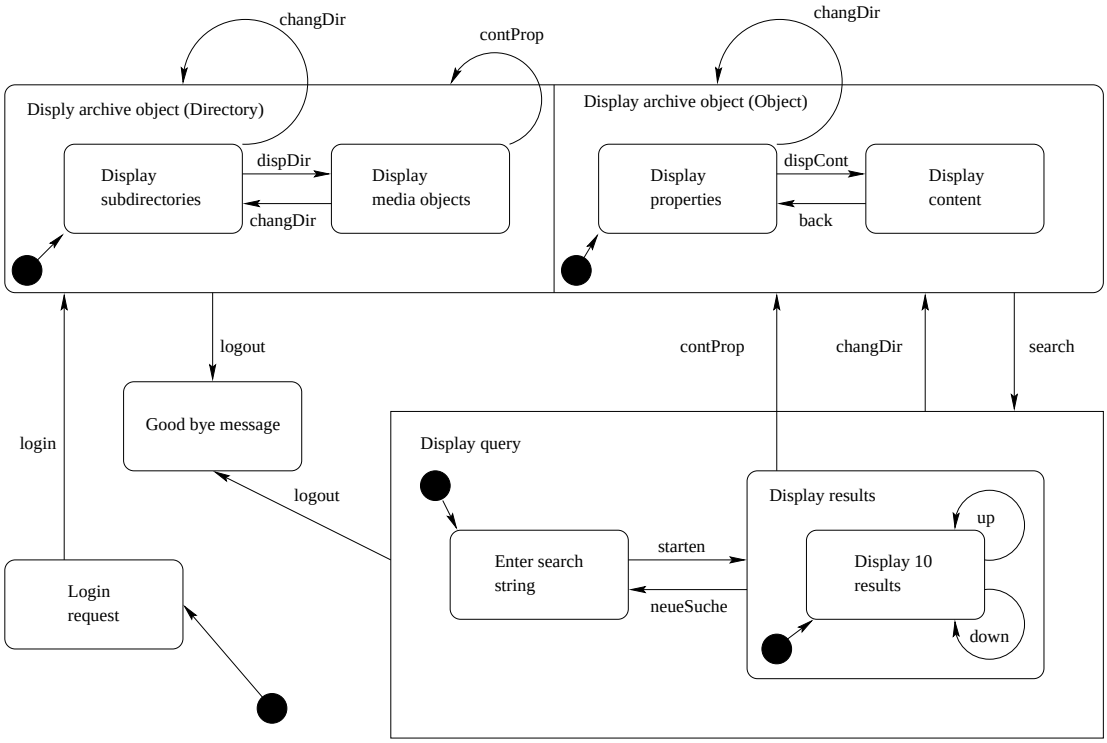

Abbildung 6.9: Zustandsautomat der WML-Anbindung

```
5   xml<(option)*> opts;
6
7   try {
8     path = arcObj.getFullPath();
9     name = arcObj.getName();
10    subDirs = arcObj.getChilds(1);
11    if (path.equals("/" + name))
12      parentDir = "/workspace";
13    else
14      parentDir = path.substring(0,parentDir.length() -
          name.length() - 1);
15
16    opts = <>;
17    for (int i=0; i < subDirs.length; i = i + 1) {
18      subDir = path + "/" + subDirs[i];
19      opts = opts + <option value="{subDir}">{subDirs[i
          ]}</option>;
20    } // for
21
```

```
22      urlChange = encodeURL("WMLSession?link=changeDir&
           searchQuery=$(directory)");
23      urlFiles = encodeURL("WMLSession?link=showFiles");
24
25      para =
26        <p>
27          <b>{arcObj.getFullPath()}</b><br/>
28          <select name="direcory">
29            <option value="{parentDir}">..</option>
30            {opts}
31          </select><br/>
32          <a accesskey="9" href="{urlChange}">Verz.wechs.</
               a>
33          <a accesskey="2" href="{urlFiles}">Verz.anz.</a>
34        </p>;
35    } // try
36    catch (RemoteException e) {
37      para = <p> Fehler: {e} </p>;
38    } // catch
39
40    return para;
41  } // subDirectoriesAsWML
```

In dieser Methode wird ebenfalls ein Element der Klasse `p` erzeugt und zurückgegeben. Der Inhalt besteht u. a. aus einer Liste von `option`-Elementen, die in einer Schleife (17-20) erzeugt wird. Damit ist es für den Anwender möglich, über eine Auswahlliste in eines der Unterverzeichnisse zu wechseln.

6.2 Übungsdatenverwaltung

In diesem Abschnitt wird als zweite Anwendung eine Übungsdatenverwaltung (ÜDV) beschrieben, die unter Verwendung der Java-Erweiterung XOBE realisiert worden ist. Die Anwendung entstand im Rahmen einer Studienarbeit [Spi02] am Institut für Informationssysteme in reiner Java-Servlet-Technik. Für diese Arbeit wurden Teile dieser Implementierung mittels XOBE neu geschrieben. Eine Übungsdatenverwaltung ist eine Anwendung, die in der universitären Lehre eingesetzt wird. In vielen Lehrveranstaltungen müssen von den Teilnehmenden sogenannte Scheinkriterien erfüllt werden, um am Ende des Semesters einen sogenannten Übungs- oder Teilnahmeschein zu erhalten. Scheinkriterien können dabei in den unterschiedlichsten Formen auftreten, wie z. B. Übungsaufgaben, Vorträgen, Ausarbeitungen oder Klausuren. Die Übungsdatenverwaltung dient zum Verwalten der durch die Scheinkriterien anfallenden Daten. Es werden Daten über Veranstaltungen, teilnehmende Studierende, Scheinkriterien und die erzielten

Ergebnisse der Studierenden für Scheinkriterien im System abgespeichert.

Durch die Übungsdatenverwaltung ergeben sich einige Erleichterungen für das Durchführen der Lehrveranstaltungen. Arbeitsabläufe, die vorher von Hand bearbeitet werden mussten, werden nun durch die Anwendung automatisch erledigt. Dazu zählt der Aushang der Ergebnislisten, der nach Korrektur einer Klausur vom System generiert wird. Ebenso werden die Übungs- oder Teilnahmescheine, die nach erfolgreicher Teilnahme an die Studierenden ausgehändigt werden, durch die Übungsdatenverwaltung erzeugt.

6.2.1 Arbeitsweise und Benutzerzugang

Die Übungsdatenverwaltung wird über die Adresse der Startseite aufgerufen. Dem Anwender präsentiert sich die Aufforderung zur Eingabe von Benutzernamen und Kennwort. Durch den

Abbildung 6.10: Aufforderung zur Anmeldung

Schutz der Daten mittels unterschiedlicher Benutzerrechte, wird der Zugriff auf die Daten im System stark eingeschränkt. Nur berechtigte Personen erhalten die Möglichkeit bestimmte Teile der Daten einzufügen, zu verändern oder zu löschen. Im Sekretariat können Veranstaltungen und Studierende eingefügt und korrigiert werden. Der Dozent ist in der Lage die Scheinkriterien für die von ihm gehaltenen Lehrveranstaltung zu definieren. Leistungsergebnisse der Studierenden werden vom korrigierenden Assistenten eingetragen.

Nach dem Anmelden im System kann der Benutzer eine Lehrveranstaltung auswählen, für die er Daten abfragen, eintragen oder ändern möchte. Selbstverständlich gibt es auch die Möglichkeit

Abbildung 6.11: Auswahl der Veranstaltungen

eine neue Veranstaltung, die noch nicht in der Übungsdatenverwaltung eingetragen ist, hinzuzufügen. Dazu werden Daten, wie Name der Lehrveranstaltung, Name des anbietenden Dozenten, Semesterangabe und Web-Adresse benötigt.

Hat der Anwender eine Lehrveranstaltung ausgewählt, bieten sich ihm die Möglichkeiten Scheinkriterien für die Veranstaltung zu definieren, teilnehmende Studierende einzutragen, die Leistungen der teilnehmenden Studierenden für die Scheinkriterien zu editieren und Listen bzw. Scheine zu erstellen. Weiterhin können Lehrveranstaltungen in kleine Übungen aufgeteilt und gruppiert werden.

Studierende, die an einer Lehrveranstaltung teilnehmen, aber noch nicht im System bekannt sind, können über eine eigene Seite eingetragen werden. Für Studierende werden die Angaben Name, Matrikelnummer, Geburtsdatum, Emailadresse sowie Studiengang abgelegt. Ist ein Studierender dem System schon aus einer vorherigen Lehrveranstaltung bekannt, entfällt die erneute Eingabe der Daten; er kann direkt als Teilnehmer der Veranstaltung zugeordnet werden.

Studierende, die einer Lehrveranstaltung zugeordnet sind, also an dieser teilnehmen, kann das System anzeigen. Es besteht die Möglichkeit Teilnehmerlisten und Listen mit Emailadressen zu

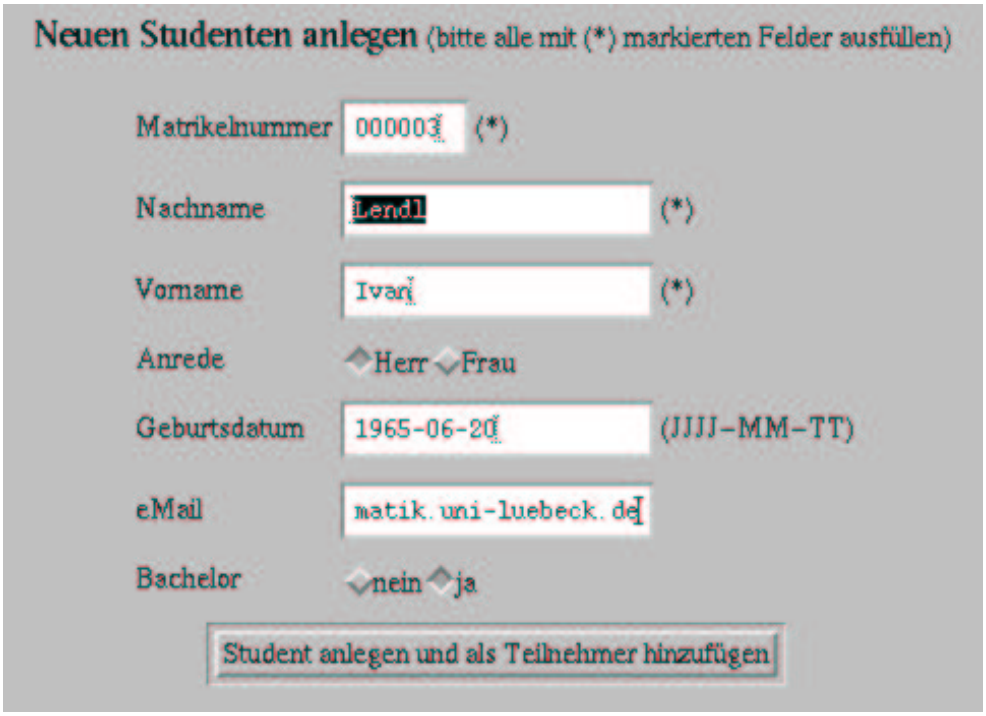

Abbildung 6.12: Eintragen eines neuen Studierenden

erstellen. Letztere dient dazu, um alle teilnehmenden Studierenden über einen elektronischen Rundbrief erreichen zu können.

Wie erwähnt, können für Lehrveranstaltungen Scheinkriterien definiert werden. Jede Anforderung besteht u. a. aus einer Bezeichnung, einer maximal erreichbaren Punktzahl, einer Bestehensgrenze und einem Abgabedatum.

Für eine ausgewählte Lehrveranstaltung können die durch die teilnehmenden Studierenden erbrachten Leistungen eingesehen und eingetragen werden. In einer Gesamtübersicht werden zunächst sämtliche Anforderungen mit den bisherigen Ergebnissen dargestellt. Für das Eintragen neuer Resultate muss ein Scheinkriterium ausgewählt werden. Für dieses lassen sich dann die Ergebnisse mit erzielten Punktzahlen für jeden Studierenden eintragen.

Für jede Vorlesung können eine Reihe von Listen und Scheinen erstellt werden. Es gibt Listen mit allen Teilnehmern der Veranstaltung, leere Listen zum Eintragen von Studierenden, Leistungsübersichten für bestimmte Scheinkriterien sowie Ausfalllisten für Klausuren und mündliche Rücksprachen. Weiterhin ist es möglich eine archivierbare Gesamtübersicht am Semesterende zu erstellen. Über einen weiteren Punkt können für Studierende, die die Scheinkriterien erfolgreich erfüllt haben, die Übungs- oder Teilnahmescheine erstellt werden. Die Überprüfung der definierten Scheinkriterien vollzieht das System dabei automatisch.

Abbildung 6.13: Anzeigen der teilnehmenden Studierenden

6.2.2 Architektur und Implementierungsdetails

Ähnlich zur WML-Anbindung ist für die Architektur der Übungsdatenverwaltung ebenfalls eine Umsetzung in mehreren Schichten gewählt worden. Die an der Anwendung beteiligten Daten werden in einem Objektmodell repräsentiert. Das Verhalten der Ein- und Ausgabe wird über einen Automaten gesteuert während die Ein- und Ausgabe selbst, die durch XHTML erfolgt, durch eine separate Präsentationsschicht definiert wird. Dieser Abschnitt stellt die drei Schichten Objektmodell, Automat und Präsentation in dieser Reihenfolge kurz vor. Neben den drei schon angesprochenen Schichten wird für die Kommunikation zur Datenbank eine weitere benötigt, worauf in dieser Arbeit aber nicht näher eingegangen wird.

In Abbildung 6.17 ist das Objektmodell der Übungsdatenverwaltung zu sehen. Mit Objekten der Klasse *ÜDVSession* wird die Sitzung eines Benutzers zum Informationssystem realisiert. Sie besteht aus zwei optionalen Objekten der Klassen *Veranstaltung* und *Übung*.

Die Objekte der Klasse *Veranstaltung* repräsentieren die Daten für Lehrveranstaltungen. Sie stehen in Beziehung mit Objekten der Klassen *Raum*, *Zeit* und *Dozent*, die entsprechend alle relevanten Daten über Räume, Zeiten und Dozenten modellieren. Analoges gilt für Anforderungen, Studierende und Übungsgruppen, die durch Objekte der Klassen *Anforderung*, *Studierender* und *Übung* repräsentiert werden. Die genauen Daten, die im System gespeichert werden sind der Abbildung 6.17 zu entnehmen und werden hier nicht weiter beschrieben.

Interessanter sind dagegen die Beziehungen, die zwischen den Klassen bestehen. Jede Veranstaltung und jede Übung findet an bestimmten Orten und Zeiten statt. Eine Beziehung zwischen Veranstaltung und Dozent bzw. zwischen Übung und Dozent gibt an, welcher Dozent die Lehrveranstaltung oder Übung leitet. Für jede Veranstaltung kann eine Menge von Scheinkriterien definiert werden, was mit einer Assoziation zwischen den Klassen *Veranstaltung* und *Anforderung*

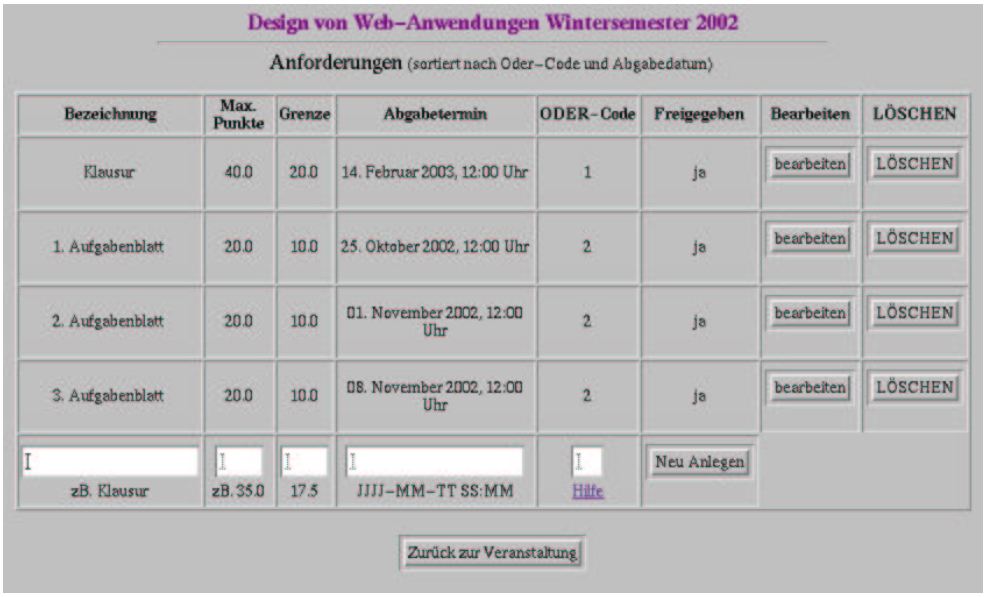

Abbildung 6.14: Scheinkriterien festlegen

ausgedrückt wird. Die Information, welcher Studierende welche Veranstaltung oder Übung besucht, wird durch die Beziehung zwischen den Klassen *Veranstaltung*, *Studierender* und *Übung* repräsentiert. Wichtig ist auch die Assoziation zwischen Studierenden und den Anforderungen, denn dort werden die Daten gespeichert, die angeben, welcher Studierende welches Scheinkriterium mit welcher Punktzahl absolviert hat.

Der Automat, den die Abbildung 6.18 darstellt, beschreibt das Ein- und Ausgabeverhalten der Übungsdatenverwaltung. Er besteht aus den sechs Zuständen *Login-Aufforderung*, *Veranstaltung auswählen*, *Veranstaltung gewählt*, *Administration*, *Veranstaltung anlegen* und *Auf-Wiedersehen-Meldung*. Der Automat wird im Zustand *Login-Aufforderung* gestartet, von dem aus dieser in den Zustand *Veranstaltung auswählen* übergeht. Dieser zentrale Zustand lässt eine Verzweigung in die Zustände *Veranstaltung gewählt*, *Administration* und *Veranstaltung anlegen* zu. Weiterhin ist auch ein Übergang in den Zustand *Auf-Wiedersehen-Meldung* möglich, der den Ablauf des Automaten beendet. Auch bei diesem Automaten werden alle Ereignisse, die zu Zustandsübergängen führen, durch Eingaben des Anwenders ausgelöst.

Um einen Eindruck zu bekommen, wie XOBE in der Anwendung Verwendung findet, werden im restlichen Abschnitt zwei Methoden aus der Implementierung vorgestellt. Die Methode `veranstaltungenAsTable` stellt eine Liste von Veranstaltungen in einer XHTML-Tabelle dar.

```
1 private table veranstaltungenAsTable(List courses) throws
      SQLException {
2   Iterator iter;
3   xml<(font)?> subTitle;
4   xml<(td)+> descs;
```

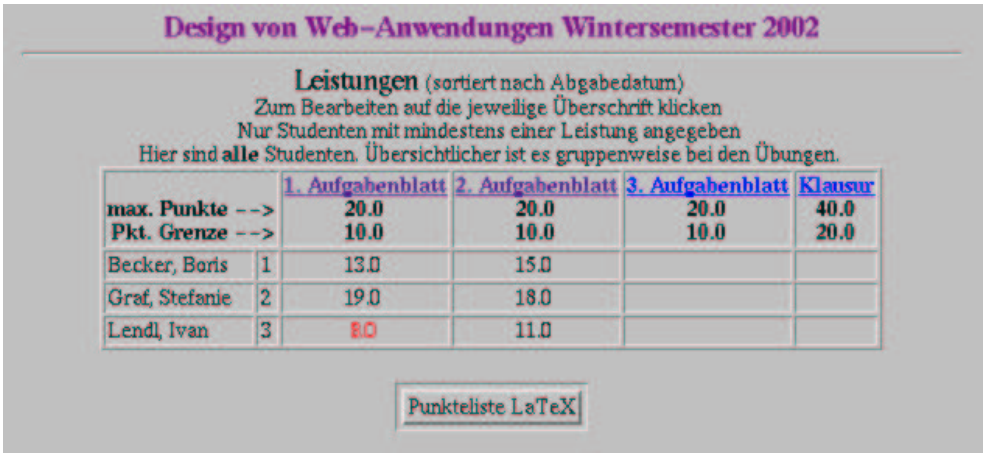

Abbildung 6.15: Eintragen der Leistungen

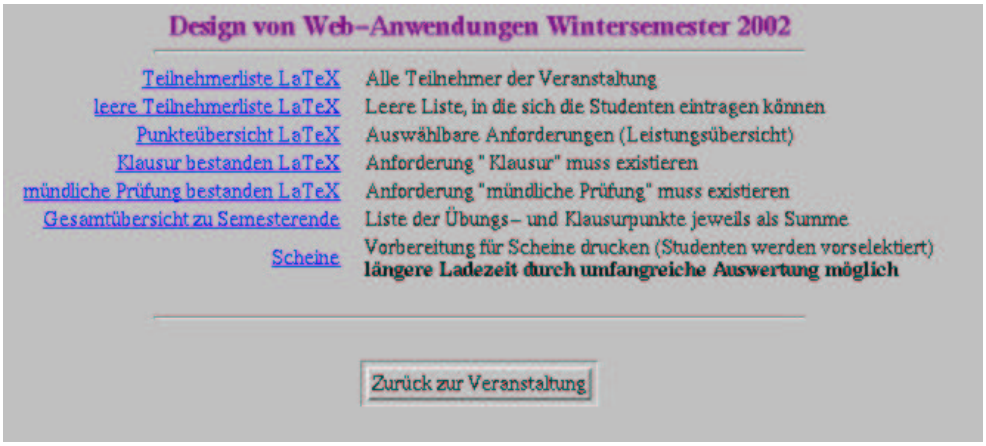

Abbildung 6.16: Erstellen von Listen und Scheinen

```
5   xml<(tr)+> rows;
6   Veranstaltung course;
7
8   rows = <tr>
9             <td><center><b>Bezeichnung</b></center></td>
10            <td><center><b>Semester</b></center></td>
11            <td><center><b>Dozent</b></center></td>
12          </tr>;
13
14  iter = courses.iterator();
15  while(iter.hasNext) {
16    course = (Veranstaltung) iter.next();
```

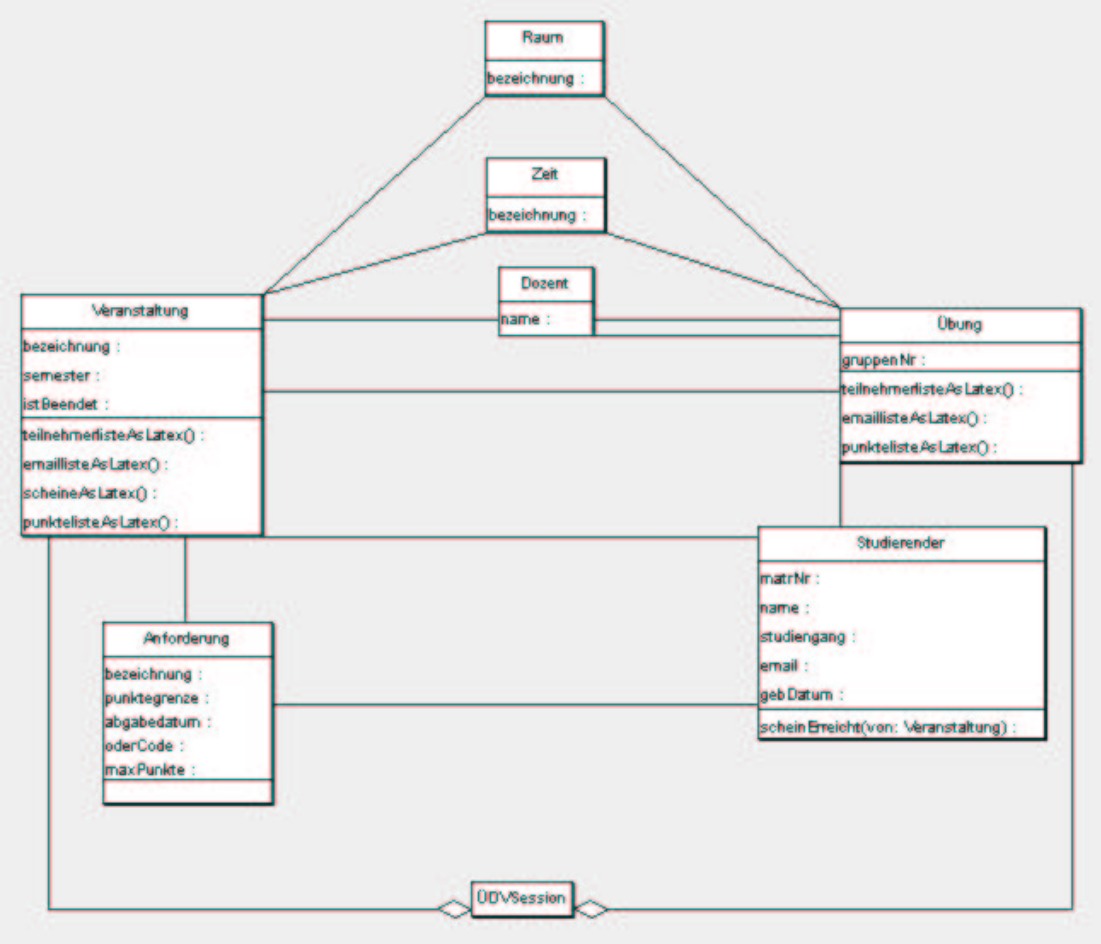

Abbildung 6.17: Datenmodell der Übungsdatenverwaltung

```
17
18      if (course.hasUntertitel())
19        subTitle = <font size="-2">{course.getUntertitel()
            }</font>;
20      else
21        subTitle = <>;
22
23      descs = <td><a href="verwaltungsservlet?link=
            veranstaltung_detail&veranstaltung_ID={course.
            getID()}">{course.getTitel()}</a>{subTitle}</td>;
24      descs = descs + <td>{course.getSemester()}</td>;
25      descs = descs + <td>{course.getDozent()}</td>;
26
27      rows = rows + <tr>{descs}</tr>;
```

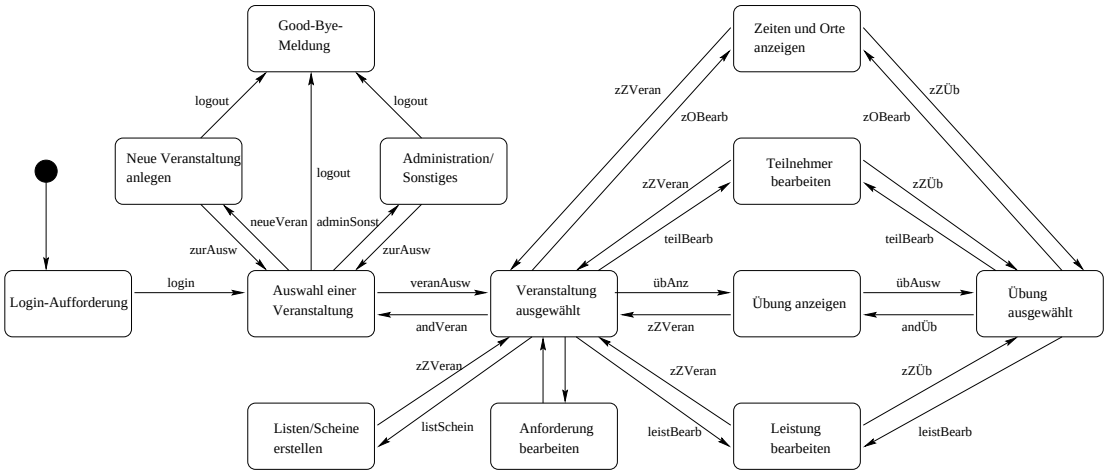

Abbildung 6.18: Zustandsautomat der Übungsdatenverwaltung

```
28      } // while
29
30      return <table>{rows}</table>;
31    } // veranstaltungenAsTable
```

Zunächst werden der Variablen `rows` die Überschriften der drei Spalten der Tabelle zugewiesen (8-12), die die Zeilen der neuen Tabelle zwischenspeichert. Anschließend werden alle Veranstaltungen der Reihe nach mittels eines Iterators (14-28) durchgegangen. Falls eine Veranstaltung einen Untertitel besitzt (18), wird dieser in der Variablen `subTitle` (19,21) abgelegt. Die Einträge der Spalten mit Titel der Veranstaltung, Semester und Dozent werden der Variablen `descs` zugewiesen (23-25). Der Titel ist dabei ein Hyperlink, der zu den Details der Veranstaltung führt. Die Methode endet mit der Rückgabe eines `table`-Elements.

In der zweiten Methode `veranstaltungWaehlenAsHTML`, die hier beispielhaft präsentiert wird, erfolgt ein Aufruf der Methode `veranstaltungenAsTable`.

```
1   private center veranstaltungWaehlenAsHTML() {
2     List courses;
3     center tab;
4
5     try {
6       courses = Veranstaltung.alleVeranstaltungen(false);
7       tab = <center>{veranstaltungenAsTable(courses)}</
            center>;
8     } // try
9     catch (SQLException e) {
10      tab = <center>Fehler: {e}<br/><br/></center>;
11    } // catch
```

```
12
13    return
14      <center>
15        <font size="+2">Bitte Veranstaltung wählen:</font>
16        {tab}
17        <br/>
18        <form action="verwaltungsservlet" method="POST">
19          <input type="hidden" name="link" value="
              alle_veranstaltungen_anzeigen"/>
20          <input type="submit" value="Auch␣beendete␣
              Veranstaltungen␣anzeigen"/>
21        </form>
22        <form action="verwaltungsservlet" method="POST">
23          <input type="hidden" name="link" value="
              veranstaltung_anlegen"/>
24          <input type="submit" value="Neue␣Veranstaltung␣
              anlegen"/>
25        </form>
26        <br/>
27        <form action="verwaltungsservlet" method="POST">
28          <input type="hidden" name="link" value="
              administration"/>
29          <input type="submit" value="Administration␣/␣
              Sonstiges"/>
30        </form>
31      </center>;
32  } // veranstaltungWaehlenAsHTML
```

Sie dient dazu den Teil einer Seite zu erstellen, der sämtliche noch nicht beendete Veranstaltungen auflistet. Es entspricht dem Hauptmenü der Anwendung, von dem aus in alle wichtigen Teile verzweigt werden kann. Der Variablen `courses` werden alle Veranstaltungen zugewiesen (6), die noch nicht beendet sind. Anschließend werden diese in eine XHTML-Tabelle umformatiert und der XML-Objekt-Variablen `tab` zugewiesen (7). Tritt dabei eine Ausnahme auf, wird die Variable `tab` auf deren Fehlermeldung gesetzt (9-11). Anschließend wird die Tabelle mit drei weiteren Menüpunkten zurückgegeben (13-31). Mit einem Punkt ist es möglich auch beendete Veranstaltungen einzusehen (18-21), ein weiterer dient zum Anlegen von neuen Veranstaltungen (22-25) und der dritte Punkt verzweigt zur Administration der Anwendung (27-30). Dort können u. a. Daten auf externen Datenträgern archiviert werden.

Kapitel 7

Zusammenfassung und Ausblick

In dieser Arbeit wurde die Erweiterung **X**ML-***Objekte*** (XOBE) der objektorientierten Programmiersprache Java vorgestellt, die eine Programmierung mit XML-Strukturen ermöglicht. Durch die Deklaration der Sprachbeschreibung einer Auszeichnungssprache im XOBE-Programm können die in der Sprachbeschreibung deklarierten Elementtypen wie in Java eingebaute Klassen eingesetzt werden. Mit dieser Bekanntmachung der verwendeten Auszeichnungssprache im Quelltext der Anwendung kann zur Zeit der Programmübersetzung weitestgehend sichergestellt werden, dass die dynamisch generierten XML-Strukturen gültig sind. Die Ausdruckskraft von Java wird dadurch nicht beeinträchtigt. Die bisherigen Spracherweiterungen von Java, die für die Programmierung von Web-Anwendungen definiert wurden, können diese Eigenschaft nicht garantieren.

In XOBE können alle durch Deklaration der Auszeichnungssprache bekannt gemachten XML-Typen als Klassen verwendet werden. Damit ist es möglich, durch spezielle Konstruktoren, die in XML-Syntax angegeben werden, XML-Objekte zu erzeugen, die wie ganz normale Java-Objekte im Programm eingesetzt werden können. Ein Zugriff auf den Inhalt von XML-Objekten wurde durch den Sprachstandard XPath ermöglicht.

Eine Formalisierung des Typsystems in XOBE wurde durch Heckensprachen mit den dazugehörigen Heckengrammatiken vorgenommen. Darauf aufbauend wurde ein Algorithmus zur Überprüfung der Subtyp-Beziehung zweier allgemeiner regulärer Heckenausdrücke vorgestellt und dessen Korrektheit bewiesen. Weiterhin wurde gezeigt, wie eine Erweiterung des Algorithmus erfolgen kann, um die zusätzlichen Möglichkeiten des Typsystems in XML-Schema zu berücksichtigen. Da allgemeine Heckengrammatiken ausdrucksstärker sind als Sprachbeschreibungen von Auszeichnungsspachen, vereinfacht sich für XML der Suptyp-Algorithmus in der Ausführung, wie in Abschnitt 4.8.2 gezeigt, was mit einer erheblichen Effizienzsteigerung verbunden ist.

Bei der Transformation von XOBE-Programmen in reines Java werden die XML-Objekte in eine Java-Implementierung übersetzt. In dieser Arbeit wurde für diesen Zweck der Programmier-

standard DOM gewählt. Die Selektionen von XML-Objekt-Inhalten durch Ausdrücke in XPath werden ebenfalls in Programmteile umgesetzt, die auf der DOM-Repräsentation beruhen.

Durch die Realisierung zweier Web-Anwendungen mit Hilfe der vorgestellten Java-Erweiterung XOBE wurde evaluiert, dass der Einsatz in der Programmierung problemlos möglich und sinnvoll ist. Dabei zeigte sich, dass die geringe Effizienzeinbuße, die durch die aufwendigere DOM-Implementierung entsteht, in der Praxis tolerierbar ist.

Die zu Beginn gesteckten Ziele aus Abschnitt 1.2 sind wie folgt erreicht worden:

1. Integration von XML-Strukturen in das objektorientierte Klassenkonzept:

   XOBE erweitert die Programmiersprache durch die Integration von XML-Strukturen. Dabei werden die Sprachbeschreibungen von Auszeichnungssprachen wie Klassendefinitionen interpretiert und stehen implizit zur Verfügung. Damit konnte das erste Ziel voll erfüllt werden.

2. Komfortable Zugriffsmöglichkeiten auf Inhalte dieser XML-Strukturen:

   Durch die Integration der Sprache für Pfadausdrücke XPath in XOBE ist es möglich auf den Inhalt von XML-Objekte zuzugreifen. Die Selektion erfolgt damit über einen weit verbreiteten und akzeptierten Standard in XML.

3. Weitestgehende Garantie der Gültigkeit der generierten Strukturen bereits zur Zeit der Programmübersetzung:

   Durch das auf XML-Objekte zugeschnittene Typsystem in XOBE kann die geforderte weitestgehende Garantie der Gültigkeit für die im XOBE-Programm verarbeiteten XML-Strukturen sichergestellt werden. Damit ist es für eine in Java implementierte Web-Anwendung erstmals möglich, auf einen Großteil der aufwendigen Testläufe zu verzichten.

Die bei der Einordnung der Arbeit (Abschnitt 2.7) dargestellte Situation, dass nach heutigem Stand viele Web-Anwendungen mit Techniken implementiert werden, durch die die Korrektheit der erzeugten XML-Fragmente gar nicht oder nur in geringem Maße sichergestellt werden, kann mit der Java-Erweiterung XOBE erheblich verbessert werden. Gerade für die Programmierung von Web-Anwendungen lässt sich XOBE gut einsetzen.

Auch wenn die zuvor genannten Ziele erreicht wurden, ergibt sich eine Fülle von weiteren Aufgaben und Erweiterungsmöglichkeiten, die im Folgenden angesprochen werden. Die bereits in den Kapiteln zum Typsystem (4.8) und der Implementierung (5.4) diskutierten Vor- und Nachteile oder möglichen Ergänzungen bzw. Änderungen werden hier bis auf eine wesentliche Ausnahme nicht noch einmal aufgeführt.

Zukünftige Arbeiten

Erweiterung des Objektmodells um Textknoten

Es kann sinnvoll sein, das XOBE zu Grunde liegende Objektmodell so zu erweitern, so dass (ähnlich dem DOM) Instanzen einer eigenen Textklasse den textuellen Inhalt von Elementen repräsentieren. Damit wird es möglich, Änderungsoperationen, die sich auf den textuellen Inhalt beziehen, direkt auf diesen Textknoten durchzuführen. Im aktuellen Objektmodell, in dem keine speziellen Textobjekte existieren, wird textueller Inhalt durch Zeichenketten dargestellt. Dadurch haben solche Änderungen stets über den Elternknoten, also den beinhaltenden Elementknoten, zu geschehen.

Manipulation von XML-Objekten

Zur Zeit ist in XOBE noch keine direkte Manipulation von XML-Fragmenten vorgesehen. Bisher können nur Elemente, Attribute und Inhalte aus einem XML-Objekt mittels XPath selektiert und zu neuen XML-Objekten durch Konstruktoren zusammengefügt werden. Eine Erweiterung von XOBE könnte also durch neue Sprachkonstrukte erfolgen, die die Änderung von Daten im XML-Objekt oder das Löschen von Elementen, Attributen oder Inhalten erlauben. Diese Sprachkonstrukte sollten sinnvollerweise so konstruiert sein, dass weiterhin eine Überprüfung der statischen Gültigkeit zum Zeitpunkt der Programmübersetzung möglich ist.

Basisdatentypen in XML-Schema

Eine weitere zukünftige Untersuchungsmöglichkeit existiert mit den Basisdatentypen in XML-Schema, für die zu betrachten ist, in welcher Form sie in XOBE integriert werden können. Der aktuelle Prototyp unterstützt lediglich die beiden Datentypen `integer` und `string`, was die Möglichkeiten von XML-Schema stark vereinfacht. Problematisch wird die Verknüpfung von Basisdatentypen mit XOBE, weil unterschiedliche Typen sich in ihrer Darstellung als Zeichenkette stark überschneiden. So kann beispielsweise die Zeichenkette `11` als `integer`, als `positiveInteger`, als `float` oder sogar als `string` interpretiert werden.

Erweiterung von XPath um strukturorientierte Selektion

Wie in Abschnitt 5.4 bereits angesprochen wurde, erfolgt die Selektion von Elementen, Attributen und Inhalten in XOBE zur Zeit ausschließlich durch XPath. XPath selbst formuliert Zugriffe auf XML-Objekte ohne Kenntnis der Sprachbeschreibung. Da in XOBE aber stets eine Sprachbeschreibung vorliegt, wäre auch eine Selektion denkbar, die sich an der deklarierten Sprachbeschreibung orientiert. Beispielsweise sollte für ein Inhaltsmodell der Form `(book*,record,`

`book*,record,book*)` eine direkte Selektion der zweiten `book`-Liste möglich sein. In XPath ist ein solcher Zugriff nur sehr umständlich ausdrückbar (`child::book[preceding-sibling::record and following-sibling::record]`).

Andere Implementierung als DOM

Eng verbunden mit strukturorientierten Selektionsmöglichkeiten ist die Frage nach einer effizienteren Implementierung der XML-Objekte. Denkbar wäre hier eine ebenfalls an der Struktur der XML-Objekt-Klassen orientierte Repräsentation. Verbindet man diese Darstellung mit der bereits angesprochenen strukturorientierten Selektion, lassen sich bestimmte Selektionsoperationen, wie der Zugriff auf die zweite `book`-Liste im vorherigen Beispiel mit konstantem Aufwand ausführen. Die aktuelle DOM-Implementierung benötigt dafür in der Abhängigkeit zur Knotenliste linear viele Zugriffe.

Persistenz von XML-Objekten

Zuletzt wäre es sicherlich wünschenswert, XML-Objekte persistent ablegen zu können, wofür eine Verbindung der Programmiersprache XOBE mit einem Datenbanksystem notwendig wäre. Mit weiteren Sprachkonstrukten wäre dann eine Erweiterung zu einer eigenständigen Datenbankprogrammiersprache möglich, in der sich vollwertige Datenbankanwendungen implementieren ließen. Gerade im Hinblick auf Web-Anwendungen, die mehr und mehr mit Datenbanken verbunden werden, erscheint eine solche Zielsetzung zweckmäßig

Anhang A

XML-Schema AOML

Dieser Anhang präsentiert die Auszeichnungssprache des Beispiels 2.2 dieser Arbeit in Form eines XML-Schemas.

```
1  <schema xmlns="http://www.w3.org/2001/XMLSchema-instance"
       >
2    <element name="aoml">
3      <complexType>
4        <sequence>
5          <element name="antiquary" type="t_antiquary"/>
6          <element name="offer" type="t_offer"/>
7        </sequence>
8        <attribute name="date" type="string"/>
9      </complexType>
10   </element>
11
12   <complexType name="t_antiquary">
13     <sequence>
14       <element name="name" type="string"/>
15       <element name="address" type="string"/>
16       <element name="email" type="string"/>
17     </sequence>
18   </complexType>
19
20   <complexType name="t_offer">
21     <choice maxOccurs="unbounded">
22       <element name="book" type="t_book"/>
23       <element name="record" type="t_record"/>
24     </choice>
25   </complexType>
```

```
26
27    <complexType name="t_book">
28      <sequence>
29        <element name="title" type="string"/>
30        <element name="author" type="string" minOccurs="0"
              />
31        <group ref="field"/>
32      </sequence>
33      <attribute name="catalog" type="string"/>
34    </complexType>
35
36    <complexType name="t_record">
37      <sequence>
38        <element name="title" type="string"/>
39        <element name="artist" type="string"/>
40        <group ref="field"/>
41      </sequence>
42    </complexType>
43
44    <group name="field">
45      <sequence>
46        <element name="article" type="integer"/>
47        <element name="condition" type="string"/>
48        <element name="price" type="t_price"/>
49      </sequence>
50    </group>
51
52    <complexType name="t_price">
53      <simpleContent>
54        <extension base="string">
55          <attribute name="currency" type="string" use="
                required"/>
56        </extension>
57      </simpleContent>
58    </complexType>
59  </schema>
```

Listing A.1: Schemadefinition AOML

Anhang B

Beweis von Satz 4.2

Um den Satz 4.2 aus Abschnitt 4.6 zu beweisen, was in diesem Anhang geschieht, wird zunächst der folgende Hilfssatz gezeigt. Dabei sei Ω die umfassende Menge.

Hilfssatz B.1 (Teilmengenbeziehung des Karthesischen Produkts)
Gegeben seien die Mengen a, b, c und d, dann gilt:

$$(a \times b) \subseteq (c \times \Omega) \cup (\Omega \times d) \quad \Leftrightarrow \quad (a \times b) \subseteq (c \times \Omega) \vee a \times b \subseteq (\Omega \times d)$$

Beweis:

"$\Rightarrow$" Hinrichtung: Indirekter Beweis

Angenommen es gilt:

$$(a \times b) \nsubseteq (c \times \Omega) \vee (a \times b) \nsubseteq (\Omega \times d) \quad \Rightarrow \quad (a \times b) \nsubseteq (c \times \Omega) \cup (\Omega \times d)$$

"$\Leftarrow$" Rückrichtung: Direkter Beweis mit Fallunterscheidung

1. Fall: Angenommen es gilt:

$$(a \times b) \subseteq (c \times \Omega) \quad \Rightarrow \quad (a \times b) \subseteq (c \times \Omega) \cup (\Omega \times d)$$

2. Fall: Angenommen es gilt:

$$(a \times b) \subseteq (\Omega \times d) \quad \Rightarrow \quad (a \times b) \subseteq (c \times \Omega) \cup (\Omega \times d)$$

□

Es folgt der Beweis von Satz 4.2.

Satz 4.2 (Teilmengenbeziehung des Karthesischen Produkts)
Gegeben seien die Mengen a, b, $c_1, \ldots, c_n$ und $d_1, \ldots, d_n$, dann gilt:

$$a \times b \subseteq (c_1 \times d_1) \cup \cdots \cup (c_n \times d_n)$$
$$\Leftrightarrow$$
$$(a \subseteq \cup_{i \in I_1} c_i \vee b \subseteq \cup_{i \in \overline{I}_1} d_i) \wedge \cdots \wedge (a \subseteq \cup_{i \in I_{2^n}} c_i \vee b \subseteq \cup_{i \in \overline{I}_{2^n}} d_i)$$
$$\text{mit } \mathcal{P}(\{1, \ldots, n\}) = \{I_1, \ldots, I_{2^n}\} \text{ und } \overline{I}_i = \{1, \ldots, n\} \backslash I_i \; .$$

Direkter Beweis:
Angenommen es gilt $r \not\leq s$, dann folgt:

$$\begin{aligned}
& a \times b \subseteq (c_1 \times d_1) \cup \cdots \cup (c_n \times d_n) \\
\Leftrightarrow \quad & a \times b \subseteq ((c_1 \times \Omega) \cap (\Omega \times d_1)) \cup \cdots \cup ((c_n \times \Omega) \cap (\Omega \times d_n)) \\
\Leftrightarrow \quad & a \times b \subseteq ((c_1 \times \Omega) \cup (c_2 \times \Omega) \cup \cdots \cup (c_n \times \Omega)) \cap \\
& ((c_1 \times \Omega) \cup (c_2 \times \Omega) \cup \cdots \cup (c_{n-1} \times \Omega) \cup (\Omega \times d_n)) \cap \cdots \cap \\
& ((\Omega \times d_1) \cup (\Omega \times d_2) \cup \cdots \cup (\Omega \times d_n)) \\
\Leftrightarrow \quad & a \times b \subseteq ((c_1 \cup \cdots \cup c_n \times \Omega) \cup (\Omega \times \emptyset)) \cap \\
& ((c_1 \cup \cdots \cup c_{n-1} \times \Omega) \cup (\Omega \times d_n)) \cap \cdots \cap \\
& ((\emptyset \times \Omega) \cup (\Omega \times d_1 \cup \cdots \cup d_n)) \\
\overset{Hilf_{ss.\ B.1}}{\Leftrightarrow} \quad & a \times b \subseteq ((\cup_{i \in I_1} c_i \times \Omega) \cup (\Omega \times \cup_{i \in \overline{I}_1} d_i)) \wedge \\
& a \times b \subseteq ((\cup_{i \in I_2} c_i \times \Omega) \cup (\Omega \times \cup_{i \in \overline{I}_2} d_i)) \wedge \cdots \wedge \\
& a \times b \subseteq ((\cup_{i \in I_{2^n}} c_i \times \Omega) \cup (\Omega \times \cup_{i \in \overline{I}_{2^n}} d_i))
\end{aligned}$$

□

Anhang C

Formalisierung DTD

In Abschnitt 4.3 wird eine Formalisierung für Sprachbeschreibungen, die als XML-Schema vorliegen, angegeben. In diesem Anhang wird eine analoge Formalisierung einer DTD als Heckengrammatik festgelegt. Sie beginnt mit den Regeln für die Ausdrucksrelation.

Definition C.1 (Formalisierung mittels Ausdrucksrelation)
Die *Ausdrucksrelation* $\hookrightarrow$ für DTDs ist durch folgende Regeln definiert:

$$\frac{}{\mathsf{\#PCDATA} \hookrightarrow string} \quad \text{(STR1)}$$

$$\frac{}{\mathsf{CDATA} \hookrightarrow string} \quad \text{(STR2)}$$

$$\frac{}{n \hookrightarrow n} \quad \text{(IDENT)}$$

$$\frac{}{\mathsf{EMPTY} \hookrightarrow \epsilon} \quad \text{(EMPTY)}$$

$$\frac{\begin{array}{c} cm_1 \hookrightarrow r_1, \\ \vdots \\ cm_k \hookrightarrow r_k \end{array}}{cm_1, \ldots, cm_k \hookrightarrow (r_1; \ldots; (r_{k-1}; r_k) \ldots)} \quad \text{(SEQ)}$$

$$\frac{\begin{array}{c} cm_1 \hookrightarrow r_1, \\ \vdots \\ cm_k \hookrightarrow r_k \end{array}}{cm_1 | \ldots | cm_k \hookrightarrow (r_1 | \ldots | (r_{k-1} | r_k) \ldots)} \quad \text{(CHOICE)}$$

$$\frac{cm \hookrightarrow r}{(cm)* \hookrightarrow (r)*} \quad \text{(KLEENE1)}$$

$$\frac{cm \hookrightarrow r}{(cm)+ \hookrightarrow r;(r)*} \quad \text{(KLEENE2)}$$

$$\frac{cm \hookrightarrow r}{(cm)? \hookrightarrow r|\epsilon} \quad \text{(OPT)}$$

$$\frac{id \hookrightarrow e, am \hookrightarrow r}{id\ am\ \texttt{\#REQUIRED}\ d \hookrightarrow \texttt{@}e[r]} \quad \text{(REQ)}$$

$$\frac{id \hookrightarrow e, am \hookrightarrow r}{id\ am\ \texttt{\#IMPLIED}\ d \hookrightarrow \texttt{@}e[r]|\epsilon} \quad \text{(IMP)}$$

$$\frac{id \hookrightarrow e, am \hookrightarrow r}{id\ am\ \texttt{\#FIXED}\ d \hookrightarrow \epsilon} \quad \text{(FIX)}$$

$$\frac{\begin{array}{c} at_1 \hookrightarrow r_1, \\ \vdots \\ at_k \hookrightarrow r_k \end{array}}{at_1 \ldots at_k \hookrightarrow r_1 \& \ldots \& r_k} \quad \text{(ATTR)}$$

mit $k \in \mathcal{N}$, $n \in N$, $\texttt{@}e \in E$ und $r, r_1, \ldots, r_k \in$ ***Reg***. □

Hinzu kommen die Regeln der Produktionenrelation.

Definition C.2 (Formalisierung mittels Produktionenrelation)
Die *Produktionenrelation* $\mapsto$ einer DTD ist durch folgende Regeln definiert:

$$\frac{id \hookrightarrow n, cm \hookrightarrow r}{\texttt{<!ENTITY \%}\ id\ \texttt{"}cm\texttt{" >} \mapsto n \to r} \quad \text{(ENTITY)}$$

$$\frac{id \hookrightarrow e, am \hookrightarrow r_{am}, cm \hookrightarrow r_{cm}}{\begin{array}{l} \texttt{<!ELEMENT}\ id\ cm\ \texttt{>} \\ \texttt{<!ATTRLIST}\ id\ am\ \texttt{>} \end{array} \mapsto e \to e[r_{am}; r_{cm}]} \quad \text{(ELEM)}$$

$$\frac{\begin{array}{c} id \hookrightarrow n, \\ de_1 \mapsto n_1 \to r_1 \text{ mit } T_1, N_1 \text{ und } P_1, \\ \vdots \\ de_k \mapsto n_k \to r_k \text{ mit } T_k, N_k \text{ und } P_k \end{array}}{\texttt{<!DOCTYPE}\ id\ de_1 \ldots de_k\ \texttt{>} \mapsto G = (\cup T_i, \cup N_i, n, \cup P_i) \text{ mit } 1 \leq i \leq k} \quad \text{(DTD)}$$

mit $k \in \mathcal{N}$, $e, n, n_1, \ldots, n_k \in N$, $e \in E$ und $r, r_{cm}, r_{am}, r_1, \ldots, r_k \in$ ***Reg***. □

Anhang D

Implementierung der XPath-Achsen

Dieser Anhang zeigt die Implementierung der restlichen Achsen aus Abschnitt 5.3, in der schon die Methode für die Kind-Achse gezeigt wurde. Die von XPath geforderte Ordnung der Knoten ist nicht implementiert, auch können Knoten in der Ergebnisliste mehrfach auftreten.

Die Nachfahr-Achse wird durch die Methode `getDescendantNodes` realisiert.

```
1   public XobeNodeList getDescendantNodes(String test) {
2     XobeNodeList children, result;
3
4     // alle Kinder, Kindeskinder usw. des Kontextknotens
5     result = new XobeNodeListImpl();
6     children = getChildNodes(test);
7     if (children.getLength() == 0)
8       return result;
9     else {
10      result.addAll(children.getDescendantNodes(test));
11      result.addAll(children);
12    } // else
13    return result;
14  } // getDescendantNodes
```

Die Eltern-Achse wird durch die zwei Methoden `getParentNodes` und `getParentNode` implementiert.

```
1   public XobeNodeList getParentNodes(String test) {
2     int i;
3     XobeNodeList result;
4
5     // Elter des Kontext-Knotens
6     result = new XobeNodeListImpl();
```

```
7       for (i = 0; i < getLength(); i = i + 1)
8         result.addAll(getParentNode(test,item(i)));
9       return result;
10    } // getParentNodes

11    private static NodeList getParentNode(String test, Node
          node) {
12      Node parent;
13      XobeNodeList result;
14
15      result = new XobeNodeListImpl();
16      parent = node.getParentNode();
17      if (parent != null && nodeTest(test,parent))
18          result.add(parent);
19      return result;
20    } // getParentNode
```

Die Vorfahr-Achse wird durch die Methode `getAncestorNodes` realisiert.

```
1     public XobeNodeList getAncestorNodes(String test) {
2       XobeNodeList parents, result;
3
4       // alle Eltern, Grossseltern usw.
5       result = new XobeNodeListImpl();
6       parents = getParentNodes(test);
7       if (parents.getLength() == 0)
8         return result;
9       else {
10        result.addAll(parents.getAncestorNodes(test));
11        result.addAll(parents);
12      } // else
13      return result;
14    } // getAncestorNodes
```

Die Nachfolgende-Geschwister-Achse wird durch die zwei Methoden `getFollowingSiblingNodes` implementiert.

```
1     public XobeNodeList getFollowingSiblingNodes(String
          test) {
2       int i;
3       XobeNodeList result;
4
5       // alle nachfolgenden Geschwister
6       result = new XobeNodeListImpl();
```

```
7       for (i = 0; i < getLength(); i = i + 1)
8         result.addAll(getFollowingSiblingNodes(test,item(i)
              ));
9       return result;
10    } // getFollowingSiblingNodes

11    public XobeNodeList getFollowingSiblingNodes(String
          test, Node node) {
12      Node sibling;
13      XobeNodeList result;
14
15      result = new XobeNodeListImpl();
16      sibling = node.getNextSibling();
17      if (sibling == null)
18        return result;
19      else {
20        result.addAll(getFollowingSiblingNodes(test,sibling
              ));
21        if (nodeTest(test,sibling))
22          result.add(sibling);
23      } // else
24      return result;
25    } // getFollowingSiblingNodes
```

Die Vorherige-Geschwister-Achse wird durch die zwei Methoden `getPrecedingSiblingNodes` realisiert.

```
1     public XobeNodeList getPrecedingSiblingNodes(String
          test) {
2       int i;
3       XobeNodeList result;
4
5       // alle vorherigen Geschwister
6       result = new XobeNodeListImpl();
7       for (i = 0; i < getLength(); i = i + 1)
8         result.addAll(getPrecedingSiblingNodes(test,item(i)
              ));
9       return result;
10    } // getPrecedingSiblingNodes

11    public XobeNodeList getPrecedingSiblingNodes(String
          test, Node node) {
12      Node sibling;
13      XobeNodeList result;
```

```
14
15      result = new XobeNodeListImpl();
16      sibling = node.getPreviousSibling();
17      if (sibling == null)
18        return result;
19      else {
20        result.addAll(getPrecedingSiblingNodes(test,sibling
            ));
21        if (nodeTest(test,sibling))
22          result.add(sibling);
23      } // else
24      return result;
25    } // getPrecedingSiblingNodes
```

Die Nachfolger-Achse wird durch die Methode `getFollowingNodes` implementiert.

```
1     public XobeNodeList getFollowingNodes(String test) {
2       // alle FollowingSiblings der Ancestors und deren
          Descendants
3       return getAncestorNodes("*").getFollowingSiblingNodes
          ("*").
4         getDescendantOrSelfNodes(test);
5     }  // getFollowingNodes
```

Die Vorgänger-Achse wird durch die Methode `getPrecedingNodes` realisiert.

```
1     public XobeNodeList getPrecedingNodes(String test) {
2       // alle PrecedingSiblings der Ancestors und deren
          Descendants
3       return getAncestorNodes("*").getPrecedingSiblingNodes
          ("*").getDescendantOrSelfNodes(test);
4     } // getPrecedingNodes
```

Die Attribut-Achse wird durch die Methode `getAttributeNodes` implementiert.

```
1     public XobeNodeList getAttributeNodes(String test) {
2       int i, j;
3       NamedNodeMap attributes;
4       XobeNodeList result;
5
6       // alle Attribute
7       result = new XobeNodeListImpl();
8       for (i = 0; i < getLength(); i = i + 1) {
9         attributes = item(i).getAttributes();
```

```
10        for (j = 0; j < attributes.getLength(); j = j + 1)
              {
11          if (nodeTest(test,attributes.item(j)))
12            result.add(attributes.item(j));
13        } // for
14      } // for
15      return result;
16    } // getAttributeNodes
```

Die Selbst-Achse wird durch die Methode `getSelfNodes` realisiert.

```
1     public XobeNodeList getSelfNodes(String test) {
2       int i;
3       XobeNodeList result;
4
5       result = new XobeNodeListImpl();
6       for (i = 0; i < getLength(); i = i + 1) {
7         if (nodeTest(test,item(i)))
8           result.add(item(i));
9       } // for
10      return result;
11    } // getSelfNodes
```

Die Vorfahr-oder-Selbst-Achse wird durch die Methode `getAncestorOrSelfNodes` implementiert.

```
1     public XobeNodeList getAncestorOrSelfNodes(String test)
          {
2       XobeNodeList result;
3
4       // alle Eltern, Grossseltern usw. und den
            Kontextknoten
5       result = new XobeNodeListImpl();
6       result.addAll(getAncestorNodes(test));
7       result.addAll(getSelfNodes(test));
8       return result;
9     } // getAncestorOrSelfNodes
```

Die Nachfahr-oder-Selbst-Achse wird durch die Methode `getDescendantOrSelfNodes` realisiert.

```
1     public XobeNodeList getDescendantOrSelfNodes(String
          test) {
2       XobeNodeList result;
3
```

```
4       // alle Kinder, Kindeskinder, usw. des Kontextknotens
            und den Kontextknoten
5       result = new XobeNodeListImpl();
6       result.addAll(getDescendantNodes(test));
7       result.addAll(getSelfNodes(test));
8       return result;
9     } // getDescendantOrSelfNodes
```

Literaturverzeichnis

[ABS00] ABITEBOUL, SERGE, PETER BUNEMAN und DAN SUCIU: *Data on the Web, From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, San Francisco, California, 2000.

[AG98] ARNOLD, KEN und JAMES GOSLING: *The Java Programming Language*. The Java Series. Addison Wesley Longman, Inc., 2. Auflage, 1998.

[Ala97] ALAGIĆ, SUAD: *The ODMG Object Model: Does it Make Sense?* In: BERMAN, A. MICHAEL (Herausgeber): *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Application (OOPSLA '97)*, Band 32(10) der Reihe *SIGPLAN Notices*, Seiten 253–284. ACM Press, October 1997.

[Ala99] ALAGIĆ, SUAD: *O_2 and the ODMG Standard: Do They Match?* Theory and Practice of Object Systems, 5(4):239–247, November 1999.

[AM91] AIKEN, ALEXANDER und BRIAN R. MURPHY: *Implementing Regular Tree Expressions*. In: HUGHES, JOHN (Herausgeber): *Proceedings of Functional Programming and Computer Architecture*, Band 523 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 427–447, Berlin Heidelberg New York, 1991. Springer-Verlag.

[Ant94] ANTIMIROV, VALENTIN: *Rewriting Regular Inequalities*. In: REICHEL (Herausgeber): *Fundamentals of Computation Theory*, Band 965 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 116–125, Berlin Heidelberg New York, 1994. Springer-Verlag.

[Apa00] APACHE SOFTWARE FOUNDATION, THE: *Apache HTTP Server Version 1.3, Apache API notes*. `http://httpd.apache.org/docs/misc/API.html`, 2000.

[Apa01] APACHE XML PROJECT, THE: *Xerces Java Parser*. `http://xml.apache.org/xerces-j/index.html`, 15. November 2001. Version 1.4.4.

[Apa03] APACHE SOFTWARE FOUNDATION, THE: *PHP*. `http://www.php.net/`, 2003.

[ASU86] AHO, A.V., R. SETHI und J.D. ULLMAN: *Compilers – Principles, Techniques and Tools*. Addison-Wesley Publishing Company, Inc., 1986.

[BKMW01] BRÜGGEMANN-KLEIN, ANNE, MAKOTO MURATA und DERICK WOOD: *Regular Tree and Regular Hedge Languages over Unranked Alphabets: Version 1*. Technischer Bericht HKUST-TCSC-2001-05, Hong Kong University of Science & Technology, April 3 2001. Theoretical Computer Science Center.

[BKW98] BRÜGGEMANN-KLEIN, ANNE und DERICK WOOD: *One-unambiguous regular languages*. Information and Computation, Academic Press, 140(2):229–253, 1998.

[BL00] BEHRENS, RALF und VOLKER LINNEMANN: *XML-basierte Informationsmodellierung am Beispiel eines Medienarchivs für die Lehre*. Technischer Bericht A-00-20, Schriftenreihe der Institute für Informatik/Mathematik, Medizinische Universität zu Lübeck, Dezember 2000. available at `http://www.ifis.mu-luebeck.de/public`, (in German).

[BLMM94] BERNERS-LEE, T., L. MASINTER und M. MCCAHILL: *Uniform Resource Locators (URL)*. Request for Comments: 1738, `http://www.w3.org/Addressing/rfc1738.txt`, December 1994. Network Working Group.

[BMS01] BRABRAND, CLAUS, ANDERS MOLLER und MICHAEL I. SCHWARTZBACH: *Static Validation of Dynamically Generated HTML*. In: *Proceedings of Workshop on Program Analysis for Software Tools and Engineering (PASTE 2001), June 18-19, Snowbird, Utah, USA*, Seiten 38–45. ACM, 2001.

[Bor01] BORLAND: *XML Application Developer's Guide, JBuilder*. Borland Software Corporation, Scotts Valley, CA, 1997,2001. Version 5.

[Bou02] BOURRET, RONALD: *XML Data Binding Resources*. web document, `http://www.rpbourret.com/xml/XMLDataBinding.htm`, 28. July 2002.

[Bra98] BRADLEY, NEIL: *The XML Companion*. Addison Wesley Longman Limited, Edinburgh Gate, Harlow, Essex CM20 2JE, United Kingdom, 1998.

[BRJ99] BOOCH, GRADY, JAMES RUMBAUGH und IVAR JACOBSON: *The Unified Modeling Language User Guide*. Object Technology Series. Addison Wesley Longman, Inc., 1999.

[Bro96] BROWN, MARK R.: *FastCGI Specification*. Document Version: 1.0, `http://www.fastcgi.com/devkit/doc/fcgi-spec.html`, 29. April 1996. Open Market, Inc.

[BS86] BERRY, GERARD und RAVI SETHI: *From regular expression to deterministic automata*. Theoretical Computer Science, Elsevier Science, 48(1):117–126, 1986.

[Car97] CARDELLI, LUCA: *Type Systems*. In: TUCKER, ALLEN B. (Herausgeber): *The Computer Science and Engineering Handbook*, Kapitel 103, Seiten 2208–2236. CRC Press, Boca Raton, FL, 1997.

[CAR98] COAR, KEN A. L., THE APACHE GROUP und D. R. T. ROBINSON: *The WWW Common Gateway Interface – Version 1.1*. Internet-Draft, `http://CGI-Spec.Golux.Com/draft-coar-cgi-v11-00.html`, 28. May 1998. Internet Engineering Task Force (IETF).

[CDG+97] COMON, HUBERT, MAX DAUCHET, RÉMI GILLERON, FLORENT JACQUEMARD, DENIS LUGIEZ, SOPHIE TISON und MARC TOMMASI: *Tree Automata Techniques and Applications*. `http://www.grappa.univ-lille3.fr/tata/`, 1997.

[CMS02] CHRISTENSEN, ASKE SIMON, ANDERS MOLLER und MICHALE I. SCHWARTZBACH: *Static analysis for dynamic XML*. In: *Informal Proceedings of the Workshop on Programming Language Technologies for XML (PLAN-X), October 3-8, PLI 2002, Pittsburgh, USA*, Seiten 32–43, 2002.

[Cow01] COWARD, DANNY: *Java Servlet Specification Version 2.3*. `http://www.jcp.org/aboutJava/communityprocess/final/jsr053/`, 17. September 2001. Sun Microsystems, Inc.

[Die00] DIESTEL, REINHARD: *Graphentheorie*. Springer-Verlag, Berlin Heidelberg New York, 2000.

[ECM99] ECMA STANDARDIZING INFORMATION AND COMMUNICATION SYSTEMS: *ECMAScript Language Specification*. Standard ECMA-262 `http://www.ecma-international.org/publications/files/ecma-st/Ecma-262.pdf`, December 1999. 3. Edition.

[EHF01] ELLIS, JOHN, LINDA HO und MAYDENE FISHER: *JDBC 3.0 Specification*. `http://java.sun.com/products/jdbc/download.html`, October 2001. Sun Microsystems, Inc.

[Exo02] EXOLAB GROUP: *Castor*. ExoLab Group, `http://castor.exolab.org/`, 2002.

[FGK02] FLORESCU, DANIELA, ANDREAS GRÜNHAGEN und DONALD KOSSMANN: *XL: An XML Programming Language for Web Service Specification and Composition*. In: *Proceedings of International World Wide Web Conference (WWW 2002), May 7-11, Honolulu, Hawaii, USA*, Seiten 65–76. ACM, 2002.

[FK00] FIELDS, DUANE K. und MARK A. KOLB: *Web Development with Java Server Pages, A practical guide for designing and building dynamic web services*. Manning Publications Co., 32 Lafayette Place, Greenwich, CT 06830, 2000.

[Fly98] FLYNN, PETER: *Understanding SGML and XML Tools, Practical programs for handling structured text*. Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusets 02061, USA, 1998.

[Fre67] FREGE, GOTTLOB: *Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought (1879)*. In: VAN HEIJENOORT, JAN (Herausgeber): *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*, Seiten 1–82, Cambridge, Massachusetts, 1967. Harvard University Press.

[Fre01] FREE SOFTWARE FOUNDATION: *The GNU JAXP Project*. `http://www.gnu.org/software/classpathx/jaxp/jaxp.html`, 2001.

[Gai95] GAITHER, M.: *Foundations of WWW-Programming with HTML and CGI*. IDG-Books Worldwide Inc., Foster City, California, USA, 1995.

[GHJV95] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDES: *Design Patterns*. Professional Computing Series. Addison-Wesley Publishing Company, Inc., 1995.

[GJS96] GOSLING, JAMES, BILL JOY und GUY STEELE: *The Java Language Specification*. The Java series. Addison Wesley Longman, Inc., 2550 Gracia Avenue, Mountain View, California 94043-1100 USA, 1996.

[GJSB00] GOSLING, JAMES, BILL JOY, GUY STEELE und GILAD BRACHA: *The Java Language Specification*. The Java series. Addison Wesley Longman, Inc., 901 San Antonio Road, Mountain View, California 94303 USA, 2. Auflage, 2000.

[GKP02] GOTTLOB, GEORG, CHRISTOPH KOCH und REINHARD PICHLER: *Efficient Algorithms for Processing XPath Queries*. In: *Proceedings of the 28th VLDB Conference, Hong Kong, China*, Seiten 95–106, 2002.

[Gol90] GOLDFARB, CHARLES F.: *The SGML Handbook*. Oxford University Press, Walton Street, Oxford OX2 6OP, 1990.

[GP00] GOLDFARB, CHARLES F. und PAUL PRESCOD: *The XML Handbook*. Prentice Hall PTR, Upper Saddle River, NJ 07458, 2. Auflage, 2000.

[Gun96] GUNDAVARAM, SHISHIR: *CGI-Programming on the World Wide Web*. O'Reilly & Associates, Inc., 1996.

[Har02] HAROLD, ELLIOTTE RUSTY: *Processing XML with Java*. Addison-Wesley Publishing Company, Inc., 2002. `http://cafeconleche.org/books/xmljava/`.

[HC98] HUNTER, JASON und WILLIAM CRAWFORD: *Java Servlet Programmierung*. O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472, 1. Auflage, October 1998.

[Hos00] HOSOYA, HARUO: *Regular Expression Types for XML*. Doktorarbeit, University of Tokyo, 2000.

[HU79] HOPCROFT, JOHN E. und JEFFREY D. ULLMAN: *Introduction to automata, languages and computations*. Addison-Wesley Publishing Company, Inc., Reading, MA, 1979.

[Hug99] HUGE, ANNE KATHRIN: *Formalisierung Objektorientierter Datenbanken auf der Grundlage von ODMG*. Doktorarbeit, Universität Bremen, Institute of Safe Systeme, Postfach 330440, 28334 Bremen, Juli 1999. Aachen, Shaker-Verlag, 2000.

[HVP00] HOSOYA, HARUO, JÉRÔME VOUILLON und BENJAMIN C. PIERCE: *Regular Expression Types for XML*. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada*, Band 35(9) der Reihe *SIGPLAN Notices*, Seiten 11–22. ACM, September 18-21 2000.

[IBM03] IBM ALPHAWORKS: *XML Parser for Java*. `http://alphaworks.ibm.com/tech/xml4j`, 2003.

[Inf97a] INFORMIX PRESS: *Informix Web DataBlade Module Users's Guide*. Informix Software, Inc., 4100 Bohannon Drive, Menlo Park, CA 94025-1032, May 1997. Version 3.3.

[Inf97b] INFORMIX SOFTWARE, INC., 4100 Bohannon Drive, Menlo Park, CA 94025: *Getting Started with INFORMIX – Universal Server*, March 1997. Version 9.1.

[Int94] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *Open System Interconnection Model*. ISO: 1738, `http://www.iso.org/`, 1994.

[Int03] INTERNET SOFTWARE CONSORTIUM: *Internet Domain Survey*. `http://www.isc.org/ds/WWW-200301/index.html`, January 2003.

[JDO] JDOM PROJECT: *JDOM FAQ*. `http://www.jdom.org/docs/faq.html`.

[KL02] KEMPA, MARTIN und VOLKER LINNEMANN: *VDOM and P-XML – Towards A Valid Programming Of XML-based Applications*. Information and Software Technology, Elsevier Science B. V., Seiten 229–236, 2002. Special Issue on Objects, XML and Databases.

[Kra01] KRAMER, JENS-CHRISTIAN: *Konzeption und Implementierung einer WAP-basierten Benutzerschnittstelle für das Medienarchiv MONTANA*. Schriftenreihe der Institute für Informatik/Mathematik B-01-03, Medizinische Universität zu Lübeck, Januar 2001.

[Kra02] KRAMER, JENS-CHRISTIAN: *Erzeugung garantiert gültiger Server-Seiten für Dokumente der Extensible Markup Language XML*. Diplomarbeit, Institut für Informationssysteme, Universität zu Lübeck, 2002. (in German).

[Kro95] KROL, ED: *Die Welt des Internet*. Handbuch & Übersicht. O'Reilly / International Thomson Verlag GmbH & Co KG, Königswinter Straße 418, 53 227 Bonn, 1. Auflage, 1995. (in German).

[KT01] KRUTWIG, MICHAEL und ROBERT TOLKSDORF: *WML und WMLScript, Informationen aufbereiten und präsentieren für WAP-Dienste*. dpunkt-Verlag GmbH, 2001.

[LEW96] LOECKX, JACQUES, HANS-DIETER EHRICH und MARKUS WOLF: *Specification of Abstract Data Types*. John Wiley & Sons Ltd., Chichester, England, 1996.

[Lin79] LINNEMANN, VOLKER: *Sprachelemente zur Generierung und Umformung syntaktischer Strukturen auf der Basis von ALGOL-68 und deren theoretische Untersuchung*. Doktorarbeit, Universität Carolo-Wilhelmina zu Braunschweig, Deutschland, 1979. (in German).

[Lin81] LINNEMANN, VOLKER: *Context-free Grammars and Derivation Trees in Algol 68*. In: *Proceedings International Conference on ALGOL68, Mathematical Centre Tracts 134*, Seiten 167–182. Math. Centrum Amsterdam, 1981.

[LK02] LINNEMANN, VOLKER und MARTIN KEMPA: *Sprachen und Werkzeuge zur Generierung von HTML- und XML-Dokumenten*. Informatik Spektrum, Springer-Verlag Heidelberg, 25(5):349–358, 2002. (in German).

[LV96] LAUSEN, GEORG und GOTTFRIED VOSSEN: *Objekt-orientierte Datenbanken: Modelle ud Sprachen*. R. Oldenbourg Verlag GmbH, München, 1996. (in German).

[LZ74] LISKOV, BARBARA und STEPHEN ZILLES: *Programming with abstract data types*. ACM SIGPLAN Notices, ACM Press, 9(4):50–59, April 1974.

[Mat01] MATSUMOTO, YUKIHIRO: *Programming Ruby, The Pragmatic Programmer's Guide*. Addison Wesley Longman, Inc., 2001. `http://www.rubycentral.com/book/`.

[Mic01] MICROSOFT CORPORATION: *.NET Framework Developer's Guide*. web document, `http://msdn.microsoft.com/library/default.asp`, 2001.

[Mit96] MITCHELL, JOHN C.: *Foundations of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1996.

[Mur01] MURATA, MAKOTO: *Extended Path Expressions for XML*. In: *Proceedings of the Symposium on Principles of Database Systems (PODS), May 21-23, Santa Barbara, California, USA*, Seiten 126–137. ACM Press, 2001.

[Net97a] NETSCAPE COMMUNICATIONS CORPORATION: *JavaScript 1.1 Language Specification*. `http://www.netscape.com/eng/javascript/index.html`, 1997.

[Net97b] NETSCAPE COMMUNICATIONS CORPORATION: *NSAPI Programmer's Guide*. `http://developer.netscape.com/docs/manuals/enterprise/nsapi/`, 1997.

[Net03] NETCRAFT: *Netcraft Web Server Survey*. `http://www.netcraft.com/Survey`, May 2003.

[Neu99] NEUMANN, ADREAS: *Parsing and Querying XML Documents in SML*. Doktorarbeit, University of Trier, Trier, 1999.

[NNH99] NIELSON, FLEMMING, HANNE RIIS NIELSON und CHRIS L. HANKIN: *Principles of Program Analysis*. Springer-Verlag, Berlin Heidelberg New York, 1999.

[Obj02] OBJECT MANAGMENT GROUP: *Common Object Request Broker, CORBA, Architecture: Core Specification*. `http://www.omg.org/technology/documents/corba_spec_catalog.htm`, December 2002. Version 3.

[Ora01] ORACLE CORPORATION: *Oracle9i, Application Developer's Guide – XML, Release 1 (9.0.1)*. Redwood City, CA 94065, USA, June 2001. Shelley Higgins, Part Number A88894-01.

[Ora02] ORACLE CORPORATION: *XML Developer's Kit for Java*. `http://otn.oracle.com/tech/xml/xdkhome.html`, 2002.

[Per90] PERRIN, D.: *Finite Automata*. In: LEEUWEN, J. VAN, A. MEYER, M. NIVAT, M. PATERSON und D. PERRIN (Herausgeber): *Handbook of Theoretical Computer Science*, Band B, Seiten 1–57. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990. Chapter 1.

[PL01] PELEGRI-LLOPART, EDUARDO: *JavaServer Pages Specification Version 1.2*. `http://www.jcp.org/aboutJava/communityprocess/final/jsr053/`, 17. September 2001. Sun Microsystems, Inc.

[PLC99] PELEGRÍ-LLOPART, EDUARDO und LARRY CABLE: *Java Server Pages Specification, Version 1.1*. Java Software, Sun Microsystems, `http://java.sun.com/products/jsp/download.html`, 30. November 1999.

[Pra65] PRAWITZ, DAG: *Natural Deduction: A Proof-Theoretical Study*. Almquist and Wiksell, Stockholm, 1965.

[RBP+91] RUMBAUGH, JAMES, MICHAEL BLAHA, WILLIAM PREMERLANI, FREDERICK EDDY und WILLIAM LORENSEN: *Object-oriented Modeling and Design*. Prentice-Hall, Inc., 1991.

[RJB99] RUMBAUGH, JAMES, IVAR JACOBSON und GRADY BOOCH: *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison Wesley Longman, Inc., 1999.

[RLHJ97] RAGGETT, DAVE, ARNAUD LE HORS und IAN JACOBS: *HTML 4.0 Specification*. Recommendation, `http://www.w3.org/TR/REC-html40-971218/`, 18. December 1997. W3Consortium.

[RS97] ROZENBERG, GRZEGORZ und ARTO SALOMAA (Herausgeber): *Handbook of Formal Languages*, Band 3. Springer-Verlag, Berlin Heidelberg New York, 1997.

[Sal73] SALOMAA, A.: *Formal Languages*. Academic Press, 1973.

[SBK+99] SHAW, PHIL, BRIAN BECKER, JOHANNES KLEIN, MARK HAPNER, GRAY CLOSSMAN und RICHARD PLEDEREDER: *SQLJ: Java and Relational Databases, Tutorial*. `http://www.sqlj.org/`, 11. September 1999.

[Sei90] SEIDL, HELMUT: *Deciding equivalence of finite tree automata*. SIAM Journal of Computing, 19(3):424–437, June 1990.

[Spi02] SPIEGLER, TORBEN: *Entwicklung und Implementierung eines Datenbankschemas zur Verarbeitung von Übungs- und Vorlesungsdaten*. Studienarbeit am Institut für Informationssysteme der Universität zu Lübeck, Oktober 2002.

[Sun01a] SUN MICROSYSTEMS, INC.: *Java 2 Platform, Standard Edition, v 1.3.1, API Specification*. `http://java.sun.com/j2se/1.3/docs/api/index.html`, December 2001.

[Sun01b] SUN MICROSYSTEMS, INC: *Java API for XML Processing (JAXP)*. `http://java.sun.com/xml/jaxp/`, 2001.

[Sun03] SUN MICROSYSTEMS, INC: *The Java Architecture for XML Binding (JAXB)*. Specification, Version 1.0, `http://www.sun.com`, 8. January 2003. Editors: Joseph Fialli, Sekhar Vajjhala.

[Tak75] TAKEUTI, GAISI: *Proof Theory*, Band 81 der Reihe *Studies in Logic and the Foundations of Mathematics*. North-Holland Pub. Co., Amsterdam, 1975.

[Tan96] TANENBAUM, ANDREW S.: *Computer Networks*. Prentice-Hall International, Inc., 1996.

[Tol97a] TOLKSDORF, ROBERT: *Die Sprache des Web: HTML 4*. dpunkt, Verlag für digitale Technologie, Ringstraße 19, D – 69115 Heidelberg, 3. Auflage, 1997. (in German).

[Tol97b] TOLKSDORF, ROBERT: *Internet, Aufbau und Dienste*. Thomson's Aktuelle Tutorien. International Thomson Publishing GmbH, 1. Auflage, 1997. (in German).

[Tur96] TURAU, VOLKER: *Algorithmische Graphentheorie*. Addison-Wesley Publishing Company, Inc., 1996.

[TWP00] TAO, KEVIN, WANJUN WANG und DR. JENS PALSBERG: *Java Tree Builder JTB*. `http://www.cs.purdue.edu/jtb/`, 15. May 2000. Version 1.2.2.

[vdV02] VLIST, ERIC VAN DER: *XML Schema*. O'Reilly & Associates, Inc., 2002.

[W3C96] W3CONSORTIUM: *Cascading Style Sheets, level 1*. Recommendation, http://www.w3.org/TR/REC-CSS1, 17. December 1996.

[W3C98a] W3CONSORTIUM: *Cascading Style Sheets, level 2, CSS2 Specification*. Recommendation, http://www.w3.org/TR/REC-CSS2, 12. May 1998.

[W3C98b] W3CONSORTIUM: *Document Object Model (DOM) Level 1 Specification, Version 1.0*. Recommendation, http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/, 1. October 1998.

[W3C98c] W3CONSORTIUM: *Extensible Markup Language (XML) 1.0*. Recommendation, http://www.w3.org/TR/1998/REC-xml-19980210/, 10. February 1998.

[W3C99a] W3CONSORTIUM: *XML Path Language (XPath), Version 1.0*. Recommendation, http://www.w3.org/TR/xpath, 16. November 1999.

[W3C99b] W3CONSORTIUM: *XSL Transformations (XSLT) Version 1.0*. Recommendation, http://www.w3.org/TR/xslt, 16. November 1999.

[W3C00a] W3CONSORTIUM: *Document Object Model (DOM) Level 2 Core Specification, Version 1.0*. Recommendation, http://www.w3.org/TR/DOM-Level-2-Core/, 13. November 2000.

[W3C00b] W3CONSORTIUM: *XHTML 1.0: The Extensible HyperText Markup Language, A Reformulation of HTML 4.0 in XML 1.0*. Recommendation, http://www.w3.org/TR/2000/REC-xhtml1-20000126/, 26. January 2000.

[W3C01a] W3CONSORTIUM: *XML Schema: Formal Description*. Working Draft, http://www.w3.org/TR/2001/WD-xmlschema-formal-20010925/, 25. September 2001.

[W3C01b] W3CONSORTIUM: *XML Schema Part 0: Primer*. Recommendation, http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/, 2. May 2001.

[W3C01c] W3CONSORTIUM: *XML Schema Part 1: Structures*. Recommendation, http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/, 2. May 2001.

[W3C01d] W3CONSORTIUM: *XML Schema Part 2: Datatypes*. Recommendation, http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/, 2. May 2001.

[W3C02a] W3CONSORTIUM: *XML Path Language (XPath) 2.0*. Working Draft, http://www.w3.org/TR/2002/WD-xpath20-20021115/, 15. November 2002.

[W3C02b] W3CONSORTIUM: *XML Pointer Language (XPointer)*. Working Draft, http://www.w3.org/TR/xptr/, 16. August 2002.

[W3C02c] W3CONSORTIUM: *XQuery 1.0: An XML Query Language*. Working Draft, `http://www.w3.org/TR/2002/WD-xquery-20021115/`, 15. November 2002.

[Web02] WEBGAIN: *Java Compiler Compiler (JavaCC) – The Java Parser Generator*. `http://www.webgain.com/products/java_cc/`, 2002. Version 2.1.

[Wil99] WILLIAMSON, A. R.: *Java Servlets by Example*. Manning Publications Co., Greenwich, 1999.

[Wir00] WIRELESS APPLICATION PROTOCOL FORUM: *Wireless Application Protocol, Wireless Makup Language Specification, Version 1.3*. `http://www1.wapforum.org/tech/documents/WAP-191-WML-20000219-a.pdf`, 19. February 2000.

[WPP+83] WIRSING, M., H. PARTSCH, P. PEPPER, W. DOSCH und M. BROY: *On Hierarchies of Abstract Data Types*. Acta Informatica, Springer-Verlag Heidelberg, 20:1–33, 1983.

[WS92] WALL, L. und R. L. SCHWARTZ: *Programming Perl*. O'Reilly & Associates, Inc., Sebastipol, California, 1992.

[Wut] WUTKA CONSULTING, INC.: *DTDParser, A Java DTD Parser*. A Product of Wutka Consulting, `http://www.wutka.com/dtdparser.html`.

Index

Lebenslauf

Persönliche Daten:	Sascha Martin Kempa geboren am 13.9.1972 in Berlin	
Schulausbildung:	1978 – 1981	31. Grundschule in Berlin – Reinickendorf
	1981 – 1984	Grundschule Evangelische Schule Frohnau
	1984 – 1991	Gymnasium Evangelische Schule Frohnau mit Abitur
	1988	4-monatiger Schulbesuch an der St. Augustin School in Oxford, England
Hochschulstudium:	10.1991 – 7.1993	Grundstudium der Informatik mit mit Wahlfach Mathematik an der Technischen Universität Berlin
	7.1993 – 2.1997	Hauptstudium der Informatik an der Technischen Universität Berlin, Schwerpunkte: Programmiersprachen und theoretische Informatik
Praktikum:	7.1994 – 9.1994	Werkstudent bei der Firma Siemens im Bereich öffentliche Kommunikationsnetze
Zivildienst:	3.1997 – 3.1998	Krankenhaus Spandau, Berlin
Forschung und Lehre:	11.1993 – 3.1996	Tutor an der Technischen Universität Berlin im Institut für Quantitative Methoden, Fachgebiet Statistik und Wirtschaftsmathematik, Betreuung der Veranstaltung Mathematik für Wirtschaftswissenschaftler
	ab 4.1998	wissenschaftlicher Mitarbeiter am Institut für Informationssysteme der Universität zu Lübeck